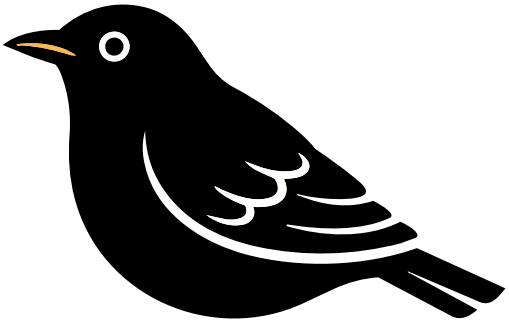
Principles of Dependent Type Theory

Carlo Angiuli
Indiana University
cangiuli@iu.edu

Daniel Gratzer
Aarhus University
gratzer@cs.au.dk

(2026-09-22)



This material will be published by Cambridge University Press as *Principles of Dependent Type Theory* by Carlo Angiuli and Daniel Gratzer. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © Carlo Angiuli and Daniel Gratzer 2024

[*Rake thudding against face*]
Eeeuughhh

Robert Onderdonk Terwilliger Jr., Ph.D.
The Simpsons, Season 5 Episode 2, "Cape Feare."

This material will be published by Cambridge University Press as *Principles of Dependent Type Theory* by Carlo Angiuli and Daniel Gratzer. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © Carlo Angiuli and Daniel Gratzer 2024

Changes

Below we summarize our major updates of this book.

2026-09-22 v0.3

This is the α release with draft versions of all planned content.

- Added a Cool Bird to the cover.
- Drafted Section 4.4 on observational type theory.
- Split a section into Sections 6.5 and 6.6.
- Added Appendix B on generalized algebraic theories.
- Removed appendix with solutions to selected exercises.

2025-07-18 v0.2

- Reformatted to Cambridge University Press specifications.
- Added Preface.
- Added Section 2.5.3 on coproduct types.
- Revised Section 2.7 on propositions in extensional type theory.
- Revised Section 5.1 on propositional univalence.
- Revised Section 5.2 on homotopy type theory.
- Drafted Section 6.7 on canonicity for extensional type theory.

2025-04-30

- Removed a section from Chapter 5 on motivating univalence.
- Redrafted Section 2.7 on propositions in extensional type theory.
- Drafted Sections 6.1 to 6.5 on the semantics of type theory.

2024-08-29

- Begin maintaining a change log.
- Added Section 3.5 on the set model of extensional type theory.

- Added solutions to selected exercises as an appendix.
- Drafted Section 2.7 on propositions in extensional type theory.
- Drafted Chapter 5 on homotopy type theory.

2024-04-14 v0.1

- Added Chapter 1.
- Added Chapter 2 on extensional type theory.
- Added Chapter 3 on metatheory and implementation.
- Added Chapter 4 on intensional type theory.
- Added Appendix A collecting the formal rules of type theory.

This material will be published by Cambridge University Press as *Principles of Dependent Type Theory* by Carlo Angiuli and Daniel Gratzer. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © Carlo Angiuli and Daniel Gratzer 2024

Contents

Changes **iii**

Contents **v**

Preface **vii**

1 Introduction **1**
1.1 Uniform dependency: length-indexed vectors . . . 2
1.2 Non-uniform dependency: computing arities . . . 6
1.3 Proving type equations . . . 11

2 Extensional type theory **17**
2.1 The simply-typed lambda calculus . . . 18
2.2 Towards the syntax of dependent type theory . . . 26
2.3 The calculus of substitutions . . . 29
2.4 Internalizing judgmental structure: **Π**, **Σ**, **Eq**, **Unit** . . . 35
2.5 Inductive types: **Void**, **Bool**, +, **Nat** . . . 47
2.6 Universes: $\mathbf{U}_0, \mathbf{U}_1, \mathbf{U}_2, \ldots$ . . . 61
2.7⋆ Propositions and propositional truncation . . . 72

3 Metatheory and implementation **89**
3.1 A judgmental reconstruction of proof assistants . . . 90
3.2 Metatheory for type-checking . . . 95
3.3⋆ A case study in elaboration: definitions . . . 102
3.4 Models for metatheory . . . 105
3.5⋆ The set model of type theory . . . 110
3.6 Equality in extensional type theory is undecidable . . . 125

4 Intensional type theory **133**
4.1 Programming with propositional equality . . . 134
4.2 Intensional identity types . . . 141
4.3 Limitations of the intensional identity type . . . 147
4.4⋆ Observational type theory (DRAFT) . . . 158

5 Univalent type theories **167**

5.1 Propositional univalence . . . 168
5.2 Homotopy type theory . . . 176
5.3 Cubical type theory (DRAFT) . . . 195
5.4⋆ Computing with coercions and compositions (DRAFT) . . . 213

6 Semantics of type theory (DRAFT) 225
6.1 Categories with families . . . 227
6.2 Pullback squares and **Π**, **Σ**, **Eq**, **Unit** . . . 235
6.3 Orthogonality and **Void**, **Bool**, +, **Nat** . . . 243
6.4 Cwf morphisms and $\mathbf{U}_0, \mathbf{U}_1, \mathbf{U}_2, \ldots$ . . . 262
6.5 Democratic models and local cartesian closure . . . 272
6.6 Strictification constructions . . . 280
6.7 Canonicity via gluing . . . 295

A Martin-Löf type theory 311

B Principles of dependent sort theory 323
B.1 Signatures of generalized algebraic theories . . . 324
B.2 The category of models of a GAT . . . 330
B.3 The initial LF algebra . . . 338

Bibliography 345

This material will be published by Cambridge University Press as *Principles of Dependent Type Theory* by Carlo Angiuli and Daniel Gratzer. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © Carlo Angiuli and Daniel Gratzer 2024

Preface

Dependent type theory (henceforth just *type theory*) often appears arcane to outside observers for a handful of reasons. First, as in the parable of the blind men and the elephant, there are myriad perspectives on type theory. The family of languages in this book, *mutatis mutandis*, can be accurately described as:

- the core language of assertions and proofs in *proof assistants* like Agda [Agda], Rocq (formerly known as Coq) [Rocq], Lean [dMU21], and Nuprl [Con+85];
- a richly-typed *functional programming language*, as in Idris [Bra13] and Pie [FC18], and the aforementioned proof assistants Agda [Stu16] and Lean [Chr23].
- an *axiom system* for reasoning synthetically in a number of mathematical settings, including locally cartesian closed 1-categories [Hof95b], homotopy types [Shu21], and Grothendieck ∞-topoi [Shu19];
- a structural [Tse17], constructive [Mar82] *foundation for mathematics* as an alternative to ZFC set theory [Alt23].

A second difficulty is that it is quite complex to even *define* type theory in a precise fashion, for reasons we shall discuss in Section 2.2, and the relative merits of different styles of definition—and even which ones satisfactorily define any object whatsoever—have been the subject of great debate among experts over the years.

Third, much of the literature on type theory is highly technical—involving either lengthy proofs by induction or advanced mathematical machinery—in order to account for its complex definition and applications.

Finally, and perhaps most confusingly of all, dependent type theory is not a single logic, language, axiom system, or foundation; it is a *family* of systems descended from the 1971 work of philosopher Per Martin-Löf [Mar71],[1] whose notable members include extensional type theory [Mar82], intensional type theory [Mar75], observational type theory [AMS07; PT22], homotopy type theory [UF13; Rij25], and various cubical type theories [CCHM18; Ang+21]. Indeed, every proof assistant and programming language mentioned above is built atop a different core type theory.

In this book, we present a modern research perspective on the design of "full-spectrum" dependent type theories, those descended from Martin-Löf's 1971 theory.

[1] Which of course has its own ancestors, including Russell's *doctrine of types* [Rus03].

Our goal is to pose and begin to answer the following questions: *What makes a good type theory, and why are there so many?* We focus on *notions of equality in Martin-Löf type theory* as a microcosm of this broader question, studying how extensional, intensional, observational, homotopy, and cubical type theories have provided increasingly sophisticated answers to this deceptively simple question.

Although the design of type theory is inextricably linked to its applications (both theoretical and practical), we stress that this book focuses only on the design of type theory, *as an object of study in its own right*; there are many other resources for readers interested in learning how to use type theory as a formal logic or programming language. After studying this book, readers should be prepared to engage with contemporary research on type theory, and to understand the motivations behind various extensions of Martin-Löf's dependent type theory.

This book is in draft form. The authors welcome any feedback, including typos and relevant citations.

How to use this book This book started as shared lecture notes for graduate courses on dependent type theory taught simultaneously by the authors at Indiana University and Aarhus University in Spring 2024. As such, it is designed to be read in a linear fashion, with each chapter and section depending on many of the sections that come before it, with a few exceptions as follows.

Sections marked with ⋆, such as Section 2.7, are considered optional and are not referenced until much later in the book if ever; these sections cover topics that we consider important but nevertheless tangential to the immediate narrative. Smaller tangents are confined to *Remarks* and *Advanced Remarks*, the latter requiring more advanced mathematical prerequisites such as category theory. These often provide useful context or intuition but are again not integral to the main narrative.

A one-semester graduate course should cover all of the non-optional material in Chapters 1 to 4, which discuss extensional type theory, metatheory and implementation, and intensional type theory. If taught at a brisk pace, this should leave a few weeks for additional topics of the lecturer's choice, which can be drawn from the remainder of the book. Chapters 5 and 6 (on homotopy type theory and semantics, respectively) depend on Chapters 1 to 4 but not on each other, and can be tackled in either order. We expect that neither will fit in its entirety into the aforementioned one-semester course, but we felt that the book would be incomplete without both present.

To the independent reader, we likewise strongly recommend reading the non-optional sections of Chapters 1 to 4 in order, and seasoning to taste with some optional sections and Chapters 5 and 6. Even the most targeted of reads should include a skim of Chapter 2, which introduces the main ideas and notations used throughout.

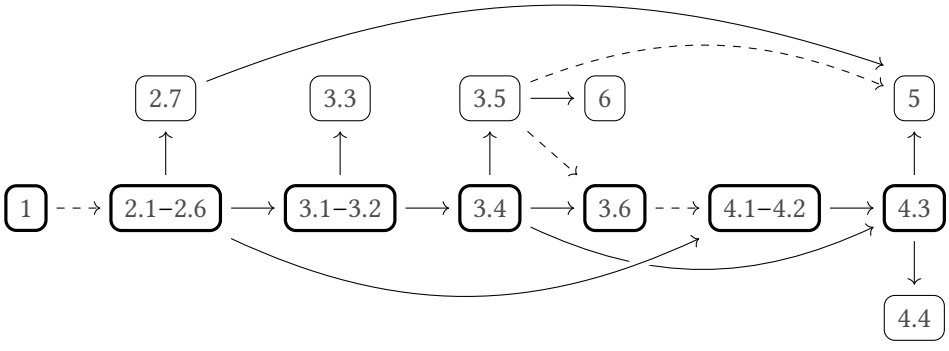


Figure 0.1: Dependency graph of sections.

For a more detailed account of cross-section dependencies, see Figure 0.1. Dashed lines represent dependencies that are more superficial in nature; thick borders indicate what we regard as the core sections of the book. Note that reading the book in textual order satisfies all dependencies, but also that some sections can be safely deferred.

We end every chapter with a discussion of related literature, and encourage readers to follow these pointers to learn about these topics in greater depth. We have also attempted to include many references throughout the main body of the text, and the lengthy bibliography should also be considered a useful resource for further study.

Finally, we have included some exercises throughout the text to reinforce important concepts; for best results, the reader should work through at least some of these.

Who is this book for? First and foremost, this book is intended as a resource for early Ph.D. and advanced master's students in computer science, mathematics, and philosophy who wish to engage with current research in dependent type theory, such as the syntax and semantics of homotopy, modal, and cubical type theories. We hope that it also serves as a comprehensive resource for seasoned researchers in adjacent areas, such as programming language theory or homotopy theory, who want to learn about the technical principles guiding the design of type theories.

We have strived to minimize the book's formal prerequisites besides a working knowledge of basic discrete mathematics and, yes, the dreaded *mathematical maturity*. We do not strictly assume prior familiarity with dependent type theory itself, but the reader should ideally have a passing familiarity with *using* dependent type theory in some proof assistant, as the book is light on background motivation.

In Chapter 1 we briefly motivate dependent types with a series of programming examples in Agda syntax which presuppose some basic familiarity with typed functional

programming. Readers who lack this familiarity but have previously seen dependent types should be able to safely skim Chapter 1 and start reading carefully at Chapter 2.

In Section 2.1 we assume the reader is familiar with judgments and inference rules as a method of specifying simpler formal systems such as predicate logic or the simply-typed lambda calculus. These topics are more than adequately covered by a one-semester course on programming language theory or logic; by the first few chapters of textbooks on programming language theory, such as Pierce [Pie02] or Harper [Har16]; and by the first few chapters of many textbooks about using dependent type theory, such as Rijke [Rij25].

Sections 3.1 to 3.3 discuss the implementation of type theory on computers, but do not intend to assume much if any computer science expertise. Sections 3.2 and 3.6 make essential reference to computability and decidability but require only a very superficial understanding of basic computability theory.

In Chapter 6 we revisit Chapters 2 and 3 using the language of category theory; naturally, this chapter—and only this chapter—requires a working knowledge of basic category theory as covered in a one-semester graduate course or the first four chapters of Riehl [Rie16]. Readers without this prerequisite can simply skip Chapter 6, although we hope that some will use this chapter as an invitation to learn category theory.

TODO below here. Add: pointers to the literature; why we wrote the book (provide a coherent narrative of the first 50 years of type theory); discuss contents in more detail; we have attempted to provide citations but it's hard and (although the field is young enough that we have been fortunate to meet many of these people) we weren't there. We have generally attempted to only include "canonical" stuff but there are a few mentions of active but less-settled things (OTT, cubical type theory). We have tried to be generally standard about terminology and notation unless we felt strongly (examples?). See CUP proposal.

Notes to the expert We briefly remark on some editorial decisions that may surprise experts in type theory. First, we emphasize that this book is about the design of type theory, not how to use it. We therefore provide relatively few examples of working within type theory, focusing instead on type theories *qua* mathematical objects in their own right. This focus sets us apart from most textbooks on the subject, which take a single theory for granted and explore its characteristics as a foundation of mathematics and/or functional programming language.

In light of this focus, experts may be surprised to find that our presentation does not explicitly rely on category theory. This was a difficult decision for the authors, both of whom view type theory from a categorical perspective, but we believe it is simply not feasible to insist that students begin their journey into type theory by

first reading a book on category theory. Early attempts to simultaneously introduce category theory and type theory felt unsatisfactory on both counts.

That said, we do not attempt in any way to hide the presence of categories, functors, and naturality in the foundations of type theory. On the contrary, in Chapter 2 we define various connectives by the functors they (co)represent, phrased in more elementary language. We hope our exposition is accessible to readers encountering type theory for the first time, but also plainly categorical in flavor to those with more mathematical background. Moreover, Chapter 6 revisits many of the topics of Chapter 2 from a purely categorical perspective.

Our perspective on type theory is deeply algebraic: we regard the judgments of type theory as being indexed by well-formed contexts and types, all defined only up to definitional equality. As a result, it is straightforward for us to introduce the notion of a model of type theory in Section 3.4, of which syntax is the initial example.

Finally, we have aimed to confine the non-optional sections of this book to fit within a semester of brisk lectures. For this reason we have elided numerous topics of interest, including a systematic treatment of inductive types, more discussion of elaboration, proofs of normalization, and countless interesting variations of type theory. Particularly painful omissions include a discussion of quotient types in Chapter 2 and an explanation of univalence as a universal property of the universe in Chapter 5.

Acknowledgements

We are grateful to Lars Birkedal for his comments and suggestions on drafts of this book, and to Sam Tobin-Hochstadt for many insightful conversations with the first author over lunch that helped us refine our narrative. We thank the students who participated in *Modern Dependent Types* (CSCI-B619) at Indiana University and *Modern Dependent Type Theory* at Aarhus University in Spring 2024—the courses which prompted us to write this book—and the participants in an abbreviated lecture series given at Oxford University by the second author during the 2024 Michaelmas term. We also thank Evan Cavallo and Ana Bove for locating and scanning a physical copy of Tasistro's licentiate thesis [Tas93]. We are particularly grateful to Thierry Coquand, who informed us of (and scanned for us!) many lesser-known historical references. We are grateful to Silke Sofia Dainese for providing the cover art for the online version of this book.

Many readers have suggested corrections and improvements to earlier drafts of this book. We owe special thanks to Naïm Favier and Hongseok Yang for their extensive feedback. We also thank Madeleine Birchfield, Nathan Corbyn, Arnaud Daby-Seesaram, Leonid Dubinsky, Fred Fu, Rasmus Kirk Jakobsen, Max Jenkins, Artem Iurchenko, Sanad Kadu, Pavel Kovalev, Kwing Hei Li, Amélia Liao, Mathias Adams

Møller, Soonwon Moon, Egor Namakonov, Thomas Porter, Ian Ray, June Rousseau, Zixiu Su, Nicolas Wu, Yiming Xu, and Yafei Yang for pointing out typos and other issues.

add names as people point out typos

Finally, we thank the first author's cat Hannah for diligently supervising many critical video calls during the preparation of this book.

This material will be published by Cambridge University Press as *Principles of Dependent Type Theory* by Carlo Angiuli and Daniel Gratzer. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © Carlo Angiuli and Daniel Gratzer 2024

1 *Introduction*

The other five chapters of this book define, motivate, and analyze a series of increasingly complex dependent type theories from first principles. This chapter has a rather different purpose: to introduce the basic concepts of full-spectrum dependent type theory—type and term dependency, definitional equality, and propositional equality—from the *user's* perspective. After all, it is difficult to understand abstract concepts without some awareness of how they may be applied.

For this task we must fix a perspective on dependent type theory, and for better or for worse we choose to view it in this chapter as a typed functional programming language, using Agda syntax [Agda]. Any choice has the effect of potentially alienating some readers, but we hope that most readers are able to at least partially follow the narrative. We assure you that the remainder of the book is rather different from this chapter, and is quite self-contained albeit perhaps lacking in top-level motivation. In particular, the remainder of the book does not assume familiarity with programming.

In this chapter In Section 1.1 we introduce dependent types through the traditional example of length-indexed vectors. In Section 1.2 we turn our attention to full-spectrum dependency and the role of definitional equality, studying a dependently-typed implementation of sprintf. Finally, in Section 1.3 we discover that typing even simple list-processing functions can require inductive equational reasoning, motivating us to introduce propositional equality and the *propositions as types* correspondence between typed functional programming and formal logic.

Goals of the chapter By the end of this chapter, you will be able to:

- Give examples of full-spectrum dependency.
- Explain the role of definitional equality in type-checking, and how and why it differs from ordinary closed-term evaluation.
- Explain the role of propositional equality in type-checking.

1.1 *Uniform dependency: length-indexed vectors*

What does it mean for a programming language to be typed? Throughout this book, we will regard a language's (static) type system as its *grammar*, not as one of many potential static analyses that might be enabled or disabled.[1] Indeed, just as a parser may reject as nonsense a program whose parentheses are mismatched, or an untyped language's interpreter may reject as nonsense a program containing unbound identifiers, a type-checker may reject as nonsense the program `1 + "hi"` on the grounds that—much like the previous two examples—there is no way to successfully evaluate it.

A type system divides a language's well-parenthesized, well-scoped expressions into a collection of sets. The *expressions of type* **Nat** are those that "clearly" compute natural numbers, such as literal natural numbers (0, 1, 120), arithmetic expressions ($1 + 1$), and fully-applied functions that return natural numbers (`fact 5`, `atoi "120"`); the expressions of type **String** are those that clearly compute strings (`"hi"`, `itoa 5`); and for any types A and B, the expressions of type $A \to B$ are those that clearly compute functions that, when passed an input of type A, clearly compute an output of type B.

What do we mean by "clearly"? One typically insists that type-checking be fully automated, much like parsing and identifier resolution. Given that determining the result of a program is in general undecidable, any automated type-checking process will necessarily compute a conservative underapproximation of the set of programs that compute (e.g.) natural numbers. (Likewise, languages may complain about unbound identifiers even in programs that can be evaluated without a runtime error!)

The goal of a type system is thus to rule out as many undesirable programs as possible without ruling out too many desirable ones, where both of these notions are subjective depending on which runtime errors one wants to rule out and which programming idioms one wants to support. Language designers engage in the never-ending process of refining their type systems to rule out more errors and accept more correct code. Full-spectrum dependent types can be seen as an extreme point in this design space in the sense that they can capture highly sophisticated invariants of functional programs, as we will see momentarily.

Every introduction to dependent types starts with the example of vectors, or lists with specified length. We start one step earlier by considering lists with a specified type of elements, a type which already exhibits a basic form of dependency.

Parameterizing by types One of the most basic data structures in functional programming languages is the *list*, which is either empty (written []) or consists of an

[1] The latter perspective is valid, but we wish to draw a sharp distinction between types *qua* (structural) grammar, and static analyses that may be non-local, non-structural, or non-substitutive in nature.

element x adjoined to a list xs (written $x :: xs$). Typed languages typically require that a list's elements all have the same type, in order to track what operations they support.

The simplest way to record this information is to have a separate type of lists for each type of element: a **ListOfNats** is either empty or a **Nat** adjoined to a **ListOfNats**, a **ListOfStrings** is either empty or a **String** adjoined to a **ListOfStrings**, etc.

```
data ListOfNats : Set² where
  [] : ListOfNats
  _::_ : Nat → ListOfNats → ListOfNats

data ListOfStrings : Set where
  [] : ListOfStrings
  _::_ : String → ListOfStrings → ListOfStrings
```

This strategy clearly results in repetition at the level of the type system, but it also causes code duplication because operations that work uniformly for any type of elements (such as reversing a list) must be defined twice, once each for the two apparently unrelated types **ListOfNats** and **ListOfStrings**.

In much the same way that functions—terms indexed by terms—promote code reuse by allowing programmers to write a series of operations once and perform them on many different inputs, we can solve both problems described above by allowing types and terms to be uniformly parameterized by types. For example, we may consider the types **ListOfNats** and **ListOfStrings** as two instances (**List Nat** and **List String**) of a single family of types **List** as follows:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

and any operation that works for all element types A, such as returning the first (or all but first) element of a list, can be written as a family of operations:

```
head : (A : Set) → List A → A
head A [] = error "List must be non-empty."
head A (x :: xs) = x

tail : (A : Set) → List A → List A
tail A [] = error "List must be non-empty."
tail A (x :: xs) = xs
```

[2]For the time being, the reader should understand – : **Set** as notation meaning "– is a type."

By partially applying head to its type argument, we see that head **Nat** has type **List Nat** → **Nat** and head **String** has type **List String** → **String**, and the expression 1 + (head **Nat** (1 :: [])) has type **Nat** whereas 1 + (head **String** ("hi" :: [])) is ill-typed because the second input to + has type **String**.

Parameterizing types by terms The perfectionist reader may find the **List** A type unsatisfactory because it does not prevent runtime errors caused by applying head and tail to the empty list []. We cannot simply augment our types to track which lists are empty, because 2 :: 1 :: [] and 1 :: [] are both nonempty but we can apply tail **Nat** twice to the former before encountering an error, but only once to the latter.

Instead, we parameterize the type of lists not only by their type of elements but also by their length—a *term* of type **Nat**—producing the following family of types:[3]

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A 0
  _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
```

Types parameterized by terms are known as *dependent types*.

Now the types of concrete lists are more informative—(2 :: 1 :: []) : **Vec Int** 2 and (1 :: []) : **Vec Int** 1—but more importantly, we can give head and tail more informative types which rule out the runtime error of applying them to empty lists. We do so by revising their input type to **Vec** A (suc n) for some n : **Nat**, which is to say that the vector has length at least one, hence is nonempty:

```
head : {A : Set} {n : Nat} → Vec A (suc n) → A
-- head [] is impossible
head (x :: xs) = x


tail : {A : Set} {n : Nat} → Vec A (suc n) → Vec A n
-- tail [] is impossible
tail (x :: xs) = xs
```

Consider now the operation that concatenates two vectors:

```
append : {A : Set} {n : Nat} {m : Nat} → Vec A n → Vec A m → Vec A (n + m)
```

Unlike our previous examples, the output type of this function is indexed not by a variable A or n, nor a constant **Nat** or 0, nor even a constructor suc −, but by an *expression* $n + m$. This introduces a further complication, namely that we would like

[3] Curly braces {n : **Nat**} indicate *implicit* arguments automatically inferred by the type-checker; the term suc n constructs the successor $1 + n$ of a natural number n : **Nat**.

this expression to be simplified as soon as n and m are known. For example, if we apply append to two vectors of length one ($n = m = 1$), then the result will be a vector of length two ($n + m = 1 + 1 = 2$), and we would like the type system to be aware of this fact in the sense of accepting as well-typed the expression head (tail (append $l\ l'$)) for l and l' of type **Vec Nat** 1.

Because head (tail x) is only well-typed when x has type **Vec** A (suc (suc n)) for some n : **Nat**, this condition amounts to requiring that the expression append $l\ l'$ not only has type **Vec** A ((suc 0) + (suc 0)) as implied by the type of append, but also type **Vec** A (suc (suc 0)) as implied by its runtime behavior. In short, we would like the two type expressions **Vec** A (1 + 1) and **Vec** A 2 to *denote the same type* by virtue of the fact that 1 + 1 and 2 *denote the same value*. In practice, we achieve this by allowing the type-checker to *evaluate expressions in types during type-checking*.

In fact, the length of a vector can be any expression whatsoever of type **Nat**. Consider filter, which takes a function $f : A \to$ **Bool** and a list and returns the sublist for which f returns true. If the input list has length n, what is the length of the output?

$$\mathsf{filter} : \{A : \mathbf{Set}\}\ \{n : \mathbf{Nat}\} \to (A \to \mathbf{Bool}) \to \mathbf{Vec}\ A\ n \to \mathbf{Vec}\ A\ ?$$

After a moment's thought we realize the length is not a function of n at all, but rather a recursive function of the input function and list:

$$\begin{aligned}&\mathsf{filter} : \{A : \mathbf{Set}\}\ \{n : \mathbf{Nat}\} \to (f : A \to \mathbf{Bool}) \to (l : \mathbf{Vec}\ A\ n) \to \\ &\quad \mathbf{Vec}\ A\ (\mathsf{filterLen}\ f\ l)\end{aligned}$$

$$\begin{aligned}&\mathsf{filterLen} : \{A : \mathbf{Set}\}\ \{n : \mathbf{Nat}\} \to (A \to \mathbf{Bool}) \to \mathbf{Vec}\ A\ n \to \mathbf{Nat} \\ &\mathsf{filterLen}\ f\ [\,] = 0 \\ &\mathsf{filterLen}\ f\ (x :: xs) = \mathbf{if}\ f(x)\ \mathbf{then}\ \mathsf{suc}\ (\mathsf{filterLen}\ f\ xs)\ \mathbf{else}\ \mathsf{filterLen}\ f\ xs\end{aligned}$$

As before, once f and l are known the type of filter $f\ l$: **Vec** A (filterLen $f\ l$) will simplify by evaluating filterLen $f\ l$, but as long as either remains a variable we cannot learn much by computation. Nevertheless, filterLen has many properties of interest: filterLen $f\ l$ is at most the length of l, filterLen ($\lambda x \to$ false) l is always 0 regardless of l, etc. We will revisit this point in Section 1.3.

Remark 1.1.1. If we regard **Nat** and + as a user-defined data type and recursive function on it, as type theorists are wont to do, then filter's type using filterLen is entirely analogous to append's type using +. We wish to emphasize that, whereas one could easily imagine natural numbers and addition being a privileged component of the type system, filter demonstrates that type indices may need to contain arbitrary user-defined recursive functions. ◇

Another approach? If our only goal was to eliminate runtime errors from head and tail, we might reasonably feel that dependent types have overcomplicated the situation—we needed to introduce a new function just to write the type of filter! And indeed there are simpler ways of keeping track of the length of lists, as follows.

First let us observe that a lower bound on a list's length is sufficient to guarantee it is nonempty and thus that an application of head or tail will succeed; this allows us to trade precision for simplicity by restricting type indices to be arithmetic expressions. Secondly, in the above examples we can perform type-checking and "length-checking" in two separate phases, where the first phase replaces every occurrence of **Vec** $A\ n$ with **List** A before applying a standard non-dependent type-checking algorithm. This is possible because we can regard the dependency in **Vec** $A\ n$ as expressing a computable *refinement*—or subset—of the non-dependent type of lists, namely $\{l : \mathbf{List}\ A \mid \mathsf{length}\ l = n\}$.

Combining these insights, we can by and large automate length-checking by recasting the type dependency of **Vec** in terms of arithmetic inequality constraints over an ML-style type system, and checking these constraints with SMT solvers and other external tools. At a very high level, this is the approach taken by systems such as Dependent ML [Xi07] and Liquid Haskell [Vaz+14]. Dependent ML, for instance, type-checks the usual definition of filter at the following type, without any auxiliary filterLen definition:

$$\mathsf{filter} : \mathbf{Vec}\ A\ m \to (\{n : \mathbf{Nat} \mid n \leq m\} \times \mathbf{Vec}\ A\ n)$$

Refinement type systems like these have proven very useful in practice and continue to be actively developed, but we will not discuss them any further for the simple reason that, although they are a good solution to head/tail and many other examples, they cannot handle full-spectrum dependency as discussed below.

1.2 *Non-uniform dependency: computing arities*

Thus far, all our examples of (type- or term-) parameterized types are *uniformly* parameterized, in the sense that the functions **List** : **Set** → **Set** and **Vec** A : **Nat** → **Set** do not inspect their arguments; in contrast, ordinary term-level functions out of **Nat** such as fact : **Nat** → **Nat** can and usually do perform case-splits on their inputs. In particular, we have not yet considered any families of types in which the head, or top-level, type constructor (→, **Vec**, **Nat**, etc.) differs between indices.

A type theory is said to have full-spectrum dependency if it permits the use of *non-uniformly term-indexed* families of types, such as the following **Nat**-indexed family:

```
nary : Set → Nat → Set
nary A 0 = A
nary A (suc n) = A → nary A n
```

Although **Vec Nat** and nary **Nat** are both functions **Nat** → **Set**, the latter's head type constructor varies between indices: nary **Nat** 0 = **Nat** but nary **Nat** 1 = **Nat** → **Nat**.

Using nary to compute the type of n-ary functions, we can now define not only variadic functions but even higher-order functions taking variadic functions as input, such as apply which applies an n-ary function to a vector of length n:

```
apply : {A : Set} {n : Nat} → nary A n → Vec A n → A
apply x [] = x
apply f (x :: xs) = apply (f x) xs
```

For A = **Nat** and n = 1, apply applies a unary function **Nat** → **Nat** to the head element of a **Vec Nat** 1; for A = **Nat** and n = 3, it applies a ternary function **Nat** → **Nat** → **Nat** → **Nat** to the elements of a **Vec Nat** 3:

```
apply suc (1 :: []) : Nat    -- evaluates to 2
apply _+_ : Vec Nat 2 → Nat
apply _+_ (1 :: 2 :: []) : Nat    -- evaluates to 3
apply (λx y z → x + y + z) (1 :: 2 :: 3 :: []) : Nat    -- evaluates to 6
```

Although apply is not the first time we have seen a function whose type involves a different recursive function—we saw this already with filter—this is our first example of a function that cannot be straightforwardly typed in an ML-style type system. Another way to put it is that nary $A\ n$ → **Vec** $A\ n$ → A is not the refinement of an ML type because nary $A\ n$ is sometimes but not always a function type.

Remark 1.2.1. For the sake of completeness, it is also possible to consider *non-uniformly type-indexed* families of types, which go by a variety of names including non-parametric polymorphism, intensional type analysis, and typecase [HM95]. These often serve as optimized implementations of uniformly type-indexed families of types; a classic non-type-theoretic example is the C++ family of types `std::vector` for dynamically-sized arrays, whose `std::vector<bool>` instance may be compactly implemented using bitfields. ◇

To understand the practical ramifications of non-uniform dependency, we will turn our attention to a more complex example: a basic implementation of sprintf in Agda (Figure 1.1). This function takes as input a **String** containing format specifiers such as `%u` (indicating a **Nat**) or `%s` (indicating a **String**), as well as additional arguments of the

```
data Token : Set where
  char : Char → Token
  intTok : Token
  natTok : Token
  chrTok : Token
  strTok : Token

lex : List Char → List Token
lex [] = []
lex ('%' :: '%' :: cs) = char '%' :: lex cs
lex ('%' :: 'd' :: cs) = intTok :: lex cs
lex ('%' :: 'u' :: cs) = natTok :: lex cs
lex ('%' :: 'c' :: cs) = chrTok :: lex cs
lex ('%' :: 's' :: cs) = strTok :: lex cs
lex (c :: cs) = char c :: lex cs

args : List Token → Set
args [] = String
args (char _ :: toks) = args toks
args (intTok :: toks) = Int → args toks
args (natTok :: toks) = Nat → args toks
args (chrTok :: toks) = Char → args toks
args (strTok :: toks) = String → args toks

printfType : String → Set
printfType s = args (lex (toList s))

sprintf : (s : String) → printfType s
sprintf s = loop (lex (toList s)) ""
  where
  loop : (toks : List Token) → String → args toks
  loop [] acc = acc
  loop (char c :: toks) acc = loop toks (acc ++ fromList (c :: []))
  loop (intTok :: toks) acc = λ i → loop toks (acc ++ showInt i)
  loop (natTok :: toks) acc = λ n → loop toks (acc ++ showNat n)
  loop (chrTok :: toks) acc = λ c → loop toks (acc ++ fromList (c :: []))
  loop (strTok :: toks) acc = λ s → loop toks (acc ++ s)
```

Figure 1.1: A basic Agda implementation of sprintf.

appropriate type for each format specifier present, and returns a **String** in which each format specifier is replaced by the corresponding argument rendered as a **String**.

```
sprintf "%s %u" "hi" 2 : String   -- evaluates to "hi 2"
sprintf "%s" : String → String
sprintf "nat %u then int %d then char %c" : Nat → Int → Char → String
sprintf "%u" 5 : String   -- evaluates to "5"
sprintf "%u%% of %s%c" 3 "GD" 'P' : String   -- evaluates to "3% of GDP"
```

Our implementation uses various types and functions imported from Agda's standard library, notably toList : **String** → **List Char** which converts a string to a list of characters (length-one strings `'x'`). It consists of four main components:

- a data type Token which enumerates all relevant components of the input **String**, namely format specifiers (such as natTok : Token for `%u` and strTok : Token for `%s`) and literal characters (char `'x'` : Token);
- a function lex which tokenizes the input string, represented as a **List Char**, from left to right into a **List** Token for further processing;
- a function args which converts a **List** Token into a function type containing the additional arguments that sprintf must take; and
- the sprintf function itself.

Let us begin by convincing ourselves that our first example type-checks:

```
sprintf "%s %u" "hi" 2 : String   -- evaluates to "hi 2"
```

Because sprintf : (*s* : **String**) → printfType *s*, the partial application sprintf `"%s %u"` has type printfType `"%s %u"`. By evaluation, the type-checker can see that

$$\begin{aligned}\text{printfType "\%s \%u"} &= \text{args (strTok} :: \text{char ' '} :: \text{natTok} :: [])\\ &= \textbf{String} \to \textbf{Nat} \to \textbf{String}\end{aligned}$$

and thus sprintf `"%s %u"` : **String** → **Nat** → **String**; the remainder of the expression type-checks easily.

Now let us consider the definition of sprintf, which uses a helper function loop : (*toks* : **List** Token) → **String** → args *toks* whose first argument stores the Tokens yet to be processed, and whose second argument is the **String** accumulated from printing the already-processed Tokens. What is needed to type-check the definition of loop? We can examine a representative case in which the next Token is natTok:

$$\text{loop (natTok} :: \textit{toks}) \; \textit{acc} = \lambda n \to \text{loop } \textit{toks} \; (\textit{acc} \mathbin{+\!\!+} \text{showNat } n)$$

Note that *toks* : **List** Token and *acc* : **String** are (pattern) variables, and the right-hand side ought to have type args (natTok :: *toks*). We can type-check the right-hand side—given that _++_ : **String** → **String** → **String** is string concatenation and showNat : **Nat** → **String** prints a natural number—and observe that it has type **Nat** → args *toks* by the type of loop.

Type-checking this clause thus requires us to reconcile the right-hand side's expected type args (natTok :: *toks*) with its actual type **Nat** → args *toks*. Although these type expressions are quite dissimilar—one is a function type and the other is not—the definition of args contains a promising clause:

args (natTok :: *toks*) = **Nat** → args *toks*

As in our earlier example of **Vec** A (1 + 1) and **Vec** A 2 we would like the type expressions args (natTok :: *toks*) and **Nat** → args *toks* to denote the same type, but unlike the equation 1 + 1 = 2, here both sides contain a free variable *toks* so we cannot appeal to evaluation, which is a relation on *closed* terms (ones with no free variables).

One can nevertheless imagine some form of *symbolic evaluation* relation that extends evaluation to open terms and *can* equate these two expressions. In this particular case, this step of closed evaluation is syntactically indifferent to the value of *toks* and thus can be safely applied even when *toks* is a variable. (Likewise, to revisit an earlier example, the equation filterLen f [] = 0 should hold even for variable f.)

Thus we would like the type expressions args (natTok :: *toks*) and **Nat** → args *toks* to denote the same type by virtue of the fact that they *symbolically evaluate to the same symbolic value*, and to facilitate this we must allow the type-checker to *symbolically evaluate* expressions in types during type-checking. The congruence relation on expressions so induced is known as *definitional equality* because it contains defining clauses like this one.

Remark 1.2.2. Semantically we can justify this equation by observing that for any closed instantiation toks of *toks*, args (natTok :: toks) and **Nat** → args toks will evaluate to the same type expression—at least, once we have defined evaluation of type expressions—and thus this equation always holds at runtime. But just as (for reasons of decidability) the condition "when this expression is applied to a natural number it evaluates to a natural number" is a necessary but not sufficient condition for type-checking at **Nat** → **Nat**, we do not want to take this semantic condition as the definition of definitional equality. It is however a necessary condition assuming that the type system is sound for the given evaluation semantics. (See Section 3.4.) ◇

Definitional equality is the central concept in full-spectrum dependent type theory because it determines which types are equal and thus which terms have which types.

In practice, it is typically defined as the congruence closure of the β-like reductions (also known as $\beta\delta\zeta\iota$-reductions) plus η-equivalence at some types; see Chapter 2.

1.3 *Proving type equations*

Unfortunately, in light of Remark 1.2.2, there are many examples of type equations that are not direct consequences of ordinary or even symbolic evaluation. On occasion these equations are of such importance that researchers may attempt to make them definitional—that is, to include them in the definitional equality relation and adjust the type-checking algorithm accordingly [AMB13]. But such projects are often major research undertakings, and there are even examples of equations that can be definitional but are in practice best omitted due to efficiency or usability issues [Alt+01].

Let us turn once again to the example of filter from Section 1.1.

```
filter : {A : Set} {n : Nat} → (f : A → Bool) → (l : Vec A n) →
  Vec A (filterLen f l)

filterLen : {A : Set} {n : Nat} → (A → Bool) → Vec A n → Nat
filterLen f [] = 0
filterLen f (x :: xs) = if f(x) then suc (filterLen f xs) else filterLen f xs
```

Suppose for the sake of argument that we want the operation of filtering an arbitrary vector by the constantly false predicate to return a **Vec** A 0:

```
filterAll : {A : Set} {n : Nat} → Vec A n → Vec A 0
filterAll l = filter (λx → false) l    -- does not type-check
```

The right-hand side above has type **Vec** A (filterLen ($\lambda x \to$ false) l) rather than **Vec** A 0 as desired, and here the expression filterLen ($\lambda x \to$ false) l cannot be simplified by (symbolic) evaluation because filterLen computes by recursion on the vector, here a variable l. However, by induction on the possible instantiations of l : **Vec** A n, either:

- $l = []$, in which case filterLen ($\lambda x \to$ false) [] is definitionally equal to (in fact, evaluates to) 0; or
- $l = x :: xs$, in which case we have the definitional equalities

```
  filterLen (λx → false) (x :: xs)
= if false then suc (filterLen (λx → false) xs) else filterLen (λx → false) xs
= filterLen (λx → false) xs
```

for any x and xs. By the inductive hypothesis on xs, filterLen $(\lambda x \to \text{false})$ $xs = 0$ and thus filterLen $(\lambda x \to \text{false})$ $(x :: xs) = 0$ as well.

By adding a type of *provable equations* $a \equiv b$ to our language, we can represent this inductive proof as a recursive function computing filterLen $(\lambda x \to \text{false})$ $l \equiv 0$:

```
_≡_ : {A : Set} → A → A → Set
refl : {A : Set} {x : A} → x ≡ x

lemma : {A : Set} {n : Nat} → (l : Vec A n) → filterLen (λl → false) l ≡ 0
lemma [] = refl
lemma (x :: xs) = lemma xs
```

The [] clause of lemma ought to have type filterLen $(\lambda l \to \text{false})$ [] $\equiv 0$, which is definitionally equal to the type $0 \equiv 0$ and thus **refl** type-checks. The $(x :: xs)$ clause must have type filterLen $(\lambda l \to \text{false})$ $(x :: xs) \equiv 0$, which is definitionally equal to filterLen $(\lambda l \to \text{false})$ $xs \equiv 0$, the expected type of the recursive call lemma xs.

Now armed with a function lemma that constructs for any l : **Vec** A n a proof that filterLen $(\lambda l \to \text{false})$ $l \equiv 0$, we can justify *casting* from the type **Vec** A (filterLen $(\lambda l \to$ false) l) to **Vec** A 0. The dependent casting operation that passes between provably equal indices of a dependent type (here **Vec** A : **Nat** $\to$ **Set**) is typically called **subst**:

```
subst : {A : Set} {x y : A} → (P : A → Set) → x ≡ y → P(x) → P(y)

filterAll : {A : Set} {n : Nat} → Vec A n → Vec A 0
filterAll {A} l = subst (Vec A) (lemma l) (filter (λx → false) l)
```

Remark 1.3.1. The **subst** operation above is a special case of a much stronger principle stating that the two types $P(x)$ and $P(y)$ are *isomorphic* whenever $x \equiv y$: we can not only cast $P(x) \to P(y)$ but also $P(y) \to P(x)$ by symmetry of equality, and both round trips cancel. So although a proof $x \equiv y$ does not make the types $P(x)$ and $P(y)$ definitionally equal, they are nevertheless equal in the sense of having the same elements up to isomorphism. ◇

Uses of **subst** are very common in dependent type theory; because dependently-typed functions can both require and ensure complex invariants, one must frequently prove that the output of some function is a valid input to another.[4] Crucially, although **subst** is an "escape hatch" that compensates for the shortcomings of definitional

[4] A more realistic variant of our lemma might account for any predicate that returns false on all the elements of the given list, not just the constantly false predicate. Alternatively, one might prove that for any s : **String**, the final return type of sprintf s is **String**.

equality, it cannot result in runtime errors—unlike explicit casts in most programming languages—because casting $P(x) \to P(y)$ requires a machine-checked proof that $x \equiv y$.

The dependent type $x \equiv y$ is known as *propositional equality*, and it is perhaps the second most important concept in dependent type theory because it is the source of all non-definitional type equations visible within the theory. There are many formulations of propositional equality; they all implement _≡_, **refl**, and **subst** but differ in many other respects, and each has unique benefits and drawbacks. We will discuss propositional equality at length in Chapters 4 and 5.

To foreshadow the design space of propositional equality, consider that the **subst** operator may itself be subject to various definitional equalities. If we apply filterAll to a closed vector *ls*, then lemma *ls* will evaluate to **refl**, so filterAll *ls* is definitionally equal to **subst** (**Vec** *A*) **refl** (filter ($\lambda x \to$ false) *ls*). At this point, filter ($\lambda x \to$ false) *ls* already has the desired type **Vec** *A* 0 because filterLen ($\lambda x \to$ false) *ls* evaluates to 0, and thus the two types involved in the cast are now definitionally equal. Ideally the **subst** term would now disappear having completed its job, and indeed the definitional equality **subst** *P* **refl** $x = x$ does hold for many versions of propositional equality.

Programming and proving The propositional equality type $a \equiv b$ has a rather different flavor than **Nat**, $A \to B$, **Vec** *A n*, printfType *s*, and all the other types we have seen so far. This is perhaps most evident in our choice of terminology: whereas terms of the latter types all represent data or computations, terms of type $a \equiv b$ are machine-checked *proofs*, intended not as computations but as justifications for casts.

Indeed, as the reader may already know, dependent type theory is not just a typed functional programming language but also an expressive higher-order logic implemented in many modern proof assistants such as Lean [dMU21] and Rocq [Rocq]. This is certainly convenient in practice; as we saw in filterAll, proving theorems quickly becomes an important ingredient of dependently-typed programming.

What is surprising is not that programming and proving are in symbiosis, but that they are in fact two sides of the same coin—terms are simultaneously programs and proofs, and types are simultaneously program specifications and logical propositions—a remarkable fact known by many names, including the *propositions as types correspondence*, the *proofs as programs correspondence*, the *Curry–Howard correspondence*, and the *Brouwer–Heyting–Kolmogorov interpretation*.

We have already witnessed the proofs as programs correspondence at work in lemma, where we rendered a proof by induction on vectors as a recursive function

$$\text{lemma} : \{A : \mathbf{Set}\}\ \{n : \mathbf{Nat}\} \to (l : \mathbf{Vec}\ A\ n) \to \text{filterLen}\ (\lambda l \to \text{false})\ l \equiv 0$$

where type-checking the function's clauses amounts to proof-checking the inductive argument. We can go one step further: if lemma is a *dependent function* that given any

input l : **Vec** A n produces a proof of filterLen $(\lambda l \to \mathsf{false})$ $l \equiv 0$, then it is also a *proof* that *for all* l : **Vec** A n, filterLen $(\lambda l \to \mathsf{false})$ $l \equiv 0$. More generally, any dependent function $f : (x : A) \to B(x)$ is simultaneously a proof of the proposition $\forall x : A.\ B(x)$.

Dependent function types are but one of the many types with a secret life as a logical connective. As a second example, non-dependent function types $A \to B$ correspond logically to implications $A \implies B$, as in the following proposition asserting that for all m, n : **Nat**, if suc $m \equiv$ suc n, then $m \equiv n$.

$$\mathsf{sucInjective} : \{m\ n : \mathbf{Nat}\} \to (\mathsf{suc}\ m \equiv \mathsf{suc}\ n) \to m \equiv n$$

Why? To know the antecedent suc $m \equiv$ suc n is to have a proof p : suc $m \equiv$ suc n, in which case sucInjective $p : m \equiv n$ is a proof of the consequent $m \equiv n$.

More examples will arise as we become acquainted with more types, but we provide just one more for illustration. Given types A and B, expressions of product type $A \times B$ are pairs (a, b) of $a : A$ and $b : B$. Logically, product types $A \times B$ correspond to conjunctions $A \wedge B$, as to have a proof of a conjunction $A \wedge B$ is to have proofs of both A and of B. Conversely, the first and second projection functions **fst** : $(A \times B) \to A$ and **snd** : $(A \times B) \to B$ prove that $A \wedge B$ implies A and, separately, implies B.

Remark 1.3.2. There are several senses in which proofs correspond to programs and propositions to types. The most straightforward but superficial correspondence is the observation that the natural deduction rules governing (e.g.) logical implication are formally identical to the typing rules governing functions [How80]. From a philosophical perspective, the *meaning explanations* of Martin-Löf [Mar82] describe why, following the tenets of intuitionism, programming and constructive proof can be seen as one and the same activity. Finally, from a mathematical perspective, one can regard type theory as a formal logic which admits an interpretation in computable functions [Hyl82]. (See Section 3.4 for a brief discussion of the latter.) ◇

Although many types have clear interpretations as both program specifications and logical propositions, we note that some types have an obvious bias toward only one of the two readings. For example, **Nat** has a clear meaning in terms of data but not as a proposition, whereas $a \equiv b$ has a clear meaning as a proposition but not as data (see however Chapter 5!). We revisit this important point in Section 2.7; for now we simply remark that the propositions as types correspondence has played a central role in the advancement of both logic and programming languages.

Further reading

Our four categories of dependency—types/terms depending on types/terms—are reminiscent of the *lambda cube* of generalized type systems in which one augments the simply-typed lambda calculus (whose functions exhibit term-on-term dependency) with any combination of the remaining three forms of dependency [Bar91]; adding all three yields the full-spectrum dependent type theory known as the calculus of constructions [CH88]. However, the technical details of this line of work differ significantly from our presentation in Chapter 2.

The propositions as types correspondence exists in many forms and has been extended by researchers over decades to encompass a wide range of logical and programming constructs. Book-length expositions include *Proofs and Types* [GLT89] and *PROGRAM = PROOF* [Mim20]. Despite its importance to type theory we will discuss it only once more, in Section 2.7.

The code in this chapter is written in Agda syntax [Agda]. For more on dependently-typed programming in Agda, see *Verified Functional Programming in Agda* [Stu16]; for a more engineering-oriented perspective on dependent types, see *Type-Driven Development with Idris* [Bra17]. The sprintf example in Section 1.2 is inspired by the paper *Cayenne — A Language with Dependent Types* [Aug99]. Conversely, to learn about using Agda as a proof assistant for programming language theory, see *Programming Language Foundations in Agda* [WKS22].

This material will be published by Cambridge University Press as *Principles of Dependent Type Theory* by Carlo Angiuli and Daniel Gratzer. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © Carlo Angiuli and Daniel Gratzer 2024

2 *Extensional type theory*

In order to understand the subtle differences between modern dependent type theories, we must first study the formal definition of a dependent type theory as a mathematical object. We will then be prepared for Chapter 3, in which we consider various mathematical properties of type theory—particularly in connection to definitional and propositional equality—and how they affect computer implementations of type theory. In this chapter we therefore present the judgmental theory of Martin-Löf's *extensional type theory* [Mar82], one of the canonical variants of dependent type theory. We strongly suggest following the exposition rather than simply reading the rules, but the rules are collected for convenience in Appendix A (ignoring the rules marked with (ITT), which are present only in intensional type theory).

To focus our discussion we do not attempt to give a comprehensive account of the syntax of type theories, nor do we present any of the many alternative methods of defining type theory, some of which—see Chapter 6—are more efficient (but more technical) than the one we present here. These questions lead to the fascinating and deep area of *logical frameworks*, one approach to which we outline in Appendix B.

In this chapter In Section 2.1 we recall the concepts of judgments and inference rules in the setting of the simply-typed lambda calculus. In Section 2.2 we consider how to adapt these methods to the dependent setting, and in Section 2.3 we develop these ideas into the basic judgmental structure of dependent type theory, in which substitution plays a key role. In Section 2.4 we extend the basic rules of type theory with rules governing dependent products, dependent sums, extensional equality, and unit types. We argue that these connectives can be understood as *internalizations of judgmental structure*, a perspective which provides a conceptual justification of these connectives' rules. In Section 2.5 we define several inductive types—the empty type, booleans, coproducts, and natural numbers—and explain how and why these types do not fit the pattern of the previous section. In Section 2.6 we discuss large elimination, which is implicit in our examples of full-spectrum dependency from Section 1.2, and its internalization via universe types. Finally, in Section 2.7 we reconsider the propositions as types correspondence and argue that only certain types are logical propositions.

Goals of the chapter By the end of this chapter, you will be able to:

- Define the core judgments of dependent type theory, and explain how and why they differ from the judgments of simple type theory.
- Explain the role of substitutions in the syntax of dependent type theory.
- Define and justify the rules of the core connectives of type theory.

2.1 The simply-typed lambda calculus

The theory of typed functional programming is built on extensions of a core language known as the *simply-typed lambda calculus*, which supports two types of data:

- functions of type $A \to B$ (for any types A, B): we write $\lambda x.b$ for the function that sends any input x of type A to an output b of type B, and write $f\ a$ for the application of a function f of type $A \to B$ to an input a of type A; and
- ordered pairs of type $A \times B$ (for any types A, B): we write (a, b) for the pair of a term a of type A with a term b of type B, and write $\mathbf{fst}(p)$ and $\mathbf{snd}(p)$ respectively for the first and second projections of a pair p of type $A \times B$.

It can also be seen as the implication–conjunction fragment of intuitionistic propositional logic, or as an axiom system for cartesian closed categories.

In this section we formally define the simply-typed lambda calculus as a collection of judgments presented by inference rules, in order to prepare ourselves for the analogous—but considerably more complex—definition of dependent type theory in the remainder of this chapter. Our goal is thus not to give a textbook account of the simply-typed lambda calculus but to draw the reader's attention to issues that will arise in the dependent setting.

Readers familiar with the simply-typed lambda calculus should be aware that our definition does not reference the untyped lambda calculus (as discussed in Remark 2.1.2) and considers terms modulo $\beta\eta$-equivalence (Section 2.1.2).

2.1.1 Contexts, types, and terms

The simply-typed lambda calculus is made up of two *sorts*, or grammatical categories, namely types and terms. We present these sorts by two well-formedness *judgments*:

- the judgment A type stating that A is a well-formed type, and

- for any well-formed type A, the judgment $a : A$ stating that a is a well-formed term of that type.

By comprehension these judgments determine respectively the collection of well-formed types and, for every element of that collection, the collection of well-formed terms of that type. (From now on we will stop writing "well-formed" because we do not consider any other kind of types or terms; see Remark 2.1.2.)

Remark 2.1.1. A judgment is simply a proposition in our ambient mathematics, one which takes part in the definition of a logical theory; we use this terminology to distinguish such meta-propositions from the propositions of the logic that is being defined [Mar87]. Similarly, a sort is a type in the ambient mathematics, as distinguished from the types of the theory being defined. We refer to the ambient mathematics (in which our definition is being carried out) as the *metatheory* and the logic being defined as the *object theory*.

In this book we will be relatively agnostic about our metatheory, which the reader can imagine as "ordinary mathematics." However, one can often simplify matters by adopting a domain-specific metatheory (a *logical framework*) well-suited to defining languages/logics, as an additional level of indirection within the ambient metatheory; see Appendix B for an example of such a framework. ◇

Types We can easily define the types as the expressions generated by the following context-free grammar:

$$\textit{Types} \quad A, B ::= \quad \mathbf{b} \mid A \times B \mid A \to B$$

We say that the judgment A type ("A is a type") holds when A is a type in the above sense. Note that in addition to function and product types we have included a base type $\mathbf{b}$; without $\mathbf{b}$ the grammar would have no terminal symbols and would thus be empty.

Equivalently, we could define the A type judgment by three *inference rules* corresponding to the three production rules in the grammar of types:

$$\frac{}{\mathbf{b}\,\text{type}} \qquad \frac{A\,\text{type} \qquad B\,\text{type}}{A \times B\,\text{type}} \qquad \frac{A\,\text{type} \qquad B\,\text{type}}{A \to B\,\text{type}}$$

Each inference rule has some number of premises (here, zero or two) above the line and a single conclusion below the line; by combining these rules into trees whose leaves all have no premises, we can produce *derivations* of judgments (here, the well-formedness of a type) at the root of the tree. The tree below is a proof that $(\mathbf{b} \times \mathbf{b}) \to \mathbf{b}$

is a type:

$$\dfrac{\dfrac{\dfrac{}{\mathbf{b}\,\mathsf{type}} \qquad \dfrac{}{\mathbf{b}\,\mathsf{type}}}{\mathbf{b}\times\mathbf{b}\,\mathsf{type}} \qquad \dfrac{}{\mathbf{b}\,\mathsf{type}}}{(\mathbf{b}\times\mathbf{b})\to\mathbf{b}\,\mathsf{type}}$$

Terms Terms are considerably more complex than types, so before attempting a formal definition we will briefly summarize our intentions. For the remainder of this section, fix a finite set I. The well-formed terms are as follows:

- for any $i \in I$, the base term $\mathbf{c}_i$ has type $\mathbf{b}$;
- pairing (a, b) has type $A \times B$ when $a : A$ and $b : B$;
- first projection $\mathbf{fst}(p)$ has type A when $p : A \times B$;
- second projection $\mathbf{snd}(p)$ has type B when $p : A \times B$;
- a function $\lambda x.b$ has type $A \to B$ when $b : B$ where b can contain (in addition to the usual term formers) the variable term $x : A$ standing for the function's input; and
- a function application $f\ a$ has type B when $f : A \to B$ and $a : A$.

The first difficulty we encounter is that unlike types, which are a single sort, there are infinitely many sorts of terms (one for each type) many of which refer to one another. A more significant issue is to make sense of the clause for functions: the body b of a function $\lambda x.b : A \to B$ is a term of type B according to our original grammar *extended by* a new constant $x : A$ representing an indeterminate term of type A. Because b can again be or contain a function $\lambda y.c$, we must account for finitely many extensions $x : A, y : B, \ldots$.

To account for these extensions we introduce an auxiliary sort of *contexts*, or lists of variables paired with types, representing local extensions of our theory by variable terms.

Contexts The judgment $\vdash \Gamma\ \mathsf{cx}$ ("Γ is a context") expresses that Γ is a list of pairs of term variables with types. We write $\mathbf{1}$ for the empty context and $\Gamma, x : A$ for the extension of Γ by a term variable x of type A. As a context-free grammar, we might write:

$$\begin{array}{lrl} \textit{Variables} & x, y ::= & x \mid y \mid z \mid \cdots \\ \textit{Contexts} & \Gamma ::= & \mathbf{1} \mid \Gamma, x : A \end{array}$$

Equivalently, in inference rule notation:

$$\frac{}{\vdash \mathbf{1}\,\mathrm{cx}} \qquad \frac{\vdash \Gamma\,\mathrm{cx} \qquad A\,\mathrm{type}}{\vdash \Gamma, x : A\,\mathrm{cx}}$$

We will not spend time discussing variables or binding in this book because variables will, perhaps surprisingly, not be a part of our definition of dependent type theory. For the purposes of this section we will simply assume that there is an infinite set of variables $x, y, z \ldots$, and that all the variables in any given context or term are distinct.

Terms revisited With contexts in hand we are now ready to define the term judgment, which we revise to be relative to a context Γ. The judgment $\Gamma \vdash a : A$ ("a has type A in context Γ") is defined by the following inference rules:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{i \in I}{\Gamma \vdash \mathbf{c}_i : \mathbf{b}} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B} \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathbf{fst}(p) : A}$$

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathbf{snd}(p) : B} \qquad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \to B} \qquad \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}$$

The rules for $\mathbf{c}_i$, pairing, projections, and application straightforwardly render our text into inference rule form, framed by a context Γ that is unchanged from premises to conclusion. The lambda rule explains how contexts are changed: the body of a lambda is typed in an extended context; and the variable rule explains how contexts are used: in context Γ, the variables of type A in Γ serve as additional terminal symbols of type A.

Rules such as pairing or lambda that describe how to create terms of a given type former are known as *introduction* rules, and rules describing how to use terms of a given type former, like projection and application, are known as *elimination* rules.

Remark 2.1.2. An alternative approach that is perhaps more familiar to programming languages researchers is to define a collection of *preterms*

$$\textit{Terms} \quad a, b ::= \quad \mathbf{c}_i \mid x \mid (a, b) \mid \mathbf{fst}(a) \mid \mathbf{snd}(a) \mid \lambda x.a \mid a\ b$$

which includes ill-formed (typeless) terms like $\mathbf{fst}(\lambda x.x)$ in addition to the well-formed (typed) ones captured by our grammar above, and the inference rules are regarded as carving out various subsets of well-formed terms [Har16]. In fact, one often gives computational meaning to *all* preterms (as an extension of the untyped lambda calculus)

and then proves that the well-typed ones are in some sense computationally well-behaved.

This is *not* the approach we are taking here; to us the term expression $\mathbf{fst}(\lambda x.x)$ does not exist any more than the type expression $\to \times \to$.[1] In fact, in light of Section 2.1.2, there will not even exist a "forgetful" map from our collections of terms to these preterms. ◇

2.1.2 *Equational rules*

One shortcoming of our definition thus far is that our projections don't actually project anything and our function applications don't actually apply functions—there is no sense yet in which $\mathbf{fst}((a,b)) : A$ or $(\lambda x.x)\ a : A$ "are" $a : A$. Rather than equip our terms with operational meaning, we will *quotient* our terms by equations that capture a notion of sameness including these examples. The reader can imagine this process as analogous to the presentation of algebras by *generators and relations*, in which our terms thus far are the generators of a "free algebra" of (well-formed but) uninterpreted expressions.

Our true motivation for this quotient is to anticipate the definitional equality of dependent type theory, but there are certainly intrinsic reasons as well, perhaps most notably that the quotiented terms of the simply-typed lambda calculus serve as an axiom system for reasoning about cartesian closed categories [Cro94, Chapter 4].

We quotient by the congruence relation generated by the following rules:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathbf{fst}((a,b)) = a : A} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathbf{snd}((a,b)) = b : B}$$

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash p = (\mathbf{fst}(p), \mathbf{snd}(p)) : A \times B}$$

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b)\ a = b[a/x] : B} \qquad \frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f = \lambda x.(f\ x) : A \to B}$$

The equations pertaining to elimination after introduction (projection from pairs and application of lambdas) are called *β-equivalences*; the equations pertaining to

[1] Perhaps one's definition of context-free grammar carves out the grammatical expressions out of arbitrary strings over an alphabet, but this process occurs at a different level of abstraction. The reader should banish such thoughts along with their thoughts about terms with mismatched parentheses.

introduction after elimination (pairs of projections and lambdas of applications) are *η-equivalences*.

We emphasize that these equations are not *a priori* directed, and are not restricted to the "top level" of terms; we genuinely take the quotient of the collection of terms at each type by these equations, automatically inducing equations such as $\lambda x.x = \lambda x.\mathbf{fst}((x, x))$.

The first two rules explain that projecting from a pair has the evident effect. The third rule states that every term of type $A \times B$ can be written as a pair (of its projections), in effect transforming the introduction rule for products from merely a sufficient condition to a necessary one as well. Similarly, the fifth rule states that every $f : A \to B$ can be written as a lambda (of its application).

The fourth rule explains that applying a lambda function $\lambda x.b$ to an argument a is equal to the body b of that lambda with all occurrences of the placeholder variable x replaced by the term a. However, this equation makes reference to a *substitution* operation $b[a/x]$ ("substitute a for x in b") that we have not yet defined.

Substitution We can define substitution $b[a/x]$ by structural recursion on b:

$$
\begin{aligned}
\mathbf{c}_i[c/x] &:= \mathbf{c}_i \\
x[c/x] &:= c \\
y[c/x] &:= y && \text{(for } x \neq y\text{)} \\
(a, b)[c/x] &:= (a[c/x], b[c/x]) \\
\mathbf{fst}(p)[c/x] &:= \mathbf{fst}(p[c/x]) \\
\mathbf{snd}(p)[c/x] &:= \mathbf{snd}(p[c/x]) \\
(\lambda y.b)[c/x] &:= \lambda y.b[c/x] && \text{(for } x \neq y \text{ and } y \text{ not a free variable in } c\text{)} \\
(f\ a)[c/x] &:= f[c/x]\ a[c/x]
\end{aligned}
$$

In the case of substituting into a lambda $(\lambda y.b)[c/x]$, we assume that the bound variable y introduced by the lambda is different from the variable x being substituted away and that y does not happen to occur freely in c. In practice both situations are possible, in which case one must rename y (and all references to y in b) before applying this rule. In any case, we intend this substitution to be *capture-avoiding* in the sense of not inadvertently changing the referent of bound variables.

However, because we have quotiented our collection of terms by $\beta\eta$-equivalence, it is not obvious that substitution is well-defined as a function out of the collection of terms; in order to map out of the quotient, we must check that substitution behaves equally on equal terms. (It is also not obvious that substitution is a function *into* the collection of terms, as we will discuss shortly.)

Consider the equation $\mathbf{fst}((a, b)) = a$. To see that substitution respects this equation, we can substitute into the left-hand side, yielding:

$$(\mathbf{fst}((a, b)))[c/x] = \mathbf{fst}((a, b)[c/x]) = \mathbf{fst}((a[c/x], b[c/x]))$$

which is β-equivalent to $a[c/x]$, the result of substituting into the right-hand side. We can check the remaining equations in a similar fashion; the $x \neq y$ condition on substitution into lambdas is necessary for substitution to respect β-equivalence of functions.

2.1.3 *Who type-checks the typing rules?*

Our stated goal in Section 2.1.1 was to define a collection of well-formed types (written A type), and for each of these a collection of well-formed terms (written $a : A$). Have we succeeded? First of all, our definition of terms is now indexed by contexts Γ and written $\Gamma \vdash a : A$, to account for variables introduced by lambdas. This is no problem: we recover the original notion of (closed) term by considering the empty context $\mathbf{1}$. Nor is there any issue defining the collections of types $\mathsf{Ty} = \{A \mid A\ \mathsf{type}\}$ and contexts $\mathsf{Cx} = \{\Gamma \mid \vdash \Gamma\ \mathsf{cx}\}$ as presented by the grammars or inference rules in Section 2.1.1.

It is less clear that the collections of *terms* are well-defined. We would like to say that the collection of terms of type A in context Γ, $\mathsf{Tm}(\Gamma, A)$, is the set of a for which there exists a derivation of $\Gamma \vdash a : A$, modulo the relation $a \sim b \iff$ there exists a derivation of $\Gamma \vdash a = b : A$. Several questions arise immediately; for instance, is it the case that whenever $\Gamma \vdash a : A$ is derivable, Γ is a context and A is a type? If not, then we have some "junk" judgments that do not correspond to elements of some $\mathsf{Tm}(\Gamma, A)$.

Lemma 2.1.3. *If* $\Gamma \vdash a : A$ *then* $\vdash \Gamma\ \mathsf{cx}$ *and* $A\ \mathsf{type}$.

To prove such a statement, one proceeds by induction on derivations of $\Gamma \vdash a : A$. If, say, the derivation ends as follows:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash p : A \times B}}{\Gamma \vdash \mathbf{fst}(p) : A}$$

then the inductive hypothesis applied to the derivation of $\Gamma \vdash p : A \times B$ tells us that $\vdash \Gamma\ \mathsf{cx}$ and $A \times B\ \mathsf{type}$. The former is exactly one of the two statements we are trying to prove. The other, $A\ \mathsf{type}$, follows from an "inversion lemma" (proven by cases on the $-\ \mathsf{type}$ judgment) that $A\ \mathsf{type}$ and $B\ \mathsf{type}$ is not only a sufficient but also a necessary condition for $A \times B\ \mathsf{type}$.

Unfortunately our proof runs into an issue at the base cases, or at least it is not clear over what Γ the following rules range:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{i \in I}{\Gamma \vdash \mathbf{c}_i : \mathbf{b}}$$

We must either add premises to these rules stating $\vdash \Gamma\,\mathsf{cx}$, or else clarify that Γ always ranges only over contexts (which will be our strategy moving forward; see Notation 2.2.1).

Another question is the well-definedness of our quotient:

Lemma 2.1.4. *If* $\Gamma \vdash a = b : A$ *then* $\Gamma \vdash a : A$ *and* $\Gamma \vdash b : A$.

But because β-equivalence refers to substitution, proving this lemma requires:

Lemma 2.1.5 (Substitution)**.** *If* $\Gamma, x : A \vdash b : B$ *and* $\Gamma \vdash a : A$ *then* $\Gamma \vdash b[a/x] : B$.

We already saw that we must check that substitution $b[a/x]$ respects equality of b, but we must also check that it produces well-formed terms, again by induction on b. Note that substitution changes a term's context because it eliminates one of its free variables.

If we resume our attempt to prove Lemma 2.1.4, we will notice that substitution is not the only time that the context of a term changes; in the right-hand side of the η-rule of functions, f is in context $\Gamma, x : A$, whereas in the premise and left-hand side it is in Γ:

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f = \lambda x.(f\ x) : A \to B}$$

And thus we need yet another lemma.

Lemma 2.1.6 (Weakening)**.** *If* $\Gamma \vdash b : B$ *and* $A\,\mathsf{type}$ *then* $\Gamma, x : A \vdash b : B$.

We will not belabor the point any further; eventually one proves enough lemmas to conclude that we have a set of contexts Cx, a set of types Ty, and for every $\Gamma \in \mathsf{Cx}$ and $A \in \mathsf{Ty}$ a set of terms $\mathsf{Tm}(\Gamma, A)$. The complexity of each result is proportional to the complexity of that sort's definition: we define types outright, contexts by simple reference to types, and terms by more complex reference to both types and contexts. The judgments of dependent type theory are both more complex and more intertwined; rather than enduring proportionally more suffering, we will adopt a slightly different approach.

Finally, whereas all the metatheorems mentioned in this section serve only to establish that our definition is mathematically sensible, there are more genuinely interesting and contentful metatheorems one might wish to prove, including *canonicity*,

the statement that (up to equality) the only closed terms of $\mathbf{b}$ are of the form $\mathbf{c}_i$ (i.e., $\mathsf{Tm}(\mathbf{1}, \mathbf{b}) = \{\mathbf{c}_i\}_{i \in I}$), and *decidability of equality*, the statement that for any $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$ we can write a program which determines whether or not $\Gamma \vdash a = b : A$.

2.2 *Towards the syntax of dependent type theory*

The reader is forewarned that the rules in this section serve to bridge the gap between Section 2.1 and our "official" rules for extensional type theory, which start in Section 2.3.

As we discussed in Chapter 1, the defining distinction between dependent and simple type theory is that in the former, types can contain term expressions and even term variables. Thus, whereas in Section 2.1 a simple context-free grammar sufficed to define the collection of types and we needed a context-sensitive system of inference rules to define the well-typed terms, in dependent type theory we will find that both the types and terms are context-sensitive because they refer to one another.

Types and contexts When is the dependent function type $(x : A) \to B$ well-formed? Certainly A and B must be well-formed types, but B is allowed to contain the term variable $x : A$ whereas A is not. In the case of $(n : \mathbf{Nat}) \to \mathbf{Vec}\ \mathbf{String}\ (\mathsf{suc}\ n)$, the well-formedness of the codomain depends on the fact that $\mathsf{suc}\ n$ is a well-formed term of type **Nat** (the indexing type of **Vec String**), which in turn depends on the fact that n is known to be an expression (in particular, a variable) of type **Nat**.

Thus as with the *term* judgment of Section 2.1, the *type* judgment of dependent type theory must have access to the context of term variables, so we replace the $A\,\mathsf{type}$ judgment ("A is a type") of the simply-typed lambda calculus with a judgment $\Gamma \vdash A\,\mathsf{type}$ ("A is a type in context Γ"). This innocuous change has many downstream implications, so we will be fastidious about the context in which a type is well-formed.

The first consequence of this change is that contexts of term variables, which we previously defined simply as lists of well-formed types, must now also take into account *in what context* each type is well-formed. Informally we say that each type can depend on all the variables before it in the context; formally, one might define the judgment $\vdash \Gamma\,\mathsf{cx}$ by the following pair of rules:

$$\frac{}{\vdash \mathbf{1}\,\mathsf{cx}} \qquad\qquad \frac{\vdash \Gamma\,\mathsf{cx} \qquad \Gamma \vdash A\,\mathsf{type}}{\vdash \Gamma, x : A\,\mathsf{cx}}$$

Notice that the rules defining the judgment $\vdash \Gamma\,\mathsf{cx}$ refer to the judgment $\Gamma \vdash A\,\mathsf{type}$, which in turn depends on our notion of context. This kind of mutual dependence will continue to crop up throughout the rules of dependent type theory.

Notation 2.2.1 (Presuppositions)**.** With a more complex notion of context, it is more important than ever for us to decide over what Γ the judgment $\Gamma \vdash A\,\mathsf{type}$ ranges. We will say that the judgment $\Gamma \vdash A\,\mathsf{type}$ is only well-formed when $\vdash \Gamma\,\mathsf{cx}$ holds, as a matter of "meta-type discipline," and similarly that the judgment $\Gamma \vdash a : A$ is only well-formed when $\Gamma \vdash A\,\mathsf{type}$ (and thus also $\vdash \Gamma\,\mathsf{cx}$).

One often says that $\vdash \Gamma\,\mathsf{cx}$ is a *presupposition* of the judgment $\Gamma \vdash A\,\mathsf{type}$, and that the judgments $\vdash \Gamma\,\mathsf{cx}$ and $\Gamma \vdash A\,\mathsf{type}$ are presuppositions of $\Gamma \vdash a : A$. We will globally adopt the convention that whenever we assert the truth of some judgment in prose or as the premise of a rule, we also implicitly assert that its presuppositions hold. Dually, we will be careful to check that none of our rules have meta-ill-typed conclusions.

Now that we have added a term variable context to the type well-formedness judgment, we can explain when $(x : A) \to B$ is a type: it is a (well-formed) type in Γ when A is a type in Γ and B is a type in $\Gamma, x : A$, as follows.

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma, x : A \vdash B\,\mathsf{type}}{\Gamma \vdash (x : A) \to B\,\mathsf{type}}$$

Rules like this describing how to create a type are known as *formation rules*, to parallel the terminology of introduction and elimination rules.

We can now sketch the formation rules for many of the types we encountered in Chapter 1. Dependent types like _≡_ and **Vec** are particularly interesting because they entangle the $\Gamma \vdash A\,\mathsf{type}$ judgment with the term well-formedness judgment $\Gamma \vdash a : A$.

$$\frac{\vdash \Gamma\,\mathsf{cx}}{\Gamma \vdash \mathbf{Nat}\,\mathsf{type}} \qquad \frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash n : \mathbf{Nat}}{\Gamma \vdash \mathbf{Vec}\ A\ n\,\mathsf{type}} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a \equiv b\,\mathsf{type}}$$

Note that the convention of presuppositions outlined in Notation 2.2.1 means that the second and third rules have an implicit $\vdash \Gamma\,\mathsf{cx}$ premise, and the third rule also has an implicit $\Gamma \vdash A\,\mathsf{type}$ premise. To see that the conclusions of these rules are meta-well-typed, we must check that $\vdash \Gamma\,\mathsf{cx}$ holds in each case; this is an explicit premise of the first rule and a presupposition of the premises of the second and third rules.

The formation rule for propositional equality _≡_ in particular is a major source of dependency because it singlehandedly allows arbitrary terms of arbitrary type to occur within types. In fact, this rule by itself causes the inference rules of all three judgments $\vdash \Gamma\,\mathsf{cx}$, $\Gamma \vdash A\,\mathsf{type}$, and $\Gamma \vdash a : A$ to depend on one another pairwise.

Exercise 2.1. Attempt to derive that $(n : \mathbf{Nat}) \to \mathbf{Vec}\ \mathbf{String}\ (\mathsf{suc}\ n)$ is a well-formed type in the empty context $\mathbf{1}$, using the rules introduced in this section thus far. Several rules are missing; which judgments can you not yet derive?

The variable rule Let us turn now to the term judgment $\Gamma \vdash a : A$, and in particular the rule stating that term variables in the context are well-formed terms. For simplicity, imagine the special case where the last variable is the one under consideration:

$$\frac{}{\Gamma, x : A \vdash x : A}\ !?$$

This rule needs considerable work, as neither of the conclusion's presuppositions, $\vdash (\Gamma, x : A)\,\mathsf{cx}$ and $\Gamma, x : A \vdash A\,\mathsf{type}$, currently hold. We can address the former by adding premises $\vdash \Gamma\,\mathsf{cx}$ and $\Gamma \vdash A\,\mathsf{type}$ to the rule, from which it follows that $\vdash (\Gamma, x : A)\,\mathsf{cx}$.[2] As for the latter, note that $\Gamma \vdash A\,\mathsf{type}$ does not actually imply $\Gamma, x : A \vdash A\,\mathsf{type}$—this would require proving a *weakening lemma* (see Lemma 2.1.6) for types! (Conversely, if the rule has the premise $\Gamma \vdash A\,\mathsf{type}$, then we cannot establish well-formedness of the context.)

There are several ways to proceed. One is to prove a weakening lemma, but given that the well-formedness of the variable rule requires weakening, it is necessary to prove all our well-formedness, weakening, and substitution lemmas by a rather heavy simultaneous induction. A second approach would be to add a silent weakening *rule* stating that $\Gamma, x : A \vdash B\,\mathsf{type}$ whenever $\Gamma \vdash B\,\mathsf{type}$; however, this introduces ambiguity into our rules regarding the context(s) in which a type or term is well-formed.

We opt for a third option, which is to add *explicit* weakening rules asserting the existence of an operation sending types and terms in context Γ to types and terms in context $\Gamma, x : A$, both written $-[\mathbf{p}]$. (This notation will become less mysterious later.)

$$\frac{\Gamma \vdash B\,\mathsf{type} \qquad \Gamma \vdash A\,\mathsf{type}}{\Gamma, x : A \vdash B[\mathbf{p}]\,\mathsf{type}} \qquad\qquad \frac{\Gamma \vdash b : B \qquad \Gamma \vdash A\,\mathsf{type}}{\Gamma, x : A \vdash b[\mathbf{p}] : B[\mathbf{p}]}$$

Note that the type weakening rule is needed to make sense of the term weakening rule.

We can now fix the variable rule we wrote above: using $-[\mathbf{p}]$ to weaken A by itself, we move A from context Γ to $\Gamma, x : A$ as required in the conclusion of the rule.

$$\frac{\vdash \Gamma\,\mathsf{cx} \qquad \Gamma \vdash A\,\mathsf{type}}{\Gamma, x : A \vdash x : A[\mathbf{p}]}$$

To use variables that occur earlier in the context, we can apply weakening repeatedly until they are the last variable. Suppose that $\mathbf{1} \vdash A\,\mathsf{type}$ and $x : A \vdash B\,\mathsf{type}$, and in the context $\mathbf{1}, x : A, y : B$ we want to use the variable x. Ignoring the $y : B$ in the context for a moment, we know that $\mathbf{1}, x : A \vdash x : A[\mathbf{p}]$ by the last variable rule; thus

[2] Of course one could just directly add the premise $\vdash (\Gamma, x : A)\,\mathsf{cx}$, but our short-term memory is robust enough to recall that our next task is to ensure that A is a type.

by weakening we have $\mathbf{1}, x : A, y : B \vdash x[\mathbf{p}] : A[\mathbf{p}][\mathbf{p}]$. In general, we can derive the following principle:

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma, x : A \vdash B_1\,\mathsf{type} \qquad \dots \qquad \Gamma, x : A, y_1 : B_1, \dots \vdash B_n\,\mathsf{type}}{\Gamma, x : A, y_1 : B_1, \dots, y_n : B_n \vdash x\underbrace{[\mathbf{p}]\dots[\mathbf{p}]}_{n\text{ times}} : A\underbrace{[\mathbf{p}]\dots[\mathbf{p}]}_{n+1\text{ times}}}$$

This approach to variables is elegant in that it breaks the standard variable rule into two simpler primitives: a rule for the last variable, and rules for type and term weakening. However, it introduces a redundancy in our notation, because the term $x[\mathbf{p}]^n$ encodes in two different ways the variable to which it refers: by the name x as well as positionally by the number of weakenings n.

A happy accident of our presentation of the variable rule is thus that we can delete variable names altogether; in Section 2.3 we will present contexts simply as lists of types $A.B.C$ with no variable names, and adopt a single notation for "the last variable in the context," an encoding of the lambda calculus known as *de Bruijn indexing* [dBru72]. Conceptual elegance notwithstanding, this notation is very unfriendly to the reader in larger examples[3] so we will continue to use named variables outside of the rules themselves; translating between the two notations is purely mechanical.

2.3 *The calculus of substitutions*

Weakening is one of two main operations in type theory that moves types and terms between contexts, the other being substitution of terms for variables. For the same reasons that we want to present weakening as an explicit type- and term-forming operation, we will also formulate substitution as an explicit operation subject to equations explicating how it computes on each construct of the theory.

However, rather than axiomatizing *single* substitutions and weakenings, we will axiomatize arbitrary compositions of substitutions and weakenings. In light of the fact that substitution shortens the context of a type/term and weakening lengthens it, these composite operations—called *simultaneous substitutions* (henceforth just substitutions)—can turn any context Γ into any other context Δ.

We thus add one final judgment to our presentation of type theory, $\Delta \vdash \gamma : \Gamma$ ("γ is a substitution from Δ to Γ"), corresponding to operations that send types/terms from context Γ to context Δ. (Not a typo; we will address the "backwards" notation later.)

Notation 2.3.1. Type theory has four basic judgments and three equality judgments:

[3] According to Conor McBride, "Bob Atkey once memorably described the capacity to put up with de Bruijn indices as a Cylon detector." (https://mazzo.li/epilogue/index.html%3Fp=773.html)

1. $\vdash \Gamma\ \mathsf{cx}$ asserts that Γ is a context.
2. $\Delta \vdash \gamma : \Gamma$, presupposing $\vdash \Delta\ \mathsf{cx}$ and $\vdash \Gamma\ \mathsf{cx}$, asserts that γ is a substitution from Δ to Γ.
3. $\Gamma \vdash A\ \mathsf{type}$, presupposing $\vdash \Gamma\ \mathsf{cx}$, asserts that A is a type in context Γ.
4. $\Gamma \vdash a : A$, presupposing $\vdash \Gamma\ \mathsf{cx}$ and $\Gamma \vdash A\ \mathsf{type}$, asserts that a is an element/term of type A in context Γ.

2′. $\Delta \vdash \gamma = \gamma' : \Gamma$, presupposing $\Delta \vdash \gamma : \Gamma$ and $\Delta \vdash \gamma' : \Gamma$, asserts that γ, γ' are equal substitutions from Δ to Γ.

3′. $\Gamma \vdash A = A'\ \mathsf{type}$, presupposing $\Gamma \vdash A\ \mathsf{type}$ and $\Gamma \vdash A'\ \mathsf{type}$, asserts that A, A' are equal types in context Γ.

4′. $\Gamma \vdash a = a' : A$, presupposing $\Gamma \vdash a : A$ and $\Gamma \vdash a' : A$, asserts that a, a' are equal elements of type A in context Γ.

Notation 2.3.2. We write Cx for the set of contexts, $\mathsf{Sb}(\Delta, \Gamma)$ for the set of substitutions from Δ to Γ, $\mathsf{Ty}(\Gamma)$ for the set of types in context Γ, and $\mathsf{Tm}(\Gamma, A)$ for the set of terms of type A in context Γ.

This presentation of dependent type theory is known as the *substitution calculus* [Mar92; Tas93]. Perhaps unsurprisingly, we must discuss a considerable number of rules governing substitutions before presenting any concrete type and term formers; we devote this section to those rules, and cover the main connectives of type theory in Section 2.4.

Contexts The rules for contexts are as in Section 2.2, but without variable names:

$$\frac{}{\vdash \mathbf{1}\ \mathsf{cx}} \qquad \frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma \vdash A\ \mathsf{type}}{\vdash \Gamma.A\ \mathsf{cx}}$$

Although there is no context equality judgment, note that two contexts *can* be equal without being syntactically identical. If $\mathbf{1} \vdash A = A'\ \mathsf{type}$ then $\mathbf{1}.A$ and $\mathbf{1}.A'$ are equal contexts on the basis that, like all operations of the theory, context extension respects equality in both arguments. We have omitted the $\vdash \Gamma = \Gamma'\ \mathsf{cx}$ judgment for the simple reason that there would be no rules governing it: the only reason why two contexts can be equal is that their types are pairwise equal.

Substitutions The purpose of a substitution $\Delta \vdash \gamma : \Gamma$ is to shift types and terms from context Γ to context Δ:

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type}}{\Delta \vdash A[\gamma]\,\mathsf{type}} \qquad\qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A}{\Delta \vdash a[\gamma] : A[\gamma]}$$

Unlike the substitution operation of Section 2.1, which was a function on terms defined by cases, these rules define two binary type- and term- forming operations that take a type (resp., term) and a substitution as input and produce a new type (resp., term). Note also that, despite sharing a notation, type and term substitution are two distinct operations.

The simplest interesting substitution is weakening, written $\mathbf{p}$:[4]

$$\frac{\Gamma \vdash A\,\mathsf{type}}{\Gamma.A \vdash \mathbf{p} : \Gamma}$$

In concert with the substitution rules above we can recover the weakening rules from the previous section, e.g., if $\Gamma \vdash B\,\mathsf{type}$ and $\Gamma \vdash A\,\mathsf{type}$ then $\Gamma, x : A \vdash B[\mathbf{p}]\,\mathsf{type}$.

Because substitutions $\Delta \vdash \gamma : \Gamma$ encode arbitrary compositions of context-shifting operations, we also have rules that close substitutions under nullary and binary composition:

$$\frac{\vdash \Gamma\,\mathsf{cx}}{\Gamma \vdash \mathbf{id} : \Gamma} \qquad\qquad \frac{\Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0}{\Gamma_2 \vdash \gamma_0 \circ \gamma_1 : \Gamma_0}$$

These operations are unital and associative as one might expect:

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \gamma \circ \mathbf{id} = \mathbf{id} \circ \gamma = \gamma : \Gamma} \qquad \frac{\Gamma_3 \vdash \gamma_2 : \Gamma_2 \qquad \Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0}{\Gamma_3 \vdash \gamma_0 \circ (\gamma_1 \circ \gamma_2) = (\gamma_0 \circ \gamma_1) \circ \gamma_2 : \Gamma_0}$$

We can summarize the rules above by stating that there is a *category* whose objects are contexts and whose morphisms are substitutions.

We have already seen that substitutions shift the contexts of types and terms by $-[\gamma]$; they also shift the context of other substitutions by precomposition. Later we will have occasion to discuss all three context-shifting functions between sorts that are induced by substitutions, as follows.

Notation 2.3.3. Given a substitution $\Delta \vdash \gamma : \Gamma$, we write γ^* for the following functions:

- $\xi \mapsto \xi \circ \gamma : \mathsf{Sb}(\Gamma, \Xi) \to \mathsf{Sb}(\Delta, \Xi)$,

[4] This mysterious name can be explained by the fact that weakening corresponds semantically to a projection map; $\mathbf{p}$ can thus be pronounced as either "weakening" or "projection".

- $A \mapsto A[\gamma] : \mathsf{Ty}(\Gamma) \to \mathsf{Ty}(\Delta)$, and
- $a \mapsto a[\gamma] : \mathsf{Tm}(\Gamma, A) \to \mathsf{Tm}(\Delta, A[\gamma])$.

Composite substitutions introduce a possible redundancy into our rules: what is the difference between substituting by γ_0 and then by γ_1 versus substituting once by $\gamma_0 \circ \gamma_1$? We add equations asserting that substituting by $\mathbf{id}$ is the identity and substituting by a composite is composition of substitutions:

$$\frac{\Gamma \vdash A\,\mathsf{type}}{\Gamma \vdash A[\mathbf{id}] = A\,\mathsf{type}} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash a[\mathbf{id}] = a : A}$$

$$\frac{\Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0 \qquad \Gamma_0 \vdash A\,\mathsf{type}}{\Gamma_2 \vdash A[\gamma_0 \circ \gamma_1] = A[\gamma_0][\gamma_1]\,\mathsf{type}}$$

$$\frac{\Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0 \qquad \Gamma_0 \vdash a : A}{\Gamma_2 \vdash a[\gamma_0 \circ \gamma_1] = a[\gamma_0][\gamma_1] : A[\gamma_0 \circ \gamma_1]}$$

We can summarize the rules above by stating that the γ^* operations respect identity and composition of substitutions, or more compactly, that the collections of types and terms form *presheaves* $\mathsf{Ty}(-)$ and $\sum_{A:\mathsf{Ty}(-)} \mathsf{Tm}(-, A)$ on the category of contexts, with restriction maps given by substitution (a perspective which inspires the notation γ^*).

Before moving on, it is instructive to once again convince ourselves that the rules above are meta-well-typed. In particular, the conclusion of the second rule is only sensible if $\Gamma \vdash a[\mathbf{id}] : A$, but according to the rule for term substitution we only have $\Gamma \vdash a[\mathbf{id}] : A[\mathbf{id}]$. To make sense of this rule we must refer to the previous rule equating the types $A[\mathbf{id}]$ and A. A consequence of this type equation is that terms of type $A[\mathbf{id}]$ are equivalently terms of type A,[5] and thus $\Gamma \vdash a[\mathbf{id}] : A$ as required. This is a paradigmatic example of the deeply intertwined nature of the rules of dependent type theory; in particular, *we cannot defer equations* to the end of our construction the way we did in Section 2.1 because many rules are only sensible after imposing certain equations.

[5] In some presentations of type theory this principle is explicit and is known as the *type conversion rule*. For us it is a consequence of the judgments respecting equality, i.e., $\mathsf{Tm}(\Gamma, A[\mathbf{id}]) = \mathsf{Tm}(\Gamma, A)$ as sets.

The variable rule revisited As in the previous section, the variable rule is restricted to the last entry in the context, which we (unambiguously) always name $\mathbf{q}$.[6]

$$\frac{\Gamma \vdash A\,\mathsf{type}}{\Gamma.A \vdash \mathbf{q} : A[\mathbf{p}]}$$

Writing $\mathbf{p}^n$ for the n-fold composition of $\mathbf{p}$ with itself (with $\mathbf{p}^0 = \mathbf{id}$), the following rule is *derivable* from other rules (notated $\Rightarrow$) and thus not explicitly included in our system:

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B_1\,\mathsf{type} \qquad \dots \qquad \Gamma.A.B_1 \dots \vdash B_n\,\mathsf{type}}{\Gamma.A.B_1 \dots B_n \vdash \mathbf{q}[\mathbf{p}^n] : A[\mathbf{p}^{n+1}]} \Rightarrow$$

Thus a variable in our system is a term of the form $\mathbf{q}[\mathbf{p}^n]$, where n is its de Bruijn index.

Terminal substitutions Our notation $\Delta \vdash \gamma : \Gamma$ for substitutions is no accident; it is indeed a good mental model to think of such substitutions as "terms of type Γ in context Δ." To understand why, let us think back to propositional logic. A term $\mathbf{1}.B \vdash c : C$ can be seen as a proof of C under the hypothesis B, i.e., a proof that $B \implies C$. Given a substitution $\mathbf{1}.A \vdash b : \mathbf{1}.B$ we can obtain a term $\mathbf{1}.A \vdash c[b] : C[b]$, or a proof that $A \implies C$. This suggests that substituting corresponds logically to a "cut," and b to a proof that $A \implies B$.

Returning to the general case, contexts are lists of hypotheses, and a substitution $\Delta \vdash \gamma : \Gamma$ states that we can prove all the hypotheses of Γ using the hypotheses of Δ. Thus anything that is true under the hypotheses Γ is also true under the hypotheses Δ—hence the contravariance of the substitution operation.

More concretely, the idea is that a substitution $\Delta \vdash \gamma : \mathbf{1}.A_1 \dots A_n$ is an n-tuple of terms $a_1, \dots, a_n$ of types $A_1, \dots, A_n$, all in context Δ, and applying the substitution γ has the effect of substituting a_1 for the first variable, a_2 for the second variable, …and a_n for the last variable. The final subtlety is that each type A_i is in general dependent on all the previous A_j for $j < i$, so the type of a_2 is not just A_2 but "$A_2[a_1/x_1]$," so to speak, all the way through "$a_n : A_n[a_1/x_1, \dots, a_{n-1}/x_{n-1}]$."

If all of this sounds very complicated, well…at any rate, the remaining rules governing substitution define such n-tuples in two cases, 0 and $n+1$. The nullary case is fairly simple: any substitution $\Gamma \vdash \delta : \mathbf{1}$ into the empty context (a length-zero list of types) is necessarily the empty tuple $\langle\rangle$, which we spell !.

$$\frac{\vdash \Gamma\,\mathsf{cx}}{\Gamma \vdash ! : \mathbf{1}} \qquad\qquad \frac{\Gamma \vdash \delta : \mathbf{1}}{\Gamma \vdash ! = \delta : \mathbf{1}}$$

[6] This mysterious name is chosen to pair well with the name $\mathbf{p}$ that we gave weakening; $\mathbf{q}$ can thus be pronounced as either "variable" or "qariable".

These rules state that $\mathbf{1}$ is a terminal object in the category of contexts, a perspective which inspires the notations $\mathbf{1}$ and !.

Substitution extension The other case concerns substitutions $\Delta \vdash - : \Gamma.A$ into a context extension. Recall that $\Gamma.A$ is an $(n+1)$-tuple of types when Γ is an n-tuple of types, and suppose that $\Delta \vdash \gamma : \Gamma$, which is to say that γ is an n-tuple of terms (in context Δ) whose types are those in Γ. To extend this n-tuple to an $(n+1)$-tuple of terms whose types are those in $\Gamma.A$, we simply adjoin one more term a in context Δ with type $A[\gamma]$, where this substitution plugs the n previously-given terms into the dependencies of A.

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Delta \vdash a : A[\gamma]}{\Delta \vdash \gamma.a : \Gamma.A}$$

The final three rules are equations governing this substitution former:

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Delta \vdash a : A[\gamma]}{\Delta \vdash \mathbf{p} \circ (\gamma.a) = \gamma : \Gamma}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Delta \vdash a : A[\gamma]}{\Delta \vdash \mathbf{q}[\gamma.a] = a : A[\gamma]} \qquad \frac{\Gamma \vdash A\,\mathsf{type} \qquad \Delta \vdash \gamma : \Gamma.A}{\Delta \vdash \gamma = (\mathbf{p} \circ \gamma).\mathbf{q}[\gamma] : \Gamma.A}$$

Imagining for the moment that $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and $\gamma = [a_1/x_1, \ldots, a_n/x_n]$, the second rule states that $x_n[a_1/x_1, \ldots, a_n/x_n] = a_n$, in other words, that substituting into the last variable x_n replaces that variable by the last term a_n. The first rule states in essence that substituting into a type/term that does not mention (is weakened by) x_n is the same as dropping the last term a_n/x_n from the substitution, i.e., $[a_1/x_1, \ldots, a_{n-1}/x_{n-1}]$.

Finally, the third rule states that every substitution γ into the context $\Gamma.A$ is of the form $\gamma_0.a$, where a is determined by the behavior of γ on the last variable, and γ_0 is determined by the behavior of γ on the first n variables. (See Lemma 2.3.7.)

All of the rules in this section determine a category (of contexts and substitutions) with extra structure, known collectively as a *category with families* [Dyb96]. We will refer to any system that extends this collection of rules as a *Martin-Löf type theory*.

We conclude with a series of basic facts and constructions regarding substitutions in Martin-Löf type theory.

Exercise 2.2. Attempt to prove the following four lemmas before reading their proofs.

Lemma 2.3.4. *Substitutions* $\Gamma \vdash \gamma : \Gamma.A$ *satisfying* $\mathbf{p} \circ \gamma = \mathbf{id}$ *are in bijection with terms* $\Gamma \vdash a : A$.

Proof. Any substitution γ into $\Gamma.A$ is of the form $(\mathbf{p} \circ \gamma).\mathbf{q}[\gamma]$, which by our hypothesis is equal to $\mathbf{id}.\mathbf{q}[\gamma]$. We can apply this substitution to a variable, obtaining the term $\Gamma \vdash \mathbf{q}[\mathbf{id}.\mathbf{q}[\gamma]] = \mathbf{q}[\gamma] : A[\mathbf{id}]$ as required. Conversely, any term $\Gamma \vdash a : A$ determines a substitution $\Gamma \vdash \mathbf{id}.a : \Gamma.A$ that satisfies $\mathbf{p} \circ (\mathbf{id}.a) = \mathbf{id}$. One round-trip follows from the previously noted equation, and the other from $\mathbf{q}[\mathbf{id}.a] = a$. □

Lemma 2.3.5. *If* $\Delta \vdash \gamma : \Gamma$, $\Delta \vdash a : A[\gamma]$, *and* $\Xi \vdash \delta : \Delta$, *then* $(\gamma.a) \circ \delta = (\gamma \circ \delta).a[\delta]$.

Proof. To show

$$\frac{\Xi \vdash \delta : \Delta \qquad \Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Delta \vdash a : A[\gamma]}{\Delta \vdash (\gamma.a) \circ \delta = (\gamma \circ \delta).a[\delta] : \Gamma.A} \Rightarrow$$

we calculate as follows:

$$\begin{aligned}(\gamma.a) \circ \delta &= (\mathbf{p} \circ (\gamma.a) \circ \delta).(\mathbf{q}[(\gamma.a) \circ \delta]) \\ &= (\gamma \circ \delta).(\mathbf{q}[\gamma.a][\delta]) \\ &= (\gamma \circ \delta).(a[\delta])\end{aligned}$$

□

Lemma 2.3.6. *Given* $\Delta \vdash \gamma : \Gamma$ *and* $\Gamma \vdash A\,\mathsf{type}$, *there is a substitution* $\gamma.A$, *satisfying* $\Delta.A[\gamma] \vdash \gamma.A : \Gamma.A$.

Proof. We define $\gamma.A := (\gamma \circ \mathbf{p}).\mathbf{q}$, i.e., the extension of the substitution $\Delta.A[\gamma] \vdash \gamma \circ \mathbf{p} : \Gamma$ by the variable $\Delta.A[\gamma] \vdash \mathbf{q} : A[\gamma \circ \mathbf{p}]$. □

Lemma 2.3.7. *Suppose that* $\Gamma \vdash A\,\mathsf{type}$ *and* $\vdash \Delta\,\mathsf{cx}$. *Substitutions* $\Delta \vdash \gamma : \Gamma.A$ *are in bijection with pairs of a substitution* $\Delta \vdash \gamma_0 : \Gamma$ *and a term* $\Delta \vdash a : A[\gamma_0]$.

Proof. In the forward direction, we send $\Delta \vdash \gamma : \Gamma.A$ to the pair of $\mathbf{p} \circ \gamma$ and $\mathbf{q}[\gamma]$; in the reverse direction, we send pairs of γ_0 and a to the substitution $\gamma_0.a$. One round-trip follows from $\gamma = (\mathbf{p} \circ \gamma).\mathbf{q}[\gamma]$ and the other from $\mathbf{p} \circ (\gamma_0.a) = \gamma_0$ and $\mathbf{q}[\gamma_0.a] = a$. □

2.4 *Internalizing judgmental structure:* **Π**, **Σ**, **Eq**, **Unit**

With the basic structure of dependent type theory finally out of the way, we are prepared to define standard type and term formers, starting with the best-behaved connectives: dependent products, dependent sums, extensional equality, and the unit type. Unlike inductive types (Section 2.5), each of these connectives can be described concisely as internalizing judgmental structure of some kind.

2.4.1 *Dependent products*

We start with dependent function types, also known as *dependent products* or **Π**-*types*. The formation rule is as in Section 2.2, but without variable names:[7]

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type}}{\Gamma \vdash \mathbf{\Pi}(A,B)\,\mathsf{type}}$$

Remark 2.4.1. The **Π** notation and terminology is inspired by this type corresponding semantically to a set-indexed product of sets $\prod_{a\in A} B_a$. Indexed products generalize ordinary products in the sense that $\prod_{a\in\{1,2\}} B_a \cong B_1 \times B_2$. ◇

Remarkably, the substitution calculus ensures that these rules are almost indistinguishable from the introduction and elimination rules of simple function types in Section 2.1, with some minor additional bookkeeping to move types to the appropriate contexts:

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash b : B}{\Gamma \vdash \lambda(b) : \mathbf{\Pi}(A,B)} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash f : \mathbf{\Pi}(A,B)}{\Gamma \vdash \mathbf{app}(f,a) : B[\mathbf{id}.a]}$$

There continue to be a few notational shifts: λs no longer come with variable names, and we write $\mathbf{app}(f,a)$ rather than $f\ a$ just to emphasize that function application is a term constructor. The reader should convince themselves that in the final rule, $\Gamma \vdash B[\mathbf{id}.a]$ type; the substitution $\mathbf{id}.a$ instantiates the last variable in B with the term a, leaving the rest of the context unchanged.

Next we must specify equations not only on the introduction and elimination forms, but on the type former itself. There are two groups of equations we must impose; the first group explains how substitutions act on all three of these operations:

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type}}{\Delta \vdash \mathbf{\Pi}(A,B)[\gamma] = \mathbf{\Pi}(A[\gamma], B[\gamma.A])\,\mathsf{type}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash b : B}{\Delta \vdash \lambda(b)[\gamma] = \lambda(b[\gamma.A]) : \mathbf{\Pi}(A,B)[\gamma]}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash f : \mathbf{\Pi}(A,B)}{\Delta \vdash \mathbf{app}(f,a)[\gamma] = \mathbf{app}(f[\gamma], a[\gamma]) : B[\gamma.a[\gamma]]}$$

Roughly speaking, these three rules state that substitutions commute past each type and term former, but B and b are well-formed in a larger context $(\Gamma.A)$ than the

[7] We have switched our notation from $(x : A) \to B$ because it is awkward without named variables.

surrounding term (Γ), requiring us to "shift" the substitution so that it leaves the bound variable of type A unchanged while continuing to act on all the free variables in Γ. (The "shifted" substitution $\gamma.A$ in these rules is the derived form defined in Lemma 2.3.6.)

Once again we should pause and convince ourselves that these rules are meta-well-typed. Echoing the phenomenon we saw in Section 2.3 with $\Gamma \vdash a[\mathbf{id}] : A$, we need to use the substitution rule for $\mathbf{\Pi}(A, B)[\gamma]$ to see that the right-hand side of the substitution rules for $\lambda(b)[\gamma]$ and $\mathbf{app}(f, a)[\gamma]$ are well-typed.

Exercise 2.3. Check that the substitution rule for $\mathbf{app}(f, a)[\gamma]$ is meta-well-typed; in particular, show that both $\mathbf{app}(f, a)[\gamma]$ and $\mathbf{app}(f[\gamma], a[\gamma])$ have the type $B[\gamma.a[\gamma]]$.

This pattern will continue: every time we introduce a new type or term former θ, we will add an equation $\theta(a_1, \ldots, a_n)[\gamma] = \theta(a_1[\gamma_1], \ldots, a_n[\gamma_n])$ stating that substitutions push past θ, adjusted as necessary in each argument. These rules are quite mechanical and can even be automatically derived in some frameworks, but they are at the heart of type theory itself. From a logical perspective, they ensure that quantifier instantiation is uniform. From a mathematical perspective, as we will see in Section 2.4.2, they assert the naturality of type-theoretic constructions. And from an implementation perspective, these rules can be assembled into a substitution algorithm, ensuring that substitutions can be computed automatically by proof assistants.

Remark 2.4.2. The difference between this approach to substitution and the one outlined in Section 2.1 is one of *derivability* vs *admissibility*. In the simply-typed setting, the fact that all terms enjoy substitution is not part of the system but rather must be proven (and even constructed in the first place) by induction over the structure of terms, and so adding new constructs to the theory may cause substitution to fail.

In the substitution calculus, we assert that all types and terms enjoy substitution as basic rules of the theory, and later add equations specifying how substitution computes; thus any extension of the theory is guaranteed to enjoy substitution. Because substitution is a crucial aspect of dependent type theory, we find this latter approach more ergonomic. ◇

The second group of equations is the β- and η-rules introduced in Section 2.1, completing our presentation of dependent product types.

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash b : B}{\Gamma \vdash \mathbf{app}(\lambda(b), a) = b[\mathbf{id}.a] : B[\mathbf{id}.a]}$$

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash f : \mathbf{\Pi}(A, B)}{\Gamma \vdash f = \lambda(\mathbf{app}(f[\mathbf{p}], \mathbf{q})) : \mathbf{\Pi}(A, B)}$$

Exercise 2.4. Carefully explain why the η-rule above is meta-well-typed, in particular why $\lambda(\mathbf{app}(f[\mathbf{p}], \mathbf{q}))$ has the right type. Explicitly point out all the other rules and equations (e.g., $\mathbf{\Pi}$-introduction, $\mathbf{\Pi}$-elimination, weakening) to which you refer.

Exercise 2.5. Show that using $\mathbf{\Pi}$-types we can define a non-dependent function type whose formation rule states that if $\Gamma \vdash A\,\mathsf{type}$ and $\Gamma \vdash B\,\mathsf{type}$ then $\Gamma \vdash A \to B\,\mathsf{type}$. Then define the introduction and elimination rules from Section 2.1 for this encoding, and check that the β- and η-rules from Section 2.1 hold. (Hint: it is incorrect to define $A \to B :\equiv \mathbf{\Pi}(A, B)$.)

Exercise 2.6. As discussed in Section 2.3, two contexts that are not syntactically identical may nevertheless be equal. Give an example.

2.4.2 Dependent products internalize hypothetical judgments

With one type constructor, two term constructors, and five equations, it is natural to wonder whether we have written "enough" or "the correct" rules to specify $\mathbf{\Pi}$-types. One may also wonder whether there is an easier way. We now introduce a methodology for making sense of this collection of rules, and show how we can use this methodology to more efficiently define the later connectives. In short, we will view connectives as *internalizations of judgmental structure*, and $\Gamma \vdash - : \mathbf{\Pi}(A, B)$ in particular as an internalization of the hypothetical judgment $\Gamma.A \vdash - : B$.

Remark 2.4.3. In this book we limit ourselves to a semi-informal discussion of this perspective, which can be made fully precise with the language of category theory. For instance, using the framework of natural models, Awodey [Awo18] shows that the rules above exactly capture that $\mathbf{\Pi}$-types classify the hypothetical judgment in a precise sense. ◇

Analyzing context extension To warm up, let us begin by recalling Lemma 2.3.7, which establishes the following bijection of sets for every Δ, Γ, and A:

$$\{\gamma \mid \Delta \vdash \gamma : \Gamma.A\} \cong \{(\gamma_0, a) \mid \Delta \vdash \gamma_0 : \Gamma \wedge \Delta \vdash a : A[\gamma_0]\}$$

Using Notation 2.3.2 we equivalently write:

$$\iota_{\Delta,\Gamma,A} : \mathsf{Sb}(\Delta, \Gamma.A) \cong \textstyle\sum_{\gamma \in \mathsf{Sb}(\Delta,\Gamma)} \mathsf{Tm}(\Delta, A[\gamma])$$

where $\sum_{a \in A} B_a$ is our notation for the set-indexed coproduct of sets $\coprod_{a \in A} B_a$.

As stated, the bijections $\iota_{\Delta,\Gamma,A}$ and $\iota_{\Delta',\Gamma',A'}$ may be totally unrelated, but it turns out that this collection of bijections is actually *natural* (or "parametric") in Δ in the sense

that the behavior of $\iota_{\Delta_0,\Gamma,A}$ and $\iota_{\Delta_1,\Gamma,A}$ are correlated when we have a substitution from Δ_0 to Δ_1.

Because these bijections have different types, to make this idea precise we must find a way to relate their differing domains $\mathsf{Sb}(\Delta_0, \Gamma.A)$ and $\mathsf{Sb}(\Delta_1, \Gamma.A)$ with one another, as well as their codomains $\sum_{\gamma \in \mathsf{Sb}(\Delta_0,\Gamma)} \mathsf{Tm}(\Delta_0, A[\gamma])$ and $\sum_{\gamma \in \mathsf{Sb}(\Delta_1,\Gamma)} \mathsf{Tm}(\Delta_1, A[\gamma])$.

We have already seen the former in Notation 2.3.3: every substitution $\Delta_0 \vdash \delta : \Delta_1$ induces a function $\delta^* : \mathsf{Sb}(\Delta_1, \Gamma.A) \to \mathsf{Sb}(\Delta_0, \Gamma.A)$. We leave the latter as an exercise:

Exercise 2.7. Given $\Delta_0 \vdash \delta : \Delta_1$, use δ^* (Notation 2.3.3) to define the function:

$$\textstyle\sum_{\delta^*} \delta^* : \sum_{\gamma \in \mathsf{Sb}(\Delta_1,\Gamma)} \mathsf{Tm}(\Delta_1, A[\gamma]) \to \sum_{\gamma \in \mathsf{Sb}(\Delta_0,\Gamma)} \mathsf{Tm}(\Delta_0, A[\gamma])$$

With these functions in hand we can now explain precisely what we mean by the naturality of $\iota_{-,\Gamma,A}$. Fix a substitution $\Delta_0 \vdash \delta : \Delta_1$. We have two different ways of turning a substitution $\Delta_1 \vdash \gamma : \Gamma.A$ into an element of $\sum_{\gamma_0 \in \mathsf{Sb}(\Delta_0,\Gamma)} \mathsf{Tm}(\Delta_0, A[\gamma_0])$, depicted by the "right then down" and "down then right" paths in the diagram below:

$$\begin{array}{ccc}
\mathsf{Sb}(\Delta_1, \Gamma.A) & \xrightarrow{\iota_{\Delta_1,\Gamma,A}} & \sum_{\gamma \in \mathsf{Sb}(\Delta_1,\Gamma)} \mathsf{Tm}(\Delta_1, A[\gamma]) \\
\Big\downarrow{\scriptstyle \delta^*} & & \Big\downarrow{\scriptstyle \sum_{\delta^*} \delta^*} \\
\mathsf{Sb}(\Delta_0, \Gamma.A) & \xrightarrow[\iota_{\Delta_0,\Gamma,A}]{} & \sum_{\gamma \in \mathsf{Sb}(\Delta_0,\Gamma)} \mathsf{Tm}(\Delta_0, A[\gamma])
\end{array}$$

Going "right then down" we obtain

$$\begin{array}{ccc}
\gamma & \longmapsto & \iota_{\Delta_1,\Gamma,A}(\gamma) \\
 & & \Big\downarrow \\
 & & (\textstyle\sum_{\delta^*} \delta^*)(\iota_{\Delta_1,\Gamma,A}(\gamma))
\end{array}$$

and going "down then right" we obtain $\gamma \mapsto \gamma \circ \delta \mapsto \iota_{\Delta_0,\Gamma,A}(\gamma \circ \delta)$.

We say that the family of isomorphisms $\Delta \mapsto \iota_{\Delta,\Gamma,A}$ is natural when these two paths always yield the same result, i.e., when $(\sum_{\delta^*} \delta^*)(\iota_{\Delta_1,\Gamma,A}(\gamma)) = \iota_{\Delta_0,\Gamma,A}(\gamma \circ \delta)$ for every $\Delta_0 \vdash \delta : \Delta_1$ and γ. In other words, $\iota_{\Delta_0,\Gamma,A}$ and $\iota_{\Delta_1,\Gamma,A}$ "do the same thing" as soon as you correct the mismatch in their types by pre- and post-composing the appropriate maps.

Exercise 2.8. Prove that ι is natural, i.e., that the following maps are equal:

$$\textstyle\sum_{\delta^*} \delta^* \circ \iota_{\Delta_1,\Gamma,A} = \iota_{\Delta_0,\Gamma,A} \circ \delta^* : \mathsf{Sb}(\Delta_1, \Gamma.A) \to \sum_{\gamma \in \mathsf{Sb}(\Delta_0,\Gamma)} \mathsf{Tm}(\Delta_0, A[\gamma])$$

The terminology of "natural" comes from category theory, where $\iota_{-,\Gamma,A}$ is known as a natural isomorphism, but we will prove and use naturality conditions without referring to the general concept. One useful consequence of naturality is the following:

Exercise 2.9. *Without* unfolding the definition of ι, show that the naturality of ι and the fact that $\iota_{\Delta,\Gamma,A}$ and $\iota^{-1}_{\Delta,\Gamma,A}$ are inverses together imply that ι^{-1} is natural, i.e., that

$$\iota^{-1}_{\Delta_0,\Gamma,A} \circ \textstyle\sum_{\delta^*} \delta^* = \delta^* \circ \iota^{-1}_{\Delta_1,\Gamma,A} : \sum_{\gamma \in \mathsf{Sb}(\Delta_1,\Gamma)} \mathsf{Tm}(\Delta_1, A[\gamma]) \to \mathsf{Sb}(\Delta_0, \Gamma.A)$$

Exercise 2.10. For categorically-minded readers: argue that ι is a natural isomorphism in the standard sense, by rephrasing Exercises 2.7 and 2.8 in terms of categories and functors.

Rather than defining context extension by the collection of rules in Section 2.3 and then characterizing it in terms of ι after the fact, we can actually define it directly as "a context $\Gamma.A$ for which $\mathsf{Sb}(-,\Gamma.A)$ is naturally isomorphic to $\sum_{\gamma \in \mathsf{Sb}(-,\Gamma)} \mathsf{Tm}(-, A[\gamma])$," which unfolds to all of the relevant rules.

In addition to its brevity, the true advantage of such characterizations is that they are less likely to "miss" some important aspect of the definition. Zooming out, this definition states that substitutions into $\Gamma.A$ are dependent pairs of a substitution γ into Γ and a term in $A[\gamma]$, which is exactly the informal description we started with in Section 2.3.

With that in mind, our program for justifying the rules of type theory is as follows:

Slogan 2.4.4. *A connective in type theory is given by (1) a natural type-forming operation and (2) a natural isomorphism relating that type's terms to judgmentally-determined structure.*

We must unfortunately remain vague here about the meaning of "judgmentally-determined structure," but it refers to sets constructed from the sorts $\mathsf{Sb}(\Delta,\Gamma)$, $\mathsf{Ty}(\Gamma)$, and $\mathsf{Tm}(\Gamma, A)$ using natural operations such as dependent products and dependent sums—operations that are implicit in the meaning of inference rules. To make this more precise requires a formal treatment of the algebra of judgments via *logical frameworks*.

In addition, although this slogan will make quick work of the remainder of Section 2.4, we will need to revise it in Sections 2.5 and 2.6.

Π-types The rules in Section 2.4.1 precisely capture the existence of an operation

$$\mathbf{\Pi}_\Gamma : (\textstyle\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)) \to \mathsf{Ty}(\Gamma)$$

natural in Γ (that is, one which commutes with substitution) along with the following family of isomorphisms also natural in Γ:

$$\iota_{\Gamma,A,B} : \mathsf{Tm}(\Gamma, \mathbf{\Pi}(A,B)) \cong \mathsf{Tm}(\Gamma.A, B)$$

The first point expresses the formation rule and $\mathbf{\Pi}(A,B)[\gamma] = \mathbf{\Pi}(A[\gamma], B[\gamma.A])$. We focus on the second point, which characterizes the remaining rules in Section 2.4.1.

The reverse map $\iota^{-1}_{\Gamma,A,B} : \mathsf{Tm}(\Gamma.A, B) \to \mathsf{Tm}(\Gamma, \mathbf{\Pi}(A,B))$ is the introduction rule, which sends terms $\Gamma.A \vdash b : B$ to $\lambda(b)$. The forward map is slightly more involved, but we can guess that it should correspond to elimination. In fact it is *application to a fresh variable*, or a combination of weakening and application—given $\Gamma \vdash f : \mathbf{\Pi}(A,B)$, we weaken to $\Gamma.A \vdash f[\mathbf{p}] : \mathbf{\Pi}(A,B)[\mathbf{p}]$ and then apply to $\mathbf{q}$, obtaining $\Gamma.A \vdash \mathbf{app}(f[\mathbf{p}], \mathbf{q}) : B$.

To complete this natural isomorphism we must check that it is an isomorphism, and that it is natural. We begin with the isomorphism: for all $\vdash \Gamma\,\mathsf{cx}$, $\Gamma \vdash A\,\mathsf{type}$, and $\Gamma.A \vdash B\,\mathsf{type}$,

$$\iota^{-1}_{\Gamma,A,B}(\iota_{\Gamma,A,B}(f)) = f$$
$$\iota_{\Gamma,A,B}(\iota^{-1}_{\Gamma,A,B}(b)) = b$$

Unfolding definitions, we see that this isomorphism boils down essentially to β and η.

$$\begin{aligned}
&\iota^{-1}_{\Gamma,A,B}(\iota_{\Gamma,A,B}(f)) \\
&= \lambda(\mathbf{app}(f[\mathbf{p}], \mathbf{q})) \\
&= f && \text{by the } \eta \text{ rule}
\end{aligned}$$

$$\begin{aligned}
&\iota_{\Gamma,A,B}(\iota^{-1}_{\Gamma,A,B}(b)) \\
&= \mathbf{app}(\lambda(b)[\mathbf{p}], \mathbf{q}) \\
&= \mathbf{app}(\lambda(b[\mathbf{p}.A]), \mathbf{q}) && \lambda(-) \text{ commutes with substitution} \\
&= b[\mathbf{p}.A \circ \mathbf{id}.\mathbf{q}] && \text{by the } \beta \text{ rule} \\
&= b[\mathbf{p}.\mathbf{q}] && \text{by Exercise 2.11 below} \\
&= b[\mathbf{id}] \\
&= b
\end{aligned}$$

Exercise 2.11. Using the definition of $\mathbf{p}.A$ from Lemma 2.3.6, prove the substitution equality needed to complete the equational reasoning above.

As for the naturality of the isomorphisms ι, as before we must first explain how to relate the types of $\iota_{\Gamma,A,B}$ and $\iota_{\Delta,A[\gamma],B[\gamma.A]}$ given a substitution $\Delta \vdash \gamma : \Gamma$. In this case, the comparison functions are the following:

$$\gamma^* : \mathsf{Tm}(\Gamma, \mathbf{\Pi}(A,B)) \to \mathsf{Tm}(\Delta, \mathbf{\Pi}(A[\gamma], B[\gamma.A]))$$
$$\gamma.A^* : \mathsf{Tm}(\Gamma.A, B) \to \mathsf{Tm}(\Delta.A[\gamma], B[\gamma.A])$$

Naturality therefore states that "right then down" and "down then right" are equal in the following diagram. (By the reader's argument in Exercise 2.9, naturality of ι automatically implies the naturality of ι^{-1}.)

$$\begin{array}{ccc} \mathsf{Tm}(\Gamma, \mathbf{\Pi}(A,B)) & \xrightarrow{\iota_{\Gamma,A,B}} & \mathsf{Tm}(\Gamma.A, B) \\ \downarrow{\scriptstyle \gamma^*} & & \downarrow{\scriptstyle \gamma.A^*} \\ \mathsf{Tm}(\Delta, \mathbf{\Pi}(A[\gamma], B[\gamma.A])) & \xrightarrow[\iota_{\Delta,A[\gamma],B[\gamma.A]}]{} & \mathsf{Tm}(\Delta.A[\gamma], B[\gamma.A]) \end{array}$$

Fixing $\Gamma \vdash f : \mathbf{\Pi}(A,B)$, we show $\iota_{\Gamma,A,B}(f)[\gamma.A] = \iota_{\Delta,A[\gamma],B[\gamma.A]}(f[\gamma])$ by computing:

$$\begin{aligned} &\iota_{\Gamma,A,B}(f)[\gamma.A] \\ &= \mathbf{app}(f[\mathbf{p}], \mathbf{q})[\gamma.A] \\ &= \mathbf{app}(f[\mathbf{p}][\gamma.A], \mathbf{q}[\gamma.A]) && \mathbf{app}(-,-) \text{ commutes with substitution} \\ &= \mathbf{app}(f[\mathbf{p} \circ \gamma.A], \mathbf{q}) \\ &= \mathbf{app}(f[\gamma \circ \mathbf{p}], \mathbf{q}) \end{aligned}$$

$$\begin{aligned} &\iota_{\Delta,A[\gamma],B[\gamma.A]}(f[\gamma]) \\ &= \mathbf{app}(f[\gamma][\mathbf{p}], \mathbf{q}) \\ &= \mathbf{app}(f[\gamma \circ \mathbf{p}], \mathbf{q}) \end{aligned}$$

Thus all of the rules of $\mathbf{\Pi}$-types are summed up by a natural operation $\mathbf{\Pi}_\Gamma$ (formation and its substitution law) along with a natural isomorphism $\iota_{\Gamma,A,B} : \mathsf{Tm}(\Gamma, \mathbf{\Pi}(A,B)) \cong \mathsf{Tm}(\Gamma.A, B)$ where ι^{-1} and ι are introduction and elimination, the round-trips are β and η, and naturality is the remaining substitution laws.

An alternative eliminator There is a strange asymmetry in the two maps ι and ι^{-1} underlying our natural isomorphism: the latter is literally the introduction rule, but the former combines elimination with weakening and the variable rule. It turns out that there is an equivalent formulation of $\mathbf{\Pi}$-elimination more faithful to our current perspective:

$$\frac{\Gamma \vdash f : \mathbf{\Pi}(A,B)}{\Gamma.A \vdash \lambda^{-1}(f) : B} \Rightarrow$$

Such a presentation replaces the current $\mathbf{app}(-,-)$, β, and η rules with the above rule along with new versions of β and η stating simply that $\lambda^{-1}(\lambda(b)) = b$ and

$\lambda(\lambda^{-1}(f)) = f$ respectively. We recover ordinary function application via $\mathbf{app}(f, a) := \lambda^{-1}(f)[\mathbf{id}.a]$.

Although in practice our original formulation of function application is much more useful than anti-λ, the latter is more semantically natural. A variant of this argument is discussed by Gratzer et al. [Gra+22], because in the context of *modal type theories* one often encounters elimination forms akin to $\lambda^{-1}(-)$ and it can be far from obvious what the corresponding $\mathbf{app}(-,-)$ operation would be.

Exercise 2.12. Verify the claim that $\lambda^{-1}(-)$ and its β and η rules do in fact imply our original elimination, β, and η rules.

2.4.3 *Dependent sums*

We now present dependent pair types, also known as *dependent sums* or Σ*-types*. In a reversal of our discussion of **Π**-types, we will *begin* by defining dependent sums as an internalization of judgmental structure before unfolding this into inference rules.

The Σ type former behaves just like the **Π** type former: a natural family of types indexed by pairs of a type A and an A-indexed family of types B,

$$\Sigma_\Gamma : (\textstyle\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)) \to \mathsf{Ty}(\Gamma)$$

or in inference rule notation,

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type}}{\Gamma \vdash \Sigma(A,B)\,\mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type}}{\Delta \vdash \Sigma(A,B)[\gamma] = \Sigma(A[\gamma], B[\gamma.A])\,\mathsf{type}}$$

(Recall that we write $\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)$ for the indexed coproduct $\coprod_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)$.)

Where Σ-types and **Π**-types differ is in their elements. Whereas $\Gamma \vdash \mathbf{\Pi}(A, B)$ type internalizes terms with a free variable $\Gamma.A \vdash b : B$, the type $\Gamma \vdash \Sigma(A, B)$ type internalizes pairs of terms $\Gamma \vdash a : A$ and $\Gamma \vdash b : B[\mathbf{id}.a]$, naturally in Γ:

$$\iota_{\Gamma,A,B} : \mathsf{Tm}(\Gamma, \Sigma(A,B)) \cong \textstyle\sum_{a \in \mathsf{Tm}(\Gamma,A)} \mathsf{Tm}(\Gamma, B[\mathbf{id}.a])$$

Remarkably, the above line completes our definition of dependent sum types, but in the interest of the reader we will proceed to unfold this natural isomorphism into inference rules in three stages. First, we will unfold the maps $\iota_{\Gamma,A,B}$ and $\iota^{-1}_{\Gamma,A,B}$ into three term formers; second, we will unfold the two round-trip equations into a pair of equational rules; and finally, we will unfold the naturality condition into three more equational rules.

Exercise 2.13. Just as in Exercise 2.5, show that using Σ-types we can define a non-dependent pair type whose formation rule states that if $\Gamma \vdash A\,\mathsf{type}$ and $\Gamma \vdash B\,\mathsf{type}$ then $\Gamma \vdash A \times B\,\mathsf{type}$. Then define the introduction and elimination rules from Section 2.1 for this encoding, and check that the β- and η-rules from Section 2.1 hold.

Remark 2.4.5. There is an unfortunate terminological collision between simple types and dependent types: although $\mathbf{\Pi}$-types seem to generalize *simple functions*, they are called *dependent products*, and although Σ-types seem to generalize *simple products* because their elements are pairs, they are called *dependent sums*.

The reason is twofold: first, the elements of indexed coproducts (known to programmers as "tagged unions") are actually pairs ("pairs of a tag bit with data"), whereas the elements of indexed products ("n-ary pairs") are actually functions (sending n to the n-th projection). Secondly, *both concepts* generalize simple finite products: the binary product $B_1 \times B_2$ is both an indexed product $\prod_{a\in\{1,2\}} B_a$ and the indexed coproduct of a constant family $\sum_{_\in B_1} B_2$. ◇

To unpack the natural isomorphism, we note first that the forward direction $\iota_{\Gamma,A,B} : \mathsf{Tm}(\Gamma, \Sigma(A,B)) \to \sum_{a\in\mathsf{Tm}(\Gamma,A)} \mathsf{Tm}(\Gamma, B[\mathbf{id}.a])$ sends terms $\Gamma \vdash p : \Sigma(A,B)$ to (meta-)pairs of terms, so we can unfold this map into a pair of term formers with the same premises:

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash \mathbf{fst}(p) : A}$$

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash \mathbf{snd}(p) : B[\mathbf{id}.\mathbf{fst}(p)]}$$

The map $\iota^{-1}_{\Gamma,A,B} : \sum_{a\in\mathsf{Tm}(\Gamma,A)} \mathsf{Tm}(\Gamma, B[\mathbf{id}.a]) \to \mathsf{Tm}(\Gamma, \Sigma(A,B))$ sends a pair of terms to a single term of type $\Sigma(A,B)$, so we unfold it into one term former with two term premises:

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}.a]}{\Gamma \vdash \mathbf{pair}(a,b) : \Sigma(A,B)}$$

Unlike in our judgmental analysis of dependent products, the standard introduction and elimination forms of dependent sums correspond exactly to the maps ι^{-1} and ι, so

the two round-trip equations are exactly the standard β and η principles:

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}.a]}{\Gamma \vdash \mathbf{fst}(\mathbf{pair}(a,b)) = a : A \qquad \Gamma \vdash \mathbf{snd}(\mathbf{pair}(a,b)) = b : B[\mathbf{id}.a]}$$

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash p = \mathbf{pair}(\mathbf{fst}(p), \mathbf{snd}(p)) : \Sigma(A,B)}$$

It remains to unpack the naturality of ι, which as we have seen previously, encodes the fact that the term formers commute with substitution. The reader may be surprised to learn, however, that the substitution rule for $\mathbf{pair}(-,-)$ actually implies the substitution rules for $\mathbf{fst}(-)$ and $\mathbf{snd}(-)$ in the presence of β and η. (Categorically, this is the fact that naturality of ι^{-1} implies naturality of ι, as we saw in Exercise 2.9.) Given the rule

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}.a]}{\Delta \vdash \mathbf{pair}(a,b)[\gamma] = \mathbf{pair}(a[\gamma], b[\gamma]) : \Sigma(A,B)[\gamma]}$$

fix a substitution $\Delta \vdash \gamma : \Gamma$ and a term $\Gamma \vdash p : \Sigma(A,B)$. Then

$$\begin{aligned}
&\mathbf{fst}(p)[\gamma] \\
&= \mathbf{fst}(\mathbf{pair}(\mathbf{fst}(p)[\gamma], \mathbf{snd}(p)[\gamma])) && \text{by the } \beta \text{ rule} \\
&= \mathbf{fst}(\mathbf{pair}(\mathbf{fst}(p), \mathbf{snd}(p))[\gamma]) && \text{by the above rule} \\
&= \mathbf{fst}(p[\gamma]) && \text{by the } \eta \text{ rule}
\end{aligned}$$

and the calculation for $\mathbf{snd}(-)$ is identical. Nevertheless it is typical to include substitution rules for all three term formers: there is nothing wrong with equating terms that are already equal, and even in type theory, discretion can be the better part of valor.

Exercise 2.14. Check that the substitution rule for **pair** above is meta-well-typed, in particular the second component $b[\gamma]$. (Hint: use Lemma 2.3.5.)

2.4.4 Extensional equality

We now turn to the simplest form of propositional equality, known as *extensional equality* or **Eq**-*types*. As their name suggests, **Eq**-types internalize the term equality judgment. They are defined as follows, naturally in Γ:

$$\mathbf{Eq}_\Gamma : (\textstyle\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Tm}(\Gamma, A) \times \mathsf{Tm}(\Gamma, A)) \to \mathsf{Ty}(\Gamma)$$
$$\iota_{\Gamma,A,a,b} : \mathsf{Tm}(\Gamma, \mathbf{Eq}(A,a,b)) \cong \{\star \mid a = b\}$$

In other words, $\mathbf{Eq}(A, a, b)$ is a type when $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$, and has a unique inhabitant exactly when the judgment $\Gamma \vdash a = b : A$ holds (otherwise it is empty). The inference rules for extensional equality are as follows:

$$\frac{\Gamma \vdash a, b : A}{\Gamma \vdash \mathbf{Eq}(A, a, b)\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a, b : A}{\Delta \vdash \mathbf{Eq}(A, a, b)[\gamma] = \mathbf{Eq}(A[\gamma], a[\gamma], b[\gamma])\ \mathsf{type}}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{refl} : \mathbf{Eq}(A, a, a)} \qquad \frac{\Gamma \vdash a, b : A \qquad \Gamma \vdash p : \mathbf{Eq}(A, a, b)}{\Gamma \vdash a = b : A}$$

$$\frac{\Gamma \vdash a, b : A \qquad \Gamma \vdash p : \mathbf{Eq}(A, a, b)}{\Gamma \vdash p = \mathbf{refl} : \mathbf{Eq}(A, a, b)}$$

The penultimate rule is known as *equality reflection*, and it is somewhat unusual because it concludes an arbitrary term equality judgment from the existence of a term. This rule is quite strong in light of the facts that (1) judgmentally equal terms can be silently exchanged at any location in any judgment, (2) the equality proof $\Gamma \vdash p : \mathbf{Eq}(A, a, b)$ is not recorded in those exchanges, and (3) p could even be a variable, e.g., in context $\Gamma.\mathbf{Eq}(A, a, b)$.

Type theories with an extensional equality type are called *extensional*. The consequences of equality reflection will be the primary motivation behind the latter half of this book, but for now we simply note that these rules are a very natural axiomatization of an equality type as the internalization of equality.

Exercise 2.15. Explain how these inference rules correspond to our $\mathbf{Eq}_\Gamma$ and $\iota_{\Gamma,A,a,b}$ definition.

Exercise 2.16. Where are the substitution rules for term formers? (Hint: there are two equivalent answers, in terms of either the natural isomorphism or the inference rules.)

2.4.5 The unit type

We conclude our tour of the best-behaved connectives of type theory with the simplest connective of all: the unit type.

$$\mathbf{Unit}_\Gamma \in \mathsf{Ty}(\Gamma)$$
$$\iota_\Gamma : \mathsf{Tm}(\Gamma, \mathbf{Unit}) \cong \{\star\}$$

This unfolds to the following rules:

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{Unit}\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Unit}[\gamma] = \mathbf{Unit}\ \mathsf{type}}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{tt} : \mathbf{Unit}} \qquad \frac{\Gamma \vdash a : \mathbf{Unit}}{\Gamma \vdash a = \mathbf{tt} : \mathbf{Unit}}$$

Exercise 2.17. Where is the elimination principle? Where are the substitution rules for term formers? (Hint: what would these say in terms of the natural isomorphism?)

2.5 *Inductive types:* Void, Bool, +, Nat

We now turn our attention to *inductive types*, data types with induction principles. Unlike the type formers in Section 2.4, which are typically "hard coded" into type theories,[8] inductive types are usually specified as extensions to the theory (data type declarations) via inductive schemas [Dyb94; CP90], or in theoretical contexts, encoded as well-founded trees known as **W**-types [Mar82; Mar84b]. These schemas can be extended *ad infinitum* to account for increasingly complex forms of inductive definition, including indexed induction [Dyb94], mutual induction, induction-recursion [Dyb00], induction-induction [NS12], quotient induction-induction [KKA19], and so forth.

For simplicity we restrict our attention to four specific types—the empty type, booleans, coproducts, and natural numbers—that illustrate the basic issues that arise when specifying inductive types in type theory. Unfortunately, we will immediately need to refine Slogan 2.4.4.

2.5.1 *The empty type*

We begin with the empty type **Void**, a "type with no elements." Logically, this type corresponds to the false proposition, so there should be no way to construct an element of **Void** (a proof of false) except by deriving a contradiction from local hypotheses. The type former is straightforward: naturally in Γ, a constant $\mathbf{Void}_\Gamma \in \mathsf{Ty}(\Gamma)$, or

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{Void}\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Void}[\gamma] = \mathbf{Void}\ \mathsf{type}}$$

[8] This is an oversimplification: in practice, Σ and **Unit** are usually obtained as special cases of *dependent record types* [Pol02], n-ary Σ-types with named projections.

As for the elements of **Void**, an obvious guess is to say that the elements of the empty type at each context are the empty set, i.e., naturally in Γ,

$$\iota_\Gamma : \mathsf{Tm}(\Gamma, \mathbf{Void}) \cong \emptyset \tag{!?}$$

This cannot be right, however, because **Void** *does* have elements in some contexts—the variable rule alone forces $\mathbf{q} \in \mathsf{Tm}(\Gamma.\mathbf{Void}, \mathbf{Void})$, and other type formers can populate **Void** even further, e.g., $\mathbf{app}(\mathbf{q}, \mathbf{tt}) \in \mathsf{Tm}(\Gamma.\mathbf{\Pi}(\mathbf{Unit}, \mathbf{Void}), \mathbf{Void})$.

Interlude: mapping in, mapping out To see how to proceed, let us take a brief sojourn into set theory. There are several ways to define the product $A \times B$ of two sets, for example by constructing it as the set of ordered pairs $\{(a, b) \mid a \in A \wedge b \in B\}$ or even more explicitly as the set $\{\{\{a\}, \{a, b\}\} \mid a \in A \wedge b \in B\}$. However, in addition to these explicit constructions, it is also possible to *characterize* the set $A \times B$ up to isomorphism, as the set such that every function $X \to A \times B$ is determined by a pair of functions $X \to A$ and $X \to B$ and vice versa.

Similarly, we can characterize one-element sets $\mathbf{1}$ as those sets for which there is exactly one function $X \to \mathbf{1}$ for all sets X. In fact, both of these characterizations are set-theoretical analogues of Slogan 2.4.4, where X plays the role of the context Γ.

After some thought, we realize that the analogous characterization of the zero-element (empty) set $\mathbf{0}$ is significantly more awkward: there is exactly one function $X \to \mathbf{0}$ when X is empty, and no functions $X \to \mathbf{0}$ when X is non-empty. As it turns out, in this case it is more elegant to consider the functions *out* of $\mathbf{0}$ rather than the functions *into* it: a zero-element set $\mathbf{0}$ has exactly one function $\mathbf{0} \to X$ for all sets X.

Exercise 2.18. Suppose that Z is a set such that for all sets X there is exactly one function $Z \to X$. Show that Z is isomorphic to the empty set.

Void revisited Recall from Section 2.3 that terms correspond to "dependent functions from Γ to A." In Section 2.4 we considered only type formers T that are easily characterized in terms of the maps *into* that type former from an arbitrary context Γ: in each case we defined maps/terms $\mathsf{Tm}(\Gamma, T)$ as naturally isomorphic to the data of T's introduction rule.

To characterize the maps *out of* **Void** into an arbitrary type A, we cannot leave the context fully unconstrained; instead, we characterize the maps/terms $\mathsf{Tm}(\Gamma.\mathbf{Void}, A)$ for all $\vdash \Gamma\,\mathsf{cx}$ and $\Gamma.\mathbf{Void} \vdash A\,\mathsf{type}$, recalling that—by the rules for $\mathbf{\Pi}$-types—these are equivalently the dependent functions out of **Void** in context Γ, i.e., $\Gamma \vdash f : \mathbf{\Pi}(\mathbf{Void}, A)$.

Advanced Remark 2.5.1. Writing $\mathcal{C}$ for the category of contexts and substitutions, terms $\mathsf{Tm}(\Gamma, A)$ are "dependent morphisms" from Γ to A in the sense of being ordinary

morphisms $\Gamma \to \Gamma.A$ in the slice category $\mathcal{C}/\Gamma$ by Lemma 2.3.4. Thus, for *right adjoint* type operations G—those in Section 2.4—it is easy to describe $\mathsf{Tm}(\Gamma, G(A))$ directly.

For *left adjoint* type operations F, the situation is more fraught. Type theory is fundamentally "right-biased" because its judgments concern maps from arbitrary contexts *into* fixed types, but not vice versa. Thus to discuss dependent morphisms $F(X) \to A$ we must speak about elements of $\mathsf{Tm}(\Gamma.F(X), A)$, quantifying not only over the ambient context/slice Γ but also the type A into which we are mapping.

Confusingly, we encountered no issues defining Σ-types, despite dependent sum being the left adjoint to pullback. This is because Σ is also the right adjoint to the functor $\mathcal{C} \to \mathcal{C}^{\to}$ sending $A \mapsto \mathbf{id}_A$, and it is the latter perspective that we axiomatize. The left adjoint axiomatization makes an appearance in some systems, notably in programming languages with existential types, phrased as $\mathbf{let}\ (a, b) = p\ \mathsf{in}\ x$. ◇

Putting all these ideas together, we define **Void** as the type for which, naturally in Γ, there is exactly one dependent function from **Void** to A for any dependent type A:

$$\rho_{\Gamma,A} : \mathsf{Tm}(\Gamma.\mathbf{Void}, A) \cong \{\star\}$$

To sum up the difference between the incorrect definition $\mathsf{Tm}(\Gamma, \mathbf{Void}) \cong \emptyset$ and the correct one above, the former states that $\mathsf{Tm}(\Gamma, \mathbf{Void})$ is the smallest set (in the sense of mapping into all other sets), whereas the latter states that in any context, **Void** is the smallest *type*. More poetically, at the level of judgments we can see that **Void** is not always empty, but at the level of types, every type "believes" that **Void** is empty.

Unwinding $\rho_{\Gamma,A}$ into inference rules, we obtain:

$$\frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma.\mathbf{Void} \vdash A\ \mathsf{type}}{\Gamma.\mathbf{Void} \vdash \mathbf{absurd}' : A}\ ✎ \qquad \frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma.\mathbf{Void} \vdash a : A}{\Gamma.\mathbf{Void} \vdash \mathbf{absurd}' = a : A}\ ✎$$

We have marked these rules with ✎ to indicate that they are provisional. In practice, as previously discussed for λ^{-1}, it is awkward to use rules whose conclusions constrain the shape of their context; it is more standard to present an equivalent axiomatization $\mathbf{absurd}(b) :\equiv \mathbf{absurd}'[\mathbf{id}.b]$ that, like **app**, "builds in a cut":

$$\frac{\Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash A\ \mathsf{type}}{\Gamma \vdash \mathbf{absurd}(b) : A[\mathbf{id}.b]} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash A\ \mathsf{type}}{\Delta \vdash \mathbf{absurd}(b)[\gamma] = \mathbf{absurd}(b[\gamma]) : A[\gamma.b[\gamma]]}$$

$$\frac{\Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash a : A}{\Gamma \vdash \mathbf{absurd}(b) = a[\mathbf{id}.b] : A[\mathbf{id}.b]}\ ✎$$

The term $\mathbf{absurd}(-)$ is known as the *induction principle* for **Void**, in the sense that it allows users to prove a theorem for all terms of type **Void** by proving that it holds for each constructor of **Void**, of which there are none.

In light of our definition of **Void**, we update Slogan 2.4.4 as follows:

Slogan 2.5.2. *A connective in type theory is given by (1) a natural type-forming operation* Υ *and (2) one of the following:*

2.1. *a natural isomorphism relating* $\mathsf{Tm}(\Gamma, \Upsilon)$ *to judgmentally-determined structure, or*

2.2. *for all* $\Gamma.\Upsilon \vdash A\,\mathsf{type}$*, a natural isomorphism relating* $\mathsf{Tm}(\Gamma.\Upsilon, A)$ *to judgmentally-determined structure.*

The final rule for $\mathbf{absurd}(-)$, the η principle, implies a very strong equality principle for terms in an inconsistent context (Exercise 2.21) which we derive in the following sequence of exercises. For this reason, and because this rule is derivable in the presence of extensional equality (Section 2.5.5), we consider it provisional ✎ for the time being.

Lemma 2.5.3. *If* $\Gamma \vdash b_0 : \mathbf{Void}$ *and* $\Gamma \vdash b_1 : \mathbf{Void}$ *then* $\Gamma \vdash b_0 = b_1 : \mathbf{Void}$.

Proof. Suppose we are given $\Gamma \vdash b_0, b_1 : \mathbf{Void}$ and consider the terms $\Gamma.\mathbf{Void}.\mathbf{Void} \vdash \mathbf{q}, \mathbf{q}[\mathbf{p}] : \mathbf{Void}$. Since both of these terms are elements of the set $\mathsf{Tm}(\Gamma.\mathbf{Void}.\mathbf{Void}, \mathbf{Void})$ which has exactly one element, they must be equal. Consequently, we conclude the following:

$$\begin{aligned} b_1 &= \mathbf{q}[\mathbf{id}.b_0.b_1] \\ &= \mathbf{q}[\mathbf{p}][\mathbf{id}.b_0.b_1] \\ &= \mathbf{q}[\mathbf{id}.b_0] \\ &= b_0 \end{aligned}$$

□

Exercise 2.19. Fixing $\Delta \vdash \gamma : \Gamma$, prove that there is at most one substitution $\Delta \vdash \bar{\gamma} : \Gamma.\mathbf{Void}$ satisfying $\mathbf{p} \circ \bar{\gamma} = \gamma$.

Exercise 2.20. Let $\Gamma.\mathbf{Void} \vdash A\,\mathsf{type}$ and $\Gamma \vdash a : A[\mathbf{id}.b]$. Show that $\Gamma.\mathbf{Void} \vdash A[\mathbf{id}.b \circ \mathbf{p}] = A\,\mathsf{type}$, and therefore that $\Gamma.\mathbf{Void} \vdash a[\mathbf{p}] : A$.

Exercise 2.21. Derive the following rule, using the previous exercise and the η rule.

$$\frac{\Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash A\,\mathsf{type} \qquad \Gamma \vdash a : A[\mathbf{id}.b]}{\Gamma \vdash a = \mathbf{absurd}(b) : A[\mathbf{id}.b]} \Rightarrow$$

Exercise 2.22. We have included the rule $\Delta \vdash \mathbf{absurd}(b)[\gamma] = \mathbf{absurd}(b[\gamma]) : A[\gamma.b[\gamma]]$ but it is in fact derivable using the η rule. Prove this.

Exercise 2.23. In Remark 2.5.1, we claimed that Σ-types admit both a "mapping in" characterization (as in Section 2.4.3) and a "mapping out" characterization. Show that naturally in Γ, there is an isomorphism

$$\mathsf{Tm}(\Gamma.\Sigma(A, B), C) \cong \mathsf{Tm}(\Gamma.A.B, C[\mathbf{p}^2.\mathbf{pair}(\mathbf{q}[\mathbf{p}], \mathbf{q})])$$

2.5.2 *Booleans*

We turn now to the booleans **Bool**, a "type with two elements." Once again the type former is straightforward: $\mathbf{Bool}_\Gamma \in \mathsf{Ty}(\Gamma)$ naturally in Γ, or

$$\frac{}{\Gamma \vdash \mathbf{Bool}\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Bool}[\gamma] = \mathbf{Bool}\ \mathsf{type}}$$

It is also clear that we want two constructors of **Bool**, **true** and **false**, natural in Γ:

$$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{Bool}} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{Bool}}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{true} = \mathbf{true}[\gamma] : \mathbf{Bool}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{false} = \mathbf{false}[\gamma] : \mathbf{Bool}}$$

Keeping Slogan 2.5.2 in mind, there are two possible ways to complete our axiomatization of **Bool**. As with **Void** it is tempting but incorrect to define $\iota : \mathsf{Tm}(\Gamma, \mathbf{Bool}) \cong \{\star, \star'\}$; although the natural transformation ι^{-1} is equivalent to our rules for **true** and **false**, ι does not account for variables of type **Bool** or other indeterminate booleans that arise in non-empty contexts.[9] Thus we must instead characterize maps *out of* **Bool** by giving a family of sets naturally isomorphic to $\mathsf{Tm}(\Gamma.\mathbf{Bool}, A)$.

So, what should terms $\Gamma.\mathbf{Bool} \vdash a : A$ be? By substitution, such a term clearly determines a pair of terms $\Gamma \vdash a[\mathbf{id}.\mathbf{true}] : A[\mathbf{id}.\mathbf{true}]$ and $\Gamma \vdash a[\mathbf{id}.\mathbf{false}] : A[\mathbf{id}.\mathbf{false}]$. Conversely, if **true** and **false** are the "only" booleans, then such a pair of terms should uniquely determine elements of $\mathsf{Tm}(\Gamma.\mathbf{Bool}, A)$ in the sense that to map out of **Bool**, it suffices to explain what to do on **true** and on **false**.

To formalize this idea, let us write $((\mathbf{id}.\mathbf{true})^*, (\mathbf{id}.\mathbf{false})^*)$ for the function sending $a \in \mathsf{Tm}(\Gamma.\mathbf{Bool}, A)$ to the pair $(a[\mathbf{id}.\mathbf{true}], a[\mathbf{id}.\mathbf{false}])$. To complete our specification of **Bool** we ask for this map to be a natural isomorphism. In all, naturally in Γ we have:

$$\mathbf{Bool}_\Gamma \in \mathsf{Ty}(\Gamma)$$
$$\mathbf{true}_\Gamma, \mathbf{false}_\Gamma \in \mathsf{Tm}(\Gamma, \mathbf{Bool})$$
$$((\mathbf{id}.\mathbf{true})^*, (\mathbf{id}.\mathbf{false})^*) : \mathsf{Tm}(\Gamma.\mathbf{Bool}, A) \cong \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{true}]) \times \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{false}])$$

Remark 2.5.4. The forward map $((\mathbf{id}.\mathbf{true})^*, (\mathbf{id}.\mathbf{false})^*)$ is already natural in Γ so, as in Exercise 2.9, asking for this map to be a natural isomorphism is equivalent to merely asking for it to have an inverse for each Γ. ◇

[9] Even if variables x : **Bool** stand for one of **true** or **false**, x itself must be an indeterminate boolean equal to neither constructor; otherwise the identity $\lambda x.x$: **Bool** → **Bool** would be a constant function.

This definition is remarkable in several ways. For the first time we are asking not only for the existence of some natural isomorphism, but for a *particular map* to be a natural isomorphism; moreover, because this map is defined in terms of **true** and **false**, these must be asserted prior to the natural isomorphism itself. We update our slogan accordingly:

Slogan 2.5.5. *A connective in type theory is given by (1) a natural type-forming operation* Υ *and (2) one of the following:*

2.1. a natural isomorphism relating $\mathsf{Tm}(\Gamma, \Upsilon)$ *to judgmentally-determined structure, or*

2.2. a collection of natural term constructors for Υ *which, for all* $\Gamma.\Upsilon \vdash A\,\mathsf{type}$*, determine a natural isomorphism relating* $\mathsf{Tm}(\Gamma.\Upsilon, A)$ *to judgmentally-determined structure.*

In the case of **Void** there were no term constructors to specify, and because there is at most one (natural) isomorphism between anything and $\{\star\}$, it was unnecessary to specify the underlying map. In general, however, we emphasize that it is essential to specify the map; doing so ensures that when we define a function "by cases" on **true** and **false**, applying that function to **true** or **false** recovers the specified case and not something else. On the other hand, because we have specified the underlying map, it being an isomorphism is a *property* rather than additional structure: there is at most one possible inverse.

Zooming out, our definition of **Bool** has a similar effect to our definition of **Void** from Section 2.5.1: $\mathsf{Tm}(\Gamma, \mathbf{Bool})$ is *not* the set $\{\mathbf{true}, \mathbf{false}\}$ at the level of judgments, but every type "believes" that it is. This is the role of type-theoretic induction principles.

Advanced Remark 2.5.6. From the categorical perspective, option 2.2 in Slogan 2.5.5 asserts that the inclusion map of Υ's constructors into Υ's terms is *left orthogonal* to all types. Maps which are left orthogonal to a class of objects and whose codomain belongs to that class are known as *fibrant replacements*; we have thus defined $\mathsf{Tm}(-, \mathbf{Void})$ and $\mathsf{Tm}(-, \mathbf{Bool})$ as fibrant replacements of the constantly zero- and two-element presheaves. This perspective is crucial to early work in homotopy type theory [AW09] and the formulation of the intensional identity type in natural models [Awo18]. ◇

It remains to unfold our natural isomorphism into inference rules. There are no additional rules for the forward map, which is substitution by $\mathbf{id}.\mathbf{true}$ and $\mathbf{id}.\mathbf{false}$. As

the reader may have already guessed, the backward map is essentially[10] dependent **if**:

$$\frac{\Gamma.\mathbf{Bool} \vdash A\,\mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}] \qquad \Gamma \vdash b : \mathbf{Bool}}{\Gamma \vdash \mathbf{if}\,(a_t, a_f, b) : A[\mathbf{id}.b]}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma.\mathbf{Bool} \vdash A\,\mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}] \qquad \Gamma \vdash b : \mathbf{Bool}}{\Delta \vdash \mathbf{if}\,(a_t, a_f, b)[\gamma] = \mathbf{if}\,(a_t[\gamma], a_f[\gamma], b[\gamma]) : A[\gamma.b[\gamma]]}$$

The fact that **if** is an inverse to $((\mathbf{id}.\mathbf{true})^*, (\mathbf{id}.\mathbf{false})^*)$ expresses the β and η laws:

$$\frac{\Gamma.\mathbf{Bool} \vdash A\,\mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]}{\Gamma \vdash \mathbf{if}\,(a_t, a_f, \mathbf{true}) = a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash \mathbf{if}\,(a_t, a_f, \mathbf{false}) = a_f : A[\mathbf{id}.\mathbf{false}]}$$

$$\frac{\Gamma.\mathbf{Bool} \vdash A\,\mathsf{type} \qquad \Gamma.\mathbf{Bool} \vdash a : A \qquad \Gamma \vdash b : \mathbf{Bool}}{\Gamma \vdash \mathbf{if}\,(a[\mathbf{id}.\mathbf{true}], a[\mathbf{id}.\mathbf{false}], b) = a[\mathbf{id}.b] : A[\mathbf{id}.b]}$$ ✎

The β laws—the first two equations—are perhaps more familiar than the η law, which effectively asserts that any two terms dependent on **Bool** are equal if (and only if) they are equal on **true** and **false**. (The η rule is sometimes decomposed into a "local expansion" and a collection of "commuting conversions.") Although semantically justified, it is typical to omit judgmental η laws for all inductive types because they are not syntax-directed and thus challenging to implement, and because they are derivable in the presence of extensional equality (Section 2.5.5).

Exercise 2.24. As with **absurd**/**absurd′** in Section 2.5.1 and **app**/λ^{-1} in Section 2.4.2, there is an equivalent presentation of **if** that more directly mirrors the natural isomorphism. Show that the following rules for **if′** are equivalent to our rules for $\mathbf{if}\,(a_t, a_f, b)$:

$$\frac{\Gamma.\mathbf{Bool} \vdash A\,\mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]}{\Gamma.\mathbf{Bool} \vdash \mathbf{if'}\,(a_t, a_f) : A}$$

$$\frac{\Gamma.\mathbf{Bool} \vdash A\,\mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]}{\begin{array}{c}\Gamma \vdash \mathbf{if'}\,(a_t, a_f)[\mathbf{id}.\mathbf{true}] = a_t : A[\mathbf{id}.\mathbf{true}] \\ \Gamma \vdash \mathbf{if'}\,(a_t, a_f)[\mathbf{id}.\mathbf{false}] = a_f : A[\mathbf{id}.\mathbf{false}]\end{array}}$$

$$\frac{\Gamma.\mathbf{Bool} \vdash A\,\mathsf{type} \qquad \Gamma.\mathbf{Bool} \vdash a : A}{\Gamma.\mathbf{Bool} \vdash \mathbf{if'}\,(a[\mathbf{id}.\mathbf{true}], a[\mathbf{id}.\mathbf{false}]) = a : A}$$

Why can we omit substitution rules for **if′**, but not for **if**?

[10] The inverse directly lands in $\Gamma.\mathbf{Bool}$ and not Γ, but as with **absurd′** (Section 2.5.1) we adopt a more standard presentation in which all conclusions have a generic context; see Exercise 2.24.

2.5.3 *Coproducts*

Our next example is the coproduct type $A + B$, the "disjoint union of A and B." This inductive type former follows the same pattern as the booleans but introduces one important subtlety. Like **Π**-types, **Σ**-types, and **Eq**-types, the + type former takes parameters, in this case a pair of types in the same context:

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash B\,\mathsf{type}}{\Gamma \vdash A + B\,\mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash B\,\mathsf{type}}{\Delta \vdash (A + B)[\gamma] = A[\gamma] + B[\gamma]\,\mathsf{type}}$$

Like **Bool**, $A + B$ has two constructors; unlike **Bool**, its constructors are unary (rather than nullary) operations whose arguments have types A and B respectively:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash B\,\mathsf{type}}{\Gamma \vdash \mathbf{inl}(a) : A + B} \qquad \frac{\Gamma \vdash b : B \qquad \Gamma \vdash A\,\mathsf{type}}{\Gamma \vdash \mathbf{inr}(b) : A + B}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma \vdash B\,\mathsf{type}}{\Delta \vdash \mathbf{inl}(a)[\gamma] = \mathbf{inl}(a[\gamma]) : A[\gamma] + B[\gamma]}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash b : B \qquad \Gamma \vdash A\,\mathsf{type}}{\Delta \vdash \mathbf{inr}(b)[\gamma] = \mathbf{inr}(b[\gamma]) : A[\gamma] + B[\gamma]}$$

The names **inl** and **inr** are customary, abbreviating the *left injection* and *right injection* of the types A and B into the coproduct $A + B$. The type $A + B$ is a disjoint union in the sense that these injections are distinguished even in the case that $A = B$.

Following the pattern established with **Bool**, we assert that maps out of $A + B$ are uniquely determined by their behavior on its two constructors $\mathbf{inl}(-)$ and $\mathbf{inr}(-)$. In this case, because the $\mathbf{inl}(-)$ constructor has type "$A \to (A + B)$," the condition of "being determined by one's behavior on $\mathbf{inl}(a) : A + B$" is properly stated relative to a variable $a : A$ (and analogously with $\mathbf{inr}(b)$ and a variable $b : B$).

The mapping-out property for $A + B$ thus involves the two substitutions

$$(\mathbf{p}.\mathbf{inl}(\mathbf{q}))^* : \mathsf{Tm}(\Gamma.(A + B), C) \to \mathsf{Tm}(\Gamma.A, C[\mathbf{p}.\mathbf{inl}(\mathbf{q})])$$
$$(\mathbf{p}.\mathbf{inr}(\mathbf{q}))^* : \mathsf{Tm}(\Gamma.(A + B), C) \to \mathsf{Tm}(\Gamma.B, C[\mathbf{p}.\mathbf{inr}(\mathbf{q})])$$

the first of which sends $c \in \mathsf{Tm}(\Gamma.(A + B), C)$ to $c[\mathbf{p}.\mathbf{inl}(\mathbf{q})] \in \mathsf{Tm}(\Gamma.A, C[\mathbf{p}.\mathbf{inl}(\mathbf{q})])$, in essence precomposing the input map of type "$(A + B) \to C$" with the left injection "$A \to (A + B)$" (except that C depends on $A + B$, and everything in sight depends on Γ).

Other than these substitutions not landing in context Γ, the specification of $A + B$ mirrors that of **Bool**. Naturally in Γ, we have the formation and introduction rules

$$+_\Gamma : (\mathsf{Ty}(\Gamma) \times \mathsf{Ty}(\Gamma)) \to \mathsf{Ty}(\Gamma)$$
$$\mathbf{inl}_{\Gamma,A,B} : \mathsf{Tm}(\Gamma, A) \to \mathsf{Tm}(\Gamma, A + B)$$
$$\mathbf{inr}_{\Gamma,A,B} : \mathsf{Tm}(\Gamma, B) \to \mathsf{Tm}(\Gamma, A + B)$$

and we assert that for all $\Gamma.(A + B) \vdash C\,\mathsf{type}$, the following map is a natural isomorphism:

$$((\mathbf{p}.\mathbf{inl}(\mathbf{q}))^*, (\mathbf{p}.\mathbf{inr}(\mathbf{q}))^*) :$$
$$\mathsf{Tm}(\Gamma.(A + B), C) \cong \mathsf{Tm}(\Gamma.A, C[\mathbf{p}.\mathbf{inl}(\mathbf{q})]) \times \mathsf{Tm}(\Gamma.B, C[\mathbf{p}.\mathbf{inr}(\mathbf{q})])$$

Unfolding the reverse direction of this natural isomorphism and building in a cut, we obtain the following "**case** distinction" eliminator and substitution rule:

$$\frac{\begin{array}{ccc}\Gamma \vdash A\,\mathsf{type} & \Gamma \vdash B\,\mathsf{type} & \Gamma.(A + B) \vdash C\,\mathsf{type} \\ \Gamma.A \vdash c_l : C[\mathbf{p}.\mathbf{inl}(\mathbf{q})] & \Gamma.B \vdash c_r : C[\mathbf{p}.\mathbf{inr}(\mathbf{q})] & \Gamma \vdash p : A + B\end{array}}{\Gamma \vdash \mathbf{case}(c_l, c_r, p) : C[\mathbf{id}.p]}$$

$$\frac{\begin{array}{cccc}\Delta \vdash \gamma : \Gamma & \Gamma \vdash A\,\mathsf{type} & \Gamma \vdash B\,\mathsf{type} & \Gamma.(A + B) \vdash C\,\mathsf{type} \\ \Gamma.A \vdash c_l : C[\mathbf{p}.\mathbf{inl}(\mathbf{q})] & \Gamma.B \vdash c_r : C[\mathbf{p}.\mathbf{inr}(\mathbf{q})] & \Gamma \vdash p : A + B & \end{array}}{\Delta \vdash \mathbf{case}(c_l, c_r, p)[\gamma] = \mathbf{case}(c_l[\gamma.A], c_r[\gamma.B], p[\gamma]) : C[\gamma.p[\gamma]]}$$

with the two β laws:

$$\frac{\begin{array}{ccc}\Gamma \vdash a : A & \Gamma \vdash B\,\mathsf{type} & \\ \Gamma.(A + B) \vdash C\,\mathsf{type} & \Gamma.A \vdash c_l : C[\mathbf{p}.\mathbf{inl}(\mathbf{q})] & \Gamma.B \vdash c_r : C[\mathbf{p}.\mathbf{inr}(\mathbf{q})]\end{array}}{\Gamma \vdash \mathbf{case}(c_l, c_r, \mathbf{inl}(a)) = c_l[\mathbf{id}.a] : C[\mathbf{id}.\mathbf{inl}(a)]}$$

$$\frac{\begin{array}{ccc}\Gamma \vdash b : B & \Gamma \vdash A\,\mathsf{type} & \\ \Gamma.(A + B) \vdash C\,\mathsf{type} & \Gamma.A \vdash c_l : C[\mathbf{p}.\mathbf{inl}(\mathbf{q})] & \Gamma.B \vdash c_r : C[\mathbf{p}.\mathbf{inr}(\mathbf{q})]\end{array}}{\Gamma \vdash \mathbf{case}(c_l, c_r, \mathbf{inr}(b)) = c_r[\mathbf{id}.b] : C[\mathbf{id}.\mathbf{inr}(b)]}$$

and the often omitted η law:

$$\frac{\Gamma.(A + B) \vdash C\,\mathsf{type} \qquad \Gamma.(A + B) \vdash c : C \qquad \Gamma \vdash p : A + B}{\Gamma \vdash \mathbf{case}(c[\mathbf{p}.\mathbf{inl}(\mathbf{q})], c[\mathbf{p}.\mathbf{inr}(\mathbf{q})], p) = c[\mathbf{id}.p] : C[\mathbf{id}.p]}$$ ✎

Exercise 2.25. Using coproduct types, we could define **Bool** := **Unit** + **Unit**. Complete this process by defining **true** and **false** and showing that the canonical map $\mathsf{Tm}(\Gamma.\mathbf{Bool}, A) \to \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{true}]) \times \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{false}])$ is a natural isomorphism.

2.5.4 Natural numbers

Our final example of an inductive type is the type of natural numbers **Nat**, the "least type closed under **zero** : **Nat** and **suc**(−) : **Nat** → **Nat**." The natural numbers are conceptually similar to **Bool** and +, but the recursive nature of **suc**(−) complicates the situation significantly. The formation and introduction rules remain straightforward:

$$\frac{}{\Gamma \vdash \mathbf{Nat}\ \mathrm{type}} \qquad \frac{}{\Gamma \vdash \mathbf{zero} : \mathbf{Nat}} \qquad \frac{\Gamma \vdash n : \mathbf{Nat}}{\Gamma \vdash \mathbf{suc}(n) : \mathbf{Nat}}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Nat}[\gamma] = \mathbf{Nat}\ \mathrm{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{zero}[\gamma] = \mathbf{zero} : \mathbf{Nat}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash n : \mathbf{Nat}}{\Delta \vdash \mathbf{suc}(n)[\gamma] = \mathbf{suc}(n[\gamma]) : \mathbf{Nat}}$$

As with **Bool** and +, we might imagine asking for maps out of **Nat** to be determined by their behavior on **zero** and **suc**(−), i.e., for the substitutions

$$(\mathbf{id}.\mathbf{zero})^* : \mathsf{Tm}(\Gamma.\mathbf{Nat}, A) \to \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{zero}])$$
$$(\mathbf{p}.\mathbf{suc}(\mathbf{q}))^* : \mathsf{Tm}(\Gamma.\mathbf{Nat}, A) \to \mathsf{Tm}(\Gamma.\mathbf{Nat}, A[\mathbf{p}.\mathbf{suc}(\mathbf{q})])$$

to determine for every $\Gamma.\mathbf{Nat} \vdash A\ \mathrm{type}$ a natural isomorphism

$$((\mathbf{id}.\mathbf{zero})^*, (\mathbf{p}.\mathbf{suc}(\mathbf{q}))^*) : \mathsf{Tm}(\Gamma.\mathbf{Nat}, A) \cong \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{zero}]) \times \mathsf{Tm}(\Gamma.\mathbf{Nat}, A[\mathbf{p}.\mathbf{suc}(\mathbf{q})]) \qquad (!?)$$

This turns out not to be the correct definition, but first, note that the first substitution moves us from $\Gamma.\mathbf{Nat}$ to Γ because the **zero** constructor is nullary, whereas the second substitution moves us from $\Gamma.\mathbf{Nat}$ also to $\Gamma.\mathbf{Nat}$ because the **suc**(−) constructor has type "**Nat** → **Nat**"; if the argument of **suc**(−) was of type X rather than **Nat**, then the latter substitution would be $\Gamma.X \vdash \mathbf{p}.\mathbf{suc}(\mathbf{q}) : \Gamma.\mathbf{Nat}$.

But given that **suc**(−) *is* recursive—taking **Nat** to **Nat**—we now for the first time are defining a judgment by a natural isomorphism whose *right-hand* side *also* has the very same judgment we are trying to define, namely $\mathsf{Tm}(\Gamma.\mathbf{Nat}, \dots)$, i.e., terms in context $\Gamma.\mathbf{Nat}$. This natural isomorphism is therefore not so much a *definition* of its left-hand side as it is an *equation* that the left-hand side must satisfy—in principle, this equation may have many different solutions for $\mathsf{Tm}(\Gamma.\mathbf{Nat}, A)$, or no solutions at all.

Interlude: initial algebras This equation asserts in essence that the natural numbers are a set N satisfying the isomorphism $N \cong \{\star\} \sqcup N$,[11] where the reverse map equips N with a choice of "implementations" of $\mathbf{zero} \in N$ and $\mathbf{suc}(-) : N \to N$. The set of natural numbers $\mathbb{N}$ with $\mathbf{zero} \coloneqq 0$ and $\mathbf{suc}(n) \coloneqq n + 1$ are a solution, but there are infinitely many *other* solutions as well, such as $\mathbb{N} \cup \{\infty\}$ with $\mathbf{zero} \coloneqq 0$, $\mathbf{suc}(n) \coloneqq n + 1$, and $\mathbf{suc}(\infty) \coloneqq \infty$.

Nevertheless one might imagine that $(\mathbb{N}, 0, - + 1)$ is a distinguished solution in some way, and indeed it is the "least" set N with a point $z \in N$ and endofunction $s : N \to N$—here we are dropping the requirement of (z, s) being an isomorphism—in the sense that for any (N, z, s) there is a unique function $f : \mathbb{N} \to N$ with $f(0) = z$ and $f(n + 1) = s(f(n))$. Such triples (N, z, s) are known as *algebras* for the signature $N \mapsto \{\star\} \sqcup N$, structure-preserving functions between algebras are known as *algebra homomorphisms*, and algebras with the above minimality property are *initial algebras*.

The above definitions extend directly to dependent algebras and homomorphisms: given an ordinary algebra (N, z, s), a *displayed algebra over* (N, z, s) is a triple of an N-indexed family of sets $\{\tilde{N}_n\}_{n \in N}$, an element $\tilde{z} \in \tilde{N}_z$, and a function $\tilde{s} : (n : N) \to \tilde{N}_n \to \tilde{N}_{s(n)}$ [KKA19]. Given any displayed algebra $(\tilde{N}, \tilde{z}, \tilde{s})$ over the natural number algebra $(\mathbb{N}, 0, - + 1)$, there is once again a unique function $f : (n : \mathbb{N}) \to \tilde{N}_n$ with $f(0) = \tilde{z}$ and $f(n + 1) = \tilde{s}(n, f(n))$. The reader is likely familiar with the special case of displayed algebras over $\mathbb{N}$ valued in *propositions* rather than sets:

$$\forall P : \mathbb{N} \to \mathbf{Prop}.\ P(0) \implies (\forall n.P(n) \implies P(n + 1)) \implies \forall n.P(n)$$

Advanced Remark 2.5.7. The data of a displayed algebra over (N, z, s) is equivalent to the data of an algebra homomorphism into (N, z, s), where the forward direction of this equivalence sends the family $\{\tilde{N}_n\}_{n \in N}$ to the first projection $(\sum_{n \in N} \tilde{N}_n) \to N$. A displayed algebra over the natural number algebra is thus a homomorphism $\tilde{N} \to \mathbb{N}$; the initiality of $\mathbb{N}$ implies this map has a unique section homomorphism, which unfolds to the dependent universal property stated above. ◇

Remark 2.5.8. As discussed in Chapters 3 and 6 and especially Appendix B, algebras will play a central role in our understanding of the syntax and semantics of type theory: we will regard the rules of type theory as a (generalized form of) algebraic signature of which type theory itself is the initial algebra, and whose models are precisely algebra homomorphisms. We encourage the reader to return to this section to observe that the foregoing discussion anticipates the model theory of type theory itself. ◇

Natural numbers revisited Coming back to our specification of **Nat**, our formation and introduction rules axiomatize an algebra $(\mathbf{Nat}, \mathbf{zero}, \mathbf{suc}(-))$ for the signature

[11] Why? In algebraic notation and ignoring dependency, the equation states that $A^{\Gamma \times N} \cong A^{\Gamma} \times A^{\Gamma \times N}$, which simplifies to $(\Gamma \times N) \cong \Gamma \sqcup (\Gamma \times N)$ and thus $N \cong \{\star\} \sqcup N$.

$N \mapsto \{\star\} \sqcup N$, but our proposed +-style natural isomorphism does not imply that this algebra is initial. The solution is to simply axiomatize that any displayed algebra over $(\mathbf{Nat}, \mathbf{zero}, \mathbf{suc}(-))$ admits a unique displayed algebra homomorphism from $(\mathbf{Nat}, \mathbf{zero}, \mathbf{suc}(-))$.

Unwinding definitions, we ask that naturally in Γ, and for any $A \in \mathsf{Ty}(\Gamma.\mathbf{Nat})$, $a_z \in \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{zero}])$, and $a_s \in \mathsf{Tm}(\Gamma.\mathbf{Nat}.A, A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])])$, we have an isomorphism:

$$\rho_{\Gamma,A,a_z,a_s} : \{a \in \mathsf{Tm}(\Gamma.\mathbf{Nat}, A) \mid a_z = a[\mathbf{id}.\mathbf{zero}] \wedge a_s[\mathbf{p}.\mathbf{q}.a] = a[\mathbf{p}.\mathbf{suc}(\mathbf{q})]\} \cong \{\star\}$$

The type of a_s is easier to understand with named variables: it is a term of type $A(\mathbf{suc}(n))$ in context $\Gamma, n : \mathbf{Nat}, a : A(n)$.

Remark 2.5.9. This is the third time we have defined a connective in terms of a natural isomorphism with $\{\star\}$. In Section 2.4.5, we used such an isomorphism to assert that **Unit** has a unique element in every context; in Section 2.5.1, we asserted dually that every dependent type over **Void** admits a unique dependent function from **Void**. The present definition is analogous to the latter, but restricted to algebras: every displayed algebra over **Nat** admits a unique displayed algebra homomorphism from **Nat**. ◇

Advanced Remark 2.5.10. In light of Remark 2.5.6 and Remark 2.5.9, we have defined **Nat** as the fibrant replacement of the initial object in the category of $(\mathbf{1} \sqcup -)$-algebras. ◇

In rule form, the reverse direction of the natural isomorphism states that any displayed algebra (A, a_z, a_s) over **Nat** gives rise to a map out of **Nat**,

$$\frac{\Gamma \vdash n : \mathbf{Nat} \qquad \Gamma.\mathbf{Nat} \vdash A\,\mathsf{type} \qquad \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])]}{\Gamma \vdash \mathbf{rec}(a_z, a_s, n) : A[\mathbf{id}.n]}$$

which commutes with substitution,

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash n : \mathbf{Nat} \qquad \Gamma.\mathbf{Nat} \vdash A\,\mathsf{type} \qquad \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])]}{\Delta \vdash \mathbf{rec}(a_z, a_s, n)[\gamma] = \mathbf{rec}(a_z[\gamma], a_s[\gamma.\mathbf{Nat}.A], n[\gamma]) : A[\gamma.n[\gamma]]}$$

and is a displayed algebra homomorphism, which is to say that the map sends **zero** to a_z and $\mathbf{suc}(n)$ to $a_s(n, \mathbf{rec}(n))$:

$$\frac{\Gamma.\mathbf{Nat} \vdash A\,\mathsf{type} \qquad \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])]}{\Gamma \vdash \mathbf{rec}(a_z, a_s, \mathbf{zero}) = a_z : A[\mathbf{id}.\mathbf{zero}]}$$

$$\frac{\Gamma \vdash n : \mathbf{Nat} \qquad \Gamma.\mathbf{Nat} \vdash A\,\mathsf{type} \qquad \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])]}{\Gamma \vdash \mathbf{rec}(a_z, a_s, \mathbf{suc}(n)) = a_s[\mathbf{id}.n.\mathbf{rec}(a_z, a_s, n)] : A[\mathbf{id}.\mathbf{suc}(n)]}$$

Finally, the η rule of **Nat**, which is again typically omitted, expresses that there is exactly one displayed algebra homomorphism from **Nat** to (A, a_z, a_s): if $\Gamma.\mathbf{Nat} \vdash a : A$ is a term that sends **zero** to a_z and **suc**(n) to $a_s(n, a[\mathbf{id}.n])$, then it is equal to **rec**$(a_z, a_s, \mathbf{q})$.

$$\frac{\begin{array}{c}\Gamma.\mathbf{Nat} \vdash A\,\mathsf{type} \qquad \Gamma.\mathbf{Nat} \vdash a : A \qquad \Gamma \vdash n : \mathbf{Nat} \\ \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma \vdash a_z = a[\mathbf{id}.\mathbf{zero}] : A[\mathbf{id}.\mathbf{zero}] \\ \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])] \\ \Gamma.\mathbf{Nat} \vdash a_s[\mathbf{p}.\mathbf{q}.a] = a[\mathbf{p}.\mathbf{suc}(\mathbf{q})] : A[\mathbf{p}.\mathbf{suc}(\mathbf{q})]\end{array}}{\Gamma \vdash \mathbf{rec}(a_z, a_s, n) = a[\mathbf{id}.n] : A[\mathbf{id}.n]}$$ ✎

Exercise 2.26. Rewrite the first **rec** rule using named variables instead of **p** and **q**, and convince yourself that it expresses a form of natural number induction.

Exercise 2.27. Define addition for **Nat** in terms of **rec**. We strongly recommend solving Exercise 2.26 prior to this exercise in order to use standard named syntax.

Inductive types are initial algebras Our definition of **Nat** is more similar to **Void**, **Bool**, and + than it may first appear. In fact, all four types are initial algebras for different signatures, although the absence of recursive constructors allowed us to sidestep this machinery until now. The empty type **Void** is the initial algebra for the signature $X \mapsto \mathbf{0}$: a (displayed) **0**-algebra is just a (dependent) type with no additional data, so initiality asserts that any $\Gamma.\mathbf{Void} \vdash A\,\mathsf{type}$ admits a unique displayed algebra homomorphism—a dependent function with no additional conditions—from **Void**.

Likewise, (**Bool**, **true**, **false**) is the initial algebra for $X \mapsto \{\star\} \sqcup \{\star\}$. A displayed $(\{\star\} \sqcup \{\star\})$-algebra over **Bool** is a type $\Gamma.\mathbf{Bool} \vdash A\,\mathsf{type}$ and two terms $\Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}]$ and $\Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]$; initiality states that for any such displayed algebra there is a unique displayed algebra homomorphism (**Bool**, **true**, **false**) $\to (A, a_t, a_f)$:

$$\rho_{\Gamma,A,a_t,a_f} : \{a \in \mathsf{Tm}(\Gamma.\mathbf{Bool}, A) \mid a_t = a[\mathbf{id}.\mathbf{true}] \wedge a_f = a[\mathbf{id}.\mathbf{false}]\} \cong \{\star\}$$

Coproduct types are more complicated because their signature involves types that may depend on the context, but setting this aside, $(A + B, \mathbf{inl}(-), \mathbf{inr}(-))$ is the initial algebra for $X \mapsto A \sqcup B$.

We refrain from restating Slogan 2.5.5 in terms of initial algebras because the general theory of displayed algebras and homomorphisms for a given signature is too significant a detour; we hope the reader is convinced that a general pattern exists.

Exercise 2.28. In Section 2.5.2, our definition of **Bool** roughly asserted a natural isomorphism between $a \in \mathsf{Tm}(\Gamma.\mathbf{Bool}, A)$ and pairs of terms $(a[\mathbf{id}.\mathbf{true}], a[\mathbf{id}.\mathbf{false}])$. Prove that this definition is equivalent to the ρ_{Γ,A,a_t,a_f} characterization above.

2.5.5 *Unicity via extensional equality*

Thus far we have defined the inductive types **Void**, **Bool**, +, and **Nat** by equipping them with constructors and asserting that dependent maps out of them are *judgmentally uniquely determined* by where they send those constructors. That is, a choice of where to send the constructors determines a map via elimination, and any two maps out of an inductive type are judgmentally equal if they agree on the constructors—no matter how differently those maps are obtained. Most alarmingly, as we have seen in Exercise 2.21, the judgmental unicity principle for **Void** implies that *all* terms in contexts containing **Void** are equal to one another!

It turns out that these η laws are derivable from the other rules of inductive types in the presence of equality reflection (Section 2.4.4), the other suspiciously strong rule of extensional type theory.

Theorem 2.5.11. *The following rule (η for* **Void***) can be derived from the other rules for* **Void** *in conjunction with the rules for* **Eq**.

$$\frac{\Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash a : A}{\Gamma \vdash \mathbf{absurd}(b) = a[\mathbf{id}.b] : A[\mathbf{id}.b]} \text{ ✎,⇒}$$

Proof. Suppose $\Gamma \vdash b : \mathbf{Void}$ and $\Gamma.\mathbf{Void} \vdash a : A$. By equality reflection (Section 2.4.4), it suffices to exhibit an element of $\mathbf{Eq}(A[\mathbf{id}.b], \mathbf{absurd}(b), a[\mathbf{id}.b])$, which we obtain easily by **Void** elimination:

$$\Gamma \vdash \mathbf{absurd}(b) : \mathbf{Eq}(A[\mathbf{id}.b], \mathbf{absurd}(b), a[\mathbf{id}.b])$$ □

In Chapter 3 we will see that all of these suspicious rules are problematic from an implementation perspective, leading us to replace extensional type theory with *intensional type theory* (Chapter 4), which differs formally in only two ways: it replaces **Eq**-types with a different equality type that does not admit equality reflection, and it deletes the η rules from **Void**, **Bool**, +, and **Nat**.

In light of the fact that the latter rules are derivable from the former, we—as is conventional—simply omit the η rules for inductive types from the official specification of extensional type theory. (These rules were all marked as provisional ✎.) Note that this does *not* apply to the η rules for **Π**, Σ, or **Unit**, which remain in both type theories.

Semantically, deleting these η rules relaxes the unique existence to simply *existence*. An algebra which admits a (possibly non-unique) algebra homomorphism to any other algebra is known as *weakly initial* instead of initial. Rather than asking for the collection of algebra homomorphisms to be naturally isomorphic to $\{\star\}$, we ask for the map from algebra homomorphisms to $\{\star\}$ to admit a natural *section* (right inverse).

Advanced Remark 2.5.12. Recalling Remark 2.5.6, Theorem 2.5.11 corresponds to the fact that a class of morphisms $\mathcal{L}$ that is weakly orthogonal to $\mathcal{R}$ is in fact orthogonal to $\mathcal{R}$ when $\mathcal{R}$ is closed under relative diagonals ($X \longrightarrow Y \in \mathcal{R} \implies X \longrightarrow X \times_Y X \in \mathcal{R}$). ⋄

Exercise 2.29. Prove that the η rule for **Bool** can be derived from the other rules for **Bool** in conjunction with the rules for **Eq**, by mirroring the proof of Theorem 2.5.11.

2.6 *Universes:* $\mathbf{U}_0, \mathbf{U}_1, \mathbf{U}_2, \ldots$

We are nearly finished with our definition of extensional type theory, but what's missing is significant: our theory is still not full-spectrum dependent in the sense described in Section 1.2! That is, we have still not introduced the ability to define a family of types whose head type constructor differs at different indices, such as a **Bool**-indexed family of types which sends **true** to **Nat** and **false** to **Unit**. A more subtle but fatal flaw with our current theory is that—despite all the logical connectives at our disposal—we cannot prove that **true** and **false** are different, i.e., we cannot exhibit a term $\mathbf{1} \vdash p : \mathbf{\Pi}(\mathbf{Eq}(\mathbf{Bool}, \mathbf{true}, \mathbf{false}), \mathbf{Void})$.

It turns out that addressing the former will solve the latter *en passant*, so in this section we will discuss two approaches for defining dependent types by case analysis. In Section 2.6.1 we introduce *large elimination*, which equips inductive types with a second elimination principle targeting type-valued algebras (which send each constructor to a *type*), in addition to their usual elimination principle targeting algebras valued in a single dependent type (which send each constructor to a *term* of that type).

Unfortunately we will see that large elimination has some serious limitations, so it will not be an official part of our definition of extensional type theory. Instead, in Section 2.6.2, we introduce *type universes*, connectives which internalize the judgment $\Gamma \vdash A$ type modulo "size issues." By internalizing types as terms of a universe type, universes reduce the problem of computing *types* by case analysis to the problem of computing *terms* by case analysis, which we solved in Section 2.5. That said, universes are a deep and complex topic that will bring us one step closer to our discussion of homotopy type theory in Chapter 5.

2.6.1 *Large elimination*

In Section 2.5 we discussed elimination principles for inductive types such as **Bool**, which allow us to define dependent functions out of inductive types by cases on that type's constructors. A direct but uncommon way of achieving full-spectrum

dependency is to equip each inductive type with a second elimination principle, *large elimination*, which allows us to define dependent *families of types* by cases [Smi89].[12]

In the case of **Bool**, large elimination is characterized by the following rules:

$$\frac{\Gamma \vdash A_t\ \mathsf{type} \qquad \Gamma \vdash A_f\ \mathsf{type} \qquad \Gamma \vdash b : \mathbf{Bool}}{\Gamma \vdash \mathbf{If}\,(A_t, A_f, b)\ \mathsf{type}}\ \text{✎}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A_t\ \mathsf{type} \qquad \Gamma \vdash A_f\ \mathsf{type} \qquad \Gamma \vdash b : \mathbf{Bool}}{\Delta \vdash \mathbf{If}\,(A_t, A_f, b)[\gamma] = \mathbf{If}\,(A_t[\gamma], A_f[\gamma], b[\gamma])\ \mathsf{type}}\ \text{✎}$$

$$\frac{\Gamma \vdash A_t\ \mathsf{type} \qquad \Gamma \vdash A_f\ \mathsf{type}}{\Gamma \vdash \mathbf{If}\,(A_t, A_f, \mathbf{true}) = A_t\ \mathsf{type} \qquad \Gamma \vdash \mathbf{If}\,(A_t, A_f, \mathbf{false}) = A_f\ \mathsf{type}}\ \text{✎}$$

If we compare these to the rules of ordinary ("small") elimination,

$$\frac{\Gamma.\mathbf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}] \qquad \Gamma \vdash b : \mathbf{Bool}}{\Gamma \vdash \mathbf{if}\,(a_t, a_f, b) : A[\mathbf{id}.b]}$$

$$\frac{\Gamma.\mathbf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]}{\Gamma \vdash \mathbf{if}\,(a_t, a_f, \mathbf{true}) = a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash \mathbf{if}\,(a_t, a_f, \mathbf{false}) = a_f : A[\mathbf{id}.\mathbf{false}]}$$

we see that the large eliminator **If** is exactly analogous to the small eliminator **if** "specialized to $A :\equiv \mathsf{type}$." Note that this statement is nonsense because the judgment "type" is not a type, but the intuition is useful and will be formalized momentarily. (Indeed, for this reason we cannot formally obtain **If** as a special case of **if**.) Continuing on with the metaphor, the rule for **If** is simpler than the rule for **if** because it has a fixed codomain "type" which is moreover *not* dependent on **Bool**: it makes no sense to ask for "$\Gamma \vdash A_t\ \mathsf{type}[\mathbf{id}.\mathbf{true}]$."

It is even more standard to omit the η rule for large elimination than for small elimination (which is itself typically omitted), but such a rule would state that dependent types indexed by **Bool** are uniquely determined by their values on **true** and **false**:

$$\frac{\Gamma.\mathbf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash b : \mathbf{Bool}}{\Gamma \vdash A[\mathbf{id}.b] = \mathbf{If}\,(A[\mathbf{id}.\mathbf{true}], A[\mathbf{id}.\mathbf{false}], b)\ \mathsf{type}}\ \text{✎✎}$$

If we include the η rule, then the rules for **If** would express that instantiating a **Bool**-indexed type at **true** and **false**, i.e. $((\mathbf{id}.\mathbf{true})^*, (\mathbf{id}.\mathbf{false})^*)$, has a natural inverse:

$$((\mathbf{id}.\mathbf{true})^*, (\mathbf{id}.\mathbf{false})^*) : \mathsf{Ty}(\Gamma.\mathbf{Bool}) \cong \mathsf{Ty}(\Gamma) \times \mathsf{Ty}(\Gamma)$$

[12] Large elimination maps **Bool** into the collection of all types, which is "large" (in the sense of being "the proper class of all sets") rather than the collection of terms of a single type, which is "small" ("a set").

Again, compare this to our original formulation of small elimination for **Bool**:

$$((\mathbf{id}.\mathbf{true})^*, (\mathbf{id}.\mathbf{false})^*) : \mathsf{Tm}(\Gamma.\mathbf{Bool}, A) \cong \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{true}]) \times \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{false}])$$

When we elide η, large elimination instead states that this map has a *section* (a right inverse), which is to say that a choice of where to send **true** and **false** determines a family of types via **If**, but *not uniquely*. This follows the discussion in Section 2.5.5, except that we *cannot* derive the η rule for large elimination from extensional equality because there is no type "$\mathbf{Eq}(\mathsf{type}, -, -)$" available to carry out the argument in Theorem 2.5.11.

Remark 2.6.1. Large elimination only applies to types defined by mapping-out properties such as inductive types; there is no corresponding principle for mapping-in connectives like $\mathbf{\Pi}(A, B)$ because these do not quantify over any target, whether "small" or "large." ◇

Remark 2.6.2. If we have both small and large elimination for **Bool**, then we can combine them into a derived induction principle for **Bool** that works for any $a_t : A_t$ and $a_f : A_f$, using large elimination to define the type family into which we perform a small elimination.

$$\frac{\Gamma \vdash a_t : A_t \qquad \Gamma \vdash a_f : A_f \qquad \Gamma \vdash b : \mathbf{Bool}}{\Gamma \vdash \mathbf{if}(a_t, a_f, b) : \mathbf{If}(A_t, A_f, b)} \;\text{✎,⇒}$$ ◇

With large elimination—or a related feature, type universes—we can prove the disjointness of the booleans. (Although the proof below uses equality reflection, the same theorem holds in intensional type theory for essentially the same reason.) Our claim that we *cannot* prove disjointness without these features is a (relatively simple) independence metatheorem requiring a model construction; see *The Independence of Peano's Fourth Axiom from Martin-Löf's Type Theory Without Universes* [Smi88].

Theorem 2.6.3. *Using the rules for* **If***, there is a term*

$$\mathbf{1} \vdash \mathsf{disjoint} : \mathbf{\Pi}(\mathbf{Eq}(\mathbf{Bool}, \mathbf{true}, \mathbf{false}), \mathbf{Void})$$

Proof. We informally describe the derivation of disjoint. By **Π**-introduction we may assume $\mathbf{Eq}(\mathbf{Bool}, \mathbf{true}, \mathbf{false})$ and prove **Void**. In order to do this, consider the following auxiliary family of types over **Bool**:

$$\mathbf{1}.\mathbf{Eq}(\mathbf{Bool}, \mathbf{true}, \mathbf{false}).\mathbf{Bool} \vdash P := \mathbf{If}(\mathbf{Unit}, \mathbf{Void}, \mathbf{q})\ \mathsf{type}$$

Then

$$\begin{aligned}\mathbf{1}.\mathbf{Eq}(\mathbf{Bool},\mathbf{true},\mathbf{false}) &\vdash \mathbf{Unit}\\ &= P[\mathbf{id}.\mathbf{true}] && \text{by } \beta \text{ for } \mathbf{If}\\ &= P[\mathbf{id}.\mathbf{false}] && \text{by equality reflection on } \mathbf{q}\\ &= \mathbf{Void}\ \mathrm{type} && \text{by } \beta \text{ for } \mathbf{If}\end{aligned}$$

and therefore $\mathbf{1}.\mathbf{Eq}(\mathbf{Bool},\mathbf{true},\mathbf{false}) \vdash \mathbf{tt} : \mathbf{Void}$. In sum, we define $\mathsf{disjoint} := \lambda(\mathbf{tt})$. □

As for other inductive types, the large elimination principle of **Void** is:

$$\frac{\Gamma \vdash a : \mathbf{Void}}{\Gamma \vdash \mathbf{Absurd}(a)\ \mathrm{type}}$$ ✎

Unfortunately, we run into a problem when stating large elimination for **Nat**.

$$\frac{\Gamma \vdash n : \mathbf{Nat} \qquad \Gamma \vdash A_z\ \mathrm{type} \qquad \Gamma.\mathbf{Nat}.\text{“type”} \vdash A_s\ \mathrm{type}}{\Gamma \vdash \mathbf{Rec}(A_z, A_s, n)\ \mathrm{type}}$$!?

In the ordinary eliminator, the branch for $\mathbf{suc}(-)$ has two variables $m : \mathbf{Nat}, a : A(m)$ binding the predecessor m and (recursively) the result a of the eliminator on m. When “$A :=$ type” the recursive result is a *type*, meaning that the $\mathbf{suc}(-)$ branch ought to bind a *type variable*, a concept which is not a part of our theory. This is a serious problem because recursive constructions of types were a major class of examples in Section 1.2.

Exercise 2.30. There is however a *non-recursive* large elimination principle for **Nat** which defines a type by case analysis on whether a number is **zero**. This principle follows from the rules in this section along with the other rules of extensional type theory; state and define it.

Exercise 2.31. Although it is highly non-standard, it is possible to consider a substitution calculus that includes rules for extending contexts by type variables:

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\vdash \Gamma.\mathrm{type}\ \mathsf{cx}}$$

Write the remaining rules governing this new form of context extension. (Hint: substitutions $\Delta \vdash \gamma : \Gamma.\mathrm{type}$ should be in bijection with a certain set.)

2.6.2 *Universes*

Although large elimination is a useful concept, it sees essentially no use in practice. We have just seen one reason: large eliminators cannot be recursive. The standard approach is instead to include *universe types*, which are "types of types," or types which internalize the judgment $\Gamma \vdash A\,\mathsf{type}$. Using universes, we can recover large elimination as small elimination into a universe; we are also able to express polymorphic type quantification using dependent functions out of a universe.

A universe is a type with no parameters, so its formation rule is once again a natural family of constants $\mathbf{U}_\Gamma \in \mathsf{Ty}(\Gamma)$, or

$$\frac{}{\Gamma \vdash \mathbf{U}\,\mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{U} = \mathbf{U}[\gamma]\,\mathsf{type}}$$

As for its terms, the most straightforward definition would be to stipulate a natural isomorphism between terms of $\mathbf{U}$ and types:

$$\iota : \mathsf{Tm}(\Gamma, \mathbf{U}) \cong \mathsf{Ty}(\Gamma) \tag{?!}$$

Note that just as we did not ask for terms of $\mathbf{\Pi}$-types to literally be terms with an extra free variable, we cannot ask for terms of $\mathbf{U}$ to literally be types: these are two different sorts!

In inference rules, the forward map of the isomorphism would introduce a new type former $\mathbf{El}(-)$[13] which "decodes" an element of $\mathbf{U}$ into a genuine type. The reverse map conversely "encodes" a genuine type as an element of $\mathbf{U}$. These intuitions lead us to often refer to elements of $\mathbf{U}$ as *codes* for types.

$$\frac{\Gamma \vdash a : \mathbf{U}}{\Gamma \vdash \mathbf{El}(a)\,\mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}}{\Delta \vdash \mathbf{El}(a)[\gamma] = \mathbf{El}(a[\gamma])\,\mathsf{type}}$$

$$\frac{\Gamma \vdash A\,\mathsf{type}}{\Gamma \vdash \mathbf{code}(A) : \mathbf{U}}\ ?! \qquad \frac{\Gamma \vdash A\,\mathsf{type}}{\Gamma \vdash \mathbf{El}(\mathbf{code}(A)) = A\,\mathsf{type}}\ ?! \qquad \frac{\Gamma \vdash c : \mathbf{U}}{\Gamma \vdash \mathbf{code}(\mathbf{El}(c)) = c : \mathbf{U}}\ ?!$$

Unfortunately we can't have nice things, as the last three rules above—the ones involving $\mathbf{code}$—are unsound. In particular they imply that $\mathbf{U}$ contains (a code for) $\mathbf{U}$, making it a "type of all types, including itself" and therefore subject to a variation on Russell's paradox known as *Girard's paradox* [Coq86], as outlined in Section 2.6.4.

We instead postulate a type $\mathbf{U}$ of type-codes which decode via $\mathbf{El}$ into types. (Indeed, with large elimination it is even possible to define such a type manually, e.g. $\mathbf{U} :\equiv \mathbf{Bool}$

[13] This name is not so mysterious: it means "*elements* of," and is pronounced "ell" or, often, omitted.

with $\mathbf{El}(\mathbf{true}) :\equiv \mathbf{Unit}$ and $\mathbf{El}(\mathbf{false}) :\equiv \mathbf{Void}$.)

$$\mathbf{U}_\Gamma \in \mathsf{Ty}(\Gamma)$$
$$\mathbf{El} : \mathsf{Tm}(\Gamma, \mathbf{U}) \to \mathsf{Ty}(\Gamma)$$

To populate this collection of type-codes, we manually equip $\mathbf{U}$ with codes decoding to each type former we have seen so far, with the crucial exception of $\mathbf{U}$ itself. This approach is somewhat verbose—for each type former we add an introduction rule for $\mathbf{U}$, a substitution rule, and an equation stating that $\mathbf{El}$ decodes it to the corresponding type—but it allows us to avoid Girard's paradox while still equipping $\mathbf{U}$ with codes for (almost) every type in our theory. It also means that universes do not fit into Slogan 2.5.5, although we will revisit this point in Remark 2.6.9.

For example, to close $\mathbf{U}$ under $\mathbf{\Pi}$-types we add the following rules:

$$\frac{\Gamma \vdash a : \mathbf{U} \qquad \Gamma.\mathbf{El}(a) \vdash b : \mathbf{U}}{\Gamma \vdash \mathbf{pi}(a,b) : \mathbf{U}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U} \qquad \Gamma.\mathbf{El}(a) \vdash b : \mathbf{U}}{\Delta \vdash \mathbf{pi}(a,b)[\gamma] = \mathbf{pi}(a[\gamma], b[\gamma.\mathbf{El}(a)]) : \mathbf{U}}$$

$$\frac{\Gamma \vdash a : \mathbf{U} \qquad \Gamma.\mathbf{El}(a) \vdash b : \mathbf{U}}{\Gamma \vdash \mathbf{El}(\mathbf{pi}(a,b)) = \mathbf{\Pi}(\mathbf{El}(a), \mathbf{El}(b))\ \mathsf{type}}$$

The third rule states that $\mathbf{pi}(a,b)$ is the code in $\mathbf{U}$ for the type $\mathbf{\Pi}(\mathbf{El}(a), \mathbf{El}(b))$. Note that the context of b in the introduction rule for $\mathbf{pi}(a,b)$ makes reference to $\mathbf{El}(a)$, mirroring the dependency structure of $\mathbf{\Pi}$-types. Although this move is forced, it means that the definitions of $\mathbf{U}$ and $\mathbf{El}$ each reference the other—the type of a constructor of $\mathbf{U}$ mentions $\mathbf{El}$, and the type of $\mathbf{El}$ itself mentions $\mathbf{U}$—so $\mathbf{U}$ and $\mathbf{El}$ must be defined simultaneously. In fact, this is the paradigmatic example of an *inductive-recursive* definition, an inductive type that is defined simultaneously with a recursive function out of it [Dyb00].

It is no more difficult to close $\mathbf{U}$ under dependent pairs, extensional equality, the unit type, and inductive types. These rules quickly become tedious, so we write only their introduction rules below, leaving the remaining rules to Appendix A.

$$\frac{\Gamma \vdash a : \mathbf{U} \qquad \Gamma.\mathbf{El}(a) \vdash b : \mathbf{U}}{\Gamma \vdash \mathbf{sig}(a,b) : \mathbf{U}} \qquad \frac{\Gamma \vdash a : \mathbf{U} \qquad \Gamma \vdash x, y : \mathbf{El}(a)}{\Gamma \vdash \mathbf{eq}(a,x,y) : \mathbf{U}}$$

$$\frac{}{\Gamma \vdash \mathbf{unit} : \mathbf{U}} \qquad \frac{}{\Gamma \vdash \mathbf{void} : \mathbf{U}} \qquad \frac{}{\Gamma \vdash \mathbf{bool} : \mathbf{U}} \qquad \frac{}{\Gamma \vdash \mathbf{nat} : \mathbf{U}}$$

$$\frac{\Gamma \vdash a, b : \mathbf{U}}{\Gamma \vdash \mathbf{coprod}(a,b) : \mathbf{U}}$$

We can now recover the large elimination principles of Section 2.6.1 in terms of small elimination into the type $\mathbf{U}$. Moreover, because we can perfectly well extend the context by a variable of type $\mathbf{U}$, we can now also construct types by recursion on natural numbers:

$$\frac{\Gamma \vdash n : \mathbf{Nat} \qquad \Gamma \vdash a_z : \mathbf{U} \qquad \Gamma.\mathbf{Nat}.\mathbf{U} \vdash a_s : \mathbf{U}}{\Gamma \vdash \mathbf{Rec}(a_z, a_s, n) \coloneqq \mathbf{El}(\mathbf{rec}(a_z, a_s, n))\ \mathsf{type}} \Rightarrow$$

Notation 2.6.4. In general, we will refer to a pair (B, E) of a type $\Gamma \vdash B\ \mathsf{type}$ and a type family $\Gamma.B \vdash E\ \mathsf{type}$ over it as a *universe* whenever it is appropriate to think of B as a collection of codes for types and E as a decoding function. For example, we can regard **Nat** as a universe of (codes of) finite types when equipped with the recursively-defined type family sending each n to the coproduct of n copies of **Unit**. We will encounter more examples of universes in Sections 2.7 and 5.2.

Remark 2.6.5. Proof assistant users are very familiar with universes, so such readers may be wondering why they have never seen **El** before. Indeed, proof assistants such as Rocq and Agda treat types and elements of $\mathbf{U}$ as indistinguishable. Historically, much of the literature calls such universes—for which $\mathsf{Tm}(\Gamma, \mathbf{U}) \subseteq \mathsf{Ty}(\Gamma)$—*universes à la Russell*, in contrast to our *universes à la Tarski*, but we find such a subset inclusion to be meta-suspicious.

Instead, we prefer to say that Rocq and Agda programs do not expose the notion of type to the user at all, instead consistently referring only to elements of $\mathbf{U}$. This obviates the need for the user to ever write or see **El**, and the necessary calls to **El** can be inserted automatically by the proof assistant in a process known as *elaboration*. ◇

Remark 2.6.6. Another more semantically natural variation of universes relaxes the judgmental equalities governing **El** to *isomorphisms* $\mathbf{El}(\mathbf{pi}(a, b)) \cong \mathbf{\Pi}(\mathbf{El}(a), \mathbf{El}(b))$, producing what are known as *weak universes à la Tarski*. However, our *strict* formulation is more standard and more convenient. ◇

Advanced Remark 2.6.7. Universes in type theory play a similar role to Grothendieck universes and their categorical counterparts in set theory and category theory. We often refer to types encoded by $\mathbf{U}$ as *small* or $\mathbf{U}$-small, and ask for small types to be closed under various operations. As a result, universes in type theory roughly have the same proof-theoretical strength as strongly inaccessible cardinals. Note, however, that the lack of choice and excluded middle in type theory (see Section 2.7.4) precludes a naïve comparison between it and ZFC or similar theories; see Section 3.5.1. ◇

2.6.3 Hierarchies of universes

Our definition of $\mathbf{U}$ is perfectly correct, but the fact that $\mathbf{U}$ lacks a code for itself means that we cannot recursively define types that mention $\mathbf{U}$. In addition, although we can quantify over "small" types with $\mathbf{\Pi}(\mathbf{U}, -)$, we cannot write any type quantifiers whose domain includes $\mathbf{U}$. We cannot fix these shortcomings directly, but we can mitigate them by defining a *second* universe type $\mathbf{U}_1$ closed under all the same type codes as before *as well as a code for* $\mathbf{U}$, but no code for $\mathbf{U}_1$ itself. The same problem occurs one level up, so we add a third universe $\mathbf{U}_2$ containing codes for $\mathbf{U}$ and $\mathbf{U}_1$ but not $\mathbf{U}_2$, and so forth.

In practice, nearly all applications of type theory require only a finite number of universes, but for uniformity and because this number varies between applications, it is typical to ask for a countably infinite (alternatively, finite but arbitrary) tower of universes each of which contains codes for the smaller ones. (For uniformity we write $\mathbf{U}_0 := \mathbf{U}$.) This collection of $\mathbf{U}_i$ is known as a *universe hierarchy*.

To define an infinite number of types and terms, we must now write *rule schemas*, collections of rules that must be repeated for every (external, not internal) natural number $i > 0$. Each of these rules follows the same pattern in $\mathbf{U}$, with one new feature: $\mathbf{U}_i$ contains a code $\mathbf{uni}_{i,j}$ for $\mathbf{U}_j$ whenever j is strictly smaller than i.

$$\frac{}{\Gamma \vdash \mathbf{U}_i\ \mathsf{type}} \qquad \frac{\Gamma \vdash a : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}_i(a)\ \mathsf{type}} \qquad \frac{\Gamma \vdash a, b : \mathbf{U}_i}{\Gamma \vdash \mathbf{coprod}_i(a,b) : \mathbf{U}_i}$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma.\mathbf{El}_i(a) \vdash b : \mathbf{U}_i}{\Gamma \vdash \mathbf{pi}_i(a,b) : \mathbf{U}_i \qquad \Gamma \vdash \mathbf{sig}_i(a,b) : \mathbf{U}_i} \qquad \frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma \vdash x, y : \mathbf{El}_i(a)}{\Gamma \vdash \mathbf{eq}_i(a,x,y) : \mathbf{U}_i}$$

$$\frac{}{\Gamma \vdash \mathbf{unit}_i : \mathbf{U}_i} \qquad \frac{}{\Gamma \vdash \mathbf{void}_i : \mathbf{U}_i} \qquad \frac{}{\Gamma \vdash \mathbf{bool}_i : \mathbf{U}_i} \qquad \frac{}{\Gamma \vdash \mathbf{nat}_i : \mathbf{U}_i} \qquad \frac{j < i}{\Gamma \vdash \mathbf{uni}_{i,j} : \mathbf{U}_i}$$

Again for uniformity we write $\mathbf{pi}_0(a,b) := \mathbf{pi}(a,b)$, etc., and we omit the associated substitution rules and the type equations explaining how each $\mathbf{El}_i$ computes on codes, such as $\mathbf{El}_i(\mathbf{eq}_i(a,x,y)) = \mathbf{Eq}(\mathbf{El}_i(a), x, y)$ and $\mathbf{El}_i(\mathbf{uni}_{i,j}) = \mathbf{U}_j$.

It is easy to see that the rules for $\mathbf{U}_{i+1}$ are a superset of the rules for $\mathbf{U}_i$: the only difference is the addition of the code $\mathbf{uni}_{i+1,i} : \mathbf{U}_{i+1}$ and codes that mention this code, such as $\mathbf{pi}_{i+1}(\mathbf{uni}_{i+1,i}, \mathbf{uni}_{i+1,i}) : \mathbf{U}_{i+1}$. Thus it should be possible to prove that every closed code of type $\mathbf{U}_i$ has a counterpart of type $\mathbf{U}_{i+1}$ that decodes to the same type, that is, "$\mathbf{U}_i \subsetneq \mathbf{U}_{i+1}$." However, this fact is not visible inside the theory. We have no induction principle for the universe, so we cannot define an "inclusion" function $f : \mathbf{U}_i \to \mathbf{U}_{i+1}$ much less prove that it satisfies $\mathbf{El}_{i+1}(f(a)) = \mathbf{El}_i(a)$. And there is simply no way, external or otherwise, to "lift" a variable of type $\mathbf{U}_i$ to the type $\mathbf{U}_{i+1}$.

We thus equip our universe hierarchy with one final operation: a *lifting* operation that includes elements of $\mathbf{U}_i$ into $\mathbf{U}_{i+1}$, which is compatible with $\mathbf{El}$ and sends type codes of $\mathbf{U}_i$ to their counterparts in $\mathbf{U}_{i+1}$. Such a strict lifting operation allows users to generally avoid worrying about universe levels, because small codes can always be hoisted up to their larger counterparts when needed.

$$\frac{\Gamma \vdash c : \mathbf{U}_i}{\Gamma \vdash \mathbf{lift}_i(c) : \mathbf{U}_{i+1}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}_i}{\Delta \vdash \mathbf{lift}_i(a)[\gamma] = \mathbf{lift}_i(a[\gamma]) : \mathbf{U}_{i+1}}$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}_{i+1}(\mathbf{lift}_i(a)) = \mathbf{El}_i(a)\ \mathsf{type}}$$

The last rule above states that a code and its lift both encode the same type. Recalling that the entire point of a universe hierarchy is to get as close as possible to "$\mathbf{U} : \mathbf{U}$" without being inconsistent, it makes sense to treat lifts as a clerical operation that does not affect the type about which we speak. In addition, this equation is actually needed to state that $\mathbf{lift}$ commutes with codes, such as $\mathbf{pi}$ (other rules omitted):

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma.\mathbf{El}_i(a) \vdash b : \mathbf{U}_i}{\Gamma \vdash \mathbf{lift}_i(\mathbf{pi}_i(a,b)) = \mathbf{pi}_{i+1}(\mathbf{lift}_i(a), \mathbf{lift}_i(b)) : \mathbf{U}_{i+1}}$$

Remark 2.6.8. We say a universe hierarchy is *(strictly) cumulative* when it is equipped with $\mathbf{lift}$ operations that commute (strictly) with codes. Historically the term "cumulativity" often refers to material subset inclusions $\mathsf{Tm}(\Gamma, \mathbf{U}_i) \subseteq \mathsf{Tm}(\Gamma, \mathbf{U}_{i+1})$ but once again such conditions are incompatible with our perspective. ◇

Remark 2.6.9. There is an equivalent presentation of universe hierarchies known as *universes à la Coquand* in which one stratifies the type judgment itself, and the ith universe precisely internalizes the ith type judgment [Coq13; Coq19; Gra+21; FAM23]. That is, we have sorts $\mathsf{Ty}_i(\Gamma)$ for $i \in \mathbb{N} \cup \{\top\}$ with $\mathsf{Ty}(\Gamma) := \mathsf{Ty}_\top(\Gamma)$, and natural isomorphisms $\mathsf{Ty}_i(\Gamma) \cong \mathsf{Tm}(\Gamma, \mathbf{U}_i)$ for $i \in \mathbb{N}$ mediated by $\mathbf{El}_i/\mathbf{code}_i$. This presentation essentially creates a new judgmental structure designed to be internalized by $\mathbf{U}$, and has the concrete benefit of unifying type formation and universe introduction into a single set of rules. ◇

Exercise 2.32. Check that the equational rule $\mathbf{lift}_i(\mathbf{pi}_i(a,b)) = \mathbf{pi}_{i+1}(\mathbf{lift}_i(a), \mathbf{lift}_i(b))$ above is meta-well-typed. (Hint: you need to use $\mathbf{El}_{i+1}(\mathbf{lift}_i(a)) = \mathbf{El}_i(a)$.)

Exercise 2.33. We only included lifts from $\mathbf{U}_i$ to $\mathbf{U}_{i+1}$, rather than from $\mathbf{U}_i$ to $\mathbf{U}_j$ for every $i < j$. Show that the latter notion of lift is derivable for any concrete $i < j$ and that it satisfies the expected equations.

2.6.4 Girard's paradox

We close our discussion of universes by substantiating our claim in Section 2.6.2 that it is inconsistent for $\mathbf{U}$ to contain a code for itself, a fact commonly known as Girard's paradox; specifically, we present a simplified argument due to Hurkens [Hur95].[14] The details of this paradox are not relevant to any later material in this book, so the reader may freely skip the rest of this section. In this subsection alone, we adopt the (inconsistent) rules of Section 2.6.2 pertaining to **code**.

At a high level, the fact that $\mathbf{U}$ contains a code for itself means that we can construct a universe Θ that admits an embedding from its own double power set $\mathcal{P}\ (\mathcal{P}\ \Theta)$; from this we can define a "set of all ordinals" and carry out a version of the Burali-Forti paradox. The details become somewhat involved, in part because the standard paradoxes of set theory rely on comprehension and extensionality principles not available to us in type theory. Indeed, historically it was far from clear that "$\mathbf{U} : \mathbf{U}$" was inconsistent, and in fact Martin-Löf's first version of type theory had this very flaw [Mar71].

$$\mathcal{P} : \mathbf{U} \to \mathbf{U}$$
$$\mathcal{P}\ A = \mathbf{code}(\mathbf{El}(A) \to \mathbf{U})$$

$$\mathcal{P}^2 : \mathbf{U} \to \mathbf{U}$$
$$\mathcal{P}^2\ A = \mathcal{P}\ (\mathcal{P}\ A)$$

$$\Theta : \mathbf{U}$$
$$\Theta = \mathbf{code}((A : \mathbf{U}) \to (\mathbf{El}(\mathcal{P}^2\ A) \to \mathbf{El}(A)) \to \mathbf{El}(\mathcal{P}^2\ A))$$

Lemma 2.6.10 (Powerful universe)**.** *The universe Θ admits maps*

$$\tau : \mathbf{El}(\mathcal{P}^2\ \Theta) \to \mathbf{El}(\Theta)$$
$$\sigma : \mathbf{El}(\Theta) \to \mathbf{El}(\mathcal{P}^2\ \Theta)$$

such that

$$(C : \mathbf{El}(\mathcal{P}^2\ \Theta)) \to (\sigma\ (\tau\ C) = \lambda(\phi : \mathbf{El}(\mathcal{P}\ \Theta)) \to C(\phi \circ \tau \circ \sigma))$$

Proof. We define:

$$\tau : \mathbf{El}(\mathcal{P}^2\ \Theta) \to \mathbf{El}(\Theta)$$

[14] An Agda formalization of Hurkens's paradox is available at `https://github.com/agda/agda/blob/master/test/Succeed/Hurkens.agda`; formalizations in other proof assistants are readily available online.

$$\tau\ (\Phi : \mathbf{El}(\mathcal{P}^2\ \Theta))\ (A : \mathbf{U})\ (f : \mathbf{El}(\mathcal{P}^2\ A) \to \mathbf{El}(A))\ (\chi : \mathbf{El}(\mathcal{P}\ A)) =$$
$$\Phi\ (\lambda(\theta : \Theta) \to \chi\ (f\ (\theta\ A\ f)))$$

$$\sigma : \mathbf{El}(\Theta) \to \mathbf{El}(\mathcal{P}^2\ \Theta)$$
$$\sigma\ \theta = \theta\ \Theta\ \tau$$

We leave the equational condition to Exercise 2.34. □

Exercise 2.34. Show that the above definitions of τ, σ satisfy the necessary equation.

As an immediate consequence of Lemma 2.6.10, we have:

$$\sigma\ (\tau\ (\sigma\ x)) = \lambda(\phi : \mathbf{El}(\mathcal{P}\ \Theta)).\ \sigma\ x\ (\phi \circ \tau \circ \sigma) \tag{2.1}$$

One way to understand the statement of Lemma 2.6.10 is that, regarding $\mathcal{P}$ as a functor whose action on $f : \mathbf{El}(Y) \to \mathbf{El}(X)$ is precomposition $f^* : \mathbf{El}(\mathcal{P}\ X) \to \mathbf{El}(\mathcal{P}\ Y)$, the equational condition is equivalent to $\sigma \circ \tau = (\tau \circ \sigma)^{**}$.

We derive a contradiction from Lemma 2.6.10 by constructing ordinals within Θ:

-- $y < x$ ("$y \in x$") when each f in $\sigma\ x$ contains y
$(<) : \mathbf{El}(\Theta) \to \mathbf{El}(\Theta) \to \mathbf{U}$
$y < x = \mathbf{code}((f : \mathbf{El}(\mathcal{P}\ \Theta)) \to \mathbf{El}(\sigma\ x\ f) \to \mathbf{El}(f\ y))$

-- f is inductive if for all x, if f is in $\sigma\ x$ then x is in f
$\mathsf{ind} : \mathbf{El}(\mathcal{P}\ \Theta) \to \mathbf{U}$
$\mathsf{ind}\ f = \mathbf{code}((x : \mathbf{El}(\Theta)) \to \mathbf{El}(\sigma\ x\ f) \to \mathbf{El}(f\ x))$

-- x is well-founded if it is in every inductive f
$\mathsf{wf} : \mathbf{El}(\Theta) \to \mathbf{U}$
$\mathsf{wf}\ x = \mathbf{code}((f : \mathbf{El}(\mathcal{P}\ \Theta)) \to \mathbf{El}(\mathsf{ind}\ f) \to \mathbf{El}(f\ x))$

Specifically, we consider $\Omega := \tau\ (\lambda f \to \mathsf{ind}\ f)$, the collection of all inductive collections. Using Lemma 2.6.10 we argue that Ω is both well-founded and not well-founded.

Lemma 2.6.11. *Ω is well-founded.*

Proof. Suppose $f : \mathbf{El}(\mathcal{P}\ \Theta)$ is inductive; we must show $\mathbf{El}(f\ \Omega)$. By the definition of ind, for this it suffices to show $\mathbf{El}(\sigma\ \Omega\ f)$. Unfolding the definition of Ω and rewriting by the equation in Lemma 2.6.10 with $C := \mathsf{ind}$, it suffices to show that $f \circ \tau \circ \sigma$ is inductive.

Thus suppose we are given $x : \mathbf{El}(\Theta)$ such that $\mathbf{El}(\sigma\ x\ (f \circ \tau \circ \sigma))$; we must show $\mathbf{El}(f\ (\tau\ (\sigma\ x)))$. By rewriting $\mathbf{El}(\sigma\ x\ (f \circ \tau \circ \sigma))$ along Equation (2.1), we conclude

that $\mathbf{El}(\sigma\ (\tau\ (\sigma\ x))\ f)$. However, by our assumption that f is inductive, this implies $\mathbf{El}(f\ (\tau\ (\sigma\ x)))$, which is what we wanted to show. □

To prove that Ω is not well-founded, we start by showing that the collection of "sets not containing themselves" $\phi := \lambda y \to \mathbf{code}(\mathbf{El}(\tau\ (\sigma\ y) < y) \to \mathbf{Void})$ is inductive.

Lemma 2.6.12. *ϕ is inductive.*

Proof. Suppose we are given x such that $\mathbf{El}(\sigma\ x\ \phi)$; we must show $\mathbf{El}(\tau\ (\sigma\ x) < x) \to \mathbf{Void}$. Thus suppose $\mathbf{El}(\tau\ (\sigma\ x) < x)$, which is to say that for any f such that $\mathbf{El}(\sigma\ x\ f)$, we have $\mathbf{El}(f\ (\tau\ (\sigma\ x)))$. Using our hypothesis we may set $f := \phi$, from which we conclude $\mathbf{El}(\tau\ (\sigma\ (\tau\ (\sigma\ x))) < \tau\ (\sigma\ x)) \to \mathbf{Void}$. We derive the required contradiction by proving that $\mathbf{El}(\tau\ (\sigma\ (\tau\ (\sigma\ x))) < \tau\ (\sigma\ x))$ holds, by $\mathbf{El}(\tau\ (\sigma\ x) < x)$ and Exercise 2.35. □

Exercise 2.35. Show that $\mathbf{El}(x < y)$ implies $\mathbf{El}(\tau\ (\sigma\ x) < \tau\ (\sigma\ y))$.

Theorem 2.6.13. *There is a closed term of type* **Void**.

Proof. Because Ω is well-founded and ϕ is inductive, we have $\mathbf{El}(\tau\ (\sigma\ \Omega) < \Omega) \to \mathbf{Void}$. To derive a contradiction, it suffices to show $\mathbf{El}(\tau\ (\sigma\ \Omega) < \Omega)$, which is to say that for any f such that $\mathbf{El}(\sigma\ \Omega\ f)$, we have $\mathbf{El}(f\ (\tau\ (\sigma\ \Omega)))$. By the definition of Ω, $\mathbf{El}(\sigma\ \Omega\ f)$ implies that $f \circ \tau \circ \sigma$ is inductive; combining this with the fact that Ω is well-founded, we obtain $\mathbf{El}(f\ (\tau\ (\sigma\ \Omega)))$ as required. □

2.7⋆ *Propositions and propositional truncation*

Throughout this chapter we have considered types as indexed collections (of functions, pairs, natural numbers, codes for other types, etc.) but types can also be regarded, by the famed propositions as types correspondence [How80], as logical propositions in an intuitionistic higher-order logic, as discussed in Section 1.3. In short:

Slogan 2.7.1 (Propositions as types)**.** *Type theory has a logical interpretation in which types are logical propositions, and terms of a given type are proofs of that proposition.*

Definition 2.7.2. We say $\Gamma \vdash A$ type is *inhabited* if there exists a term $\Gamma \vdash a : A$. Thus under Slogan 2.7.1 types are propositions and inhabited types are true propositions.

As the reader may be aware, the propositions as types correspondence extends far beyond the basic type and term judgments: contexts are local hypotheses, **Unit** is the true proposition, **Void** is the false proposition, non-dependent **Π**-types (→,

see Exercise 2.5) are implication, non-dependent Σ-types (×, see Exercise 2.13) are conjunction, and **Π**-types are universal quantification.

To formally substantiate this correspondence, we observe that the rules for **Unit**, **Void**, →-types, and ×-types exactly match the corresponding rules of propositional logic when we replace type-theoretic judgments by logical judgments. For example, the rules governing implication in propositional logic exactly match the formation, introduction, and elimination rules of non-dependent **Π**-types:

$$\frac{p\ \mathit{prop} \qquad q\ \mathit{prop}}{p \to q\ \mathit{prop}} \qquad \frac{\Gamma, p \vdash q\ \mathit{true}}{\Gamma \vdash p \to q\ \mathit{true}} \qquad \frac{\Gamma \vdash p \to q\ \mathit{true} \qquad \Gamma \vdash p\ \mathit{true}}{\Gamma \vdash q\ \mathit{true}}$$

This perfect formal correspondence starts to break down for **Π**-types, because predicate logic consists of two distinct syntactic classes—the logical propositions and predicates on the one hand, and the domains of quantification, or *sorts*, on the other—whereas in type theory both the propositions and the domains of quantification are drawn from a single syntactic class of types. Worse yet, extensional type theory lacks connectives corresponding to logical disjunction and existential quantification!

In this section we will take a closer look at the logical content of type theory, paying close attention to the distinction between propositions and sorts, a distinction which clarifies both of the problems described above. In Section 2.7.1 we propose a refinement to the naïve propositions as types correspondence of Slogan 2.7.1. In Section 2.7.2 we discover a minor but fatal discrepancy between the behaviors of existential quantification (resp., disjunction) in logic and Σ-types (resp., coproduct types) in type theory. In Section 2.7.3 we consider a new type former, *propositional truncation*, which allows us to recover disjunction and existential quantification. Finally, in Section 2.7.4 we discuss the constructive nature of type theory's higher-order logic.

Warning 2.7.3. Although propositional truncation (Section 2.7.3) is well established in the setting of extensional type theory [Hof97; AB04], we—and in our estimation, most authors—do not consider it one of the "canonical" connectives of extensional type theory. The reader may safely skip to Chapter 3 and return to this section in advance of reading Section 5.1.

Not sure who to cite for propositional truncation in this section...

2.7.1 *Propositions as some types*

In predicate logic, universal quantification $\forall x : \tau.\ \phi(x)$ is a proposition when ϕ is a proposition with a free variable x of *sort* τ. Sorts are the collections over which the quantifiers range, and they are grammatically distinct from propositions.

Remark 2.7.4. "Predicate logic" often refers to *single-sorted* predicate logic in which all quantifiers range over a single (anonymous) collection, but one can equally well consider many-sorted predicate logics including "typed" logics whose sorts are the types of the simply-typed lambda calculus [LS88]. For example, ordinary (ZFC) set theory is formally a collection of axioms in single-sorted predicate logic with a binary relation symbol $\in$, where the single sort is the collection of sets. ◇

Under the propositions as types correspondence, types serve both roles: as propositions whose terms are proofs ($p : \mathbf{Eq}(A, a, b)$ is a "proof" of $a = b$) and as sorts whose terms are elements ($n : \mathbf{Nat}$ is an "element" of the collection of natural numbers). When we translate the logical proposition $\forall x : \mathbb{N}.\ x = x$ into the type $\mathbf{\Pi}(\mathbf{Nat}, \mathbf{Eq}(\mathbf{Nat}, \mathbf{q}, \mathbf{q}))$, we think of **Nat** as a sort and $\mathbf{Eq}(\mathbf{Nat}, \mathbf{q}, \mathbf{q})$ as a proposition, but type theory does not make any such distinction. Indeed both arguments of a **Π**-type are just types, and it is no less valid to consider the **Π**-type $\mathbf{\Pi}(\mathbf{Eq}(\mathbf{Nat}, \mathbf{zero}, \mathbf{zero}), \mathbf{Nat})$ whose domain is the equality "proposition" and whose codomain is the natural number "sort."

So how can we tell whether a type is a proposition or a sort? Many types are intrinsically biased toward one of these interpretations. The types **Unit**, **Bool**, and **Nat** are all inhabited and thus "true propositions," but **Bool** and **Nat** have *multiple* inhabitants whereas **Unit** does not. For this reason, rendering the judgment $\Gamma \vdash b : \mathbf{Bool}$ as "**Bool** is true" loses valuable information (which b?), but rendering $\Gamma \vdash a : \mathbf{Unit}$ as "**Unit** is true" does not, suggesting that **Unit** tends toward a proposition whereas **Bool** and **Nat** tend toward sorts. Unfortunately, other connectives are less straightforward; the **Π**-type $\mathbf{\Pi}(\mathbf{Nat}, \mathbf{Eq}(\mathbf{Nat}, \mathbf{q}, \mathbf{q}))$ is the "proposition" that all natural numbers are equal to themselves, but $\mathbf{\Pi}(\mathbf{Nat}, \mathbf{Nat})$ is the "sort" of functions $\mathbb{N} \to \mathbb{N}$.

Following the intuition that a proposition should be a type without multiple inhabitants, we formally define propositions in type theory as types whose terms are all equal to one another:

Definition 2.7.5. A *proposition* is a type $\Gamma \vdash A$ type for which the judgment $\Gamma.A.A[\mathbf{p}] \vdash \mathbf{q}[\mathbf{p}] = \mathbf{q} : A[\mathbf{p}^2]$ holds.

Propositions are also known as *mere propositions* to emphasize that they lack information beyond inhabitation, and as *subsingletons* because they have at most one element. We can revise Slogan 2.7.1 accordingly, at the expense of its catchiness:

Slogan 2.7.6 (Propositions as some types)**.** *Type theory has a logical interpretation in which the logical propositions are types whose terms are all equal, and proofs of a given proposition are terms of the corresponding type (which are unique if they exist).*

Exercise 2.36. Show that **Unit** and **Void** are propositions.

Warning 2.7.7. The type $\Gamma \vdash A$ type being a proposition is *not* equivalent to the cardinality of the set $\mathsf{Tm}(\Gamma, A)$ being at most one. Instead, $\Gamma \vdash A$ type is a proposition if and only if *for all substitutions* $\Delta \vdash \gamma : \Gamma$, $|\mathsf{Tm}(\Delta, A[\gamma])| \leq 1$.

For a counterexample, let us take on faith for the moment that type theory is consistent (Theorem 3.4.8) in the sense that there are no terms $\mathbf{1} \vdash a : \mathbf{Void}$. Then there are no terms of type $\mathbf{1}.\mathbf{U} \vdash \mathbf{El}(\mathbf{q})$ type, because such a term $\mathbf{1}.\mathbf{U} \vdash b : \mathbf{El}(\mathbf{q})$ would induce a term $\mathbf{1} \vdash b[\mathbf{id}.\mathbf{void}] : \mathbf{Void}$. But $\mathbf{1}.\mathbf{U} \vdash \mathbf{El}(\mathbf{q})$ type is not a proposition; if it were, every type encoded in $\mathbf{U}$ would have to be a proposition, again contradicting consistency via the disjointness of **Bool** (Theorem 2.6.3).

Using $\mathbf{\Pi}$-types and **Eq**-types we can internalize the property of being a proposition. For any $\Gamma \vdash A$ type we define $\mathsf{isProp}(A) :\equiv (a\ b : A) \to \mathbf{Eq}(A, a, b)$, or more formally:

$$\Gamma \vdash \mathsf{isProp}(A) :\equiv \mathbf{\Pi}(A, \mathbf{\Pi}(A[\mathbf{p}], \mathbf{Eq}(A[\mathbf{p}^2], \mathbf{q}[\mathbf{p}], \mathbf{q}))) \ \mathsf{type}$$

Exercise 2.37. Show that $\Gamma \vdash A$ type is a proposition in the sense of Definition 2.7.5 if and only if $\Gamma \vdash \mathsf{isProp}(A)$ type is inhabited. (Feel free to use more informal type-theoretic notation for this exercise.)

Exercise 2.38. Show that **Bool** and **Nat** are *not* propositions, in the sense that the types $\Gamma \vdash \mathsf{isProp}(\mathbf{Bool}) \to \mathbf{Void}$ type and $\Gamma \vdash \mathsf{isProp}(\mathbf{Nat}) \to \mathbf{Void}$ type are inhabited. (Hint: adapt the proof of Theorem 2.6.3. Feel free to use more informal type-theoretic notation for this exercise.)

Our third source of propositions after the true proposition **Unit** and the false proposition **Void** will be **Eq**-types. In extensional type theory, equalities are sometimes true and sometimes false but are always propositions, due to the η-rule stating that all terms of type $\mathbf{Eq}(A, a, b)$ are judgmentally equal to **refl**.

Lemma 2.7.8. *If* $\Gamma \vdash a, b : A$ *then* $\Gamma \vdash \mathbf{Eq}(A, a, b)$ type *is a proposition.*

Proof. We must show $\Gamma.\mathbf{Eq}(A, a, b).\mathbf{Eq}(A, a, b)[\mathbf{p}] \vdash \mathbf{q}[\mathbf{p}] = \mathbf{q} : \mathbf{Eq}(A, a, b)[\mathbf{p}^2]$. The η-rule for **Eq**-types states that all terms of **Eq**-type are equal to **refl**; in particular, both $\mathbf{q}[\mathbf{p}]$ and $\mathbf{q}$ are equal to **refl** and thus to each other. □

Remark 2.7.9. What does it mean for a proposition to be *false*? Given that a proposition $\Gamma \vdash A$ type is true if it is inhabited, one might imagine that a proposition is false if $\mathsf{Tm}(\Gamma, A)$ is empty—but then no proposition can be false, as even the so-called false proposition **Void** is inhabited in some contexts. Recalling from Section 2.5.1 that **Void** is the "smallest type," it is also the "falsest proposition" in the sense that in any context where **Void** is inhabited, so is every other proposition (by $\mathbf{absurd}(-)$). The correct notion of a proposition $\Gamma \vdash A$ type being false is therefore that $\Gamma \vdash A \to \mathbf{Void}$ type is inhabited, or equivalently that $A \iff \mathbf{Void}$.

It is not a coincidence that the correct notions of being a proposition, being true, and being false are all preserved by substitution and expressible internally in type theory, whereas the incorrect notions of $\mathsf{Tm}(\Gamma, A)$ having cardinality ≤ 1, $= 1$, and $= 0$ satisfy neither of these properties. (See Warning 2.7.7 and Exercise 2.37.) ◇

Whereas $\mathbf{Eq}(A, a, b)$ is a proposition for any A, a, b, the types $\mathbf{\Pi}(A, B)$ and $\Sigma(A, B)$ may or may not be propositions depending on what A, B are; returning to our earlier example, $\mathbf{\Pi}(\mathbf{Nat}, \mathbf{Nat})$ has multiple inhabitants but $\mathbf{\Pi}(\mathbf{Nat}, \mathbf{Eq}(\mathbf{Nat}, \mathbf{q}, \mathbf{q}))$ does not.

Lemma 2.7.10. *If* $\Gamma \vdash A\,\mathsf{type}$, $\Gamma.A \vdash B\,\mathsf{type}$, *and* B *is a proposition, then their dependent product* $\Gamma \vdash \mathbf{\Pi}(A, B)\,\mathsf{type}$ *is a proposition.*

Proof. Unfolding Definition 2.7.5, we must show

$$\Gamma.\mathbf{\Pi}(A, B).\mathbf{\Pi}(A, B)[\mathbf{p}] \vdash \mathbf{q}[\mathbf{p}] = \mathbf{q} : \mathbf{\Pi}(A, B)[\mathbf{p}^2]$$

By the natural isomorphism defining $\mathbf{\Pi}$-types, this condition is equivalent to the two functions being equal when applied to a new variable of type A:

$$\Gamma.\mathbf{\Pi}(A, B).\mathbf{\Pi}(A, B)[\mathbf{p}].A[\mathbf{p}^2] \vdash \mathbf{app}(\mathbf{q}[\mathbf{p}^2], \mathbf{q}) = \mathbf{app}(\mathbf{q}[\mathbf{p}], \mathbf{q}) : B[\mathbf{p}^3.\mathbf{q}]$$

which follows from our assumption that B, hence any $B[\gamma]$, is a proposition. □

Corollary 2.7.11. *For any* $\Gamma \vdash A\,\mathsf{type}$, $\Gamma \vdash \mathsf{isProp}(A)\,\mathsf{type}$ *is a proposition.*

Proof. We must show that

$$\Gamma \vdash \mathbf{\Pi}(A, \mathbf{\Pi}(A[\mathbf{p}], \mathbf{Eq}(A[\mathbf{p}^2], \mathbf{q}[\mathbf{p}], \mathbf{q})))\,\mathsf{type}$$

is a proposition. Applying Lemma 2.7.10 twice, it suffices to show that

$$\Gamma.A.A[\mathbf{p}] \vdash \mathbf{Eq}(A[\mathbf{p}^2], \mathbf{q}[\mathbf{p}], \mathbf{q})\,\mathsf{type}$$

is a proposition, which is immediate by Lemma 2.7.8. □

The requirements for $\Sigma(A, B)$ to be a proposition are more severe than for $\mathbf{\Pi}(A, B)$: both A and B must be propositions. Worse yet, $A + B$ may not be a proposition even when both A and B are propositions!

Exercise 2.39. Find propositions $\Gamma \vdash A\,\mathsf{type}$ and $\Gamma \vdash B\,\mathsf{type}$ such that $A + B$ is not a proposition, in the sense that $\Gamma \vdash \mathsf{isProp}(A + B) \to \mathbf{Void}\,\mathsf{type}$ is inhabited. Can you find an additional condition on A and B that ensures $A + B$ is a proposition?

Exercise 2.40. Show that if $\Gamma \vdash A\,\mathsf{type}$, $\Gamma.A \vdash B\,\mathsf{type}$, and both A and B are propositions, then their dependent sum $\Gamma \vdash \Sigma(A, B)\,\mathsf{type}$ is a proposition.

Exercise 2.41. Find a type $\Gamma \vdash A\,\mathrm{type}$ and a proposition $\Gamma.A \vdash B\,\mathrm{type}$ such that $\Gamma \vdash \Sigma(A,B)\,\mathrm{type}$ is not a proposition, in the sense that $\Gamma \vdash \mathsf{isProp}(\Sigma(A,B)) \to \mathbf{Void}\,\mathrm{type}$ is inhabited.

In the case that $\Gamma.A \vdash B\,\mathrm{type}$ is a proposition and $\Gamma \vdash A\,\mathrm{type}$ is not, $\Sigma(A,B)$ is not a proposition but rather the *subtype* of A on which the predicate B holds, because its elements are pairs of an element $a : A$ and a proof $b : B[\mathbf{id}.a]$ that B holds on a.

Exercise 2.42. Suppose that $\Gamma \vdash A\,\mathrm{type}$, $\Gamma.A \vdash B\,\mathrm{type}$, and B is a proposition. Show that "$\Sigma(A,B)$ is the subtype of A on which B holds" in the sense that internally to type theory, (1) there is an injective function $\Sigma(A,B) \to A$, and (2) $B(a)$ holds if and only if $a : A$ is in the image of that function. What happens if B is not a proposition?

We may summarize our results as follows:

Corollary 2.7.12. *The "propositions as some types" interpretation of type theory (Slogan 2.7.6) supports the following logical connectives:*

- **Unit** *is the true proposition.*
- **Void** *is the false proposition.*
- $\mathbf{Eq}(A,a,b)$ *is the proposition* $a = b$ *for sort* A.
- *If* B *is a proposition then* $\mathbf{\Pi}(A,B)$ *is the proposition* $\forall x : A.\, B(x)$ *for sort* A.
- *If* A *and* B *are propositions then* $A \to B$ *is the proposition* $A \implies B$.
- *If* A *and* B *are propositions then* $A \times B$ *is the proposition* $A \wedge B$.
- *If* A *is a proposition then* $A \to \mathbf{Void}$ *is the proposition* $\neg A$.

Proof. Each bullet point above asserts both that the given type is a proposition in the sense of Definition 2.7.5, and that its rules match those of a particular logical connective. We have proven most of these claims above and leave the rest to the reader. □

Although Corollary 2.7.12 spans most of the connectives of predicate logic, it omits two connectives, namely disjunction and existential quantification. Coproduct types are a natural candidate for logical disjunction: by +-introduction, if either A or B are inhabited then $A + B$ is inhabited, and by +-elimination, given functions $A \to C$ and $B \to C$ we may construct a function $A + B \to C$. Likewise Σ-types are a natural candidate for existential quantification: by Σ-introduction, if $a : A$ and $B(a)$ is inhabited then $\Sigma(A,B)$ is inhabited, and by Σ-elimination, if $\Sigma(A,B)$ is inhabited then there is an $a : A$ for which $B(a)$ is inhabited.

Unfortunately, these types are not propositions; as we have seen in Exercises 2.39 and 2.41, it is neither the case that $A + B$ is a proposition whenever A and B are propositions, nor that $\Sigma(A, B)$ is a proposition whenever B is a proposition. We are faced with two possibilities: is "propositions as some types" too restrictive, or is it genuinely incorrect to use Σ-types (resp., coproduct types) as existential quantifiers (resp., disjunction)? We will find in Section 2.7.2 that it is the latter.

Universes of propositions On a more positive note, in light of Corollary 2.7.11 we can use Σ-types to define a *hierarchy of universes of propositions* Prop_i as the subtypes of $\mathbf{U}_i$ (Exercise 2.42) spanned by codes of propositions:

$$\frac{\vdash \Gamma\,\mathsf{cx}}{\Gamma \vdash \mathsf{Prop}_i :\equiv \Sigma(\mathbf{U}_i, \mathsf{isProp}(\mathbf{El}(\mathbf{q}))) : \mathbf{U}_{i+1}} \Rightarrow \qquad \frac{\Gamma \vdash p : \mathsf{Prop}_i}{\Gamma \vdash \mathsf{Prf}_i(p) :\equiv \mathbf{El}(\mathbf{fst}(p))\,\mathsf{type}} \Rightarrow$$

Note that each $(\mathsf{Prop}_i, \mathsf{Prf}_i(-))$ is a universe in the sense of Notation 2.6.4.

To close Prop_i under logical connectives we simply combine the closure conditions of $\mathbf{U}_i$ from Section 2.6 with the closure conditions of propositions in Corollary 2.7.12.

Exercise 2.43. Provide definitions of $\mathbf{bot}_i$, $\mathbf{and}_i(a, b)$, and $\mathbf{forall}_i(a, b)$ satisfying:

$$\frac{\vdash \Gamma\,\mathsf{cx}}{\Gamma \vdash \mathbf{bot}_i : \mathsf{Prop}_i} \Rightarrow \qquad \frac{\vdash \Gamma\,\mathsf{cx}}{\Gamma \vdash \mathsf{Prf}_i(\mathbf{bot}_i) = \mathbf{Void}\,\mathsf{type}} \Rightarrow$$

$$\frac{\Gamma \vdash a, b : \mathsf{Prop}_i}{\Gamma \vdash \mathbf{and}_i(a, b) : \mathsf{Prop}_i} \Rightarrow \qquad \frac{\Gamma \vdash a, b : \mathsf{Prop}_i}{\Gamma \vdash \mathsf{Prf}_i(\mathbf{and}_i(a, b)) = \mathsf{Prf}_i(a) \times \mathsf{Prf}_i(b)\,\mathsf{type}} \Rightarrow$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma.\mathbf{El}_i(a) \vdash b : \mathsf{Prop}_i}{\Gamma \vdash \mathbf{forall}(a, b) : \mathsf{Prop}_i} \Rightarrow$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma.\mathbf{El}_i(a) \vdash b : \mathsf{Prop}_i}{\Gamma \vdash \mathsf{Prf}_i(\mathbf{forall}(a, b)) = \mathbf{\Pi}(\mathbf{El}_i(a), \mathsf{Prf}_i(b))\,\mathsf{type}} \Rightarrow$$

Notation 2.7.13. Mirroring our notation for type universes, we write $\mathsf{Prop} :\equiv \mathsf{Prop}_0$ and $\mathsf{Prf}(-) :\equiv \mathsf{Prf}_0(-)$. We will also suppress universe levels when they are immaterial to the point at hand.

As with $\mathbf{U}_i$, having a type of propositions allows us to formulate logical notions and principles that quantify over or otherwise refer to propositions. For example, we will say that a *predicate over the type* A is a function $P : A \to \mathsf{Prop}$, and a *binary relation over* A *and* B is a function $R : A \times B \to \mathsf{Prop}$. Because quantification over

Prop may be nested arbitrarily deeply, the logical interpretation of type theory extends automatically to higher-order logic.

Remark 2.7.14. It is worth asking whether one really needs a *hierarchy* of universes of propositions. On the one hand, such a hierarchy $\mathsf{Prop}_0, \mathsf{Prop}_1, \ldots$ falls out naturally from our definition of each Prop_i as a subtype of $\mathbf{U}_i$. On the other hand, recall that we introduced the hierarchy $\mathbf{U}_0, \mathbf{U}_1, \ldots$ in the first place to approximate the idea that $\mathbf{U}$ is a "type of all types, including itself" without falling victim to Girard's paradox; but Prop should not include itself for the much simpler reason that it is not a proposition!

In stark contrast to the situation with type universes, it is perfectly consistent to have a single type Prop containing codes for all propositions regardless of their universe level; these single universes are known as *impredicative* universes of propositions, in contrast to the *predicative* hierarchy described above. Although they constitute an extension to the type theories discussed in this book, they are widely (but not universally) accepted, appearing for instance in the Rocq and Lean proof assistants. We will discuss them in more depth in Section 5.1. ◇

2.7.2 *The illusion of choice*

Although $\Sigma(A, B)$ appears to satisfy the logical rules governing existential quantification, it is not a proposition in the sense of Definition 2.7.5 and thus cannot be part of the "propositions as some types" interpretation of type theory. We now illustrate why it is problematic that $\Sigma(A, B)$ is not a proposition when B is a predicate, by considering a naïve type-theoretic translation of the axiom of choice using Σ-types as existentials.

One formulation of the axiom of choice is that for any one-to-many binary relation R between sorts τ and σ, there exists a function $f : \tau \to \sigma$ satisfying $\forall x : \tau. R(x, f(x))$. Such a function is often called a *choice function*, in the sense that it chooses for each $x : \tau$ one of the (possibly many) $y : \sigma$ to which x must be related.

Definition 2.7.15. In typed higher-order logic, the *axiom of choice* is the proposition that for any sorts τ, σ and any predicate R over $\tau \times \sigma$,

$$(\forall x : \tau. \exists y : \sigma. R(x, y)) \implies (\exists f : \tau \to \sigma. \forall x : \tau. R(x, f(x)))$$

Suppose we follow the logical interpretation of type theory described in Corollary 2.7.12, and moreover interpret $\Sigma(A, B)$ as $\exists x : A. B(x)$. The result is a type which states that for all types A, B and for all relations $P : A \times B \to \mathsf{Prop}$,

$$\begin{aligned}\mathsf{NaiveChoice} :\equiv\ & ((a : A) \to \textstyle\sum_{b:B} \mathsf{Prf}(P(a, b))) \to \\ & (\textstyle\sum_{f:A\to B}(a : A) \to \mathsf{Prf}(P(a, f(a))))\end{aligned}$$

Note that the types A and B correspond respectively to the sorts τ and σ in Definition 2.7.15, and are not assumed to be propositions. As a result, neither the antecedent nor the consequent of NaiveChoice is in general a proposition.

Lemma 2.7.16. NaiveChoice *is inhabited in type theory.*

Proof. Suppose that $F : (a : A) \to \sum_{b:B} \mathsf{Prf}(P(a,b))$. We must construct a term of type $\sum_{f:A\to B}(a : A) \to \mathsf{Prf}(P(a, f(a)))$. By Σ-introduction, it suffices to exhibit a term $f : A \to B$, for which we choose $f(a) = \mathbf{fst}(F(a))$, as well as a term $g : (a : A) \to \mathsf{Prf}(P(a, f(a)))$, for which $g(a) = \mathbf{snd}(F(a))$ is sufficient. Putting it all together,

$$\lambda F \to ((\lambda a \to \mathbf{fst}(F\ a)), (\lambda a \to \mathbf{snd}(F\ a))) : \mathsf{NaiveChoice}$$ □

Traditionally, the force of the axiom of choice is that from the mere fact—the *logical proposition*—that for every x there exists some y with $R(x, y)$, one can obtain an actual function—data in the *sort* $\tau \to \sigma$—that concretely chooses one such y for each x. Regardless of how one feels about the axiom of choice, it is clear that the proof of Lemma 2.7.16 is doing something altogether different: it directly extracts the choice of $b : B$ from the "proof" of the antecedent $F : (a : A) \to \sum_{b:B} \mathsf{Prf}(P(a,b))$ by sending each $a : A$ to the first projection of $F(a)$.

In other words, our proof of NaiveChoice relies essentially on the fact that we can extract non-trivial data from the "proof" $F(a)$ of an existential. Rereading Lemma 2.7.16, NaiveChoice simply states that from a pair-valued function $a \mapsto (b, p)$ we can obtain a pair of functions $a \mapsto b$ and $a \mapsto p$—hardly the axiom of choice!

Remark 2.7.17. The type NaiveChoice is sometimes known as the *type-theoretic axiom of choice* despite being neither an axiom nor a choice principle. We concede however that it is type-theoretic. ◇

What has gone wrong? The antecedent of NaiveChoice is much stronger than the antecedent of the axiom of choice: from a term of type $\sum_{b:B} \mathsf{Prf}(P(a,b))$ we can project out an element b of sort B, whereas in higher-order logic, knowing the proposition $\exists b : B.\, P(a, b)$ does not license one to obtain a concrete witness of *sort* B. In logic, one can assume that such a $b : B$ exists, but only in service of proving another *proposition*. Put simply, inhabitation of $\sum_{b:B} \mathsf{Prf}(P(a,b))$ in type theory is too informative. It contains too much data; it is not a proposition.

2.7.3 *Truncating types to propositions*

Now that we have seen why Σ-types are not existential quantifiers, we turn to the problem of correctly capturing existential quantification in type theory. After deriv-

ing a "mapping out" characterization of existentials, we will present a simpler but equally expressive connective known variously as *propositional truncation*, *squash types*, or *bracket types* [UF13; Hof97; Con+85; AB04] which we will consider an optional extension to the extensional type theory defined in this chapter.

2.7.3.1 Existentials in type theory

In Section 2.7.2 we identified two key discrepancies between Σ-types and existential quantification. First, $\Sigma(A, B)$ is not a proposition even when B is a proposition. Secondly, given a proof of $\exists x : A.\, B(x)$, the witness (first projection) of sort A should be accessible only for purposes of inhabiting other propositions, not sorts.

Recalling Slogan 2.5.5, to specify an existential quantification type $\exists(A, B)$ in type theory we must first decide whether to characterize the maps in or out of that type. The second discrepancy above suggests that we need to restrict the maps *out* of existentials—what one can do with a term of type $\exists(A, B)$—so we will start there.

The formation rule for $\exists(A, B)$ is identical to that of $\Sigma(A, B)$. Naturally in Γ,

$$\exists_\Gamma : \left(\textstyle\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)\right) \to \mathsf{Ty}(\Gamma) \qquad (\text{✎})$$

As for the mapping out property, implications $\exists(A, B) \to C$ should correspond to proofs of C under the assumption that there is a witness of sort A and an inhabitant of B at that witness. Phrased as a natural isomorphism, we require that naturally in Γ and for every $\Gamma \vdash A$ type, $\Gamma.A \vdash B$ type, and $\Gamma \vdash C$ type *where C is a proposition*,

$$\rho_{\Gamma,A,B,C} : \mathsf{Tm}(\Gamma.\exists(A, B), C[\mathbf{p}]) \cong \mathsf{Tm}(\Gamma.A.B, C[\mathbf{p}^2]) \qquad (\text{✎})$$

Note that ρ is exactly the (non-dependent case of the) mapping out property satisfied by $\Sigma(A, B)$ as shown in Exercise 2.23, the only difference being that for $\exists$-types we restrict C to be a proposition. In particular, this restriction prevents us from setting $C = A$ and thence deriving a first projection map $\exists(A, B) \to A$.

Remark 2.7.18. The naturality requirement above is superfluous; the family of isomorphisms $\rho_{\Gamma,A,B,C}$ is necessarily natural because C is a proposition and thus all maps into $\mathsf{Tm}(\Gamma.A.B, C[\mathbf{p}^2])$ must be equal. ◇

Now that we have suitably restricted the maps out of $\exists$-types, we complete our specification by requiring that $\exists(A, B)$ is a proposition for every A, B, for example by asserting that any two terms of type $\exists(A, B)$ are equal.

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash p, q : \exists(A, B)}{\Gamma \vdash p = q : \exists(A, B)}\ \text{✎}$$

Given all the type formers now at our disposal, one might wonder whether there is some clever way to encode ∃-types in terms of Σ-types and perhaps other connectives from this chapter. Swan [Swa25] has shown recently that the answer is no. There are however quite a few different and reasonable extensions to extensional type theory which allow one to define a type satisfying the specification of ∃-types, including of course the provisional rules for $\exists(A, B)$ themselves, or an impredicative universe of propositions (Remark 2.7.14), or our next topic of discussion, propositional truncation.

Exercise 2.44. Following the pattern of ∃-types, write a set of rules for *disjunction types* $A \vee B$. The type $A \vee B$ should be similar to $A + B$ except that, like ∃-types, it is a proposition and its mapping out property is restricted to maps into propositions.

Deriving introduction and dependent elimination Our specification of ∃-types departs from Slogan 2.5.5 in several important ways: we did not specify any introduction rules, the elimination principle only describes non-dependent maps out of $\exists(A, B)$, and the elimination principle does not state that a particular substitution map is an isomorphism. For the curious reader, we now explain how all of these properties follow from our more compact specification of ∃-types.

We start with the introduction rule. Setting $C :\equiv \exists(A, B)$ in our mapping out property, we have a natural isomorphism:

$$\rho_{\Gamma,A,B,\exists(A,B)} : \mathsf{Tm}(\Gamma.\exists(A,B), \exists(A,B)[\mathbf{p}]) \cong \mathsf{Tm}(\Gamma.A.B, \exists(A,B)[\mathbf{p}^2])$$

The unique element of the left-hand side is the variable $\mathbf{q} \in \mathsf{Tm}(\Gamma.\exists(A,B), \exists(A,B)[\mathbf{p}])$, so $\rho_{\Gamma,A,B,\exists(A,B)}(\mathbf{q})$ must be the unique element of $\mathsf{Tm}(\Gamma.A.B, \exists(A,B)[\mathbf{p}^2])$, which by substitution induces a function

$$\mathbf{epair} : (\textstyle\sum_{a \in \mathsf{Tm}(\Gamma,A)} \mathsf{Tm}(\Gamma, B[\mathbf{id}.a])) \to \mathsf{Tm}(\Gamma, \exists(A,B))$$

corresponding to the introduction rule for ∃-types. Note that there is a unique such function because $\Gamma \vdash \exists(A, B)$ type is a proposition.

The full elimination principle is more challenging. We must show that for any proposition $\Gamma.\exists(A, B) \vdash C$ type, the following map is a natural isomorphism:

$$(\mathbf{p}^2.\mathbf{epair}(\mathbf{q}[\mathbf{p}], \mathbf{q}))^* : \mathsf{Tm}(\Gamma.\exists(A,B), C) \cong \mathsf{Tm}(\Gamma.A.B, C[\mathbf{p}^2.\mathbf{epair}(\mathbf{q}[\mathbf{p}], \mathbf{q})])$$

As before, because C is a proposition this is the only such function, so we can forget about the particular choice of map and construct any such isomorphism whatsoever.

To reduce this principle to the non-dependent principle stated earlier, we observe that all possible dependencies on a proposition (in this case, $\Gamma \vdash \exists(A, B)$ type) are

equal to one another. That is, for any type $\Gamma.P \vdash C\,\mathsf{type}$ depending on a proposition $\Gamma \vdash P\,\mathsf{type}$, we have isomorphisms

$$\mathsf{Tm}(\Gamma.P.P[\mathbf{p}], C[\mathbf{p}^2.\mathbf{q}[\mathbf{p}]]) \cong \mathsf{Tm}(\Gamma.P.P[\mathbf{p}], C[\mathbf{p}^2.\mathbf{q}]) \cong \mathsf{Tm}(\Gamma.P, C)$$

We may therefore replace the dependency of C on the copy of $\exists(A,B)$ in the context with a "local" dependency introduced on the right-hand side by a Σ-type:

$$\mathsf{Tm}(\Gamma.\exists(A,B), C) \cong \mathsf{Tm}(\Gamma.\exists(A,B), \Sigma(\exists(A,B), C)[\mathbf{p}])$$

and similarly remove the dependency in the codomain of $(\mathbf{p}^2.\mathbf{epair}(\mathbf{q}[\mathbf{p}], \mathbf{q}))^*$:

$$\mathsf{Tm}(\Gamma.A.B, C[\mathbf{p}^2.\mathbf{epair}(\mathbf{q}[\mathbf{p}], \mathbf{q})]) \cong \mathsf{Tm}(\Gamma.A.B, \Sigma(\exists(A,B), C)[\mathbf{p}^2])$$

We complete the argument by composing the above isomorphisms with the isomorphism $\rho_{\Gamma,A,B,\Sigma(\exists(A,B),C)}$, noting that $\Gamma \vdash \Sigma(\exists(A,B), C)[\mathbf{p}^2]\,\mathsf{type}$ is a proposition.

Exercise 2.45. Complete this argument by defining the omitted isomorphisms and checking that they compose to an isomorphism between the required sets.

2.7.3.2 Propositional truncation

The rules for $\exists$-types differ from the rules of Σ-types in two essential ways: they assert that $\exists(A,B)$ is a proposition, and they restrict the mapping out property to propositions. It turns out to be useful to isolate the process of replacing any type A with a proposition that maps out only into other propositions. We call this type the *propositional truncation* of A; it is, in a precise sense, the proposition that most closely approximates the type A.

We notate the propositional truncation of A as $\mathbf{Trunc}(A)$, although other popular notations include $[A]$ [AB04] and $\|A\|$ [UF13]. Its formation rule is straightforward:

$$\frac{\Gamma \vdash A\,\mathsf{type}}{\Gamma \vdash \mathbf{Trunc}(A)\,\mathsf{type}}\ \text{✎} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type}}{\Delta \vdash \mathbf{Trunc}(A)[\gamma] = \mathbf{Trunc}(A[\gamma])\,\mathsf{type}}\ \text{✎}$$

The remaining rules for $\mathbf{Trunc}(A)$ are similar to those of $\exists$-types but stripped of any resemblance to Σ-types. First, for every $a : A$ we have a term $\mathbf{seal}(a) : \mathbf{Trunc}(A)$. Secondly, $\mathbf{Trunc}(A)$ is a proposition. Finally, for every proposition $\Gamma \vdash C\,\mathsf{type}$, we have an isomorphism $\mathsf{Tm}(\Gamma.\mathbf{Trunc}(A), C[\mathbf{p}]) \cong \mathsf{Tm}(\Gamma.A, C[\mathbf{p}])$.

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathbf{seal}(a) : \mathbf{Trunc}(A)}\ \text{✎} \qquad \frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash p, q : \mathbf{Trunc}(A)}{\Gamma \vdash p = q : \mathbf{Trunc}(A)}\ \text{✎}$$

$$\frac{\begin{array}{c}\Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash C\,\mathsf{type} \\ \Gamma.C.C[\mathbf{p}] \vdash \mathbf{q}[\mathbf{p}] = \mathbf{q} : C[\mathbf{p}^2] \qquad \Gamma \vdash a : \mathbf{Trunc}(A) \qquad \Gamma.A \vdash c : C[\mathbf{p}]\end{array}}{\Gamma \vdash \mathbf{open}(a, c) : C}\ \text{✎}$$

Exercise 2.46. Why does the $\mathbf{open}(-,-)$ rule give rise to an isomorphism

$$\mathsf{Tm}(\Gamma.\mathbf{Trunc}(A), C[\mathbf{p}]) \cong \mathsf{Tm}(\Gamma.A, C[\mathbf{p}])$$

for every proposition $\Gamma \vdash C$ type, and not just a map in the reverse direction? Where are the β and η principles? And where are the substitution rules for term formers?

If we take **Trunc**-types as primitive, we can use them to define a type $\exists'(A,B)$ satisfying the rules of $\exists$-types from Section 2.7.3.1:

$$\exists'(A,B) :\equiv \mathbf{Trunc}(\Sigma(A,B))$$

Clearly $\exists'(A,B)$ is a proposition and has the correct natural formation rule. As for the mapping out property of $\exists$-types, suppose $\Gamma \vdash C$ type is a proposition. Then:

$$\begin{aligned}
&\mathsf{Tm}(\Gamma.\exists'(A,B), C[\mathbf{p}]) \\
&\cong \mathsf{Tm}(\Gamma.\Sigma(A,B), C[\mathbf{p}]) && \text{by mapping out for } \mathbf{Trunc}\text{-types} \\
&\cong \mathsf{Tm}(\Gamma.A.B, C[\mathbf{p}^2]) && \text{by mapping out for } \Sigma\text{-types}
\end{aligned}$$

Exercise 2.47. Conversely, if we take $\exists$-types as primitive, we can define a type $\mathbf{Trunc}'(A)$ satisfying the rules of $\mathbf{Trunc}(A)$, namely $\mathbf{Trunc}'(A) :\equiv \exists(A, \mathbf{Unit})$. Show that $\mathbf{Trunc}'(A)$ satisfies the same mapping out property as $\mathbf{Trunc}(A)$, using only the mapping out property of $\exists$-types.

Exercise 2.48. If we take **Trunc**-types as primitive, we may also define disjunction types as $A \vee B :\equiv \mathbf{Trunc}(A + B)$. Show that this definition satisfies the rules proposed in Exercise 2.44.

Although it is beyond the scope of this book, we note that one can develop a considerable amount of theory about **Trunc**-types [AB04]. For example, they provide us with yet another characterization of propositions, namely as the types A for which A and $\mathbf{Trunc}(A)$ are isomorphic (internally to type theory). In fact:

Lemma 2.7.19. *The type A is a proposition if and only if there exists a retraction of $\mathbf{seal}(-) : A \to \mathbf{Trunc}(A)$, i.e., a function $f : \mathbf{Trunc}(A) \to A$ such that $\mathbf{seal}(-)$ followed by f is the identity $A \to A$. In this case f is necessarily an isomorphism.*

It follows that $\mathbf{Trunc}(\mathbf{Trunc}(A))$ is isomorphic to $\mathbf{Trunc}(A)$, and in fact that propositional truncation forms an idempotent monad.

Finally, if we take **Trunc**-types as primitive, it is natural to also close each type universe $\mathbf{U}_i$ under propositional truncation with the following rules:

$$\frac{\Gamma \vdash a : \mathbf{U}_i}{\Gamma \vdash \mathbf{trunc}_i(a) : \mathbf{U}_i} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}_i}{\Delta \vdash \mathbf{trunc}_i(a)[\gamma] = \mathbf{trunc}_i(a[\gamma]) : \mathbf{U}_i}$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}_i(\mathbf{trunc}_i(a)) = \mathbf{Trunc}(\mathbf{El}_i(a))\ \mathsf{type}}$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i}{\Gamma \vdash \mathbf{lift}_i(\mathbf{trunc}_i(a)) = \mathbf{trunc}_{i+1}(\mathbf{lift}_i(a)) : \mathbf{U}_{i+1}}$$

Recalling that we defined the universes of propositions Prop_i as the subtypes of $\mathbf{U}_i$ spanned by codes of propositions, the above rules imply that each Prop_i is closed under propositional truncation of types in $\mathbf{U}_i$ and thus induce a map $\mathbf{U}_i \to \mathsf{Prop}_i$ sending each type in $\mathbf{U}_i$ to its propositional truncation in Prop_i. By replaying our earlier constructions at the level of codes, they also imply that every Prop_i is closed under existential quantification and disjunction in the appropriate sense.

Advanced Remark 2.7.20. If we regard $\mathbf{U}_i$ and Prop_i as categories, then the aforementioned propositional truncation map $\mathbf{U}_i \to \mathsf{Prop}_i$ is the left adjoint to the inclusion functor from propositions to types, and thus that propositional truncation exhibits Prop_i as a *reflective subcategory* of $\mathbf{U}_i$. The substitution rules for **Trunc**-types ensure that this reflection extends to each slice and commutes with pullbacks.

Chasing this thread further, the condition of a type theory admitting **Trunc**-types is a syntactic analogue of its category of closed types being *regular*, in the sense of having a stable factorization system of effective epimorphisms followed by a monomorphism. The reader may consult Awodey and Bauer [AB04] for further discussion, including the connection between intuitionistic predicate logic and type theory with **Trunc**-types. ⋄

2.7.4 *The logic of type theory*

We close our exploration of propositions in extensional type theory by returning to the discussion of choice principles from Section 2.7.2. Suppose that we add **Trunc**-types to extensional type theory and properly formulate the axiom of choice in terms of "mere existence" (∃-types or truncated Σ-types):

$$\begin{aligned}\mathsf{Choice} := (A\ B : \mathbf{U}) \to (P : \mathbf{El}(A) \times \mathbf{El}(B) \to \mathsf{Prop}) \to \\ ((a : \mathbf{El}(A)) \to \mathbf{Trunc}(\textstyle\sum_{b:\mathbf{El}(B)} \mathsf{Prf}(P(a,b)))) \to\end{aligned}$$

$$\mathbf{Trunc}(\textstyle\sum_{f:A\to B}(a:A) \to \mathsf{Prf}(P(a, f(a))))$$

Is Choice inhabited in type theory?

Theorem 2.7.21. *The proposition* Choice *is* independent *of extensional type theory with* **Trunc***-types, in the sense that neither* Choice *nor* Choice → **Void** *is inhabited.*

Remark 2.7.22. We will not prove Theorem 2.7.21 or any of the other independence theorems in this section; however, in Section 6.6 we will discuss how these theorems can be derived from models of type theory in various topoi. ◇

Theorem 2.7.21 demonstrates that extensional type theory with **Trunc**-types does not agree with the higher-order logic of classical sets, in which the axiom of choice holds. In fact, type theory is a *constructive* logic, in the sense that it does not admit (nor does it refute) several notable principles of classical logic[15] such as the axiom of choice, the law of excluded middle (LEM), and double-negation elimination (DNE):

$$\mathsf{LEM} := (P : \mathsf{Prop}) \to (\mathsf{Prf}(P) \vee (\mathsf{Prf}(P) \to \mathbf{Void}))$$
$$\mathsf{DNE} := (P : \mathsf{Prop}) \to ((\mathsf{Prf}(P) \to \mathbf{Void}) \to \mathbf{Void}) \to \mathsf{Prf}(P)$$

Theorem 2.7.23. *The propositions* LEM *and* DNE *imply one another, and are independent of extensional type theory with* **Trunc***-types.*

Surprisingly, type theory is also consistent with some principles that are *false* in classical logic, such as the principle that every function **Nat** → **Nat** is computable.

Theorem 2.7.24. *Fix some Gödel numbering of Turing machines, and let* Church's thesis *be the proposition that for every* f : **Nat** → **Nat**, *there merely exists some* n : **Nat** *such that the Turing machine encoded by* n *computes the function* f. *Church's thesis is independent of type theory.*

Church's thesis is incompatible with the law of excluded middle. Using excluded middle one can easily define a function f : **Nat** → **Nat** that sends every Turing machine code to 0 if the encoded machine halts and 1 otherwise; by the standard halting argument, f cannot be computed by a Turing machine.

The flexibility of type theory to be extended by a wide range of axioms, both classical and anti-classical, allows mathematicians to use it as a powerful domain-specific logic for reasoning *synthetically* about certain classes of objects that are difficult to manipulate explicitly. In Section 5.2 we will encounter examples of synthetic reasoning in homotopy type theory.

[15] Note that there are several distinct senses in which a logic may be constructive; see Remark 2.7.25.

Remark 2.7.25. Constructivity in the sense of omitting classical principles—and thus being compatible with a range of classical and anti-classical axioms—is sometimes known as *neutral constructivism*. Constructivity can also refer to more opinionated reasoning systems, including *Brouwerian intuitionism*, in which (for instance) all functions from the real numbers to the real numbers are continuous, and *Russian constructivism*, which admits recursion-theoretic principles such as Church's thesis. ◇

Further reading

The literature on type theory is unfortunately neither notationally nor conceptually coherent, particularly regarding syntax and how it is defined. We summarize a number of important references that most closely match the perspective outlined in this book; note however that many references will agree in some ways and differ in others.

Historical references Nearly all of the ideas in this chapter can be traced back in some form to the philosopher Per Martin-Löf, whose collected works are available in the GitHub repository `michaelt/martin-lof`. Over the decades, Martin-Löf has considered many different variations on type theory; the closest to our presentation are his notes on substitution calculus [Mar92] and the "Bibliopolis book" presenting what is now called extensional type theory [Mar84b]. For a detailed philosophical exploration of the *judgmental methodology* that types internalize judgmental structure, see his "Siena lectures" [Mar96]. Finally, the book *Programming in Martin-Löf's Type Theory* [NPS90] remains one of the best pedagogical introductions to type theory as formulated in Martin-Löf's logical framework.

Syntax of dependent type theory The presentation of type theory most closely aligned to ours can be found in the second author's Ph.D. thesis [Gra23, Chapter 2]. Another valuable reference is Hofmann's *Syntax and Semantics of Dependent Types* [Hof97, Sections 1 and 2]. Hofmann is very careful in his definition of syntax, but the technical details of capture-avoiding substitution and presyntax have largely been supplanted by subsequent work on logical frameworks, so we suggest that readers gloss over these technical details.

Logical frameworks In this book we have attempted to largely sidestep the question of what constitutes a valid collection of inference rules. The mathematics of syntax can and has occupied entire books, but in short, the natural families of constants and isomorphisms considered in this chapter can be formulated precisely in systems known as *logical frameworks*. A good introduction to logical frameworks is the seminal work of Harper, Honsell, and Plotkin [HHP93] on the Edinburgh Logical Framework, in which object-level judgments can be encoded as meta-level types.

For logical frameworks better suited to defining dependent type theory in particular, we refer readers to Appendix B, where we give a terse account of a subset of Cartmell's *generalized algebraic theories* [Car86] known as *quotient inductive-inductive types* [AK16; Dij17; KKA19; Kov22]. For logical frameworks specifically designed to accommodate the binding and substitution of dependent type theory, we refer the reader to the Ph.D. theses of Haselwarter [Has21] and Uemura [Uem21].

This material will be published by Cambridge University Press as *Principles of Dependent Type Theory* by Carlo Angiuli and Daniel Gratzer. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © Carlo Angiuli and Daniel Gratzer 2024

3 *Metatheory and implementation*

In Chapter 2 we carefully defined Martin-Löf type theory as a formal mathematical object: a kind of "algebra" of indexed sets (of types and terms) equipped with various operations. We believe this perspective is essential to understanding both the *what* and the *why* of type theory, providing both a precise definition that can be unfolded into inference rules, as well as an explanation of what these rules intend to axiomatize.

This perspective is not, however, how most users of type theory interact with it. Most users of type theory interact with *proof assistants*, software systems for interactively developing and verifying large-scale proofs in type theory. Even when type theorists work on paper rather than on a computer, many of the conveniences of proof assistants bleed into their informal notation. Indeed, in Chapter 1 we used definitions, implicit arguments, data type declarations, and pattern matching without a second thought.

Although this book focuses on theoretical rather than practical considerations, it is impossible to discuss the design space of type theory without discussing the pragmatics of proof assistants, as these have exerted a profound influence on the theory. Our goal in this chapter is to explain how to square our mathematical notion of type theory with (idealized) implementations[1] of type theory, and to discover and unpack the substantial constraints that the latter must place on the former.

In this chapter In Section 3.1 we axiomatize the core functionality of proof assistants in terms of algorithmic elaboration judgments, and outline a basic implementation. In Section 3.2 we continue to refine our implementation, taking a closer look at how the equality judgments of type theory impact elaboration, and the metatheoretic properties we need equality to satisfy. In Section 3.3 we consider how to extend our elaborator to account for definitions. In Section 3.4 we discuss other metatheorems of type theory and their relationship to program extraction. In Section 3.5 we construct a set-theoretic model of extensional type theory and explore some of its metatheoretic consequences—including a counterexample to one of the properties discussed in Section 3.2. Finally, in Section 3.6 we disprove a second important metatheoretic property, leading us to consider alternatives to extensional type theory (Chapter 2) in Chapters 4 and 5.

[1] At the end of this chapter, we provide some pointers to literature and implementations specifically geared to readers interested in learning how to actually implement type theory.

Goals of the chapter By the end of this chapter, you will be able to:

- Explain why and how we define type-checking in terms of elaboration.
- Define the consistency, canonicity, normalization, and invertibility metatheorems, and identify why each is important.
- Explain which metatheorems are disrupted by extensional equality, and sketch why.

3.1 *A judgmental reconstruction of proof assistants*

What exactly is the relation between Agda code (or the code in Chapter 1) and the type theory in Chapter 2? Certainly, Rocq and Agda—even without extensions—include many convenience features that the reader would not be surprised to see omitted in a theoretical description of type theory: implicit arguments, typeclasses/instance arguments, libraries, reflection, tactics… For the moment we set aside not only these but even more fundamental features such as data type declarations, pattern matching, and the ability to write definitions, in order to consider the simplest possible "Agda": a *type-checker*. That is, our idealized Agda takes as input two expressions e and τ and *accepts* in the case that e is a closed term of closed type τ, and *rejects* if not.

Slogan 3.1.1. *Proof assistants are fancy type-checkers.*

Remark 3.1.2. For the purposes of this book, "proof assistant" refers only to proof assistants in the style of Rocq, Agda, and Lean. We will not discuss LCF-style systems [GMW79] such as Nuprl [Con+85] and Andromeda [Bau+21], or systems not based on dependent type theory, such as Isabelle [NPW02] or HOL Light [Har09]. ◇

Convenience features of proof assistants are generally aimed at making it easier for users to write down the inputs e and τ, perhaps by allowing some information to be omitted and reconstructed mechanically, or even by presenting a totally different interface for building e and τ interactively or from high-level descriptions. We start our investigation with the most generous possible assumptions—in which e and τ contain all the information we might possibly need, including type annotations—and will find that type-checking is already a startlingly complex problem.

Remark 3.1.3. The title of this section is an homage to *A judgmental reconstruction of modal logic* [PD01], an influential article that reconsiders intuitionistic modal logic under the mindset that *types internalize judgmental structure.* ◇

$$
\begin{array}{lll}
\textit{Pretypes} & \tau ::= & (\texttt{Pi}\ \tau\ \tau) \mid (\texttt{Sigma}\ \tau\ \tau) \mid \texttt{Unit} \mid \texttt{U} \mid (\texttt{El}\ e) \mid \cdots \\
\textit{Preterms} & e ::= & (\texttt{var}\ i) \mid (\texttt{lam}\ \tau\ \tau\ e) \mid (\texttt{app}\ \tau\ \tau\ e\ e) \mid (\texttt{pair}\ \tau\ \tau\ e\ e) \mid \cdots \\
\textit{Indices} & i ::= & 0 \mid 1 \mid 2 \mid \cdots
\end{array}
$$

Figure 3.1: Syntax of pretypes and preterms.

3.1.1 *Type-checking as elaboration*

In Section 2.1 we emphasized that we do *not* assume that the types and terms of type theory are obtained as the "well-formed" subsets of some collections of possibly-ill-formed *pretypes* or *preterms*, nor do we even assume that they are obtained as "$\beta\eta$-equivalence classes" of well-formed-but-unquotiented terms.

Instead, *types* and *terms* are just the elements of the sets $\mathsf{Ty}(\Gamma)$ and $\mathsf{Tm}(\Gamma, A)$, which are defined in terms of each other and the sets Cx and $\mathsf{Sb}(\Delta, \Gamma)$. When we write e.g. $\lambda(b)$, we are naming a particular element of a particular set $\mathsf{Tm}(\Gamma, \mathbf{\Pi}(A, B))$ obtained by applying the "$\mathbf{\Pi}$-introduction" map to $b \in \mathsf{Tm}(\Gamma.A, B)$; in particular, the values of Γ, A, B should be regarded as implicitly present, as they are in Appendix A where we write $\lambda_{\Gamma,A,B}(b)$.

In Chapter 2 we reaped the benefits of this perspective, but it has come time to pay the piper: what, then, is a type-checker supposed to take as input? We certainly cannot say that a type-checker is given "a type A and a term a" because this assumes that A and a are well-formed. Type-checking *cannot be a membership query*; instead, it is a *partial function* from concrete syntax to the sets of genuine types and terms. For an input expression to "type-check" means that it *names* a type/term, not that it "is" one (which is a meta-type error, as types/terms are mathematical objects, and input expressions are strings).

For simplicity we assume that the inputs to type-checkers are not strings but abstract syntax trees (or well-formed formulas) conforming to the simple grammar in Figure 3.1.[2] We call these semi-structured input expressions *pretypes* τ and *preterms* e, and write them as `teletype` s-expressions. In programming language theory, the process of mapping semi-structured input expressions into structured core language terms is known as *elaboration*.

Slogan 3.1.4. *Type-checkers for dependent type theory are elaborators.*

Remark 3.1.5. What is the relationship between features of the concrete syntax of a proof assistant, and features of the core syntax? According to Slogan 3.1.4, the concrete syntax should be seen as "instructions" for building core syntax. These instructions

[2] In other words, we only consider input expressions that successfully parse; expressions that fail to parse (e.g., because their parentheses are mismatched) automatically fail to type-check.

may be very close to or very far from that core syntax, but in either case, new user-facing features should only induce new core primitives when they cannot be (relatively compositionally) accounted for by the existing core language. ◇

Algorithmic judgments Elaborators are partial functions that recursively consume pretypes and preterms (abstract syntax trees) and produce types and terms. In a real proof assistant, types and terms are of course not abstract mathematical entities but elements of some data type, but for our purposes we will imagine an idealized elaborator that outputs elements of $\mathsf{Ty}(\Gamma)$ and $\mathsf{Tm}(\Gamma, A)$. We present this elaborator not as functional programs written in pseudocode, but as *algorithmic judgments* defined by inference rules. Unlike the rules in Chapter 2, these rules are intended to define an algorithm, so we will take care to ensure that any given elaboration judgment can be derived by at most one rule. (In other words, we define our elaborator as a deterministic logic program.)

We have already argued that pretype elaboration should take as input a pretype τ and output a type A, but what about contexts? Just as well-formedness of closed types ($\mathbf{1} \vdash \mathbf{\Pi}(A, B)$ type) refers to well-formedness of open types ($\mathbf{1}.A \vdash B$ type), it is perhaps unsurprising that elaborating closed pretypes requires elaborating open pretypes. However, we note that we do not need or want "precontexts"; we will only descend under binders after successfully elaborating their pretypes. For example, to elaborate (`Pi` τ_0 τ_1) we will first elaborate τ_0 to the closed type A, and only then in context $\mathbf{1}.A$ elaborate τ_1 to B.

Thus our two main algorithmic elaboration judgments are as follows:

1. $\Gamma \vdash \tau$ type $\rightsquigarrow A$ asserts that elaborating the pretype τ relative to $\vdash \Gamma$ cx succeeds and produces the type $\Gamma \vdash A$ type.

2. $\Gamma \vdash e : A \rightsquigarrow a$ asserts that elaborating the preterm e relative to $\vdash \Gamma$ cx and $\Gamma \vdash A$ type succeeds and produces the term $\Gamma \vdash a : A$.

In pseudocode, the first judgment corresponds to a partial function $\texttt{elabTy}(\Gamma, \tau) = A$ with the invariant that if $\vdash \Gamma$ cx and `elabTy` terminates successfully, then $\Gamma \vdash A$ type. Likewise, the second judgment is a partial function $\texttt{elabTm}(\Gamma, A, e) = a$ whose successful outputs are terms $\Gamma \vdash a : A$.

Elaborating pretypes The rules for $\Gamma \vdash \tau$ type $\rightsquigarrow A$ are straightforward translations of the type-well-formedness rules of Chapter 2. (When it is necessary to contrast algorithmic and non-algorithmic rules, the latter are often referred to as *declarative*.)

$$\frac{\Gamma \vdash \tau_0\ \mathsf{type} \rightsquigarrow A \qquad \Gamma.A \vdash \tau_1\ \mathsf{type} \rightsquigarrow B}{\Gamma \vdash (\mathtt{Pi}\ \tau_0\ \tau_1)\ \mathsf{type} \rightsquigarrow \mathbf{\Pi}(A,B)} \qquad \frac{\Gamma \vdash \tau_0\ \mathsf{type} \rightsquigarrow A \qquad \Gamma.A \vdash \tau_1\ \mathsf{type} \rightsquigarrow B}{\Gamma \vdash (\mathtt{Sigma}\ \tau_0\ \tau_1)\ \mathsf{type} \rightsquigarrow \mathbf{\Sigma}(A,B)}$$

$$\frac{}{\Gamma \vdash \mathtt{Unit}\ \mathsf{type} \rightsquigarrow \mathbf{Unit}} \qquad \frac{}{\Gamma \vdash \mathtt{U}\ \mathsf{type} \rightsquigarrow \mathbf{U}} \qquad \frac{\Gamma \vdash e : \mathbf{U} \rightsquigarrow a}{\Gamma \vdash (\mathtt{El}\ e)\ \mathsf{type} \rightsquigarrow \mathbf{El}(a)}$$

3.1.2 Elaborating preterms: the problem of type equality

Elaborating preterms is significantly more fraught. But first, let us remind ourselves of the process of type-checking $(\mathtt{lam}\ \tau_0\ \tau_1\ e) : \tau$. First, we attempt to elaborate the pretype $\mathbf{1} \vdash \tau\ \mathsf{type} \rightsquigarrow C$; if this succeeds, we then attempt to elaborate the preterm $\mathbf{1} \vdash (\mathtt{lam}\ \tau_0\ \tau_1\ e) : C \rightsquigarrow c$. If this also succeeds, then the type-checker reports success, having transformed the input presyntax to a well-formed term $\mathbf{1} \vdash c : C$.

Since `lam` is our presyntax for λ, elaborating `lam` via $\mathbf{1} \vdash (\mathtt{lam}\ \tau_0\ \tau_1\ e) : C \rightsquigarrow c$ should produce a term $c := \lambda_{1,A,B}(b)$ for some A, B, b determined by τ_0, τ_1, e respectively. We determine these by a series of recursive calls to the elaborator: first $\mathbf{1} \vdash \tau_0\ \mathsf{type} \rightsquigarrow A$, then $\mathbf{1}.A \vdash \tau_1\ \mathsf{type} \rightsquigarrow B$, and finally $\mathbf{1}.A \vdash e : B \rightsquigarrow b$. Note that these steps must be performed sequentially and in this order, because each step uses the outputs of the previous steps as inputs: we elaborate τ_1 in a context extended by A, the result of elaborating τ_0, and we elaborate e at type B, the result of elaborating τ_1.

At the end we obtain $\mathbf{1}.A \vdash b : B$, and thence by $\mathbf{\Pi}$-introduction a term $\mathbf{1} \vdash \lambda_{1,A,B}(b) : \mathbf{\Pi}_1(A,B)$ that should be the elaborated form of e. But the elaborated form of e is supposed to have type C—the result of elaborating τ! Thus before returning $\lambda_{1,A,B}(b)$ we need to check that $\mathbf{1} \vdash C = \mathbf{\Pi}(A,B)$ type. This is where "type-checking" actually happens: we have seen that τ determines a real type and that e determines a real term, but until this point we have not actually checked whether "e has type τ."

In pseudocode, we can define elaboration of $(\mathtt{lam}\ \tau_0\ \tau_1\ e)$ as follows:

```
elabTm(Γ, C, (lam τ₀ τ₁ e)) =
  let A = elabTy(Γ, τ₀) in
  let B = elabTy(Γ.A, τ₁) in
  let b = elabTm(Γ.A, B, e) in
  if (Γ ⊢ C = Π_Γ(A, B) type) then return λ_{Γ,A,B}(b) else error
```

or equivalently, in algorithmic judgment notation:

$$\frac{\Gamma \vdash \tau_0\,\mathsf{type} \rightsquigarrow A \qquad \Gamma.A \vdash \tau_1\,\mathsf{type} \rightsquigarrow B \qquad \Gamma.A \vdash e : B \rightsquigarrow b \qquad \Gamma \vdash C = \mathbf{\Pi}(A,B)\,\mathsf{type}}{\Gamma \vdash (\mathtt{lam}\ \tau_0\ \tau_1\ e) : C \rightsquigarrow \lambda_{\Gamma,A,B}(b)}$$

This will be the only rule that concludes $\Gamma \vdash e : C \rightsquigarrow c$ for $e := (\mathtt{lam}\ \tau_0\ \tau_1\ e_0)$, ensuring that this rule "is the `lam` clause of `elabTm`," so to speak. Elaboration of other introduction forms will follow a similar pattern.

Exercise 3.1. Write the algorithmic rule for elaborating the preterm $(\mathtt{pair}\ \tau_0\ \tau_1\ e_0\ e_1)$.

Let us pause to make several remarks. First, note that our algorithm needs to check judgmental equality of types $\Gamma \vdash C = \mathbf{\Pi}_\Gamma(A,B)\,\mathsf{type}$. This step is, at least implicitly, part of all type-checking algorithms for all programming languages: if we define a function of type $A \to B$ that returns e, we have to check whether the type of e matches the declared return type B. Sometimes this is as simple as checking the syntactic equality of two type expressions, but often this is non-trivial, perhaps a subtyping check.

In our present setting, checking type equality is *extremely* non-trivial. Suppose that $C := \mathbf{El}(c)$ and so we are checking $\Gamma \vdash \mathbf{El}(c) = \mathbf{\Pi}(A,B)\,\mathsf{type}$ for $\Gamma \vdash c : \mathbf{U}$. This type equality depends on the entire equational theory of *terms*: we may need to "rewrite along" arbitrarily many term equations before concluding $\Gamma \vdash c = \mathbf{pi}(c_0,c_1) : \mathbf{U}$; this only reduces the problem to $\Gamma \vdash \mathbf{\Pi}(\mathbf{El}(c_0),\mathbf{El}(c_1)) = \mathbf{\Pi}(A,B)\,\mathsf{type}$ for which it suffices to check $\Gamma \vdash \mathbf{El}(c_0) = A\,\mathsf{type}$ and $\Gamma.A \vdash \mathbf{El}(c_1) = B\,\mathsf{type}$, each of which may once again require arbitrary amounts of computation. We will revisit this point in Section 3.2.1.

Secondly, note that we have assumed for now that the preterm $(\mathtt{lam}\ \tau_0\ \tau_1\ e)$ contains pretype annotations τ_0, τ_1 telling us the domain and codomain of the $\mathbf{\Pi}$-type. In practice, a type-checker is essentially unusable unless it can *reconstruct* (most of) these annotations; we describe this reconstruction process in Section 3.2.2.

Remark 3.1.6. Naïvely, one might think that *including* these annotations is the source of our problem, because it forces us to compare the type C computed from τ to the type $\mathbf{\Pi}(A,B)$ computed from the annotations τ_0, τ_1. This is not the case. If we omit τ_0, τ_1, then to elaborate e we must *recover* A and B from C, which upgrades "does $\Gamma \vdash C = \mathbf{\Pi}(A,B)\,\mathsf{type}$?" to the strictly harder question "do there *exist* A, B such that $\Gamma \vdash C = \mathbf{\Pi}(A,B)\,\mathsf{type}$?" In addition, we will need to wonder whether this existence is unique: otherwise, it could be that $\Gamma.A \vdash e : B \rightsquigarrow b$ for some choices of A, B but not others. ◇

Elaborating elimination forms is not much harder than elaborating introduction forms. To elaborate $(\mathtt{app}\ \tau_0\ \tau_1\ e_0\ e_1)$, we elaborate the pretype annotations

$\Gamma \vdash \tau_0\,\mathsf{type} \rightsquigarrow A$ and $\Gamma.A \vdash \tau_1\,\mathsf{type} \rightsquigarrow B$ in sequence, then the function $\Gamma \vdash e_0 : \mathbf{\Pi}(A,B) \rightsquigarrow f$ and its argument $\Gamma \vdash e_1 : A \rightsquigarrow a$ in either order, before finally checking that the type of the computed term $\mathbf{app}_{\Gamma,A,B}(f,a)$, namely $B[\mathbf{id}.a]$, agrees with the expected type C.

$$\frac{\Gamma \vdash \tau_0\,\mathsf{type} \rightsquigarrow A \qquad \Gamma.A \vdash \tau_1\,\mathsf{type} \rightsquigarrow B \qquad \Gamma \vdash e_0 : \mathbf{\Pi}(A,B) \rightsquigarrow f \qquad \Gamma \vdash e_1 : A \rightsquigarrow a \qquad \Gamma \vdash C = B[\mathbf{id}.a]\,\mathsf{type}}{\Gamma \vdash (\mathsf{app}\ \tau_0\ \tau_1\ e_0\ e_1) : C \rightsquigarrow \mathbf{app}_{\Gamma,A,B}(f,a)}$$

Elaboration of other elimination forms follows a similar pattern. The only remaining case is term variables ($\mathsf{var}\ i$), which we have chosen to represent as de Bruijn indices. To elaborate ($\mathsf{var}\ i$) we check that the context has length at least $i+1$; if so, then it remains only to check that the type of the variable $\mathbf{q}[\mathbf{p}^i]$ agrees with the expected type.

$$\frac{\Gamma = \Gamma'.A_i.A_{i-1}.\cdots.A_0 \qquad \Gamma \vdash C = A_i[\mathbf{p}^{i+1}]\,\mathsf{type}}{\Gamma \vdash (\mathsf{var}\ i) : C \rightsquigarrow \mathbf{q}[\mathbf{p}^i]}$$

In the above rule, our algorithm needs to check judgmental equality of *contexts*, and to project Γ and A from $\Gamma.A$. Unlike for type equality, we have no rules generating non-trivial context equalities, so structural induction on contexts is perfectly well-defined.

Remark 3.1.7. It is straightforward to extend our concrete syntax to support named variables: in our elaboration judgments, we replace Γ with an *environment* Θ that is a list of pairs of genuine types with the "surface name" of the corresponding term variable. Every environment determines a context by forgetting the names; in the variable elaboration rule, we simply look up the de Bruijn index corresponding to the given name. ◇

Exercise 3.2. Write the algorithmic rules for elaborating ($\mathtt{fst}\ \tau_0\ \tau_1\ e$) and ($\mathtt{snd}\ \tau_0\ \tau_1\ e$).

3.2 *Metatheory for type-checking*

In Section 3.1 we saw that we can reduce type-checking to the problem of deciding the equality of types (at least, assuming that our input preterms have all type annotations). Deciding the equality of types in turn requires deciding the equality of terms, particularly in the presence of universes (Section 2.6.2). It is far from obvious that these relations are decidable—in fact, as we will see in Section 3.6, they are actually *undecidable* for the theory presented in Chapter 2—and proving their decidability relies

on a difficult metatheorem known as *normalization*. In this section, we continue our exploration of elaboration with an emphasis on normalization and other metatheorems necessary for type-checking.

Remark 3.2.1. Recall from Section 2.1 that a *metatheorem* is just an ordinary theorem in the ambient metatheory, particularly one concerning the object type theory. ◇

Before we can discuss computability-theoretic properties of the judgments of type theory, however, we must fix an encoding. We have taken pains to treat the rules of type theory as defining abstract sets $\mathsf{Ty}(\Gamma)$ and $\mathsf{Tm}(\Gamma, A)$ equipped with functions (type and term formers) satisfying various equations (β and η laws), which is the right perspective for understanding the mathematical structure of type theory. But to discuss the *computational* properties of type theory it is essential to exhibit an effective encoding of types and terms that is suitable for manipulation by a Turing machine or other model of computation: Turing machines cannot take mathematical entities as inputs, and whether equality of types is decidable can depend on how we choose to encode them!

This is analogous to the issue that arises in elementary computability theory when formalizing the halting problem: we must agree on how to encode Turing machines as inputs to other Turing machines, and we must ensure that this encoding is suitably effective. It is possible to pick an encoding of computable functions that trivializes the halting problem, at the expense of this encoding itself necessarily being uncomputable.

Returning to type theory, derivation trees of inference rules (e.g., as in Appendix A) turn out to be a perfectly suitable encoding. That is, when discussing computability-theoretic properties of types, terms, and equality judgments, we shall assume that each of these is encoded by equivalence classes of closed derivation trees; for example, we encode $\mathsf{Ty}(\Gamma)$ by the set of derivation trees with root $\Gamma \vdash A\ \mathsf{type}$ for some A. (Just as there are many Turing machines realizing any given function $\mathbb{N} \to \mathbb{N}$, there will be many derivation trees encoding any given type $A \in \mathsf{Ty}(\Gamma)$.) When we say "equality of types is decidable," what we shall mean is that "it is decidable whether two derivations encode the same type." But having fixed a convention, we will avoid belaboring the point any further.

3.2.1 Normalization and the decidability of equality

To complete the pretype and preterm elaboration algorithms presented in Section 3.1, it remains only to show that type and term equality are decidable, which is equivalent to the following normalization condition.

Remark 3.2.2. Type and term equality are automatically *semidecidable* because derivation trees are recursively enumerable. That is, to check whether two types $A, B \in \mathsf{Ty}(\Gamma)$ are equal, we can enumerate every derivation tree of type theory, terminating if we encounter a derivation of $\Gamma \vdash A = B$ type. Obviously, this is not a realistic implementation strategy. ◇

Definition 3.2.3. A *normalization structure* for a type theory is a pair of computable, injective functions $\mathtt{nfTy} : \mathsf{Ty}(\Gamma) \to \mathbb{N}$ and $\mathtt{nfTm} : \mathsf{Tm}(\Gamma, A) \to \mathbb{N}$.

Definition 3.2.4. A type theory enjoys *normalization* if it admits a normalization structure.

The reader may find these definitions surprising: where did $\mathbb{N}$ come from, and where is the rest of the definition? We have chosen $\mathbb{N}$ because it is a countable set with decidable equality, but any other such set would suffice. In practice, one instead defines two sets of abstract syntax trees $\mathtt{TyNf}$, $\mathtt{TmNf}$ with discrete equality, and constructs a pair of computable, injective functions $\mathtt{nfTy} : \mathsf{Ty}(\Gamma) \to \mathtt{TyNf}$ and $\mathtt{nfTm} : \mathsf{Tm}(\Gamma, A) \to \mathtt{TmNf}$. It is trivial to exhibit computable, injective Gödel encodings of $\mathtt{TyNf}$ and $\mathtt{TmNf}$, which when composed with $\mathtt{nfTy}, \mathtt{nfTm}$ exhibit a normalization structure in the sense of Definition 3.2.3.

As for Definition 3.2.3 being sufficient, the force of normalization is that it gives us a decision procedure for type/term equality as follows: given $A, B \in \mathsf{Ty}(\Gamma)$, A and B are equal if and only if $\mathtt{nfTy}(A) = \mathtt{nfTy}(B)$ in $\mathbb{N}$. Asking for these maps to be computable ensures that this procedure is computable; injectivity ensures that it is *complete* in the sense that $\mathtt{nfTy}(A) = \mathtt{nfTy}(B)$ implies $A = B$. The *soundness* of this procedure—that $A = B$ implies $\mathtt{nfTy}(A) = \mathtt{nfTy}(B)$—is implicit in the statement that $\mathtt{nfTy}$ is a function out of $\mathsf{Ty}(\Gamma)$, the set of types considered modulo judgmental equality.

Warning 3.2.5. In Section 3.6 we shall see that extensional type theory *does not* admit a normalization structure, but we will proceed under the assumption that the theory we are elaborating satisfies normalization. In Chapter 4 we will see how to modify our type theory to substantiate this assumption.

Assuming normalization, we can define algorithmic equality judgments

1. $\Gamma \vdash A \Leftrightarrow B$ type asserts that the types $\Gamma \vdash A$ type and $\Gamma \vdash B$ type are judgmentally equal according to some decision procedure.
2. $\Gamma \vdash a \Leftrightarrow b : A$ asserts that the terms $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$ are judgmentally equal according to some decision procedure.

as follows:

$$\frac{\mathtt{nfTy}(A) = \mathtt{nfTy}(B)}{\Gamma \vdash A \Leftrightarrow B \,\mathsf{type}} \qquad \frac{\mathtt{nfTm}(a) = \mathtt{nfTm}(b)}{\Gamma \vdash a \Leftrightarrow b : A}$$

We notate algorithmic equality differently from the declarative equality judgments $\Gamma \vdash A = B\ \mathrm{type}$ and $\Gamma \vdash a = b : A$ to stress that their definitions are completely different, even though (by our argument above) two types/terms are algorithmically equal if and only if they are declaratively equal. We thus complete the elaborator from Section 3.1 by replacing the "calls" to $\Gamma \vdash C = \mathbf{\Pi}(A, B)\ \mathrm{type}$ with calls to $\Gamma \vdash C \Leftrightarrow \mathbf{\Pi}(A, B)\ \mathrm{type}$.

Remark 3.2.6. It may seem surprising that normalization is so difficult; why can't algorithmic equality just *orient* each declarative equality rule (e.g., $\mathbf{fst}(\mathbf{pair}(a, b)) \rightsquigarrow a$) and check whether the resulting rewriting system is confluent and terminating? Unfortunately, while this strategy suffices for some dependent type theories such as the calculus of constructions [CH88], it is very difficult to account for judgmental η rules. (What direction should $p \leftrightsquigarrow \mathbf{pair}(\mathbf{fst}(p), \mathbf{snd}(p))$ go? What about the η rule of **Unit**, $a \leftrightsquigarrow \mathbf{tt}$?) These rules require a type-sensitive decision procedure known as *normalization by evaluation*, whose soundness and completeness for declarative equality is nontrivial [ACD07; Abe13]. ◇

Exercise 3.3. We argued that the existence of a normalization structure implies that judgmental equality is decidable. In fact, this is a biimplication. Assume that definitional equality is decidable, and construct from this a normalization structure. (Hint: some classical reasoning is required, such as Markov's principle or the law of excluded middle.)

3.2.2 *Injectivity and bidirectional type-checking*

We have seen how to define a rudimentary elaborator for type theory assuming that normalization holds, but the preterms that we can elaborate (Figure 3.1) are quite verbose, making our proof assistant more of a proof adversary. For instance, $(\mathsf{app}\ \tau_0\ \tau_1\ e_0\ e_1)$ requires annotations for both the domain and codomain of the $\mathbf{\Pi}$-type.

These annotations are highly redundant, but it is far from clear how many of them can be mechanically reconstructed by our elaborator, nor if there is a consistent strategy for doing so. Users of typed functional programming languages like OCaml or Haskell might imagine that virtually all types can be inferred automatically, but unfortunately type inference for dependent type theory is undecidable [Dow93].

It turns out there is a fairly straightforward, local, and usable approach to type reconstruction known as *bidirectional type-checking* [Coq96; PT00; McB18; McB19]. The core insight of bidirectional type-checking is that for some preterms it is easy to reconstruct or *synthesize* its type (e.g., if we know a function's type then we know the type of its applications), but for other preterms we must be given a type at which to *check* it (e.g., to type-check a function we need to be told the type of its input variable).

$$\begin{array}{lll} \textit{Pretypes} & \tau ::= & (\texttt{Pi}\ \tau\ \tau) \mid (\texttt{Sigma}\ \tau\ \tau) \mid \texttt{Unit} \mid \texttt{U} \mid \cdots \\ \textit{Preterms} & e ::= & (\texttt{var}\ i) \mid (\texttt{chk}\ e\ \tau) \mid (\texttt{lam}\ e) \mid (\texttt{app}\ e\ e) \mid (\texttt{pair}\ e\ e) \mid \cdots \end{array}$$

Figure 3.2: Syntax of pretypes and preterms for a bidirectional elaborator.

By explicitly splitting elaboration into two mutually-defined algorithms—type-checking and type synthesis—we can dramatically reduce type annotations. In fact, in Figure 3.2 we can see that our new preterm syntax has no type annotations whatsoever except for a single annotation form $(\texttt{chk}\ e\ \tau)$ that we will use sparingly. The ebb and flow of information between terms and types—between checking and synthesis—leads to the eponymous bidirectional flow of information that has proven easily adaptable to new type theories. But when should we check, and when should we synthesize?

Slogan 3.2.7. *Types are checked in introduction rules, and synthesized in elimination rules.*

We replace our two algorithmic elaboration judgments $\Gamma \vdash \tau\,\mathsf{type} \rightsquigarrow A$ and $\Gamma \vdash e : A \rightsquigarrow a$ with three algorithmic judgments as follows:

1. $\Gamma \vdash \tau \Leftarrow \mathsf{type} \rightsquigarrow A$ ("check τ") asserts that elaborating the pretype τ relative to $\vdash \Gamma\,\mathsf{cx}$ succeeds and produces the type $\Gamma \vdash A\,\mathsf{type}$.

2. $\Gamma \vdash e \Leftarrow A \rightsquigarrow a$ ("check e against A") asserts that elaborating the (unannotated) preterm e relative to $\vdash \Gamma\,\mathsf{cx}$ and a *given* type $\Gamma \vdash A\,\mathsf{type}$ succeeds with $\Gamma \vdash a : A$.

3. $\Gamma \vdash e \Rightarrow A \rightsquigarrow a$ ("synthesize A from e") asserts that elaborating the (unannotated) preterm e relative to $\vdash \Gamma\,\mathsf{cx}$ succeeds and *produces* both $\Gamma \vdash A\,\mathsf{type}$ and $\Gamma \vdash a : A$.

The first two judgments, $\Gamma \vdash \tau \Leftarrow \mathsf{type} \rightsquigarrow A$ and $\Gamma \vdash e \Leftarrow A \rightsquigarrow a$, are similar to our previous judgments; when elaborating a preterm we are given a context and a type at which to check it. In the third judgment, $\Gamma \vdash e \Rightarrow A \rightsquigarrow a$, we are also given a preterm and a context, but we output *both a term and its type.* The arrows are meant to indicate the direction of information flow: when checking $e \Leftarrow A$ we are given A and using it to elaborate e, but when synthesizing $e \Rightarrow A$ we are producing A from e.

The rules for $\Gamma \vdash \tau \Leftarrow \mathsf{type} \rightsquigarrow A$ are the same as those for $\Gamma \vdash \tau\,\mathsf{type} \rightsquigarrow A$, except that they reference the new checking judgment $\Gamma \vdash e \Leftarrow A \rightsquigarrow a$ instead of $\Gamma \vdash e : A \rightsquigarrow a$. But for each old $\Gamma \vdash e : A \rightsquigarrow a$ rule we must decide whether to check or synthesize, and in the latter case, how to reconstruct the type.

The easiest case is the variable $(\texttt{var}\ i)$. Elaboration always takes place with respect to a context which records the types of each variable, so it is easy to synthesize the

variable's type. Notably, unlike in our previous variable rule, we do not need to check type equality!

$$\frac{\Gamma = \Gamma'.A_i.A_{i-1}.\cdots.A_0}{\Gamma \vdash (\mathtt{var}\ i) \Rightarrow A_i[\mathbf{p}^{i+1}] \rightsquigarrow \mathbf{q}[\mathbf{p}^i]}$$

Next, let us consider the rules for $\mathbf{\Pi}$-types. According to Slogan 3.2.7, the introduction form $(\mathtt{lam}\ e)$ should be checked. As in Section 3.1, to check $\Gamma \vdash (\mathtt{lam}\ e) \Leftarrow C \rightsquigarrow \lambda(b)$ we must recursively check the body of the lambda, $\Gamma.A \vdash e \Leftarrow B \rightsquigarrow b$. But where do A and B come from? (Last time, we elaborated them from `lam`'s annotations.) We might imagine that we can recover A and B from the given type C,

$$\frac{\Gamma \vdash C \Leftrightarrow \mathbf{\Pi}(A,B)\ \mathsf{type} \qquad \Gamma.A \vdash e \Leftarrow B \rightsquigarrow b}{\Gamma \vdash (\mathtt{lam}\ e) \Leftarrow C \rightsquigarrow \lambda(b)}\ !?$$

but this rule does not make sense as written. $\Gamma \vdash C \Leftrightarrow D\,\mathsf{type}$ is an algorithm which takes two types and returns "yes" or "no"; we cannot use it to invent the types A and B.

Worse yet, as foreshadowed in Remark 3.1.6, even if we can find A and B such that $\Gamma \vdash C \Leftrightarrow \mathbf{\Pi}(A,B)\,\mathsf{type}$, there is no reason to expect this choice to be unique. That is, it could be that $\Gamma \vdash C \Leftrightarrow \mathbf{\Pi}(A,B)\,\mathsf{type}$ and $\Gamma \vdash C \Leftrightarrow \mathbf{\Pi}(A',B')\,\mathsf{type}$ both hold, but $A \neq A'$ (or alternatively, $A = A'$ and $B \neq B'$). If so, it is possible that e elaborates with respect to one of these choices but not the other, i.e., $\Gamma.A \vdash e \Leftarrow B \rightsquigarrow b$ succeeds but $\Gamma.A' \vdash e \Leftarrow B' \rightsquigarrow\ ?$ fails; even if both succeed, they will necessarily elaborate two different terms! We must foreclose these possibilities in order for elaboration to be well-defined.

Definition 3.2.8. A type theory has *injective* $\mathbf{\Pi}$*-types* if $\Gamma \vdash \mathbf{\Pi}(A,B) = \mathbf{\Pi}(A',B')\,\mathsf{type}$ implies $\Gamma \vdash A = A'\,\mathsf{type}$ and $\Gamma.A \vdash B = B'\,\mathsf{type}$.

Definition 3.2.9. A type theory has *invertible* $\mathbf{\Pi}$*-types* if it has injective $\mathbf{\Pi}$-types and admits a computable function which, given $\Gamma \vdash C\,\mathsf{type}$, either produces the unique $\Gamma \vdash A\,\mathsf{type}$ and $\Gamma.A \vdash B\,\mathsf{type}$ for which $\Gamma \vdash C = \mathbf{\Pi}(A,B)\,\mathsf{type}$, or determines that no such A, B exist.

Remark 3.2.10. That is, a type theory has injective $\mathbf{\Pi}$-types if the type former $\mathbf{\Pi}_\Gamma : (\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)) \to \mathsf{Ty}(\Gamma)$ is injective. A type theory has invertible $\mathbf{\Pi}$-types if the image of $\mathbf{\Pi}_\Gamma$ is decidable and $\mathbf{\Pi}_\Gamma$ admits a (computable) partial inverse $\mathbf{\Pi}_\Gamma^{-1} : \mathrm{Im}(\mathbf{\Pi}_\Gamma) \to (\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A))$. ◇

Particularly in light of Remark 3.2.10, one can easily extend the terminology of injectivity and invertibility to non-$\mathbf{\Pi}$ type formers.

Definition 3.2.11. If all the type constructors of a type theory are injective (resp., invertible), we say that the type theory *has injective (resp., invertible) type constructors.*

Having injective or invertible type constructors does not follow from normalization. (A type theory in which all empty types are equal may be normalizing but will not satisfy injectivity.) In practice, however, having invertible type constructors is almost always an immediate consequence of the *proof* of normalization. As we mentioned in Section 3.2.1, normalization proofs generally construct abstract syntax trees $\mathtt{TyNf}$, $\mathtt{TmNf}$ of "β-short, η-long" types and terms for which equality is both syntactic as well as sound and complete for judgmental equality. Given a type $\Gamma \vdash C$ type, we invert its head constructor by computing $\mathtt{nfTy}(C) \in \mathtt{TyNf}$, checking its head constructor in $\mathtt{TyNf}$, and projecting its arguments.

Injectivity and invertibility are very strong conditions; function types in set theory are not injective, nor are $\mathbf{\Pi}$-types injective in extensional type theory.

Exercise 3.4. Give an example of three sets X, Y, Z such that $X \ncong Y$, but the set of functions $X \to Z$ is equal to the set of functions $Y \to Z$.

Exercise 3.5. We will see in Section 3.5 that type theory admits an interpretation in which closed types are sets. Exercise 3.4 shows that sets do not have injective $\mathbf{\Pi}$-types, but these two facts together do not imply that type theory lacks injective $\mathbf{\Pi}$-types. Why not?

Warning 3.2.12. In Section 3.5 we shall see that extensional type theory *does not* have injective type constructors, due to interactions between equality reflection and large elimination or universes (Theorem 3.5.19). We will proceed under the assumption that the theory we are elaborating has invertible type constructors, and in Chapter 4 we will see how to modify our type theory to substantiate this assumption.

Completing our elaborator The force of having invertible $\mathbf{\Pi}$-types is to have an algorithm $\mathtt{unPi}$ which takes $\Gamma \vdash C$ type and returns the unique pair of types A, B for which $\Gamma \vdash C = \mathbf{\Pi}(A, B)$ type, or raises an exception if this pair does not exist. Using $\mathtt{unPi}$ we can repair our earlier attempt at checking $(\mathtt{lam}\ e)$, and define the synthesis rule for $(\mathtt{app}\ e_0\ e_1)$:

$$\frac{\mathtt{unPi}(C) = (A, B) \qquad \Gamma.A \vdash e \Leftarrow B \rightsquigarrow b}{\Gamma \vdash (\mathtt{lam}\ e) \Leftarrow C \rightsquigarrow \lambda(b)}$$

$$\frac{\Gamma \vdash e_0 \Rightarrow C \rightsquigarrow f \qquad \mathtt{unPi}(C) = (A, B) \qquad \Gamma \vdash e_1 \Leftarrow A \rightsquigarrow a}{\Gamma \vdash (\mathtt{app}\ e_0\ e_1) \Rightarrow B[\mathbf{id}.a] \rightsquigarrow \mathbf{app}(f, a)}$$

This is the only elaboration rule for $(\mathtt{lam}\ e)$; in particular, there is no *synthesis* rule for lambda, because we cannot elaborate e without knowing what type A to add to the context. On the other hand, to synthesize the type of $(\mathtt{app}\ e_0\ e_1)$, we *synthesize*

the type of e_0; if it is of the form $\mathbf{\Pi}(A, B)$, we then *check* that e_1 has type A and then return B, suitably instantiated. Putting these rules together, the reader might notice that we cannot type-check $(\mathtt{app}\ (\mathtt{lam}\ e_0)\ e_1)$, because this would require *synthesizing* $(\mathtt{lam}\ e_0)$. In fact, bidirectional type-checking cannot generally type-check β-redexes.

For this reason, we have included a *type-annotation* preterm $(\mathtt{chk}\ e\ \tau)$ which allows users to explicitly annotate a preterm with a pretype. The type of this preterm is trivially synthesizable: it is the result of elaborating τ! In order to synthesize $(\mathtt{chk}\ e\ \tau)$, we simply *check* e against τ, and if successful, return that type.

$$\frac{\Gamma \vdash \tau \Leftarrow \mathsf{type} \rightsquigarrow A \qquad \Gamma \vdash e \Leftarrow A \rightsquigarrow a}{\Gamma \vdash (\mathtt{chk}\ e\ \tau) \Rightarrow A \rightsquigarrow a}$$

In particular, we can type-check the β-redex from before, as long as we annotate the lambda with its intended type: $(\mathtt{app}\ (\mathtt{chk}\ (\mathtt{lam}\ e_0)\ (\mathtt{Pi}\ \tau_0\ \tau_1))\ e_1)$.

The above rule allows us to treat a checkable term as synthesizable. The converse is much easier: to *check* the type of a synthesizable term, we simply compare the synthesized type to the expected type.

$$\frac{\Gamma \vdash e \Rightarrow B \rightsquigarrow a \qquad \Gamma \vdash A \Leftrightarrow B\,\mathsf{type}}{\Gamma \vdash e \Leftarrow A \rightsquigarrow a}$$

As written, the above rule applies to *any* checking problem because its conclusion is unconstrained. In our elaboration algorithm, we should only apply this rule if no other rule matches. It is the final "catch-all" clause for situations where we have not one but two sources of type information: on the one hand, we can synthesize e's type directly, and on the other hand, we are also given the type that e is supposed to have. Interestingly, this is the *only* rule where our bidirectional elaborator checks type equality $\Gamma \vdash A \Leftrightarrow B\,\mathsf{type}$.

Exercise 3.6. For each of $(\mathtt{pair}\ e_0\ e_1)$, $(\mathtt{fst}\ e)$, and $(\mathtt{snd}\ e)$, decide whether this preterm should be checked or synthesized, then write the algorithmic rule for elaborating it. (Hint: you must assume that Σ-types are invertible.)

3.3⋆ A case study in elaboration: definitions

To round out our discussion of elaboration, we sketch how to extend our concrete syntax and type-checker to account for *definitions*, a key part of any proof assistant. The input to a proof assistant is typically not a single $e : \tau$ but a *sequence* of definitions

$$\begin{aligned}
&\mathsf{def}_1 : \tau_1 = e_1\\
&\mathsf{def}_2 : \tau_2 = e_2\\
&\vdots\\
&\mathsf{def}_n : \tau_n = e_n
\end{aligned}$$

where every e_j and τ_j can mention def_i for $i < j$.

To account for this cross-definition dependency, we might imagine elaborating each definition one at a time, adding a new (nameless) variable to the context for each successful definition. Such a strategy might proceed as follows:

1. elaborate $\mathbf{1} \vdash \tau_1 \Leftarrow \mathsf{type} \rightsquigarrow A_1$ and $\mathbf{1} \vdash e_1 \Leftarrow A_1 \rightsquigarrow a_1$; if successful,
2. elaborate $\mathbf{1}.A_1 \vdash \tau_2 \Leftarrow \mathsf{type} \rightsquigarrow A_2$ and $\mathbf{1}.A_1 \vdash e_2 \Leftarrow A_2 \rightsquigarrow a_2$; if successful,
3. continue elaborating each τ_i and e_i in context $\mathbf{1}.A_1.\ldots.A_{i-1}$ as above.

Unfortunately this algorithm is too naïve: if we treat def_1 as a *variable* of type A_1, the type-checker will not have access to the definition $\mathsf{def}_1 = a_1$. Consider:

```
const : Nat
const = 2

proof : const ≡ 2
proof = refl
```

Here const will successfully elaborate in the empty context to $\mathbf{suc}(\mathbf{suc}(\mathbf{zero})) : \mathbf{Nat}$, but the elaboration problem for proof will be $\mathbf{1}.\mathbf{Nat} \vdash \mathtt{refl} \Leftarrow \mathbf{q} \equiv \mathbf{suc}(\mathbf{suc}(\mathbf{zero})) \rightsquigarrow$?, which will fail: an arbitrary variable of type **Nat** is surely not equal to 2!

Remark 3.3.1. For readers familiar with functional programming, we summarize the above discussion as "**let** is no longer λ," in reference to the celebrated encoding of $(\mathbf{let}\ x = a\ \mathbf{in}\ b)$ as $((\lambda x.\ b)\ a)$ often adopted in Lisp-family languages. This slogan is not unique to dependent type theory; users of ML-family languages may already be familiar with this phenomenon in light of the Hindley-Milner approach to typing **let**. ◇

To solve this problem, we must somehow instrument our elaborator with the ability to remember not only the *type* of a definition but its *definiens* as well. There are several ways to accomplish this; one possibility is to add a new form of *definitional context extension* "$\Gamma.(\mathbf{q} := a : A)$" in which the variable is judgmentally equal to a given term a [McB99; SP94]. We opt for an indirect but less invasive encoding of this idea: taking inspiration from Section 2.6.2, wherein we encoded "extending the context by a type variable" by adding a new type **U** whose terms are codes for types, we will add a new type former, *singleton types*, whose terms are elements of A judgmentally equal to a.

Singleton types The singleton type of $\Gamma \vdash a : A$, written $\Gamma \vdash \mathbf{Sing}(A, a)$ type, is a type whose elements are in bijection with the elements of $\mathsf{Tm}(\Gamma, A)$ that are equal to a, namely the singleton subset $\{a\}$ [Asp95; SH06]. That is, naturally in Γ,

$$\mathbf{Sing}_\Gamma : (\textstyle\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Tm}(\Gamma, A)) \to \mathsf{Ty}(\Gamma)$$
$$\iota_{\Gamma,A,a} : \mathsf{Tm}(\Gamma, \mathbf{Sing}(A, a)) \cong \{b \in \mathsf{Tm}(\Gamma, A) \mid b = a\}$$

In inference rules,

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{Sing}(A, a)\ \mathrm{type}} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{in}(a) : \mathbf{Sing}(A, a)} \qquad \frac{\Gamma \vdash s : \mathbf{Sing}(A, a)}{\Gamma \vdash \mathbf{out}(s) : A}$$

$$\frac{\Gamma \vdash s : \mathbf{Sing}(A, a)}{\Gamma \vdash \mathbf{out}(s) = a : A} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash s : \mathbf{Sing}(A, a)}{\Gamma \vdash \mathbf{in}(\mathbf{out}(s)) = s : \mathbf{Sing}(A, a)}$$

This definition may seem rather odd, but note that a variable of type $\mathbf{Sing}(A, a)$ determines a term $\mathbf{out}(\mathbf{q}) : A[\mathbf{p}]$ that is judgmentally equal to $a[\mathbf{p}]$, thereby allowing us to extend contexts by "defined variables."

Remark 3.3.2. In extensional type theory, we can define singleton types as pairs of an element of A and a proof that it equals a, i.e., $\mathbf{Sing}(A, a) \coloneqq \Sigma(A, \mathbf{Eq}(A[\mathbf{p}], \mathbf{q}, a[\mathbf{p}]))$ with $\mathbf{in}(a) \coloneqq \mathbf{pair}(a, \mathbf{refl})$ and $\mathbf{out}(s) \coloneqq \mathbf{fst}(s)$. This encoding makes essential use of equality reflection, but singleton types can also be added as a primitive type former to type theories without equality reflection, without disrupting normalization. ◇

Extending our elaborator We begin by introducing concrete syntax for lists of $e : \tau$ pairs, which we call *declarations*:

Declarations	$ds ::=$	$(\mathtt{decls}\ (e_1\ \tau_1)\ \ldots)$
Pretypes	$\tau ::=$	$\cdots$
Preterms	$e ::=$	$\cdots$

We extend our bidirectional elaborator as follows. First, we parameterize all our judgments by a second context Θ that keeps track of which variables in Γ are ordinary "local" variables (introduced by types/terms such as $\mathbf{\Pi}$ or λ), and which variables refer to declarations. We write Θ as a list $\mathbf{1}.\mathrm{decl}.\mathrm{decl}.\mathrm{local}.\ldots$ with the same length as $\Gamma = \mathbf{1}.A_1.A_2.A_3.\ldots$, to indicate in this case that only the variable of type A_3 is local. We will replace the variable rule shortly; the remaining elaboration rules do not interact with Θ except to extend Θ by local whenever a new variable is added to the context Γ.

Secondly, we introduce a new algorithmic judgment $\Gamma; \Theta \vdash ds$ ok which type-checks a list of declarations ds by elaborating the first declaration $(e_1\ \tau_1)$ in context

$\Gamma;\Theta$ into the term $a_1 : A_1$, and then elaborating the remaining declarations in context $\Gamma.\mathbf{Sing}(A_1, a_1);\Theta.\mathsf{decl}$.

$$\frac{}{\Gamma;\Theta \vdash (\mathtt{decls})\ \mathsf{ok}} \qquad \frac{\Gamma;\Theta \vdash \tau_1 \Leftarrow \mathsf{type} \rightsquigarrow A_1 \qquad \Gamma;\Theta \vdash e_1 \Leftarrow A_1 \rightsquigarrow a_1 \qquad \Gamma.\mathbf{Sing}(A_1, a_1);\Theta.\mathsf{decl} \vdash (\mathtt{decls}\ (e_2\ \tau_2)\ \ldots)\ \mathsf{ok}}{\Gamma;\Theta \vdash (\mathtt{decls}\ (e_1\ \tau_1)\ (e_2\ \tau_2)\ \ldots)\ \mathsf{ok}}$$

Finally, we must edit our variable rule to account for whether a variable is an ordinary local variable or refers to an earlier declaration; in the latter case, we apply $\mathbf{out}(-)$ to the variable so that it has the correct type A rather than $\mathbf{Sing}(A, a)$.

$$\frac{\Gamma = \Gamma'.A_i.A_{i-1}.\cdots.A_0 \qquad \Theta = \Theta'.\mathsf{local}.x_{i-1}.\cdots.x_0}{\Gamma;\Theta \vdash (\mathtt{var}\ i) \Rightarrow A_i[\mathbf{p}^{i+1}] \rightsquigarrow \mathbf{q}[\mathbf{p}^i]} \qquad \frac{\Gamma = \Gamma'.A_i.A_{i-1}.\cdots.A_0 \qquad \Theta = \Theta'.\mathsf{decl}.x_{i-1}.\cdots.x_0 \qquad \mathtt{unSing}(A_i) = (A, a)}{\Gamma;\Theta \vdash (\mathtt{var}\ i) \Rightarrow A[\mathbf{p}^{i+1}] \rightsquigarrow \mathbf{out}(\mathbf{q}[\mathbf{p}^i])}$$

In the second rule above, the rules of singleton types ensure that the elaborated term $\mathbf{out}(\mathbf{q}[\mathbf{p}^i])$ is judgmentally equal to $a[\mathbf{p}^{i+1}]$, where a is the previously-elaborated definiens of the corresponding declaration. Putting everything together, to check the input $(\mathtt{decls}\ (e_1\ \tau_1)\ (e_2\ \tau_2)\ \ldots)$ we attempt to derive $\mathbf{1};\mathbf{1} \vdash (\mathtt{decls}\ (e_1\ \tau_1)\ (e_2\ \tau_2)\ \ldots)\ \mathsf{ok}$.

To sum up, we emphasize once again that although this book focuses on the *core calculi* of proof assistants, it is impossible to have a satisfactory understanding of this topic without paying heed to their *surface languages* as well; often, the best way to understand a new surface language feature is to add a new feature in the core language to accommodate it. Ideally, our alterations to the core language will be minor but will significantly simplify elaboration.

3.4 *Models for metatheory*

Our focus on type-checking has led us to normalization (Definition 3.2.4) and invertible type constructors (Definition 3.2.11) as metatheorems essential to the implementation of type theory. Notably, these metatheorems are stated with respect to types and terms in *arbitrary* contexts; in this section, we will discuss two more important metatheorems that concern only terms in the empty context $\mathbf{1}$, namely *consistency* and *canonicity*. Neither of these properties is needed to implement a type-checker, but as we will see, they are essential to the applications of type theory to logic and programming languages respectively.

Definition 3.4.1. A type theory is *consistent* if there is no closed term $\mathbf{1} \vdash a : \mathbf{Void}$.

Consistency is the lowest bar that a type theory must pass to serve as a logic. When we interpret types as logical propositions, **Void** corresponds to the false proposition. By the rules of **Void** (Section 2.5.1), the existence of a closed term $\mathbf{1} \vdash a : \mathbf{Void}$ (an assumption-free proof of false) implies that every closed type has at least one closed term $\mathbf{1} \vdash \mathbf{absurd}(a) : A$, or in other words, that every proposition has a proof. Thus Definition 3.4.1 corresponds to logical consistency in the traditional sense.

Exercise 3.7. Another common definition of consistency is that there is no closed term $\mathbf{1} \vdash a : \mathbf{Eq}(\mathbf{Nat}, \mathbf{zero}, \mathbf{suc}(\mathbf{zero}))$. Show that these two definitions are equivalent.

At this point we pause to sketch the model theory of type theory. In Chapter 2 we were careful to formulate the judgments of type theory as (indexed) sets, and the rules of type theory as (dependently-typed) operations between these sets and equations between these operations. As a result we can regard this data as a kind of generalized *algebra signature*, in the sense of Section 2.5.4; in particular, we obtain a general notion of "implementation" of, or *algebra* for, this signature—more commonly known as a *model of type theory*.

Definition 3.4.2. A *model of type theory* $\mathcal{M}$ consists of the following data:

1. a set $\mathsf{Cx}_{\mathcal{M}}$ of *$\mathcal{M}$-contexts*,
2. for each $\Delta, \Gamma \in \mathsf{Cx}_{\mathcal{M}}$, a set $\mathsf{Sb}_{\mathcal{M}}(\Delta, \Gamma)$ of *$\mathcal{M}$-substitutions from Δ to Γ*,
3. for each $\Gamma \in \mathsf{Cx}_{\mathcal{M}}$, a set $\mathsf{Ty}_{\mathcal{M}}(\Gamma)$ of *$\mathcal{M}$-types in Γ*, and
4. for each $\Gamma \in \mathsf{Cx}_{\mathcal{M}}$ and $A \in \mathsf{Ty}_{\mathcal{M}}(\Gamma)$, a set $\mathsf{Tm}_{\mathcal{M}}(\Gamma, A)$ of *$\mathcal{M}$-terms of A in Γ*,
5. an *empty $\mathcal{M}$-context* $\mathbf{1}_{\mathcal{M}} \in \mathsf{Cx}_{\mathcal{M}}$,
6. for each $\Gamma \in \mathsf{Cx}_{\mathcal{M}}$ and $A \in \mathsf{Ty}_{\mathcal{M}}(\Gamma)$, an *$\mathcal{M}$-context extension* $\Gamma.{}_{\mathcal{M}}A \in \mathsf{Cx}_{\mathcal{M}}$,
7. for $\Gamma \in \mathsf{Cx}_{\mathcal{M}}$, $A \in \mathsf{Ty}_{\mathcal{M}}(\Gamma)$, and $B \in \mathsf{Ty}_{\mathcal{M}}(\Gamma.{}_{\mathcal{M}}A)$, an *$\mathcal{M}$-$\mathbf{\Pi}$ type* $\mathbf{\Pi}_{\mathcal{M}}(A, B) \in \mathsf{Ty}_{\mathcal{M}}(\Gamma)$,
8. and every other context, substitution, type, and term forming operation described in Appendix A, all subject to all the equations stated in Appendix A.

Definition 3.4.3. Given two models of type theory $\mathcal{M}, \mathcal{N}$, a *homomorphism of models of type theory* $f : \mathcal{M} \to \mathcal{N}$ consists of the following data:

1. a function $\mathsf{Cx}_f : \mathsf{Cx}_{\mathcal{M}} \to \mathsf{Cx}_{\mathcal{N}}$,
2. for each $\Delta, \Gamma \in \mathsf{Cx}_{\mathcal{M}}$, a function $\mathsf{Sb}_f(\Delta, \Gamma) : \mathsf{Sb}_{\mathcal{M}}(\Delta, \Gamma) \to \mathsf{Sb}_{\mathcal{N}}(\mathsf{Cx}_f(\Delta), \mathsf{Cx}_f(\Gamma))$,

3. for each $\Gamma \in \mathsf{Cx}_{\mathcal{M}}$, a function $\mathsf{Ty}_f(\Gamma) : \mathsf{Ty}_{\mathcal{M}}(\Gamma) \to \mathsf{Ty}_{\mathcal{N}}(\mathsf{Cx}_f(\Gamma))$, and
4. for each $\Gamma \in \mathsf{Cx}_{\mathcal{M}}$ and $A \in \mathsf{Ty}_{\mathcal{M}}(\Gamma)$, a function $\mathsf{Tm}_f(\Gamma, A) : \mathsf{Tm}_{\mathcal{M}}(\Gamma, A) \to \mathsf{Tm}_{\mathcal{N}}(\mathsf{Cx}_f(\Gamma), \mathsf{Ty}_f(\Gamma)(A))$,
5. such that $\mathsf{Cx}_f(\mathbf{1}_{\mathcal{M}}) = \mathbf{1}_{\mathcal{N}}$,
6. and every other context, substitution, type, and term forming operation of $\mathcal{M}$ is also sent to the corresponding operation of $\mathcal{N}$ in a similar fashion.

Definition 3.4.4. The sets Cx, $\mathsf{Sb}(\Delta, \Gamma)$, $\mathsf{Ty}(\Gamma)$, and $\mathsf{Tm}(\Gamma, A)$, equipped with the context, substitution, type, and term forming operations described in Appendix A, tautologically form a model of type theory $\mathcal{T}$ known as the *syntactic model*.

Theorem 3.4.5. *The syntactic model $\mathcal{T}$ is the* initial model of type theory*; that is, for any model of type theory $\mathcal{M}$, there exists a unique homomorphism of models $\mathcal{T} \to \mathcal{M}$.*

The notions of model and homomorphism are quite complex, but they are mechanically derivable from the rules of type theory as presented in Appendix A, viewed as the signature of a quotient inductive-inductive type (QIIT) [KKA19] or generalized algebraic theory (GAT) [Car86]. The initiality of the syntactic model expresses the fact that type theory is the "least" model of type theory, in the sense that it—by definition—satisfies all the rules of type theory and no others; this mirrors the sense in which initiality of $\mathbb{N}$ with respect to $(\mathbf{1} \sqcup -)$-algebras expresses that the natural numbers are generated by **zero** and **suc**$(-)$. Readers interested in learning more about GATs/QIITs and the proof of Theorem 3.4.5 are encouraged to consult Appendix B.

Remark 3.4.6. Theorem 3.4.5 should be regarded as stating the soundness and completeness of type theory with respect to this notion of model. The homomorphism $\mathcal{T} \to \mathcal{M}$ expresses soundness: the syntax of type theory can be interpreted into any model $\mathcal{M}$. Conversely, the fact that the syntax constitutes a model $\mathcal{T}$ expresses completeness: any result that holds for all models must in particular hold for $\mathcal{T}$ and thence for the syntax.

We note that Definitions 3.4.2 and 3.4.3 were carefully chosen so as to make soundness and completeness nearly tautological, and indeed, this is evidenced by the fact that these definitions and theorems can be mechanically derived by the general machinery of generalized algebraic theories. Unimpressed readers may commiserate with Girard's "broccoli logic" critique of such semantics [Gir99]. ⋄

While the definition of a model does not lend much insight into type theory on its own, the model theory of type theory is an essential tool in the metatheorist's toolbox; to prove any property of the syntactic model $\mathcal{T}$, we simply produce a model of type

theory $\mathcal{M}$ such that Theorem 3.4.5 implies the property in question. In the case of consistency, it suffices to exhibit any non-trivial model of type theory whatsoever.

Theorem 3.4.7. *Suppose there exists a model of type theory $\mathcal{M}$ where $\mathsf{Tm}_{\mathcal{M}}(\mathbf{1}_{\mathcal{M}}, \mathbf{Void}_{\mathcal{M}})$ is empty; then type theory is consistent.*

Proof. We must show that from the existence of $\mathcal{M}$ and a term $a \in \mathsf{Tm}(\mathbf{1}, \mathbf{Void})$ we can derive a contradiction. By Theorem 3.4.5, there is a homomorphism of models $f : \mathcal{T} \to \mathcal{M}$, and in particular a function $\mathsf{Tm}_f(\mathbf{1}, \mathbf{Void}) : \mathsf{Tm}(\mathbf{1}, \mathbf{Void}) \to \mathsf{Tm}_{\mathcal{M}}(\mathbf{1}_{\mathcal{M}}, \mathbf{Void}_{\mathcal{M}})$; applying this to a produces an element of $\mathsf{Tm}_{\mathcal{M}}(\mathbf{1}_{\mathcal{M}}, \mathbf{Void}_{\mathcal{M}})$, an empty set. □

Exercise 3.8. Show that Theorem 3.4.7 can actually be upgraded to a biimplication.

In Section 3.5 we will see that there is a "standard" set-theoretic model $\mathcal{S}$ of extensional type theory in which contexts are sets, types are families of sets indexed by their context, and each type former is interpreted as the corresponding construction on indexed sets. As a trivial corollary of this model and Theorem 3.4.7, we obtain the consistency of extensional type theory. We postpone further details of the set-theoretic model to Section 3.5, and postpone our discussion of other models to Chapter 6.

Theorem 3.4.8 (Martin-Löf [Mar84b])**.** *Extensional type theory is consistent.*

Note that while an inconsistent type theory is useless as a logic, it may still be useful for programming; indeed, many modern functional programming languages include some limited forms of dependent types despite being inconsistent.

Exercise 3.9. Consider an unrestricted fixed-point operator $\mathsf{fix} : (A \to A) \to A$, i.e.,

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash a : A[\mathbf{p}]}{\Gamma \vdash \mathsf{fix}(a) : A}$$ ✎

Show that adding such a rule results in an inconsistent type theory.

Our final metatheorem connects directly to the interpretation of type theory as a programming language, although the connection may not be immediately apparent.

Definition 3.4.9. A type theory enjoys *canonicity* if for every closed $\mathbf{1} \vdash b : \mathbf{Bool}$ either $\mathbf{1} \vdash b = \mathbf{true} : \mathbf{Bool}$ or $\mathbf{1} \vdash b = \mathbf{false} : \mathbf{Bool}$, but not both.

Remark 3.4.10. Another common statement of canonicity is that for every closed $\mathbf{1} \vdash n : \mathbf{Nat}$ either $\mathbf{1} \vdash n = \mathbf{zero} : \mathbf{Nat}$ or $\mathbf{1} \vdash n = \mathbf{suc}(m) : \mathbf{Nat}$ where $\mathbf{1} \vdash m : \mathbf{Nat}$. This statement is not equivalent to Definition 3.4.9 in general, but in practice one only considers type theories that satisfy both or neither, and proofs of one also imply the other *en passant*. ◇

Remark 3.4.11. Consistency states that $\mathsf{Tm}(\mathbf{1}, \mathbf{Void}) \cong \emptyset$, whereas canonicity states that $\mathsf{Tm}(\mathbf{1}, \mathbf{Bool}) \cong \{\star, \star'\}$ and $\mathsf{Tm}(\mathbf{1}, \mathbf{Nat}) \cong \mathbb{N}$. As discussed at length in Section 2.5, none of these properties hold in Γ because variables can produce noncanonical terms at any type; however, there are indeed no noncanonical *closed* terms of type **Void**, **Bool**, or **Nat**. ◇

Theorem 3.4.12. *Extensional type theory enjoys canonicity.*

Proof. See Section 6.7. □

Frustratingly, although Theorem 3.4.12 was certainly known to researchers in the 1970s and 1980s, we are unable to locate a precise reference from that time period.

Like consistency, normalization, and invertibility of type constructors, canonicity can be established by constructing a model of type theory, although the proofs of the latter three metatheorems are considerably more involved than the proof of consistency. As we will see in Section 6.7, canonicity models interpret the contexts, substitutions, types, and terms of type theory as pairs of that syntactic object along with additional data which explains how that object may be placed in canonical form [Fre78; LS88; MS93; Cro94; Fio02; AK16; Coq19; KHS19]. Such models can be seen as *displayed models* of type theory over the syntactic model, and are called *gluing models* in the categorical literature.

Exercise 3.10. In light of Remark 3.4.11, we might imagine that canonicity follows from the existence of a model of type theory $\mathcal{M}$ for which $\mathsf{Tm}_{\mathcal{M}}(\mathbf{1}_{\mathcal{M}}, \mathbf{Bool}_{\mathcal{M}})$ has exactly two elements. This is not the case; why? (Why can't we mimic the proof of Theorem 3.4.7?)

The force of canonicity is that it implies the existence of an "evaluation" algorithm that, given a closed boolean $\mathbf{1} \vdash a : \mathbf{Bool}$, reports whether a is equal to **true** or to **false**. There are two ways to obtain such an algorithm; the first is to prove canonicity in a constructive metatheory, so that the proof itself constitutes such an algorithm. The second is to appeal to Markov's principle: because derivation trees are recursively enumerable, a classical proof of canonicity implies that the naïve enumeration algorithm will terminate.

In a direct sense, such an algorithm is indeed an *interpreter* for closed terms of type theory. But canonicity also produces a much richer notion of computational adequacy for type theory; giving this theory its due weight would take us too far afield, but we will briefly sketch the highlights. By results in categorical realizability [Jac99; vOos08], essentially every model of computation gives rise to a highly structured and well-behaved category known as a *realizability topos*; these categories support models of dependent type theory in which terms of type **Bool** are (equivalence classes of) boolean

computations in some idealized model of computation. For instance, in the *effective topos* [Hyl82], closed terms of type **Bool** are equivalence classes of Turing machines modulo Kleene equivalence (i.e., two machines are equivalent if they coterminate with the same value).

Because models of type theory in realizability topoi interpret terms in concrete (albeit theoretical) notions of computation such as Turing machines or combinator calculi, they can be regarded abstractly as *compilers* for type theory. Alternatively, they serve to justify the *program extraction* mechanisms found in proof assistants such as Rocq and Agda, which associate to each term an OCaml or Haskell program whose observable behavior is compatible with the definitional equality of type theory.

From this perspective, canonicity guarantees that definitional equality fully constrains the observable behaviors of extracted programs: for any closed boolean $\mathbf{1} \vdash b : \mathbf{Bool}$, every possible extract of b must evaluate to (the extract of) either **true** or **false**, as predetermined by whether $b = \mathbf{true}$ or $b = \mathbf{false}$. Note that it is still possible for two different extracts of b to have very different execution traces; canonicity only constrains their observable behavior, considered modulo some appropriate notion of observational equivalence.

Remark 3.4.13. The above discussion may clarify why canonicity is harder to prove than consistency: consistency implies the *existence* of a non-trivial model of type theory, whereas canonicity places a constraint on *all* models of type theory. ◇

We emphasize once more that, unlike normalization and invertibility of type constructors, neither consistency nor canonicity is required to implement a bidirectional type-checker for type theory. However, it seems safe to assume that anybody writing such a type-checker is interested in type theory's applications to logic or programming or both, in which case consistency and canonicity are relevant properties. In addition, failures of canonicity often indicate a paucity of definitional equalities that can have a negative effect on the usability of a type theory even as a logic.

3.5⋆ *The set model of type theory*

We now spell out the details of the set-theoretic model $\mathcal{S}$ of extensional type theory alluded to in Section 3.4 [Hof97]. Although the details are somewhat complicated, understanding $\mathcal{S}$ is important for readers who are interested in better understanding the model theory of type theory or how type theory relates to traditional mathematics. It is also one of the original models of type theory—the earliest published description is given by Salvesen [Sal84, Chapter 5], but was well-known folklore prior to this [Pal22].

In short, $\mathcal{S}$ interprets the contexts of type theory as sets, substitutions as functions,

dependent types as indexed families of sets, terms as indexed families of elements, and every type- and term-forming operation as its "standard" mathematical counterpart. For example, the $\mathcal{S}$-interpretation of the closed functions from **Nat** to **Nat**, $\mathsf{Tm}_{\mathcal{S}}(\mathbf{1}_{\mathcal{S}}, \mathbf{\Pi}_{\mathcal{S}}(\mathbf{Nat}_{\mathcal{S}}, \mathbf{Nat}_{\mathcal{S}}))$, is (isomorphic to) the set of ordinary mathematical functions $\mathbb{N} \to \mathbb{N}$.

The main subtlety in defining $\mathcal{S}$ is that we would like the set $\mathsf{Cx}_{\mathcal{S}}$ of $\mathcal{S}$-contexts to be "the collection of all sets," but this collection is unfortunately not a set: by Russell's paradox, having a "set of all sets including itself" leads to contradiction. To properly circumvent this issue we must introduce the notion of *Grothendieck universes*, the set-theoretic cousins of the type-theoretic universes introduced in Section 2.6.

3.5.1 Grothendieck universes

Grothendieck universes are sets that resemble a "set of all sets" without falling victim to Russell's paradox. Roughly speaking, they are collections of sets that are closed under all the operations of set theory: they contain $\emptyset$ and are closed under formation of powersets, unions, set comprehensions, and so forth.

Definition 3.5.1. A *Grothendieck universe* $\mathcal{V}$ is a set with the following properties:

1. $\emptyset \in \mathcal{V}$.
2. *Transitivity:* If $X \in \mathcal{V}$ and $Y \in X$, then $Y \in \mathcal{V}$.
3. *Closure under powersets:* If $X \in \mathcal{V}$ then $\mathcal{P}(X) \in \mathcal{V}$.
4. *Closure under indexed unions:* If $X \in \mathcal{V}$ and $f : X \to \mathcal{V}$, then $\bigcup_{x \in X} f(x) \in \mathcal{V}$.
5. $\mathbb{N} \in \mathcal{V}$. (This condition is omitted by many authors.)

We admit that Definition 3.5.1 is somewhat mysterious; unfortunately, thoroughly justifying these axioms is beyond the scope of this book. We refer the reader to Shulman [Shu08] for a reference which assumes relatively little set-theoretic background.

For our purposes, the axioms of Grothendieck universes have three important features. First, all the closure properties of Grothendieck universes are closure properties of sets: replacing $X \in \mathcal{V}$ with "X is a set," it is true that $\emptyset$ and $\mathbb{N}$ are sets, and that sets are transitive and closed under powersets and indexed unions. In other words, the collection of all sets looks like a Grothendieck universe—except that a Grothendieck universe must be a set, which the collection of all sets is not.

Secondly, these closure conditions imply all the other usual closure conditions of sets. For example, $\mathcal{V}$ is also closed under subsets, products, and function spaces,

defined by their standard set-theoretic encodings. We prove a number of these closure conditions below, noting that these are not intended to be exhaustive.

Lemma 3.5.2. *Every Grothendieck universe $\mathcal{V}$ is closed under the following constructions:*

1. Subsets: *If $X \in \mathcal{V}$ and $Y \subseteq X$, then $Y \in \mathcal{V}$.*
2. Binary unions: *If $X, Y \in \mathcal{V}$ then $X \cup Y \in \mathcal{V}$.*
3. Products: *If $X, Y \in \mathcal{V}$ then $X \times Y \in \mathcal{V}$.*
4. Function spaces: *If $X, Y \in \mathcal{V}$ then $X \to Y \in \mathcal{V}$.*
5. Indexed coproducts: *If $X \in \mathcal{V}$ and $f : X \to \mathcal{V}$, then $\sum_{x \in X} f(x) \in \mathcal{V}$.*
6. Indexed products: *If $X \in \mathcal{V}$ and $f : X \to \mathcal{V}$, then $\prod_{x \in X} f(x) \in \mathcal{V}$.*

Proof.

1. This follows directly from $Y \in \mathcal{P}(X) \in \mathcal{V}$ and transitivity.
2. We obtain binary unions as a special case of indexed unions, using the fact that the two-element set $\mathcal{P}(\mathcal{P}(\emptyset)) = \{\emptyset, \{\emptyset\}\}$ is an element of $\mathcal{V}$. Let $f : \mathcal{P}(\mathcal{P}(\emptyset)) \to \mathcal{V}$ be the function sending $\emptyset$ to X and $\{\emptyset\}$ to Y; then we define $X \cup Y := \bigcup_{x \in X} f(x) \in \mathcal{V}$.
3. Following the usual set-theoretic construction, we define $X \times Y$ to be the subset of $\mathcal{P}(\mathcal{P}(X \cup Y))$ consisting of *ordered pairs* (x, y) with $x \in X$ and $y \in Y$, where $(x, y) := \{\{x\}, \{x, y\}\}$. We observe that $X \times Y \in \mathcal{V}$ by the closure of $\mathcal{V}$ under binary unions, powersets, and subsets.
4. Functions $f : X \to Y$ are in bijection with subsets $S \subseteq X \times Y$ satisfying the condition that for all $x \in X$, there exists a unique $y \in Y$ such that the ordered pair (x, y) is in S. We may therefore take the collection of all such S—a subset of $\mathcal{P}(X \times Y)$ and thus an element of $\mathcal{V}$—as the definition of the function space $X \to Y$.
5. We define the indexed disjoint union $\sum_{x \in X} f(x)$ as the subset of $X \times \bigcup_{x \in X} f(x)$ consisting of ordered pairs (x, y) for which $y \in f(x)$.
6. Similarly, we define the indexed product $\prod_{x \in X} f(x)$ as the subset of $X \to \bigcup_{x \in X} f(x)$ consisting of the functions g for which $g(x) \in f(x)$ for all $x \in X$. □

Finally and most importantly, although the existence of Grothendieck universes is independent from the axioms of ordinary (ZFC) set theory, it is consistent to assume that they exist,[3] and the resulting theory is well-understood albeit stronger than ZFC.

[3] In particular, it does not follow from the axioms that $\mathcal{V}$ contains itself.

Advanced Remark 3.5.3. In fact, assuming the existence of a Grothendieck universe $\mathcal{V}$ is exactly the same as assuming the existence of a strongly inaccessible cardinal. This is fairly modest as far as large cardinal axioms are concerned, but it is strong enough that ZFC+$\mathcal{V}$ proves Con(ZFC). Indeed, $\mathcal{V}$ is a model of ZFC! ⋄

Remark 3.5.4. As we will see in Section 3.5.4, one consequence of the set-theoretic model of type theory is the consistency of type theory. By Gödel's incompleteness theorem, constructing this model must require a metatheory stronger than extensional type theory. Although ZFC and extensional type theory are not exactly aligned in strength, we should not be surprised that plain ZFC is too weak. In fact, extensional type theory augmented with an impredicative universe of propositions (Section 2.7) and a few axioms becomes exactly as strong as ZFC with a universe hierarchy [Wer97]. ⋄

In the remainder of Section 3.5, we will rely on an ambient assumption that there is a $(\omega + 1)$-indexed hierarchy of nested Grothendieck universes, in the following sense.

Definition 3.5.5. For a partial order I, an *I-hierarchy of Grothendieck universes* $(\mathcal{V}_i)_{i \in I}$ is a family of Grothendieck universes $\mathcal{V}_i$ such that $\mathcal{V}_i \in \mathcal{V}_j$ whenever $i < j$.

Axiom 3.5.6. There is an $(\omega + 1)$-hierarchy of Grothendieck universes $\mathcal{V}_0 \in \cdots \in \mathcal{V}_\omega$.

Intuitively, Axiom 3.5.6 states that $\mathcal{V}_0$ contains all the sets that exist in ZFC, $\mathcal{V}_1$ contains all the sets of ZFC+$\mathcal{V}_0$, $\mathcal{V}_2$ contains all the sets of ZFC+$\mathcal{V}_0$+$\mathcal{V}_1$, and so forth. One often refers to the sets of ZFC as *small sets* for emphasis, and in general for a Grothendieck universe $\mathcal{V}$ we say that a set X is *$\mathcal{V}$-small* if $X \in \mathcal{V}$. Thus Axiom 3.5.6 equivalently states that small sets are $\mathcal{V}_i$-small and $\mathcal{V}_i$ is $\mathcal{V}_j$-small for all $i < j$.

3.5.2 *The substitution calculus of sets*

Exhibiting a model $\mathcal{M}$ of type theory (Definition 3.4.2) requires an enormous amount of data, but we can break the process down into three steps:

1. First, one must define the sets of $\mathcal{M}$-contexts $\mathsf{Cx}_{\mathcal{M}}$, $\mathcal{M}$-substitutions $\mathsf{Sb}_{\mathcal{M}}(-,-)$, $\mathcal{M}$-types $\mathsf{Ty}_{\mathcal{M}}(-)$, and $\mathcal{M}$-terms $\mathsf{Tm}_{\mathcal{M}}(-,-)$.

2. Next, one must provide the $\mathcal{M}$-interpretations of the rules of the substitution calculus (Section 2.3), the core structure of type theory governing variables and substitutions, and verify that these satisfy the associated equations.

3. Finally, for each connective (**Π**-types, **Void**, $\mathbf{U}_i$, *etc.*) one gives $\mathcal{M}$-interpretations of the associated rules, and again verifies the associated equations.

The steps must be performed in this order, because the choice of sets (*e.g.*, $\mathsf{Cx}_{\mathcal{M}}$) in the first step affects the interpretation of the substitution calculus (*e.g.*, $\mathbf{p}_{\mathcal{M}}$) in the second step, which in turn affects the interpretation of every connective. However, the interpretations of non-**U** connectives do not depend on one another and can be added in any order, because we were careful in Chapter 2 to avoid mentioning (*e.g.*) **Π**-types in the rules for Σ-types.

We will now carry out the first two steps of defining the set model $\mathcal{S}$. By the end of this subsection, we will have a model of a dependent type theory with no connectives, mirroring the situation at the end of Section 2.3.

The basic sets With the machinery of Grothendieck universes (Definition 3.5.1) under our belt, we can now define the basic sets of the $\mathcal{S}$-interpretation of type theory: the $\mathcal{S}$-contexts, $\mathcal{S}$-substitutions, $\mathcal{S}$-types, and $\mathcal{S}$-terms. Rather than defining the set of $\mathcal{S}$-contexts $\mathsf{Cx}_{\mathcal{S}}$ to be the nonexistent "set of all sets," we will define it to be a Grothendieck universe, a set of *some* sets which is closed under all the set-forming operations of set theory. For reasons that will become clear later, we choose the set of $\mathcal{S}$-contexts to be $\mathcal{V}_\omega$, the largest Grothendieck universe asserted by Axiom 3.5.6.

$$\mathsf{Cx}_{\mathcal{S}} \coloneqq \mathcal{V}_\omega$$

For any two $\mathcal{S}$-contexts $\Delta, \Gamma \in \mathsf{Cx}_{\mathcal{S}}$, the set of $\mathcal{S}$-substitutions from Δ to Γ is simply the set of ordinary functions from Δ to Γ:

$$\mathsf{Sb}_{\mathcal{S}}(\Delta, \Gamma) \coloneqq \Delta \to \Gamma \qquad (\Delta, \Gamma \in \mathsf{Cx}_{\mathcal{S}})$$

Notation 3.5.7. Throughout this section, the variables $\Gamma, \gamma, A, a, \ldots$ range over $\mathcal{S}$-contexts, substitutions, types, and terms, *not* syntactic contexts, substitutions, types, and terms as they generally have throughout this book. We believe this notation is the least confusing in the long run, but the reader should proceed cautiously.

Intuitively, an $\mathcal{S}$-type A in $\mathcal{S}$-context Γ should be a family of sets indexed by the set Γ, *i.e.*, a choice of set $A(x)$ for each $x \in \Gamma$. As in our definition of $\mathsf{Cx}_{\mathcal{S}}$, we can obtain a set of such families by restricting all the sets $A(x)$ to be elements of $\mathcal{V}_\omega$:

$$\mathsf{Ty}_{\mathcal{S}}(\Gamma) \coloneqq \Gamma \to \mathcal{V}_\omega \qquad (\Gamma \in \mathsf{Cx}_{\mathcal{S}})$$

Finally, given an $\mathcal{S}$-context $\Gamma \in \mathcal{V}_\omega$ and an $\mathcal{S}$-type $A : \Gamma \to \mathcal{V}_\omega$ in that context, an $\mathcal{S}$-term $a \in \mathsf{Tm}_{\mathcal{S}}(\Gamma, A)$ should be a family of elements of each $A(x)$ for each $x \in \Gamma$. In other words, a should be a dependent function $(x : \Gamma) \to A(x)$, where $a(x) \in A(x)$ for all $x \in \Gamma$. Set-theoretically, such functions are more commonly understood as elements of the Γ-indexed product of the sets $A(-)$; see Remarks 2.4.1 and 2.4.5.

$$\mathsf{Tm}_{\mathcal{S}}(\Gamma, A) \coloneqq \textstyle\prod_{x \in \Gamma} A(x) \qquad (\Gamma \in \mathsf{Cx}_{\mathcal{S}}, A \in \mathsf{Ty}_{\mathcal{S}}(\Gamma))$$

Summing up, we define $\mathcal{S}$-contexts as ($\mathcal{V}_\omega$-small) sets, $\mathcal{S}$-substitutions as functions, $\mathcal{S}$-types as indexed families of ($\mathcal{V}_\omega$-small) sets, and $\mathcal{S}$-terms as indexed families of elements.

The category of substitutions Having now defined the basic sets underlying the $\mathcal{S}$-interpretation of type theory, our next task is to define the operations of the substitution calculus (collected in the first section of Appendix A), starting with the identity and composition of substitutions.

For every $\mathcal{S}$-context $\Gamma \in \mathsf{Cx}_{\mathcal{S}}$, we must define an identity $\mathcal{S}$-substitution $\mathbf{id}_{\mathcal{S}}$ in $\mathsf{Sb}_{\mathcal{S}}(\Gamma, \Gamma)$. Unfolding the definitions of $\mathsf{Cx}_{\mathcal{S}}$ and $\mathsf{Sb}_{\mathcal{S}}(\Gamma, \Gamma)$, this is for every $\Gamma \in \mathcal{V}_\omega$ a function $\Gamma \to \Gamma$, which we can simply take to be the identity function:

$$\begin{aligned} &\mathbf{id}_{\mathcal{S}} : \textstyle\prod_{\Gamma \in \mathcal{V}_\omega} \Gamma \to \Gamma \\ &\mathbf{id}_{\mathcal{S}}\ \Gamma\ x \coloneqq x \end{aligned}$$

Next, given any $\Gamma_0, \Gamma_1, \Gamma_2 \in \mathsf{Cx}_{\mathcal{S}}$, $\gamma_1 \in \mathsf{Sb}_{\mathcal{S}}(\Gamma_2, \Gamma_1)$, and $\gamma_0 \in \mathsf{Sb}_{\mathcal{S}}(\Gamma_1, \Gamma_0)$ we must define the composite $\mathcal{S}$-substitution $\gamma_0 \circ_{\mathcal{S}} \gamma_1 \in \mathsf{Sb}_{\mathcal{S}}(\Gamma_2, \Gamma_0)$, namely by function composition:

$$\begin{aligned} &_\circ_{\mathcal{S}}_ : \textstyle\prod_{\Gamma_0, \Gamma_1, \Gamma_2 \in \mathsf{Cx}_{\mathcal{S}}} \mathsf{Sb}_{\mathcal{S}}(\Gamma_1, \Gamma_0) \to \mathsf{Sb}_{\mathcal{S}}(\Gamma_2, \Gamma_1) \to \mathsf{Sb}_{\mathcal{S}}(\Gamma_2, \Gamma_0) \\ &(\gamma_0 \circ_{\mathcal{S}} \gamma_1)(x) \coloneqq \gamma_0(\gamma_1(x)) \end{aligned}$$

Notation 3.5.8. Starting with the above definition, we suppress unambiguous arguments for clarity: in this case, the $\mathcal{S}$-contexts $\Gamma_0, \Gamma_1, \Gamma_2$.

In the substitution calculus, identity and composition satisfy various equations, namely that composition is associative with identity as a left and right unit. We must therefore verify that our definitions of $\mathcal{S}$-identity and $\mathcal{S}$-composition validate the same equations:

Exercise 3.11. Verify the following equations:

- For all $\gamma \in \mathsf{Sb}_{\mathcal{S}}(\Delta, \Gamma)$, $\mathbf{id}_{\mathcal{S}} \circ_{\mathcal{S}} \gamma = \gamma = \gamma \circ_{\mathcal{S}} \mathbf{id}_{\mathcal{S}}$.
- For all $\gamma_2 \in \mathsf{Sb}_{\mathcal{S}}(\Gamma_3, \Gamma_2)$, $\gamma_1 \in \mathsf{Sb}_{\mathcal{S}}(\Gamma_2, \Gamma_1)$, and $\gamma_0 \in \mathsf{Sb}_{\mathcal{S}}(\Gamma_1, \Gamma_0)$, $\gamma_0 \circ_{\mathcal{S}} (\gamma_1 \circ_{\mathcal{S}} \gamma_2) = (\gamma_0 \circ_{\mathcal{S}} \gamma_1) \circ_{\mathcal{S}} \gamma_2$.

The empty context Next we define the empty $\mathcal{S}$-context $\mathbf{1}_{\mathcal{S}} \in \mathsf{Cx}_{\mathcal{S}}$ and the terminal $\mathcal{S}$-substitution $!_{\mathcal{S}} \in \mathsf{Sb}_{\mathcal{S}}(\Gamma, \mathbf{1}_{\mathcal{S}})$ for every $\Gamma \in \mathsf{Cx}_{\mathcal{S}}$. Notably, although we call $\mathbf{1}$ the *empty* context, it is in fact interpreted as a *one-element* set.

$$\mathbf{1}_{\mathcal{S}} \in \mathcal{V}_\omega$$
$$\mathbf{1}_{\mathcal{S}} := \{\star\}$$

Remark 3.5.9. We write $\{\star\}$ to emphasize that it does not matter which one-element set in $\mathcal{V}_\omega$ we choose. The most natural concrete choice of one-element set is perhaps $\{\emptyset\}$, which we note is an element of $\mathcal{V}_\omega$ by axioms (1), (2) and (3) of Definition 3.5.1. ◇

Exercise 3.12. In light of the definition of $\mathbf{1}_{\mathcal{S}}$ above, show that closed $\mathcal{S}$-types are just sets and closed $\mathcal{S}$-terms are just elements of those sets. To be precise, construct isomorphisms $\iota : \mathsf{Ty}_{\mathcal{S}}(\mathbf{1}_{\mathcal{S}}) \cong \mathcal{V}_\omega$ and $\kappa_A : \mathsf{Tm}_{\mathcal{S}}(\mathbf{1}_{\mathcal{S}}, A) \cong \iota(A)$ for all $A \in \mathsf{Ty}_{\mathcal{S}}(\mathbf{1}_{\mathcal{S}})$.

The terminal $\mathcal{S}$-substitution into $\mathbf{1}_{\mathcal{S}}$ is the constant function returning $\star$.

$$!_{\mathcal{S}} : \textstyle\prod_{\Gamma \in \mathsf{Cx}_{\mathcal{S}}} \mathsf{Sb}_{\mathcal{S}}(\Gamma, \mathbf{1}_{\mathcal{S}})$$
$$!_{\mathcal{S}}(x) := \star$$

We have one equation to check before moving on.

Lemma 3.5.10. *For all $\delta \in \mathsf{Sb}_{\mathcal{S}}(\Gamma, \mathbf{1}_{\mathcal{S}})$, $\delta = !_{\mathcal{S}}$.*

Proof. Unfolding definitions, we see that δ and $!_{\mathcal{S}}$ are both functions $\Gamma \to \{\star\}$. There is only one such function, so they must be equal. □

Applying substitutions Applying an $\mathcal{S}$-substitution $\Delta \to \Gamma$ to an $\mathcal{S}$-type (resp., $\mathcal{S}$-term) in context Γ must produce an $\mathcal{S}$-type (resp., $\mathcal{S}$-term) in context Δ:

$$_[_]_{\mathcal{S}} : \textstyle\prod_{\Delta,\Gamma \in \mathsf{Cx}_{\mathcal{S}}} \prod_{\gamma \in \mathsf{Sb}_{\mathcal{S}}(\Delta,\Gamma)} \mathsf{Ty}_{\mathcal{S}}(\Gamma) \to \mathsf{Ty}_{\mathcal{S}}(\Delta)$$
$$_[_]_{\mathcal{S}} : \textstyle\prod_{\Delta,\Gamma \in \mathsf{Cx}_{\mathcal{S}}} \prod_{\gamma \in \mathsf{Sb}_{\mathcal{S}}(\Delta,\Gamma)} \prod_{A \in \mathsf{Ty}_{\mathcal{S}}(\Gamma)} \mathsf{Tm}_{\mathcal{S}}(\Gamma, A) \to \mathsf{Tm}_{\mathcal{S}}(\Delta, A[\gamma]_{\mathcal{S}})$$

Thankfully, the types of these operations are significantly more intimidating than their definitions. Unfolding definitions in the first line, we must take a function $\gamma : \Delta \to \Gamma$ and a function $A : \Gamma \to \mathcal{V}_\omega$ and produce a function $\Delta \to \mathcal{V}_\omega$, which is easily accomplished by composing A and γ. Substitution on terms is identical:

$$A[\gamma]_{\mathcal{S}} := A \circ \gamma$$
$$a[\gamma]_{\mathcal{S}} := a \circ \gamma$$

The substitution calculus includes a number of equations governing $_[_]$, namely that substituting by $\mathbf{id}$ is the identity and substituting by a composite substitution is the same as a composition of substitutions; checking these for $\mathcal{S}$ is again straightforward.

Exercise 3.13. Verify the following, where $\Gamma \in \mathsf{Cx}_{\mathcal{S}}$, $A \in \mathsf{Ty}_{\mathcal{S}}(\Gamma)$, and $a \in \mathsf{Tm}_{\mathcal{S}}(\Gamma, A)$:

- $A[\mathbf{id}_{\mathcal{S}}]_{\mathcal{S}} = A$.
- $a[\mathbf{id}_{\mathcal{S}}]_{\mathcal{S}} = a$.
- If $\gamma_1 \in \mathsf{Sb}_{\mathcal{S}}(\Gamma_2, \Gamma_1)$ and $\gamma_0 \in \mathsf{Sb}_{\mathcal{S}}(\Gamma_1, \Gamma)$, then $A[\gamma_0 \circ_{\mathcal{S}} \gamma_1]_{\mathcal{S}} = A[\gamma_0]_{\mathcal{S}}[\gamma_1]_{\mathcal{S}}$.
- If $\gamma_1 \in \mathsf{Sb}_{\mathcal{S}}(\Gamma_2, \Gamma_1)$ and $\gamma_0 \in \mathsf{Sb}_{\mathcal{S}}(\Gamma_1, \Gamma)$, then $a[\gamma_0 \circ_{\mathcal{S}} \gamma_1]_{\mathcal{S}} = a[\gamma_0]_{\mathcal{S}}[\gamma_1]_{\mathcal{S}}$.

Extending contexts The remaining operations of the substitution calculus are context extension $\Gamma.A$, substitution extension $\gamma.a$, the weakening substitution $\mathbf{p}$, and the variable term $\mathbf{q}$. We must start by defining the $\mathcal{S}$-interpretation of context extension, because it occurs in the types of all the other operations.

Recall from Sections 2.3 and 2.4.2 that substitutions into $\Gamma.A$ are roughly "pairs of a substitution into Γ and a term of type A." More precisely, there is a natural isomorphism between substitutions $\gamma \in \mathsf{Sb}(\Delta, \Gamma.A)$ and pairs $(\gamma_0 \in \mathsf{Sb}(\Delta, \Gamma), a \in \mathsf{Tm}(\Delta, A[\gamma_0]))$. Unfolding $\mathcal{S}$-interpretations and setting $\Delta = \mathbf{1}_{\mathcal{S}}$, in light of Exercise 3.12 we see that elements of the set $\Gamma._{\mathcal{S}}A$ must be in bijection with pairs $(x_0 \in \Gamma, a \in A(x_0))$, and so we might as well take this as the definition of $\Gamma._{\mathcal{S}}A$.

$$\begin{aligned} &_\cdot_{\mathcal{S}}_ : \textstyle\prod_{\Gamma \in \mathsf{Cx}_{\mathcal{S}}} \mathsf{Ty}_{\mathcal{S}}(\Gamma) \to \mathsf{Cx}_{\mathcal{S}} \\ &\Gamma._{\mathcal{S}}A := \textstyle\sum_{x \in \Gamma} A(x) \end{aligned}$$

We must be careful to check that this set is actually an element of $\mathsf{Cx}_{\mathcal{S}} = \mathcal{V}_\omega$, which follows from the closure of Grothendieck universes under indexed coproducts (Lemma 3.5.2).

Once again, to define $\mathbf{p}_{\mathcal{S}}$, $\mathbf{q}_{\mathcal{S}}$, and $_\cdot_{\mathcal{S}}_$ we must unfold their types, which will turn out to be significantly more intimidating than their definitions. Weakening, for example, is simply the first projection from $\sum$:

$$\begin{aligned} &\mathbf{p}_{\mathcal{S}} : \textstyle\prod_{\Gamma \in \mathcal{V}_\omega} \prod_{A \in \Gamma \to \mathcal{V}_\omega} (\sum_{x \in \Gamma} A(x)) \to \Gamma \\ &\mathbf{p}_{\mathcal{S}}(x, a) := x \end{aligned}$$

Similarly, variables and substitution extension are respectively the second projection and pairing operations of $\sum$. For any $\Delta, \Gamma \in \mathcal{V}_\omega$ and $A \in \Gamma \to \mathcal{V}_\omega$:

$$\begin{aligned} &\mathbf{q}_{\mathcal{S}} : \textstyle\prod_{p \in (\sum_{x \in \Gamma} A(x))} A(\mathbf{p}_{\mathcal{S}}(p)) \\ &\mathbf{q}_{\mathcal{S}}(x, a) := a \end{aligned}$$

$$\begin{aligned} &_\cdot_{\mathcal{S}}_ : \textstyle\prod_{\gamma \in \Delta \to \Gamma} (\prod_{y \in \Delta} A(\gamma(y))) \to \Delta \to \sum_{x \in \Gamma} A(x) \\ &(\gamma._{\mathcal{S}}a)(y) := (\gamma(y), a(y)) \end{aligned}$$

Exercise 3.14. Check that the types given above for $\mathbf{p}_{\mathcal{S}}$, $\mathbf{q}_{\mathcal{S}}$, and $_.\mathcal{S}_$ match the types given in Section 2.3, by unfolding the $\mathcal{S}$-interpretations given throughout this section.

The $\mathcal{S}$-interpretations of context extension, substitution extension, weakening, and variables as $\sum$, pairing, first projection, and second projection may in fact clarify the meaning of these operations in the substitution calculus. At any rate, it is straightforward to verify the necessary equations, which correspond to the β- and η-laws of $\sum$.

Lemma 3.5.11. *If $\Delta, \Gamma \in \mathsf{Cx}_{\mathcal{S}}$ and $A \in \mathsf{Ty}_{\mathcal{S}}(\Gamma)$, then:*

- *If $\gamma \in \mathsf{Sb}_{\mathcal{S}}(\Delta, \Gamma)$ and $a \in \mathsf{Tm}_{\mathcal{S}}(\Delta, A[\gamma]_{\mathcal{S}})$, then $\mathbf{p}_{\mathcal{S}} \circ_{\mathcal{S}} (\gamma.\mathcal{S}a) = \gamma$.*
- *If $\gamma \in \mathsf{Sb}_{\mathcal{S}}(\Delta, \Gamma)$ and $a \in \mathsf{Tm}_{\mathcal{S}}(\Delta, A[\gamma]_{\mathcal{S}})$, then $\mathbf{q}_{\mathcal{S}}[\gamma.\mathcal{S}a] = a$.*
- *If $\gamma \in \mathsf{Sb}_{\mathcal{S}}(\Delta, \Gamma.\mathcal{S}A)$ then $\gamma = (\mathbf{p}_{\mathcal{S}} \circ_{\mathcal{S}} \gamma).\mathcal{S}(\mathbf{q}_{\mathcal{S}}[\gamma]_{\mathcal{S}})$.*

Proof. These all follow essentially by definition. For the first equation, fix $\gamma : \Delta \to \Gamma$ and $a \in \prod_{y \in \Delta} A(\gamma(y))$; we must show $\pi_1 \circ (\lambda y \to (\gamma(y), a(y))) = \gamma$. Because both sides are functions, it suffices to check that they agree on all $y \in \Delta$, and indeed both produce $\gamma(y)$ when applied to y. For the second equation we must show $\pi_2 \circ (\lambda y \to (\gamma(y), a(y))) = a$, which again follows by applying both sides to $y \in \Delta$.

For the third, fix $\gamma : \Delta \to \sum_{x \in \Gamma} A(x)$ and show $\gamma = \lambda y \to (\pi_1(\gamma(y)), \pi_2(\gamma(y)))$. This follows by applying both sides to $y \in \Delta$ and noting that $\gamma(y) \in \sum_{x \in \Gamma} A(x)$ is by definition of the form (x_0, a). □

The reader should now verify that we have provided an $\mathcal{S}$-interpretation of every rule of the substitution calculus, covering the first section of Appendix A.

Notation 3.5.12. We note that we can safely reuse notations from Chapter 2 for their $\mathcal{S}$ counterparts. In particular, following Lemma 2.3.6, we write $\gamma.\mathcal{S}A$ for $(\gamma \circ_{\mathcal{S}} \mathbf{p}_{\mathcal{S}}).\mathcal{S}(\mathbf{q}_{\mathcal{S}})_{\mathcal{S}}$.

3.5.3 *The type-theoretic connectives of sets*

Now that we have defined the $\mathcal{S}$-interpretation of the basic structure of type theory, we can extend $\mathcal{S}$ with any connectives of our choice. Unlike the operations considered in Section 3.5.2, the connectives of type theory are (generally) defined independently of one another, allowing us to model them in a modular fashion. We consider some representative cases, namely, the $\mathcal{S}$-interpretations of **Π**-types, **Eq**-types, **Void**, **Bool**, and $\mathbf{U}_0$.

Π-types Taking advantage of the compact representation of the rules of Π-types introduced in Section 2.4.2, the $\mathcal{S}$-interpretation of Π-types consists of an $\mathcal{S}$-type-forming operation and a family of isomorphisms of sets:

$$\mathbf{\Pi}_{\mathcal{S}} : \textstyle\prod_{\Gamma \in \mathsf{Cx}_{\mathcal{S}}} (\sum_{A \in \mathsf{Ty}_{\mathcal{S}}(\Gamma)} \mathsf{Ty}_{\mathcal{S}}(\Gamma.{}_{\mathcal{S}}A)) \to \mathsf{Ty}_{\mathcal{S}}(\Gamma)$$
$$\iota_{\mathcal{S}} : \textstyle\prod_{\Gamma \in \mathsf{Cx}_{\mathcal{S}}} \prod_{A \in \mathsf{Ty}_{\mathcal{S}}(\Gamma)} \prod_{B \in \mathsf{Ty}_{\mathcal{S}}(\Gamma.{}_{\mathcal{S}}A)} \mathsf{Tm}_{\mathcal{S}}(\Gamma, \mathbf{\Pi}_{\mathcal{S}}\ \Gamma\ (A, B)) \cong \mathsf{Tm}_{\mathcal{S}}(\Gamma.{}_{\mathcal{S}}A, B)$$

subject to the following equations expressing their naturality in $\Gamma \in \mathsf{Cx}_{\mathcal{S}}$:

$$(\mathbf{\Pi}_{\mathcal{S}}\ \Gamma\ (A, B))[\gamma]_{\mathcal{S}} = \mathbf{\Pi}_{\mathcal{S}}\ \Delta\ (A[\gamma]_{\mathcal{S}}, B[\gamma.{}_{\mathcal{S}}A]_{\mathcal{S}}) \qquad (\gamma \in \mathsf{Sb}_{\mathcal{S}}(\Delta, \Gamma))$$
$$(\iota_{\mathcal{S}}\ \Gamma\ A\ B\ f)[\gamma.{}_{\mathcal{S}}A]_{\mathcal{S}} = \iota_{\mathcal{S}}\ \Delta\ (A[\gamma]_{\mathcal{S}})\ (B[\gamma.{}_{\mathcal{S}}A]_{\mathcal{S}})\ (f[\gamma]_{\mathcal{S}}) \qquad (\gamma \in \mathsf{Sb}_{\mathcal{S}}(\Delta, \Gamma))$$

To get a handle on the situation, let us consider the types of $\mathbf{\Pi}_{\mathcal{S}}$ and $\iota_{\mathcal{S}}$ when specialized to the empty context $\mathbf{1}_{\mathcal{S}}$, and simplified along the isomorphisms of Exercise 3.12:

$$\mathbf{\Pi}_{\mathcal{S}}\ \mathbf{1}_{\mathcal{S}} : (\textstyle\sum_{A \in \mathcal{V}_\omega} (A \to \mathcal{V}_\omega)) \to \mathcal{V}_\omega$$
$$\iota_{\mathcal{S}}\ \mathbf{1}_{\mathcal{S}} : \textstyle\prod_{A \in \mathcal{V}_\omega} \prod_{B \in A \to \mathcal{V}_\omega} \mathbf{\Pi}_{\mathcal{S}}\ \mathbf{1}_{\mathcal{S}}\ (A, B) \cong \prod_{a \in A} B(a)$$

That is, in the empty context, for any $A \in \mathcal{V}_\omega$ and $B : A \to \mathcal{V}_\omega$ we must choose a set $\mathbf{\Pi}_{\mathcal{S}}\ \mathbf{1}_{\mathcal{S}}\ (A, B) \in \mathcal{V}_\omega$ to serve as the $\mathcal{S}$-Π-type of A and B, and this set must be isomorphic to the set-theoretic indexed product $\prod_{a \in A} B(a)$.

The situation for arbitrary contexts is essentially the same, except that all three of A, B, and their $\mathcal{S}$-Π-type are additionally indexed by a set Γ. We define $\mathbf{\Pi}_{\mathcal{S}}$ as follows:

$$\mathbf{\Pi}_{\mathcal{S}}\ \Gamma\ (A, B)\ x := \textstyle\prod_{a \in A(x)} B(x, a) \qquad (x \in \Gamma)$$

noting that $B : (\sum_{x \in \Gamma} A(x)) \to \mathcal{V}_\omega$ by the definition of $\Gamma.{}_{\mathcal{S}}A$ in Section 3.5.2. Finally, we must verify that our definition $\prod_{a \in A(x)} B(x, a) \in \mathcal{V}_\omega$, which indeed holds by Lemma 3.5.2.

Lemma 3.5.13. *$\mathbf{\Pi}_{\mathcal{S}}$ is natural in Γ,* i.e., *$(\mathbf{\Pi}_{\mathcal{S}}\ \Gamma\ (A, B))[\gamma]_{\mathcal{S}} = \mathbf{\Pi}_{\mathcal{S}}\ \Delta\ (A[\gamma]_{\mathcal{S}}, B[\gamma.{}_{\mathcal{S}}A]_{\mathcal{S}})$ in $\mathsf{Ty}_{\mathcal{S}}(\Delta)$ for any $\gamma \in \mathsf{Sb}_{\mathcal{S}}(\Delta, \Gamma)$.*

Proof. Unfolding the operations of the substitution calculus, we must show:

$$(\mathbf{\Pi}_{\mathcal{S}}\ \Gamma\ (A, B)) \circ \gamma = \mathbf{\Pi}_{\mathcal{S}}\ \Delta\ (A \circ \gamma, \lambda(y, a) \to B(\gamma(y), a))$$

These are both functions $\Delta \to \mathcal{V}_\omega$, so it suffices to check that they agree on all $y \in \Delta$:

$$\begin{aligned}
&((\mathbf{\Pi}_{\mathcal{S}}\ \Gamma\ (A, B)) \circ \gamma)(y) \\
&= \mathbf{\Pi}_{\mathcal{S}}\ \Gamma\ (A, B)\ (\gamma(y)) \\
&= \textstyle\prod_{a \in A(\gamma(y))} B(\gamma(y), a) \\
&= \textstyle\prod_{a \in (A \circ \gamma)(y)} (\lambda(y, a) \to B(\gamma(y), a))(y, a) \\
&= \mathbf{\Pi}_{\mathcal{S}}\ \Delta\ (A \circ \gamma, \lambda(y, a) \to B(\gamma(y), a))\ y
\end{aligned}$$

□

As for the isomorphism $\iota_{\mathcal{S}}$, unfolding definitions we must construct:

$$\iota_{\mathcal{S}} : \prod_{\Gamma \in \mathcal{V}_\omega} \prod_{A \in \Gamma \to \mathcal{V}_\omega} \prod_{B \in (\sum_{x \in \Gamma} A(x)) \to \mathcal{V}_\omega} (\prod_{x \in \Gamma} \prod_{a \in A(x)} B(x,a)) \cong \prod_{p \in (\sum_{x \in \Gamma} A(x))} B(p)$$

Fixing Γ, A, B, this isomorphism is simply the dependent (un)currying isomorphism $(x : \Gamma) \to (a : A(x)) \to B(x,a) \cong (p : \sum_{x:\Gamma} A(x)) \to B(p)$, defined as follows:

$$\iota_{\mathcal{S}}\ \Gamma\ A\ B\ f\ (x,a) := f\ x\ a$$
$$\iota^{-1}_{\mathcal{S}}\ \Gamma\ A\ B\ g\ x\ a := g\ (x,a)$$

Exercise 3.15. Verify that $\iota_{\mathcal{S}}$ and $\iota^{-1}_{\mathcal{S}}$ are inverses.

Exercise 3.16. Verify that $\iota_{\mathcal{S}}$ is natural in Γ. (Hint: show that

$$(\iota_{\mathcal{S}}\ \Gamma\ A\ B\ f) \circ (\lambda(y,a) \to (\gamma(y), a)) = \iota_{\mathcal{S}}\ \Delta\ (A \circ \gamma)\ (\lambda(y,a) \to B(\gamma(y), a))\ (f \circ \gamma)$$

for any $f \in \prod_{x \in \Gamma} \prod_{a \in A(x)} B(x,a)$ and $\gamma \in \mathrm{Sb}_{\mathcal{S}}(\Delta, \Gamma)$, by showing that they agree on all $(y,a) \in \sum_{y \in \Delta} A(\gamma(y))$.)

Eq-types The $\mathcal{S}$-interpretation of extensional equality types is analogous to that of $\mathbf{\Pi}$-types. Following Section 2.4.4, we must define an $\mathcal{S}$-type-forming operation and a family of isomorphisms of sets, both natural in Γ:

$$\mathbf{Eq}_{\mathcal{S}} : \prod_{\Gamma \in \mathsf{Cx}_{\mathcal{S}}} (\sum_{A \in \mathsf{Ty}_{\mathcal{S}}(\Gamma)} \mathsf{Tm}_{\mathcal{S}}(\Gamma, A) \times \mathsf{Tm}_{\mathcal{S}}(\Gamma, A)) \to \mathsf{Ty}_{\mathcal{S}}(\Gamma)$$
$$\iota_{\mathcal{S}} : \prod_{\Gamma \in \mathsf{Cx}_{\mathcal{S}}} \prod_{A \in \mathsf{Ty}_{\mathcal{S}}(\Gamma)} \prod_{a,b \in \mathsf{Tm}_{\mathcal{S}}(\Gamma, A)} \mathsf{Tm}_{\mathcal{S}}(\Gamma, \mathbf{Eq}_{\mathcal{S}}\ \Gamma\ (A,a,b)) \cong \{\star \mid a = b\}$$

We define $\mathbf{Eq}_{\mathcal{S}}\ \Gamma\ (A,a,b)$ to be the Γ-indexed family of sets that maps $x \in \Gamma$ to a one-element set when $a(x) = b(x) \in A(x)$, and an empty set otherwise.

$$\mathbf{Eq}_{\mathcal{S}}\ \Gamma\ (A,a,b)\ x := \{\star \mid a(x) = b(x)\}$$

To define $\iota_{\mathcal{S}}$, we note that $\mathcal{S}$-terms $e \in \mathsf{Tm}_{\mathcal{S}}(\Gamma, \mathbf{Eq}_{\mathcal{S}}\ \Gamma\ (A,a,b))$ are constant functions sending every $x \in \Gamma$ to the unique element $\star$. In particular, the existence of such an e implies that $a(x) = b(x)$ for all $x \in \Gamma$, and any two such terms e, e' must agree on all $x \in \Gamma$ and thus be equal. Thus:

$$\iota_{\mathcal{S}}\ \Gamma\ A\ a\ b\ e := \star$$
$$\iota^{-1}_{\mathcal{S}}\ \Gamma\ A\ a\ b\ \star\ x := \star$$

Exercise 3.17. Verify that $\iota_{\mathcal{S}}$ and $\iota^{-1}_{\mathcal{S}}$ are inverses.

Exercise 3.18. State and prove the naturality equations for $\mathbf{Eq}_{\mathcal{S}}$ and $\iota_{\mathcal{S}}$. (Hint: reference the naturality equations in Section 2.4.4, and unfold definitions.)

The empty type Our next type **Void** is defined not by a mapping-in property but a mapping-out property. However, as discussed in Section 2.5.1, it can nevertheless be axiomatized as a natural type-forming operation with a natural family of isomorphisms:

$$\begin{aligned}&\mathbf{Void}_{\mathcal{S}} : \textstyle\prod_{\Gamma \in \mathsf{Cx}_{\mathcal{S}}} \mathsf{Ty}_{\mathcal{S}}(\Gamma)\\ &\rho_{\mathcal{S}} : \textstyle\prod_{\Gamma \in \mathsf{Cx}_{\mathcal{S}}} \prod_{A \in \mathsf{Ty}_{\mathcal{S}}(\Gamma ._{\mathcal{S}} \mathbf{Void}_{\mathcal{S}})} \mathsf{Tm}_{\mathcal{S}}(\Gamma ._{\mathcal{S}} \mathbf{Void}_{\mathcal{S}}, A) \cong \{\star\}\end{aligned}$$

Given that **Void** is called the empty type, it is perhaps unsurprising that $\mathcal{S}$ interprets it as the empty set, regarded as a constant family over $\Gamma \in \mathcal{V}_\omega$ and $x \in \Gamma$.

$$\mathbf{Void}_{\mathcal{S}}\ \Gamma\ x := \emptyset$$

Elements of the $\mathcal{S}$-context $\Gamma ._{\mathcal{S}} \mathbf{Void}_{\mathcal{S}}$ are pairs of $x \in \Gamma$ and $y \in \mathbf{Void}_{\mathcal{S}}\ x$, but the latter set is defined to be empty, so no such pairs exist and $\Gamma ._{\mathcal{S}} \mathbf{Void}_{\mathcal{S}} = \emptyset$. Accordingly, $\mathcal{S}$-terms $f \in \mathsf{Tm}_{\mathcal{S}}(\Gamma ._{\mathcal{S}} \mathbf{Void}_{\mathcal{S}}, A)$ are (dependent) functions out of an empty set. As discussed in Section 2.5.1, there is exactly one such function for every A, and this is precisely the content of the isomorphism $\rho_{\mathcal{S}}$.

$$\rho_{\mathcal{S}}\ \Gamma\ A\ a := \star$$

Exercise 3.19. Complete the $\mathcal{S}$-interpretation of **Void**: verify that $\rho_{\mathcal{S}}$ is an isomorphism, and prove the naturality equations for $\mathbf{Void}_{\mathcal{S}}$ and $\rho_{\mathcal{S}}$, following Section 2.5.1.

Booleans Like **Void**, the booleans are also defined by a mapping-out property. Recalling Section 2.5.2, the specification of **Bool** has three components, the first two being a natural type-former and two natural term-formers:

$$\begin{aligned}&\mathbf{Bool}_{\mathcal{S}} : \textstyle\prod_{\Gamma \in \mathsf{Cx}_{\mathcal{S}}} \mathsf{Ty}_{\mathcal{S}}(\Gamma)\\ &\mathbf{true}_{\mathcal{S}}, \mathbf{false}_{\mathcal{S}} : \textstyle\prod_{\Gamma \in \mathsf{Cx}_{\mathcal{S}}} \mathsf{Tm}_{\mathcal{S}}(\Gamma, \mathbf{Bool}_{\mathcal{S}})\end{aligned}$$

The third component is once again a natural isomorphism, but unlike the previous examples in which the two directions of the isomorphism encode introduction and elimination, here the forward map is fixed by the choice of $\mathbf{true}_{\mathcal{S}}$ and $\mathbf{false}_{\mathcal{S}}$, and the reverse map expresses the principle that maps out of **Bool** are determined by their instantiations at **true** and **false**. Writing $\rho\ \Gamma\ A$ for the map which sends $a \in \mathsf{Tm}_{\mathcal{S}}(\Gamma ._{\mathcal{S}} \mathbf{Bool}_{\mathcal{S}}, A)$ to the pair of $\mathcal{S}$-terms $(a[\mathbf{id}_{\mathcal{S}} ._{\mathcal{S}} \mathbf{true}_{\mathcal{S}}]_{\mathcal{S}}, a[\mathbf{id}_{\mathcal{S}} ._{\mathcal{S}} \mathbf{false}_{\mathcal{S}}]_{\mathcal{S}})$, we require ρ to be an isomorphism.

$$\begin{aligned}&\rho : \textstyle\prod_{\Gamma \in \mathsf{Cx}_{\mathcal{S}}} \prod_{A \in \mathsf{Ty}_{\mathcal{S}}(\Gamma ._{\mathcal{S}} \mathbf{Bool}_{\mathcal{S}})} \mathsf{Tm}_{\mathcal{S}}(\Gamma ._{\mathcal{S}} \mathbf{Bool}_{\mathcal{S}}, A) \cong\\ &\qquad\qquad \mathsf{Tm}_{\mathcal{S}}(\Gamma, A[\mathbf{id}_{\mathcal{S}} ._{\mathcal{S}} \mathbf{true}_{\mathcal{S}}]_{\mathcal{S}}) \times \mathsf{Tm}_{\mathcal{S}}(\Gamma, A[\mathbf{id}_{\mathcal{S}} ._{\mathcal{S}} \mathbf{false}_{\mathcal{S}}]_{\mathcal{S}})\\ &\rho\ \Gamma\ A\ a := (a[\mathbf{id}_{\mathcal{S}} ._{\mathcal{S}} \mathbf{true}_{\mathcal{S}}]_{\mathcal{S}}, a[\mathbf{id}_{\mathcal{S}} ._{\mathcal{S}} \mathbf{false}_{\mathcal{S}}]_{\mathcal{S}})\end{aligned}$$

We can define $\mathbf{Bool}_{\mathcal{S}}$ to be any fixed two-element set $\{\mathbf{true}_{\mathcal{S}}, \mathbf{false}_{\mathcal{S}}\}$, regarded as a constant family over $\Gamma \in \mathcal{V}_\omega$ and $x \in \Gamma$.

$$\begin{aligned}\mathbf{Bool}_{\mathcal{S}}\ \Gamma\ x &\coloneqq \{0,1\}\\ \mathbf{true}_{\mathcal{S}}\ \Gamma\ x &\coloneqq 1\\ \mathbf{false}_{\mathcal{S}}\ \Gamma\ x &\coloneqq 0\end{aligned}$$

Exercise 3.20. State and prove the naturality equations for $\mathbf{Bool}_{\mathcal{S}}$, $\mathbf{true}_{\mathcal{S}}$, and $\mathbf{false}_{\mathcal{S}}$.

It remains only to check that $\rho\ \Gamma\ A$ is indeed an isomorphism.

Lemma 3.5.14. *The map* $a \mapsto (a[\mathbf{id}_{\mathcal{S}}.{}_{\mathcal{S}}\mathbf{true}_{\mathcal{S}}]_{\mathcal{S}}, a[\mathbf{id}_{\mathcal{S}}.{}_{\mathcal{S}}\mathbf{false}_{\mathcal{S}}]_{\mathcal{S}})$ *is an isomorphism*

$$\mathsf{Tm}_{\mathcal{S}}(\Gamma.{}_{\mathcal{S}}\mathbf{Bool}_{\mathcal{S}}, A) \cong \mathsf{Tm}_{\mathcal{S}}(\Gamma, A[\mathbf{id}_{\mathcal{S}}.{}_{\mathcal{S}}\mathbf{true}_{\mathcal{S}}]_{\mathcal{S}}) \times \mathsf{Tm}_{\mathcal{S}}(\Gamma, A[\mathbf{id}_{\mathcal{S}}.{}_{\mathcal{S}}\mathbf{false}_{\mathcal{S}}]_{\mathcal{S}})$$

Proof. Unfolding definitions, the $\mathcal{S}$-context $\Gamma.{}_{\mathcal{S}}\mathbf{Bool}_{\mathcal{S}}$ is the set $\Gamma \times \{0,1\}$, so $\mathcal{S}$-types $A \in \mathsf{Ty}_{\mathcal{S}}(\Gamma.{}_{\mathcal{S}}\mathbf{Bool}_{\mathcal{S}})$ are families of sets $\Gamma \times \{0,1\} \to \mathcal{V}_\omega$, and $\mathcal{S}$-terms $a \in \mathsf{Tm}_{\mathcal{S}}(\Gamma.{}_{\mathcal{S}}\mathbf{Bool}_{\mathcal{S}}, A)$ are dependent functions $\prod_{p \in \Gamma \times \{0,1\}} A(p)$. But

$$\begin{aligned}&\prod_{p \in \Gamma \times \{0,1\}} A(p)\\ \cong\ &\prod_{x \in \Gamma} \prod_{b \in \{0,1\}} A(x,b)\\ \cong\ &\prod_{x \in \Gamma} A(x,1) \times A(x,0)\\ \cong\ &\left(\prod_{x \in \Gamma} A(x,1)\right) \times \left(\prod_{x \in \Gamma} A(x,0)\right)\end{aligned}$$

where the forward composite map is $a \mapsto ((\lambda x \to a(x,1)), (\lambda x \to a(x,0)))$. Unfolding definitions, this is precisely the map we wanted to show is an isomorphism. □

Universes The final connective we discuss is $\mathbf{U}$, a "type of types" whose terms $\Gamma \vdash a : \mathbf{U}$ decode to types $\Gamma \vdash \mathbf{El}(a)$ type. As we saw in Section 2.6, universe types require far more rules than the other connectives: type theory has a countably infinite hierarchy of universes $\mathbf{U} = \mathbf{U}_0 : \mathbf{U}_1 : \mathbf{U}_2 : \ldots$, each closed under codes for every type-former and satisfying definitional equalities involving $\mathbf{El}$, with **lift** operations between these universes commuting with all the aforementioned operations. In addition, the $\mathcal{S}$-interpretation of $\mathbf{U}$ as a "set of sets" will force us to confront some set-theoretic technicalities.

The good news is that all of this structure will fall quite neatly into place. The astute reader may have noticed that Axiom 3.5.6 postulates an infinite hierarchy of Grothendieck universes $\mathcal{V}_0 \in \cdots \in \mathcal{V}_\omega$ of which we have only used $\mathcal{V}_\omega$ thus far; the remaining $\mathcal{V}_i$ serve as the $\mathcal{S}$-interpretations of the type-theoretic universe hierarchy.

Let us begin by defining $(\mathbf{U}_0)_{\mathcal{S}} = \mathbf{U}_{\mathcal{S}}$ and $(\mathbf{El}_0)_{\mathcal{S}} = \mathbf{El}_{\mathcal{S}}$:

$$\begin{aligned}
&\mathbf{U}_{\mathcal{S}} : \textstyle\prod_{\Gamma \in \mathcal{V}_\omega} \mathsf{Ty}_{\mathcal{S}}(\Gamma) \\
&\mathbf{U}_{\mathcal{S}}\ \Gamma\ x := \mathcal{V}_0 \\
&\mathbf{El}_{\mathcal{S}} : \textstyle\prod_{\Gamma \in \mathcal{V}_\omega} \mathsf{Tm}_{\mathcal{S}}(\Gamma, \mathbf{U}_{\mathcal{S}}) \to \mathsf{Ty}_{\mathcal{S}}(\Gamma) \\
&\mathbf{El}_{\mathcal{S}}\ \Gamma\ c := c
\end{aligned}$$

To make sense of the last definition, we note that $\mathbf{El}_{\mathcal{S}}\ \Gamma : (\Gamma \to \mathcal{V}_0) \to (\Gamma \to \mathcal{V}_\omega)$. By our hypothesis $\mathcal{V}_0 \in \mathcal{V}_\omega$ and Lemma 3.5.2, $\mathcal{V}_0 \subseteq \mathcal{V}_\omega$, so in particular $(\Gamma \to \mathcal{V}_0) \subseteq (\Gamma \to \mathcal{V}_\omega)$.

Exercise 3.21. State and prove the naturality equations for $\mathbf{U}_{\mathcal{S}}$ and $\mathbf{El}_{\mathcal{S}}$.

Following Section 2.6.2, the $\mathcal{S}$-interpretation of $\mathbf{U}$ must include codes for $\mathbf{\Pi}$-types:

$$\mathbf{pi}_{\mathcal{S}} : \textstyle\prod_{\Gamma \in \mathsf{Cx}_{\mathcal{S}}} (\sum_{A \in \mathsf{Tm}_{\mathcal{S}}(\Gamma, \mathbf{U}_{\mathcal{S}})} \mathsf{Tm}_{\mathcal{S}}(\Gamma.{}_{\mathcal{S}}\mathbf{El}_{\mathcal{S}}(A), \mathbf{U}_{\mathcal{S}})) \to \mathsf{Tm}_{\mathcal{S}}(\Gamma, \mathbf{U}_{\mathcal{S}})$$

satisfying a naturality equation as well as the following equation in $\Gamma \to \mathcal{V}_\omega$:

$$\mathbf{El}_{\mathcal{S}}\ \Gamma\ (\mathbf{pi}_{\mathcal{S}}\ \Gamma\ (A, B)) = \mathbf{\Pi}_{\mathcal{S}}\ \Gamma\ (\mathbf{El}_{\mathcal{S}}\ \Gamma\ A, \mathbf{El}_{\mathcal{S}}\ (\Gamma.{}_{\mathcal{S}}\mathbf{El}_{\mathcal{S}}(A))\ B)$$

Because $\mathbf{El}_{\mathcal{S}}\ \Gamma$ is just the inclusion $(\Gamma \to \mathcal{V}_0) \subseteq (\Gamma \to \mathcal{V}_\omega)$, we can simply take the above equation as a *definition*—setting $\mathbf{pi}_{\mathcal{S}}\ \Gamma\ (A, B) := \mathbf{\Pi}_{\mathcal{S}}\ \Gamma\ (A, B)$—as long as we prove that the right-hand side lands inside of $\Gamma \to \mathcal{V}_0$ when A and B are pointwise $\mathcal{V}_0$-small.

Lemma 3.5.15. *If $\Gamma \in \mathcal{V}_\omega$, $A \in \Gamma \to \mathcal{V}_0$, and $B \in (\sum_{x \in \Gamma} A(x)) \to \mathcal{V}_0$, then*

$$(\textstyle\prod_{x \in \Gamma} \prod_{a \in A(x)} B(x, a)) \in \Gamma \to \mathcal{V}_0$$

Proof. Note that this statement refines a similar observation in our construction of $\mathcal{S}$-$\mathbf{\Pi}$-types, in which all the $\mathcal{V}_0$ are replaced by $\mathcal{V}_\omega$. The proof is identical: because $\mathcal{V}_0$ is a Grothendieck universe, Lemma 3.5.2 implies that $\prod_{a \in A(x)} B(x, a) \in \mathcal{V}_0$ for all $x \in \Gamma$. □

The naturality equation for $\mathbf{pi}_{\mathcal{S}}$ then follows immediately from the naturality of $\mathbf{\Pi}_{\mathcal{S}}$. The codes for other connectives proceed identically, using the fact that $\mathcal{V}_0$ is closed under every relevant construction. For the remainder of the universe hierarchy, we define $(\mathbf{U}_i)_{\mathcal{S}}\ \Gamma\ x := \mathcal{V}_i$ and check that $(\mathbf{El}_i)_{\mathcal{S}}$ and $(\mathbf{lift}_i)_{\mathcal{S}}$ are subset inclusions.

3.5.4 Using the set model

We finally arrive at the main result of this section.

Theorem 3.5.16. *$\mathcal{S}$ is a model of extensional type theory.*

Although extensional type theory is often considered an *alternative* to set theory, the fact that $\mathcal{S}$ allows us to reduce questions about type theory to questions about sets makes the set model one of the most powerful tools for studying the properties of type theory. In Section 3.6, we appeal to $\mathcal{S}$ in two proofs that equality in extensional type theory is undecidable; in the remainder of this section, we will quickly rattle off several other corollaries of Theorem 3.5.16, starting with the consistency of type theory (Theorem 3.4.8).

Proof of Theorem 3.4.8. To show that type theory is consistent, by Theorem 3.4.7 it suffices to exhibit a model $\mathcal{M}$ in which $\mathsf{Tm}_{\mathcal{M}}(\mathbf{1}_{\mathcal{M}}, \mathbf{Void}_{\mathcal{M}})$ is empty. Choosing $\mathcal{M} = \mathcal{S}$, by Exercise 3.12 we have $\mathsf{Tm}_{\mathcal{S}}(\mathbf{1}_{\mathcal{S}}, \mathbf{Void}_{\mathcal{S}}) \cong \mathbf{Void}_{\mathcal{S}}\ \mathbf{1}_{\mathcal{S}}\ \star := \emptyset$. □

More generally, $\mathcal{S}$ tells us that any term in extensional type theory—that is, in its syntactic model $\mathcal{T}$ (Definition 3.4.4)—gives rise to a corresponding function of sets. On the one hand, this lets us construct functions on sets by writing down terms in type theory; on the other hand, we can *disprove* the existence of terms by showing that their image under the $\mathcal{S}$-interpretation does not exist, as we just did in the proof of consistency.

Lemma 3.5.17. *Within type theory, there are no injective functions* $(\mathbf{Nat} \to \mathbf{Nat}) \to \mathbf{Nat}$*; that is, there are no closed terms of type*

$$\textstyle\sum_{f:(\mathbf{Nat}\to\mathbf{Nat})\to\mathbf{Nat}} (g_1, g_2 : \mathbf{Nat} \to \mathbf{Nat}) \to f(g_1) = f(g_2) \to g_1 = g_2$$

Proof. Unfolding definitions, the image of such a term under $\mathcal{S}$ is a pair whose first projection is an ordinary set-theoretic function $f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$, and whose second projection is a three-argument function that takes two functions $g_1, g_2 : \mathbb{N} \to \mathbb{N}$ and $x \in \{\star \mid f(g_1) = f(g_2)\}$, and returns $\{\star \mid g_1 = g_2\}$. In particular, although the second projection is unique when it exists, it exists only when f is injective. But $\mathbb{N} \to \mathbb{N}$ is uncountable, so there can be no injective functions from it to $\mathbb{N}$. □

Remark 3.5.18. This argument does not go through if we restrict attention to the syntactic model, because the set $\mathsf{Tm}(\mathbf{1}, \mathbf{\Pi}(\mathbf{Nat}, \mathbf{Nat}))$ of closed terms of type $\mathbf{Nat} \to \mathbf{Nat}$ *is countable*: it is a quotient of a subset of finite derivation trees, which are countable. ◇

Theorem 3.5.19. *Extensional type theory does not have injective* **Π***-types (Definition 3.2.8).*

Proof. Using equality reflection and universes, the following judgment holds:

$$\mathbf{1}.\mathbf{Eq}(\mathbf{U}, \mathbf{pi}(\mathbf{unit}, \mathbf{void}), \mathbf{pi}(\mathbf{bool}, \mathbf{void})) \vdash \mathbf{\Pi}(\mathbf{Unit}, \mathbf{Void}) = \mathbf{\Pi}(\mathbf{Bool}, \mathbf{Void}) \text{ type}$$

If extensional type theory had injective **Π**-types, this would imply:

$$\mathbf{1}.\mathbf{Eq}(\mathbf{U}, \mathbf{pi}(\mathbf{unit}, \mathbf{void}), \mathbf{pi}(\mathbf{bool}, \mathbf{void})) \vdash \mathbf{Unit} = \mathbf{Bool} \text{ type}$$

This implies in particular that **true** and **false** are elements of **Unit** in this context. By the η rule for **Unit** this implies that **true** $=$ **false** in this context, and hence by Theorem 2.6.3,

$$\mathbf{1}.\mathbf{Eq}(\mathbf{U}, \mathbf{pi}(\mathbf{unit}, \mathbf{void}), \mathbf{pi}(\mathbf{bool}, \mathbf{void})) \vdash \mathbf{tt} : \mathbf{Void}$$

The $\mathcal{S}$-interpretation of the above context is a one-element set if $\mathbf{\Pi}_{\mathcal{S}}(\mathbf{Unit}_{\mathcal{S}}, \mathbf{Void}_{\mathcal{S}})$ and $\mathbf{\Pi}_{\mathcal{S}}(\mathbf{Bool}_{\mathcal{S}}, \mathbf{Void}_{\mathcal{S}})$ are equal sets, which is indeed the case because both are $\emptyset$. Thus the $\mathcal{S}$-interpretation of the above term must be a function from a one-element set to $\emptyset$, which does not exist. We conclude that there is no such term, and thus extensional type theory does not have injective **Π**-types. □

Finally, recall that all of the constructions in this section have assumed an $(\omega + 1)$-hierarchy of Grothendieck universes $\mathcal{V}_0 \in \cdots \in \mathcal{V}_\omega$ (Axiom 3.5.6): we use $\mathcal{V}_\omega$ to model contexts and types, and smaller $\mathcal{V}_i$ to model $\mathbf{U}_i$. In general, we need $n + 1$ Grothendieck universes to model a type theory with n universes.

Theorem 3.5.20. *An $(n + 1)$-hierarchy of Grothendieck universes $\mathcal{V}_0 \in \cdots \in \mathcal{V}_n$ suffices to construct a set-theoretic model of extensional type theory with n universes $\mathbf{U}_0 : \cdots : \mathbf{U}_{n-1}$.*

3.6 *Equality in extensional type theory is undecidable*

In this section we present two proofs that term equality in extensional type theory is undecidable, and hence by Exercise 3.3 that extensional type theory does not admit a normalization structure.

Theorem 3.6.1. *There is no algorithm which, given any pair of terms $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$ in extensional type theory, decides whether $\Gamma \vdash a = b : A$.*

The first proof, due to Castellan, Clairambault, and Dybjer [CCD17], is conceptually straightforward but requires an appeal to the set-theoretic model (Section 3.5). The second proof, due to Hofmann [Hof95a], requires only the assumption that extensional type theory is consistent (Theorem 3.4.8), but is more complex, requiring the machinery of recursively inseparable sets. Both of these ideas arise with some frequency in the metatheory of type theory, so we cover both proofs in some detail.

Remark 3.6.2. The failure of injectivity of **Π**-types in ETT (Theorem 3.5.19) already rules out bidirectional elaboration for ETT in the style of Section 3.2. The undecidability of equality further precludes the possibility of even a *non-bidirectional* algorithm (Section 3.1) that elaborates $\Gamma \vdash (\mathtt{refl}\ \tau\ e) : \mathbf{Eq}(A, a, b) \rightsquigarrow p$ if and only if $\Gamma \vdash \mathbf{refl} : \mathbf{Eq}(A, a, b)$, because in the presence of equality reflection, the latter judgment holds if and only if $\Gamma \vdash a = b : A$. ◇

3.6.1 *The first proof: deciding equality of SK terms*

The strategy of our first proof is to exhibit a context Γ_{SK} and an encoding $[\![-]\!]$ of terms of the SK combinator calculus into type-theoretic terms in context Γ_{SK}, such that two SK terms are convertible if and only if their encodings are judgmentally equal. Because convertibility of SK terms is undecidable, judgmental equality is as well.

Recall that the SK combinator calculus is an extremely minimal Turing-complete language generated by application and two combinators named S and K:

$$\textit{Combinators} \quad x ::= \quad S \mid K \mid x\ x$$

Combinators compute according to the following rewriting system $\mapsto$. We say that two combinators are *convertible*, written $x \sim y$, if they are related by the reflexive, symmetric, and transitive closure of $\mapsto$.

$$\frac{}{S\ x\ y\ z \mapsto (x\ z)\ (y\ z)} \qquad \frac{}{K\ x\ y \mapsto x} \qquad \frac{x \mapsto x'}{x\ y \mapsto x'\ y} \qquad \frac{y \mapsto y'}{x\ y \mapsto x\ y'}$$

We define the following context, written in Agda-style notation:

$$\begin{aligned}\Gamma_{\mathrm{SK}} \coloneqq \mathbf{1},&\\ &A : \mathbf{U},\\ &_\bullet_ : A \to A \to A,\\ &s : A,\\ &k : A,\\ &e_1 : (a\,b : A) \to \mathbf{Eq}(A, (k \bullet a) \bullet b, a),\\ &e_2 : (a\,b\,c : A) \to \mathbf{Eq}(A, ((s \bullet a) \bullet b) \bullet c, (a \bullet c) \bullet (b \bullet c))\end{aligned}$$

Writing Λ for the set of SK combinator terms, we can straightforwardly define a function $[\![-]\!] : \Lambda \to \mathsf{Tm}(\Gamma_{\mathrm{SK}}, A)$ by sending application, S, and K to $\bullet$, s, and k respectively, and this function respects convertibility of combinators.

Lemma 3.6.3. *There is a function* $[\![-]\!] : \Lambda \to \mathsf{Tm}(\Gamma_{\mathrm{SK}}, A)$ *with* $x \sim y \implies [\![x]\!] = [\![y]\!]$.

Exercise 3.22. The context Γ_{SK} only includes two of the four generating rules of $\mapsto$. Why haven't we included the other two, or reflexivity, symmetry, or transitivity?

Lemma 3.6.3 implies that term equality is sound for an undecidable problem, but not yet that term equality is undecidable; it is possible, for example, that *all* terms in the image of $[\![-]\!]$ are equal. To complete our proof, we must observe that term equality is also *complete* for convertibility; we argue this by using the set-theoretic model of type theory to recover the convertibility class of x from the term $[\![x]\!]$.

Theorem 3.6.4. *If* $[\![x]\!] = [\![y]\!]$ *then* $x \sim y$.

Proof. Let us write $f : \mathcal{T} \to \mathcal{S}$ for the homomorphism from the syntactic model $\mathcal{T}$ to the set-theoretic model $\mathcal{S}$. This homomorphism interprets syntactic contexts Γ as sets $\mathsf{Cx}_f(\Gamma)$, syntactic types $A \in \mathsf{Ty}(\Gamma)$ as $\mathsf{Cx}_f(\Gamma)$-indexed families of sets, and syntactic context extensions as indexed coproducts of those families. (See Section 3.5.)

Unwinding definitions, elements of $\mathsf{Cx}_f(\Gamma_{\mathrm{SK}})$ are "SK-algebras," or dependent tuples of a set along with application, S, and K operations satisfying the convertibility axioms. Combinators modulo convertibility form such an algebra in the evident way; writing $[x]$ for the convertibility equivalence class of $x \in \Lambda$, we have

$$\gamma_{\mathrm{SK}} \coloneqq (\Lambda/\!\sim, (\lambda[x]\ [y] \to [x\ y]), [s], [k], \star, \star) \in \mathsf{Cx}_f(\Gamma_{\mathrm{SK}})$$

Homomorphisms of models respect equality, so from $[\![x]\!] = [\![y]\!] \in \mathsf{Tm}(\Gamma_{\mathrm{SK}}, A)$ we see that these terms are interpreted in $\mathcal{S}$ as equal functions $\prod_{(A,\dots):\mathsf{Cx}_f(\Gamma_{\mathrm{SK}})} A$, and in particular, applying these functions to γ_{SK} produces two equal elements of $\Lambda/\!\sim$. We can prove by induction on combinators that for any $z \in \Lambda$ this procedure recovers z up to convertibility (*i.e.*, sends $[\![z]\!]$ to $[z]$) and thus $[x] = [y]$ as required. □

Theorem 3.6.5. *Equality of terms $a, b \in \mathsf{Tm}(\Gamma_{\mathrm{SK}}, A)$ is undecidable.*

Proof. Suppose it were decidable; then for any $x, y \in \Lambda$ we could decide the equality of $[\![x]\!], [\![y]\!] \in \mathsf{Tm}(\Gamma_{\mathrm{SK}}, A)$. By Lemma 3.6.3 and Theorem 3.6.4, $[\![x]\!] = [\![y]\!]$ if and only if $x \sim y$, so this lets us decide convertibility of SK-combinators, which is impossible. □

It immediately follows that term equality in ETT is not, in general, decidable.

3.6.2 *The second proof: separating classes of Turing machines*

In the first proof we reduce an undecidable problem to the judgmental equality of open terms, but establishing the completeness of this reduction requires appealing to the set-theoretic model of type theory. Our second proof relies only on the consistency of extensional type theory, showing that deciding judgmental equality of closed functions would allow us to algorithmically separate two recursively inseparable subsets of $\mathbb{N}$.

Notation 3.6.6. Fix a standard, effective Gödel encoding of Turing machines, in which the standard operations on Turing machines are definable by primitive recursion. We write ϕ_n for the partial function induced by the Turing machine encoded by n.

Theorem 3.6.7 (Rosser [Ros36], Trakhtenbrot [Tra53], and Kleene [Kle50])**.** *Consider the following two subsets of the natural numbers:*

$$A = \{n \in \mathbb{N} \mid \phi_n(n) \text{ terminates with result } 0\}$$
$$B = \{n \in \mathbb{N} \mid \phi_n(n) \text{ terminates with result } 1\}$$

There is no Turing machine which terminates on all inputs and separates A from B.

Proof. Suppose we are given a Turing machine e which always terminates with value 0 or 1, such that $e(n) = 0$ when $n \in A$ and $e(n) = 1$ when $n \in B$. Consider the algorithm

$$F(n) := \begin{cases} \mathtt{halt}(1) & e(n) = 0 \\ \mathtt{halt}(0) & e(n) = 1 \end{cases}$$

Because e terminates on all inputs, so does F. Note that $e(F(n)) \neq e(n)$ by construction: if $e(F(n)) = 1$ then $e(n) = 0$ and vice versa. By the second recursion theorem, there exists a Turing machine f realizing F applied to its own Gödel number. However, $e(f)$ can be neither 0 nor 1 as $e(f) = e(F(f))$ by definition, but $e(f) \neq e(F(f))$. □

We will show that the existence of a normalization structure for extensional type theory contradicts the above theorem. First, we observe that we can write a "small-step interpreter" for Turing machines in type theory. Let us write TM and State for **Nat** to indicate that we are interpreting a natural number as a Turing machine or Turing machine state respectively, as encoded by ϕ. Then we can define the following functions in type theory by primitive recursion:

- init : TM → **Nat** → State
- hasHalted : State → $\sum_{b:\mathbf{Bool}} \mathbf{if}(\mathbf{Nat}, \mathbf{Unit}, b)$
- step : State → State

Using these operations, we can run a Turing machine for an arbitrary but finite number of steps on any input, determine whether it has halted, and if so, extract the result. We can therefore define the following function:

```
-- returns true iff TM n halts on n with result 1 in fewer than t steps
returnOne : TM → Nat → Bool
returnOne n t = go (init n n) t
  where
  go : State → Nat → Bool
  go s zero = false
  go s (suc n) =
    if fst (hasHalted s) then isOne (snd (hasHalted s)) else go (step s) n
```

Let $H_0 \in \mathbb{N}$ be the encoding of a Turing machine which immediately halts with result 0 on all inputs. Then, writing $\bar{m}$ for the element of $\mathsf{Tm}(\mathbf{1}, \mathbf{Nat})$ corresponding to $m \in \mathbb{N}$, we will show that $\mathsf{returnOne}(\bar{n}), \mathsf{returnOne}(\bar{H_0}) \in \mathsf{Tm}(\mathbf{1}, \mathbf{\Pi}(\mathbf{Nat}, \mathbf{Bool}))$ are equal (resp., unequal) when n is a Turing machine which halts with result 0 (resp., 1).

Lemma 3.6.8. *If $n \in \mathbb{N}$ is such that $\phi_n(n) = 0$, then*

$$\mathbf{1} \vdash \mathsf{returnOne}\ \bar{n} = \mathsf{returnOne}\ \bar{H_0} : \mathbf{\Pi}(\mathbf{Nat}, \mathbf{Bool}).$$

Proof. By the η rule for $\mathbf{\Pi}$-types, it suffices to show

$$\mathbf{1}, t : \mathbf{Nat} \vdash \mathsf{returnOne}\ \bar{n}\ t = \mathsf{returnOne}\ \bar{H_0}\ t : \mathbf{Bool}$$

By equality reflection, this follows from:

$$\mathbf{1}, t : \mathbf{Nat} \vdash P_t : \mathbf{Eq}(\mathbf{Bool}, \mathsf{returnOne}\ \bar{n}\ t, \mathsf{returnOne}\ \bar{H_0}\ t)$$

In Exercise 3.23 the reader will establish this by **Nat** elimination on t. Note that by $\phi_n(n) = 0$, there exists some number ℓ such that the Turing machine encoded by n halts in t steps on n with result 0. Thus we must in essence construct terms

$$\begin{aligned}
&\mathbf{1}, t : \mathbf{Nat} \vdash P_0 : \mathbf{Eq}(\mathbf{Bool}, \mathsf{returnOne}\ \bar{n}\ \mathbf{zero}, \mathsf{returnOne}\ \bar{H_0}\ \mathbf{zero}) \\
&\mathbf{1}, t : \mathbf{Nat} \vdash P_1 : \mathbf{Eq}(\mathbf{Bool}, \mathsf{returnOne}\ \bar{n}\ (\mathbf{suc}\ \mathbf{zero}), \mathsf{returnOne}\ \bar{H_0}\ (\mathbf{suc}\ \mathbf{zero})) \\
&\vdots \\
&\mathbf{1}, t : \mathbf{Nat} \vdash P_{\ell+1} : \mathbf{Eq}(\mathbf{Bool}, \mathsf{returnOne}\ \bar{n}\ (\mathbf{suc}^{\ell+1}\ t), \mathsf{returnOne}\ \bar{H_0}\ (\mathbf{suc}^{\ell+1}\ t))
\end{aligned}$$

In the above, we write $\mathbf{suc}^{\ell+1}\ t$ for the $(\ell+1)$-fold application of $\mathbf{suc}(-)$ to t. When $i \leq \ell$ it is straightforward to construct P_i, as both sides equal **false**. For $P_{\ell+1}$, we note that $\mathsf{returnOne}\ m\ (\mathbf{suc}^k\ t) = \mathbf{false}$ when m encodes a machine which halts in fewer than k steps with a result other than 1, completing the proof. □

Exercise 3.23. Fill in the gap in the above argument using **Nat**-elimination.

The remaining condition is easier to show.

Lemma 3.6.9. *If $n \in \mathbb{N}$ is such that $\phi_n(n) = 1$, then if the equality*

$$\mathbf{1} \vdash \mathsf{returnOne}\ \bar{n} = \mathsf{returnOne}\ \bar{H_0} : \mathbf{\Pi}(\mathbf{Nat}, \mathbf{Bool})$$

holds, extensional type theory is inconsistent.

Proof. Because $\phi_n(n)$ terminates, there is some number of steps t with $\mathsf{returnOne}\ \bar{n}\ t = \mathbf{true}$. On the other hand, $\mathsf{returnOne}\ \bar{H_0}\ t = \mathbf{false}$ for every t, so by applying both of these equal functions to t we conclude that $\mathbf{1} \vdash \mathbf{true} = \mathbf{false} : \mathbf{Bool}$. By Theorem 2.6.3 this implies extensional type theory is inconsistent. □

Theorem 3.6.10. *Equality of terms $f, g \in \mathsf{Tm}(\mathbf{1}, \mathbf{\Pi}(\mathbf{Nat}, \mathbf{Bool}))$ is undecidable.*

Proof. If equality in this set were decidable, we could decide $\mathbf{1} \vdash \mathsf{returnOne}\ \bar{n} = \mathsf{returnOne}\ \bar{H_0} : \mathbf{\Pi}(\mathbf{Nat}, \mathbf{Bool})$ for all $n \in \mathbb{N}$. By Lemma 3.6.8, this equation holds for n if $\phi_n(n) = 0$; by Lemma 3.6.9 and Theorem 3.4.8, it does not hold if $\phi_n(n) = 1$. If this judgmental equality were decidable, we would be able to define a terminating algorithm which separates the subsets of $n \in \mathbb{N}$ for which $\phi_n(n) = 0$ and $\phi_n(n) = 1$, contradicting Theorem 3.6.7. □

Again it follows immediately that term equality in ETT is not in general decidable.

Remark 3.6.11. Equality reflection is far from the only natural type-theoretic rule that causes equality to become undecidable. A variation of our second proof establishes that the η-rule for **Nat** still results in undecidable equality even when **Eq**-types are removed from ETT altogether. Specifically, our uses of **Eq**-types in Lemma 3.6.8 can be replaced by judgmental equalities and Exercise 3.23 can be established by the η-rule for **Nat** instead of **Nat**-elimination into **Eq**-types. (See the sequence of exercises below.)

Similarly, the η-rule for **Void** results in undecidable equality even in the absence of **Eq**-types or **Nat**. The proof proceeds as follows: by the η-rule for **Void**, $\Gamma \vdash \mathbf{true} = \mathbf{false} : \mathbf{Bool}$ if and only there exists some $\Gamma \vdash b : \mathbf{Void}$. The inhabitation problem for **Void** can in turn be reduced to undecidable problems such as SK convertibility or the halting problem, using the set model for completeness.

Lest the reader think that all η-rules for inductive types result in undecidable equality, at the time of writing it is generally conjectured that intensional type theory (Chapter 4) extended with an η-rule for **Bool** has *decidable* equality [Alt+01]. ◇

Exercise 3.24. Given $\Gamma \vdash f, g : \mathbf{\Pi}(\mathbf{Nat}, \mathbf{Bool})$ with $\Gamma \vdash f(\mathbf{zero}) = g(\mathbf{zero}) : \mathbf{Bool}$ and $\Gamma, n : \mathbf{Nat} \vdash f(\mathbf{suc}\ n) = g(\mathbf{suc}\ n) : \mathbf{Bool}$, use the η-rule for **Nat** (and not **Eq**-types) to show that $\Gamma \vdash f = g : \mathbf{\Pi}(\mathbf{Nat}, \mathbf{Bool})$. (Hint: construct a displayed algebra.)

Exercise 3.25. Suppose $\Gamma \vdash f, g : \mathbf{\Pi}(\mathbf{Nat}, \mathbf{Bool})$ and

$$\begin{aligned}
&\Gamma \vdash f(\mathbf{zero}) = g(\mathbf{zero}) : \mathbf{Bool} \\
&\Gamma \vdash f(\mathbf{suc}\ \mathbf{zero}) = g(\mathbf{suc}\ \mathbf{zero}) : \mathbf{Bool} \\
&\vdots \\
&\Gamma \vdash f(\mathbf{suc}^\ell\ \mathbf{zero}) = g(\mathbf{suc}^\ell\ \mathbf{zero}) : \mathbf{Bool} \\
&\Gamma, t : \mathbf{Nat} \vdash f(\mathbf{suc}^{\ell+1}\ t) = g(\mathbf{suc}^{\ell+1}\ t) : \mathbf{Bool}
\end{aligned}$$

for some external natural number ℓ. Use Exercise 3.24 and external induction on ℓ to show that $\Gamma \vdash f = g : \mathbf{\Pi}(\mathbf{Nat}, \mathbf{Bool})$.

Exercise 3.26. Use Exercises 3.24 and 3.25 to reprove Lemma 3.6.8 without using **Eq**-types.

Further reading

There are a number of excellent pedagogical resources on type-checkers for dependent type theory that we encourage our implementation-inclined readers to explore. Coquand [Coq96] describes algorithms for bidirectional type-checking and deciding equality along with a proof sketch of correctness. Löh, McBride, and Swierstra [LMS10] include additional exposition and a complete Haskell implementation that extends a type-checker for a simply-typed calculus that is also described in the paper. The `Mini-TT` tutorial by Coquand et al. [Coq+09] includes a Haskell implementation of a type theory which is unsound (allowing arbitrary fixed-points) but supports data type declarations and basic pattern matching.

In addition to the aforementioned papers, there are numerous online resources, including a tutorial by Christiansen [Chr19] on the *normalization by evaluation* algorithm for deciding equality, and the `elaboration-zoo` of Kovács [Kov16] which is an excellent resource for more advanced implementation techniques.

This material will be published by Cambridge University Press as *Principles of Dependent Type Theory* by Carlo Angiuli and Daniel Gratzer. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © Carlo Angiuli and Daniel Gratzer 2024

4 *Intensional type theory*

In Chapter 3 we outlined several key properties of type theories: consistency states that type theory can be viewed as a logic, canonicity states that type theory can be viewed as a programming language, normalization allows us to define a type-checking algorithm, and invertibility of type constructors improves that algorithm. Unfortunately, we also saw in Section 3.6 that extensional type theory does not satisfy the latter two properties due to the *equality reflection* rule of its **Eq**-types (Section 2.4.4).

If we remove **Eq**-types from extensional type theory then it will satisfy all four metatheorems above, but it becomes unusably weak. A foreseeable consequence is that type theory would no longer have an equality proposition; a more subtle issue is that many equations stop holding altogether, judgmentally or otherwise. This is because inductive types are characterized by maps into other *types* only, so what properties they enjoy depends on what types exist. Indeed we have already seen that **Eq**-types allow us to prove their η-rules and universes allow us to prove disjointness of their constructors; without **Eq**-types their η-rules will no longer be provable, and disjointness cannot even be stated!

We are left asking: *how should we internalize judgmental equality as a type, if not* **Eq***?* This question has preoccupied type theorists for decades and—fortunately for their continued employment—has no clear-cut answer. We will find that deleting equality reflection causes equality types to become underconstrained, and their most canonical replacement, *intensional identity types*, lack several important reasoning principles. The decades-long quest for a suitable identity type has resulted in many subtle variations as well as some major innovations in type theory, as we will explore in Chapter 5. But first we turn our attention to *intensional type theory*, or type theory with intensional identity types, the system on which most type-theoretic proof assistants are based. (Its rules are collected in Appendix A, ignoring the rules marked with (ETT) which are present only in extensional type theory.)

Notation 4.0.1. We adopt the common acronyms ETT and ITT for extensional type theory and intensional type theory respectively.

In this chapter In Section 4.1 we explore the basic properties that any propositional equality connective must satisfy, and show that a small set of primitive operations suffices to recover many of the positive consequences of equality reflection while allowing for normalization. In Section 4.2 we formally define the intensional identity

type according to the framework of inductive types outlined in Section 2.5, and show that this type precisely satisfies the properties of equality outlined above. In Section 4.3 we compare extensional and intensional identity types, noting that the latter lacks several important principles, but by adding two axioms to it we can recover all the reasoning principles of extensional type theory in a precise sense. Finally, in Section 4.4, we summarize a line of research on *observational type theory* [AMS07], which attempts to improve intensional identity types without sacrificing normalization.

Goals of the chapter By the end of this chapter, you will be able to:

- Define subst and contractibility of singletons, use them to prove other properties of equality, and implement them using intensional identity types.
- Explain how intensional identity types fit into the framework of internalizing judgmental structure that we developed in Chapter 2.
- Discuss the relationship and tradeoffs between intensional and extensional equality.
- Informally describe observational type theory, and explain how it addresses the shortcomings of intensional and extensional type theory.

4.1 *Programming with propositional equality*

In this section we will informally consider what properties should be satisfied by any "type of equations." Recall from Section 1.3 that such a *propositional* (or typal, or internal) notion of equality is important for proving equations between types that type-checkers cannot handle automatically, and that such type equations allow us to cast (coerce) between the types involved. In Section 3.1 we discussed how type-checkers automatically handle definitional (judgmental) type equalities; one can therefore think of propositional type equalities as "verified casts" that users manually insert into terms.

Our starting point will be the type theory described in Chapter 2 but without **Eq**-types. Instead we will add an *identity type* **Id**[1] with the same formation (and universe introduction) rule but no other properties yet:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash \mathbf{Id}(A, a, b)\ \mathrm{type}} \qquad \frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma \vdash x : \mathbf{El}(a) \qquad \Gamma \vdash y : \mathbf{El}(a)}{\begin{array}{c}\Gamma \vdash \mathbf{id}(a, x, y) : \mathbf{U}_i \\ \Gamma \vdash \mathbf{El}(\mathbf{id}(a, x, y)) = \mathbf{Id}(\mathbf{El}(a), x, y)\ \mathrm{type}\end{array}}$$

[1] Although beyond the scope of this book, we expect the **Superego** connective to internalize the rules of type theory; arguably singleton types internalize the self and thus serve as the **Ego**.

The primary way to use a proof of $\mathbf{Id}(A, a, a')$ is in concert with an A-indexed family of types $b : A \to \mathbf{U}$; namely, we conclude that the a and a' instances of this family are themselves equal in the sense that we have a proof of $\mathbf{Id}(\mathbf{U}, b\ a, b\ a')$, and as a result we are able to cast between the types $\mathbf{El}(b\ a)$ and $\mathbf{El}(b\ a')$. Notably, because type equality is central to this story, universes will play a major role in this section.

Notation 4.1.1. What should we call terms of type $\mathbf{Id}(A, a, b)$? This type will no longer precisely internalize the equality judgment so it can be misleading to call them *equalities* between a and b. On the other hand, calling them "*proofs of equality* between a and b" is too cumbersome. We will refer to them as *identifications* between a and b.

Notation 4.1.2. In the remainder of this section we will return to the informal notation of Chapter 1; in particular, we omit $\mathbf{El}(-)$, thereby suppressing the difference between types and terms of type $\mathbf{U}$. We resume our more rigorous notation in Section 4.2.

4.1.1 Constructing identifications

Following the discussion above, we can already formulate two necessary conditions on $\mathbf{Id}(A, a, b)$. First, we must have some source of identifications between terms. As with **Eq**-types we choose reflexivity; in concert with definitional equality, this allows us to prove any terms are identified as long as they differ only by β, η, and expanding definitions:

$$\mathsf{refl} : \{A : \mathbf{U}\} \to (a : A) \to \mathbf{Id}(A, a, a)$$

Secondly, given an identification $\mathbf{Id}(A, a, a')$ and a dependent type $B : A \to \mathbf{U}$, we must be able to convert terms of type $B(a)$ to $B(a')$, a process (confusingly) known as *substitution*:

$$\mathsf{subst} : \{A : \mathbf{U}\}\ \{a\ a' : A\} \to (B : A \to \mathbf{U}) \to \mathbf{Id}(A, a, a') \to B\ a \to B\ a'$$

Remark 4.1.3. The subst function did not emerge in our discussion of **Eq**-types for the simple reason that equality reflection trivializes it: $\mathsf{subst}\ B\ p\ b = b$. Indeed, all of the operations we discuss in this section are trivial in the presence of equality reflection. ◇

By assuming that **Id**-types satisfy refl and subst we are off to a good start, but *a priori* these are only two of the many combinators that we expect to be definable for $\mathbf{Id}(A, a, b)$; for starters, as an equality relation, identifications ought to be not only reflexive but also symmetric and transitive. Fortunately and somewhat surprisingly, it turns out that both symmetry and transitivity are consequences of refl and subst.

Lemma 4.1.4. *Using* refl *and* subst*, we can prove symmetry of identifications, i.e.,*

$$\mathsf{sym} : \{A : \mathbf{U}\}\ \{a\ b : A\} \to \mathbf{Id}(A, a, b) \to \mathbf{Id}(A, b, a)$$

Proof. Fix $A : \mathbf{U}$ and $a\ b : A$ and $p : \mathbf{Id}(A, a, b)$. To construct a term of type $\mathbf{Id}(A, b, a)$, we simply choose a clever B at which to instantiate subst:

$$B : A \to \mathbf{U}$$
$$B\ x = \mathbf{id}(A, x, a)$$

In particular, note that $B(a) = \mathbf{Id}(A, a, a)$ is easily proven by refl, and $B(b) = \mathbf{Id}(A, b, a)$ is our goal; thus subst $B\ p$ is a function $B(a) \to B(b)$ and our goal follows soon after:

$$\mathsf{sym} : \{A : \mathbf{U}\}\ \{a\ b : A\} \to \mathbf{Id}(A, a, b) \to \mathbf{Id}(A, b, a)$$
$$\mathsf{sym}\ \{A\ a\ b\}\ p = \mathsf{subst}\ (\lambda x \to \mathbf{id}(A, x, a))\ p\ (\mathsf{refl}\ a)$$

□

Lemma 4.1.5. *Using* refl *and* subst*, we can prove transitivity of identifications, i.e.,*

$$\mathsf{trans} : \{A : \mathbf{U}\}\ \{a\ b\ c : A\} \to \mathbf{Id}(A, a, b) \to \mathbf{Id}(A, b, c) \to \mathbf{Id}(A, a, c)$$

Proof. Fix $A : \mathbf{U}$, $a\ b\ c : A$, $p : \mathbf{Id}(A, a, b)$, and $q : \mathbf{Id}(A, b, c)$. To construct a term of type $\mathbf{Id}(A, a, c)$, we again choose a clever instantiation of subst, in this case $B(x) = \mathbf{Id}(A, a, x)$. Once again, $B(b)$ is easily proven by our assumption p, and $B(c)$ is our goal. Substituting along $q : \mathbf{Id}(A, b, c)$ completes our proof:

$$\mathsf{trans} : \{A : \mathbf{U}\}\ \{a\ b\ c : A\} \to \mathbf{Id}(A, a, b) \to \mathbf{Id}(A, b, c) \to \mathbf{Id}(A, a, c)$$
$$\mathsf{trans}\ \{A\ a\ b\ c\}\ p\ q = \mathsf{subst}\ (\lambda x \to \mathbf{id}(A, a, x))\ q\ p$$

□

Exercise 4.1. Provide an alternative proof trans′ of Lemma 4.1.5 which substitutes along p rather than q, using a slightly different choice of B.

In fact, refl and subst also allow us to prove that identifications are a congruence, in the sense that given $\mathbf{Id}(A, a, a')$ and $f : A \to B$, we obtain an identification $\mathbf{Id}(B, f\ a, f\ a')$.

Lemma 4.1.6. *Using* subst*, we can prove congruence of identifications, i.e.,*

$$\mathsf{cong} : \{A\ B : \mathbf{U}\}\ \{a\ a' : A\} \to (f : A \to B) \to \mathbf{Id}(A, a, a') \to \mathbf{Id}(B, f\ a, f\ a')$$

Proof. The proof strategy remains the same, so we proceed directly to the term:

$$\mathsf{cong} : \{A\ B : \mathbf{U}\}\ \{a\ a' : A\} \to (f : A \to B) \to \mathbf{Id}(A, a, a') \to \mathbf{Id}(B, f\ a, f\ a')$$
$$\mathsf{cong}\ \{A\ B\ a\ a'\}\ f\ p = \mathsf{subst}\ (\lambda x \to \mathbf{id}(B, f\ a, f\ x))\ p\ (\mathsf{refl}\ (f\ a))$$

□

Finally, we must consider how subst ought to compute. Because subst can produce terms of any type, including **Bool** and **Nat**, we must impose some definitional equalities on it if our type theory is to satisfy canonicity (Section 3.4). One equation springs to mind immediately: if we apply subst B to refl a, the resulting coercion $B\ a \to B\ a$ has the type of the identity function, so it is reasonable to ask for it to *be* the identity function. That is, we ask for the following definitional equality:

$$\mathsf{subst}\ B\ (\mathsf{refl}\ a)\ b \equiv b : B\ a$$

4.1.2 *Constructing identifications of identifications*

Although refl and subst go quite a long way, they *do not* suffice to derive all the properties of identifications we might expect; we start encountering their limits as soon as we consider identifications between elements of $\mathbf{Id}(A, a, b)$ itself. These *identifications of identifications* arise very naturally in practice. Quite often we must use subst when constructing a dependently-typed term in order to align various type indices; if we ever construct a type that depends on such a term, we will very quickly be in the business of proving that two potentially distinct sequences of subst casts are themselves equal.

For the sake of concreteness, consider the following pair of operations that "rotate" a **Vec**tor (a list of specified length, as defined in Chapter 1):

```
append : {A : U} {n m : Nat} → Vec A n → Vec A m → Vec A (n + m)
comm : (n m : Nat) → Id(Nat, n + m, m + n)

rot1 : {A : U} {n : Nat} → Vec A n → Vec A n
rot1 [] = []
rot1 {A} {suc n} (x :: xs) = subst (Vec A) (comm n 1) (append xs (x :: []))

rot2 : {A : U} {n : Nat} → Vec A n → Vec A n
rot2 [] = []
rot2 (x :: []) = x :: []
rot2 {A} {suc (suc n)} (x₀ :: x₁ :: xs) =
  subst (Vec A) (comm n 2) (append xs (x₀ :: x₁ :: []))
```

We expect to be able to prove that rot1 twice is the same as rot2:

```
{A : U} {n : Nat} → (xs : Vec A (2 + n)) →
  Id(Vec A (2 + n), rot1 (rot1 xs), rot2 xs)
```

However, this will not be possible with our current set of primitives. In our definitions of rot1 and rot2 we were forced to include various applications of subst to correct

mismatches between the indices $(n+1)$, $(1+n)$ and $(n+2)$, $(2+n)$, and these subst terms will get in our way as we try to establish the above identification. Specifically, we will get stuck attempting to construct an identification between

$$\mathsf{rot1}\ (\mathsf{subst}\ (\mathbf{Vec}\ A)\ (\mathsf{comm}\ (\mathsf{suc}\ n)\ 1)\ (\mathsf{append}\ (x_1 :: xs)\ (x_0 :: [])))$$

and

$$\mathsf{subst}\ (\mathbf{Vec}\ A)\ (\mathsf{comm}\ n\ 2)\ (\mathsf{append}\ xs\ (x_0 :: x_1 :: []))$$

of type $\mathsf{Vec}\ A\ (2+n)$. Unfortunately, because n is a variable, neither $\mathsf{comm}\ (\mathsf{suc}\ n)\ 1$ nor $\mathsf{comm}\ n\ 2$ are the reflexive identification, so we can make no further progress.

The above example is a bit involved, but there are many smaller (albeit more contrived) examples of identifications that are beyond our reach; for example, given a variable $p : \mathbf{Id}(A, a, b)$ we cannot construct an identification $\mathbf{Id}(\mathbf{Id}(A, a, b), p, \mathsf{sym}\ (\mathsf{sym}\ p))$.

Our "API" for identity types is thus missing an operation that allows us to prove identifications between two identifications. To hit upon this operation, we introduce the concept of *(propositional) singleton types* (in contrast to the "definitional singleton types" of Section 3.3). Given a type A and a term $a : A$, the singleton type $[a]$ is defined as follows:

$$[a] = \textstyle\sum_{b:A} \mathbf{Id}(A, a, b)$$

That is, $[a]$ is the type of "elements of A that can be identified with a." Intuitively, there should only be one such element, namely a itself—or to be more precise, $(a, \mathsf{refl}\ a)$. But this, too, is not yet provable. Certainly, given an arbitrary element $(b, p) : [a]$ we can see that (by p) their first projections a and b are identified, but we have no way of identifying their second projections $\mathsf{refl}\ a$ and p.

In fact, most of our "coherence problems" of identifying identifications can be reduced to the problem of identifying all elements of $[a]$: this is in some sense the *ur*-coherence problem. Intuitively this is because being able to identify arbitrary (b, p) with $(a, \mathsf{refl}\ a)$ allows us to transform subst terms involving the arbitrary identification p into subst terms involving the distinguished identification $\mathsf{refl}\ a$, the latter of which "compute away."

Lemma 4.1.7. *Suppose we are given some $A : \mathbf{U}$ and $a : A$ such that all elements of $[a]$ are identified; then for any $b : A$ and $p : \mathbf{Id}(A, a, b)$ we have $\mathbf{Id}(\mathbf{Id}(A, a, b), p, \mathsf{sym}\ (\mathsf{sym}\ p))$.*

Proof. Fixing A, a, b, and p, we notice that $(a, \mathsf{refl}\ a), (b, p) : [a]$ by definition, and thus by assumption we have an identification $q : \mathbf{Id}([a], (a, \mathsf{refl}\ a), (b, p))$. As before, we shall choose a clever B for which subst B solves our problem, namely:

$$B : [a] \to \mathbf{U}$$

$$B\ (b_0, p_0) = \mathbf{id}(\mathbf{id}(A, a, b_0), p_0, \mathsf{sym}\ (\mathsf{sym}\ p_0))$$

Inspecting our definition of Lemma 4.1.4, we see that $\mathsf{sym}\ (\mathsf{refl}\ x) = \mathsf{refl}\ x$ definitionally, and thus the following definitional equalities hold:

$$\begin{aligned} B\ (a, \mathsf{refl}\ a) &= \mathbf{Id}(\mathbf{Id}(A, a, a), \mathsf{refl}\ a, \mathsf{sym}\ (\mathsf{sym}\ (\mathsf{refl}\ a))) \\ &= \mathbf{Id}(\mathbf{Id}(A, a, a), \mathsf{refl}\ a, \mathsf{sym}\ (\mathsf{refl}\ a)) \\ &= \mathbf{Id}(\mathbf{Id}(A, a, a), \mathsf{refl}\ a, \mathsf{refl}\ a) \\ B\ (b, p) &= \mathbf{Id}(\mathbf{Id}(A, a, b), p, \mathsf{sym}\ (\mathsf{sym}\ p)) \end{aligned}$$

It is easy to produce an element of the former type (namely, $\mathsf{refl}\ (\mathsf{refl}\ a)$), the latter type is our goal, and q is an identification between the two indices. Thus:

$$\begin{aligned} &\mathsf{symsym} : \{A : \mathbf{U}\}\ \{a\ b : A\} \to (p : \mathbf{Id}(A, a, b)) \to \mathbf{Id}(\mathbf{Id}(A, a, b), p, \mathsf{sym}\ (\mathsf{sym}\ p)) \\ &\mathsf{symsym}\ \{A\ a\ b\}\ p = \mathsf{subst} \\ &\quad (\lambda(b_0, p_0) \to \mathbf{id}(\mathbf{id}(A, a, b_0), p_0, \mathsf{sym}\ (\mathsf{sym}\ p_0))) \\ &\quad ? : \mathbf{Id}([a], (a, \mathsf{refl}\ a), (b, p)) \quad \text{-- by assumption} \\ &\quad (\mathsf{refl}\ (\mathsf{refl}\ a)) \end{aligned}$$

□

We substantiate the assumption of Lemma 4.1.7 with a new primitive operation on identity types, uniq, that identifies $(a, \mathsf{refl}\ a)$ with arbitrary elements of $[a]$. (By sym and trans, it follows that any two arbitrary elements of $[a]$ are also identified.) As with subst, we also assert that a certain definitional equality holds when uniq is supplied with the reflexive identification. This operation is often called *singleton contractibility* [Coq14; UF13], and it will feature prominently in Chapter 5.

$$\begin{aligned} \mathsf{uniq} : \{A : \mathbf{U}\}\ \{a : A\} \to (x : [a]) \to \mathbf{Id}([a], (a, \mathsf{refl}\ a), x) \\ \mathsf{uniq}\ (a, \mathsf{refl}\ a) = \mathsf{refl}\ (a, \mathsf{refl}\ a) \end{aligned}$$

Exercise 4.2. Like subst, uniq is definable in extensional type theory; show this.

Exercise 4.3. Recalling trans (Lemma 4.1.5) and trans′ (Exercise 4.1), use subst and uniq to construct a term of the following type:

$$\begin{aligned} &\{A : \mathbf{U}\}\ \{a\ b\ c : A\} \to (p : \mathbf{Id}(A, a, b)) \to (q : \mathbf{Id}(A, b, c)) \to \\ &\quad \mathbf{Id}(\mathbf{Id}(A, a, c), \mathsf{trans}\ p\ q, \mathsf{trans}'\ p\ q) \end{aligned}$$

4.1.3 *Intensional identity types*

To summarize Sections 4.1.1 and 4.1.2, we have asked for $\mathbf{Id}(A, a, b)$ to support the following three operations subject to two definitional equalities:

$$\begin{aligned}
&\mathsf{refl} : \{A : \mathbf{U}\} \to (a : A) \to \mathbf{Id}(A, a, a)\\
&\mathsf{subst} : \{A : \mathbf{U}\}\ \{a\ a' : A\} \to (B : A \to \mathbf{U}) \to \mathbf{Id}(A, a, a') \to B\ a \to B\ a'\\
&\mathsf{uniq} : \{A : \mathbf{U}\}\ \{a : A\} \to (x : [a]) \to \mathbf{Id}([a], (a, \mathsf{refl}\ a), x)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{subst}\ B\ (\mathsf{refl}\ a)\ b = b\\
&\mathsf{uniq}\ (a, \mathsf{refl}\ a) = \mathsf{refl}\ (a, \mathsf{refl}\ a)
\end{aligned}$$

Definition 4.1.8. An *intensional identity type* is any type $\mathbf{Id}(A, a, b)$ equipped with the three operations above satisfying the two definitional equalities above.

Intensional identity types were introduced by Martin-Löf [Mar75] and have been the "standard" formulation of propositional equality in type theory for most of the intervening years, although various authors have presented them via different but equivalent sets of primitive operations and equations [CP90; PP90; Pau93; Str93; Coq14].[2] Our presentation most closely follows Coquand [Coq14] which, to our knowledge, was first proposed by Steve Awodey in 2009. In Sections 4.3 and 4.4 we will also consider related but *non-equivalent* presentations endowing $\mathbf{Id}(A, a, b)$ with more properties [Str93; Hof95a; AMS07].

Let us be clear, however, that this broad agreement in the literature is not an indication of happiness. On the contrary, most type theorists have many complaints about intensional identity types: there are several important properties that they do *not* satisfy, and they can be frustrating in practice for a number of reasons. They have persisted for so long because of a relative lack of compelling alternatives that also satisfy the two crucial properties of:

1. Capturing the most important properties of equality—reflexivity, symmetry, transitivity, congruence, substitutivity, etc.—thus enabling a wide range of constructions.

2. Their inclusion in a type theory is compatible with all the metatheorems discussed in Chapter 3, especially—unlike **Eq**-types—normalization.

In Section 4.3 we will discuss the shortcomings of **Id**-types in more detail, but it will turn out that these shortcomings can be mostly overcome by adding several axioms (postulated terms, or in essence, free variables) to type theory. Adding such axioms causes canonicity to fail, but as discussed in Section 3.4, type theories without canonicity are merely frustrating (requiring more manual reasoning by identifications), whereas type theories without normalization are essentially un-type-checkable. As

[2] The equivalence between the presentations of Martin-Löf [Mar75] and Paulin-Mohring [Pau93] is due to Hofmann [Str93, Addendum].

a result, many users of type theory opt to work with **Id**-types with some additional axioms.

But before we get ahead of ourselves, we proceed by formally defining **Id**-types and thus the type theory known as *intensional type theory*.

4.2 Intensional identity types

In this section we formally define intensional identity types, or **Id**-types, returning to the style of definition adopted throughout Chapter 2. Although it is possible to add **Id**-types to extensional type theory, we are primarily interested in defining *intensional type theory*, which is obtained by replacing certain rules of ETT by the rules in this section. Specifically, we remove from the theory of Chapter 2 all rules pertaining to **Eq**-types; in Appendix A those rules are annotated (ETT), and the rules added in this section are annotated (ITT).

Although the rules for **Id**-types appear complicated and unmotivated at first, it will turn out that they arise naturally from our methodology that types internalize judgmental structure. Recalling Slogan 2.5.5, connectives in type theory are specified by a natural type-forming operation whose terms are either defined by a mapping-in property (a natural isomorphism with judgmentally-defined structure) or a mapping-out property (an algebra signature for which the type carries a weakly initial algebra).

The formation rule of $\mathbf{Id}(A, a, b)$ is identical to that of $\mathbf{Eq}(A, a, b)$:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash \mathbf{Id}(A, a, b)\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Delta \vdash \mathbf{Id}(A, a, b)[\gamma] = \mathbf{Id}(A[\gamma], a[\gamma], b[\gamma])\ \mathsf{type}}$$

Or equivalently, the following type-forming operation natural in Γ:

$$\mathbf{Id}_\Gamma : (\textstyle\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Tm}(\Gamma, A) \times \mathsf{Tm}(\Gamma, A)) \to \mathsf{Ty}(\Gamma)$$

We must now decide whether to define $\mathbf{Id}(A, a, b)$ by a mapping-in property or a mapping-out property. In Chapter 2 we saw that mapping-in properties are generally both simpler and better-behaved, but we already defined **Eq**-types by the mapping-in property of internalizing judgmental equality (i.e., $\mathsf{Tm}(\Gamma, \mathbf{Eq}(A, a, b)) \cong \{\star \mid a = b\}$), and it is unclear what other structure we could ask for **Id**-types to internalize.[3]

Faced with no other options, we are forced to consider a mapping-out property instead. Per the discussion in Sections 2.5.2 to 2.5.4, such a property starts with a

[3] Cubical type theory in fact invents a new judgmental structure for propositional equality to internalize, but we will return to this point in Section 5.3.

collection of natural term constructors of $\mathbf{Id}(A, a, b)$, in this case only **refl**exivity:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{refl} : \mathbf{Id}(A, a, a)} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A}{\Delta \vdash \mathbf{refl}[\gamma] = \mathbf{refl} : \mathbf{Id}(A[\gamma], a[\gamma], a[\gamma])}$$

Or equivalently, the following term-forming operation natural in Γ:

$$\mathbf{refl}_{\Gamma,A,a} \in \mathsf{Tm}(\Gamma, \mathbf{Id}(A, a, a))$$

Whereas the mapping-in property of **Eq**-types asserts that **refl** is their only inhabitant, the mapping-out property of **Id**-types will assert that every type *believes* that **refl** is their only inhabitant, in just the same way that every type "believes" that **true** and **false** are the only elements of **Bool**, namely that to map out of **Bool** it suffices to explain how to behave on **true** and **false**.

Remark 4.2.1. Like the induction principles of inductive types, the subst and uniq primitives of Section 4.1 are both maps out of $\mathbf{Id}(A, a, b)$ that have prescribed behavior on the constructor **refl**. We will see shortly that both subst and uniq are definable via the **Id**-elimination principle we are about to present, and remarkably, that **Id**-elimination can conversely be recovered as a combination of subst and uniq!

Compared to subst and uniq, **Id**-elimination is more clearly motivated by general considerations (mapping-out properties), more self-contained (not requiring Σ-types), and even often more ergonomic in practice. But subst and uniq are nevertheless very important combinators that certainly merit special discussion. ◇

Luckily **refl** is not a recursive constructor, so we can avoid the displayed algebras of Section 2.5.4 and return to the simpler characterization of mapping-out properties in Sections 2.5.3 and 2.5.5 as a section (right inverse) to substitution of constructors.

Suppose we have a dependent type over an identity type:

$$\Gamma.A.A[\mathbf{p}].\mathbf{Id}(A[\mathbf{p}^2], \mathbf{q}[\mathbf{p}], \mathbf{q}) \vdash C\,\mathsf{type}$$

Into any term of the above type we can substitute **refl**:

$$(\mathbf{id}.\mathbf{q}.\mathbf{refl})^* : \mathsf{Tm}(\Gamma.A.A[\mathbf{p}].\mathbf{Id}(A[\mathbf{p}^2], \mathbf{q}[\mathbf{p}], \mathbf{q}), C) \to \mathsf{Tm}(\Gamma.A, C[\mathbf{id}.\mathbf{q}.\mathbf{refl}])$$

The elimination principle for **Id**-types is precisely a section of the above map.

Let us unpack this a bit. First, we rewrite the above map using named variables:

$$[a/b, \mathbf{refl}/p] :$$
$$\mathsf{Tm}(\Gamma, a : A, b : A, p : \mathbf{Id}(A, a, b), C(a, b, p)) \to \mathsf{Tm}(\Gamma, a : A, C(a, a, \mathbf{refl}(a)))$$

A section to this map tells us that to construct an element of $C(a,b,p)$ for any $a,b:A$ and $p:\mathbf{Id}(A,a,b)$, it suffices to say what to do on $a,a,\mathbf{refl}$ (i.e., provide a term of type $C(a,a,\mathbf{refl})$). Compared to our definition of **if** in Section 2.5.2, the context on the left is more complex because the domain of a dependent type $C:\mathbf{Id}(A,a,b)\to\mathbf{U}$ is itself dependent on $a,b:A$, and the context on the right is more complex because the constructor **refl** is dependent on $a:A$.

Remark 4.2.2. From a more nuts-and-bolts perspective, imagine that we asked for C not to depend on all three of a,b,p as $\Gamma, a:A, b:A, p:\mathbf{Id}(A,a,b)\vdash C(a,b,p)$ type, but only on p, i.e., $\Gamma, p:\mathbf{Id}(A,a,b)\vdash C(p)$ type for some fixed $a,b:A$. Then we would not be able to even *state* what it means to substitute **refl** for p, because **refl** only has type $\mathbf{Id}(A,a,b)$ when a and b are definitionally equal. Instead, we ask for all of a,b,p to be variables, and consider the substitution of $a,a,\mathbf{refl}$ for a,b,p. ◇

Unfolding the above section into inference rules, we once again "build in a cut" by applying the stipulated term in context $\Gamma.A.A[\mathbf{p}].\mathbf{Id}(A[\mathbf{p}^2],\mathbf{q}[\mathbf{p}],\mathbf{q})$ to arguments $a:A$, $b:A$, and $p:\mathbf{Id}(A,a,b)$ all in context Γ. The first rule below is the section map itself, the second rule is naturality of the section map, and the third states that applying the section map followed by $(\mathbf{id}.\mathbf{q}.\mathbf{refl})^*$ is the identity:

$$\frac{\Gamma\vdash a:A \qquad \Gamma\vdash b:A \qquad \Gamma\vdash p:\mathbf{Id}(A,a,b) \qquad \Gamma.A.A[\mathbf{p}].\mathbf{Id}(A[\mathbf{p}^2],\mathbf{q}[\mathbf{p}],\mathbf{q})\vdash C\,\mathsf{type} \qquad \Gamma.A\vdash c:C[\mathbf{id}.\mathbf{q}.\mathbf{refl}]}{\Gamma\vdash \mathbf{j}(c,p):C[\mathbf{id}.a.b.p]}$$

$$\frac{\Delta\vdash\gamma:\Gamma \qquad \Gamma\vdash a:A \qquad \Gamma\vdash b:A \qquad \Gamma\vdash p:\mathbf{Id}(A,a,b) \qquad \Gamma.A.A[\mathbf{p}].\mathbf{Id}(A[\mathbf{p}^2],\mathbf{q}[\mathbf{p}],\mathbf{q})\vdash C\,\mathsf{type} \qquad \Gamma.A\vdash c:C[\mathbf{id}.\mathbf{q}.\mathbf{refl}]}{\Delta\vdash \mathbf{j}(c,p)[\gamma]=\mathbf{j}(c[(\gamma\circ\mathbf{p}).\mathbf{q}],p[\gamma]):C[\gamma.a[\gamma].b[\gamma].p[\gamma]]}$$

$$\frac{\Gamma\vdash a:A \qquad \Gamma.A.A[\mathbf{p}].\mathbf{Id}(A[\mathbf{p}^2],\mathbf{q}[\mathbf{p}],\mathbf{q})\vdash C\,\mathsf{type} \qquad \Gamma.A\vdash c:C[\mathbf{id}.\mathbf{q}.\mathbf{refl}]}{\Gamma\vdash \mathbf{j}(c,\mathbf{refl})=c[\mathbf{id}.a]:C[\mathbf{id}.a.a.\mathbf{refl}]}$$

These rules complete our definition of **Id**-types and thus of intensional type theory.

As with the eliminators of **Void**, **Bool**, and **Nat**, it can be helpful to think of $\mathbf{j}(c,p)$ as somehow "pattern-matching on p" with clause c.

$$\begin{array}{l}\mathbf{match}\ (a,b,p)\ \mathbf{with}\\ \quad (a,a,\mathbf{refl})\to c\ a\end{array}$$

From this perspective, the definitional equality $\mathbf{j}(c,\mathbf{refl})=c[\mathbf{id}.a]$ states that the entire **match** expression reduces to c when (a,b,p) is indeed of the form $(a,a,\mathbf{refl})$.

Remark 4.2.3. The name of **j** for **Id**-elimination dates back to Martin-Löf [Mar84a], in which Martin-Löf notates **Id**-types as I, and he seems to have chosen J simply because it is the next letter of the alphabet. At any rate, unlike **Id**entity or **refl**exivity, it has no obvious meaning as the initial letter of some pre-existing mathematical term.

For readers who might find this notational choice to be singularly arbitrary, we recall Scott's story of mailing Church a postcard asking why λ was chosen as the symbol for function abstraction in his lambda calculus, and receiving the same postcard with the annotation "eeny, meeny, miny, moe" [Sco18]. ◇

Like extensional type theory, intensional type theory satisfies consistency and canonicity; unlike extensional type theory, it also satisfies the metatheorems on open terms discussed in Chapter 3 and is therefore exceptionally well-behaved from the perspective of both theory and implementability.

Theorem 4.2.4 (Martin-Löf [Mar71; Mar75] and Coquand [Coq91])**.** *Intensional type theory satisfies consistency, canonicity, normalization, and has invertible type constructors.*

One typically deduces all of these properties from the proof of normalization: given that normalization amounts to concretely characterizing the sets $\mathsf{Tm}(\Gamma, A)$ for all Γ, A, consistency and canonicity amount to verifying that these characterizations of $\mathsf{Tm}(\mathbf{1}, \mathbf{Void})$ and $\mathsf{Tm}(\mathbf{1}, \mathbf{Bool})$ contain zero and two elements respectively, and invertibility of $\mathbf{\Pi}$-types amounts to inverting the induced $\mathbf{\Pi}(-,-)$ map on normal forms. There are many proofs of normalization for intensional type theory and minor variations on it, some relying on semantic model constructions [AK16; Coq19; Ste21] and others more closely connected to algorithms used in real implementations [ACD07; Abe13; AÖV17].

From j to subst **and** uniq We close this section by showing that **j** is interprovable with the combination of subst and uniq, starting by deriving subst and uniq from **j**.

Notation 4.2.5. Our $\mathbf{j}(b, p)$ notation is not well-suited to informal constructions with named variables, because b silently binds a variable of type A, and moreover, the type C can be hard to infer by inspection. In our informal notation we will therefore wrap **j** as a function with the following type, satisfying the definitional equality $\mathsf{j}\ B\ b\ a\ a\ \mathbf{refl} = b\ a$.

$$\mathsf{j} : \{A : \mathbf{U}\}\ (C : (a\ b : A) \to \mathbf{Id}(A, a, b) \to \mathbf{U}) \to ((a : A) \to C\ a\ a\ \mathbf{refl}) \to (a\ b : A)\ (p : \mathbf{Id}(A, a, b)) \to C\ a\ b\ p$$

Likewise we introduce the functions $\mathsf{pi}, \mathsf{sig} : (A : \mathbf{U})\ (B : A \to \mathbf{U}) \to \mathbf{U}$ as wrappers for the codes $\mathbf{pi}(-,-)$ and $\mathbf{sig}(-,-)$ respectively.

Exercise 4.4. Use the elimination principle **j** to define the function j above, and check that your definition of j satisfies the stipulated definitional equality.

The flexibility and complexity of **j** come from the fact that the *motive* [McB02] C can depend not only on the two elements of A but also the identification itself, both in arbitrary ways; many principles fall immediately out of **j** given a sufficiently clever choice of C.

Lemma 4.2.6. *Using* j *we can define* subst*, i.e., a term of type*

$$\mathsf{subst} : \{A : \mathbf{U}\}\ \{a\ a' : A\} \to (B : A \to \mathbf{U}) \to \mathbf{Id}(A, a, a') \to B\ a \to B\ a'$$

satisfying the definitional equality subst B **refl** $b = b$.

Proof. We will apply j to the same $a, a' : A$ and $p : \mathbf{Id}(A, a, a')$ as subst, choosing a motive such that the type of the fully-applied j will be $B\ a \to B\ a'$:

$$C\ x\ y\ _ = \mathsf{pi}\ (B\ x)\ (\lambda_ \to B\ y)$$

We have $C\ a\ a'\ p = B\ a \to B\ a'$ as desired, and it remains only to exhibit a term of type $(a : A) \to C\ a\ a\ \mathbf{refl} = (a : A) \to B\ a \to B\ a$, which is easy to do. In total:

$$\begin{aligned}&\mathsf{subst} : \{A : \mathbf{U}\}\ \{a\ a' : A\} \to (B : A \to \mathbf{U}) \to \mathbf{Id}(A, a, a') \to B\ a \to B\ a'\\ &\mathsf{subst}\ \{A\ a\ a'\}\ B\ p = \mathsf{j}\ (\lambda x\ y\ _ \to \mathsf{pi}\ (B\ x)\ (\lambda_ \to B\ y))\ (\lambda_\ x \to x)\ a\ a'\ p\end{aligned}$$

The reader can verify that the stipulated definitional equality holds. □

Exercise 4.5. Check that the above definition of subst satisfies the required equation.

Lemma 4.2.7. *Using* j *we can define* uniq*, i.e., a term of type*

$$\mathsf{uniq} : \{A : \mathbf{U}\}\ \{a : A\} \to (x : [a]) \to \mathbf{Id}([a], (a, \mathbf{refl}), x)$$

satisfying the definitional equality uniq $(a, \mathbf{refl}) = \mathbf{refl}$.

Proof. Writing $A : \mathbf{U}$, $a : A$, and $x := (b, p) : \sum_{b:A} \mathbf{Id}(A, a, b)$ for the arguments of uniq, we will apply j to a, b, p with a motive that allows us to reduce the general case of a, b, p to the particular and easy case of $a, a, \mathbf{refl}$:

$$C\ x\ y\ p' = \mathbf{id}(\mathsf{sig}\ A\ (\lambda z \to \mathbf{id}(A, x, z)), (x, \mathbf{refl}), (y, p'))$$

Then $C\ a\ b\ p = \mathbf{Id}([a], (a, \mathbf{refl}), (b, p))$, and it remains only to exhibit a term of type $(a : A) \to C\ a\ a\ \mathbf{refl} = (a : A) \to \mathbf{Id}([a], (a, \mathbf{refl}), (a, \mathbf{refl}))$, which is again easy:

$$\begin{aligned}&\mathsf{uniq} : \{A : \mathbf{U}\}\ \{a : A\} \to (x : [a]) \to \mathbf{Id}([a], (a, \mathbf{refl}), x)\\ &\mathsf{uniq}\ \{A\ a\}\ (b, p) = \mathsf{j}\ (\lambda x\ y\ p' \to \mathbf{id}(\mathsf{sig}\ A\ (\lambda z \to \mathbf{id}(A, x, z)), (x, \mathbf{refl}), (y, p')))\\ &\quad (\lambda x \to \mathbf{refl}_{(x,\mathrm{refl})})\ a\ b\ p\end{aligned}$$

The reader can again verify that the stipulated definitional equality holds.

Note that unlike the motive we used in Lemma 4.2.6, the motive here depends not only on $x, y : A$ but also the identification $p' : \mathbf{Id}(A, x, y)$. Note also that the motive actually generalizes our goal: rather than proving that for a fixed $a : A$ we can identify $(a, \mathbf{refl})$ and $(b, p) : [a]$, we prove that for any $x, y : A$ we can identify $(x, \mathbf{refl})$ and $(y, p') : [x]$. □

Exercise 4.6. Check that the above definition of uniq satisfies the required equation.

And back again Conversely, using subst and uniq it is also possible to define a term j satisfying the required definitional equality. We leave most of the construction to the reader in the following series of exercises. In these exercises we fix the arguments of j as $A : \mathbf{U}$, $C : (a\ b : A)\ (p : \mathbf{Id}(A, a, b)) \to \mathbf{U}$, $c : (a : A) \to (C\ a\ a\ \mathbf{refl})$, $a, b : A$, and $p : \mathbf{Id}(A, a, b)$, and we define the following "partially uncurried" type family:

$$C_a : (x : [a]) \to \mathbf{U}$$
$$C_a\ x = C\ a\ (\mathbf{fst}\ x)\ (\mathbf{snd}\ x)$$

Exercise 4.7. Define a term $c_a : C_a\ (a, \mathbf{refl})$.

Exercise 4.8. Without using **j**, define a term $q : \mathbf{Id}([a], (a, \mathbf{refl}), (b, p))$.

Exercise 4.9. Using c_a and q but not **j**, define a term $c_b : C_a\ (b, p)$.

Exercise 4.10. Show that the type of c_b is equal to $C\ a\ b\ p$, and use this to combine the previous three exercises into a definition of j that uses subst and uniq but not **j**.

Exercise 4.11. Check that your solution to Exercise 4.10 satisfies $\mathsf{j}\ C\ c\ a\ a\ \mathbf{refl} = c\ a$.

Exercise 4.12. We have seen in Remark 4.1.3 and Exercise 4.2 that subst and uniq are definable for **Eq**-types in ETT; from Exercise 4.10 it follows that j is also definable in ETT for **Eq**-types. Give an explicit definition of j for **Eq**-types in ETT. (Hint: you can combine the above results, but it is also fairly straightforward to arrive at the definition independently.)

Although it is perhaps easier to wrap one's head around subst and uniq rather than **j**, as we noted in Remark 4.2.1 it is often more straightforward in practice to use **j** directly. Consider for instance the function cong (Lemma 4.1.6) which we really ought to have stated for *dependent* functions:

$$\mathsf{dcong} : \{A : \mathbf{U}\}\ \{B : A \to \mathbf{U}\}\ (f : (a : A) \to B\ a)\ \{a\ a' : A\}\ (p : \mathbf{Id}(A, a, a')) \to \mathbf{Id}(B\ a', \mathsf{subst}\ B\ p\ (f\ a), f\ a')$$

Defining dcong in terms of subst and uniq is a headache, because one must use both simultaneously to handle the occurrence of p in the type. It is, however, straightforward to define with **j**:

$$\mathsf{dcong}\ f = \mathsf{j}\ (\lambda a\ a'\ p \to \mathbf{Id}(B\ a', \mathsf{subst}\ B\ p\ (f\ a), f\ a'))\ (\lambda a \to \mathbf{refl}_{f(a)})$$

Exercise 4.13. Use j to give an alternative definition of sym from Lemma 4.1.4.

Exercise 4.14. Prove Lemma 4.1.7 for the definition of sym constructed in Exercise 4.13.

4.3 *Limitations of the intensional identity type*

We have now seen that the rules for **Id**-types are well-motivated from a theoretical perspective as the mapping-out formulation of equality, and that they support the operations of subst and uniq presented in Section 4.1, which in turn imply many properties including the symmetry, transitivity, and congruence of equality. We have also seen that ITT is more well-behaved than ETT (Theorem 4.2.4), and that all the rules of **Id**-types are validated by the **Eq**-types of ETT (Exercise 4.12).

Have we even lost anything at all by moving from ETT to ITT? Well, yes; the entire point of moving to ITT was to remove equality reflection from our theory, in light of its undecidability (Section 3.6). Removing equality reflection does come at a cost: in ETT whenever we can prove $p : \mathbf{Eq}(A, a, a')$ we can freely use terms of type $B\ a$ at type $B\ a'$, but in ITT we must explicitly appeal to the proof p with $\mathsf{subst}\ B\ p : B\ a \to B\ a'$.

So then are types and terms of ITT simply more *bureaucratic* than those of ETT, or does ITT actually "prove fewer statements" than ETT in some meaningful sense? This is an excellent question, and one that requires some care to set up precisely.

Given that closed types (of a consistent type theory) can be seen as logical propositions and their terms as their proofs, we might naïvely wonder *is every non-empty closed type of ETT also non-empty in ITT?* This question does not make sense as posed because, by equality reflection, well-formed types in ETT need not be well-formed in ITT. Consider for instance the following closed type of ETT:

$$(p : \mathbf{Eq}(\mathbf{Bool}, \mathbf{true}, \mathbf{false})) \to \mathbf{Eq}(\mathbf{Eq}(\mathbf{Bool}, \mathbf{true}, \mathbf{false}), \mathbf{refl}, p)$$

On the other hand, closed types of ITT *do* correspond to closed types of ETT in a more-or-less straightforward way, because their rules differ only in their choice of equality type, and the **Eq**-types of ETT satisfy all the rules of the **Id**-types of ITT (and more); to make this translation precise we once again turn to model theory.

Definition 4.3.1. We define a *model of ITT*, a *homomorphism of models of ITT*, and the *syntactic model* $\mathcal{T}_{\mathrm{ITT}}$ *of ITT* following Definitions 3.4.2 to 3.4.4, but replacing the structure corresponding to **Eq**-types with that of **Id**-types; as in Theorem 3.4.5, $\mathcal{T}_{\mathrm{ITT}}$ is the initial model of ITT. For clarity we rename the subjects of Definitions 3.4.2 to 3.4.4 to *model of ETT*, *homomorphism of models of ETT*, and *syntactic model* $\mathcal{T}_{\mathrm{ETT}}$ *of ETT*.

Theorem 4.3.2. *The underlying sets of* $\mathcal{T}_{\mathrm{ETT}}$ *support a model of ITT.*

Proof. Intuitively, this means that the syntax of ETT "satisfies the rules of ITT." Formally, we construct a model $\mathcal{M}$ of ITT whose contexts are the contexts of the syntax of ETT, $\mathrm{Cx}_{\mathcal{M}} \coloneqq \mathrm{Cx}_{\mathcal{T}_{\mathrm{ETT}}}$; whose substitutions are the substitutions of the syntax of ETT, $\mathrm{Sb}_{\mathcal{M}}(\Delta, \Gamma) \coloneqq \mathrm{Sb}_{\mathcal{T}_{\mathrm{ETT}}}(\Delta, \Gamma)$; and likewise for types and terms. For all the rules of ITT that are also present in ETT, we choose the corresponding structure, e.g., $\mathbf{1}_{\mathcal{M}} \coloneqq \mathbf{1}_{\mathcal{T}_{\mathrm{ETT}}}$.

The only subtlety is how to define the **Id**-types of $\mathcal{M}$, and for this we choose the **Eq**-types of $\mathcal{T}_{\mathrm{ETT}}$, i.e., $\mathbf{Id}_{\mathcal{M}}(A, a, b) \coloneqq \mathbf{Eq}_{\mathcal{T}_{\mathrm{ETT}}}(A, a, b)$ and $\mathbf{refl}_{\mathcal{M}} \coloneqq \mathbf{refl}_{\mathcal{T}_{\mathrm{ETT}}}$. The reader has already verified in Exercise 4.12 that the **j** eliminator is definable in ETT. □

Corollary 4.3.3. *There is a function* $[\![-]\!]$ *that sends contexts (resp., substitutions, types, terms) of ITT to contexts (resp., substitutions, types, terms) of ETT.*

Proof. By Theorem 4.3.2 and the initiality of the syntactic model of ITT, there is a unique homomorphism $f : \mathcal{T}_{\mathrm{ITT}} \to \mathcal{M}$ of models of ITT, and thus a function $\mathrm{Cx}_f : \mathrm{Cx}_{\mathcal{T}_{\mathrm{ITT}}} \to \mathrm{Cx}_{\mathcal{M}} = \mathrm{Cx}_{\mathcal{T}_{\mathrm{ETT}}}$ and likewise for substitutions, types, and terms. □

By construction, this translation $[\![-]\!]$ of ITT to ETT "does nothing" except at **Id**-types, where $[\![\mathbf{Id}(A, a, b)]\!] = \mathbf{Eq}([\![A]\!], [\![a]\!], [\![b]\!])$. Intuitively, this is possible because **Eq**-types are defined to have only **refl** as elements, which is strictly stronger than the definition of **Id**-types as "appearing *to other types* to have only **refl** as elements."

Exercise 4.15. Using Corollary 4.3.3, describe how any model of ETT induces a model of ITT. Conclude that the set model of ETT described in Section 3.5 induces a model of ITT which we will call the *set model of ITT*.

Exercise 4.16. Prove that intensional type theory is consistent. (Hint: use Exercise 4.15 and adapt the proof of Theorem 3.4.8.)

We can now ask a more precise question:

Question 4.3.4. *Suppose that* $\mathbf{1} \vdash A\,\mathrm{type}$ *in ITT, and that in ETT there is a term* $\mathbf{1} \vdash a : [\![A]\!]$. *Then does there necessarily exist a term* $\mathbf{1} \vdash a' : A$ *in ITT?*

Remark 4.3.5. Types containing at least one term are said to be *inhabited* (Definition 2.7.2), so Question 4.3.4 equivalently asks, "if $[\![A]\!]$ is inhabited in ETT, is A inhabited in ITT?" ◇

By focusing only on types that are well-formed in ITT, this formulation avoids the pitfalls discussed earlier. Perhaps the converse of Question 4.3.4 is more intuitive: *do there exist types that can be formed **without** equality reflection, but that can only be inhabited **with** equality reflection?* Unfortunately, such types *do* exist, and thus the answer to Question 4.3.4 is *no*; even worse, the counterexamples are ones that users of type theory are likely to encounter frequently in practice.

Independence The famed propositions-as-types correspondence (Section 2.7) states that types can be read as logical propositions and terms as proofs. Under this reading, counterexamples to Question 4.3.4 are propositions that are *independent* of intensional type theory, i.e., propositions A for which neither A nor $A \to \mathbf{Void}$ are provable.[4]

Lemma 4.3.6. *If* $\mathbf{1} \vdash A\,\mathsf{type}$ *is a counterexample to Question 4.3.4, then* A *is independent of intensional type theory.*

Proof. By definition, there must exist a term $\mathbf{1} \vdash a : [\![A]\!]$ in ETT, but no term $\mathbf{1} \vdash a' : A$ in ITT. Thus A is by definition not provable in ITT, so it suffices to show that $A \to \mathbf{Void}$ is also not provable in ITT. Suppose that there were a term $\mathbf{1} \vdash f : A \to \mathbf{Void}$ in ITT; then there would also be a term $\mathbf{1} \vdash [\![f]\!] : [\![A]\!] \to \mathbf{Void}$ in ETT, but this would mean there is a closed proof $[\![f]\!](a)$ of $\mathbf{Void}$ in ETT, contradicting its consistency (Theorem 3.4.8). □

Of course, there are other kinds of independent propositions too; as a sufficiently strong formal system, ITT is subject to Gödel's incompleteness theorem and thus one can construct independent propositions roughly corresponding to "the type of consistency proofs of ITT." But for now we restrict our attention to counterexamples to Question 4.3.4, exploring two in particular: function extensionality and uniqueness of identity proofs.

4.3.1 *Function extensionality*

The principle of *function extensionality* states that for any two functions $f, g : (a : A) \to B(a)$, if $f(a)$ and $g(a)$ are equal for all $a : A$, then f and g are equal. We reproduce the formal statement of Funext below, along with its non-dependent special case Funext′:

$$\mathsf{Funext} = (A : \mathbf{U}) \to (B : A \to \mathbf{U}) \to (f\ g : (a : A) \to B\ a) \to$$

[4] For the purposes of this section we refer only to the naïve reading of *all* types as propositions (Slogan 2.7.1), ignoring for the moment any issues related to mere propositions (Sections 2.7 and 5.1).

$$((a : A) \to \mathbf{Id}(B\ a, f\ a, g\ a)) \to \mathbf{Id}((a : A) \to B\ a, f, g)$$

$$\begin{aligned}\mathsf{Funext}' = (A\ B : \mathbf{U}) &\to (f\ g : A \to B) \to \\ ((a : A) &\to \mathbf{Id}(B, f\ a, g\ a)) \to \mathbf{Id}(A \to B, f, g)\end{aligned}$$

Both of these are counterexamples to Question 4.3.4 and thus independent of ITT. First, we check that $[\![\mathsf{Funext}]\!]$ is provable in ETT.

Exercise 4.17. Construct a closed term of type $[\![\mathsf{Funext}]\!]$ in extensional type theory.

Next, we must check that Funext is not provable in intensional type theory. As with consistency (Theorem 3.4.7), it suffices to exhibit a model of ITT in which the set of closed terms of type Funext is empty. However, it is surprisingly difficult to do so![5] One such model is—tautologically—the syntax of ITT itself, or $\mathcal{T}_{\mathrm{ITT}}$; however, showing that this is the case is precisely what we are already trying to prove. A more useful observation is that the models used to prove normalization contain concrete characterizations of $\mathsf{Tm}(\Gamma, A)$ for all Γ, A and thus it is possible to unfold such a model and explicitly verify that there are no normal forms—and hence no elements whatsoever—of $\mathsf{Tm}(\mathbf{1}, \mathsf{Funext})$ [Hof95a].

Remark 4.3.7. The latter approach is tantamount to the proof-theoretic technique of showing that a formula is not derivable by proving cut elimination for a calculus and then checking by induction that the formula has no cut-free proofs. ◇

One can also imagine more "mathematical" (and non-initial) models that refute function extensionality. An early example of such a model based on realizability and gluing was given by Streicher [Str93, Chapter 3]; a more recent example is the (categorical) polynomial model of von Glehn [vGle14]. In both cases the model construction is somewhat involved but checking that they refute Funext is comparatively straightforward. In any case, any of these arguments allows us to conclude:

Theorem 4.3.8. *There is no closed term of type* Funext *in intensional type theory.*

The authors are uncertain to whom this result should be attributed. Turner [Tur89] suggests that it was known to Martin-Löf and it was certainly known to type theorists in the 1980s, but the earliest explicit discussion of the independence of Funext we have located is the countermodel of Streicher [Str93].

There are many examples of function extensionality arising in practice. For instance, in ITT we can prove $(n\ m : \mathbf{Nat}) \to \mathbf{Id}(\mathbf{Nat}, n + m, m + n)$ but not $\mathbf{Id}(\mathbf{Nat} \to$

[5] There are many simple "countermodels of function extensionality" which fail to validate the η-rule of $\mathbf{\Pi}$-types and are therefore not models of ITT as we have defined it. They are, however, models of the calculus of inductive constructions, which lacks η for $\mathbf{\Pi}$-types.

Nat → **Nat**, (+), (+) ∘ flip). Similarly, although mergeSort, bubbleSort : **List Nat** → **List Nat** agree on all inputs, we cannot prove they are equal functions. This has real consequences in practice: if we write a function that calls bubbleSort, is it equal to the same function where these calls have been replaced by calls to mergeSort? If function extensionality held this would follow immediately from cong; as it stands, one must manually argue that the text of the function respects swapping subroutines in this way—even though it is impossible to define a function that *doesn't*!

We view the independence of function extensionality as perhaps the greatest failing of intensional type theory, as it frequently causes problems with no benefit,[6] and it is therefore common to simply *postulate* Funext when working in ITT, that is, to add a rule

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathsf{funext} : \mathsf{Funext}}$$ ✎

Postulating an axiom in this way is equivalent to prepending every context by a variable of type Funext, and it therefore preserves normalization (a property of *all* contexts) while disrupting canonicity (a property of the *empty* context, which is "no longer empty").[7]

Exercise 4.18. Argue that postulating Funext causes canonicity to fail. That is, produce a closed term of type **Bool** in ITT adjoined with the above rule that appears to be judgmentally equal to neither **true** nor **false**. (You do not need to formally prove this fact.)

4.3.2 *Uniqueness of identity proofs*

Our second counterexample to Question 4.3.4 is the principle of *uniqueness of identity proofs* (UIP), which states that any two identifications between the same two terms are themselves identified.

$$\mathsf{UIP} = (A : \mathbf{U}) \to (a\ b : A) \to (p\ q : \mathbf{Id}(A, a, b)) \to \mathbf{Id}(\mathbf{Id}(A, a, b), p, q)$$

In short, UIP asserts that identifications are unique: up to identification, there is at most one proof of $\mathbf{Id}(A, a, b)$ for any $a, b : A$. Types with at most one element are

[6] There are occasions where one may wish to *not* identify all pointwise-equal procedures, e.g., when studying the runtime of algorithms, but we stress that ITT also does not allow us to *distinguish* pointwise-equal functions; studying runtime in this way requires other axioms and, likely, the removal of β-rules.

[7] Canonicity continues to hold in the presence of certain classes of axioms, e.g., consistent axioms of the form $A \to$ **Void** [Coq+13].

called *propositions* (Sections 2.7 and 5.1), so we might equivalently phrase UIP as the principle that propositional equality is a proposition.[8] Like Funext, UIP is independent of ITT. On the one hand, it holds in ETT and thus cannot be *refuted* by ITT:

Exercise 4.19. Construct a closed term of type $[\![\mathsf{UIP}]\!]$ in extensional type theory.

To see that UIP is not *provable* in ITT, it again suffices to exhibit a countermodel, a model of ITT in which the set of closed terms of type UIP is empty. The original such countermodel, the *groupoid model of type theory* of Hofmann and Streicher [HS98], is both instructive and historically significant as a precursor to homotopy type theory (Chapter 5), so unlike the countermodels of Funext we will sketch it in some detail.

The groupoid model is similar to the set-theoretic model of type theory (Section 3.5) except that it replaces sets with groupoids, sets equipped with additional structure:

Definition 4.3.9. A *groupoid* $X = (|X|, \mathcal{R}, \mathsf{id}, (-)^{-1}, \circ)$ consists of a set $|X|$, a family of sets $\mathcal{R}$ indexed over $|X| \times |X|$, and dependent functions:

- $\mathsf{id} : \{x : |X|\} \to \mathcal{R}(x, x)$,
- $(-)^{-1} : \{x\ y : |X|\} \to \mathcal{R}(x, y) \to \mathcal{R}(y, x)$, and
- $(\circ) : \{x\ y\ z : |X|\} \to \mathcal{R}(y, z) \to \mathcal{R}(x, y) \to \mathcal{R}(x, z)$,

such that $\mathsf{id} \circ f = f = f \circ \mathsf{id}$, $f \circ f^{-1} = \mathsf{id}$, $\mathsf{id} = f^{-1} \circ f$, and $f \circ (g \circ h) = (f \circ g) \circ h$.

Definition 4.3.10. Given two groupoids X, Y, a *homomorphism of groupoids* $F : X \to Y$ is a pair of functions $F_0 : |X| \to |Y|$ and $F_1 : \{x\ x' : |X|\} \to \mathcal{R}_X(x, x') \to \mathcal{R}_Y(F_0(x), F_0(x'))$ for which F_1 commutes with the groupoid operations, i.e.,

- $F_1(\mathsf{id}) = \mathsf{id}$,
- $F_1(f^{-1}) = F_1(f)^{-1}$, and
- $F_1(g \circ f) = F_1(g) \circ F_1(f)$.

Exercise 4.20. For categorically-minded readers: argue that a groupoid is exactly the same as a category all of whose morphisms are isomorphisms, and a homomorphism of groupoids is exactly a functor.

Advanced Remark 4.3.11. The name "groupoid" comes from the perspective that these are a weaker notion of group in which the multiplication is a partial operation. ◇

[8] The terminology of "propositional equality" is perhaps ill-advised.

We can think of a groupoid as equipping its underlying set with a "proof-relevant notion of equality" which like ordinary equality is reflexive, symmetric, transitive, and respected by functions (groupoid homomorphisms), but unlike ordinary equality "can hold in more than one way." Following this intuition, we will model closed types $A \in \mathsf{Ty}(\mathbf{1})$ not as sets X but as groupoids $(|X|, \mathcal{R}, \dots)$, closed terms $a \in \mathsf{Tm}(\mathbf{1}, A)$ as elements of $|X|$, and closed identifications $p \in \mathsf{Tm}(\mathbf{1}, \mathbf{Id}(A, a, b))$ as elements of $\mathcal{R}(a, b)$.

Before outlining the model itself, we give a few examples of groupoids.

Example 4.3.12. Every set A can be regarded as a *discrete groupoid* ΔA in which $\mathcal{R}_{\Delta A}(x, y) = \{\star \mid x = y\}$. The remaining structure is uniquely determined: $\mathsf{id} = \star$, $\star^{-1} = \star$, etc.

Example 4.3.13. Given two groupoids X, Y, the set of groupoid homomorphisms $X \to Y$ (Definition 4.3.10) admits a natural groupoid structure in which

$$\mathcal{R}_{X \to Y}(F, G) = \\ \{T : (x : |X|) \to \mathcal{R}(F_0\, x, G_0\, x) \mid \forall f : \mathcal{R}(x, y).\ G_1(f) \circ T(x) = T(y) \circ F_1(f)\}$$

In light of Exercise 4.20, categorically-minded readers might observe that T is exactly a natural transformation from F to G. We leave the remaining structure as an exercise.

Example 4.3.14. For an explicit example of a groupoid that is not discrete, consider the groupoid traditionally called $B(\mathbb{Z}/2)$, whose underlying set is the singleton $\{\star\}$, $\mathcal{R}_{B(\mathbb{Z}/2)}(\star, \star) = \mathbb{Z}/2 = \{0, 1\}$, and the remaining structure is as follows:

$$\begin{aligned} \mathsf{id} &= 0 \\ x \circ y &= x + y \mod 2 \\ x^{-1} &= x \end{aligned}$$

The reader can check that these operations satisfy the necessary equations. (Hint: this is equivalent to checking that $\mathbb{Z}/2$ with the above id, $\circ$, and $(-)^{-1}$ forms a group.)

Example 4.3.15. There is a "large" groupoid $\mathbf{S}$ of all "small" sets, where $\mathcal{R}_{\mathbf{S}}(X, Y)$ is the set of bijections between the sets X and Y, and the operations are the identity, inverse, and composition of bijections. This groupoid is not discrete because there can be more than one bijection between a pair of sets, e.g., $\mathsf{id}, \mathsf{swap} \in \mathcal{R}_{\mathbf{S}}(\{\star, \star'\}, \{\star, \star'\})$.

Example 4.3.16. There is a "large" groupoid $\mathbf{G}$ of all "small" groupoids, whose underlying collection is the proper class of all groupoids, and for which $\mathcal{R}_{\mathbf{G}}(X, Y)$ is the set of all groupoid isomorphisms (invertible homomorphisms, or homomorphisms for which F_0 and each F_1 are bijections) from X to Y. The groupoid $\mathbf{S}$ from Example 4.3.15 embeds into $\mathbf{G}$, so $\mathbf{G}$ is also not discrete.

As in the set-theoretic model of type theory, groupoids and groupoid-indexed families of groupoids form a model of type theory. Writing $\mathcal{G}$ for the groupoid model of (intensional) type theory and $f : \mathcal{T}_{\mathrm{ITT}} \to \mathcal{G}$ for the homomorphism from the syntactic model to $\mathcal{G}$, f interprets syntactic contexts Γ as groupoids $\mathsf{Cx}_f(\Gamma)$, the closed context $\mathbf{1}$ as the one-element, one-identification groupoid, syntactic substitutions as groupoid homomorphisms, and syntactic types $A \in \mathsf{Ty}(\Gamma)$ as $\mathsf{Cx}_f(\Gamma)$-indexed families of groupoids $(\mathsf{Ty}_f(\Gamma)(A))_{\gamma \in \mathsf{Cx}_f(\Gamma)}$. Such a family assigns to each groupoid element $\gamma \in \mathsf{Cx}_f(\Gamma)$ a groupoid $(\mathsf{Ty}_f(\Gamma)(A))_\gamma$, and to each identification $\alpha \in \mathcal{R}_{\mathsf{Cx}_f(\Gamma)}(\gamma, \gamma')$ a homomorphism $(\mathsf{Ty}_f(\Gamma)(A))_\gamma \to (\mathsf{Ty}_f(\Gamma)(A))_{\gamma'}$ in a manner compatible with identity and composition. (Using Example 4.3.16, we can repackage the data of such a family quite simply as a groupoid homomorphism $\mathsf{Cx}_f(\Gamma) \to \mathbf{G}$.) Finally, f interprets syntactic terms $a \in \mathsf{Tm}(\Gamma, A)$ as dependent functions assigning to each element $\gamma \in \mathsf{Cx}_f(\Gamma)$ of the context an element of the groupoid $(\mathsf{Ty}_f(\Gamma)(A))_\gamma$ in a manner that respects identifications. (We can again phrase this condition as a groupoid homomorphism, but we will not pursue the details further.)

Most of the structure of the groupoid model of type theory mirrors that of the set-theoretic model, with some added complication to account for identifications; for example, rather than interpreting the universe as the large set of all small sets, we interpret it as the large groupoid $\mathbf{G}$ of all small groupoids (Example 4.3.16). The key departure is in the interpretation of **Id**-types: for closed $A \in \mathsf{Ty}_{\mathcal{G}}(\mathbf{1}_{\mathcal{G}})$ and $a, b \in \mathsf{Tm}_{\mathcal{G}}(\mathbf{1}_{\mathcal{G}}, A)$, the $\mathcal{G}$-identity type $\mathbf{Id}_{\mathcal{G}}(A, a, b)$ is chosen to be (the discrete groupoid on) the set of identifications in the groupoid A between a and b, namely $\Delta\mathcal{R}_A(a, b)$.

It is not at all obvious that such an interpretation supports **j**, but this is the force of the groupoid model: because all types and terms respect identifications, it is in fact the case that dependent functions from **Id**-types into any $\mathcal{G}$-type are generated by the data of where to send **refl**. Interested readers can find these and all the other details in the paper of Hofmann and Streicher [HS98].

Theorem 4.3.17 (Hofmann and Streicher [HS98])**.** *There is no closed term of type* UIP *in intensional type theory.*

Proof. This follows immediately from the fact that the groupoid model interprets UIP as the empty groupoid, whose proof we sketch below. Recall that:

$$\mathsf{UIP} = (A : \mathbf{U}) \to (a\ b : A) \to (p\ q : \mathbf{Id}(A, a, b)) \to \mathbf{Id}(\mathbf{Id}(A, a, b), p, q)$$

A term of this type in $\mathcal{G}$ would be a dependent function out of the interpretation of $\mathbf{U}$, which is the groupoid of groupoids $\mathbf{G}$. Suppose that such a function exists; then we could apply it to the groupoid $B(\mathbb{Z}/2) \in \mathbf{G}$ defined in Example 4.3.14, then twice to the unique element $\star \in |B(\mathbb{Z}/2)|$ of that groupoid, and then to the two distinct identifications $0, 1 \in \mathcal{R}_{B(\mathbb{Z}/2)}(\star, \star)$. The result would have to be a proof that $0 = 1$, which is false. □

4.3.2.1 *Towards homotopy type theory*

The busy reader may wish to skip this section initially. The groupoid model demonstrates that **Id**-types support richer interpretations than merely equations: identifications can be any data that is respected by all the constructs of type theory.

Although the groupoid model provides us with interesting examples of identity types, we note that the identity types of any groupoid X, $\Delta\mathcal{R}_X(x, y)$, are always discrete groupoids with no interesting identifications of their own. Thus the groupoid model *does* validate the "uniqueness of identity proofs of identity proofs":

$$\begin{aligned}\mathsf{UIPIP} = (A : \mathbf{U}) \to (a\ b : A) \to (p\ q : \mathbf{Id}(A, a, b)) \to \\ (\alpha\ \beta : \mathbf{Id}(\mathbf{Id}(A, a, b), p, q)) \to \mathbf{Id}(\mathbf{Id}(\mathbf{Id}(A, a, b), p, q), \alpha, \beta)\end{aligned}$$

Like UIP, this principle is also independent of ITT, and we can construct a countermodel in 2-*groupoids*, which contain a second level of "2-identifications" $\mathcal{R}^2(p, q)$ between any pair of identifications $p, q \in \mathcal{R}(a, b)$ between elements a, b. Although we will not define these precisely, we note that the passage from groupoids to 2-groupoids is analogous to the passage from sets to groupoids; for instance, every groupoid can be regarded as a discrete 2-groupoid with the same elements and 1-identifications but with trivial 2-identifications.

The story once again repeats for the 2-groupoid model of type theory, and in fact for any n: there is a model of ITT in which closed types are interpreted as n-groupoids, and this model refutes $\mathsf{U}(\mathsf{IP})^n$ but validates $\mathsf{U}(\mathsf{IP})^{n+1}$. In fact, this suggests correctly that ordinary groupoids ought to be thought of as 1-*groupoids* and sets as 0-*groupoids*; indeed, the set (0-groupoid) model of type theory validates $\mathsf{U}(\mathsf{IP})^1$. Looking downward, the large 0-groupoid of (-1)-groupoids is the *set of propositions* $\{\emptyset, \{\star\}\}$.

But what about for *all* n? Is it possible to construct a model that simultaneously refutes $\mathsf{U}(\mathsf{IP})^n$ for every $n \in \mathbb{N}$? Intuitively, such a model would have to interpret closed types as "∞-groupoids" with countably infinite towers of identifications. The answer is *yes* [War08, Corollary 4.26], and in fact Voevodsky's simplicial model of homotopy type theory [KL21] can be seen as precisely such a model [KS15].

4.3.3 *Hofmann's conservativity theorem*

We have generated an infinite stream of counterexamples to Question 4.3.4, or propositions that are provable in ETT but not ITT, namely Funext and $\mathsf{U}(\mathsf{IP})^n$ for $n \geq 1$. Is there a third class of counterexamples? Surprisingly, *no*: all counterexamples to Question 4.3.4 are generated by Funext and UIP in a precise sense. (Note that UIP implies $\mathsf{U}(\mathsf{IP})^n$ for $n > 1$.)

To state this claim more precisely, let us write

$$\Gamma_{\mathrm{ax}} := \mathbf{1}, \mathit{funext} : \mathsf{Funext}, \mathit{uip} : \mathsf{UIP}$$

for the ITT context containing two variables, one of type Funext and one of type UIP; types and terms of ITT in context Γ_{ax} are in bijection with closed types and terms of intensional type theory extended by two rules postulating Funext and UIP. Then, Hofmann's celebrated *conservativity result* states that:

Theorem 4.3.18 (Hofmann [Hof95a])**.** *Suppose that* $\Gamma_{\mathrm{ax}} \vdash A\,\mathsf{type}$ *in ITT, and* $[\![\Gamma_{\mathrm{ax}}]\!] \vdash a : [\![A]\!]$ *in ETT; then there exists a term* $\Gamma_{\mathrm{ax}} \vdash a' : A$ *in ITT.*

In Exercises 4.17 and 4.19 the reader has constructed proofs $\mathbf{1} \vdash p : [\![\mathsf{Funext}]\!]$ and $\mathbf{1} \vdash q : [\![\mathsf{UIP}]\!]$ of function extensionality and UIP in ETT, so we can discharge the hypotheses of $[\![\Gamma_{\mathrm{ax}}]\!]$ to obtain the following corollary:

Corollary 4.3.19. *If* $\Gamma_{\mathrm{ax}} \vdash A\,\mathsf{type}$ *in ITT and* $\mathbf{1} \vdash a : [\![A]\!][p/\mathit{funext}, q/\mathit{uip}]$ *in ETT, then there exists a term* $\Gamma_{\mathrm{ax}} \vdash a' : A$ *in ITT.*

Corollary 4.3.19 is great news: although ITT is weaker than ETT, it is weaker by exactly two principles, namely function extensionality and uniqueness of identity proofs. We are led naturally to wonder whether there is a "best of both worlds":

Question 4.3.20. *Can we extend intensional type theory (with new terms and/or equations) in such a way that* Funext *and* UIP *are derivable, and the resulting type theory enjoys both canonicity* and *normalization?*

If we are satisfied with only *one of* canonicity or normalization, note that ETT is such an extension of ITT satisfying canonicity (Theorem 3.4.12) but not normalization (Section 3.6); on the other hand, extending ITT with axioms for Funext and UIP trivially makes these provable and satisfies normalization (Theorem 4.2.4) but not canonicity (Exercise 4.18).

Remark 4.3.21. Such tradeoffs are common in the design of type theory: canonicity says that a type theory has "enough" equations, whereas normalization generally requires that there not be "too many"; it can be hard to find the right balance. ◇

Type theorists have considered Question 4.3.20 since the 1990s, and there is some good news to report. If we are content for the moment to solve only the problem of UIP (ignoring Funext), there is in fact a rather modest extension of ITT that satisfies canonicity and normalization and in which UIP is provable.

For this, it will help us to consider an equivalent formulation of UIP due to Streicher [Str93] known as *Axiom K*:[9]

[9] In light of Remark 4.2.3, perhaps the reader can guess where the name *K* comes from.

$$\mathsf{K} = (A : \mathbf{U}) \to (a : A) \to (p : \mathbf{Id}(A, a, a)) \to \mathbf{Id}(\mathbf{Id}(A, a, a), p, \mathbf{refl})$$

It is easy to see that K follows from UIP, as it is the special case of UIP in which a and b are the same and one of the identity proofs is **refl**. The other direction of the biimplication is more subtle, but follows from a careful application of **j**, or identity elimination.

Exercise 4.21. Prove that K implies UIP in ITT.

As with subst and uniq, there is a sensible definitional equality with which to equip $\mathsf{k} : \mathsf{K}$, namely $\mathsf{k}\ A\ a\ \mathbf{refl} = \mathbf{refl}$, and we can even rephrase k as a "second elimination principle" of **Id**-types as follows:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash p : \mathbf{Id}(A, a, a) \qquad \Gamma.A.\mathbf{Id}(A[\mathbf{p}], \mathbf{q}, \mathbf{q}) \vdash B\,\mathsf{type} \qquad \Gamma.A \vdash b : B[\mathbf{id}.\mathbf{refl}]}{\Gamma \vdash \mathbf{k}(b, p) : B[\mathbf{id}.a.p]}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A.\mathbf{Id}(A[\mathbf{p}], \mathbf{q}, \mathbf{q}) \vdash B\,\mathsf{type} \qquad \Gamma.A \vdash b : B[\mathbf{id}.\mathbf{refl}]}{\Gamma \vdash \mathbf{k}(b, \mathbf{refl}) = b[\mathbf{id}.a] : B[\mathbf{id}.a.\mathbf{refl}]}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma \vdash p : \mathbf{Id}(A, a, a) \qquad \Gamma.A.\mathbf{Id}(A[\mathbf{p}], \mathbf{q}, \mathbf{q}) \vdash B\,\mathsf{type} \qquad \Gamma.A \vdash b : B[\mathbf{id}.\mathbf{refl}]}{\Delta \vdash \mathbf{k}(b, p)[\gamma] = \mathbf{k}(b[(\gamma \circ \mathbf{p}).\mathbf{q}], p[\gamma]) : B[\gamma.a[\gamma].p[\gamma]]}$$

It is instructive to compare the rules for **k** to those of **j**, whose motives

$$\Gamma.A.A[\mathbf{p}].\mathbf{Id}(A[\mathbf{p}^2], \mathbf{q}[\mathbf{p}], \mathbf{q}) \vdash C\,\mathsf{type}$$

quantify over *both* sides of the identification. Although **j** may seem superficially more general, neither **j** nor **k** imply the other. On the one hand, **k** is equivalent to UIP, which is independent of ITT; on the other hand, we needed the additional flexibility of **j** to define subst (Lemma 4.2.6), and we invite the reader to attempt this definition with **k** alone.

Although adding the above rules for **k** to intensional type theory breaks the pattern of inductive types we established in Section 2.5, the resulting theory continues to enjoy all the good properties of intensional type theory.

Theorem 4.3.22. *Intensional type theory plus the above rules for* **k** *satisfies consistency, canonicity, normalization, has invertible type constructors, and also validates* UIP.

In fact, K was originally introduced not to restore extensionality to ITT but in the study of *dependent pattern-matching*, where early formulations of pattern-matching for dependent type theory [Coq92] were found to derive K and were thus stronger than the standard rules of ITT. Although researchers have subsequently formulated a weaker notion of pattern-matching that does not derive K [CDP14], many proof assistants such as Agda still include K by default, often via pattern-matching.

Unfortunately it is significantly more challenging to add function extensionality to ITT in a satisfactory (canonicity-preserving) fashion, either in tandem with or independently of K/UIP. There are a number of type theories that admit function extensionality and satisfy all the relevant metatheorems, most notably *observational type theories* (Section 4.4, which also validate UIP) and *cubical type theories* (Chapter 5, which intentionally do not validate UIP), but these systems are quite a bit more complex than ITT and have not supplanted it.

Thus, despite its shortcomings, many practitioners choose to work in ITT extended by an axiom for function extensionality and either an axiom for UIP or a version of dependent pattern-matching that validates K.

4.4⋆ *Observational type theory* (DRAFT)

The structure of this book may lead the reader to believe that type theory is a linear progression of improvement from extensional (Chapter 2), to intensional (Chapter 4), to finally homotopy type theory (Chapter 5). This, however, is too simple a story to be accurate and the last step is particularly questionable: there are many type theories which aim to extend ITT in order to address Question 4.3.20 and no clear winner has yet emerged. It is impractical to give a full survey of such attempts, but we wished to briefly discuss a line of work in this vein before moving on: *observational type theory*. This work originates with Altenkirch, McBride, and Swierstra [AMS07],[10] but traces back to Hofmann and Streicher's groupoid model discussed in Section 4.3 as well as closely related models of type theory [Alt99]. Our discussion closely follows the recent line of work on OTT [PT22; PT23], especially its culmination by Pujet, Leray, and Tabareau [PLT25].

Warning 4.4.1. With this section, we have reached the frontier of research in type theory; the variant of OTT we exposit [PLT25] was published contemporaneously with this book itself. We therefore warn the reader that this material is commensurately rougher, though hopefully they will be all the more excited for it.

[10] See also Altenkirch and McBride [AM06]

Just as with any intensional type theory, OTT will provide an identity type supporting refl, subst, and uniq along with the elusive function extensionality and UIP principles. As ever, the main challenge is in OTT's equality rules which must maintain the normalization while ensuring canonicity. OTT solves this problem by restructuring the interface to the identity type so that no equation depends on the shape of a proof $p : \mathbf{Id}(A, a, b)$. Instead, the equations for **Id**-elimination will depend on A along with a and b. This is quite different than ITT where, for instance, the **j** was specified by describing its behavior on **refl**. After this restructuring however, we are free to (1) definitionally identify all elements of $\mathbf{Id}(A, a, b)$ (making UIP trivial) and (2) add a constant for function extensionality, all without endangering canonicity.

While this is easy to state, it is harder to build a type theory around this idea. After all, it is not clear what equation(s) should be used in place of the standard **j**-on-**refl** rule. As a starting point, after definitionally identifying all elements of the **Id**-type, uniq is a consequence of subst. Consequently, rather than dealing with the full generality of **j**, it suffices to handle subst. Moreover, subst itself can be decomposed into two more manageable special cases:

1. substitution specialized to identifications in the universe and,
2. substitution in families $B : A \to \mathbf{U}$ where each $B\,a$ contains at most one element up to definitional equality (*i.e.*, families of *propositions*—see Section 2.7).

While we can later use both these cases to recover subst, the equational theory for these cases is markedly simpler. Even with this, however, complications still occur. For instance, making sense of the second case of substitution (that of propositional families) is more difficult than it might appear on first blush. When we discussed propositions in Section 2.7, we were able to define a universe of (strict) propositions using the extensional equality. In our present intensional setting, however, we cannot give such a straightforward definition.[11] Consequently, OTT includes a special universe of strict propositions $\mathbf{\Omega}$ and a slew of rules managing it. All together, OTT extends our base type theory with quite a few new rules!

To keep this section manageable, we therefore only scratch the surface of OTT. Rather than attempting to give a rigorous and systematic definition of OTT, we will instead adopt the conventions of Section 4.1 and work with informal type theory once more. We will specify the novel connectives and rules of OTT as a sequence of constants and equations in informal type theory, emphasizing more how they fit together than the precise rules which define them. Moreover, we deal with only the core extensions of OTT (the observational identity type $\sim$ and a universe of strict

[11] Replicating the definition from Section 2.7 gives rise instead to the universe of so-called *homotopy propositions*, the subject of Section 5.1.

propositions Ω) as well as their consequences for a few types. The reader who finds their curiosity piqued should be able to easily dive into Pujet and Tabareau [PT22] and the subsequent work [PT23; PLT25] after completing this section.

4.4.1 *The basic interface for an observational identity type*

We begin by describing the core extensions OTT makes to type theory. Accordingly, we shall consider type theory as it stood at the beginning of this chapter: without either **Id** or **Eq**. To head off potential confusion, we use an alternative notation for OTT's variant of the identity type: if $A : \mathbf{U}$ and $a, b : A$, then $a \sim_A b$ is the *observational identity type*. Our goal in this section is to highlight the three operations (refl, cast, and dcast) which comprise the core interface for $\sim$. Even before discussing these operations, however, we can point out one way in which OTT deviates from ITT: in OTT all elements of $a \sim_A b$ are definitionally equal *i.e.*, if $p, q : a \sim_A b$ then $p = q$.

The most standard is refl, witnessing the reflexivity of $\sim$:

$$\mathsf{refl} : \{A : \mathbf{U}\} \to (a : A) \to a \sim_A a$$

Taking refl along with our earlier assertion that if $p, q : a \sim_A b$ then $p = q$, we now see that UIP holds for $\sim_A$.

We now turn to the second operation OTT provides for manipulating $a \sim_A b$, a limited version of subst:

$$\mathsf{cast} : \{A, B : \mathbf{U}\} \to A \sim_{\mathbf{U}} B \to A \to B$$

Many definitional equations govern cast and we shall meet some in Section 4.4.2. For now, we note only the most fundamental: casting from a type to itself is the identity:

$$\mathsf{cast}\ \{A\}\ \{A\}\ p\ a = a$$

In fact, because $A \sim_{\mathbf{U}} A$ is a strict proposition, this rule is equivalent to $\mathsf{cast}\ (\mathsf{refl}\ A)\ a = a$. This is a glimpse of the general phenomenon that equations dealing with strict propositions can be quite difficult to control and it foreshadows Section 4.4.2.

The story for $\sim_A$ cannot stop here if we are to recover the full API for identity types. For instance, if we are to prove the symmetry of $\sim_A$, we must be able to substitute in the type family $a \sim_A -$. Given that cast was specifically chosen to avoid the complexity of substituting in arbitrary dependent types, adding more general substitution operations seems rather counterproductive. Fortunately, however, the same feature of OTT which complicates subst in general makes substituting within $a \sim_A -$ simpler: $a \sim_A b$ is a proposition. After all, if we are substituting in the dependent type $B(a') = a \sim_A a'$

then computation rules for this substitution operation are irrelevant: every element of $a \sim_A a'$ is equal to every other element and so every possible rule we could think to add is redundant! This is the key insight behind the final OTT primitive for manipulating $\sim_A$: a version of subst which allows for substitution in families of strict propositions.

Given $B : A \to \mathbf{U}$, however, we cannot state internally the condition "for every $a : A$, $B(a)$ is a strict proposition". In order to express the aforementioned special case of subst, OTT includes a special universe of strict propositions:

$$\begin{aligned}&\Omega : \mathbf{U}\\ &\mathbf{Prf} : \Omega \to \mathbf{U}\end{aligned}$$

Crucially, elements $\phi : \Omega$ are codes for strict propositions so that if $\phi : \Omega$ then for all $z, z' : \mathbf{Prf}(\phi)$, we have $z = z'$. We warn the reader that there are two ways of reading the above declarations if we mind universe levels: either there exists a single constant $\Omega : \mathbf{U}_0$ or for each i there exists $\Omega_i : \mathbf{U}_{i+1}$. Either reading is permissible (c.f., Remark 2.7.14) and the difference is not important at our level of discussion. Finally, we populate Ω with a handful of codes for important propositions:

$$\begin{aligned}&\sim : (A : \mathbf{U}) \to A \to A \to \Omega\\ &\mathbf{\Pi} : (A : \mathbf{U}) \to (A \to \Omega) \to \Omega\\ &\Sigma : (\phi : \Omega) \to (\mathbf{Prf}(\phi) \to \Omega) \to \Omega\end{aligned}$$

We require that these codes are mapped by **Prf** to the expected types *i.e.*, $\mathbf{Prf}(\mathbf{\Pi}(c, \phi)) = (a : \mathbf{El}(c)) \to \mathbf{Prf}(\phi(a))$. Note that we do not require the domain of $\mathbf{\Pi}$ to land in Ω, while this is necessary for Σ (c.f., Section 2.7.2). Furthermore, in keeping with our informal notation we generally suppress **Prf**.

With this in place, we can state the final operation needed for $\sim$:

$$\mathsf{dcast} : \{A : \mathbf{U}\}\{a_0, a_1 : A\} \to (B : A \to \Omega) \to a_0 \sim_A a_1 \to B(a_0) \to B(a_1)$$

As mentioned earlier, no special equality rules are needed to govern dcast; we already have $\mathsf{dcast}\, B\, \mathsf{refl}\, b = b$ (along with any other imaginable equation) by virtue of the special properties of Ω.

To tie off this discussion, we now show that these three operations together allow us to recover the expected API for the intensional identity type. Reflexivity holds by virtue of refl, so it remains only to show that cast and dcast can define **j**.

Lemma 4.4.2. *There is a definable operation j in OTT with the following type:*

$$\begin{aligned}j : \{A : \mathbf{U}\} &\to (B : (a_0, a_1 : A) \to (p : a_0 \sim_A a_1) \to \mathbf{U})\\ &\to ((a : A) \to B\, a\, a\, \mathsf{refl})\\ &\to (a_0, a_1 : A) \to (p : a_0 \sim_A a_1) \to B\, a_0\, a_1\, p\end{aligned}$$

Moreover, there is a definitional equality $j\ B\ b\ a\ a\ \mathsf{refl} = b\ a$.

Proof. To begin with, let us fix $A : \mathbf{U}$, $B : (a_0, a_1 : A) \to (p : a_0 \sim_A a_1) \to \mathbf{U}$, and $b : (a : A) \to B\ a\ a\ \mathsf{refl}$.

Our first and main goal is to construct a function $(a_0\ a_1 : A)(p : a_0 \sim_A a_1) \to B\ a_0\ a_1\ p$. We therefore consider $a_0, a_1 : A$ and $p : a_0 \sim_A a_1$. First, using dcast with the motive $\lambda a_0\ a_1\ p \to B\ a_0\ a_0\ \mathsf{refl} \sim_{\mathbf{U}} B\ a_0\ a_1\ p$ along with p, we obtain a proof $q : B\ a_0\ a_0\ \mathsf{refl} \sim_{\mathbf{U}} B\ a_0\ a_1\ p$. Next, we use cast $q\ (b\ a_0)$ to finish the construction.

To see that this satisfies the required definitional equality, let us suppose we are given $a : A$ and try to show that cast $(\dots)\ (b\ a)$ where $(\dots)$ is our construction of an element $B\ a\ a\ \mathsf{refl} \sim_{\mathbf{U}} B\ a\ a\ \mathsf{refl}$. However, by definitional proof irrelevance, we may suppose that $(\dots)$ is actually refl and so the required definitional equality is a corollary of the more general equality that cast on refl is the identity. □

Exercise 4.22. Use the fact that $a \sim_A b$ is a strict proposition to give a direct definition of uniq (c.f., Section 4.1.2):

$$[-] : \{A : \mathbf{U}\} \to (a : A) \to \mathbf{U}$$
$$[a] = \textstyle\sum_{b:A} a \sim_A b$$

$$\mathsf{uniq} : \{A : \mathbf{U}\} \to (a : A) \to (x : [a]) \to (a, \mathsf{refl}\ a) \sim_{[a]} x$$

4.4.2 *Integrating function extensionality*

Taking stock at this point, we have discussed $a \sim_A b$ along with its critical operations (refl, cast, and dcast) and the attendant universe of strict propositions $\mathbf{\Omega}$. We have seen that, together, this suffices to implement our standard API for intensional identity type and that, by virtue of $a \sim_A b$ landing in $\mathbf{\Omega}$, we know that UIP holds for the observational identity type. What about function extensionality? As it stands, function extensionality is not derivable and the remaining suite of rules for OTT are designed to rectify this. In fact, the next extension we consider accomplishes this directly:

$$\mathsf{funext} : \{A : \mathbf{U}\}\{B : A \to \mathbf{U}\} \to (f, g : (a : A) \to B(a)) \to$$
$$((a : A) \to f\ a \sim_{B\ a} g\ a) \to f \sim_{(a:A)\to B(a)} g$$

Clearly, with this in hand we can prove function extensionality. The problem is the familiar issue from Section 4.3—adding this alone will disrupt canonicity.

Unlike with ITT, where we could use $\mathbf{j}$ to immediately produce a stuck term, it is a little non-obvious why funext will cause problems if added alone to OTT and so we take a moment to spell out the details of an example. Suppose we use funext to construct

$p : f \sim_{\mathbf{Nat}\to\mathbf{Nat}} g$ where f and g are not definitionally equal. Using dcast, we may then construct an identification $q : (\sum_{h:\mathbf{Nat}\to\mathbf{Nat}} f \sim_{\mathbf{Nat}\to\mathbf{Nat}} h) \sim_{\mathbf{U}} (\sum_{h:\mathbf{Nat}\to\mathbf{Nat}} g \sim_{\mathbf{Nat}\to\mathbf{Nat}} h)$. These two elements of the universe are not definitionally equal and so there is no rule which will allow us to reduce cast $q\,(f, \mathsf{refl})$. One thing to observe about this situation: there is an obvious candidate for what cast $q\,(f, \mathsf{refl})$ ought to reduce to: (f, p).

Pondering this example leads to the hypothesis that while we may use funext to identify types which are not definitionally equal, we cannot use it to identify types with distinct "outermost constructors"—at least, not in a closed context. In other words, we cannot prove $((a : A) \to B\,a) \sim_{\mathbf{U}} \sum_{c:C} D\,c$, only that $\sum_{a:A} B\,a \sim_{\mathbf{U}} \sum_{c:C} D\,c$. Moreover, in the cases where $p : \sum_{a:A} B\,a \sim_{\mathbf{U}} \sum_{c:C} D\,c$, it seems that these identifications will come from pairs of identifications $p_0 : A \sim_{\mathbf{U}} C$ and $p_1 : (a : A) \to B\,a \sim_{\mathbf{U}} D(\mathsf{cast}\,p_0\,a)$. If we write $\mathsf{cong}_\Sigma(p_0, p_1)$ for the identification $\sum_{a:A} B\,a \sim_{\mathbf{U}} \sum_{c:C} D\,c$ built from the identifications $p_0 : A \sim_{\mathbf{U}} C$ and $p_1 : (a : A) \to B\,a \sim_{\mathbf{U}} D(\mathsf{cast}\,p_0\,a)$, our earlier candidate for reducing cast leads us to the following:

$$\mathsf{cast}\,(\mathsf{cong}_\Sigma(p_0, p_1))\,(a, b) = (\mathsf{cast}\,p_0\,a, \mathsf{cast}\,(p_1\,a)\,b) \qquad (\text{✎})$$

In the case where all identifications involved are reflexivity, this is the ordinary η-rule for Σ-types and, more promisingly, this equation will handle our earlier example of an obstruction to canonicity. If we add further congruence type rules for cast allowing it to compute past $\mathbf{\Pi}$ and other connectives, in fact, we could prove canonicity even in the presence of function extensionality.

A final wrinkle remains: we have added these rules to ensure that canonicity holds for this theory, but what about normalization? Unfortunately, adding these rules as-is will make it impossible to implement a type-checking procedure for this theory. In light of the fact that $\sim$ is a strict proposition, the above equation really says that $\mathsf{cast}\,p\,(a, b) = (\mathsf{cast}\,q\,a, \mathsf{cast}\,r\,b)$, with no given relation between p, q, and r. A normalization algorithm would therefore have to infer whether q and r existed only knowing that p exists—an undecidable question in general. To resolve this, OTT modifies the theory to make the above equality *always* apply by making Σ propositionally injective:

$$\begin{aligned}
&\mathsf{inj}^0_\Sigma : \{A, C : \mathbf{U}\}\{B : A \to \mathbf{U}\}\{D : C \to \mathbf{U}\} \to \textstyle\sum_{a:A} B\,a \sim_{\mathbf{U}} \sum_{c:C} D\,c \to A \sim_{\mathbf{U}} C\\
&\mathsf{inj}^1_\Sigma : \{A, C : \mathbf{U}\}\{B : A \to \mathbf{U}\}\{D : C \to \mathbf{U}\} \to\\
&\quad (p : \textstyle\sum_{a:A} B\,a \sim_{\mathbf{U}} \sum_{c:C} D\,c) \to (a : A) \to B\,a \sim_{\mathbf{U}} D\,(\mathsf{cast}\,(\mathsf{inj}^0_\Sigma\,p)\,a)\\
&\mathsf{cast}\,p\,(a, b) = (\mathsf{cast}\,(\mathsf{inj}^0_\Sigma\,p)\,a, \mathsf{cast}\,(\mathsf{inj}^1_\Sigma\,p\,a)\,b)
\end{aligned}$$

Each connective we wish to include in OTT needs such injectivity operations and a suitable rule for cast. For another example, consider $\mathbf{\Pi}$:

$$\mathsf{inj}^0_{\mathbf{\Pi}} : \{A, C : \mathbf{U}\}\{B : A \to \mathbf{U}\}\{D : C \to \mathbf{U}\} \to$$

$$
\begin{aligned}
&((a : A) \to B\,a) \sim_{\mathbf{U}} ((c : C) \to D\,c) \to A \sim_{\mathbf{U}} C \\
&\mathsf{inj}^1_{\mathbf{\Pi}} : \{A, C : \mathbf{U}\}\{B : A \to \mathbf{U}\}\{D : C \to \mathbf{U}\} \to \\
&\quad (p : ((a : A) \to B\,a) \sim_{\mathbf{U}} ((c : C) \to D\,c)) \to (a : A) \to \\
&\qquad B\,a \sim_{\mathbf{U}} D\,(\mathsf{cast}\,(\mathsf{inj}^0_{\mathbf{\Pi}}\,p)\,a) \\
&\mathsf{cast}\,p\,f = \lambda c \to \mathbf{let}\ a = \mathsf{cast}\,(\mathsf{sym}\,(\mathsf{inj}^0_{\mathbf{\Pi}}\,p))\,c\ \mathbf{in}\ \mathsf{cast}\,(\mathsf{inj}^1_{\mathbf{\Pi}}\,p\,a)\,(f\,a)
\end{aligned}
$$

After coagulating the above discussion into a formal type theory, Pujet, Leray, and Tabareau [PLT25] have proven the following:

Theorem 4.4.3. *OTT provides an affirmative answer to Question 4.3.20.*

In fact, they have proven a stronger result: OTT extended with any constant $a : A$ where $A : \mathbf{\Omega}$ continues to satisfy canonicity provided that A does not imply $A \sim_{\mathbf{U}} B$ where A and B have distinct "outermost constructors" in an empty context. This property also allows OTT to support other extensionality-type principles, including the propositional univalence, which we will meet in Section 5.1.

4.4.3 *Final observations on $\sim_A$*

The reader may have noticed that our treatment of OTT was substantially different from ETT and ITT, where each connective was introduced with careful justification and, vitally, a mapping-in or mapping-out property. More crisply, how does $\sim_A$ fit into Slogan 2.5.5? Simply put, it does not! In fact, OTT and $\sim_A$ were not inspired by Slogan 2.5.5, but by an attempt to capture a particular model of type theory.

To see this more clearly, we remind the reader that OTT originates from work on the setoid model of type theory [Alt99]. The basic object of this model, a *setoid*, consists of a set X together with an equivalence relation $\sim_X \subseteq X \times X$. These were popularized by Bishop [Bis67] in the context of constructive analysis, but proved useful in giving models of type theory in comparatively weak metatheories (such as intensional type theory itself). Roughly, the intensional identity type of a type $[\![A]\!] = (X, \sim_X)$ was defined using $\sim_X$ and, consequently, the setoid model validated both function extensionality and UIP, even if the metatheory (*e.g.*, ITT) did not.[12]

OTT attempts to consolidate some of the remarkable properties of this setoid model into a new variant of type theory. In its first iterations especially, this *intended model* in setoids was emphasized by having $\sim_A$ "compute" to different types depending on

[12] Setting aside the question of metatheory, the setoid model is strikingly similar to the groupoid model introduced in Section 4.3.2. More precisely, setoids are groupoids without non-trivial automorphisms and the setoid model is given by restricting the groupoid model along this identification.

A, mirroring how the setoid model realizes equality on $(X, \sim_X)$ by $\sim_X$. For instance, some versions of OTT feature rules which thereby trivialize Funext and UIP:

$$f \sim_{(a:A)\to B\,a} g = ((a : A) \to f\ a \sim_{B\,a} g\ a) \qquad p \sim_{a \sim_A b} q = \mathbf{Unit}$$

The observational equality type is therefore a distillation of how $\mathbf{Id}_A$ is interpreted by the setoid model and so $\sim_A$ ought to internalize some judgmental structure stating that each type is a setoid. As it stands, however, no such judgmental structure exists in OTT and so $\sim_A$ has rather idiosyncratic rules. Without a fully defined specification for $\sim_A$, the rules were tweaked and minimized over time and so the connection to the setoid model became less visible.

It is from the setoid model, however, that the most peculiar aspects of OTT ($\mathsf{inj}^{\epsilon}_{\mathbf{\Pi}}$ and $\mathsf{inj}^{\epsilon}_{\Sigma}$) are drawn. In particular, the universe $(U, \sim_U)$ in the setoid model is quite different than we encountered in Section 3.5. The set U is given by an inductive-recursive definition [Dyb00] and so not only is $\mathbf{\Pi}$ injective in this model, there is an induction principle associated with U which allows one to pattern-match on codes in the universe. This induction principle was used to define $\sim_U$ and the injectivity operations included now in OTT are trivial corollaries of this definition.

Remark 4.4.4. Another natural question may have occurred to the reader: given the additional properties OTT imposes on the universe, does it still support familiar models from ETT? Indeed, important models of ETT do not admit an interpretation of $\mathsf{inj}^{\epsilon}_{\mathbf{\Pi}}$. For instance, the reader may find it interesting to prove that the set model does not interpret either $\mathsf{inj}^{\epsilon}_{\mathbf{\Pi}}$ or $\mathsf{inj}^{\epsilon}_{\Sigma}$. Fortunately, Pujet and Tabareau [PT23] and Gratzer [Gra22] have shown that any model of type theory with a hierarchy of universes without injective codes can be modified to produce a model with injective codes and so a model of OTT is essentially the same as a model of ETT. ◇

The state of affairs described above for OTT is quite common for cutting-edge type theories. While type theorists often design a new type theory with inspiration from a particular model, passing from a model to a new type theory is more art than science. Indeed, even for just the setoid model, other type theories have been proposed as attempts to better capture the setoid model [Alt+19]. We will encounter another example of this process in Section 5.3 with cubical type theory. More broadly all of univalent type theories are examples of this process.

Further reading

We have mentioned previously that proof assistants decide equality of terms using a type-sensitive algorithm known as normalization by evaluation (NbE). Proofs of the normalization metatheorem for intensional type theory proceed by establishing that NbE is sound and complete for the equational theory of ITT, using a proof technique known as Kripke logical relations. There are many papers dedicated to proving normalization for variants of ITT; Abel [Abe13] includes a lengthy exposition starting with the non-dependent case, Abel, Öhman, and Vezzosi [AÖV17] formalize their proof in Agda, and Coquand [Coq19] and Sterling [Ste21] present semantic formulations of NbE that are significantly less technical but require more mathematical sophistication.

As for the independence and conservativity theorems discussed in Section 4.3, the theses of Streicher [Str93] and Hofmann [Hof95a] remain excellent references; however, a more modern account is available in the thesis of Winterhalter [Win20]. Furthermore, recent advances in semantics have enabled much shorter albeit sophisticated proofs of conservativity results (such as Theorem 4.3.18) [KL25].

The independence of function extensionality from ITT has led to a cottage industry of observational type theories as discussed in Section 4.4; the authors are biased but recommend Sterling, Angiuli, and Gratzer [SAG22, Section 1] for a brief history of equality in type theory. On the other hand, the independence of UIP has spawned an entire *subdiscipline*, homotopy type theory (Chapter 5). Models of homotopy type theory, such as Voevodsky's simplicial model [KL21], can be seen as vast generalizations of the groupoid model of Hofmann and Streicher [HS98].

This material will be published by Cambridge University Press as *Principles of Dependent Type Theory* by Carlo Angiuli and Daniel Gratzer. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © Carlo Angiuli and Daniel Gratzer 2024

5 *Univalent type theories*

In Chapter 4 we saw that replacing **Eq**-types with **Id**-types allows type theory to enjoy normalization and other properties needed for practical implementation, at the cost of losing two of extensional type theory's reasoning principles: function extensionality and uniqueness of identity proofs (UIP). We then considered how one may restore these principles to type theory, both with and without sacrificing canonicity.

In this chapter we consider a radically different extension of intensional type theory known as *homotopy type theory* (HoTT). Rather than attempting to restore UIP to type theory, homotopy type theory extends ITT with a reasoning principle known as univalence, which in fact *refutes* UIP in a vast generalization of the groupoid countermodel discussed in Section 4.3.2. Univalence is a rather subtle principle that is difficult to even state properly, so we will begin by considering univalence for propositions before moving on to the full univalence principle and its many consequences.

Univalence is traditionally stated as an axiom which breaks the canonicity property of intensional type theory. As a result, although its repercussions are vastly different from those of UIP, it has the same drawbacks from the perspective of implementation. In the second half of this chapter we turn our attention to *cubical type theory*, which extends the judgmental structure of type theory and reimagines the **Id**-types of homotopy type theory as defined by a mapping in property. The result is a type theory that admits the principles of univalence and function extensionality while simultaneously enjoying the properties of canonicity, normalization, and decidability of type-checking.

This chapter brings the reader up to present-day research in type theory. The univalence principle was first proposed in 2010 by Voevodsky [Voe10], and much of our current understanding of its consequences was developed in 2013 in the book *Homotopy Type Theory: Univalent Foundations of Mathematics* [UF13]. The first cubical type theories emerged around 2016 [CCHM18; Ang+21], and their first normalization proof was given in 2021 [SA21]. Univalent type theories remain far from settled; at the time of writing, researchers continue to grapple with the semantics and consequences of univalence, and are even developing successors to cubical type theory [Shu22].

In this chapter In Section 5.1 we revisit universes of propositions (Section 2.7) in the setting of intensional type theory, and define the axioms of propositional univalence and resizing. In Section 5.2 we formulate the full univalence axiom and give a whirlwind tour of homotopy type theory, including homotopy levels, higher inductive types, and

some applications and consequences of univalence. In Section 5.3 we introduce the novel judgmental structure of cubical type theory and use it to define a "mapping in" variation of intensional identity types known as **Path**-types. Finally, in Section 5.4, we explain how the cubical apparatus allows us to define a univalent type theory which enjoys the canonicity property.

Goals of the chapter By the end of this chapter, you will be able to:

- Define universes of propositions in intensional type theory, and state propositional univalence and propositional resizing.
- State the univalence axiom, including the notion of equivalence.
- Explain several core concepts of homotopy type theory, including homotopy levels and higher inductive types.
- Discuss the goals of cubical type theory, and explain how the interval, coercion, and composition address these goals.

5.1 *Propositional univalence*

In Section 4.3 we identified two principles of ETT absent from ITT, function extensionality and UIP, which characterize the identity types of **Π**-types and **Id**-types respectively. One can understand the univalence principle (Section 5.2.1) as yet another absent principle which characterizes the identity type of $\mathbf{U}_i$, but univalence is significantly harder to motivate because it is not a principle of ETT—in fact, it contradicts UIP—and because type universes are already the most complex connective of type theory.

Before presenting the full univalence principle in Section 5.2 we will warm up with a discussion of *propositional univalence*, the univalence principle restricted to universes of propositions Prop_i (Section 2.7). Propositional univalence simplifies full univalence in several ways: it is easier to state, is consistent with UIP, and is in a certain sense validated by the set model (Section 3.5). On top of these pedagogical advantages, propositional univalence is an important reasoning principle in its own right, especially when coupled with the related principle of *propositional resizing*.

Notation 5.1.1. Throughout this section we return to the informal notation for intensional type theory used in Chapter 1 and Section 4.1.

Axiom 5.1.2. In order to avoid annoying technicalities, we work in ITT extended by the function extensionality axiom funext : Funext as defined in Section 4.3.

5.1.1 *Homotopy propositions*

Before stating propositional univalence, we must adapt the notion of *proposition* introduced in Section 2.7 from ETT to ITT. Recall that propositions are "types whose terms are all equal" or "types with at most one element." More formally, we said that a type A is a proposition in ETT if one of these equivalent conditions holds:

1. any two terms $a, b : A$ are judgmentally equal (Definition 2.7.5), or
2. the type $\mathsf{isProp}(A) :\equiv (a\ b : A) \to \mathbf{Eq}(A, a, b)$ is inhabited (Exercise 2.37).

In intensional type theory we must replace the **Eq**-type in (2) with an **Id**-type, at which point it becomes clear that these conditions are no longer equivalent because **Id**-types—by design!—lack equality reflection.

2′. the type $\mathsf{IsHProp}(A) :\equiv (a\ b : A) \to \mathbf{Id}(A, a, b)$ is inhabited.

As a result, ITT has two natural notions of proposition: (1) types with one term up to definitional equality, known as *strict propositions*, and (2′) types with one term up to identification, known as *homotopy propositions*.[1]

Both notions have their advantages and disadvantages. Strict propositions are more convenient because definitional equality is silent, avoiding the need for subst casts; unfortunately, very few types are strict propositions in ITT. (Not even **Void** is a strict proposition, due to its lack of η-rule!) In contrast, homotopy propositions are less convenient but much more common.

Exercise 5.1. Argue that every strict proposition is a homotopy proposition.

Exercise 5.2. Show that $\mathsf{IsHProp}(\mathbf{Void})$ is inhabited.

Another important advantage of homotopy propositions is that the property of being a homotopy proposition can be stated internally ($\mathsf{IsHProp}_i : \mathbf{U}_i \to \mathbf{U}_i$), so we can define a hierarchy of universes of propositions HProp_i as the subtypes of each $\mathbf{U}_i$ spanned by homotopy propositions, as in Section 2.7.1:

$$\mathsf{HProp}_i : \mathbf{U}_{i+1}$$
$$\mathsf{HProp}_i :\equiv \textstyle\sum_{A:\mathbf{U}_i} \mathsf{IsHProp}_i(A)$$

For these reasons, and because they play a central role in Section 5.2, we will focus exclusively on homotopy propositions in the remainder of this book.

[1] There is nothing yet intrinsically "homotopical" about homotopy propositions. We will soon see that they are part of a more comprehensive taxonomy of types known as *homotopy levels* (Section 5.2.2).

Notation 5.1.3. Mirroring our notation for Prop_i, we will suppress the indices on HProp_i and $\mathsf{IsHProp}_i$ when they are immaterial. We will also collapse the distinction between elements of HProp and types $A : \mathbf{U}$ for which $\mathsf{IsHProp}(A)$ is inhabited, by treating the projection $\mathsf{HProp} \to \mathbf{U}$ as silent and writing $A : \mathsf{HProp}$ when $A : \mathbf{U}$ and A is a homotopy proposition. (In Exercise 5.12 we will see that it is safe to suppress the choice of proof $p : \mathsf{IsHProp}(A)$ because $\mathsf{IsHProp}(A)$ is itself a homotopy proposition.)

Homotopy propositions in ITT are closed under many of the same connectives as propositions in ETT (Corollary 2.7.12) with a few notable exceptions. In particular, $\mathbf{Id}(A, a, b)$ is not in general a homotopy proposition. (Uniqueness of identity proofs is precisely the statement that all identity types *are* propositions!) More frustratingly, one needs function extensionality to show that propositions are closed under $\mathbf{\Pi}$-types.

Exercise 5.3. Show that if $B : A \to \mathsf{HProp}$ then $(a : A) \to B(a) : \mathsf{HProp}$.

5.1.2 *Propositional univalence*

When are two homotopy propositions—henceforth just "propositions"—identified in intensional type theory? Unfortunately not very often, given that HProp is defined in terms of $\mathbf{U}$, which is in turn defined as a collection of codes for (i.e., names of) types.

For the third time, we have found a type of identifications lacking. In Section 4.3.1, we saw that $\mathbf{Id}(\mathbf{\Pi}(A, B), f, g)$ is underspecified in ITT and proposed that, as in ETT, such identifications should be given by pointwise identifications of $f(a)$ and $g(a)$ for all $a : A$. In Section 4.3.2, we saw that $\mathbf{Id}(\mathbf{Id}(A, a, b), p, q)$ is underspecified in ITT and proposed that, as in ETT, this type should always be inhabited. This time, it is not so clear what $\mathbf{Id}(\mathsf{HProp}, A, B)$ should or even could be. We cannot look to ETT for inspiration, because the ETT type $\mathbf{Eq}(\mathsf{Prop}, A, B)$ is just as underspecified!

After some thought, we notice that from any identification $\mathbf{Id}(\mathsf{HProp}, A, B)$ we can obtain (by subst and symmetry) a pair of functions $A \to B$ and $B \to A$, which is to say that $A \iff B$. Our desired characterization of $\mathbf{Id}(\mathsf{HProp}, A, B)$ should therefore imply, or at least be compatible with, the propositions A and B being interprovable.

We further note that the converse implication holds in an analogous scenario in set theory. Suppose we have two propositions (predicates) ϕ, ψ over a set X. These induce a pair of subsets of X, $\{x \in X \mid \phi(x)\}$ and $\{x \in X \mid \psi(x)\}$, which by the extensionality axiom of set theory are equal *if and only if* the predicates ϕ and ψ are interprovable.

Emboldened by these observations, we introduce the principle of propositional univalence: interprovable propositions are identified.

$_\Leftrightarrow_ : \mathsf{HProp} \to \mathsf{HProp} \to \mathsf{HProp}$

$$A \Leftrightarrow B = (A \to B) \times (B \to A)$$

$$\begin{aligned}&\mathsf{HPropIsUnivalent} : \mathbf{U}\\&\mathsf{HPropIsUnivalent} = (A\ B : \mathsf{HProp}) \to (A \Leftrightarrow B) \to \mathbf{Id}(\mathsf{HProp}, A, B)\end{aligned}$$

Exercise 5.4. Show that if $A, B : \mathsf{HProp}$ then $(A \Leftrightarrow B) : \mathsf{HProp}$.

Warning 5.1.4. Although HPropIsUnivalent is a correct statement of propositional univalence, in Section 5.1.3 we will propose a more abstract formulation which—unlike the above statement—is validated by the set model of type theory (Theorem 5.1.8).

The axiom HPropIsUnivalent is quite a bit stronger than it may first appear.

Theorem 5.1.5. *For any propositions A and B,* HPropIsUnivalent $A\ B$ *is an isomorphism between* $(A \Leftrightarrow B)$ *and* $\mathbf{Id}(\mathsf{HProp}, A, B)$.

Proof. That is, we must define a map $\mathsf{inv} : \mathbf{Id}(\mathsf{HProp}, A, B) \to (A \Leftrightarrow B)$ such that both round trips cancel up to identification. The map itself is a direct consequence of subst: $\mathsf{inv}\ p \coloneqq (\mathsf{subst}\ \mathsf{id}\ p, \mathsf{subst}\ \mathsf{id}\ (\mathsf{sym}\ p))$. The first round trip

$$(x : A \Leftrightarrow B) \to \mathbf{Id}(A \Leftrightarrow B, \mathsf{inv}\ (\mathsf{HPropIsUnivalent}\ A\ B\ x), x)$$

is immediate from Exercise 5.4. For the second round trip

$$(p : \mathbf{Id}(\mathsf{HProp}, A, B)) \to \mathbf{Id}(\mathbf{Id}(\mathsf{HProp}, A, B), \mathsf{HPropIsUnivalent}\ A\ B\ (\mathsf{inv}\ p), p)$$

we note that the composite

$$i \coloneqq (\mathsf{HPropIsUnivalent}\ A\ B) \circ \mathsf{inv} : \mathbf{Id}(\mathsf{HProp}, A, B) \to \mathbf{Id}(\mathsf{HProp}, A, B)$$

is *idempotent* in the sense that (by the first round trip) there is an identification between $i \circ i$ and i. We complete the proof in Exercise 5.5 by showing that all idempotent maps $\mathbf{Id}(A, a, b) \to \mathbf{Id}(A, a, b)$ are the identity function. □

Exercise 5.5. Show the following result due to Escardó [Esc14]: if $i : \{a\ b : A\} \to \mathbf{Id}(A, a, b) \to \mathbf{Id}(A, a, b)$ is idempotent then it is, up to identification, the identity function. (Hint: start by identifying $i\ p$ and trans $(i\ \mathbf{refl})\ p$ for all p using identity elimination, then identify trans $(i\ \mathbf{refl})\ (i\ \mathbf{refl})$ and $(i\ \mathbf{refl})$.)

Corollary 5.1.6. *Propositional univalence holds if and only if the canonical map* $\mathbf{Id}(\mathsf{HProp}, A, B) \to (A \Leftrightarrow B)$ *induced by* subst *is an isomorphism.*

Exercise 5.6. Show that the following type is inhabited if and only if propositional univalence holds:

$$\{A\ B : \mathsf{Prop}\} \to (P : \mathsf{Prop} \to \mathbf{U}) \to (A \Leftrightarrow B) \to P(A) \to P(B)$$

Given that our statement of propositional univalence took inspiration from set theory, one might wonder whether it holds in the set model of ITT (Exercise 4.15); if it did, we would conclude not only that HPropIsUnivalent is consistent but also that it is compatible with UIP. Sadly this is not the case given how we have defined HProp.

Lemma 5.1.7. *The set model of ITT interprets* HPropIsUnivalent *as the empty set.*

Proof. Recall that we construct the set model of ITT by interpreting ITT into ETT (Corollary 4.3.3) and further interpreting ETT into sets (Section 3.5). The first interpretation simply translates **Id**-types into **Eq**-types:

$$[\![\mathsf{HPropIsUnivalent}]\!] = (A\ B : \mathsf{Prop}) \to (A \Leftrightarrow B) \to \mathbf{Eq}(\mathsf{Prop}, A, B)$$

It remains to show that $\mathsf{Tm}_{\mathcal{S}}(\mathbf{1}_{\mathcal{S}}, [\![\mathsf{HPropIsUnivalent}]\!]_{\mathcal{S}}) = \emptyset$ where $\mathcal{S}$ is the set model of ETT. Unfolding the $\mathcal{S}$-interpretations of **Π**-types, **Eq**-types, and **U**-types, it suffices to exhibit a pair of elements $A, B \in \mathsf{Prop}_{\mathcal{S}} = \{X \in \mathcal{V}_0 \mid \forall x, y \in X.\ x = y\}$ such that the following set is empty:

$$((A \to B) \times (B \to A)) \to \{\star \mid A = B\}$$

In other words, we must find a pair of subsingleton sets $A \neq B$ for which there are functions $A \to B$ and $B \to A$. For this we can exhibit any pair of unequal one-element sets, such as $A :\equiv \{\star\}$ and $B :\equiv \{\{\star\}\}$. □

Zooming out, because HProp is defined as the subtype of **U** spanned by types with at most one element, it is interpreted in the set model as the collection of all sets with at most one element—and certainly not all one-element sets are equal! We note that there *are* set-theoretic representations of "the set of all propositions" that are propositionally univalent, most notably the two-element set $\{\top, \bot\}$. However, in order to formally connect this set to propositional univalence, we must come up with a more abstract notion of a "univalent universe of propositions" for which it is a valid model.

5.1.3 *Abstracting propositional univalence*

Recall from Notation 2.6.4 that a "universe" is any pair of a type U with a function sending elements of U to types. Following this logic, a "universe of propositions" should be any type Ω equipped with a decoding function $f : \Omega \to \mathbf{U}$ subject to the additional condition that $f(x)$ is a proposition for every $x : \Omega$. If we bundle the latter two conditions into a single map $\mathrm{dec} : \Omega \to \mathsf{HProp}$ and add universe levels, we conclude that *abstract universes of (*$\mathbf{U}_i$*-small) propositions* are elements of the **Σ**-type

$$\mathsf{PropUniverse}_i = \textstyle\sum_{\Omega:\mathbf{U}_{i+1}} (\Omega \to \mathsf{HProp}_i)$$

The "concrete" universe of propositions HProp_i is of course one such universe, when paired with the trivial decoding $\mathsf{id} : \mathsf{HProp}_i \to \mathsf{HProp}_i$.

The statement of propositional univalence for HProp, HPropIsUnivalent, immediately generalizes to abstract universes of propositions.

$$\begin{aligned}&\mathsf{IsUnivalent}_i : \mathsf{PropUniverse}_i \to \mathbf{U}_{i+1}\\ &\mathsf{IsUnivalent}_i\ (\Omega, \mathsf{dec}) = (x\ y : \Omega) \to (\mathsf{dec}(x) \Leftrightarrow \mathsf{dec}(y)) \to \mathbf{Id}(\Omega, x, y)\end{aligned}$$

We say that an abstract universe of propositions (Ω, dec) is *univalent* if the type $\mathsf{IsUnivalent}\ (\Omega, \mathsf{dec})$ is inhabited, and we note that HPropIsUnivalent is precisely the statement that $(\mathsf{HProp}, \mathsf{id})$ is univalent.

One important difference between HProp and an arbitrary universe of propositions (Ω, dec) is that the former has many inhabitants (**Unit**, **Void**, $A \to$ **Void**, etc.) because it contains all the propositions in $\mathbf{U}$, whereas the latter need not be inhabited at all.

Exercise 5.7. Show that $(\mathbf{Void}, \mathbf{absurd}(-))$ and $(\mathbf{Unit}, \lambda_ \to \mathbf{Unit})$ are univalent universes of propositions.

There are several ways we might rule out these trivial examples. One is to require our universes of propositions to be closed under a variety of logical operations (true, false, implication, etc.). Instead, we will require that our universe contains a code for every proposition in HProp up to interprovability. We say that an abstract universe of $\mathbf{U}_i$-small propositions (Ω, dec) is *adequate* if there is a map $\mathsf{enc} : \mathsf{HProp}_i \to \Omega$ such that $\mathsf{dec}(\mathsf{enc}(A)) \Leftrightarrow A$ for all $A : \mathsf{HProp}_i$, a condition clearly satisfied by $(\mathsf{HProp}_i, \mathsf{id})$.

$$\begin{aligned}&\mathsf{IsAdequate}_i : \mathsf{PropUniverse}_i \to \mathbf{U}_{i+1}\\ &\mathsf{IsAdequate}_i\ (\Omega, \mathsf{dec}) = \textstyle\sum_{\mathsf{enc}:\mathsf{HProp}_i \to \Omega} (A : \mathsf{HProp}_i) \to \mathsf{dec}(\mathsf{enc}(A)) \Leftrightarrow A\end{aligned}$$

We may then ask whether it is consistent to assume that ITT has an adequate, univalent universe of $\mathbf{U}_i$-small propositions for every i, or in other words, whether it is consistent to postulate an axiom of the following type:

$$\mathsf{PropositionalUnivalence}_i = \textstyle\sum_{X:\mathsf{PropUniverse}_i} \mathsf{IsUnivalent}_i(X) \times \mathsf{IsAdequate}_i(X)$$

Note that if (Ω, dec) is univalent, then $\mathsf{IsAdequate}_i(\Omega, \mathsf{dec})$ can be rephrased as the condition that $\mathsf{dec} : \Omega \to \mathsf{HProp}_i$ has a left inverse.

Theorem 5.1.8. *The set model of ITT extends to a model of ITT with the axioms* $\mathsf{PropositionalUnivalence}_i$ *for every* i.

Proof. Constructing such a model amounts to choosing an element of the set

$$\mathsf{Tm}_{\mathcal{S}}(\mathbf{1}_{\mathcal{S}}, [\![\mathsf{PropositionalUnivalence}_i]\!]_{\mathcal{S}})$$

for each i, where $\mathcal{S}$ is the set model of ETT and $[\![-]\!]$ is the translation of ITT into ETT.

Unfolding definitions, we must choose an element of $\mathsf{Tm}_{\mathcal{S}}(\mathbf{1}_{\mathcal{S}}, [\![\mathsf{PropUniverse}_i]\!])$ and verify that it is univalent (by checking an equation) and adequate (by constructing a map satisfying some condition). Elements of $\mathsf{Tm}_{\mathcal{S}}(\mathbf{1}_{\mathcal{S}}, [\![\mathsf{PropUniverse}_i]\!])$ are in turn pairs of a set $\Omega \in \mathcal{V}_{i+1}$ and a map $\mathsf{dec} : \Omega \to \mathcal{V}_i$ such that $|\mathsf{dec}(x)| \leq 1$ for all $x \in \Omega$. We choose $\Omega := \{\top, \bot\}$ with $\mathsf{dec}(\top) := \{\star\}$ and $\mathsf{dec}(\bot) := \emptyset$.

To see that (Ω, dec) is univalent, we must show that for all $x, y \in \Omega$, if there are functions $\mathsf{dec}(x) \to \mathsf{dec}(y)$ and $\mathsf{dec}(y) \to \mathsf{dec}(x)$ then $x = y$. This is immediate by case analysis: whenever $x \neq y$, $\mathsf{dec}(x)$ and $\mathsf{dec}(y)$ are $\{\star\}$ and $\emptyset$ (or vice versa), in which case there are no functions $\{\star\} \to \emptyset$.

Finally, to see that (Ω, dec) is adequate, we must construct a function

$$\mathsf{enc} : \{X \in \mathcal{V}_i \mid |X| \leq 1\} \to \Omega$$

along with, for every $X \in \mathcal{V}_i$ satisfying $|X| \leq 1$, a choice of maps $X \to \mathsf{dec}(\mathsf{enc}(X))$ and $\mathsf{dec}(\mathsf{enc}(X)) \to X$. We take enc to be the function that sends $\emptyset$ to $\bot$ and all one-element sets to $\top$. We leave it to the reader to verify that the required maps exist uniquely in both cases. □

Corollary 5.1.9. *Intensional type theory is consistent with the axioms* Funext, UIP, *and* $\mathsf{PropositionalUnivalence}_i$ *for all* i.

Propositional resizing The proof of Theorem 5.1.8 establishes a stronger result than the theorem statement: the set model supports a *single* univalent universe of propositions $\{\top, \bot\}$ that is simultaneously adequate for $\mathbf{U}_i$-small propositions of every universe level i. To make this statement precise, we observe that elements of $\mathsf{PropUniverse}_i$ can be "lifted" to $\mathsf{PropUniverse}_{i+1}$ by using the **lift** operations between type universes $\mathbf{U}_i \to \mathbf{U}_{i+1}$ stipulated in Section 2.6.3.

Exercise 5.8. Define functions $\mathsf{lift}'_i : \mathsf{PropUniverse}_i \to \mathsf{PropUniverse}_{i+1}$ for every i, in terms of the operators $\mathbf{lift}_i(-) : \mathbf{U}_i \to \mathbf{U}_{i+1}$.

Notation 5.1.10. We suppress applications of lift'_i and $\mathbf{lift}_i(-)$ in our informal notation. Instead, we write $(\mathsf{HProp}_0, \mathsf{id})$ for both the element of $\mathsf{PropUniverse}_0$ and its image under lift'_0 in $\mathsf{PropUniverse}_1$.

We conclude that the proof of Theorem 5.1.8 shows that the set model of ITT is consistent with axioms stating that (1) there is a universe $(\Omega, \mathsf{dec}) : \mathsf{PropUniverse}_0$ such that (2) $\mathsf{IsUnivalent}_0(\Omega, \mathsf{dec})$ holds, and (3) $\mathsf{IsAdequate}_i(\Omega, \mathsf{dec})$ holds for all i.

Before moving on, we simplify this axiomatization by rephrasing axiom schema (3) so as to not refer to the univalent universe stipulated in axioms (1) and (2). Indeed, the salient point of (3) is simply that it is possible to fit all of the propositions of type theory into a single universe of propositions, a property introduced in Remark 2.7.14 under the name of *impredicativity*.

To rephrase axiom schema (3), we note that it implies $\mathsf{IsAdequate}_i(\mathsf{HProp}_0, \mathsf{id})$ for all i, where the necessary encoding maps $\mathsf{HProp}_i \to \mathsf{HProp}_0$ are obtained by composing $\mathsf{enc}_i : \mathsf{HProp}_i \to \Omega$ with $\mathsf{dec} : \Omega \to \mathsf{HProp}_0$. These encoding maps are often called "resize" because they take any large proposition HProp_i to an equivalent small proposition HProp_0 (i.e., one for which $\mathsf{resize}(A) \Leftrightarrow A$), and the corresponding axiom schema expressing impredicativity is known as *propositional resizing*:

$$\mathsf{PropositionalResizing}_i = \mathsf{IsAdequate}_i(\mathsf{HProp}_0)$$

Corollary 5.1.11. *Intensional type theory is consistent with the axioms* Funext, UIP, $\mathsf{PropositionalUnivalence}_0$, *and* $\mathsf{PropositionalResizing}_i$ *for all* i.

Exercise 5.9. Show that the axioms of Corollary 5.1.11 imply the axioms (1), (2), and (3) mentioned above. Conclude that they imply $\mathsf{PropositionalUnivalence}_i$ for all i.

Remark 5.1.12. In category theory, univalent and impredicative universes of propositions are known as *subobject classifiers*. Subobject classifiers play a central role in elementary topoi, the categorical axiomatization of the category of sets, and type theory extended by the axioms listed in Corollary 5.1.11 is a common "type-theorist's substitute" for set theory. ◇

Remark 5.1.13. The set model of ITT factors through the set model of ETT, so from the model constructions above we may deduce that ETT is consistent with the axioms $[\![\mathsf{PropositionalUnivalence}_0]\!]$ and $[\![\mathsf{PropositionalResizing}_i]\!]$ for all i. ◇

What are these principles good for? Because propositional univalence characterizes the otherwise underspecified identity type of HProp_0, it upgrades many properties of HProp_0 from holding only morally to holding literally. For example, propositions should form a meet-semilattice under conjunction, but in ITT (and even ETT) this holds only up to interprovability unless we assume propositional univalence.

The real power, however, appears when we combine univalence with resizing. Readers familiar with category theory may know that subobject classifiers (along with some other connectives of type theory) suffice to construct finite colimits such as booleans, coproducts, and even quotients. Even resizing by itself lets us construct previously out-of-reach connectives such as propositional truncation.

Lemma 5.1.14. *Assuming* $\mathsf{PropositionalResizing}_i$ *for all* i, *propositional truncation (in the sense of Section 2.7.3) is definable.*

Proof. Using propositional resizing, we define propositional truncation as follows:

$$\begin{aligned}&\mathbf{Trunc} : \mathbf{U}_i \to \mathsf{HProp}_0\\&\mathbf{Trunc}(A) = \mathsf{resize}((X : \mathsf{HProp}_0) \to (A \to X) \to X)\end{aligned}$$

By construction, $\mathbf{Trunc}(A)$ is a proposition. To establish the mapping out property, it suffices to show that $(A \to B) \Leftrightarrow (\mathbf{Trunc}(A) \to B)$ for all propositions B; by resizing, it moreover suffices to consider the case where $B : \mathsf{HProp}_0$. In the forward direction, given $f : A \to B$ we send $p : \mathbf{Trunc}(A)$ to $p\ B\ f : B$. In the reverse direction, given $g : \mathbf{Trunc}(A) \to B$ we send $a : A$ to $g\ (\lambda C\ f \to f\ a) : B$. □

From propositional truncation we also obtain disjunction and existential quantification, as discussed in Section 2.7.3. We conclude that the set models of ITT and ETT extend to models of all three of these connectives; we can moreover show that ITT and ETT are consistent with the law of excluded middle (Section 2.7.4). Adapting the earlier statement of the law of excluded middle from ETT to ITT, we arrive at

$$\begin{aligned}&\mathsf{LEM}_i : \mathbf{U}_{i+1}\\&\mathsf{LEM}_i := (A : \mathsf{HProp}_i) \to A + (A \to \mathbf{Void})\end{aligned}$$

Exercise 5.10. Show that ITT is consistent with LEM_i for all i. (Hint: first observe that the set model of ITT extends a model of ITT with the axiom $\mathsf{IsAdequate}_i(\mathbf{Bool}, \lambda b \to \mathbf{if}\,(\mathbf{Unit}, \mathbf{Void}, b))$, then show that this axiom implies LEM_i.)

Exercise 5.11. Show that ITT extended with LEM_i satisfies $\mathsf{PropositionalUnivalence}_i$ as well as $\mathsf{PropositionalResizing}_i$.

5.2 *Homotopy type theory*

Having warmed up with propositional univalence, we now turn our attention to the full *univalence* principle of Voevodsky [Voe10]. Just as function extensionality, UIP, and propositional univalence respectively characterize the identity types of **Π**-types, **Id**-types, and universes of propositions, the univalence principle characterizes the identity types of $\mathbf{U}_i$, the universes of arbitrary types.

Recall from Section 5.1 that a universe of propositions is univalent if interprovable propositions in that universe are identified. To generalize this condition to type universes, we assert roughly that *isomorphic types* in **U** are identified, but we must immediately add a caveat: unlike propositional univalence, there are many nonequivalent ways one might state this principle formally, and many of them are inconsistent!

Compounding our difficulties, there is often more than one isomorphism between pairs of isomorphic types $A, B : \mathbf{U}$—even **Bool** is isomorphic to itself in two distinct ways—so the assertion that isomorphisms induce identifications $\mathbf{Id}(\mathbf{U}, A, B)$ is incompatible with UIP. As a result, we can no longer model types as sets; in fact, models of univalence require mathematical machinery far outside the scope of this book [KL21].

In Section 5.2.1 we precisely state and then analyze the univalence principle. In Section 5.2.2 we will show that univalence refutes every possible variation $\mathsf{U}(\mathsf{IP})^n$ of uniqueness of identity proofs as introduced in Section 4.3.2.1. We then reintroduce these principles on a type-by-type basis as homotopy levels, one of the most important concepts in homotopy type theory (intensional type theory extended by univalence). In Section 5.2.3 we introduce higher inductive types, a generalization of inductive types suggested by the failure of UIP. Finally, in Section 5.2.4 we briefly survey a few interesting applications of homotopy type theory.

Remark 5.2.1. Homotopy type theory (HoTT) is a new and active subfield of dependent type theory, and we cannot possibly do it justice in this section. We strongly encourage interested readers to seek out other resources, such as the community-written "HoTT Book" [UF13] and Rijke's *Introduction to Homotopy Type Theory* [Rij25]. Throughout this section we will reference the HoTT Book for a number of workhorse lemmas; the same results can be found in Part II of Rijke [Rij25]. ◇

Notation 5.2.2. Throughout this section we continue in the informal notation for intensional type theory used in Chapter 1 and Sections 4.1 and 5.1.

Axiom 5.2.3. We continue to work in ITT extended by the function extensionality axiom funext : Funext as defined in Section 4.3.

5.2.1 *The univalence principle*

In order to state the univalence principle we must first present a series of strange auxiliary definitions; we assure the reader that a lengthy discussion will follow. First, we observe that whenever we have an identification $\mathbf{Id}(\mathbf{U}, A, B)$ between types, we can use subst to obtain a "cast" (or "coercion") function $A \to B$.

$$\begin{aligned}&\mathsf{coe} : \{A\ B : \mathbf{U}\} \to \mathbf{Id}(\mathbf{U}, A, B) \to A \to B\\ &\mathsf{coe}\ p = \mathsf{subst}\ \mathsf{id}\ p\end{aligned}$$

In fact, using **Id**-elimination we can show that coe p is always an isomorphism. (We will prove this momentarily, but note that the inverse map to coe p is coe (sym p), or coercion along the reverse identification.) There are several ways of stating that a

map is an isomorphism, and we will insist on a slightly peculiar one: a map $f : A \to B$ is an *equivalence*, written $\mathsf{IsEquiv}(f)$, if every $b : B$ has a unique preimage $f^{-1}(b)$ in A.

$$\mathsf{IsContr}(X) = \textstyle\sum_{x:X}(y : X) \to \mathbf{Id}(X, x, y)$$
$$\mathsf{IsEquiv}(f) = (b : B) \to \mathsf{IsContr}(\textstyle\sum_{a:A} \mathbf{Id}(B, b, f(a)))$$

Here $\mathsf{IsContr}(X)$ ("X is *contractible*") is the statement that the type X has a unique element—there is a choice of $x : X$ such that every $y : X$ is the same as x—and thus $\mathsf{IsEquiv}(f)$ states that for all $b : B$ there is a unique $a : A$ such that b is $f(a)$.[2]

To prove that $\mathsf{coe}\ p$ is an equivalence, we can use the **Id**-eliminator j (Exercise 4.4) to consider only the case where p is **refl**. Coercion along **refl** is definitionally equal to the identity function (by $\mathsf{coe}\ \mathbf{refl} = \mathsf{subst}\ \mathsf{id}\ \mathbf{refl} = \mathsf{id}$) so it suffices to show that for all $X : \mathbf{U}$ and $x : X$, the type $\sum_{y:X} \mathbf{Id}(X, x, y)$ is contractible. This type is clearly inhabited by $(x, \mathbf{refl})$; the statement that all elements of $\sum_{y:X} \mathbf{Id}(X, x, y)$ are identified with $(x, \mathbf{refl})$ is precisely singleton contractibility (uniq from Section 4.1.2).

$$\mathsf{coeisequiv} : (A\ B : \mathbf{U}) \to (p : \mathbf{Id}(\mathbf{U}, A, B)) \to \mathsf{IsEquiv}(\mathsf{coe}\ p)$$
$$\mathsf{coeisequiv} = \mathsf{j}\ (\lambda A\ B\ p \to \mathsf{IsEquiv}(\mathsf{coe}\ p))\ (\lambda X\ x \to ((x, \mathbf{refl}), \mathsf{uniq}))$$

We can bundle coe and coeisequiv into a single map $\mathsf{idtoequiv}_i$ from identifications $\mathbf{Id}(\mathbf{U}_i, A, B)$ to *equivalences* $A \simeq B$, i.e., pairs of a function $f : A \to B$ and a proof of $\mathsf{IsEquiv}(f)$. Simply put, identifications of types induce equivalences.

$$_\simeq_ : \mathbf{U}$$
$$A \simeq B = \textstyle\sum_{f:A\to B} \mathsf{IsEquiv}(f)$$

$$\mathsf{idtoequiv}_i : (A\ B : \mathbf{U}_i) \to \mathbf{Id}(\mathbf{U}_i, A, B) \to A \simeq B$$
$$\mathsf{idtoequiv}_i\ A\ B\ p = (\mathsf{coe}\ p, \mathsf{coeisequiv}\ A\ B\ p)$$

The univalence principle is the assertion that $\mathsf{idtoequiv}_i\ A\ B$ is an equivalence for all $A, B : \mathbf{U}_i$. That is, not only do identifications of types in $\mathbf{U}_i$ induce equivalences, but equivalences of types in $\mathbf{U}_i$ conversely induce identifications!

$$\mathsf{Univalence}_i = (A\ B : \mathbf{U}_i) \to \mathsf{IsEquiv}(\mathsf{idtoequiv}_i\ A\ B)$$

We emphasize that univalence asserts that a *particular* map $\mathbf{Id}(\mathbf{U}_i, A, B) \to A \simeq B$ is an equivalence, not just that there exists a map $A \simeq B \to \mathbf{Id}(\mathbf{U}_i, A, B)$ going in the opposite direction. As we have previously seen e.g. in the definition of **Bool** (Section 2.5.2), such conditions let us deduce not only the existence of a map in the opposite direction but also how that map must behave on certain inputs.

[2] The type $\sum_{a:A} \mathbf{Id}(B, b, f(a))$ is known as the *(homotopy) fiber* of f at b. Using this terminology, f is an equivalence if all of its fibers are contractible.

Remark 5.2.4. The name of *univalence* for this property is due to Voevodsky, who explained in a 2014 lecture [Voe14] that he intended it to evoke the not-quite universal property satisfied by univalent universes, namely that every family of types has exactly one "classifying map" into $\mathbf{U}$—unless the family is too large, in which case there is no such map. Voevodsky also attributes the term in part to a translation quirk in the Russian edition of Boardman and Vogt [BV73], in which *faithful functor* was translated as *univalent functor* (*univalentnyj funktor*), perhaps in reference to the univalent functions from complex analysis. ◇

Definition 5.2.5. *Homotopy type theory*, or *HoTT*, is intensional type theory extended by axioms $\mathsf{funext} : \mathsf{Funext}$ and $\mathsf{univalence}_i : \mathsf{Univalence}_i$ for every i.

Remark 5.2.6. Homotopy type theory refers both to a particular formal system (Definition 5.2.5) and to the entire subfield of type theory concerned with the univalence principle and related topics. When there is possibility for confusion, we will use the phrase *Book HoTT* (as in, "HoTT as formulated in the HoTT Book [UF13]") to refer unambiguously to the formal system. ◇

Univalence is not validated by the set model; unlike in Section 5.1, there is no way to get around the fact that isomorphic sets are not equal. In fact, for reasons that will become clear in Section 5.2.2, it is quite difficult to construct *any* model of HoTT. The first and "standard" model of HoTT is due to Voevodsky [KL21] and interprets types as Kan complexes, a common definition of ∞-groupoid (Section 4.3.2.1). Perhaps the most important consequence of this model is:

Theorem 5.2.7 (Kapulkin and Lumsdaine [KL21])**.** *Homotopy type theory is consistent.*

As discussed in Section 4.3.1, adding axioms to ITT preserves the properties of normalization and decidable type-checking but generally—including in the case of univalence—disrupts canonicity. In Sections 5.3 and 5.4 we will present an alternative to Book HoTT known as *cubical type theory*, which admits the univalence principle while also enjoying the metatheoretic properties of canonicity, normalization, etc.

The statement of univalence is rather involved and has likely raised more questions than it has answered, including why we have introduced this particular definition of equivalence, and how univalence relates to propositional univalence. It is difficult to fully answer these questions without first developing considerable machinery—for which we again recommend that the reader consult a more comprehensive resource—but we will nevertheless attempt some explanations.

On equivalences Our definition of $\mathsf{IsEquiv}$ is likely unfamiliar to the reader, so we start our discussion by exploring its properties. First and foremost, a map $f : A \to B$ is an equivalence if and only if it has an inverse $B \to A$ in the usual sense:

$$\mathsf{HasInverse}(f) = \textstyle\sum_{g:B\to A}(\mathbf{Id}(A \to A, g \circ f, \mathsf{id}_A)) \times (\mathbf{Id}(B \to B, f \circ g, \mathsf{id}_B))$$

Lemma 5.2.8 (Sections 4.1 to 4.4 [UF13]). *For all $f : A \to B$, $\mathsf{IsEquiv}(f)$ is inhabited if and only if $\mathsf{HasInverse}(f)$ is inhabited.*

Proof sketch. In the forward direction we are given $p : \mathsf{IsEquiv}(f)$ and must construct a function $g : B \to A$. For any $b : B$ we note that $p(b) : \mathsf{IsContr}(\sum_{a:A} \mathbf{Id}(B, b, f(a)))$, so $\mathbf{fst}(p(b)) : \sum_{a:A} \mathbf{Id}(B, b, f(a))$ is a pair of an element of A and a proof that f sends that element to b up to identification. We therefore define $g := \lambda b \to \mathbf{fst}(\mathbf{fst}(p\ b))$; the round trip identifications follow from function extensionality, the aforementioned identity proof, and the proof of contractibility.

In the reverse direction we are given $g : B \to A$, $\alpha : \mathbf{Id}(A \to A, g \circ f, \mathsf{id}_A)$, and $\beta : \mathbf{Id}(B \to B, f \circ g, \mathsf{id}_B)$ and must construct a proof of $\mathsf{IsEquiv}(f)$. In particular, for every $b : B$ we must exhibit an element $a : A$ such that $\mathbf{Id}(B, b, f(a))$. (In fact we must show that the type $\sum_{a:A} \mathbf{Id}(B, b, f(a))$ is contractible, but we leave that part of the proof to the HoTT Book.) For the element of A, we choose $g(b)$; the identification $\mathbf{Id}(B, b, f(g(b)))$ follows by applying β to b in an appropriate sense (Remark 5.2.14). □

If $f : A \to B$ is an equivalence if and only if it has an inverse, then why did we introduce the notion of equivalence at all? The definition of $\mathsf{IsEquiv}(f)$ satisfies one critical property that $\mathsf{HasInverse}(f)$ does not: $\mathsf{IsEquiv}(f)$ is a homotopy proposition.

Lemma 5.2.9 (Lemma 3.11.4 [UF13]). *For any type X, $\mathsf{IsContr}(X)$ is a proposition.*

Corollary 5.2.10. *For all $f : A \to B$, $\mathsf{IsEquiv}(f)$ is a proposition.*

Proof. Immediate from Exercise 5.3 and Lemma 5.2.9. □

In fact, if we change the statement of univalence to refer to HasInverse instead of IsEquiv, the resulting axiom is actually *inconsistent* with ITT!

$$\mathsf{idtohasinv}_i : (A\ B : \mathbf{U}_i) \to \mathbf{Id}(\mathbf{U}_i, A, B) \to \textstyle\sum_{f:A\to B} \mathsf{HasInverse}(f)$$
$$\mathsf{idtohasinv}_i\ A\ B\ p = (\mathsf{coe}\ p, \dots)$$

$$\mathsf{Ambivalence}_i = (A\ B : \mathbf{U}_i) \to \mathsf{HasInverse}(\mathsf{idtohasinv}\ A\ B) \qquad (!?)$$

Theorem 5.2.11. *ITT extended by* Funext *and* $\mathsf{Ambivalence}_i$ *for all i is inconsistent.*

Proof sketch. We obtain a contradiction by using Ambivalence to (1) construct a type A and $\alpha : \mathbf{Id}(A \to A, \mathsf{id}_A, \mathsf{id}_A)$ such that $\mathbf{Id}(\mathbf{Id}(A \to A, \mathsf{id}_A, \mathsf{id}_A), \alpha, \mathbf{refl}) \to \mathbf{Void}$, and (2) prove that $\mathbf{Id}(\mathbf{Id}(A \to A, \mathsf{id}_A, \mathsf{id}_A), \alpha, \mathbf{refl})$. We note that (2) is the source of inconsistency here; (1) also holds in HoTT as a consequence of univalence.

To establish claim (1) it suffices to exhibit a type A for which $(a : A) \to \mathbf{Id}(A, a, a)$ is not a homotopy proposition. The simplest such constructions rely on higher inductive types (Section 5.2.3) such as the circle S^1 (Section 5.2.3.1) or propositional truncation (as shown explicitly in the HoTT Book [UF13, Theorem 4.1.3]), but the latter argument can be adapted to ITT with two nested univalent or ambivalent universes.

For claim (2) we first prove an intermediate lemma. Suppose $f : A \to B$, $(g, \beta, \gamma) : \mathsf{HasInverse}(f)$, and $a : A$. There are two ways to identify $f(g(f(a)))$ and $f(a)$: by using β to cancel $g \circ f$ (i.e., $\mathsf{cong}\ f\ (\beta\ a)$) and by using γ to cancel $f \circ g$ (i.e., $\mathsf{cong}\ (\lambda x \to x\ (f\ a))\ \gamma$). These two identifications need not be related but Ambivalence allows us to construct an identification between them (and in fact to prove that $\mathsf{HasInverse}(f)$ is a proposition for all $f : A \to B$). We obtain (2) by setting $f, g = \mathsf{id}$, $\beta = \alpha$, and $\gamma = \mathbf{refl}$.

As for the lemma we recall that by Ambivalence,

$$\mathsf{idtohasinv}\ A\ B : \mathbf{Id}(\mathbf{U}, A, B) \to \textstyle\sum_{f:A\to B} \mathsf{HasInverse}(f)$$

is an isomorphism, so $(f, (g, \beta, \gamma)) : \sum_{f:A\to B} \mathsf{HasInverse}(f)$ must be identified with $\mathsf{idtohasinv}\ A\ B\ p$ for some $p : \mathbf{Id}(\mathbf{U}, A, B)$. By $\mathbf{Id}$-elimination it suffices to consider $p = \mathbf{refl}$, in which case $f = g = \mathsf{id}_A$, $\beta = \mathbf{refl}$, $\gamma = \mathbf{refl}$, and the result is immediate. □

That said, there are many suitable alternatives to IsEquiv in the statement of univalence. The reader can find several such definitions along with an extensive discussion in Chapter 4 of the HoTT Book [UF13].

We remark that equivalences appear *twice* in the statement of univalence—once in the codomain of idtoequiv and once in the assertion that idtoequiv is an equivalence—and the foregoing discussion applies only to the former ("inner") occurrence. As a trivial consequence of Lemma 5.2.8, the following statement is interprovable with $\mathsf{Univalence}_i$ and thus a perfectly acceptable formulation of the univalence principle:

$$\mathsf{Univalence}'_i = (A\ B : \mathbf{U}_i) \to \mathsf{HasInverse}(\mathsf{idtoequiv}_i\ A\ B)$$

As observed by Licata [Lic16], $\mathsf{Univalence}_i$ is also interprovable with the even simpler statement that for all $A, B : \mathbf{U}_i$ there is a map $\mathsf{ua} : A \simeq B \to \mathbf{Id}(\mathbf{U}_i, A, B)$ such that for all equivalences $(f, p) : A \simeq B$, there is an identification—often called uaβ because it resembles a β-rule—between the functions $\mathsf{coe}\ (\mathsf{ua}\ (f, p))$ and f.

$$\mathsf{Univalence}''_i = (A\ B : \mathbf{U}_i) \to \\ \textstyle\sum_{ua:(A\simeq B)\to \mathbf{Id}(\mathbf{U}_i,A,B)} ((e : A \simeq B) \to \mathbf{Id}(A \to B, \mathsf{coe}\ (ua\ e), \mathbf{fst}(e)))$$

Comparison to other principles A good starting point for understanding univalence is to compare it to the other principles we have considered adding to ITT: function extensionality, UIP, propositional univalence, and propositional resizing.

The relationship between univalence and function extensionality is straightforward albeit surprising: univalence actually *implies* function extensionality [UF13, Section 4.9]! As a result, our assumption of Funext throughout this section and in the definition of homotopy type theory (Definition 5.2.5) is actually redundant; however, by separately asserting Funext we were able to avoid some technicalities in the discussion of alternative definitions of IsEquiv and Univalence.

As their names suggest, univalence also implies propositional univalence.

Lemma 5.2.12 (Theorem 2.7.2 [UF13])**.** *For any* $x, y : \sum_{a:A} B(a)$ *there is an equivalence*

$$\mathbf{Id}\left(\textstyle\sum_{a:A} B(a), x, y\right) \simeq \sum_{p:\mathbf{Id}(A,\mathbf{fst}(x),\mathbf{fst}(y))} \mathbf{Id}(B(y), \mathsf{subst}\ B\ p\ \mathbf{snd}(x), \mathbf{snd}(y))$$

Lemma 5.2.13. *Univalence for* $\mathbf{U}_i$ *implies* HPropIsUnivalent *for* HProp_i.

Proof. Recall that $\mathsf{HProp}_i = \sum_{X:\mathbf{U}_i} \mathsf{IsHProp}_i(X)$, and suppose $(A, h_A), (B, h_B) : \mathsf{HProp}_i$ and $(f, g) : A \Leftrightarrow B$; we must show $\mathbf{Id}(\mathsf{HProp}_i, (A, h_A), (B, h_B))$. By Lemma 5.2.12 it suffices to exhibit $p : \mathbf{Id}(\mathbf{U}_i, A, B)$ such that $\mathbf{Id}(\mathsf{IsHProp}_i(B), \mathsf{subst}\ \mathsf{IsHProp}_i\ p\ h_A, h_B)$. To construct p we observe that by univalence and Lemma 5.2.8 it suffices to exhibit a pair of mutually inverse maps $A \to B$ and $B \to A$. By function extensionality and the fact that A and B are propositions, the given maps $f : A \to B$ and $g : B \to A$ are necessarily mutually inverse. Finally, the required identification between proofs of $\mathsf{IsHProp}_i(B)$ is automatic from the fact that $\mathsf{IsHProp}_i$ is a proposition (Exercise 5.12). □

One consequence of Lemma 5.2.13 is that all models of HoTT validate our original statement of propositional univalence for HProp_i, whereas in Section 5.1 we only produced a model of the "abstract" version of propositional univalence introduced in Section 5.1.3. A large class of models of HoTT additionally validate propositional resizing between all pairs of universes of propositions $\mathsf{HProp}_i, \mathsf{HProp}_j$ [Shu19], but this principle is known to be independent of univalence [Uem19].

While we are discussing propositional univalence, it is worth understanding why the statement of HPropIsUnivalent is so much simpler than that of univalence. There are two major differences between the statements besides the universes in question. First, univalence refers to equivalences $\sum_{f:A\to B} \mathsf{IsEquiv}(f)$ whereas propositional univalence refers to pairs of maps $(A \to B) \times (B \to A)$. But in the case where A and B are propositions, these two types are equivalent (in fact, $\mathsf{IsEquiv}(f) \simeq B \to A$)! Secondly, univalence states that (a version of) coe is an equivalence whereas HPropIsUnivalent simply asserts a map going in the opposite direction; we have already seen in Corollary 5.1.6 that these conditions are interprovable for HProp.

Remark 5.2.14. Surprisingly, function extensionality is yet another case in which a canonical map is an equivalence if and only if there is a map in the opposite direction. Let $A, B : \mathbf{U}$ and $f, g : A \to B$, and recall from Section 4.3.1 that (non-dependent)

function extensionality posits a map $((a : A) \to \mathbf{Id}(B, f(a), g(a))) \to \mathbf{Id}(A \to B, f, g)$ which takes pointwise identifications to identifications of functions.

Although we did not note it at the time, in ITT one can always define a map which takes identifications of functions to pointwise identifications:

$$\begin{aligned}&\mathsf{happly} : \mathbf{Id}(A \to B, f, g) \to (a : A) \to \mathbf{Id}(B, f(a), g(a))\\&\mathsf{happly}\ h\ a = \mathsf{cong}\ (\lambda x \to x\ a)\ h\end{aligned}$$

The function extensionality map implies that happly is an equivalence [Lum11]. ◇

Our final comparison is the most interesting: how does univalence relate to UIP? As we will now see, HoTT's hierarchy of univalent universes refutes not only the uniqueness of identity proofs but also the "uniqueness of identity proofs of identity proofs" and all of its further iterations as introduced in Section 4.3.2.1.

5.2.2 *Homotopy levels and the failure of UIP*

Perhaps the most significant consequence of univalence is that it refutes UIP, the statement that for all types A and elements $a, b : A$, all identity proofs $p, q : \mathbf{Id}(A, a, b)$ are identified. In fact, no matter how deeply we nest **Id**-types $\mathbf{Id}(\mathbf{Id}(\mathbf{Id}(\ldots, a, b), p, q), \alpha, \beta)$ in HoTT, there is no point at which identifications necessarily trivialize.

Do not however confuse the failure of a global property with the global failure of a property: there are many types in HoTT that do satisfy UIP "locally," and many others that satisfy $\mathsf{U}(\mathsf{IP})^n$ for some $n > 1$. Reformulated as a **Nat**-indexed family of predicates over types, $\mathsf{U}(\mathsf{IP})^n$ will turn out to be a crucial measure of the "homotopical complexity" of a type known as its *homotopy level* or *h-level*.

We begin by rephrasing UIP and UIPIP from Section 4.3 as predicates.

$$\begin{aligned}&\mathsf{HasUIP} : \mathbf{U} \to \mathbf{U}\\&\mathsf{HasUIP}\ A = (a\ b : A) \to (p\ q : \mathbf{Id}(A, a, b)) \to \mathbf{Id}(\mathbf{Id}(A, a, b), p, q)\end{aligned}$$

$$\begin{aligned}&\mathsf{HasUIPIP} : \mathbf{U} \to \mathbf{U}\\&\mathsf{HasUIPIP}\ A = (a\ b : A) \to (p\ q : \mathbf{Id}(A, a, b)) \to (\alpha\ \beta : \mathbf{Id}(\mathbf{Id}(A, a, b), p, q)) \to\\&\quad \mathbf{Id}(\mathbf{Id}(\mathbf{Id}(A, a, b), p, q), \alpha, \beta)\end{aligned}$$

Our original statement of UIP is precisely that every type satisfies HasUIP, i.e., $\mathsf{UIP} = (A : \mathbf{U}) \to \mathsf{HasUIP}(A)$ and similarly for UIPIP.

It remains to generalize HasUIP and HasUIPIP to $\mathsf{HasU}(\mathsf{IP})^n$. Extending to $n > 2$ is straightforward as soon as we notice $\mathsf{HasUIPIP}(A) = (a\ b : A) \to \mathsf{HasUIP}(\mathbf{Id}(A, a, b))$. But we can also extend the hierarchy downward:

Homotopy levels	$\mathsf{HasU(IP)}^n$	n**-types**	**Common name**
IsOfHLevel 0	$\mathsf{HasU(IP)}^{-1}$	(-2)-type	contractible
IsOfHLevel 1	$\mathsf{HasU(IP)}^{0}$	(-1)-type	proposition
IsOfHLevel 2	HasUIP	0-type	h-set
IsOfHLevel 3	HasUIPIP	1-type	1-groupoid
IsOfHLevel $(n+2)$	$\mathsf{HasU(IP)}^{n+1}$	n-type	n-groupoid

Figure 5.1: Competing terminologies for homotopy levels.

$$\mathsf{HasU(IP)}^0 : \mathbf{U} \to \mathbf{U}$$
$$\mathsf{HasU(IP)}^0\ A = (a\ b : A) \to \mathbf{Id}(A, a, b)$$

We have already seen this definition in Section 5.1: this is exactly $\mathsf{IsHProp}(A)$, the statement that A is a homotopy proposition! And although it may seem that the hierarchy stops here, in light of Lemma 5.2.15 we may take one final step downward:

$$\mathsf{HasU(IP)}^{-1} : \mathbf{U} \to \mathbf{U}$$
$$\mathsf{HasU(IP)}^{-1}\ A = \mathsf{IsContr}\ A$$

Lemma 5.2.15 (Lemma 3.11.10 [UF13])**.** *For all* $A : \mathbf{U}$, $\mathsf{IsHProp}(A)$ *if and only if* $(a\ b : A) \to \mathsf{IsContr}(\mathbf{Id}(A, a, b))$.

Following the convention introduced by Voevodsky [VAG+20], we may define a **Nat**-indexed family of predicates IsOfHLevel with the base case IsContr at $n = 0$:

$$\mathsf{IsOfHLevel} : \mathbf{Nat} \to \mathbf{U} \to \mathbf{U}$$
$$\mathsf{IsOfHLevel}\ 0\ A = \mathsf{IsContr}\ A$$
$$\mathsf{IsOfHLevel}\ (\mathsf{suc}\ n)\ A = (a\ b : A) \to \mathsf{IsOfHLevel}\ n\ (\mathbf{Id}(A, a, b))$$

It is natural (pun intended) to start numbering h-levels at 0, although this has the unfortunate consequence of disagreeing with our numbering scheme for $\mathsf{HasU(IP)}^n$ which starts at -1. Worse yet, mathematicians have yet a third numbering scheme for homotopy levels, called *homotopy n-types*;[3] see Figure 5.1 for reference.

Remark 5.2.16. The h-level numbering scheme is common in homotopy type theory but nowhere else; as far as we know, the $\mathsf{HasU(IP)}^n$ numbering scheme is unique to our book, where it is introduced purely for pedagogical reasons. For most purposes we strongly recommend following the standard numbering of homotopy n-types. ◇

[3] As discussed in Section 4.3.2.1, types satisfying UIPIP correspond to 1-groupoids, and types satisfying UIP correspond to 0-groupoids or ordinary sets, suggesting that contractible types are (-2)-groupoids.

The theory of h-levels contains many of the most important lemmas in homotopy type theory. We will briefly sketch some of the main results and direct the reader to Chapter 7 of the HoTT Book [UF13] or Chapter 12 of Rijke [Rij25] for more information.

Exercise 5.12. Show that IsOfHLevel n A is a proposition for all A, and conclude that IsHProp(A) is a proposition. (Hint: use induction, Lemma 5.2.9, and Exercise 5.3.)

Exercise 5.12 suggests that we can define universes of types of h-level n as the subtypes of each $\mathbf{U}_i$ spanned by types satisfying IsOfHLevel n. A particularly important example is the universe of types with HasUIP (i.e., of h-level 2), also known as *h-sets*.

$$\begin{aligned}&\mathsf{HSet}_i : \mathbf{U}_{i+1}\\&\mathsf{HSet}_i = \textstyle\sum_{A:\mathbf{U}_i} \mathsf{HasUIP}(A)\end{aligned}$$

In Corollary 2.7.12 we showed that propositions in ETT are closed under various connectives. Generalizations of those statements hold for h-levels in HoTT:

- IsOfHLevel 0 **Unit**, IsOfHLevel 1 **Void**, and IsOfHLevel 2 **Bool**.
- IsOfHLevel n $(\mathbf{Id}(A, a, b))$ if IsOfHLevel $(\mathsf{suc}\ n)$ A.
- IsOfHLevel n $((a : A) \to B(a))$ if $(a : A) \to$ IsOfHLevel n $(B(a))$.
- IsOfHLevel n $(\sum_{a:A} B(a))$ if IsOfHLevel n A and $(a : A) \to$ IsOfHLevel n $(B(a))$.
- IsOfHLevel $(\mathsf{suc}\ n)$ A if IsOfHLevel n A.

Exercise 5.13. Show that HSet_i is closed under **Π**-types and **Σ**-types. Is it also closed under **Id**-types? Why or why not?

Refuting UIP We have seen now that many types in HoTT satisfy HasUIP, but a direct consequence of univalence is that $\mathbf{U}_0$ is not one of them.

Theorem 5.2.17. $\mathbf{U}_0$ *is not an h-set, i.e.,* $\mathsf{HasUIP}(\mathbf{U}_0) \to \mathbf{Void}$ *is inhabited.*

Proof. Suppose $\phi : \mathsf{HasUIP}(\mathbf{U}_0)$, and recall from Section 5.2.1 that the univalence principle for $\mathbf{U}_0$ implies—in fact, is interprovable with—the statement

$$(A\ B : \mathbf{U}_0) \to \textstyle\sum_{ua:(A\simeq B)\to \mathbf{Id}(\mathbf{U}_0,A,B)} ((e : A \simeq B) \to \mathbf{Id}(A \to B, \mathsf{coe}\ (ua\ e), \mathbf{fst}(e)))$$

In particular, every equivalence $(f, pf) : \mathbf{Bool} \simeq \mathbf{Bool}$ induces an identification $\mathsf{ua}\ (f, pf) : \mathbf{Id}(\mathbf{U}_0, \mathbf{Bool}, \mathbf{Bool})$ such that $\mathsf{coe}\ (\mathsf{ua}\ (f, pf))$ is identified with f.

Applying the above to the two equivalences $\mathsf{id}_{\mathbf{Bool}}, \mathsf{not} : \mathbf{Bool} \to \mathbf{Bool}$, the latter being the map sending **true** to **false** and vice versa (which can be shown to be an

equivalence), we obtain identifications $p, q : \mathbf{Id}(\mathbf{U}_0, \mathbf{Bool}, \mathbf{Bool})$ for which $\mathbf{Id}(\mathbf{Bool} \to \mathbf{Bool}, \mathrm{coe}\ p, \mathrm{id}_{\mathbf{Bool}})$ and $\mathbf{Id}(\mathbf{Bool} \to \mathbf{Bool}, \mathrm{coe}\ q, \mathrm{not})$.

On the other hand, by $\phi : \mathsf{HasUIP}(\mathbf{U}_0)$ we have an identification between p and q:

$$\alpha : \mathbf{Id}(\mathbf{Id}(\mathbf{U}_0, \mathbf{Bool}, \mathbf{Bool}), p, q)$$
$$\alpha = \mathbf{fst}(\phi\ \mathbf{Bool}\ \mathbf{Bool}\ p\ q)$$

Thus cong coe α is an identification between coe p and coe q, and by symmetry and transitivity we obtain an identification $\beta : \mathbf{Id}(\mathbf{Bool} \to \mathbf{Bool}, \mathrm{id}_{\mathbf{Bool}}, \mathrm{not})$. But then happly $\beta\ \mathbf{true} : \mathbf{Id}(\mathbf{Bool}, \mathbf{true}, \mathbf{false})$, from which we derive a contradiction. □

The above argument is not specific to **Bool**, but it does require that $\mathbf{U}_0$ contain at least one type with provably distinct elements—that is, that $\mathbf{U}_0$ contain a non-proposition. (Indeed, the univalent universe HProp *is* an h-set!) Similarly, given that $\mathbf{U}_1$ contains the type $\mathbf{U}_0$ which is not an h-set, by a more sophisticated version of this argument due to Kraus and Sattler [KS15], one can show that $\mathbf{U}_1$ is not a 1-groupoid. Iterating this process, for any $i \in \mathbb{N}$ the type $\mathbf{U}_i$ does not have h-level $i + 2$.

Theorem 5.2.18 (Kraus and Sattler [KS15])**.** *For every external natural number $i \in \mathbb{N}$, the type* $\mathsf{IsOfHLevel}\ (\mathbf{suc}^{i+2}(\mathbf{zero}))\ \mathbf{U}_i \to \mathbf{Void}$ *is inhabited in HoTT.*

Corollary 5.2.19. *HoTT does not validate the axiom* $\mathsf{U(IP)}^n$ *for any n.*

One consequence of Theorem 5.2.18 is that every type in HoTT is equipped with an infinite tower of non-trivial **Id**-types. These identifications are symmetric and transitive, but groupoid laws such as symsym (Lemma 4.1.7) and associativity hold only *weakly*, or up to higher identifications. Remarkably, such "infinite dimensional" towers of identifications—which in type theory exist as a consequence of **j** without UIP—are known in mathematics as ∞-groupoids, where they were introduced to abstract the structure of arbitrary-dimensional *paths* in topological spaces [Por21]. (See Sections 2.1–2.4 of the HoTT Book [UF13] for more on this perspective.)

Remark 5.2.20. The branch of mathematics devoted to ∞-groupoids and their generalizations is known as *homotopy theory*, and ∞-groupoids are also known as *homotopy types*. The name "homotopy type theory" is a pun in which "type" refers both to homotopy types and to type theory. ◇

Advanced Remark 5.2.21. The homotopical perspective on type theory sheds light on many mysterious aspects of **Id**-types. In particular, people often struggle to reconcile **Id**-elimination—which states that maps out of $a : A, b : A, p : \mathbf{Id}(A, a, b)$ are controlled by their behavior on **refl**—with the existence of provably non-**refl** identifications. In homotopy theoretic terms, **Id**-elimination expresses that the inclusion of A into its path space $PA = \sum_{a,b:A} \mathbf{Id}(A, a, b)$ is a trivial cofibration, which is perfectly compatible with

A having non-trivial paths, whereas UIP expresses that the free loop space fibration $\Omega A \to A$ is a trivial fibration, which holds only when A is a discrete space (set). ◇

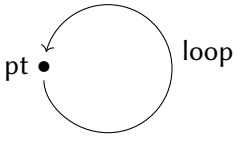
5.2.3 *Higher inductive types*

Although the behavior of identifications in HoTT mirrors the behavior of paths in topological spaces, univalent universes are thus far the only source of non-trivial paths (or non-h-sets) in the theory. To take more advantage of the connection between **Id**-types and topology, most users of HoTT consider a further extension of type theory known as *higher inductive types* (HITs), a form of inductive type generated not only by (ordinary) "point" constructors but also "path" constructors, freely-added elements of their **Id**-types. HITs allow us to axiomatize many important topological spaces in HoTT, although we note that they do increase the strength of the theory: the HITs in this section allow us to construct types with no finite h-level (i.e., $\sum_{A:\mathrm{U}}(n : \mathbf{Nat}) \to \mathsf{IsOfHLevel}\ n\ A \to \mathbf{Void}$) which is not possible with only univalence [KS15].

As in Section 2.5 we will define several representative HITs, namely the circle, suspensions, and set truncations, and make no attempt to develop a general schema for higher inductive definitions [CH19].

5.2.3.1 *The circle type*

Our first example of a HIT is S^1, the type "generated by a point $\mathrm{pt} : \mathrm{S}^1$ and a path $\mathrm{loop} : \mathbf{Id}(\mathrm{S}^1, \mathrm{pt}, \mathrm{pt})$." If we depict paths as arrows between points, S^1 is a *circle*:



As in Section 2.5, it is clear that postulating the circle must involve postulating the type former S^1 itself along with its two constructors pt and loop:

$\mathrm{S}^1 : \mathbf{U}$
$\mathrm{pt} : \mathrm{S}^1$
$\mathrm{loop} : \mathbf{Id}(\mathrm{S}^1, \mathrm{pt}, \mathrm{pt})$

and the difficulty lies entirely in specifying its elimination principle, which captures the idea that S^1 is in some sense "generated" or "determined" by pt and loop.

Of course, unlike in Section 2.5, the loop constructor of S^1 has type $\mathbf{Id}(\mathrm{S}^1, \mathrm{pt}, \mathrm{pt})$ rather than S^1 itself. We make sense of this by thinking of identifications in S^1 as part

of the higher structure of S^1 itself rather than as elements of some unrelated type, even though we can only "access" these identifications with **Id**-types.

Remark 5.2.22. We already saw in Section 2.5 that ordinary inductive types have terms that are not constructors, such as variables $\mathbf{q} \in \mathsf{Tm}(\Gamma.\mathbf{Bool}, \mathbf{Bool})$. Path constructors further complicate this situation: by applying **Id**-type operations such as sym and trans to loop, we can obtain infinitely many elements of $\mathbf{Id}(\mathsf{S}^1, \mathsf{pt}, \mathsf{pt})$—even in the empty context—that are provably not identified with loop! For this reason we say S^1 is *generated by* pt and loop, as even *internally* one can see that it has non-loop paths. ◇

As with ordinary inductive types, the introduction data allows us to define a canonical evaluation map which applies any function $\mathsf{S}^1 \to A$ to the constructors:

$$\begin{aligned}&\mathsf{eval}_{\mathsf{S}^1} : \{A : \mathbf{U}\} \to (\mathsf{S}^1 \to A) \to \textstyle\sum_{a:A} \mathbf{Id}(A, a, a)\\ &\mathsf{eval}_{\mathsf{S}^1}\ f = (f\ \mathsf{pt}, \mathsf{cong}\ f\ \mathsf{loop})\end{aligned}$$

with the caveat that "applying" $f : \mathsf{S}^1 \to A$ to $\mathsf{loop} : \mathbf{Id}(\mathsf{S}^1, \mathsf{pt}, \mathsf{pt})$ requires cong. The (non-dependent) elimination principle expresses that maps $\mathsf{S}^1 \to A$ are determined by their behavior on constructors, which we formalize by asking for a section to $\mathsf{eval}_{\mathsf{S}^1}$:

$$\begin{aligned}&\mathsf{rec}_{\mathsf{S}^1} : \{A : \mathbf{U}\} \to \textstyle\sum_{a:A} \mathbf{Id}(A, a, a) \to (\mathsf{S}^1 \to A)\\ &\mathsf{rec}_{\mathsf{S}^1}\mathsf{IsSection} : \mathbf{Id}((\textstyle\sum_{a:A} \mathbf{Id}(A, a, a)) \to (\textstyle\sum_{a:A} \mathbf{Id}(A, a, a)), \mathsf{eval}_{\mathsf{S}^1} \circ \mathsf{rec}_{\mathsf{S}^1}, \mathsf{id})\end{aligned}$$

Unfolding the above, $\mathsf{rec}_{\mathsf{S}^1}$ is a "recursion principle" for S^1 which states that for any type A, maps $\mathsf{S}^1 \to A$ are defined by a choice of point $a : A$ and path $p : \mathbf{Id}(A, a, a)$, and $\mathsf{rec}_{\mathsf{S}^1}\mathsf{IsSection}$ states that we have two "β-rule" identifications: between $\mathsf{rec}_{\mathsf{S}^1}(a, p)\ \mathsf{pt}$ and a, and (roughly) between $\mathsf{cong}\ (\mathsf{rec}_{\mathsf{S}^1}(a, p))\ \mathsf{loop}$ and p.

It is natural to wonder whether these "β-rules" should be definitional equalities rather than just identifications. It is certainly more convenient to make them definitional; in the absence of a definitional equality $\mathsf{rec}_{\mathsf{S}^1}(a, p)\ \mathsf{pt} = a$, the left-hand side of the second β-rule, $\mathsf{cong}\ (\mathsf{rec}_{\mathsf{S}^1}(a, p))\ \mathsf{loop}$, actually has type

$$\mathbf{Id}(A, \mathsf{rec}_{\mathsf{S}^1}(a, p)\ \mathsf{pt}, \mathsf{rec}_{\mathsf{S}^1}(a, p)\ \mathsf{pt})$$

whereas the right-hand side has type $\mathbf{Id}(A, a, a)$, so subst is required to even state the second β-rule. On the other hand, it would be quite strange to include a definitional equality that mentions cong—a user-defined function that exists independently of S^1—and moreover the intended models of higher inductive types often do not validate the second β-rule definitionally [LS19].

There is no fully satisfactory solution to this problem in Book HoTT, but the standard compromise is to assert β-rules on points as definitional equalities, and β-rules on paths as holding only up to identification [UF13]. For simplicity, we will continue treating all β-rules of HITs as holding only up to identification.

Stating the dependent elimination principle for S^1 requires us to generalize the evaluation map to dependent functions out of S^1:

$$\mathsf{deval}_{\mathsf{S}^1} : (A : \mathsf{S}^1 \to \mathbf{U}) \to ((x : \mathsf{S}^1) \to A\,x) \to \textstyle\sum_{a:A\ \mathrm{pt}} \mathbf{Id}(A\ \mathrm{pt}, \mathsf{subst}\ A\ \mathsf{loop}\ a, a)$$
$$\mathsf{deval}_{\mathsf{S}^1}\ A\ f = (f\ \mathrm{pt}, \mathsf{dcong}\ f\ \mathsf{loop})$$

Before arriving at the eliminator, we can simplify matters slightly by recalling that in Section 2.5.5 we showed that η-rules for inductive types hold whether or not they are added explicitly. That argument took place in ETT and used equality reflection to derive judgmental η-rules, but in HoTT one can replay a prefix of that argument and conclude that asking for a family of sections to $\mathsf{deval}_{\mathsf{S}^1}$ is the same as asking for $\mathsf{deval}_{\mathsf{S}^1}$ to be an equivalence; in both cases, the second round trip is automatic.

Summing up, our specification of the circle HIT therefore consists of a type $\mathsf{S}^1 : \mathbf{U}$, constructors $\mathrm{pt} : \mathsf{S}^1$ and $\mathsf{loop} : \mathbf{Id}(\mathsf{S}^1, \mathrm{pt}, \mathrm{pt})$, and the following elimination principle:

$$\mathsf{Elimination}_{\mathsf{S}^1} : (A : \mathsf{S}^1 \to \mathbf{U}) \to \mathsf{isEquiv}(\mathsf{deval}_{\mathsf{S}^1}\ A)$$

The inverse map to $\mathsf{deval}_{\mathsf{S}^1}\ A$ is of course the dependent eliminator for S^1, and the two round trips unfold precisely to identifications expressing the β- and η-laws.

Remark 5.2.23. Yet again we are asking for a canonical map to be an equivalence; phrasing S^1-elimination in this way bundles the eliminator and its β- and η-laws into a very concise package which is moreover a proposition. ◇

We conclude our discussion of the circle by using univalence to prove that the loop constructor is not identified with **refl**, and thus that S^1 is not an h-set.

Lemma 5.2.24. *S^1 is not an h-set; in particular,* $\mathbf{Id}(\mathbf{Id}(\mathsf{S}^1, \mathrm{pt}, \mathrm{pt}), \mathsf{loop}, \mathbf{refl}) \to \mathbf{Void}$.

Proof. Using S^1-elimination, we can define a function $f : \mathsf{S}^1 \to \mathbf{U}$ sending pt to **Bool** and loop to the path $\mathsf{ua}\ \mathsf{not} : \mathbf{Id}(\mathbf{U}, \mathbf{Bool}, \mathbf{Bool})$ induced by univalence applied to $\mathsf{not} : \mathbf{Bool} \simeq \mathbf{Bool}$, reusing notation from the proof of Theorem 5.2.17. Suppose loop and **refl** are identified. Then $\mathsf{cong}\ f\ \mathsf{loop}$ and $\mathsf{cong}\ f\ \mathbf{refl}$ would be identified, and in turn $\mathsf{ua}\ \mathsf{not}$ and **refl** would be identified. But $\mathsf{coe}\ (\mathsf{ua}\ \mathsf{not})$ is not and $\mathsf{coe}\ \mathbf{refl}$ is $\mathrm{id}_{\mathbf{Bool}}$, and as we have already seen in the proof of Theorem 5.2.17, these are not identified. □

The proof of Lemma 5.2.24 makes essential use of univalence. In the absence of univalence, the rules for S^1 can be validated in the set model of ITT—and are thus consistent with global UIP—by interpreting S^1 as a one-element set. In type theories with UIP, higher inductive types do not behave as topological spaces but rather as inductive types subject to equations (or "data types with laws" [Tur85]). Because there is nothing "higher" about such HITs, they are often called *quotient inductive types* (QITs); the reader has already met one, namely propositional truncation (Section 2.7.3).

Exercise 5.14. Write down a specification of **Bool** in the style of our specification of S^1, then show that the usual rules for **Bool** imply that $\mathsf{deval}_{\mathbf{Bool}} : (A : \mathbf{Bool} \to \mathbf{U}) \to ((b : \mathbf{Bool}) \to A\ b) \to (A\ \mathbf{true} \times A\ \mathbf{false})$ is an equivalence for all A.

Exercise 5.15. Show that if $B : \mathbf{U}$, $b_t, b_f : B$, and the evaluation map $(A : B \to \mathbf{U}) \to ((b : B) \to A\ b) \to (A\ b_t \times A\ b_f)$ is an equivalence for all A, then $B \simeq \mathbf{Bool}$.

5.2.3.2 Suspensions

Our next example of a HIT is the *suspension* **Susp** A of a type A, which is generated by two point constructors **north**, **south** : **Susp** A and a *family* of path constructors **merid** : $(a : A) \to \mathbf{Id}(\mathbf{Susp}\ A, \mathbf{south}, \mathbf{north})$. The intuition behind these strange constructor names is that **Susp** A can be pictured as a globe consisting of a north and south pole joined by a meridian path for every $a : A$.

As with the circle, we start our specification of **Susp** A by postulating constructors:

$\mathbf{Susp} : \mathbf{U} \to \mathbf{U}$
$\mathbf{north} : \{A : \mathbf{U}\} \to \mathbf{Susp}\ A$
$\mathbf{south} : \{A : \mathbf{U}\} \to \mathbf{Susp}\ A$
$\mathbf{merid} : \{A : \mathbf{U}\} \to (a : A) \to \mathbf{Id}(\mathbf{Susp}\ A, \mathbf{south}, \mathbf{north})$

Using these constructors we define a dependent evaluation map, noting that (as in coproduct types and **Nat**) evaluation at **merid** is parameterized by $a : A$.

$$\begin{aligned}&\mathsf{deval}_{\mathbf{Susp}} : \{A : \mathbf{U}\} \to (B : \mathbf{Susp}\ A \to \mathbf{U}) \to ((x : \mathbf{Susp}\ A) \to B\ x) \to \\ &\quad \textstyle\sum_{b_s : B\ \mathbf{south}} \sum_{b_n : B\ \mathbf{north}} (a : A) \to \mathbf{Id}(B\ \mathbf{north}, \mathsf{subst}\ B\ (\mathbf{merid}\ a)\ b_s, b_n) \\ &\mathsf{deval}_{\mathbf{Susp}}\ B\ f = (f\ \mathbf{south}, f\ \mathbf{north}, \lambda a \to \mathsf{dcong}\ f\ (\mathbf{merid}\ a))\end{aligned}$$

Finally, the elimination principle states that evaluation is an equivalence for all A, B.

$$\mathsf{Elimination}_{\mathbf{Susp}} : \{A : \mathbf{U}\} \to (B : \mathbf{Susp}\ A \to \mathbf{U}) \to \mathsf{isEquiv}(\mathsf{deval}_{\mathbf{Susp}}\ B)$$

Suspensions are an important tool in homotopy theory.

Exercise 5.16. Show that $S^1 \simeq \mathbf{Susp}\ \mathbf{Bool}$. (Hint: draw a picture.)

In fact one can define n-spheres S^n as the n-fold suspensions of **Bool**: the 1-sphere is the circle, the 2-sphere $S^2 = \mathbf{Susp}\ (\mathbf{Susp}\ \mathbf{Bool})$ is the ordinary sphere, and so forth.

$S^- : \mathbf{Nat} \to \mathbf{U}$
$S^0 = \mathbf{Bool}$
$S^{\mathsf{suc}(n)} = \mathbf{Susp}\ S^n$

Lemma 5.2.24 generalizes to the statement that S^n refutes IsOfHLevel (suc n) for all $n : \mathbf{Nat}$ [UF13, Theorem 8.6.17]. As a result, the disjoint union of all n-spheres is a single type with no finite h-level, resolving the remark at the start of this section.[4]

Corollary 5.2.25. *For all* $m : \mathbf{Nat}$*, the type* $\sum_{n:\mathbf{Nat}} S^n$ *does not have h-level* m*.*

5.2.3.3 *Set truncations*

For our final example of HITs we consider *set truncations* $|A|$, which are considerably more complex than S^1 and **Susp** due to having a recursive path constructor. In our last encounter with recursive constructors—namely suc : **Nat** → **Nat** in Section 2.5.4—we needed to phrase the mapping out property of **Nat** in terms of displayed algebra homomorphisms out of an initial algebra; the same will be true here.

The set truncation $|A|$ of a type A has two constructors, starting with a point $[a] : |A|$ for every $a : A$. Then, for every pair of paths $p, q : \mathbf{Id}(|A|, x, y)$ in the set truncation itself, it has a path **trunc** p q between those two paths.

$$
\begin{aligned}
&|_| : \mathbf{U} \to \mathbf{U}\\
&[_] : \{A : \mathbf{U}\} \to A \to |A|\\
&\mathbf{trunc} : \{A : \mathbf{U}\}\ \{x\ y : |A|\} \to (p\ q : \mathbf{Id}(|A|, x, y)) \to \mathbf{Id}(\mathbf{Id}(|A|, x, y), p, q)
\end{aligned}
$$

Because we think of the **Id**-types of a HIT as part of its specification, we consider not only $x, y : |A|$ but also $p, q : \mathbf{Id}(|A|, x, y)$ as recursive arguments of **trunc**.

To specify the elimination principle we must adapt the notions of (displayed) algebra and (displayed) algebra homomorphism from Section 2.5.4 to the HIT setting. Skipping to the end of this process, for each $A : \mathbf{U}$ we define *displayed algebras over* $(|A|, [-], \mathbf{trunc})$ as triples (B, b, β) of:

- a type $B : |A| \to \mathbf{U}$,
- a function $b : (a : A) \to B\ [a]$, and
- a function β which for any $x, y : |A|$, $\tilde{x} : B(x)$, $\tilde{y} : B(y)$, $p, q : \mathbf{Id}(|A|, x, y)$, $\tilde{p} : \mathbf{Id}(B(y), \mathsf{subst}\ B\ p\ \tilde{x}, \tilde{y})$, and $\tilde{q} : \mathbf{Id}(B(y), \mathsf{subst}\ B\ q\ \tilde{x}, \tilde{y})$ produces an identification $\mathbf{Id}(\mathbf{Id}(B(y), \mathsf{subst}\ B\ q\ \tilde{x}, \tilde{y}), \tilde{p}', \tilde{q})$ where

$$\tilde{p}' = \mathsf{subst}\ (\lambda r \to \mathbf{Id}(B(y), \mathsf{subst}\ B\ r\ \tilde{x}, \tilde{y}))\ (\mathbf{trunc}\ p\ q)\ \tilde{p}$$

[4] In the standard model of HoTT, even S^2 does not have finite h-level. To the authors' knowledge, it is still open whether this classic result of Serre [Ser53] can be shown inside HoTT.

and *displayed algebra homomorphisms from* $(|A|, [-], \mathbf{trunc})$ *to* (B, b, β) as functions $f : (x : |A|) \to B(x)$ that send $[-]$ to b and **trunc** to β in an appropriate sense.

These definitions are mechanically derivable from the constructors but are admittedly rather unwieldy. There is however a considerably simpler perspective in the case of set truncation, starting from the observation that the constructors for $|A|$ are just a pair of a map $[-] : A \to |A|$ and a proof **trunc** : HasUIP $|A|$ that $|A|$ is an h-set.

Lemma 5.2.26 (Lemma 6.9.1 [UF13])**.** *A displayed algebra* (B, b, β) *over* $(|A|, [-], \mathbf{trunc})$ *is equivalently a type* $B : |A| \to \mathbf{U}$*, a function* $b : (a : A) \to B\ [a]$*, and a proof that* $B(x)$ *is an h-set for every* $x : |A|$*.*

Combining the first and third data, we may define the type DAlgebra of displayed algebras as pairs of a family of h-sets $B : |A| \to \mathsf{HSet}$ and a function $b : (a : A) \to B\ [a]$.

$$\begin{aligned}&\mathsf{DAlgebra} : \mathbf{U} \to \mathbf{U}\\ &\mathsf{DAlgebra}\ A = \textstyle\sum_{B:|A|\to\mathsf{HSet}} (a : A) \to B\ [a]\end{aligned}$$

Given a displayed algebra $(B, b) : \mathsf{DAlgebra}\ A$, we define displayed algebra homomorphisms from $|A|$ to (B, b) as functions $f : (x : |A|) \to B(x)$ that send $[a]$ to $b\ a$. (The requirement that f send **trunc** to β is automatic because the type of β is a proposition.)

$$\begin{aligned}&\mathsf{DAlgebraHom} : (A : \mathbf{U}) \to \mathsf{DAlgebra}\ A \to \mathbf{U}\\ &\mathsf{DAlgebraHom}\ A\ (B, b) = \textstyle\sum_{f:(x:|A|)\to B(x)} \mathbf{Id}((a : A) \to B\ [a], b, \lambda a \to f\ [a])\end{aligned}$$

In Section 2.5.4, the elimination principle for **Nat** stated that for every displayed algebra (A, a_z, a_s) over $(\mathbf{Nat}, \mathbf{zero}, \mathbf{suc}(-))$ there is a unique displayed algebra homomorphism from the latter to the former. The analogous condition here is that the type of such displayed algebra homomorphisms is contractible.

$$\mathsf{Elimination}_{|-|} : \{A : \mathbf{U}\} \to (\tilde{B} : \mathsf{DAlgebra}\ A) \to \mathsf{IsContr}(\mathsf{DAlgebraHom}\ A\ \tilde{B})$$

Another way to phrase the elimination principle is to ask that for all $B : |A| \to \mathsf{HSet}$, precomposition by $[-]$ determines an equivalence:

$$\begin{aligned}&\mathsf{comp} : \{A : \mathbf{U}\} \to (B : |A| \to \mathsf{HSet}) \to ((x : |A|) \to B\ x) \to ((a : A) \to B\ [a])\\ &\mathsf{comp}\ B\ f\ a = f\ [a]\end{aligned}$$

$$\mathsf{Elimination}'_{|-|} : \{A : \mathbf{U}\} \to (B : |A| \to \mathsf{HSet}) \to \mathsf{IsEquiv}(\mathsf{comp}\ B)$$

In Section 2.7.3 we said that the propositional truncation of A is the proposition that most closely approximates the type A. As one might expect, the set truncation of A is likewise the *h-set* that most closely approximates A.

Theorem 5.2.27. *For all* $A : \mathbf{U}$ *and* $B : \mathsf{HSet}$*,* $(A \to B) \simeq (|A| \to B)$*.*

Proof. This is exactly the non-dependent case of $\mathsf{Elimination}'_{|-|}$. □

Propositional truncation and set truncation are the first two in a series of *n-truncation* operations that best approximate A by an n-type. In practice, propositional truncation ((-1)-truncation) and set truncation (0-truncation) come up most often.

5.2.4 *Applications of homotopy type theory*

Higher inductive types and univalent universes (of non-propositions) are rather exotic features for which many users of type theory lack intuition, so it is reasonable to wonder how they came about and what makes them interesting. Unfortunately, an early attempt to motivate univalence proved too much of a digression to include in this book, so instead we close this section by outlining three major applications of homotopy type theory in an attempt to convey a sense of the field.

Synthetic homotopy theory One of the earliest recognized applications of HoTT was as a framework for reasoning *synthetically* or axiomatically about homotopy types, as opposed to the traditional practice of working with respect to a particular "analytic" realization of homotopy types as topological spaces, Kan complexes, etc. To this end, it is natural to wonder how many classical results about homotopy types may be replayed within HoTT, starting with foundational results from algebraic topology (a field which classifies topological spaces by means of algebraic invariants).

In algebraic topology, the *fundamental group* $\pi_1(X, x)$ of the space X based at $x \in X$ is the set of homotopy equivalence classes of loops based at x, with loop concatenation as its multiplication. Within type theory, we can define the fundamental group of $X : \mathbf{U}$ based at $x : X$ as the set truncation of the space of identifications of x with itself, i.e., $\pi_1(X, x) = |\mathbf{Id}(X, x, x)|$. Armed with this definition, a more complex version of our proof that the circle is not an h-set (Lemma 5.2.24) establishes a standard result:

Theorem 5.2.28 (Licata and Shulman [LS13])**.** *The fundamental group of the circle,* $\pi_1(\mathsf{S}^1, \mathsf{pt})$*, is isomorphic to the integers under addition.*

In other words, not only is loop distinct from **refl**, but (up to identification) the type $\mathbf{Id}(\mathsf{S}^1, \mathsf{pt}, \mathsf{pt})$ consists precisely of all the n-fold compositions of loop, i.e., the n-fold applications of trans to loop and sym loop.

It is worth noting that the HoTT proof of Theorem 5.2.28 is very different from the standard proof one might encounter in an algebraic topology course, as it relies on a univalent universe and makes no reference to real numbers or even topology (e.g., the Lebesgue covering lemma). This is to be expected, as constructions inside HoTT are valid in any Grothendieck ∞-topos and not just topological spaces [Shu19].

Descent and colimits The role of univalent universes in characterizations of higher inductive types such as Theorem 5.2.28 is part of the higher-categorical phenomenon of *descent*, in which universes improve the behavior of colimits.

In topology, space-indexed families (such as vector bundles or covering spaces) are typically encoded as a "total space" Y equipped with a continuous projection map $\pi : Y \to X$ to the indexing space X, where the value of the family at $x \in X$ is recovered as the preimage $\pi^{-1}(x)$. This suggests that there are two ways to encode X-indexed families of types in type theory: as the usual maps into the universe $X \to \mathbf{U}$ or as maps into X, $\mathsf{Fam}(X) = \sum_{Y:\mathbf{U}}(Y \to X)$. Univalence implies that these definitions agree in the sense that $\mathsf{Fam}(X) \simeq (X \to \mathbf{U})$, and in fact univalence is interprovable with this equivalence (along with function extensionality and propositional univalence).

In the case of $X = \mathsf{S}^1$, by combining the above equivalence with the elimination principle for the circle at $\mathbf{U}$, we can prove that S^1 is equivalent to what homotopy theorists know as the *classifying space of* $\mathbb{Z}$*-torsors*.

Theorem 5.2.29. *There is an equivalence* $(\sum_{Y:\mathbf{U}} Y \to S^1) \simeq (\sum_{Y_0:\mathbf{U}} Y_0 \simeq Y_0)$.

More generally, this equivalence allows us to characterize maps *into* homotopy colimits—types with mapping out properties—by transforming them into maps *out* of those types and applying the elimination principle specialized to a univalent universe.

Some descent-type theorems hold in the absence of univalence. For instance, our proof that universes imply the disjointness of booleans (Theorem 2.6.3) is descent for coproducts, which holds in ETT (and sets). However, it is only in HoTT (and ∞-topoi) that descent holds for *all* colimits such as S^1, the homotopy pushout $\mathbf{1} \leftarrow (\mathbf{1} \sqcup \mathbf{1}) \to \mathbf{1}$.

Structure Identity Principle A third and perhaps more familiar application of univalence is the *structure identity principle* (SIP), which states that identifications of structured types are equivalent to structure-preserving equivalences [CD13].

Suppose we define the type of *monoids* in HoTT in the usual way, as tuples of an h-set X along with an identity element $e : X$, a binary multiplication $_\cdot_ : X \to X \to X$, and proofs that the multiplication is unital and associative. (We require that the carrier is an h-set to ensure that the identifications are propositions and not interesting data.)

$$\begin{aligned}\mathsf{Mon} = \sum_{X:\mathsf{HSet}} \sum_{e:X} \sum_{_\cdot_:X\to X\to X} &((x : X) \to \mathbf{Id}(X, x \cdot e, x) \times \mathbf{Id}(X, e \cdot x, x)) \times \\ &((x\ y\ z : X) \to \mathbf{Id}(X, x \cdot (y \cdot z), (x \cdot y) \cdot z))\end{aligned}$$

Remarkably, a direct consequence of univalence (as well as some standard **Id**-type manipulations such as Lemma 5.2.12) is that identifications of monoids are equivalent to monoid isomorphisms. In other words, the type theory has somehow "predicted" the correct notion of sameness for monoids!

Theorem 5.2.30. *For all* $X, Y : \mathsf{Mon}$*, the type of identifications* $\mathbf{Id}(\mathsf{Mon}, X, Y)$ *is equivalent to the type of* monoid isomorphisms *between* X *and* Y*, or equivalences of the carriers* $\mathbf{fst}(X) \simeq \mathbf{fst}(Y)$ *sending the unit and multiplication of* X *to those of* Y*.*

As a consequence, because all constructions within type theory respect identification, all constructions on monoids automatically transfer across monoid isomorphisms.

Corollary 5.2.31. *Any property of monoids* $P : \mathsf{Mon} \to \mathbf{U}$ *respects monoid isomorphism.*

Although we have illustrated the SIP as it applies to monoids, analogous statements hold not only for all algebraic structures on h-sets but even for (higher) categories and many other structures up to the appropriate notion of equivalence [Ahr+25].

5.3 *Cubical type theory* (DRAFT)

Thus far in this chapter, we have introduced the univalence axiom and studied a few of its consequences. Hopefully the reader has been convinced that this is an interesting principle with which to extend type theory and that it at least offers partial compensation for the loss of the extensional equality type. However, so far we have considered only the extension of ITT by a simple axiom to obtain univalence and, consequently, the resulting theory does not satisfy canonicity.

In particular, it is not difficult to encounter interesting closed elements of type **Nat** which are constructed via univalence, but in core HoTT these programs cannot be evaluated to closed numerals. Famously, Brunerie [Bru18] gave a concise construction of an element of the type $\sum_{n:\mathbf{Nat}} \pi_4(S^3) = \mathbb{Z}/n\mathbb{Z}$ but the lack of canonicity meant that actually working out the concrete $n : \mathbf{Nat}$ for which this equation held was considerably more difficult [Bru16]. This is far from the only example: the proof that $\pi_1(S^1) \simeq \mathbb{Z}$ referenced in Section 5.2 ought to give an algorithm for computing the *winding number* of a map $S^1 \to S^1$, but this algorithm can only be run if canonicity holds.

Remark 5.3.1. In fact, in Section 5.2 we assumed function extensionality along with univalence. A more careful account would allow us to derive the former from the latter and in fact our solution to canonicity and univalence will handle funext en passant. ◇

At first, one might hope that this problem can be fixed "locally" and that one can simply add a definitional equality to ua to recover canonicity. Unfortunately, no such obvious equalities present themselves. A moment's contemplation will reveal how while there is a reasonable candidate for coe applied to $\mathsf{ua}(\dots)$, the general case of **j** and ua is far murkier; such an equation must correctly handle, for instance, the

application of sym and trans to ua along with any other number of constructions. More generally, we justified our definition of **Id** around the idea that every element of $\mathbf{Id}(A, a, b)$ was controlled by **refl**, but this is simply no longer the case in the presence of ua.

Accordingly, our approach to balancing canonicity alongside univalence will involve a more global and radical change. We shall reimagine the intensional identity type in order to give it a new mapping-in property by adding additional judgmental structure. This new identity type will give us the flexibility we need to implement univalence. The result of these changes will be cubical type theory [CCHM18; AFH18; Ang+21].

Unfortunately, cubical type theory is vastly more complex than any other type theory we have discussed in this book. Accordingly, we cannot realistically present it in the same detail that we have given to ETT or ITT. Our compromise is to introduce what we term *core cubical type theory* in this section. We detail the required modifications to the judgmental structure of type theory, present the additional operations necessary to manipulate them, and sketch how these operations behave and can be used to implement univalence. The last step, however, will mostly be cursory and we will omit most of the rules governing these operations. We do, however, return to them in Section 5.4 where we discuss some of these details more thoroughly (though still not exhaustively). Our goal is to provide a working knowledge of cubical type theory rather than a precise account. For the latter, we refer the reader to Angiuli et al. [Ang+21] which does include a more exhaustive account of the theory.

The basis of cubical type theory In this section, we discuss the rules that must be added to intensional type theory in order to arrive at cubical type theory. For concreteness, we will take our base type theory to type theory without any sort of identity type. It is possible to include the intensional identity type as it is possible to extend cubical type theory with indexed inductive types more generally. However, we shall set about to find a better behaved identity type (*path types*) and so its inclusion is superfluous.

5.3.1 *A judgmental structure for identity types*

We begin by convincing ourselves that the judgmental structure of cubical type theory is, in fact, helpful for our problem of giving the identity type a mapping-in property. We begin by observing that we have already attempted to provide an identity type with such a characterization: this was the extensional identity type we moved away from in Chapter 4. There is not an obvious alternative judgmental structure in intensional

type theory for the identity type to internalize, so we shall invent one.

This entire process will be broken up into two steps:

1. introduce a new form of judgment and define the new identity type to internalize it,

2. equip each type with additional operations such that this new identity type can implement the expected operations.

We shall eventually see that the first step occupies our attention in Section 5.3.2, while the second takes up Sections 5.3.4 and 5.3.6

It is notable that these two steps are actually distinct: with both **Eq** and **Id**, once we fixed the judgmental structure we intended to internalize all the rules of the identity type came more-or-less for free. In fact, the same will be true here: the second step does not alter the behavior of the identity type per se. The issue is that the judgmental structure being internalized is no longer definitional equality and so we must add additional structure to all types in order to ensure that this new structure is a useful approximation of equality.

More heuristically, we cannot internalize actual judgmental equality via a mapping-in property and so we internalize a new judgmental structure for *identifications*. We then attempt to paper over the difference between these new judgmental identifications and actual definitional equality by equipping every single type with additional operations ensuring the former is closer to the latter.

Notation 5.3.2. With an eye towards cubical type theory, we will refer to our new identity type as a *path type* and write $\mathbf{Path}(A, a, b)$ and occasionally refer to identifications as *paths*.

In particular, it is only after both steps are completed that we will have a replacement for **Id** that we can contemplate using for univalence. There is a degree of flexibility in how we draw the line between these two steps in cubical type theory. We can make the judgmental structure relatively light-weight by making the operations on types more onerous or vice versa. This division is the source of the differences between the various flavors of cubical type theory, but overall the differences are slight. We will choose to follow Angiuli et al. [Ang+21] and adopt a relatively minimal judgmental structure at the expense of slightly more complex operations on types.

Let us warm up by considering a direct approach following Licata and Harper [LH12] loosely. We need a new judgment to internalize identifications, so let us simply introduce a new sort of *identifications* α, β, γ which reify identifications and a new judgment $\Gamma \vdash \alpha : a = b : A$ stating that α is such an identification between $a, b : A$. As usual, we will write $\mathsf{Id}(\Gamma, a, b, A)$ for the set of α satisfying $\Gamma \vdash \alpha : a = b : A$. The

idea is that we can now at least easily define $\mathbf{Path}(A, a, b)$ via the following natural isomorphism:

$$\mathsf{Tm}(\Gamma, \mathbf{Path}(A, a, b)) \cong \mathsf{Id}(\Gamma, a, b, A)$$

This completes our goal of defining $\mathbf{Path}(A, a, b)$ and it yields all the necessary rules for this type. The reader will immediately notice, however, that this type is impossible to use and absolutely not a substitute for the identity type. Indeed, just because we claimed that $\Gamma \vdash \alpha : a = b : A$ reifies identifications does nothing to actually force α to behave like any sort of equality. We have only shifted the work into specifying this judgment. For instance, we might choose to include a "reflexivity identification" via the following rule:

$$\frac{\Gamma \vdash a = b : A}{\Gamma \vdash \mathbf{reflId} : a = b : A}$$ ✎

Of course, this cannot be the only rule governing our new judgment; the point of this exercise was to allow for additional identifications (such as ua) to arise naturally. In order to do this, we can simply add other inference rules to this judgment! While we do not detail the process here, the reader can imagine that *e.g.*, an identification of pairs can be constructed from identifications of components.

These rules ensure that we can construct elements of $\mathbf{Path}(A, a, b)$, but do not actually give us much leverage in *using* elements of this new type. Our elimination rule for $\mathbf{Path}(A, a, b)$ lets us conclude that there is some identification between a and b, but this is of limited use: there is nothing like **j** for identifications or even the equivalent of subst.

When identity types internalized definitional equality, we relied on the fact that everything in type theory was automatically congruent and substitutive with respect to definitional equality. Now we are internalizing *identifications* and nothing forces types in our theory to respect identifications in the same way. This is what the second step of the procedure above referred to: we will require additional operations on terms to bridge this gap. For instance, for each type family $\Gamma.A \vdash B$ type there must be an equivalent of subst which sends identifications $\Gamma \vdash \alpha : a = b : A$ to maps between $B(a)$ and $B(b)$.

However, we will not attempt to unfold this further. The problem is that it is difficult to present the full set of rules governing $\Gamma \vdash \alpha : a = b : A$ as well as to present the set of operations all types must enjoy in order to force them to respect identifications. The first problem is the most serious and stems from our desire to support univalence. If we are to have univalence, then we know that there will be elements $a, b : A$ such that the collection of identifications between a and b contains distinct elements and, accordingly, we will quickly run into the need for non-trivial identifications between identifications.

In fact, one can imagine these arising even without univalence: we had discussed that a pair of identifications $\Gamma \vdash \alpha : a = a' : A$ and $\Gamma \vdash b : b' = B :$ ought to induce an identification $\Gamma \vdash (\alpha, \beta) : \mathbf{pair}(a, b) = \mathbf{pair}(a', b') : A \times B$ and we ought to arrange that $(\mathbf{reflId}, \mathbf{reflId}) = \mathbf{reflId}$. To properly account for this and other "higher identifications", we are quickly led to introducing a new judgment for governing identifications between identifications. As the reader might guess, however, the problem does not stop here and we require judgments for identifications between identifications between identifications... This infinite regress then becomes plain. Accordingly, rather than special-casing a judgment for identifications between terms, we shall design an apparatus which smoothly handles identifications of arbitrary "height".

It is here that we encounter cubes for the first time. Cubical type theory starts from the insight that an identification between $a, b : A$ can be viewed as a function $\mathbb{I} \to A$ from some "type" $\mathbb{I}$. Since we already have a good idea of how to account for functions within the judgments of type theory, if we could recast identity types more into the shape of functions we could reuse this knowledge.

Of course, it is not obvious that functions and identity types share much in common.[5] A small amount of topological intuition can help motivate this approach: we can say that two points in a space $x, y \in X$ are path-connected just when there is a continuous function from the real interval $p : [0, 1] \to X$ such that $p(0) = x$ and $p(1) = y$. The geometry of $[0, 1]$ ensures that this notion of identification is actually an equivalence relation. For instance, transitivity comes from the map $[0, 1] \to [0, 1] \vee [0, 1]$ dividing the interval into two halves and the continuity of $1 - x$ provides symmetry. A major advantage of this definition is that it stacks to identifications between identifications without additional effort: we just take functions from $[0, 1] \times [0, 1]$ satisfying the relevant boundary conditions.

Of course, we have nothing like the real interval in type theory, nor do we intend to add it. However, we can add a judgmental structure which simulates some of its properties and use this as the basis for our definition of an identification in a type. We shall add a faux type $\mathbb{I}$ to our theory and extend our grammar of context to hypothesize over "variables" of type $\mathbb{I}$ such that an identification is then just a term in a context containing such an interval variable.

Remark 5.3.3. It is not yet clear why $\mathbb{I}$ must be a separate structure rather than an ordinary type. Indeed, this is a subtle point and relates to the additional operations necessary to implement the equivalent **j** and its related operations. In fact, we shall see that $\mathbb{I}$ cannot support these operations and so it cannot be a type. However, in all other respects it *does* behave like a type: we shall see that the substitution calculus rules for $\mathbb{I}$ as well as the rules for the elements of $\mathbb{I}$ are essentially the same as for ordinary types. For this reason, one often refers to $\mathbb{I}$ as a *pre-type*. ◇

[5] We have already seen hints of this in Section 5.2.3 with the higher-inductive type for the interval

5.3.2 The interval and its structure

Let us make this discussion more formal. We introduce a new judgment $\Gamma \vdash r : \mathbb{I}$ which signifies that r is an element of this interval "pre-type" and has the presupposition $\vdash \Gamma\,\mathsf{cx}$. We then introduce a new form of context stating that one may hypothesize over elements of $\mathbb{I}$. All told, the rules for this are given as follows:

$$\frac{\vdash \Gamma\,\mathsf{cx}}{\vdash \Gamma.\mathbb{I}\,\mathsf{cx}} \qquad \frac{\vdash \Gamma\,\mathsf{cx}}{\Gamma.\mathbb{I} \vdash \mathsf{p} : \Gamma} \qquad \frac{\vdash \Gamma\,\mathsf{cx}}{\Gamma.\mathbb{I} \vdash \mathsf{q} : \mathbb{I}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash r : \mathbb{I}}{\Delta \vdash r[\gamma] : \mathbb{I}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Delta \vdash r : \mathbb{I}}{\Delta \vdash \mathsf{p} \circ \gamma.r = \gamma : \Gamma} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Delta \vdash r : \mathbb{I}}{\Delta \vdash \mathsf{q}[\gamma.r] = r : \mathbb{I}} \qquad \frac{\Delta \vdash \gamma : \Gamma.\mathbb{I}}{\Delta \vdash \gamma = (\mathsf{p} \circ \gamma).\mathsf{q}[\gamma] : \Gamma.\mathbb{I}}$$

$$\frac{\Gamma_1 \vdash \gamma_2 : \Gamma_2 \qquad \Gamma_0 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_2 \vdash r : \mathbb{I}}{\Gamma_0 \vdash r[\gamma_2 \circ \gamma_1] = r[\gamma_2][\gamma_1] : \mathbb{I}} \qquad \frac{\Gamma \vdash r : \mathbb{I}}{\Gamma \vdash r[\mathbf{id}] = r : \mathbb{I}}$$

We call $\Gamma \vdash r : \mathbb{I}$ *dimension terms* and q *dimension variables*.

Notation 5.3.4. We write $\gamma.\mathbb{I}$ for the analogous substitution to $\gamma.A$.

Exercise 5.17. Define $\mathsf{Int}(\Gamma)$ to be the set $\{r \mid \Gamma \vdash r : \mathbb{I}\}$. Rephrase the above substitution rules and equalities using $\mathsf{Int}(\Gamma)$ and, in particular, isolate a mapping-in property for $\Gamma.\mathbb{I}$.

Notation 5.3.5. The reader will notice that while context extension with an interval is formally distinct from $\Gamma.A$, the substitution calculus is the same around both. Consequently, it is not difficult to adapt the translation from named variables to formal syntax with explicit substitutions to account for interval "variables". When we write informal programs in cubical type theory, we shall therefore use essentially the same notation for variables of $\mathbb{I}$ as we have for variables of a type A. By convention, we shall use the letters i, j, k for these *dimension variables*.

All told then, $\mathbb{I}$ has been added to our theory such that it behaves more-or-less like a type without any introduction or elimination rules and we can only hypothesize variables of type $\mathbb{I}$ and pass them around. This is far less structure than the real interval $[0, 1]$, but it is already almost enough to realize our judgmental structure for identifications: an identification in A can simply be taken as an element $\Gamma.\mathbb{I} \vdash a : A[\mathsf{p}]$. What we are missing is some means of stating what, precisely, is being identified by such an a. In the topological case, an identification was a continuous function $p : [0, 1] \to X$ which identified $p(0)$ with $p(1)$. Accordingly, we now augment $\mathbb{I}$ with

two closed dimension terms $0, 1$ and understand $\Gamma.\mathbb{I} \vdash a : A[\mathbb{p}]$ to be identifying $a[\mathbf{id}.0]$ and $a[\mathbf{id}.1]$.

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash 0, 1 : \mathbb{I}}$$

Lemma 5.3.6. *For every $\Gamma \vdash a : A$ there is an identification of a with itself.*

Proof. Given such an a, the term $p = a[\mathbb{p}]$ is an element of $\Gamma.\mathbb{I} \vdash A[\mathbb{p}]$ type and it is routine to check that $p[\mathbf{id}.0] = a = p[\mathbf{id}.1]$. □

We have used $\mathbb{I}$ to recover the bespoke identification judgment from before and in a less ad-hoc manner. Just as before, we may define a path type to internalize this new structure directly:

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash a, b : A}{\Gamma \vdash \mathbf{Path}(A, a, b)\ \mathsf{type}} \qquad \frac{\Gamma.\mathbb{I} \vdash p : A[\mathbb{p}]}{\Gamma \vdash \lambda(p) : \mathbf{Path}(A, p[\mathbf{id}.0], p[\mathbf{id}.1])}$$

$$\frac{\Gamma \vdash p : \mathbf{Path}(A, a, b) \qquad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash \mathbf{papp}(p, r) : A}$$

$$\frac{\Gamma \vdash p : \mathbf{Path}(A, a, b)}{\Gamma \vdash \mathbf{papp}(p, 0) = a : A \qquad \Gamma \vdash \mathbf{papp}(p, 1) = b : A}$$

$$\frac{\Gamma.\mathbb{I} \vdash p : A[\mathbb{p}] \qquad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash \mathbf{papp}(\lambda(p), r) = p[\mathbf{id}.r] : A} \qquad \frac{\Gamma \vdash p : \mathbf{Path}(A, a, b)}{\Gamma \vdash \lambda(\mathbf{papp}(p[\mathbb{p}], \mathbb{q})) = p : \mathbf{Path}(A, a, b)}$$

Notice that, unlike an ordinary function type, a path type specifies the behavior of its elements on 0 and 1. In particular if p is an element of $\mathbf{Path}(A, a, b)$ then we have definitional equalities $\mathbf{papp}(p, 0) = a$ and $\mathbf{papp}(p, 1) = b$ to enforce the intuition that p is a path from a to b. These equations are justified by the introduction rule which requires additional *boundary conditions* ensuring that elements of $\mathbf{Path}(A, a, b)$ correspond not just to arbitrary elements of A depending on $\mathbb{I}$ but to elements which satisfy the necessary equations.

Exercise 5.18. Show that the above rules are precisely equivalent to requiring the following mapping-in property for $\mathbf{Path}(A, a, b)$:

$$\mathsf{Tm}(\Gamma, \mathbf{Path}(A, a, b)) \cong \{p \in \mathsf{Tm}(\Gamma.\mathbb{I}, A[\mathbb{p}]) \mid p[\mathbf{id}.0] = a \wedge p[\mathbf{id}.1] = b\}$$

Exercise 5.19. Use Lemma 5.3.6 to define an element $\mathsf{refl}(a) : \mathbf{Path}(A, a, a)$ for every $\Gamma \vdash a : A$.

Notation 5.3.7. For expository purposes, it is also helpful to have a type $\mathbf{\Pi}(\mathbb{I}, A)$ with the following mapping-in property:

$$\mathsf{Tm}(\Gamma, \mathbf{\Pi}(\mathbb{I}, A)) \cong \mathsf{Tm}(\Gamma.\mathbb{I}, A)$$

We shall not regard this as part of our official definition of cubical type theory and use it only for small informal examples. In these few occurrences of this "Π-type", we shall use the ordinary syntax for functions, using the observation above that we can translate "named interval variables" into the formal substitution calculus for $\mathbb{I}$.

5.3.3 *Cultivating intuition for path types*

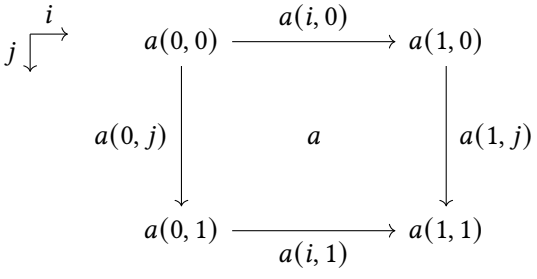
Before proceeding to the other rules of cubical type theory, we take a moment to explore the consequences of including the interval within type theory. For this, and in cubical type theory more generally, it is helpful to use a small amount of topological intuition, guided by the observation that a term $\mathbf{1}.\mathbb{I}\ldots.\mathbb{I} \vdash a : A[\mathbb{p}^n]$ which depends on n copies of $\mathbb{I}$ can be visualized as an n-dimensional cube in A. In low dimension, we therefore have points in A, lines in A, squares in A, and cubes in A for $n = 0, 1, 2, 3$ respectively. Let us illustrate the case where $n = 2$ more thoroughly. Given $\mathbf{1}.\mathbb{I}.\mathbb{I} \vdash a : A[\mathbb{p}^2]$, the two dimension variables serve as "axes" for this square and so we can "draw" a as the following square:

$$\begin{array}{ccc} a[\mathbf{id}.0.0] & \xrightarrow{a[\mathbf{id}.0]} & a[\mathbf{id}.1.0] \\ {\scriptstyle a[\mathbf{id}.0.\mathbb{I}]}\downarrow & a & \downarrow{\scriptstyle a[\mathbf{id}.1.\mathbb{I}]} \\ a[\mathbf{id}.0.1] & \xrightarrow[a[\mathbf{id}.1]]{} & a[\mathbf{id}.1.1] \end{array}$$

The four closed terms one obtains by specializing a with the four substitutions $\mathbf{1} \vdash \mathbf{id}.\epsilon.\epsilon' : \mathbf{1}.\mathbb{I}.\mathbb{I}$ are the vertices. Next, there are four substitutions from $\mathbf{1}.\mathbb{I}$ to $\mathbf{1}.\mathbb{I}.\mathbb{I}$ that instantiate the first or second $\mathbb{I}$ with a constant and the other $\mathbb{I}$ with $\mathbb{q}$. Applying each of these substitutions to a yields the edges of the square. Finally, a itself is the entire square.

We have chosen to draw this square with the leftmost $\mathbb{I}$ in $\mathbf{1}.\mathbb{I}.\mathbb{I}$ as the horizontal axis and the rightmost as the vertical axis. We further oriented the horizontal axis to grow to

the right and the vertical axis to grow down. This convention is reasonably standard—it matches the typical orientation of commutative diagrams in category theory—but it is often helpful to disambiguate these diagrams by using named variables and labeling axes. For instance, we might have written $i : \mathbb{I}, j : \mathbb{I} \vdash a(i, j) : A$ and depicted the above square as follows:



Remark 5.3.8. We note that in the above example we have assumed that A does not depend on either dimension variable but this restriction is not mandatory. We will have occasion to study such *heterogeneous* squares at various points. ◇

This schematic visualization highlights one of the major benefits of using $\mathbb{I}$ to structure identifications compared to a direct judgment $\Gamma \vdash \alpha : a = b : A$: we can now seamlessly account for identifications between identifications simply by adding more than one copy of $\mathbb{I}$ to the context. Moreover, path types between path types of A are really no more complex to manipulate than ordinary path types as both are simply kinds of functions valued in A.

There is another major benefit to using $\mathbb{I}$: we have no need to add further rules of $\mathbb{I}$ to customize the behavior of path types in each connective. For instance, there is no need for a rule that "identifications in a pair can be built from a pair of identifications". This fact is already derivable from those rules governing dependent sums generally. In fact, path types enjoy a number of remarkable extensionality principles (including function extensionality) without additional effort on our part.

This traces back to a subtle point: when we isolated identifications as a new judgment, nothing connected it to the behavior of types or terms. Here, however, we have smuggled identifications in through the existing apparatus of contexts and substitutions and so the existing equations for types and terms automatically apply to identifications.

For instance, the η law for dependent sums states that $\mathsf{Tm}(\Gamma, \Sigma(A, B))$ is isomorphic to $\sum_{a \in \mathsf{Tm}(\Gamma, A)} \mathsf{Tm}(\Gamma, B[\mathbf{id}.a])$. If we choose $\Gamma = \Gamma_0 \mathbb{I}$ and specialize to the case where $B = B_0[\mathbf{p}]$ for simplicity, this immediately yields the following:

Lemma 5.3.9. *There is a natural bijection of the following shape:*

$$\begin{aligned}&\mathsf{Tm}(\Gamma_0, \mathbf{Path}(\Sigma(A, B_0[\mathbf{p}]), x, y))\\ &\qquad \cong \mathsf{Tm}(\Gamma_0, \mathbf{Path}(A, \mathbf{fst}(x), \mathbf{fst}(y))) \times \mathsf{Tm}(\Gamma_0, \mathbf{Path}(B_0, \mathbf{snd}(x), \mathbf{snd}(y)))\end{aligned}$$

Exercise 5.20. Prove Lemma 5.3.9

Note that while we have specialized to the simpler case of non-dependent Σ-types, it is only for notational convenience. Even more striking is the case for dependent products.

Lemma 5.3.10. *There is a natural bijection of the following shape:*

$$\mathsf{Tm}(\Gamma, \mathbf{Path}(\mathbf{\Pi}(A, B), f, g)) \cong \mathsf{Tm}(\Gamma.A, \mathbf{Path}(B, \mathbf{app}(f[\mathbf{p}], \mathbf{q}), \mathbf{app}(g[\mathbf{p}], \mathbf{q})))$$

In other words, function extensionality is automatically true for path types.

Proof. Let us begin by observing that, by the mapping-in property of path types, we can rephrase our goal as the following:

$$\{p \in \mathsf{Tm}(\Gamma.\mathbb{I}, \mathbf{\Pi}(A, B)[\mathbb{p}]) \mid \dots\} \cong \{p' \in \mathsf{Tm}(\Gamma.A.\mathbb{I}, B[\mathbb{p}]) \mid \dots\}$$

However, we can further apply the mapping-in property for Π-types to replace the left-hand set with $\{p \in \mathsf{Tm}(\Gamma.\mathbb{I}.A[\mathbb{p}], B[\mathbb{p}.A]) \mid \dots\}$. The conclusion then follows immediately from the isomorphism of contexts $\Gamma.\mathbb{I}.A[\mathbb{p}] \cong \Gamma.A.\mathbb{I}$ (Exercise 5.21). □

Exercise 5.21. Prove that if $\Gamma \vdash A$ type then there are mutually inverse substitutions $\Gamma.\mathbb{I}.A[\mathbb{p}] \vdash \tau_0 : \Gamma.A.\mathbb{I}$ and $\Gamma.A.\mathbb{I} \vdash \tau_1 : \Gamma.\mathbb{I}.A[\mathbb{p}]$.

This has certainly improved on our earlier attempt which simply added a new explicit judgment of identifications but the story cannot stop here. In particular, we still have done nothing to address the link between $\mathbf{Path}(A, a, b)$ and the actual ability to substitute a for b in a type. That is, we have no operation like that of subst or, more generally, **j**. As mentioned earlier, these operations do not come directly from the interval or judgments upon it. Instead, we shall add them more-or-less as constants to our theory and then, to preserve canonicity, add type-specific equations telling us how they compute.

5.3.4 Coercing along paths

We now introduce the first and most fundamental operation of the two operations we shall add to cubical type theory: $\mathbf{coe}_A$ (short for *coerce*). Roughly, this operation ensures

that, from the perspective of a type, all elements of the interval are interchangeable and we shall see momentarily that this is precisely what is required to implement a version of subst for $\mathbf{Path}(A, a, b)$.

The addition of $\mathbf{coe}_A$ also means a change in the status of $\mathbb{I}$ in our type theory. While we have not added any sort of elimination principle for $\mathbb{I}$, the reader may have noticed that up till this point there was really nothing which distinguished it from **Bool**; the rules we required of $\mathbb{I}$ were a strict subset of those for **Bool**. The coercion operation firmly rules out the possibility that $\mathbb{I} = \mathbf{Bool}$: a type depending on **Bool** can be quite different over **true** and **false** which is precisely the possibility excluded by **coe**.

Specifically, if $\Gamma.\mathbb{I} \vdash A\,\mathsf{type}$ then $A[\mathbf{id}.r]$ and $A[\mathbf{id}.s]$ are equivalent for every $\Gamma \vdash r, s : \mathbb{I}$. The typing rule for this constant is given as follows:

$$\frac{\Gamma.\mathbb{I} \vdash A\,\mathsf{type} \qquad \Gamma \vdash r, s : \mathbb{I} \qquad \Gamma \vdash a : A[\mathbf{id}.r]}{\Gamma \vdash \mathbf{coe}_A^{r \to s}(a) : A[\mathbf{id}.s]}$$

$$\frac{\Gamma.\mathbb{I} \vdash A\,\mathsf{type} \qquad \Gamma \vdash r : \mathbb{I} \qquad \Gamma \vdash a : A[\mathbf{id}.r]}{\Gamma \vdash \mathbf{coe}_A^{r \to r}(a) = a : A[\mathbf{id}.r]}$$

A priori, **coe** may seem as though it does little to advance our goal of implementing subst for $\mathbf{Path}(A, a, b)$. However, suppose we are given a path $\Gamma \vdash p : \mathbf{Path}(A, a, b)$ along with a type $\Gamma.A \vdash C\,\mathsf{type}$, applying the ordinary rule for substitution, we obtain $\Gamma.\mathbb{I} \vdash C' = C[\mathbb{p}.\mathbf{papp}(p[\mathbb{p}], \mathbb{q})]\,\mathsf{type}$. Inspection reveals that instantiating C' at 0 and 1 yields $C[\mathbf{id}.a]$ and $C[\mathbf{id}.b]$ and so **coe** yields the following operation:

$$\Gamma \vdash \lambda(\mathbf{coe}^{0 \to 1}_{C'[\mathbf{p}.\mathbb{I}]}(\mathbf{q})) : C[\mathbf{id}.a] \to C[\mathbf{id}.b]$$

In other words, **coe** can be used to define subst. The advantage to **coe** over subst is that we can now set about equipping **coe** with a collection of definitional equalities in order to recover canonicity. Unlike subst, there shall be no single rule for how **coe** computes in general but, instead, $\mathbf{coe}_A$ will compute depending on the form of A. For example, for closed types such as **Nat**, **U**, or **Bool**, we constrain **coe** with the following:

$$\frac{\Gamma \vdash r, s : \mathbb{I} \qquad \Gamma \vdash b : \mathbf{Bool}}{\Gamma \vdash \mathbf{coe}^{r \to s}_{\mathbf{Bool}}(b) = b : \mathbf{Bool}}$$

Of course, this strategy only works in the simplest example: when the type constructor is closed and cannot depend on the interval in any meaningful way. Most commonly, when A is a type former *e.g.* $\Sigma(B_0, B_1)$, $\mathbf{coe}_A$ will be defined in terms of $\mathbf{coe}_{B_i}$. In the case of non-dependent case $A = B_0 \times B_1$, for instance, one must add a

definitional equality stating $\mathbf{coe}^{r\to s}_{A}(a) = \mathbf{pair}(\mathbf{coe}^{r\to s}_{B_0}(\mathbf{fst}(a)), \mathbf{coe}^{r\to s}_{B_1}(\mathbf{snd}(a)))$.[6] In Section 5.4, we shall see that while it is unfeasible to see how univalence ought to compute relative to **j**, it is possible (if difficult) to describe its computation with respect to **coe**.

Our strategy of defining $\mathbf{coe}_A$ in terms of the constituents of A is responsible for another surprising feature of **coe**: if subst is defined by instantiating $r = 0$ and $s = 1$, why do we bother to allow for arbitrary r, s? We shall see that in various situations we require this additional flexibility in order to build up **coe** at more complex types from simpler ones.

We will not detail the equations governing **coe** here, but do provide examples in Section 5.4. Instead, we focus on the equation which leads to the next structure necessary for core cubical type theory: **coe** in $\mathbf{Path}(A, a, b)$. At present, we lack the operations necessary to provide an equation specifying how $\mathbf{coe}^{r\to s}_{\mathbf{Path}(A,a,b)}(p)$ must compute. It is worth sketching the problem informally, so as to properly situate the solution. We wish to formulate a rule of the following shape:

$$\frac{\Gamma_\circ\mathbb{I} \vdash A\,\mathrm{type} \qquad \Gamma_\circ\mathbb{I} \vdash a, b : A \qquad \Gamma \vdash r, s : \mathbb{I} \qquad \Gamma \vdash p : \mathbf{Path}(A, a, b)[\mathbf{id}_\circ r]}{\Gamma \vdash \mathbf{coe}^{r\to s}_{\mathbf{Path}(A,a,b)}(p) = \;?\; : \mathbf{Path}(A, a, b)[\mathbf{id}_\circ s]}$$

This hole must be filled by a path in A built from $\mathbf{coe}_A$. The straightforward approach is roughly to "compose" p (the function from $\mathbb{I}$ to $A[\mathbf{id}_\circ r]$) with $\mathbf{coe}^{r\to s}_{A}(-)$ (a function $A[\mathbf{id}_\circ r] \to A[\mathbf{id}_\circ s]$). However, the resulting term does not satisfy the necessary boundary conditions to be an element of $\mathbf{Path}(A, a, b)[\mathbf{id}_\circ s]$. Instead, we obtain an element of the following:

$$\mathbf{Path}(A[\mathbf{id}_\circ s], \mathbf{coe}^{r\to s}_{A}(a[\mathbf{id}_\circ r]), \mathbf{coe}^{r\to s}_{A}(b[\mathbf{id}_\circ r]))$$

In other words, we are confronted by the fact that while there is a "line" interpolating between *e.g.*, $\mathbf{coe}^{r\to s}_{A}(a[\mathbf{id}_\circ r])$ and $a[\mathbf{id}_\circ s]$, they are not equal. This mismatch is solved by the second operation for manipulating terms depending on $\mathbb{I}$: homogeneous composition or **hcomp**. To a first approximation, this operation allows us to take our collection of three lines and compose them into a single path.

However, while the motivating example given above comes from stitching together three sides of a square into a single line, our need to provide type-specific equations for computing this operation in each type forces us to provide a more general composition operator. In order to properly formulate **hcomp** in Section 5.3.6, we begin by extending the judgmental apparatus with the necessary tools to support it.

[6] This is often expressed by stating that **coe** is defined "by induction" on the type, but this is misleading. After all, types do not come equipped with any sort of induction principle in general!

5.3.5 *Cofibrations and faces*

Let us fix $\mathbf{1}.\mathbb{I}.\mathbb{I} \vdash a : A[\mathbb{p}^2]$ and recall the visualization of a as a square:

$$\begin{array}{ccc} a[\mathbf{id}.0.0] & \xrightarrow{a[\mathbf{id}.0]} & a[\mathbf{id}.1.0] \\ {\scriptstyle a[\mathbf{id}.0.\mathbb{I}]}\downarrow & a & \downarrow{\scriptstyle a[\mathbf{id}.1.\mathbb{I}]} \\ a[\mathbf{id}.0.1] & \xrightarrow[a[\mathbf{id}.1]]{} & a[\mathbf{id}.1.1] \end{array}$$

The edges and vertices in the above square are called the *faces* of a. More generally, a face of a term p is the result from specializing interval variables p depends upon.

The **hcomp** operation which we use to compose paths does so by solving a more general problem. It provides a uniform way to assemble certain collections of matching faces into an entire n-cube. For instance, our earlier desire to combine three lines into a single line can be rephrased into taking three terms representing three faces of a square and extending them to a term representing the entire square.

In general, we should not expect that every matching collection of faces assembles into a cube. For instance, the question of whether a and b are identifiable amounts to asking if a and b are the 0 and 1 faces of some term p. Since we do not expect (or want!) all terms to be identifiable, clearly some subsets of cubes should not always be extendable.

Heuristically, we should be allowed to extend subcubes which are suitably "connected", but this becomes subtle in higher dimensions. As isolating these well-behaved subcubes is complex, it is helpful to have a judgmental apparatus for isolating particular faces of a given term or type. We do this by introducing a special grammar of propositions which we call *cofibrations*. Informally, these are propositions built from (1) comparing dimension terms for equality and (2) conjunction, disjunction, and universal quantification of $\mathbb{I}$. We realize this with a new judgment $\Gamma \vdash \alpha\ \mathsf{cof}$:

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \top, \bot\ \mathsf{cof}} \qquad \frac{\Gamma \vdash \phi, \psi\ \mathsf{cof}}{\Gamma \vdash \phi \wedge \psi, \phi \vee \psi\ \mathsf{cof}} \qquad \frac{\Gamma \vdash r, s : \mathbb{I}}{\Gamma \vdash r = s\ \mathsf{cof}} \qquad \frac{\Gamma.\mathbb{I} \vdash \phi\ \mathsf{cof}}{\Gamma \vdash \forall \phi\ \mathsf{cof}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash \phi\ \mathsf{cof}}{\Delta \vdash \phi[\gamma]\ \mathsf{cof}} \qquad \frac{\Gamma_1 \vdash \gamma_2 : \Gamma_2 \qquad \Gamma_0 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_2 \vdash \phi\ \mathsf{cof}}{\Gamma_0 \vdash \phi[\gamma_2 \circ \gamma_1] = \phi[\gamma_2][\gamma_1]\ \mathsf{cof}}$$

$$\frac{\Gamma \vdash \phi\ \mathsf{cof}}{\Gamma \vdash \phi[\mathbf{id}] = \phi\ \mathsf{cof}}$$

We have omitted the long but unsurprising list of rules shaping how substitutions $\phi[\gamma]$ interact with the various cofibration formers.

In keeping with their obvious relationship to propositions, we add another judgment $\Gamma \vdash \phi\ \mathsf{true}$ which states that some cofibration ϕ holds in context Γ. For instance, we require the following rules:

$$\frac{\Gamma \vdash r = s : \mathbb{I}}{\Gamma \vdash r = s\ \mathsf{true}} \qquad \frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \top\ \mathsf{true}} \qquad \frac{\Gamma \vdash \phi, \psi\ \mathsf{cof} \qquad \Gamma \vdash \phi\ \mathsf{true}}{\Gamma \vdash \phi \vee \psi, \psi \vee \phi\ \mathsf{true}} \qquad \frac{\Gamma \vdash 0 = 1\ \mathsf{true}}{\Gamma \vdash \bot\ \mathsf{true}}$$

$$\frac{\Gamma \vdash \phi\ \mathsf{true} \qquad \Delta \vdash \gamma : \Gamma}{\Delta \vdash \phi[\gamma]\ \mathsf{true}} \qquad \frac{\Gamma \vdash \bot\ \mathsf{true} \qquad \Gamma \vdash \phi\ \mathsf{cof}}{\Gamma \vdash \phi\ \mathsf{true}}$$

In order to give the full set of rules governing $\phi \vee \psi$, we require the ability to hypothesize the truth of a proposition just as we can presently hypothesize over elements of a type. Explicitly, given cofibration $\Gamma \vdash \phi\ \mathsf{cof}$, we also require a context $\Gamma_\circ\phi$ governed by the following rules:

$$\frac{\Gamma \vdash \phi\ \mathsf{cof}}{\vdash \Gamma_\circ\phi\ \mathsf{cx}} \qquad \frac{\Gamma \vdash \phi\ \mathsf{cof}}{\Gamma_\circ\phi \vdash \mathbb{p} : \Gamma} \qquad \frac{\Gamma \vdash \phi\ \mathsf{cof}}{\Gamma_\circ\phi \vdash \phi[\mathbb{p}]\ \mathsf{true}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash \phi\ \mathsf{cof} \qquad \Delta \vdash \phi[\gamma]\ \mathsf{true}}{\Delta \vdash \gamma_\circ\star : \Gamma_\circ\phi} \qquad \frac{\Gamma \vdash \phi\ \mathsf{cof} \qquad \Delta \vdash \gamma : \Gamma_\circ\phi}{\Delta \vdash (\mathbb{p} \circ \gamma)_\circ\star = \gamma : \Gamma_\circ\phi}$$

$$\frac{\Gamma \vdash \phi\ \mathsf{cof}}{\Gamma_\circ\phi \vdash \mathbb{p}_\circ\star = \mathbf{id} : \Gamma_\circ\phi}$$

It is helpful to understand $\Gamma_\circ\phi$ as an analog of $\Gamma.A$ but where ϕ is an exceptionally strict form of proposition rather than a full type. For instance, the substitution extension rule for cofibrations $\gamma_\circ\star$ does not allow the user to supply alternative "proofs" or "terms" witnessing that ϕ is true. Instead, it simply requires that the judgment $\Gamma \vdash \phi\ \mathsf{true}$ holds and uses $\star$. In fact, the user is not responsible for providing any evidence whatsoever in their term that $\Gamma \vdash \phi\ \mathsf{true}$ holds. In this way, the rule is reminiscent of the conversion rule stating that definitionally equal terms may be exchanged without any explicit instruction by the user: cofibrations may be judged true without the user having to provide any explicit witness.

For this reason, it is apparent that we must maintain strict control over the judgment $\Gamma \vdash \phi\ \mathsf{true}$. If this judgment becomes too complex and, for instance, becomes sensitive to what *types* are inhabited in a given context Γ, then it will surely become impossible for our system to enjoy decidable type-checking. Fortunately, the grammar of cofibrations is sufficiently simple that $\Gamma \vdash \phi\ \mathsf{true}$ is, in fact, decidable.

Returning to our specification of $\Gamma \vdash \phi\,\mathsf{true}$, we present the final rule around $\phi \vee \psi$ using $\Gamma \triangleright \phi$:

$$\frac{\Gamma \vdash \phi \vee \psi\,\mathsf{true} \qquad \Gamma \vdash \xi\,\mathsf{cof} \qquad \Gamma \triangleright \phi \vdash \xi[\mathbb{p}]\,\mathsf{true} \qquad \Gamma \triangleright \psi \vdash \xi[\mathbb{p}]\,\mathsf{true}}{\Gamma \vdash \xi\,\mathsf{true}}$$

For brevity, we will not present all the rules of $\Gamma \vdash \phi\,\mathsf{cof}$ and choose to omit *e.g.*, those governing $\phi \wedge \psi$ and $\forall\phi$. The reader may trust that they are unsurprising versions of the ordinary rules for propositional logic. We conclude our selection of the rules for $\Gamma \vdash \phi\,\mathsf{cof}$ with the following pair:

$$\frac{\Gamma \vdash r, s : \mathbb{I} \qquad \Gamma \vdash r = s\,\mathsf{true}}{\Gamma \vdash r = s : \mathbb{I}} \qquad \frac{\Gamma \vdash \phi, \psi\,\mathsf{cof} \qquad \Gamma \triangleright \phi \vdash \psi[\mathbb{p}]\,\mathsf{true} \qquad \Gamma \triangleright \psi \vdash \phi[\mathbb{p}]\,\mathsf{true}}{\Gamma \vdash \phi = \psi\,\mathsf{cof}}$$

The first rule is reminiscent of equality reflection from Chapter 2 and the second is akin to very strong propositional univalence principle for cofibrations. That is, the first rule guarantees that if the proposition $r = s$ holds then this can be 'reflected' to obtain a definitional equality between r and s. The second rule states that cofibrations which are inter-provable are *definitionally* equal such that, *e.g.*, one may silently exchange $\phi \vee \psi$ and $\psi \vee \phi$ in any term or type.

These last two rules imply that the truth of a cofibration can impact whether or not a term or type is well-formed by, for instance, controlling whether two dimension terms are equal. However, we will also add two principles which much more directly allow cofibrations to influence terms, types, and substitutions. Namely, if $\Gamma \vdash \phi \vee \psi\,\mathsf{true}$, we will add a rule stating that to *e.g.*, construct a type in Γ it suffices to give a type A_ϕ under the assumption of ϕ and a second type A_ψ under the assumption of ψ such that $A_\phi = A_\psi$ when $\phi \wedge \psi$ is assumed. We require similar rules for terms and substitutions as well as a twin principle for $\bot$ which simply states that all these judgments collapse if $\Gamma \vdash \bot\,\mathsf{true}$. These rules are designed to ensure that $\Gamma \triangleright \phi \vee \psi$ behaves like the "union" of the contexts $\Gamma \triangleright \phi$ and $\Gamma \triangleright \psi$. For reasons of space, we give the rules carefully for only types and sketch those for terms:

$$\frac{\begin{array}{ccc} & \Gamma \vdash \phi, \psi\,\mathsf{cof} & \Gamma \vdash \phi \vee \psi\,\mathsf{true} \\ \Gamma \triangleright \phi \vdash A_\phi\,\mathsf{type} & \Gamma \triangleright \psi \vdash A_\psi\,\mathsf{type} & \Gamma \triangleright \phi \wedge \psi \vdash A_\phi[\mathbb{p} \triangleright \star] = A_\psi[\mathbb{p} \triangleright \star]\,\mathsf{type} \end{array}}{\Gamma \vdash [\phi \hookrightarrow A_\phi \mid \psi \hookrightarrow A_\psi]\,\mathsf{type}}$$

$$\frac{\begin{array}{ccc} & \Gamma \vdash \phi_1, \phi_2\,\mathsf{cof} & \Gamma \triangleright \phi_1 \vdash A_{\phi_1}\,\mathsf{type} \\ \Gamma \triangleright \phi_2 \vdash A_{\phi_2}\,\mathsf{type} & \Gamma \triangleright \phi_1 \wedge \phi_2 \vdash A_{\phi_1}[\mathbb{p} \triangleright \star] = A_{\phi_2}[\mathbb{p} \triangleright \star]\,\mathsf{type} & \Gamma \vdash \phi_i\,\mathsf{true} \end{array}}{\Gamma \vdash [\phi_1 \hookrightarrow A_{\phi_1} \mid \phi_2 \hookrightarrow A_{\phi_2}] = A_{\phi_i}[\mathbf{id} \triangleright \star]\,\mathsf{type}}$$

$$\frac{\Gamma \vdash \phi, \psi \,\mathsf{cof} \qquad \Gamma \vdash \phi \vee \psi \,\mathsf{true} \qquad \Gamma \vdash A \,\mathsf{type}}{\Gamma \vdash [\phi \hookrightarrow A[\mathbb{p}] \mid \psi \hookrightarrow A[\mathbb{p}]] = A \,\mathsf{type}}$$

$$\frac{\Gamma \vdash \bot \,\mathsf{true}}{\Gamma \vdash \mathbf{Abort} \,\mathsf{type}} \qquad \frac{\Gamma \vdash \bot \,\mathsf{true} \qquad \Gamma \vdash A \,\mathsf{type}}{\Gamma \vdash \mathbf{Abort} = A \,\mathsf{type}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \phi, \psi \,\mathsf{cof} \qquad \Gamma \vdash \phi \vee \psi \,\mathsf{true} \qquad \Gamma \vdash A \,\mathsf{type} \\ \Gamma \cdot \phi \vdash a_\phi : A[\mathbb{p}] \qquad \Gamma \cdot \psi \vdash a_\psi : A[\mathbb{p}] \qquad \Gamma \cdot \phi \wedge \psi \vdash a_\phi[\mathbb{p} \cdot \star] = a_\psi[\mathbb{p} \cdot \star] : A[\mathbb{p} \cdot \star]\end{array}}{\Gamma \vdash [\phi \hookrightarrow a_\phi \mid \psi \hookrightarrow a_\psi] : A}$$

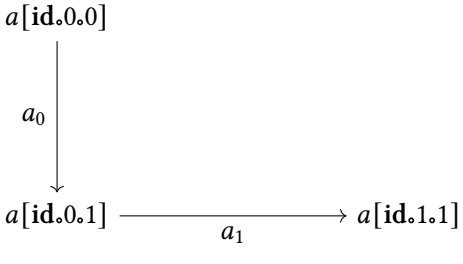
$$\frac{\Gamma \vdash \bot \,\mathsf{true} \qquad \Gamma \vdash A \,\mathsf{type}}{\Gamma \vdash \mathbf{abort} : A} \qquad \frac{\Gamma \vdash \bot \,\mathsf{true} \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathbf{abort} = a : A}$$

Advanced Remark 5.3.11. More concisely, these conditions ensure that $\Gamma \cdot \phi \vee \psi$ is a pushout of $\Gamma \cdot \phi$ and $\Gamma \cdot \psi$ over $\Gamma \cdot \phi \wedge \psi$ and that the presheaves for terms, types, *etc.* carry these pushouts to pullbacks. Similarly, they guarantee that $\Gamma \cdot \bot$ is initial and that all relevant presheaves carry this initial object to a terminal object. ◇

From cofibrations to subcubes These rules finally allow us to deliver on an earlier promise: we can now use cofibrations to isolate particular combinations of faces from a term. Let us consider the context consisting of two dimension variables extended by a cofibration stating that either the first variable is 0 or the second is 1:

$$\Gamma = \mathbf{1} \cdot \mathbb{I} \cdot \mathbb{I} \cdot (\mathbb{q}[\mathbb{p}] = 0 \vee \mathbb{q} = 1)$$

We know by the above rules for disjunction that giving a term $\Gamma \vdash a : A[\mathbb{p}^3]$ is equivalent to giving two terms $\mathbf{1} \cdot \mathbb{I} \cdot \mathbb{I} \cdot (\mathbb{q}[\mathbb{p}] = 0) \vdash a_0 : A[\mathbb{p}^3]$ and $\mathbf{1} \cdot \mathbb{I} \cdot \mathbb{I} \cdot (\mathbb{q} = 1) \vdash a_1 : A[\mathbb{p}^3]$ that agree on the overlap. Next, one may use the equality reflection rule for $\mathbb{q} = 1$ to show that *e.g.*, the substitution $\mathbf{1} \cdot \mathbb{I} \cdot \mathbb{I} \cdot (\mathbb{q} = 1) \vdash \mathbb{p} \circ \mathbb{p} : \mathbf{1} \cdot \mathbb{I}$ is invertible. We may therefore visualize a_0 and a_1 as lines in A which share a common boundary:



More generally, if ϕ is any cofibration then $\Gamma \cdot \phi \vdash a_\phi : A[\mathbb{p}]$ will consist of some *coherent* collection of faces in A. In other words, ϕ isolates some subset of the faces of an n-cube, and the rule for splitting on disjunctions of cofibrations ensures that a_ϕ consists of a term for each face *such that these terms agree on all overlaps.* The question of whether these faces can be stitched together into a single n-cube amounts to asking whether or not there exists some $\Gamma \vdash a : A$ such that $\Gamma \cdot \phi \vdash a[\mathbb{p}] = a_\phi : A[\mathbb{p}]$. This rephrasing in terms of cofibrations offers two important advantages. First, this formulation has better behavior with respect to substitution: it is clear that any extension in the above sense is stable under substitution and it also ensures that we can sensibly discuss applying substitutions to collections of faces. Second, cofibrations allow us to discuss more exotic faces like the line carved out by the cofibration $i = j$ for two dimension variables i, j. This corresponds to the *diagonal* of a square, rather than any of its standard edges.

Notation 5.3.12. In Section 5.4, we will wish to manipulate cofibrations when working informally with type theory. In general, like dimension variables the substitution calculus ensures that we can largely pretend ϕ is a "type", but the exceptionally strict properties around cofibrations ensure that we need never actually pass one around. When working informally, we shall therefore treat them in much the same way proof assistants handle implicit arguments: abstracting over them with a bespoke function type (the *partial element type*) but never needing to actually provide explicit terms to apply these function types. We present only the mapping-in property for this type and leave it to the reader to see how ordinary implicit function syntax may be translated to this isomorphism:

$$\mathsf{Tm}(\Gamma, \phi \to A) \cong \mathsf{Tm}(\Gamma \cdot \phi, A)$$

In the above, $\Gamma \vdash \phi \to A$ type just when $\Gamma \cdot \phi \vdash A$ type. However, since we shall only use this connective for informal explanations, we will not regard it as part of our definition of cubical type theory and content ourselves with this sketch of its rules.

5.3.6 *Composing and filling paths*

We are now ready to describe the second operation for manipulating paths **hcomp** and the final component of core cubical type theory. Recall that this operation is intended to take collections of faces—a subset of an n-cube in A—and assemble them into a single n-cube in A. As noted earlier, it is unsound to provide such an operation for *arbitrary* subcubes, but with the apparatus of cofibrations to hand, it is possible to describe a flexible class of shapes for which it is sound: given a term $\Gamma \vdash a_0 : A$ representing an n-cube in A along with a cofibration $\Gamma \vdash \phi\ \mathsf{cof}$ and a "ϕ-partial line" $\Gamma \cdot \phi \cdot \mathbb{I} \vdash a_\phi : A$,

which matches a_0 appropriately, we may glue and extend them using **hcomp** to an $(n+1)$-cube in A. The formal rules are as follows with a_0 and a_ϕ packaged into a single partial term using disjunction of cofibrations:

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash r, s : \mathbb{I} \qquad \Gamma \vdash \phi\,\mathsf{cof} \qquad \Gamma.\mathbb{I}.(\mathsf{q} = r[\mathsf{p}] \vee \phi[\mathsf{p}]) \vdash a : A[\mathsf{p}^2]}{\Gamma \vdash \mathbf{hcomp}_A^{r \to s}(\phi, a) : A}$$

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash r, s : \mathbb{I} \qquad \Gamma \vdash \phi\,\mathsf{cof} \qquad \Gamma \vdash \phi\,\mathsf{true} \qquad \Gamma.\mathbb{I}.(\mathsf{q} = r[\mathsf{p}] \vee \phi[\mathsf{p}]) \vdash a : A[\mathsf{p}^2]}{\Gamma \vdash \mathbf{hcomp}_A^{r \to s}(\phi, a) = a[\mathbf{id}.s.\star] : A}$$

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash r : \mathbb{I} \qquad \Gamma \vdash \phi\,\mathsf{cof} \qquad \Gamma.\mathbb{I}.(\mathsf{q} = r[\mathsf{p}] \vee \phi[\mathsf{p}]) \vdash a : A[\mathsf{p}^2]}{\Gamma \vdash \mathbf{hcomp}_A^{r \to r}(\phi, a) = a[\mathbf{id}.r.\star] : A}$$

With **hcomp** to hand, we will be able to complete the necessary "programming exercise" implementing **coe** in **Path**. Having added **hcomp**, however, we have unleashed another avalanche of necessary programming exercises: we must discuss how **hcomp** can be reduced for each type constructor. Fortunately, however, at this point we have all the necessary tools to do this for every connective except the universe. We discuss the rules governing **hcomp** for the non-universe connectives in Section 5.4, but they are largely unsurprising.

The real complexity of **hcomp** comes in defining $\mathbf{hcomp}_{\mathbf{U}}^{r \to s}(\phi, A)$. The problem is that, as an element of the universe, this composition is a code for a type and so one must describe the type $\mathbf{El}(\mathbf{hcomp}_{\mathbf{U}}^{r \to s}(\phi, A))$. It is not obvious, but the constraints of **hcomp** mean that this type must be non-empty and so non-trivial introduction and elimination rules must be given to govern this type. As with any other type we must describe also the behavior of **hcomp** and **coe** in $\mathbf{El}(\mathbf{hcomp}_{\mathbf{U}}^{r \to s}(\phi, A))$ and these "nested" composition problems are rather intricate.

This complexity, however, is the essential tool by which cubical type theory supports a computational account of univalence. We will return to this topic in Section 5.4, so we provide only the intuition here. Recall that the univalence axiom provides an inverse to a certain map $\mathbf{Path}(\mathbf{U}, A, B) \to \mathbf{El}(A) \simeq \mathbf{El}(B)$. The domain of this map now consists of certain lines in the universe—codes of types depending on $\mathbb{I}$—and so to interpret univalence, it suffices to define a family of types depending on $A, B : \mathbf{U}$, an equivalence $e : \mathbf{El}(A) \simeq \mathbf{El}(B)$ and a dimension term $r : \mathbb{I}$. This type is typically written $\mathbf{V}(r, A, B, e)$ (as in univalence).

The idea is that this type must collapse to A when the interval variable is specialized to 0 and to B when it is specialized with 1. This type, by definition, is a path in the

universe $\mathbf{Path}(\mathbf{U}, A, B)$. As with any other type, one must describe composition and coercion in this line of types and it is here that the invertibility of the given map $\mathbf{El}(A) \simeq \mathbf{El}(B)$ is crucial: it is this map which is used to supply coercions from one end of $\mathbf{V}$ to the other.

While this sketch omits a great many details—even simplifying the shape of $\mathbf{V}$ slightly—this is the crucial idea and payoff for recasting identity types as path types. By forcing identity types in the universe to take this more flexible form, we can define novel type formers which themselves implement the novel identifications mandated by univalence. The details vary greatly between presentations, but this general strategy is ubiquitous: (1) using an interval to encode identity types as path types, (2) adding additional operations to all types to force these path types to be symmetric, transitive, *etc.* and (3) implementing univalence by a specific type family depending on the interval.

The reward for the complexity of cubical type theory is the following theorem.

Theorem 5.3.13. *Cubical type theory enjoys consistency, canonicity, and normalization.*

These theorems were established over several years, for several different variations of cubical type theory. The consistency of the theory was proven in the first papers on cubical type theory [CCHM18; AFH17]. Canonicity was established by Huber [Hub18] and Angiuli, Hou (Favonia), and Harper [AFH17]. Normalization was proven by Sterling and Angiuli [SA21].

5.4⋆ *Computing with coercions and compositions* (DRAFT)

Section 5.3 presented the core aspects of cubical type theory, but with many rules and details elided. In this section, we endeavor to fill in a few of these gaps by explaining some of the rules governing the computation of **hcomp** and **coe** in various types. Even in a dedicated section, however, we will not provide all of these rules. A complete set can be found in *e.g.*, Angiuli et al. [Ang+21].

Fortunately, the remaining rules do not introduce new judgmental structure. Instead, they are more akin to programming exercises and show how to build *e.g.*, $\mathbf{hcomp}_{\mathbf{\Pi}(A,B)}$ in terms of composition and coercion in A and B. Accordingly, while the previous section was replete with rules and substitutions, we shall see far fewer of these in this section. Instead, we shall focus on these "programming exercises" and often write out the resulting terms for computing composition and coercion in more informal type-theoretic notation. We will present a few examples for how these are

turned into actual formal rules to be added to cubical type theory but thereafter leave this mechanical task to the reader.

Notation 5.4.1. In order to facilitate writing informal terms with **coe** and **hcomp**, we shall treat them as closed elements of the following types:

$$\mathbf{coe} : (A : \mathbb{I} \to \mathbf{U})(i, j : \mathbb{I}) \to A(i) \to A(j)$$
$$\mathbf{hcomp}_{\phi} : (A : \mathbf{U})(i, j : \mathbb{I})(a : (k : \mathbb{I}) \to (i = k \vee \phi) \to A) \to A$$

5.4.1 coe *for* Π *and* Σ

We begin by describing coercion for dependent products and sums. These two examples contain all the interesting structure one finds in the definitions of **coe** for the types of base Martin-Löf type theory and so we give them a fair bit of attention.

We begin by specifying the right-hand side of the following definitional equality:

$$\frac{\Gamma.\mathbb{I} \vdash A\,\mathrm{type} \qquad \Gamma.\mathbb{I}.A \vdash B\,\mathrm{type} \qquad \Gamma \vdash r, s : \mathbb{I} \qquad \Gamma \vdash p : \Sigma(A,B)[\mathbf{id}.r]}{\Gamma \vdash \mathbf{coe}^{r\to s}_{\Sigma(A,B)}(p) = \;?\; : \Sigma(A,B)[\mathbf{id}.s]}$$

This is one of the many, many "programming exercises" in cubical type theory. Our goal shall be to produce a term using **coe** for A and B which has the appropriate type to fit into the above rule, subject to the additional condition that when $r = s$ then this term is equal to p. This last point is not strictly necessary for the rule to be well-formed, but it is an important sanity check. After all, the definitional equality for $\mathbf{coe}^{r\to r}_{\Sigma(A,B)}(p)$ will force this to be true by transitivity and so it makes sense to ensure that this forced equality is sensible.

We shall divide this process up into two steps. First, we present this term using informal type theory and second, we shall list out the formal term in proper notation.

Lemma 5.4.2. *Fix* $A : \mathbb{I} \to \mathbf{U}$, $B : (i : \mathbb{I}) \to A(i) \to \mathbf{U}$, $r, s : \mathbb{I}$, *and* $p : \sum_{a:A\,r} B\,r\,a$. *Using* **coe** *for* A *and* B, *we can construct a term* $\mathbf{coe}\,(\lambda i \to \sum_{a:A\,i} B\,i\,a)\,r\,s\,p : \sum_{a:A\,s} B\,s\,a$ *which is definitionally equal to* p *if* $r = s$.

Proof. By the η law for dependent sums, this term must be of the form (a, b) for some element of $A\,s$ and of $B\,s\,a$. In fact, it is straightforward to find the first component of this pair: $a = \mathbf{coe}\,A\,r\,s\,\mathbf{fst}(p)$.

The second component of the pair is more complex. Naïvely, one might hope that one could mirror the construction for a and use $\mathbf{coe}\,B$ in some manner. However, this is not well-typed! After all, B is not in the correct shape for **coe**: it is an element of $(i : \mathbb{I}) \to A\,i \to \mathbf{U}$ and not the required $\mathbb{I} \to \mathbf{U}$. Accordingly, to apply **coe** we must

choose some element of A with which to specialize B. In fact, the situation is more fraught than this: A itself depends on $\mathbb{I}$ and so if we wish to obtain a specialization of B with the type $\mathbb{I} \to \mathbf{U}$, we will require an element of $\bar{a} : (i : \mathbb{I}) \to A\,i$. Given such an $\bar{a}$, however, we can then use **coe** with $B_{\bar{a}} = \lambda i \to B\,i\,(a\,i)$ to attempt to construct b.

We can further narrow things down with this in mind. After all, our goal is to set $b = \mathbf{coe}\,B_{\bar{a}}\,r\,s\,\mathbf{snd}(p)$ and if this is to be type-correct we must have $\bar{a}\,r = \mathbf{fst}(p)$. Moreover, since we wish to have $b : B\,a\,s$ we must have $\bar{a}\,s$ be $a = \mathbf{coe}\,A\,r\,s\,\mathbf{fst}(p)$.

In order to obtain $\bar{a}$, we take advantage of the flexibility of **coe** to coerce from r to a variable dimension, rather than 0 or 1. Specifically, we define $\bar{a}$ as follows:

$$\bar{a} \coloneqq \lambda i \to \mathbf{coe}\,A\,r\,i\,a$$

With $\bar{a}$ to hand, we choose $b \coloneqq \mathbf{coe}\,B_{\bar{a}}\,r\,s\,\mathbf{snd}(p)$, completing the required term. We leave it to the reader to check the required definitional equality holds when $r = s$. □

Rendering the above term in formal notation, the rule can be completed to the following:

$$\frac{\Gamma.\mathbb{I} \vdash A\,\mathsf{type} \qquad \Gamma.\mathbb{I}.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash r, s : \mathbb{I} \qquad \Gamma \vdash p : \Sigma(A,B)[\mathbf{id}\circ r]}{\Gamma \vdash \mathbf{coe}^{r\to s}_{\Sigma(A,B)}(p) = \mathbf{pair}(\mathbf{coe}^{r\to s}_{A}(\mathbf{fst}(p)), \mathbf{coe}^{r\to s}_{B[\mathbf{id}.\mathbf{coe}^{r[\mathbb{p}]\to q}_{A[\mathbb{p}\circ\mathbb{I}]}(\mathbf{fst}(p[\mathbb{p}]))]}(\mathbf{snd}(p))) : \Sigma(A,B)[\mathbf{id}\circ s]}$$

While the translation is largely mechanical, the reader can hopefully appreciate that the informal term is far more legible than the formal cousin!

We now turn to the case of dependent products. The process is mostly similar and we use the coercion operations on A and B to specify how $\mathbf{coe}_{\mathbf{\Pi}(A,B)}$ ought to compute. Our goal is once more to fill in the right-hand side of the following equality:

$$\frac{\Gamma.\mathbb{I} \vdash A\,\mathsf{type} \qquad \Gamma.\mathbb{I}.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash r, s : \mathbb{I} \qquad \Gamma \vdash f : \mathbf{\Pi}(A,B)[\mathbf{id}\circ r]}{\Gamma \vdash \mathbf{coe}^{r\to s}_{\mathbf{\Pi}(A,B)}(f) = \;?\; : \mathbf{\Pi}(A,B)[\mathbf{id}\circ s]}$$

Lemma 5.4.3. *Fix $A : \mathbb{I} \to \mathbf{U}$, $B : (i : \mathbb{I}) \to A(i) \to \mathbf{U}$, $r, s : \mathbb{I}$, and $p : (a : A\,r) \to B\,r\,a$. Using* **coe** *for A and B, we can construct a term* $\mathbf{coe}\,(\lambda i \to (a : A\,i) \to B\,i\,a)\,r\,s\,p : (a : A\,s) \to B\,s\,a$ *which is definitionally equal to p if $r = s$.*

Proof. Our goal is to construct an element of $(a : A(s)) \to B\,s\,a$ and, accordingly, we fix $a : A(s)$ and set about constructing $B\,s\,a$. We begin by defining $a_r = \mathbf{coe}\,A\,s\,r\,a$ such that we obtain $b_r = f(a_r) : B\,r\,a_r$. We would like to coerce b_r to obtain our desired element of $B\,s\,a$, but along what type should this coercion occur? We must find some $\bar{a} : (i : \mathbb{I}) \to A(i)$ such that $\bar{a}(r) = \mathbf{coe}\,A\,s\,r\,a$ and $\bar{a}(s) = a$. Capitalizing on the fact that $\mathbf{coe}\,A\,s\,s\,a = a$, we choose $\bar{a}$ to be $\lambda k \to \mathbf{coe}\,A\,s\,k\,a$. The full term then becomes the following:

$$\lambda a \to \mathbf{coe}\,(\lambda k \to B\,k\,(\mathbf{coe}\,A\,s\,k\,a))\,r\,s\,(f(\mathbf{coe}\,A\,s\,r\,a))$$

Once again, we leave it to the intrepid reader to confirm that if $r = s$ then this term is simply equivalent to f. □

For the final time, we provide a translation of this informal definition into formal notation. Hereafter, we shall leave this mechanical (if tedious) process to the reader:

$$\frac{\Gamma \circ \mathbb{I} \vdash A\,\mathsf{type} \qquad \Gamma \circ \mathbb{I}.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash r, s : \mathbb{I} \qquad \Gamma \vdash f : \mathbf{\Pi}(A,B)[\mathbf{id} \circ r]}{\Gamma \vdash \mathbf{coe}^{r \to s}_{\mathbf{\Pi}(A,B)}(f) = \lambda(\mathbf{coe}^{r[\mathbf{p}] \to s[\mathbf{p}]}_{B[\mathbf{p} \circ \mathbf{q}.\mathbf{coe}^{s[\mathbf{p} \circ \mathbf{p}] \to \mathbf{q}}_{A[\mathbf{p} \circ \mathbb{I}]}(\mathbf{q}[\mathbf{p}])]}(\mathbf{app}(f[\mathbf{p}], \mathbf{coe}^{s[\mathbf{p}] \to r[\mathbf{p}]}_{A[\mathbf{p} \circ \mathbb{I}]}(\mathbf{q})))) : \mathbf{\Pi}(A,B)[\mathbf{id} \circ s]}$$

Undeniably, these rules are complicated.[7] They are, however, really just a sequence of programming exercises and share many characteristics and so describing the first few cases is the most painful.

The next novelty, as already mentioned, comes in the definition of **coe** for path types. We turn to this next and, consequently, shift our attention to the second operator we must define for every type: **hcomp**.

5.4.2 *Working with the homogeneous composition operator*

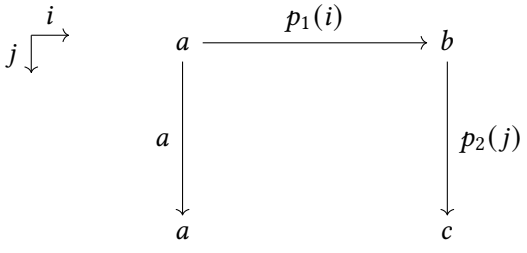
A common challenge when one begins to study cubical type theory is to "visualize" **hcomp**. While **coe** matched closely enough with the already familiar subst operator, the homogeneous composition operator is quite different than any of the combinators one typically encounters in intensional type theory. Prior to using it to compute coercion in path types, we give a few simple worked examples of **hcomp** to help demystify this operator.

Composing two paths using hcomp Let us begin cultivating intuition for **hcomp** by showing how we can use it to compose two paths $p_1 : \mathbf{Path}(A, a, b)$ and $p_2 : \mathbf{Path}(A, b, c)$. We shall do this using $\mathbf{hcomp}\, A$, so it remains only to choose (1) the $r \to s$ direction we wish to compose along and (2) the cofibration ϕ to restrict along.

To visualize this situation, let us briefly fix two dimension variables $i, j : \mathbb{I}$ and

[7] Indeed, the reader may wonder how the authors managed to get these complicated terms correct. The answer is simple: they did not. They found numerous typos in the process of editing this section.

instantiate p_1 with i and p_2 with j. We can draw the situation as follows:



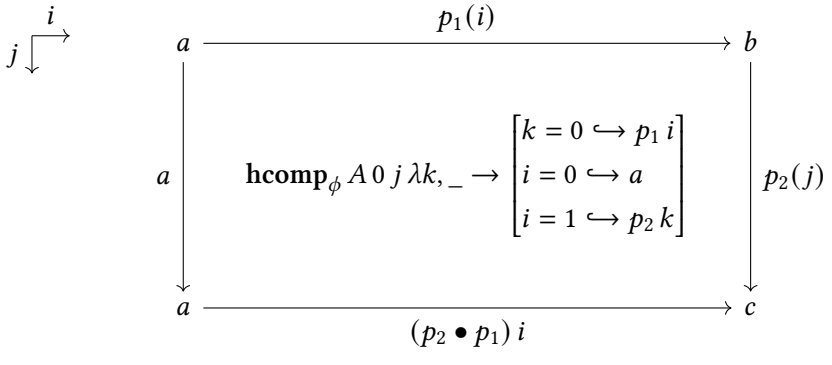
In anticipation of what is to come, we have added a "degenerate" edge corresponding to reflexivity along a. In order to construct the composite of our two edges, it suffices to find a line which joins the bottom two vertices. It is here we invoke **hcomp**. Since our goal is to fill "down" in the j direction, *e.g.*, to push the top edge along the two vertical edges, we shall apply **hcomp** from 0 to 1. The cofibration shall be used to isolate the two sides in this direction that we possess so $\phi \coloneqq i = 0 \vee i = 1$.

Let us put these pieces of intuition together into a term. Our goal is to construct a path in A, so we will begin by binding a dimension variable $i : \mathbb{I}$. We then define the composite path as follows:

$$(p_2 \bullet p_1)\, i = \mathbf{hcomp}_\phi\, A\, 0\, 1\, (\lambda k, _ \to [k = 0 \hookrightarrow p_1\, i \mid \phi \hookrightarrow [i = 0 \hookrightarrow a \mid i = 1 \hookrightarrow p_2\, k]])$$

Exercise 5.22. Argue that $p_2 \bullet p_1$ has the expected boundary *i.e.* that $(p_2 \bullet p_1)\, 0 = a$ and that $(p_2 \bullet p_1)\, 1 = c$.

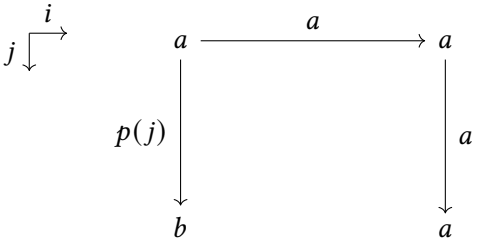
What if we wish to obtain not just the bottom edge of the square, but the entire 2-dimensional term? Just as we could produce lines by using **coe** with a variable dimension as the target, we can "**hcomp** to the middle" using a dimension variable to obtain the entire square. We represent this with the following diagram:



Exercise 5.23. Check that this 2-dimensional term has the relevant boundary conditions. In particular, if $j = 0$ check it collapses to $p_1\, i$.

Inverting a path using hcomp For a second example, suppose we are given p : $\mathbf{Path}(A, a, b)$. We show how **hcomp** may be used to construct an inverse path $\mathbf{Path}(A, b, a)$. Once more, we shall fill a square involving p alongside two degenerate paths.

To visualize this situation, let us fix $i, j : \mathbb{I}$ and consider the following three lines:



In order to compose paths, we have already shown how to use **hcomp** to complete these three edges to a square. The same general procedure applies, though the result is now the inverse to p. In particular, we have the following:

$$p^{-1}\, i \coloneqq \mathbf{hcomp}_{\phi}\, A\, 0\, 1\, \lambda k, _ \to \begin{bmatrix} k = 0 \hookrightarrow a \\ i = 0 \hookrightarrow p(k) \\ i = 1 \hookrightarrow a \end{bmatrix}$$

In fact, with further effort we could use **hcomp** to construct higher paths witnessing *e.g.*, a path between $p \bullet p^{-1}$ and a constant path. Rather than pursuing this more fully, however, we return to the original example which prompted this detour.

Coercion in path types from hcomp We can now complete the loop that motivated this detour and show how to implement coercion in **Path**. Crucially, this requires both **coe** and **hcomp** working in concert.

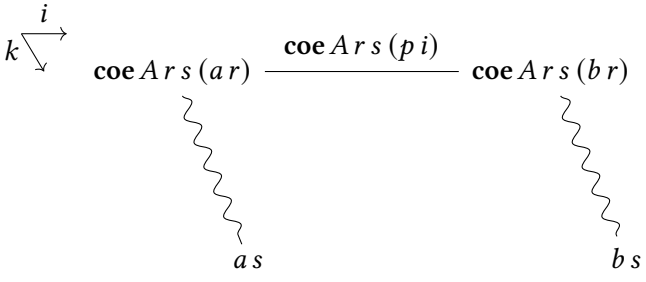
Lemma 5.4.4. *Fix* $A : \mathbb{I} \to \mathbf{U}$, $a, b : (i : \mathbb{I}) \to A(i)$ *alongside* $r, s : \mathbb{I}$ *and* p : $\mathbf{Path}(A(r), a(r), b(r))$. *Using* **hcomp** *and* **coe** *for* A, *there exists a term of the following type:*

$$\mathbf{coe}\, (\lambda i \to \mathbf{Path}(A(i), a(i), b(i)))\, r\, s\, p : \mathbf{Path}(A(s), a(s), b(s))$$

Moreover, this term is definitionally equal to p *when* $r = s$.

Proof. As before, let us fix $k : \mathbb{I}$ such that it now suffices to define an element of $A(s)$ which specializes to $a(s)$ and $b(s)$ when $k = 0$ or $k = 1$. This latter condition is the sort of problem well-addressed by $\mathbf{hcomp}_{\phi}$ where $\phi \coloneqq k = 0 \vee k = 1$: one of the definitional equalities governing the construction precisely allows us to guarantee these equations. It remains to work out the direction in which we ought to apply $\mathbf{hcomp}_{\phi}$ as well as

the "top" of the square we are filling. Let us revisit the drawing of the situation we encountered when first attempting to construct **coe** in **Path**:



Here, we have depicted the vertical lines as "wavy" since they do not actually form a path with the top corresponding to 0 and the bottom to 1. Instead, they represent lines in $A(s)$ that *e.g.*, when specialized with r become $\mathbf{coe}\, A\, r\, s\, (b\, r)$ and with s become $b\, s$. This, however, is precisely what we require if we use composition from r to s, rather than from 0 to 1. All told then, we arrive at the following term:

$$\mathbf{coe}\, A\, r\, s\, p \coloneqq \lambda i \to \mathbf{hcomp}_{i=0 \vee i=1}\, (A\, s)\, r\, s\ \lambda k, _ \to \begin{bmatrix} k = r \hookrightarrow \mathbf{coe}\, A\, r\, s\, (p\, i) \\ i = 0 \hookrightarrow \mathbf{coe}\, A\, k\, s\, (a\, k) \\ i = 1 \hookrightarrow \mathbf{coe}\, A\, k\, s\, (b\, k) \end{bmatrix}$$

We leave it to the reader to confirm that all three of the branches of the disjunction match as required on their overlaps and that when $r = s$ this term collapses to p. □

5.4.3 *Unfolding* **hcomp** *in various type constructors*

While we have discussed the core rules governing **coe** at this point, it remains to do so for **hcomp**. Just as with coercion, specifying core connectives amounts to a sequence of programming exercises and we give the details only for dependent products and path types.

Lemma 5.4.5. *Fix a cofibration ϕ, types $A : \mathbf{U}$, $B : A \to \mathbf{U}$, dimension terms $r, s : \mathbb{I}$, and a term $f : (i : \mathbb{I}) \to (i = r \vee \phi) \to (a : A) \to B(a)$. There exists* $\mathbf{hcomp}_{\phi}\, ((a : A) \to B\, a)\, r\, s\, f$ *of type $(a : A) \to B(a)$ built from composition in B satisfying the expected definitional equalities.*

Proof. Let us fix $a : A$ such that we must build $b : B(a)$ such that if either $r = s$ or ϕ holds then $b = f\, s\, a$. To this end, we shall use composition in $B(a)$:

$$b = \mathbf{hcomp}_{\phi}\, (B\, a)\, r\, s\, \lambda i, _ \to f\, i\, _\, a$$

It is routine to see that this gives rise to the required term using the boundary conditions of $\mathbf{hcomp}_\phi\ (B\,a)\ r\ s$. □

Lemma 5.4.6. *Fix a cofibration ϕ, a type $A : \mathbf{U}$, elements $a, b : A$, dimension terms $r, s : \mathbb{I}$, and a partial term $p : (i : \mathbb{I}) \to (i = r \vee \phi) \to \mathbf{Path}(A, a, b)$. There exists a term $\mathbf{hcomp}_\phi\ (\mathbf{Path}(A, a, b))\ r\ s\ p : \mathbf{Path}(A, a, b)$ built from composition and coercion in A satisfying the expected definitional equalities.*

Proof. The required term is an application of **hcomp** in A. Since we intend to construct a path, we fix $i : \mathbb{I}$ such that $\lambda j, _ \to p\ j\ _\ i : (j : \mathbb{I}) \to j = r \vee \phi \to A$ is a partial element suitable as input for **hcomp**.

This is almost sufficient, but we must also ensure that the resulting extended term satisfies the boundary condition necessary to form an element of $\mathbf{Path}(A, a, b)$. To fix these boundaries, we extend ϕ with faces to govern the behavior of this term when $i = 0$ or $i = 1$. The final term is given as follows:

$$\mathbf{hcomp}_\phi\ (\mathbf{Path}(A, a, b))\ r\ s\ p \coloneqq \lambda i \to \mathbf{hcomp}_{\phi \vee i=0 \vee i=1}\ A\ r\ s\ \lambda j, _ \to \begin{bmatrix} \phi \vee j = r \hookrightarrow p\ j\ _\ i \\ i = 0 \hookrightarrow a \\ i = 1 \hookrightarrow b \end{bmatrix}$$

We once more leave it to the reader to check that this satisfies the necessary boundary conditions. □

5.4.4 $\mathbf{V}$ *and univalence*

Finally, we turn to the rules necessary to animate both **hcomp** in $\mathbf{U}$ and univalence. The crucial idea behind both is the same: paths in the universe are, by definition, codes which depend on $\mathbb{I}$ and so to implement either **hcomp** or univalence, it suffices to define new types. We shall focus largely on the new type necessary to implement univalence $\mathbf{V}$, but much of the process transfers to **hcomp**.

Suppose we are given $A, B : \mathbf{U}$ along with $e : A \simeq B$. We wish to construct a path $\mathrm{ua}\ e : \mathbf{Path}(\mathbf{U}, A, B)$ or, equivalently, a map $p : \mathbb{I} \to \mathbf{U}$ such that $p\ 0 = A$ and $p\ 1 = B$. We intend for ua to be inverse to the canonical map $\mathrm{idToEquiv} : \mathbf{Id}(\mathbf{U}, A, B) \to A \simeq B$ which, in this setting, amounts to requiring that $\mathbf{coe}\ p\ 0\ 1 : A \to B$ is equal to e.

Remark 5.4.7. The reader may wonder whether we need an additional constraint ensuring that $\mathrm{ua}(\mathbf{coe}\ p\ 0\ 1)$ can be identified with p. As we remarked in Section 5.2, this direction holds automatically. ◇

Our goal shall be to define a new type $\mathbf{V}(A, B, e, r)$ and to set $\mathrm{ua}\, e \coloneqq \lambda i \to \mathbf{V}(A, B, e, i)$. We begin with the (provisional) formation rule for $\mathbf{V}$:

$$\frac{\Gamma \vdash A, B\,\mathsf{type} \qquad \Gamma \vdash e : A \simeq B \qquad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash \mathbf{V}(A, B, e, r)\,\mathsf{type}}$$

We note that this definition is sensitive to the precise realization of equivalence we choose. However, any of the notions presented in Section 5.2 suffice and so we shall ignore this detail. Moreover, we must ensure that our universe is closed under $\mathbf{V}$ in order to actually carry out the definition of ua. It is more convenient to specify rules for the type $\mathbf{V}$ rather than the code, however, and so we shall focus on that.

The above set of constraints on ua and path types generally translate into the following requirements for $\mathbf{V}(A, B, e, r)$:

- We must have definitional equalities $\mathbf{V}(A, B, e, 0) = A$ and $\mathbf{V}(A, B, e, 1) = B$.
- It must be the case that $\mathbf{coe}\ (\lambda i \to \mathbf{V}(A, B, e, i))\ 0\ 1 = e$.
- We must be able to implement **hcomp** and **coe** for $\mathbf{V}$.

Of course, $\lambda i \to \mathbf{V}(A, B, e, i)$ is always fully constrained up to equivalence: it is the unique inhabitant of $\mathbf{Path}(\mathbf{U}, A, B)$ sent to e by **coe**. In this way, it is largely unimportant how precisely $\mathbf{V}$ is realized. What matters is only that such a type can exist and satisfy the list of required properties. To this end, these constraints are useful for nailing down the particular rules which define $\mathbf{V}$ more precisely and, unfortunately, we must give new rules. $\mathbf{V}$ cannot be defined by a clever combination of existing type formers because of the first requirement; we presently have no means of defining a type which degenerates to two distinct types depending on the endpoints of an interval.

In fact, given all these constraints there are precious few valid choices for the introduction and elimination rules of $\mathbf{V}$. The difficulty is that it is not obvious whether any given choice of rules will suffice until one carefully checks each condition. Accordingly, we will present the correct rules below and only then discuss some of the subtleties:

$$\frac{\Gamma \vdash A, B\,\mathsf{type} \qquad \Gamma \vdash e : A \simeq B}{\Gamma \vdash \mathbf{V}(A, B, e, 0) = A\,\mathsf{type} \qquad \Gamma \vdash \mathbf{V}(A, B, e, 1) = B\,\mathsf{type}}$$

$$\frac{\begin{array}{c}\Gamma \vdash A, B\,\mathsf{type} \qquad \Gamma \vdash e : A \simeq B \qquad \Gamma \vdash r : \mathbb{I} \\ \Gamma, r = 0 \vdash a : A[\mathbb{p}] \qquad \Gamma \vdash b : B \qquad \Gamma, r = 0 \vdash \mathbf{app}(e[\mathbb{p}], a) = b[\mathbb{p}] : B[\mathbb{p}]\end{array}}{\begin{array}{c}\Gamma \vdash \mathbf{Vin}(a, b, r) : \mathbf{V}(A, B, e, r) \\ \Gamma, r = 0 \vdash \mathbf{Vin}(a, b, r)[\mathbb{p}] = a : A[\mathbb{p}] \qquad \Gamma, r = 1 \vdash \mathbf{Vin}(a, b, r)[\mathbb{p}] = b[\mathbb{p}] : B[\mathbb{p}]\end{array}}$$

$$\frac{\Gamma \vdash A, B\,\mathsf{type} \qquad \Gamma \vdash e : A \simeq B \qquad \Gamma \vdash r : \mathbb{I} \qquad \Gamma \vdash v : \mathbf{V}(A, B, e, r)}{\begin{array}{c}\Gamma \vdash \mathbf{Vout}(v) : B \\ \Gamma, r = 0 \vdash \mathbf{Vout}(v[\mathbb{p}]) = \mathbf{app}(e[\mathbb{p}], v[\mathbb{p}]) : B[\mathbb{p}] \\ \Gamma, r = 1 \vdash \mathbf{Vout}(v[\mathbb{p}]) = v[\mathbb{p}] : B[\mathbb{p}]\end{array}}$$

$$\frac{\begin{array}{ccc}\Gamma \vdash A, B\,\mathsf{type} & \Gamma \vdash e : A \simeq B & \Gamma \vdash r : \mathbb{I} \\ \Gamma, r = 0 \vdash a : A[\mathbb{p}] & \Gamma \vdash b : B & \Gamma, r = 0 \vdash \mathbf{app}(e[\mathbb{p}], a) = b[\mathbb{p}] : B[\mathbb{p}]\end{array}}{\Gamma \vdash \mathbf{Vout}(\mathbf{Vin}(a, b, r)) = b : B}$$

$$\frac{\Gamma \vdash A, B\,\mathsf{type} \qquad \Gamma \vdash e : A \simeq B \qquad \Gamma \vdash r : \mathbb{I} \qquad \Gamma \vdash v : \mathbf{V}(A, B, e, r)}{\Gamma \vdash \mathbf{Vin}(v[\mathbb{p}], \mathbf{Vout}(v), r) = v : \mathbf{V}(A, B, e, r)}$$

In total then, an element of $\mathbf{V}(A, B, e, r)$ contains a partial element of A and a full element of B which *match up* according to e when both are defined. The introduction and elimination rules (along with their β and η principles) are then nearly routine from this perspective. The complexity comes from the various rules which apply if $r = 0$ or $r = 1$.

These are a consequence of having $\mathbf{V}(A, B, e, r)$ collapse definitionally to A and B. We have not encountered rules similar to this with other type formers and they impose a number of unique constraints on the rules around $\mathbf{V}$ if we are to avoid having terms of $\mathbf{V}(A, B, e, r)$ polluting A and B. For instance, we must add rules ensuring that $\mathbf{Vin}(a, b, r)$ correctly equates to a or b where this is required. Similarly, $\mathbf{Vout}(v)$ cannot come only with a β rule to govern its behavior, as it must account for the situations where v becomes an ordinary element of A and B.

To illustrate the delicacy of these rules, imagine a simple possible replacement: instead of requiring $\Gamma, r = 0 \vdash \mathbf{app}(e[\mathbb{p}], a) = b[\mathbb{p}] : B[\mathbb{p}]$, what if we required that $\mathbf{app}(e^{-1}, b)$ was definitionally equal to a? While this is seemingly innocuous, e and e^{-1} are inverses only up to a path and not necessarily definitionally inverses. Consequently, this exchange would make it impossible to properly specify the behavior of $\mathbf{Vout}(v)$ when $r = 0$; depending on the order in which rules were applied one could obtain distinct (but path equal!) terms.

Another mysterious aspect of these rules is the asymmetry between A and B. Why a is required to be a partial element whereas b is total as opposed to defined only when $r = 1$ holds. What matters is not so much whether a or b is partial, but merely that one of the two is fully defined and one is not. If neither is fully defined, it becomes impossible to state that a and b are equated by e. More subtly, if both are fully defined it becomes impossible to specify **coe** in $\mathbf{V}$.

The definitions of **hcomp** and **coe** in $\mathbf{V}$ are complex and we will not attempt to detail them here. The interested reader should consult Appendix B of Angiuli [Ang19]

for a precise account of $\mathbf{V}$.

Finally, we note that the same chain of reasoning that leads to this definition of $\mathbf{V}$ can be used to produce the type implementing $\mathbf{hcomp}_{\mathrm{U}}^{r\to s}(\phi, A)$. We can once more list out the various definitional equalities which such a type must satisfy as well as what types it must be equivalent to. Unfurling these, we determine that elements of $\mathbf{hcomp}_{\mathrm{U}}^{r\to s}(\phi, A)$ are essentially smaller formal composition problems, just as elements of $\mathbf{V}$ were "suspended coercions along e". Unfortunately, the details and bookkeeping around such formal composition problems (and composition problems *of* formal composition problems) is taxing. A curious reader should once again consult Angiuli [Ang19].

Further reading

We have only scratched the surface of univalent type theories in this chapter, as it is an active and ongoing area of research. For the student curious to learn more, we once again recommend one of the book-length treatments on homotopy type theory: "HoTT Book" [UF13] and *Introduction to Homotopy Type Theory* [Rij25]. The authors also recommend studying one of the many formalization projects based on HoTT. There are many to choose from, in a variety of different proof assistants and styles [VAG+20; Bau+17; Esc+10; Rij+21].

Synthetic homotopy theory We have already discussed some of the applications of homotopy type theory to homotopy theory at the end of Section 5.2. To highlight one additional recent and striking example of synthetic homotopy theory, we also wish to mention the fully-formalized proof of the Serre finiteness theorem [Bar+26]. Some results first proven in homotopy type theory have gone on to receive interest and analysis by ∞-category theorists and homotopy theorists, such as the theory of modalities [RSS20]. Finally, while we have regretfully omitted it, the homotopical theory of descent offers an elegant explanation of univalence [Ane19].

Semantics of homotopy type theory We have deferred treatment of the semantics of either HoTT or cubical type theory in this chapter. For the reader interested in the semantics of HoTT, the foundational simplicial model is described by Kapulkin and Lumsdaine [KL21]. Shulman [Shu19] affirmatively resolved the long-standing conjecture that all $(\infty, 1)$-topoi support a model of homotopy type theory. For a higher-level treatment, Riehl [Rie24] provides a summary of both of these results. The semantics of cubical type theory begin with Bezem, Coquand, and Huber [BCH14], but we recommend Orton and Pitts [OP16] and Licata et al. [Lic+18] for a higher-level treatment based on the internal language of cubical sets. See also Angiuli et al. [Ang+21] for a description of the semantics of the particular flavor of cubical type theory we discussed in this chapter. A more categorical treatment of the semantics of cubical type theory was also recently described in Awodey's "Cartesian Cubical Model Categories" [Awo26].

This material will be published by Cambridge University Press as *Principles of Dependent Type Theory* by Carlo Angiuli and Daniel Gratzer. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © Carlo Angiuli and Daniel Gratzer 2024

6 *Semantics of type theory* (DRAFT)

In Chapter 2, we formulated the syntax of (extensional) type theory via rules inductively defining sets of contexts, substitutions, types, and terms. In Chapter 3, we introduced the notion of a general *model* of type theory (Definition 3.4.2) by observing that those rules could alternatively be seen as a *signature* imposing various closure conditions on four arbitrary sets of contexts, etc., recovering the notion of syntax as a free or *initial* model. Although we defined the set model of type theory in Section 3.5 and discussed the groupoid model in Section 4.3, our focus throughout this book has been on syntactic models of type theory. In this chapter, we systematically consider models of type theory.

Many readers may have encountered the phrase "categorical semantics" in discussion of models of type theory. We have chosen to eschew the adjective "categorical" in the title of this chapter because, fundamentally, there is nothing categorical about the definition of model given in Definition 3.4.2. It is much closer in spirit to models in classical *universal algebra* such as groups, rings, or modules: a collection of sets together with operations and equations. Of course, a model of dependent type theory requires some of these sets to be *indexed* by elements of others, making it more general than an algebraic theory (more precisely, it is a *generalized* algebraic theory [Car86; Dyb96; KKA19]; see Appendix B).

In fact, the connection to category theory is much more pedestrian than one might assume: it so happens that the definition of a category is hiding within the definition of a model of type theory. Accordingly, every model of type theory can be seen as a category equipped with additional properties and structures. Thus, in a very real sense, we have been using the categorical semantics of type theory since Chapter 3.

Starting in this chapter however, we shall take advantage of this observation to repackage the definition of a model into a smaller and more tractable form. This process is a more exaggerated form of the simplification of replacing the fully unfolded definition of a ring with the more compact "an abelian group equipped with a multiplication operation $\cdot$ satisfying …". Mathematically, very little has changed but it is often practically easier to construct examples after this reorganization since we can reuse categorical intuitions.

Warning 6.0.1. With this in mind, for this chapter only we shall assume that the reader has a working knowledge of category theory. In particular, we shall assume familiarity with categories, functors, natural transformations, presheaves, the Yoneda embedding,

and adjunctions to the level of, for instance, the first four chapters of Riehl [Rie16] or the first nine chapters of Awodey [Awo10].

Remark 6.0.2. The reader without exposure to category theory may find this chapter useful motivation to begin studying category theory in its own right. Indeed, while it is perhaps not mandatory, a working knowledge of category theory is an invaluable tool for engaging with contemporary literature on type theory. For a reader ready to take the plunge, we recommend either of the two aforementioned books. ◇

In this chapter In Sections 6.1 to 6.4 we reorganize the definition of a model of type theory given in Section 3.2 into the concise notion of a *category with families* (cwf) [Dyb96]. We observe how the natural isomorphisms used in Chapter 2 to define connectives can be repurposed to give a succinct and efficient definition. We systematically use a more modern reformulation of cwfs as *natural models* as put forward by Awodey [Awo18].

In Sections 6.5 and 6.6 we set out to connect cwfs to locally cartesian closed categories (LCCCs). We describe the slogan originating with Seely [See84] that LCCCs are models of extensional type theory and illustrate how various *coherence* issues complicate this fact. We also describe at some length the local universes coherence construction [LW15; Awo18] and how it resolves these issues to construct a cwf on top of an arbitrary LCCC.

Finally, Section 6.7 is devoted to proving a claim from Chapter 3: extensional type theory satisfies canonicity. We do this by constructing a particular model of type theory based on a *gluing* construction and deriving canonicity from this model together with the fact that syntax organizes into the initial model.

Remark 6.0.3. Throughout this chapter, we focus on extensional type theory. We emphasize, however, that none of this material is specific to ETT. The curious reader may refer to *e.g.*, Awodey [Awo18] for a treatment of the intensional identity type. ◇

Goals of the chapter By the end of this chapter, you will be able to:

- Explain the definition of a cwf and why it constitutes a model of type theory.
- Explain how the locally cartesian closed category (LCCC) relates to a cwf.
- Use the local universes construction to construct a cwf from an LCCC.
- Prove metatheorems of type theory using semantic methods.

Glossary of category theory

Accumulate notations/etc here; later expand into a short section.

Here is the category theory we need. We will briefly recall definitions as we go, but not enough for a first exposure.

- $\hom_C(c, d)$
- $\mathbf{Pr}(C)$
- $C_{/c}$
- $f^* : \mathbf{Pr}(C) \longrightarrow \mathbf{Pr}(\mathcal{D})$
- $f^* : C_{/C} \longrightarrow C_{/D}$
- $f_* : C_{/C} \longrightarrow C_{/D}$
- $\mathbf{y}$
- $X \times_Y Z$
- $\ulcorner _ \urcorner$
- "Gap map"
- "Locally cartesian closed"

And here is a list of category theory concepts that we will explain more carefully.

6.1 Categories with families

We begin by reformulating the definition of a model of extensional type theory from Chapter 3 into a more palatable form. Our starting point is the following observation:

Lemma 6.1.1. *If $\mathcal{M}$ is a model of ETT (Definition 3.4.2), then $\mathsf{Cx}_{\mathcal{M}}$ is a category where the hom-sets $\hom_{\mathsf{Cx}_{\mathcal{M}}}(\Gamma, \Delta)$ are given by $\mathsf{Sb}_{\mathcal{M}}(\Gamma, \Delta)$.*

Proof. This is very nearly a tautology. We must construct a composition operation for morphisms along with an identity arrow and show that they satisfy the expected properties. However, the composition operation for substitutions $\circ_{\mathcal{M}}$ and the identity substitution $\mathbf{id}_{\mathcal{M}}$ are defined so as to precisely fit this specification. □

The immediate pay-off of this observation is that we may collapse 7 points in Definition 3.4.2 (two sets, two operations, and three equations) into a single structure. What is less obvious—though more important—is that a good number of the other points of Definition 3.4.2 can also be rephrased and compacted in this manner. In particular, category theory is designed for naturality and therefore is exceptionally well-suited to capturing the aspects of type theory based on naturality:

workshop this phrasing

Slogan 6.1.2. *Re-expressing the connectives of type theory using category theory allows us to automatically obtain concise descriptions which bake in naturality requirements.*

We shall split up the process of formulating these categorical versions across this and the following three sections (Sections 6.1 to 6.4), roughly mirroring the progression found in Sections 2.3 to 2.6.

In this section we reformulate the portions of Definition 3.4.2 that do not involve specific connectives into more categorical terms. In so doing, we shall arrive at the definition of a *category with families* [Dyb96]—or, rather, the equivalent notion of *natural model* [Awo18]. Coincidentally, this discussion closely parallels the path taken by Dybjer [Dyb96] when he introduced the notion, but many of the concrete results are due to Awodey [Awo18].

6.1.1 *Contexts and substitutions*

We begin by analyzing the operations dealing only with contexts and substitutions without touching on types and terms. We have already seen that the majority of these operations and equations shape $\mathsf{Cx}_{\mathcal{M}}$ into a category. One further point remains to be elucidated:

Lemma 6.1.3. *The operations and equations for the empty context $\mathbf{1}_{\mathcal{M}}$ are precisely equivalent to the requirement that $\mathsf{Cx}_{\mathcal{M}}$ possess a chosen terminal object.*

Proof. Recall that a terminal object $X : C$ is one such that $\hom_C(Y, X) \cong \{\star\}$ for all objects Y. Inspecting the rules governing $\mathbf{1}_{\mathcal{M}}$, we see that $!_{\mathcal{M}}$ furnishes an inverse to the unique map $\hom_{\mathsf{Cx}_{\mathcal{M}}}(\Gamma, \mathbf{1}_{\mathcal{M}}) \to \{\star\}$. □

6.1.2 *Types and terms as presheaves*

In order to consolidate other aspects of $\mathcal{M}$, we must deal with $\mathsf{Ty}_{\mathcal{M}}(-)$ and $\mathsf{Tm}_{\mathcal{M}}(-,-)$. These are a bit less straightforward than contexts and substitutions. While $\mathsf{Cx}_{\mathcal{M}}$ being a category was hard to miss, one must look a little harder to find the clean categorical descriptions of $\mathsf{Ty}_{\mathcal{M}}(-)$ and (especially) $\mathsf{Tm}_{\mathcal{M}}(-,-)$.

Lemma 6.1.4. *The family of sets* $\mathsf{Ty}_{\mathcal{M}}(-)$ *and the operations and equations for applying substitutions to types* $-[-]_{\mathcal{M}}$ *are precisely equivalent to a presheaf over* $\mathsf{Cx}_{\mathcal{M}}$.

Proof. Let us recall that a presheaf $X : C^{\mathrm{op}} \to \mathbf{Set}$ consists of (1) a family of sets $X(c)$ for each $c : C$, (2) a collection of functions $X(f) : X(c') \to X(c)$ for each $f : c \longrightarrow c'$, (3) equations stating that $X(\mathrm{id})$ is the identity function and $X(f \circ g) = X(g) \circ X(f)$. Reviewing the operations and equations for $\mathsf{Ty}_{\mathcal{M}}(-)$ and $-[-]_{\mathcal{M}}$, we find a perfect match. □

A similar story can be told for $\mathsf{Tm}_{\mathcal{M}}(-,-)$ and substitution on terms, but there is a wrinkle: since terms are indexed over both context and types, $\mathsf{Tm}_{\mathcal{M}}(-,-)$ is not a presheaf over $\mathsf{Cx}_{\mathcal{M}}$. Instead, it lives over the *category of elements* $\int_{\Gamma:\mathsf{Cx}_{\mathcal{M}}} \mathsf{Ty}_{\mathcal{M}}(\Gamma)$:

Definition 6.1.5. If C is a category and $X : \mathbf{Pr}(C)$, the category of elements $\int_C X$ is defined as follows:

- Objects are pairs $(c : C, x : X(c))$.
- A morphism $(c, x) \longrightarrow (d, y)$ consists of a morphism $f : c \longrightarrow d$ such that $X(f)\, y = x$.
- Composition and identity are defined using the corresponding operations from C.

See Riehl [Rie16, Section 2.4] for more details.

To gain intuition for this definition, let us consider the specific case which motivated it: $\int_{\Gamma:\mathsf{Cx}_{\mathcal{M}}} \mathsf{Ty}_{\mathcal{M}}(\Gamma)$. Its objects are pairs of a context Γ and a type $A : \mathsf{Ty}_{\mathcal{M}}(\Gamma)$ and morphisms $(\Delta, B) \longrightarrow (\Gamma, A)$ are substitutions $\gamma : \mathsf{Sb}_{\mathcal{M}}(\Delta, \Gamma)$ such that $B = A[\gamma]$. Such pairs and substitutions are precisely the inputs to $\mathsf{Tm}_{\mathcal{M}}(-,-)$ and so we conclude the following:

Lemma 6.1.6. *The family of sets* $\mathsf{Tm}_{\mathcal{M}}(-,-)$ *and the operations and equations for applying substitution to terms* $-[-]_{\mathcal{M}}$ *are precisely equivalent to a presheaf over* $\int_{\Gamma:\mathsf{Cx}_{\mathcal{M}}} \mathsf{Ty}_{\mathcal{M}}(\Gamma)$.

Exercise 6.1. Carefully check that the data around $\mathsf{Tm}_{\mathcal{M}}(-,-)$ arranges into the promised presheaf.

A digression: slicing presheaf categories A classical result in category theory is that there exists an equivalence between $\mathbf{Pr}(C)_{/X}$ and $\mathbf{Pr}(\int_C X)$; most often, this is used to prove that the slice category of a presheaf category is itself a presheaf category. For our purposes it is often vital to pass between these perspectives when studying $\mathsf{Tm}_{\mathcal{M}}(-,-)$ and so we include both a sketch of this proof and note its specialization to $\mathsf{Tm}_{\mathcal{M}}(-,-)$.

Surely there is a reference for this somewhere?

First, we define the functor U sending $\mathbf{Pr}(C)_{/X}$ to $\mathbf{Pr}(\int_C X)$. This functor sends $\sigma : Y \longrightarrow X$ to the following presheaf over $\int_C X$:

$$U(\sigma)\,(c,x) = \{y : Y(c) \mid \sigma_c(y) = x\}$$

Given $\alpha : \hom_{\mathbf{Pr}(C)_{/X}}(\sigma,\tau)$, the functorial action $U(\alpha)$ is defined as follows:

$$U(\alpha)\,(c,x)\,y = \alpha_c\,y$$

In particular, since $\tau \circ \alpha = \sigma$ and $\sigma_c(y) = x$ by definition of $U(\sigma)$, we must have $\tau_c(\alpha_c\,y) = x$ so that this definition is well-typed.

Exercise 6.2. Check that U satisfies the equations necessary to be a functor.

Exercise 6.3. Argue that U is fully faithful.

In light of Exercise 6.3, to check that U is an equivalence, it suffices to check that it is essentially surjective. That is, we must show that if $Y : \mathbf{Pr}(\int_C X)$ then there exists $\sigma : Y_0 \longrightarrow X$ such that $U(\sigma) \cong Y$. Fixing $Y : \mathbf{Pr}(\int_C X)$, we define σ and Y_0 as follows:

$$Y_0\,c = \textstyle\sum_{x:X(c)} Y(c,x) \qquad \sigma_c = \pi_1$$

We leave it to the reader to carry out the routine verification that Y_0 is functorial and σ is natural. We may now compute $U(\sigma)$:

$$U(\sigma)\,(c,x) = \{(x_0,y) : \textstyle\sum_{x_0:X(c)} Y(c,x_0) \mid x_0 = x\} \cong Y(c,x)$$

It is routine to check that these bijections organize into the required natural isomorphism. All told, we conclude the following:

Theorem 6.1.7. *U is an equivalence.*

We may specialize this discussion to $\mathsf{Ty}_{\mathcal{M}} : \mathbf{Pr}(\mathsf{Cx}_{\mathcal{M}})$ and $\mathsf{Tm}_{\mathcal{M}} : \mathbf{Pr}(\int_{\mathsf{Cx}_{\mathcal{M}}} \mathsf{Ty}_{\mathcal{M}})$:

Corollary 6.1.8. *The family of sets $\mathsf{Tm}_{\mathcal{M}}(-,-)$ and the operations and equations for applying substitution to terms $-[-]_{\mathcal{M}}$ are precisely equivalent to an object in $\mathbf{Pr}(\mathsf{Cx}_{\mathcal{M}})_{/\mathsf{Ty}_{\mathcal{M}}}$.*

We denote the induced object of the slice category $\pi : \mathsf{Tm}^{\bullet}_{\mathcal{M}} \longrightarrow \mathsf{Ty}_{\mathcal{M}}$ and it is explicitly given as follows:

$$\mathsf{Tm}^{\bullet}_{\mathcal{M}}\,\Gamma = \textstyle\sum_{A:\mathsf{Ty}_{\mathcal{M}}(\Gamma)} \mathsf{Tm}_{\mathcal{M}}(\Gamma,A) \qquad \pi_\Gamma = \pi_1$$

6.1.3 *The categorical formulation of context extension*

With $\mathsf{Tm}^\bullet_{\mathcal{M}}$ to hand, we reformulate one final piece of Definition 3.4.2: context extensions. This definition is a bit more complex since it mixes together all four of contexts, substitutions, terms and types. However, our discussion of the mapping-in property of context extension in Section 2.4.2 should lead us to guess that it too can be expressed categorically.

Definition 6.1.9. If $\alpha : X \longrightarrow Y$ where $X, Y : \mathbf{Pr}(\mathcal{C})$, we say α is *representable* whenever the pullback $\mathbf{y}(c) \times_Y X$ is representable for every $\mathbf{y}(c) \longrightarrow Y$.

In other words, a natural transformation is representable if for every $y : \mathbf{y}(c) \longrightarrow Y$ there exists some $c_y : \mathcal{C}$ along with morphisms $p_y : c_y \longrightarrow c$ and $q_y : \mathbf{y}(c_y) \longrightarrow X$ such that the following diagram is a pullback:

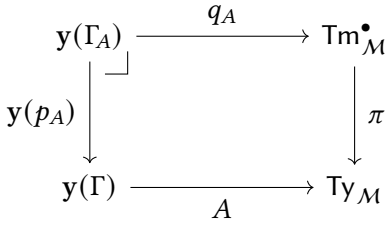
$$\begin{array}{ccc} \mathbf{y}(c_y) & \xrightarrow{q_y} & X \\ {\scriptstyle \mathbf{y}(p_y)}\downarrow & & \downarrow{\scriptstyle \alpha} \\ \mathbf{y}(c) & \xrightarrow[y]{} & Y \end{array} \tag{6.1}$$

We call a particular choice of triples (c_y, p_y, q_y) a *representability structure* on α. Representability structures are all suitably uniquely isomorphic to one another, but need not be equal (in much the same way that limits are determined only up to unique isomorphism).

Lemma 6.1.10. *The operations and equations around context extension (including the variable term and the weakening substitution) in $\mathcal{M}$ are precisely equivalent to requiring a representability structure on $\pi : \mathsf{Tm}^\bullet_{\mathcal{M}} \longrightarrow \mathsf{Ty}_{\mathcal{M}}$.*

Proof. Let us begin by unfolding what is involved in a representability structure on π and, in particular, what the universal property of Diagram 6.1 determines when specialized to π. First note that a morphism $A : \mathbf{y}(\Gamma) \longrightarrow \mathsf{Ty}_{\mathcal{M}}$ is equivalent by Yoneda to a type $A : \mathsf{Ty}_{\mathcal{M}}(\Gamma)$. Accordingly, a representability structure is an assignment of every Γ and $A : \mathsf{Ty}_{\mathcal{M}}(\Gamma)$ to a triple $(\Gamma_A : \mathsf{Cx}_{\mathcal{M}}, p_A : \Gamma_A \longrightarrow \Gamma, q_A : \mathbf{y}(\Gamma_A) \longrightarrow \mathsf{Tm}^\bullet_{\mathcal{M}})$ such

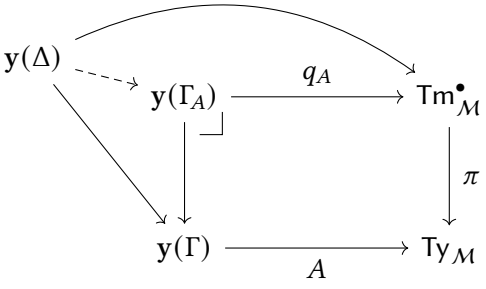
that the following square commutes and is a pullback:



Let us apply the Yoneda lemma once more to see that q_A is equivalent to a pair $A' : \mathsf{Ty}_{\mathcal{M}}(\Gamma_A), q : \mathsf{Tm}_{\mathcal{M}}(\Gamma_A, A')$. Moreover, by the naturality of the Yoneda lemma and the commutation of the above square, we conclude that $\pi\, \Gamma_A\, (A', q) = A[p_A]_{\mathcal{M}}$ and so, unfolding the left-hand side of this equality, $A' = A[p_A]_{\mathcal{M}}$. Accordingly, the data of the commuting square corresponds to $\Gamma._{\mathcal{M}} A$, $\mathbf{p}_{\mathcal{M}}$, and $\mathbf{q}_{\mathcal{M}}$.

What's left is to analyze the universal property of this pullback square. As a general matter, a square in a presheaf category has the universal property of a pullback square just when it has the correct universal property with respect to *representable* presheaves. There are several ways to prove this, but perhaps the simplest is to recall that (co)limits in presheaves are computed pointwise and to apply the Yoneda lemma.

Accordingly, the fact that the above commuting square is a pullback amounts to the following: for every $(\Delta : \mathsf{Cx}_{\mathcal{M}}, \mathbf{y}(\Delta) \longrightarrow \mathbf{y}(\Gamma), \mathbf{y}(\Delta) \longrightarrow \mathsf{Tm}^\bullet_{\mathcal{M}})$ fitting into the below diagram, there is a unique dashed arrow making the diagram commute:



Applying the Yoneda lemma, we see that the maps $\mathbf{y}(\Delta) \longrightarrow \mathbf{y}(\Gamma)$ and $\mathbf{y}(\Delta) \longrightarrow \mathsf{Tm}^\bullet_{\mathcal{M}}$ correspond to a substitution $\gamma : \mathsf{Sb}_{\mathcal{M}}(\Delta, \Gamma)$ and a term $a : \mathsf{Tm}_{\mathcal{M}}(\Delta, A[\gamma]_{\mathcal{M}})$. Finally, we see that the dashed arrow encodes $\gamma._{\mathcal{M}} a$ and the commutation of the diagram and the unicity of the dashed arrow correspond to the equations around $\gamma._{\mathcal{M}} a$, completing the proof. □

We emphasize that while the reshuffling was more involved to relate representability structures and context extensions, the two notions are completely equivalent. The

purpose of this reformulation is not to favor one over the other, but to have both available for when the representability structure notion is easier (*e.g.*, in Section 6.6) and for when the $((-.\mathcal{M}-), \mathbf{p}_{\mathcal{M}}, \mathbf{q}_{\mathcal{M}})$ is easier (eg, in Section 6.7).

Exercise 6.4. Suppose that $\Delta, \Gamma : \mathsf{Cx}_{\mathcal{M}}$ and that $A : \mathsf{Ty}_{\mathcal{M}}(\Gamma)$ and $\gamma : \mathsf{Sb}_{\mathcal{M}}(\Delta, \Gamma)$. Show that the following is a pullback diagram:

$$\begin{array}{ccc} \Delta.\mathcal{M}A[\gamma]_{\mathcal{M}} & \xrightarrow{\gamma.\mathcal{M}A} & \Gamma.\mathcal{M}A \\ \downarrow{\scriptstyle \mathbf{p}_{\mathcal{M}}} & & \downarrow{\scriptstyle \mathbf{p}_{\mathcal{M}}} \\ \Delta & \xrightarrow[\gamma]{} & \Gamma \end{array}$$

(Hint: there is a slick proof based on the 3-for-2 lemma for pullbacks and Lemma 6.1.10.)

6.1.4 *The definition of a cwf*

Collecting all these reformulations together, we arrive at the definition of a cwf [Dyb96] or, more precisely, a cwf recast into the language of natural models [Awo18]:

Definition 6.1.11. A category with families (cwf) consists of the following data:

- A category C
- A chosen terminal object $\mathbf{1} : C$
- A pair of presheaves and a natural transformation $\pi_C : \mathsf{Tm}^{\bullet}_C \longrightarrow \mathsf{Ty}_C$
- A representability structure on π_C

Standardize terminology around “bare type theory” and cwf (not natural model).

Theorem 6.1.12. *A category with families is equivalent to a model of type theory without any connectives.*

Remark 6.1.13. Different authors package the data of a model (or a cwf) in different ways. Since they are all equivalent, these differences are fundamentally unimportant. However, they can be useful in different situations and it is important to feel comfortable passing between a fully unfolded definition of a model (Definition 3.4.2) or a more compressed variant (Definition 6.1.11). Not only because many variations appear in

Connective	Unfolded structure	Categorical version
The unit type (**Unit**)	Structure 6.2.2	Lemma 6.2.5
The equality type (**Eq**)	Structure 6.2.6	Lemma 6.2.9
Dependent products (**Π**)	Structure 6.2.18	Lemma 6.2.20
Dependent sums (**Σ**)	Exercise 6.10	Lemma 6.2.21
Booleans (**Bool**)	Structure 6.3.2	Lemma 6.3.7
Coproducts (+)	Structure 6.3.8	Lemma 6.3.12
The empty type (**Void**)	Structure 6.3.14	Lemma 6.3.15
The natural numbers (**Nat**)	Structure 6.3.16	Lemma 6.3.25
A single universe ($\mathbf{U}_0$)	Structure 6.4.17	Theorem 6.4.22
A universe hierarchy ($\mathbf{U}_i$)	Exercise 6.20	Lemma 6.4.23

Figure 6.1: Table of categorical reformulation of the connectives of type theory

the literature, but because often one formulation is more perspicacious in a particular situation. ◇

We refer to a model of type theory without connectives as a *model of base type theory*. Our goal is to now explore how to reformulate the specification of various connectives from Definition 3.4.2 on top of the definition of a cwf. Since we will have a great deal of data to manipulate when discussing equipping models of base type theory with connectives, we take a moment to discuss the global structure of this process. Essentially every subsection of Sections 6.2 to 6.4 will deal with a single connective and in each we will follow the same process. First, we begin by recalling the relevant portion of Definition 3.4.2 and then work to reformulate it into a more concise categorical definition. The final result will be a statement of the form "a model of base type theory supports an interpretation of the connective Θ just when it comes equipped with the following categorical structures". For ease of reference, we have gathered a table describing where each structure is introduced and the result where it is reformulated in Figure 6.1.

As in Chapter 2, once the substitution calculus is in place the connectives of type theory are essentially orthogonal and may be introduced in any order. An exception to this pattern is **U**, as the closure conditions required of the universe are of course sensitive to the other connectives available within the theory. When dealing with individual connectives, it is frequently convenient to consider models of type theory which support only a specific subset of connectives. In particular, we may define a model of type theory with *e.g.*, only **Π** and **Unit** as a base model together with the structures in Definition 3.4.2 specifically related to *e.g.*, **Π** and **Unit**. The main result of the following sections may be summarized by the following "theorem schema":

Theorem 6.1.14. *A model of type theory with any set of connectives consists of (1) a category with families (Definition 6.1.11) and (2) the categorical reformulation of structures pertaining to each of those connectives.*

In particular, a model of type theory with **Π** and **Unit** consists of Definition 6.1.11 satisfying the additional requirements described in Lemmas 6.2.5 and 6.2.20.

6.2 *Pullback squares and* **Π**, **Σ**, **Eq**, **Unit**

We now continue our quest to reformulate Definition 3.4.2 in more categorical terms by turning our attention to connectives with *mapping-in* specifications: **Π**, **Unit**, **Σ**, and **Eq**. As with contexts and substitutions, our goal is to find equivalent "repackaged" definitions which consolidate the operations and equations for each connective.

Notation 6.2.1. In the previous section, we were careful to subscript $\mathsf{Ty}_{\mathcal{M}}(-)$, $\mathsf{Tm}_{\mathcal{M}}(-,-)$, *etc.* with $\mathcal{M}$ to emphasize that they were part of the data of some model $\mathcal{M}$. However, in this section the notational burden of subscripting virtually every operation with $\mathcal{M}$ outweighs the benefits of being explicit. Accordingly, within this section we fix a model $\mathcal{M}$ and write *e.g.* Ty rather than $\mathsf{Ty}_{\mathcal{M}}$.

6.2.1 *The unit type*

We begin with **Unit**, as it is the simplest case. Let us begin by recalling the relevant portions of Definition 3.4.2 which are required to interpret the rules of the **Unit** (Section 2.4):

Structure 6.2.2. A unit type structure on $\mathcal{M}$ consists of the following:

- An operation $\mathbf{Unit} : \{\Gamma : \mathsf{Cx}\} \to \mathsf{Ty}(\Gamma)$
- For every substitution $\gamma : \mathsf{Sb}(\Delta, \Gamma)$ an equation $\mathbf{Unit} = \mathbf{Unit}[\gamma]$
- A collection of isomorphisms $\iota : (\Gamma : \mathsf{Cx}) \to \mathsf{Tm}(\Gamma, \mathbf{Unit}) \cong \{\star\}$
- For every substitution $\gamma : \mathsf{Sb}(\Delta, \Gamma)$ an equation $\iota_\Delta \circ \gamma^* = \iota_\Gamma$.[1]

Let us begin by noting that our prior intuition that these equations enforced naturality was justified:

[1] This requirement is vacuous since both sides are maps into $\{\star\}$, but we include it for consistency.

Lemma 6.2.3. **Unit** *and the associated equations form a natural transformation* $\mathbf{Unit} : \mathbf{1} \longrightarrow \mathsf{Ty}$.

To recast ι into a natural transformation, we note that there is a presheaf sending Γ to $\mathsf{Tm}(\Gamma, \mathbf{Unit})$. In fact, one can construct this functor by pulling back $\mathsf{Tm}^\bullet \longrightarrow \mathsf{Ty}$ along the map $\mathbf{Unit} : \mathbf{1} \longrightarrow \mathsf{Ty}$. In light of this, we denote this presheaf by $\mathbf{Unit}^*\mathsf{Tm}^\bullet$.

Exercise 6.5. Check that $\mathsf{Tm}^\bullet \times_{\mathsf{Ty}} \mathbf{1} \cong \mathsf{Tm}(-, \mathbf{Unit}\ -)$.

Lemma 6.2.4. ι *and its equations form a natural isomorphism* $\mathbf{Unit}^*\mathsf{Tm}^\bullet \cong \mathbf{1}$.

All told, we can replace our original four points with two:

- a natural transformation $\mathbf{Unit} : \mathbf{1} \longrightarrow \mathsf{Ty}$,
- a natural isomorphism $\mathbf{Unit}^*\mathsf{Tm}^\bullet \cong \mathbf{1}$.

In fact, we can bundle these two points into one:

Lemma 6.2.5 (Categorical reformulation of **Unit**)**.** *A unit type structure on* $\mathcal{M}$ *is equivalent to a choice of pullback of the following shape:*

$$\begin{array}{ccc} \mathbf{1} & \longrightarrow & \mathsf{Tm}^\bullet \\ \downarrow & \lrcorner & \downarrow{\scriptstyle \pi} \\ \mathbf{1} & \longrightarrow & \mathsf{Ty} \end{array} \tag{6.2}$$

Proof. The natural transformation $\mathbf{Unit} : \mathbf{1} \longrightarrow \mathsf{Ty}$ is precisely what is required to construct the base of this pullback and the natural isomorphism is equivalent to the data of the top map together with the property that it forms a pullback. □

This result leads us to a reformulation of our slogan for specifying types with a mapping-in universal property: they ought to be determined by a pullback square involving π. Before crystallizing this slogan, we consider a slightly less trivial example to see the pattern more clearly.

6.2.2 *The extensional equality type*

We next turn our attention to the extensional equality type. Once more, we begin by isolating the subset of Definition 3.4.2 required to interpret the rules of **Eq** given in Section 2.4.4.

Structure 6.2.6. An equality structure on $\mathcal{M}$ consists of the following operations and equations:

- An operation

$$\mathbf{Eq} : \{\Gamma : \mathsf{Cx}\}(A : \mathsf{Ty}(\Gamma)) \to \mathsf{Tm}(\Gamma, A) \to \mathsf{Tm}(\Gamma, A) \to \mathsf{Ty}(\Gamma)$$

- For every substitution $\gamma : \mathsf{Sb}(\Delta, \Gamma)$ along with $A : \mathsf{Ty}(\Gamma)$ and $a, b : \mathsf{Tm}(\Gamma, A)$, an equation

$$\mathbf{Eq}(A[\gamma], a[\gamma], b[\gamma]) = \mathbf{Eq}(A, a, b)[\gamma]$$

- A collection of isomorphisms

$$\iota : (\Gamma : \mathsf{Cx})(A : \mathsf{Ty}(\Gamma))(a, b : \mathsf{Tm}(\Gamma, A)) \to \mathsf{Tm}(\Gamma, \mathbf{Eq}(A, a, b)) \cong \{\star \mid a = b\}$$

- For every substitution $\gamma : \mathsf{Sb}(\Delta, \Gamma)$ and $A : \mathsf{Ty}(\Gamma)$ and $a, b : \mathsf{Tm}(\Gamma, A)$, an equation

$$\iota_\Delta(A[\gamma], a[\gamma], b[\gamma]) \circ \gamma^* = \iota_\Gamma(A, a, b)$$

Once more, we wish to parlay these operations and equations into natural transformations into Ty and $\mathsf{Tm}^\bullet$. However, this time there is non-trivial formation data: A along with a, b. Accordingly, the domain of natural transformation **Eq** is not $\mathbf{1}$ like with **Unit**, but instead a presheaf whose value at Γ is $\sum_{A:\mathsf{Ty}(\Gamma)} \mathsf{Tm}(\Gamma, A) \times \mathsf{Tm}(\Gamma, A)$. We can construct this presheaf out of Ty and $\mathsf{Tm}^\bullet$:

Exercise 6.6. Show $(\mathsf{Tm}^\bullet \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet)\Gamma \cong \sum_{A:\mathsf{Ty}(\Gamma)} \mathsf{Tm}(\Gamma, A) \times \mathsf{Tm}(\Gamma, A)$.

In light of the above exercise, the following is nearly a tautology.

Lemma 6.2.7. *The operation* **Eq** *and the equations around it are equivalent to a natural transformation* $\mathsf{Tm}^\bullet \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet \longrightarrow \mathsf{Ty}$.

We next turn to the isomorphism ι. This step requires some creativity, as both $\mathsf{Tm}(\Gamma, \mathbf{Eq}(A, a, b))$ and $\{\star \mid a = b\}$ depend on Γ, A, a, and b. Accordingly, ι is a family of isomorphisms between objects indexed not just over the context but on the formation data as well; it consists not merely of a natural isomorphism in $\mathbf{Pr}(\mathsf{Cx})$ but instead in $\mathbf{Pr}(\int_{\mathsf{Cx}} \mathsf{Tm}^\bullet \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet)$. Accordingly, we are asking for a natural transformation between the following two presheaves:

$$X(\Gamma, A, a, b) = \mathsf{Tm}(\Gamma, \mathbf{Eq}(A, a, b)) \qquad Y(\Gamma, A, a, b) = \{\star \mid a = b\}$$

Lemma 6.2.8. *ι organizes into an isomorphism $X \cong Y$ in* $\mathbf{Pr}(\int_{\mathsf{Cx}} \mathsf{Tm}^\bullet \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet)$.

Our final step is to use the equivalence $\mathbf{Pr}(\int_{\mathsf{Cx}} \mathsf{Tm}^\bullet \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet) \simeq \mathbf{Pr}(\mathsf{Cx})_{/\mathsf{Tm}^\bullet \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet}$ to present this isomorphism in $\mathbf{Pr}(\mathsf{Cx})$.

Exercise 6.7. Under the above equivalence, show that X is isomorphic to the left hand vertical map of the following diagram:

$$\begin{array}{ccc} \mathbf{Eq}^*\mathsf{Tm}^\bullet & \longrightarrow & \mathsf{Tm}^\bullet \\ \downarrow & & \downarrow \\ \mathsf{Tm}^\bullet \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet & \xrightarrow[\mathbf{Eq}]{} & \mathsf{Ty} \end{array} \tag{6.3}$$

Exercise 6.8. Under the above equivalence, show that Y is isomorphic to the diagonal map $\mathsf{Tm}^\bullet \longrightarrow \mathsf{Tm}^\bullet \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet$.

Accordingly, ι determines a natural isomorphism between $\mathbf{Eq}^*\mathsf{Tm}^\bullet \cong \mathsf{Tm}^\bullet$ fitting into a commuting triangle:

$$\begin{array}{ccc} \mathsf{Tm}^\bullet & \longrightarrow & \mathbf{Eq}^*\mathsf{Tm}^\bullet \\ & \searrow \quad \swarrow & \\ & \mathsf{Tm}^\bullet \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet & \end{array}$$

Let us recall that this top map has a recognizable name: it is the natural transformation corresponding to **refl**. If we paste this commuting triangle onto the end of Diagram 6.3, we arrive at the following characterization of extensional equality types:

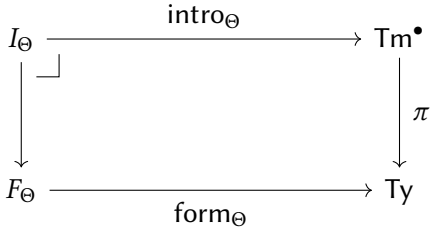
Lemma 6.2.9 (Categorical reformulation of **Eq**). *An equality structure on $\mathcal{M}$ is equivalent to a choice of pullback square of the following form:*

$$\begin{array}{ccc} \mathsf{Tm}^\bullet & \xrightarrow{\mathbf{refl}} & \mathsf{Tm}^\bullet \\ \downarrow{\scriptstyle \delta} & & \downarrow{\scriptstyle \pi} \\ \mathsf{Tm}^\bullet \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet & \xrightarrow[\mathbf{Eq}]{} & \mathsf{Ty} \end{array}$$

In fact, here we can see all the key elements of the equality type at play: the domain and codomain of the left map is the introduction and formation data of **Eq** with the top

and bottom horizontal maps encoding the introduction and formation rules. Finally, the fact that the square is a pullback encodes the elimination principle (along with its β and η equations). All told, we arrive at a categorical version of Slogan 2.4.4:

Slogan 6.2.10. *A connective Θ with a mapping-in universal property is determined by a choice of pullback of the following shape:*



Here F_Θ encodes the formation data of Θ, I_Θ the introduction data, and the top and bottom maps the introduction and formation operations, respectively. The elimination rule along with all the equations are handled by naturality and the universal property of a pullback.

6.2.3 *An interlude: polynomial functors*

Our next goal will be to apply Slogan 6.2.10 to **Π** and **Σ**, but these types are substantially more complicated than **Eq** and **Unit**. The wrinkle is the formation and introduction data involve premises which hypothesize over variables. For instance, the formation data of both **Π** and **Σ** are presheaves of the following shape:

$$\Gamma \mapsto \textstyle\sum_{A:\mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)$$

We now show that, remarkably, operations like these—those which hypothesize over a variable—also admit an elegant description within $\mathbf{Pr}(\mathsf{Cx})$. First, we lay some groundwork. We begin with the following result (see, for instance, Awodey [Awo10, Corollary 9.17]).

Lemma 6.2.11. *If $f : C \longrightarrow \mathcal{D}$ then $f^* : \mathbf{Pr}(\mathcal{D}) \longrightarrow \mathbf{Pr}(C)$ has a right adjoint f_*.*

Theorem 6.2.12. *The pullback functor $f^* : \mathbf{Pr}(C)_{/Y} \longrightarrow \mathbf{Pr}(C)_{/X}$ admits a right adjoint f_*.*

Proof. Passing along the equivalences $\mathbf{Pr}(C)_{/X} \simeq \mathbf{Pr}(\int X)$ and $\mathbf{Pr}(C)_{/Y} \simeq \mathbf{Pr}(\int Y)$, we must show that the precomposition functor $(\int_C f)^* : \mathbf{Pr}(\int Y) \to \mathbf{Pr}(\int X)$ has a right adjoint. We now apply Lemma 6.2.11. □

We now show that we can model "a type or term in an extended context" using π_*.

Notation 6.2.13. We write X^* for the pullback functor $X \longrightarrow \mathbf{1}$ or, equivalently, the functor $Y \mapsto X \times Y$. Furthermore, we write $Y_!$ for the forgetful functor $C_{/Y} \longrightarrow C$ (the left adjoint to Y^*).

Definition 6.2.14. If $f : X \longrightarrow Y$ is a map in $\mathbf{Pr}(C)$ the *polynomial functor over* f $\mathbf{P}_f : \mathbf{Pr}(C) \longrightarrow \mathbf{Pr}(C)$ is defined as follows:

$$\mathbf{P}_f = Y_! \circ f_* \circ X^*$$

Lemma 6.2.15 (Awodey [Awo18, Proposition 6])**.** *There is an isomorphism between* $\mathbf{P}_\pi(\mathsf{Ty})\,\Gamma$ *and sets of pairs* $\sum_{A:\mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)$.

Proof. We prove this through the Yoneda lemma:

$$\mathbf{P}_\pi(\mathsf{Ty})\,\Gamma \cong \hom_{\mathbf{Pr}(\mathsf{Cx})}(\mathbf{y}(\Gamma), \mathbf{P}_\pi(\mathsf{Ty}))$$

Let us break $\hom_{\mathbf{Pr}(\mathsf{Cx})}(\mathbf{y}(\Gamma), \mathbf{P}_\pi(\mathsf{Ty}))$ into two components using the identification $\mathbf{P}_\pi(\mathsf{Ty}) = \mathsf{Ty}_!\pi_*(\mathsf{Tm}^\bullet)^*\mathsf{Ty}$: a morphism $A : \mathbf{y}(\Gamma) \longrightarrow \mathsf{Ty}$ (equivalently, an element of $\mathsf{Ty}(\Gamma)$) and a morphism $\hom_{\mathbf{Pr}(\mathsf{Cx})_{/\mathsf{Ty}}}(A, \pi_*(\mathsf{Tm}^\bullet)^*\mathsf{Ty})$. Let us further investigate the second morphism:

$$\begin{aligned}
&\hom_{\mathbf{Pr}(\mathsf{Cx})_{/\mathsf{Ty}}}(A, \pi_*(\mathsf{Tm}^\bullet)^*\mathsf{Ty}) \\
&\cong \hom_{\mathbf{Pr}(\mathsf{Cx})_{/\mathsf{Tm}^\bullet}}(\mathbf{y}(\Gamma) \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet, (\mathsf{Tm}^\bullet)^*\mathsf{Ty}) \\
&\cong \hom_{\mathbf{Pr}(\mathsf{Cx})}(\mathbf{y}(\Gamma.A), \mathsf{Ty}) \\
&\cong \mathsf{Ty}(\Gamma.A)
\end{aligned}$$

□

We can replay exactly this proof with $\mathsf{Tm}^\bullet$ to obtain the following:

Lemma 6.2.16. $\mathbf{P}_\pi(\mathsf{Tm}^\bullet)\,\Gamma \cong \sum_{A:\mathsf{Ty}(\Gamma)} \sum_{B:\mathsf{Ty}(\Gamma.A)} \mathsf{Tm}(\Gamma.A, B)$.

One last result is necessary: we wish to find a presheaf which encodes the formation data for a Σ-type:

$$\sum_{A:\mathsf{Ty}(\Gamma)} \sum_{B:\mathsf{Ty}(\Gamma.A)} \sum_{a:\mathsf{Tm}(\Gamma,A)} \mathsf{Tm}(\Gamma, B[\mathbf{id}.a])$$

This is slightly more complex (Awodey [Awo18] uses the *internal language* to give a succinct description of this presheaf). The most straightforward approach is to define such a presheaf manually:

$$P(\Gamma) = \sum_{A:\mathsf{Ty}(\Gamma)} \sum_{B:\mathsf{Ty}(\Gamma.A)} \sum_{a:\mathsf{Tm}(\Gamma,A)} \mathsf{Tm}(\Gamma, B[\mathbf{id}.a])$$

Exercise 6.9. Define the functorial action of P using substitution.

We note—more for completeness than necessity—that it is possible to build this presheaf just using $\mathbf{P}_\pi$ and other purely categorical constructs:

Lemma 6.2.17 (Awodey [Awo18, Remark 13], Uemura [Uem21, Lemma 6.2.1])**.** *There is a canonical square of the following form and, moreover, it is a pullback:*

$$\begin{array}{ccc} P & \longrightarrow & \mathsf{Tm}^\bullet \\ \downarrow & \lrcorner & \downarrow{\scriptstyle \pi} \\ \mathbf{P}_\pi(\mathsf{Ty}) \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet & \xrightarrow[\epsilon]{} & \mathsf{Ty} \end{array} \tag{6.4}$$

Here ϵ is the counit of the adjunction $\pi^* \dashv \pi_*$.

Proof. For concision, we write $X = \mathbf{P}_\pi(\mathsf{Ty}) \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet$ within this proof. First, we note that the canonical square is defined using the evident projections from P. To show that this square is a pullback, we use the Yoneda lemma to characterize $X \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet$ whereby it will be clear that the unique induced map $P \longrightarrow X \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet$ is an equivalence. To do this, we apply the Yoneda lemma such that it suffices to characterize $\hom(\mathbf{y}(\Gamma), X \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet)$. By the universal property, this consists of the following:

- an element of $\hom(\mathbf{y}(\Gamma), \mathsf{Tm}^\bullet)$ or, equivalently, $B_a : \mathsf{Ty}(\Gamma)$ and $b : \mathsf{Tm}(\Gamma, B_a)$.
- an element of $\hom(\mathbf{y}(\Gamma), \mathbf{P}_\pi(\mathsf{Ty}) \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet)$ or, equivalently, $A : \mathsf{Ty}(\Gamma)$ and $a : \mathsf{Tm}(\Gamma, A)$ along with $B : \mathsf{Ty}(\Gamma.A)$ (the latter by Lemma 6.2.15)
- an equality $B[\mathbf{id}.a] = B_a$. □

We define $\pi \otimes \pi : P \longrightarrow \mathbf{P}_\pi(\mathsf{Ty})$ to be the composite:

$$P \longrightarrow \mathbf{P}_\pi(\mathsf{Ty}) \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet \longrightarrow \mathbf{P}_\pi(\mathsf{Ty})$$

Hereafter we refer to P as $\mathrm{dom}(\pi \otimes \pi)$. This map projects (A, B, a, b) onto (A, B).

6.2.4 *Dependent products and sums*

Having expended the effort to calculate the effect of these polynomial functors in $\mathbf{Pr}(\mathsf{Cx})$, it requires only a little more effort to apply Slogan 6.2.10 to dependent products and sums.

We begin with dependent products. In the now familiar routine, we begin by isolating the structure on a model needed to interpret dependent products.

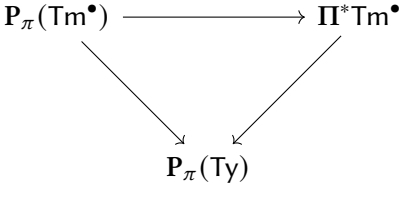
Structure 6.2.18. A dependent product structure on $\mathcal{M}$ consists of the following operations and equations:

- An operator $\mathbf{\Pi} : \{\Gamma : \mathsf{Cx}\}(A : \mathsf{Ty}(\Gamma)) \to \mathsf{Ty}(\Gamma.A) \to \mathsf{Ty}(\Gamma)$
- For every $\gamma : \mathsf{Sb}(\Delta, \Gamma)$ along with $A : \mathsf{Ty}(\Gamma)$ and $B : \mathsf{Ty}(\Gamma.A)$, an equality
$$\mathbf{\Pi}(A, B)[\gamma] = \mathbf{\Pi}(A[\gamma], B[\gamma.A])$$
- A family of isomorphisms:
$$\iota : \{\Gamma : \mathsf{Cx}\}(A : \mathsf{Ty}(\Gamma))(B : \mathsf{Ty}(\Gamma.A)) \to \mathsf{Tm}(\Gamma, \mathbf{\Pi}(A, B)) \cong \mathsf{Tm}(\Gamma.A, B)$$
- For every $\gamma : \mathsf{Sb}(\Delta, \Gamma)$ along with $A : \mathsf{Ty}(\Gamma)$ and $B : \mathsf{Ty}(\Gamma.A)$, an equality
$$\iota(A[\gamma], B[\gamma.A]) \circ \gamma^* = (\gamma.A)^* \circ \iota(A, B)$$

In light of Lemma 6.2.15, we can bundle together $\mathbf{\Pi}$ into a natural transformation:

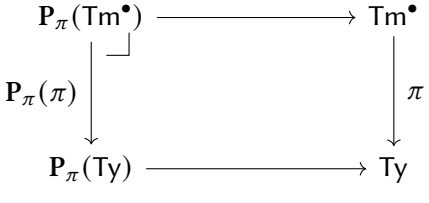
Lemma 6.2.19. *$\mathbf{\Pi}$ and its equation organize into a map $\mathbf{P}_\pi(\mathsf{Ty}) \longrightarrow \mathsf{Ty}$.*

Moreover, by the same reasoning as we applied in the case of **Eq**, the isomorphism ι is equivalent to a natural isomorphism $\mathbf{P}_\pi(\mathsf{Tm}^\bullet) \cong \mathbf{\Pi}^*\mathsf{Tm}^\bullet$ fitting into the following commuting triangle:



All told, we arrive at the following:

Lemma 6.2.20 (Categorical reformulation of $\mathbf{\Pi}$)**.** *A dependent product structure on $\mathcal{M}$ is equivalent to a choice of pullback square of the following shape:*



The bottom morphism of this pullback square corresponds to $\mathbf{\Pi}$ while the top corresponds to the introduction form $\lambda(-)$.

Finally, we content ourselves with providing "the answer" for dependent sums and leaving it to the intrepid reader to fill in the details:

Exercise 6.10. Isolate the operations and equations in the style of Definition 3.4.2 necessary to interpret the rules of dependent sums (Section 2.4.3).

Lemma 6.2.21 (Categorical reformulation of Σ)**.** *$\mathcal{M}$ supports dependent sums if and only if it is equipped with a choice of pullback square of the following shape:*

$$\begin{array}{ccc} \mathrm{dom}(\pi \otimes \pi) & \longrightarrow & \mathsf{Tm}^\bullet \\ \downarrow{\scriptstyle \pi \otimes \pi} & & \downarrow{\scriptstyle \pi} \\ \mathbf{P}_\pi(\mathsf{Ty}) & \longrightarrow & \mathsf{Ty} \end{array}$$

The bottom morphism of this pullback square corresponds to Σ while the top corresponds to the introduction form **pair**.

6.3 *Orthogonality and* Void, Bool, +, Nat

We next turn to connectives without a mapping-out property and, in particular, to **Void**, **Bool**, +, and **Nat**. Following the notation of Section 6.2, we fix a model $\mathcal{M}$ for this section and systematically reformulate the requirements for $\mathcal{M}$ to support these connectives into more categorical terms. As before, we will avoid subscripting each operation with $\mathcal{M}$ as it is the only model we discuss in this section.

In light of Section 2.5, it should come as no surprise that to explain these connectives, we cannot merely rely on Slogan 6.2.10. In fact, we can give a crisp explanation of why this slogan is doomed to failure for **Void**:

Exercise 6.11. Show that there can be no pullback diagram of the following shape:[2]

$$\begin{array}{ccc} \mathbf{0} & \longrightarrow & \mathsf{Tm}^\bullet \\ \downarrow & & \downarrow \\ \mathbf{1} & \longrightarrow & \mathsf{Ty} \end{array}$$

(Hint: use the representability of π.)

[2]The authors once ran headlong into this fact as part of a project with Jonathan Sterling in 2019. The result was an extremely elegant construction which sadly only applied under unsatisfiable hypotheses.

Fortunately, the failure of Slogan 6.2.10 to account for types with mapping-out universal properties provides us with an excuse to introduce the categorical theory of *orthogonality*. Roughly, we shall find that while the above square fails to be a pullback, the degree to which this fails is "invisible" to π. This concretizes an intuition presented in Section 2.5: from the perspective of other types, **Void** is always empty.

Warning 6.3.1. Mirroring Section 2.5, we will start by giving specifications of these types that explicitly include their η laws. In Section 6.3.5 we will show how to modify these specifications to omit η laws, as required in intensional type theory. (Recall from Section 2.5.5 that in extensional type theory, η principles for inductive types can be derived from **Eq**-types.)

6.3.1 *Orthogonality and* **Bool**

We will work our way towards a definition of orthogonal maps by investigating **Bool**. We start with **Bool** over the simpler **Void** as the latter is a bit *too* simple (both trivial formation data and no introduction rules) which makes it difficult to see some parts of the story. Let us begin by recalling the operations and equations governing this type:

Structure 6.3.2. A boolean structure on $\mathcal{M}$ consists of the following operations, equations, and properties:

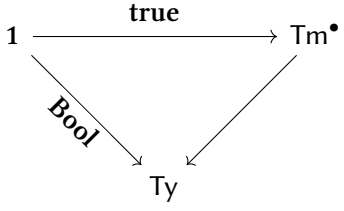
- An operator $\mathbf{Bool} : \{\Gamma : \mathsf{Cx}\} \to \mathsf{Ty}(\Gamma)$
- An equation $\mathbf{Bool}[\gamma] = \mathbf{Bool}$ for every $\gamma : \mathrm{Sb}(\Delta, \Gamma)$.
- A pair of operators $\mathbf{true}, \mathbf{false} : \{\Gamma : \mathsf{Cx}\} \to \mathsf{Tm}(\Gamma, \mathbf{Bool})$
- Equations $\mathbf{true}[\gamma] = \mathbf{true}$ and $\mathbf{false}[\gamma] = \mathbf{false}$ for every $\gamma : \mathrm{Sb}(\Delta, \Gamma)$.

Finally, we require that the following maps are bijections for all Γ and $A \in \mathsf{Ty}(\Gamma.\mathbf{Bool})$:

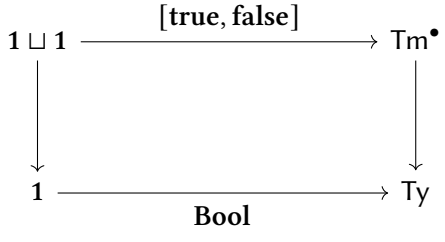
$$(-[\mathbf{id}.\mathbf{true}], -[\mathbf{id}.\mathbf{false}]) : \qquad \dagger$$
$$\mathsf{Tm}(\Gamma.\mathbf{Bool}, A) \cong \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{true}]) \times \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{false}])$$

We will refer to the final point in this list as Property † as it will bear the brunt of our scrutiny.

Inspecting the rules and equations for **Bool**, **true**, and **false**, we see that they all organize into natural transformations *e.g.*,



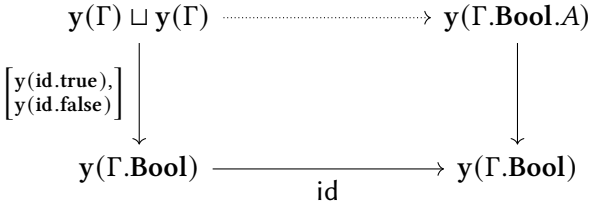
Here, the commutativity expresses the fact that **true** has the expected type. We can place **true** and **false** in the same diagram by using $\mathbf{1} \sqcup \mathbf{1}$ in $\mathbf{Pr}(\mathsf{Cx})$:



(6.5)

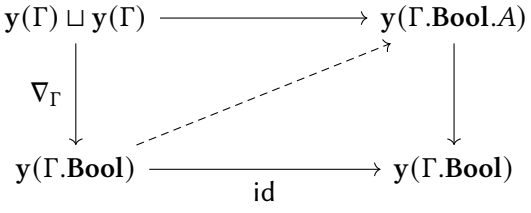
In light of Exercise 6.11, this square is never a pullback square. We can 'measure' the failure of Diagram 6.5 to be a pullback by studying the induced map $i : \mathbf{1} \sqcup \mathbf{1} \longrightarrow \mathbf{1} \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet = \mathbf{Bool}^* \mathsf{Tm}^\bullet$; the square is a pullback if and only if i is an isomorphism. Unfolding definitions, i is the map which includes **true**, **false** into $\mathsf{Tm}(\Gamma, \mathbf{Bool})$. Firstly, this makes it clear why i can never be an isomorphism—think of variable elements of **Bool**. On the other hand, our analysis of **Bool** in Section 2.5 indicates that i should be an isomorphism "from the perspective of other types". We shall now see that this idea is really the force of Property †.

We begin by restructuring Property † slightly to make the connection to i more visible. First, we note that $\mathsf{Tm}(\Gamma.\mathbf{Bool}, A)$ is equivalent to the set of *sections* of the weakening map $\Gamma.\mathbf{Bool}.A \longrightarrow \Gamma.\mathbf{Bool}$. For $\mathsf{Tm}(\Gamma, A[\mathbf{true}])$ and $\mathsf{Tm}(\Gamma, A[\mathbf{false}])$, we can combine the above remark about sections with Exercise 6.4. In particular, a pair of elements from $\mathsf{Tm}(\Gamma, A[\mathbf{true}])$ and $\mathsf{Tm}(\Gamma, A[\mathbf{false}])$ corresponds to a choice of dotted top arrow of the following diagram:

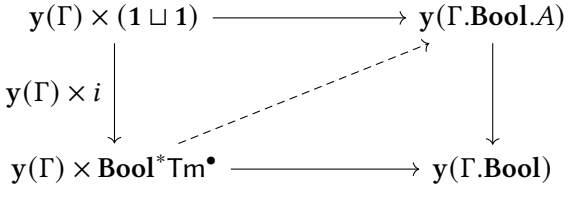


Note that we must express this diagram in $\mathbf{Pr}(\mathsf{Cx})$ via the Yoneda embedding because there is no guarantee that Cx will have coproducts. Let us denote $[\mathbf{y}(\mathbf{id}.\mathbf{true}), \mathbf{y}(\mathbf{id}.\mathbf{false})]$ by ∇_Γ in what follows.

In light of these observations, Property † is equivalent to requiring that for all Γ and $A \in \mathsf{Ty}(\Gamma)$, whenever there is a commuting square of the following shape then there is a unique dashed map making the following commute:

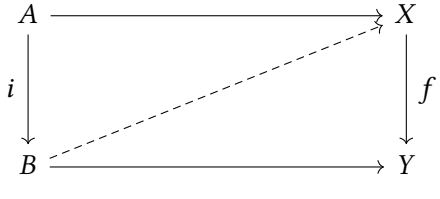


We can further simplify matters by scrutinizing ∇_Γ and "factoring out" the Γ. Note that $\mathbf{y}(\Gamma) \sqcup \mathbf{y}(\Gamma) \cong \mathbf{y}(\Gamma) \times (\mathbf{1} \sqcup \mathbf{1})$ and $\mathbf{y}(\Gamma.\mathbf{Bool}) \cong \mathbf{y}(\Gamma) \times \mathbf{Bool}^*\mathsf{Tm}^\bullet$. Pre- and post-composing ∇_Γ with these isomorphisms yields $\mathbf{y}(\Gamma) \times \nabla_{\mathbf{1}}$. Fortunately, we have already encountered $\nabla_{\mathbf{1}}$: it is precisely the aforementioned $i : \mathbf{1} \sqcup \mathbf{1} \longrightarrow \mathbf{Bool}^*\mathsf{Tm}^\bullet$ which measures the failure of Diagram 6.5 to be a pullback. We therefore rewrite the above diagram to the following equivalent:



Having made the connection between Property † and i more transparent, our next goal is to link both to the categorical notion of *orthogonality*:

Definition 6.3.3. If $i : A \longrightarrow B$ and $f : X \longrightarrow Y$ are morphisms in $\mathcal{C}$, we say that $i \perp f$ (i *is orthogonal to* f) if every commuting square of the following shape has a unique diagonal map making it commute:

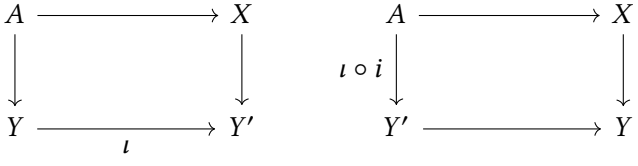


We also say that i is left orthogonal to f and f is right orthogonal to i.

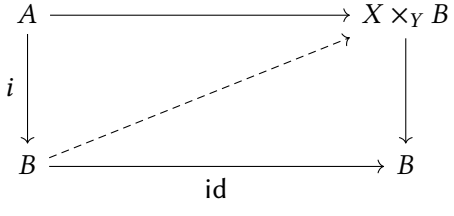
Remark 6.3.4. One should interpret $i \perp f$ as f "believing" that i is an isomorphism (or, dually, that i believes f is an isomorphism). This viewpoint is foundational in homotopical algebra, where one systematically studies orthogonality and weaker notions thereof. It is also critical in type theory, where elimination principles for inductive types can almost always be phrased as some suitable notion of orthogonality. See Borceux [Bor94, §5.4] for a quick introduction to orthogonality. ◇

Property † as we have presented it above then *almost* states that $\mathbf{y}(\Gamma) \times i$ is orthogonal to $\mathbf{p}$. However, there is a slight mismatch: we have unique lifts only when the bottom map is the isomorphism $\mathbf{y}(\Gamma) \times \mathbf{Bool}^* \mathsf{Tm}^\bullet \cong \mathbf{y}(\Gamma.\mathbf{Bool})$, while orthogonality requires arbitrary maps. The following results show that this is a distinction without a difference.

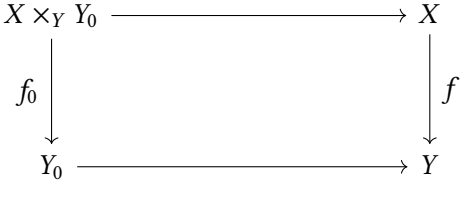
Exercise 6.12. Show that if $i : A \longrightarrow Y$ and $f : X \longrightarrow Y'$ are morphisms in $\mathcal{C}$ and $\iota : Y \cong Y'$, then each of the following diagrams has a unique diagonal map precisely when the other does:



Exercise 6.13. Show that if $i : A \longrightarrow B$ and $f : X \longrightarrow Y$ are morphisms in $\mathcal{C}$ then $i \perp f$ if and only if each $g_0 : B \longrightarrow Y$ and $g_1 : A \longrightarrow B \times_Y X$, the following diagram has a unique diagonal map:



Exercise 6.14. Show that if $i : A \longrightarrow B$ and $f : X \longrightarrow Y$ are morphisms in $\mathcal{C}$ where $i \perp f$ then $i \perp f_0$ for any f_0 fitting into a pullback diagram of the following shape:



Note that weakening maps $\Gamma.\mathbf{Bool}.A \longrightarrow \Gamma.\mathbf{Bool}$ are precisely the pullbacks π along a map $\Gamma.\mathbf{Bool} \longrightarrow \mathsf{Ty}$. Combining this with the above exercises, we conclude the following:

Lemma 6.3.5. *Property † is equivalent to requiring* $\mathbf{y}(\Gamma) \times i \perp \pi$ *for every* Γ.

Our next goal is to try and further simplify matters by, in particular, bundling away the "$\mathbf{y}(\Gamma) \times -$ for all Γ" portion of the above definition. To this end, note that while we have phrased orthogonality in terms of a diagram having a unique lift, a moment's thought reveals that $\mathbf{y}(\Gamma) \times i \perp \pi$ is equivalent to requiring that the following map be a bijection of sets:

$$\begin{array}{c} \hom(\mathbf{y}(\Gamma) \times \mathbf{Bool}^*\mathsf{Tm}^\bullet, \mathsf{Tm}^\bullet) \\ \downarrow \\ \hom(\mathbf{y}(\Gamma) \times \mathbf{Bool}^*\mathsf{Tm}^\bullet, \mathsf{Ty}) \times_{\hom(\mathbf{y}(\Gamma)\times(\mathbf{1}\sqcup\mathbf{1}),\mathsf{Ty})} \hom(\mathbf{y}(\Gamma) \times (\mathbf{1} \sqcup \mathbf{1}), \mathsf{Tm}^\bullet) \end{array}$$

Recall that presheaves form a cartesian closed category and that, in particular, the exponential of presheaves $X^Y : \mathbf{Pr}(\mathsf{Cx})$ is given by the functor sending $\Gamma \mapsto \hom(\mathbf{y}(\Gamma) \times X, Y)$. Consequently, requiring that the above map of sets be a bijection for all Γ is equivalent to requiring that the following canonical map be an isomorphism of presheaves:

$$(\mathsf{Tm}^\bullet)^{\mathbf{Bool}^*\mathsf{Tm}^\bullet} \longrightarrow \mathsf{Ty}^{\mathbf{Bool}^*\mathsf{Tm}^\bullet} \times_{\mathsf{Ty}^{\mathbf{1}\sqcup\mathbf{1}}} (\mathsf{Tm}^\bullet)^{\mathbf{1}\sqcup\mathbf{1}}$$

Here we see that the Γ has been bundled away into the fact that this map is a natural transformation of presheaves rather than a map of sets. We can codify this situation more generally:

Definition 6.3.6. If C is a category with finite limits and exponentials, we say that $i : A \longrightarrow B$ is *internally orthogonal* to $f : X \longrightarrow Y$, written $i \pitchfork f$, if the canonical map $X^B \longrightarrow Y^B \times_{Y^A} X^A$ is an isomorphism.

The above calculation then shows that we can detect internal orthogonality as the (external) orthogonality of a collection of maps. This fact holds in more generality:

Exercise 6.15. Fix a category C along with morphisms of presheaves $i : A \longrightarrow B$ and $f : X \longrightarrow Y$. Show that the following are equivalent:

- $\mathbf{y}(c) \times i \perp f$ for all $c : C$.
- $X \times i \perp f$ for all $X : \mathbf{Pr}(C)$.

- $i \pitchfork f$.

Putting this together, we conclude the following:

Lemma 6.3.7 (Categorical reformulation of **Bool**)**.** *A boolean structure on $\mathcal{M}$ is equivalent to a choice of Diagram 6.5 such that the gap map i satisfies $i \pitchfork \pi$.*

For future use and for further practice, we record the following facts as exercises:

Exercise 6.16. Given $F : I \longrightarrow C^{\to}$ such that $F(i) \perp g$ for all $i : I$, show that $\varinjlim_i F(i) \perp g$. Prove the same for $\pitchfork$.

Exercise 6.17. Fix $L : \mathcal{C} \longrightarrow \mathcal{D}$ such that $L \dashv R$, if $i : A \longrightarrow B : \mathcal{C}$ and $f : X \longrightarrow Y : \mathcal{D}$ then $L(i) \perp f$ if and only if $i \perp R(f)$. This same fact does not hold for $\pitchfork$, can you see why?

Before we introduce a refinement of Slogan 2.5.5, we replay this story for coproduct types to see an example with non-trivial formation data.

6.3.2 *Coproducts*

The step generalizing from booleans to coproduct types is not large. We go through the same process again for some extra practice with orthogonality. As before, we begin by collecting together the operations and equations necessary for a model to support coproducts:

Structure 6.3.8. A coproduct structure on $\mathcal{M}$ consists of the following operations, equations, and properties:

- An operator $+ : \{\Gamma : \mathsf{Cx}\} \to \mathsf{Ty}(\Gamma) \to \mathsf{Ty}(\Gamma) \to \mathsf{Ty}(\Gamma)$
- An equation $(A + B)[\gamma] = A[\gamma] + B[\gamma]$ for every $\gamma : \mathsf{Sb}(\Delta, \Gamma)$ and $A, B : \mathsf{Ty}(\Gamma)$.
- A pair of operators

$$\begin{aligned} \mathbf{inl} &: \{\Gamma : \mathsf{Cx}\}(A, B : \mathsf{Ty}(\Gamma)) \to \mathsf{Tm}(\Gamma, A) \to \mathsf{Tm}(\Gamma, A + B) \\ \mathbf{inr} &: \{\Gamma : \mathsf{Cx}\}\{A, B : \mathsf{Ty}(\Gamma)\} \to \mathsf{Tm}(\Gamma, B) \to \mathsf{Tm}(\Gamma, A + B) \end{aligned}$$

- Equations $\mathbf{inl}(a)[\gamma] = \mathbf{inl}(a[\gamma])$ and $\mathbf{inr}(b)[\gamma] = \mathbf{inr}(b[\gamma])$ for every $\gamma : \mathsf{Sb}(\Delta, \Gamma)$, $A, B : \mathsf{Ty}(\Gamma)$, $a : \mathsf{Tm}(\Gamma, A)$ and $b : \mathsf{Tm}(\Gamma, B)$.

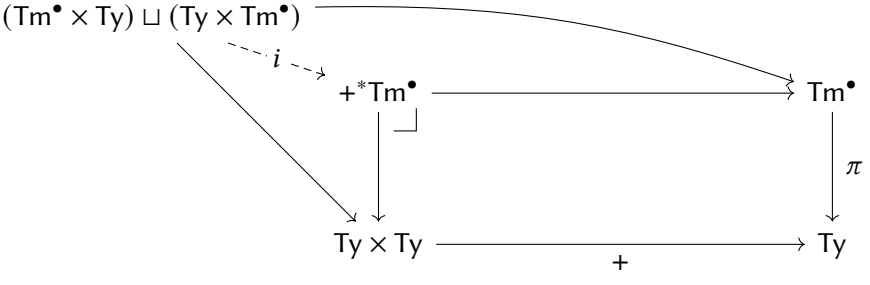
- Proofs that the following maps are bijections for all Γ and $A, B \in \mathsf{Ty}(\Gamma)$ and $C \in \mathsf{Ty}(\Gamma.A + B)$

$$
\begin{aligned}
&(-[\mathbf{p}.\mathbf{inl}(\mathbf{q})], -[\mathbf{p}.\mathbf{inr}(\mathbf{q})]) : \\
&\quad \mathsf{Tm}(\Gamma.A + B, C) \\
&\quad \cong \mathsf{Tm}(\Gamma.A, C[\mathbf{p}.\mathbf{inl}(\mathbf{q})]) \times \mathsf{Tm}(\Gamma.B, C[\mathbf{p}.\mathbf{inr}(\mathbf{q})])
\end{aligned}
$$

We once more refer to this final property as Property † and, just as before, note that we can use coproducts in $\mathbf{Pr}(\mathsf{Cx})$ to capture the first four items with a single commuting diagram in $\mathbf{Pr}(\mathsf{Cx})$:

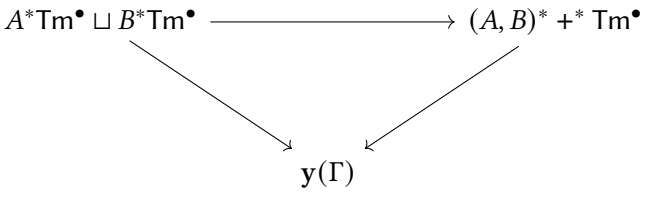
$$
\begin{array}{ccc}
(\mathsf{Tm}^\bullet \times \mathsf{Ty}) \sqcup (\mathsf{Ty} \times \mathsf{Tm}^\bullet) & \xrightarrow{[\mathbf{inl}, \mathbf{inr}]} & \mathsf{Tm}^\bullet \\
\downarrow{\scriptstyle [\pi \times \mathsf{id}, \mathsf{id} \times \pi]} & & \downarrow{\scriptstyle \pi} \\
\mathsf{Ty} \times \mathsf{Ty} & \xrightarrow[+]{} & \mathsf{Ty}
\end{array}
\tag{6.6}
$$

We now turn our attention to Property † and connecting it with orthogonality. As before, Diagram 6.6 induces a map $i : (\mathsf{Tm}^\bullet \times \mathsf{Ty}) \sqcup (\mathsf{Ty} \times \mathsf{Tm}^\bullet) \longrightarrow +^*\mathsf{Tm}^\bullet$:



In fact, more is true. Since the above diagram commutes, we know that i induces a morphism in $\mathbf{Pr}(\mathsf{Cx})_{/\mathsf{Ty}\times\mathsf{Ty}}$ between $[\pi \times \mathsf{id}, \mathsf{id} \times \pi]$ and π_1. Something similar was also true for **Bool** but there it was trivial: i induced a morphism in the slice category of $\mathbf{1}$ which is simply equivalent to $\mathbf{Pr}(\mathsf{Cx})$. This is a reflection of the fact that the type of coproducts—unlike that of booleans—has non-trivial formation data. Consequently, the introduction operation sending *e.g.*, an element of A to an element of $A + B$ is parameterized not just by the context but also by the two types A and B. This additional parameterization gives rise to a natural transformation in $\mathbf{Pr}(\int_{\mathsf{Cx}} \mathsf{Ty} \times \mathsf{Ty})$ or, equivalently, $\mathbf{Pr}(\mathsf{Cx})_{/\mathsf{Ty}\times\mathsf{Ty}}$.

To get a better understanding of i, let us calculate a little with it. Fix a pair of types $A, B : \mathbf{y}(\Gamma) \longrightarrow \mathsf{Ty}$ and consider the pullback functor $(A,B)^* : \mathbf{Pr}(\mathsf{Cx})_{/\mathsf{Ty}\times\mathsf{Ty}} \longrightarrow \mathbf{Pr}(\mathsf{Cx})_{/\mathbf{y}(\Gamma)}$. Applying this to i, we obtain the following morphism in $\mathbf{Pr}(\mathsf{Cx})_{/\mathbf{y}(\Gamma)}$:



Exercise 6.18. Carefully check that $(A,B)^*(i)$ has the required form.

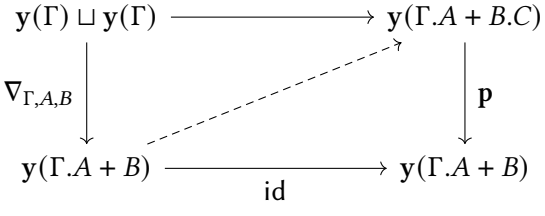
We can further simplify this by noting that $A^*\mathsf{Tm}^\bullet \cong \mathbf{y}(\Gamma.A)$ and $B^*\mathsf{Tm}^\bullet \cong \mathbf{y}(\Gamma.B)$. Moreover, more-or-less by definition of $+ : \mathsf{Ty}\times\mathsf{Ty} \longrightarrow \mathsf{Ty}$ there is an isomorphism $(A,B)^* +^* \mathsf{Tm}^\bullet \cong \mathbf{y}(\Gamma.A + B)$. All told then, $(A,B)^*(i)$ gives, up to isomorphism, the following map over $\mathbf{y}(\Gamma)$:

$$\nabla_{\Gamma,A,B} : \mathbf{y}(\Gamma.A) \sqcup \mathbf{y}(\Gamma.B) \longrightarrow \mathbf{y}(\Gamma.A + B)$$

Following our intuitions from the boolean case, we arrive at the following lemma:

Lemma 6.3.9. *Property † is equivalent to requiring* $\nabla_{\Gamma,A,B} \perp \pi$ *for all* $\Gamma : \mathsf{Cx}$ *and* $A, B : \mathsf{Ty}(\Gamma)$.

Proof. Recall that $\nabla_{\Gamma,A,B} \perp \pi$ holds if and only if for each $C : \mathbf{y}(\Gamma.A + B) \longrightarrow \mathsf{Ty}$ (equivalently, a type $C : \mathsf{Ty}(\Gamma.A + B)$), every diagram of the following shape has a unique diagonal map:



Unfolding and using the full and faithfulness of $\mathbf{y}$, this is equivalent to Property †. □

Our next and final step is to attempt to "factor out" the Γ, A, and B from $\nabla_{\Gamma,A,B}$ to characterize Property † using internal orthogonality. This process is more-or-less identical to what we have just seen with **Bool**, only slightly complicated by the presence of formation data for +.

We have already seen that (up to isomorphism) $U_{\mathbf{y}(\Gamma)}((A,B)^*(i)) = \nabla_{\Gamma,A,B}$ whenever $A, B : \mathbf{y}(\Gamma) \longrightarrow \mathsf{Ty}$. Recalling that products in a slice category are computed as pullbacks in the underlying category, this is equivalent to the statement $U_{\mathsf{Ty}\times\mathsf{Ty}}((A,B)\times i) \cong \nabla_{\Gamma,A,B}$ where we now regard $(A,B) : \mathbf{y}(\Gamma) \longrightarrow \mathsf{Ty}\times\mathsf{Ty}$ as an object of $\mathbf{Pr}(\mathsf{Cx})_{/\mathsf{Ty}\times\mathsf{Ty}}$.

Lemma 6.3.10. *Property † holds if and only if* $U((A,B)\times i) \perp \pi$ *for every* $A, B : \mathbf{y}(\Gamma) \longrightarrow \mathsf{Ty}$.

The forgetful functor $U_C : \mathcal{C}_{/C} \longrightarrow \mathcal{C}$ has a right adjoint whenever $\mathcal{C}$ has products: $X \mapsto C \times X$. Using Exercise 6.17, we may rephrase Property † again:

Lemma 6.3.11. *Property † holds if and only if* $(A,B)\times i \perp (\mathsf{Ty}\times\mathsf{Ty})\times\pi$ *holds in* $\mathbf{Pr}(\mathsf{Cx})_{/\mathsf{Ty}\times\mathsf{Ty}}$ *for every* $A, B : \mathbf{y}(\Gamma) \longrightarrow \mathsf{Ty}$.

Modulo the equivalence $\mathbf{Pr}(\int_{\mathsf{Cx}} \mathsf{Ty}\times\mathsf{Ty}) \simeq \mathbf{Pr}(\mathsf{Cx})_{/\mathsf{Ty}\times\mathsf{Ty}}$, Exercise 6.15 shows that the above condition is precisely $i \pitchfork (\mathsf{Ty}\times\mathsf{Ty})\times\pi$.

Exercise 6.19. Fill in the details of the above argument by showing that representable functors in $\mathbf{y}(c,x) : \mathbf{Pr}(\int_{\mathcal{C}} X)$ correspond precisely to maps $\mathbf{y}(c) \longrightarrow X : \mathbf{Pr}(\mathcal{C})_{/X}$ under the equivalence $\mathbf{Pr}(\int_{\mathcal{C}} X) \simeq \mathbf{Pr}(\mathcal{C})_{/X}$.

All told then, we arrive at the following:

Lemma 6.3.12 (Categorical reformulation of +)**.** *A coproduct structure on* $\mathcal{M}$ *is equivalent to a choice of commuting square (Diagram 6.6) such that the gap map* i *satisfies* $i \pitchfork (\mathsf{Ty}\times\mathsf{Ty})\times\pi$ *in* $\mathbf{Pr}(\mathsf{Cx})_{/\mathsf{Ty}\times\mathsf{Ty}}$.

6.3.3 *The empty type*

We may consolidate this into an extension of Slogan 6.2.10 which accounts for non-recursive inductive types:

Slogan 6.3.13. *A non-recursive inductive type* Υ *is specified by a commuting square:*

$$\begin{array}{ccc} I_\Upsilon & \xrightarrow{\mathsf{intro}_\Upsilon} & \mathsf{Tm}^\bullet \\ \downarrow & & \downarrow \\ F_\Upsilon & \xrightarrow[\mathsf{form}_\Upsilon]{} & \mathsf{Ty} \end{array}$$

Where form_Υ *is the formation map and* intro_Υ *is the introduction operation. Moreover, if* $i : I \longrightarrow \mathsf{form}_\Upsilon^* \mathsf{Tm}^\bullet$ *in* $\mathbf{Pr}(\mathsf{Cx})_{/F}$ *is the gap map, we require that* $i \pitchfork F \times \pi$ *in* $\mathbf{Pr}(\mathsf{Cx})_{/F}$.

We can apply this slogan to quickly reformulate the specification of **Void**:

Structure 6.3.14. An empty type structure on a model $\mathcal{M}$ consists of the following operations, equations, and properties:

- An operator $\mathbf{Void} : \{\Gamma : \mathsf{Cx}\} \to \mathsf{Ty}(\Gamma)$
- An equation $\mathbf{Void}[\gamma] = \mathbf{Void}$ for every $\gamma : \mathsf{Sb}(\Delta, \Gamma)$.

Finally, we require that the following unique map is a bijection for all Γ and $A \in \mathsf{Ty}(\Gamma.\mathbf{Void})$:

$$\mathsf{Tm}(\Gamma.\mathbf{Void}, A) \to \{\star\}$$

Lemma 6.3.15 (Categorical reformulation of **Void**)**.** *An empty type structure on a model is equivalent to the following:*

- *A commuting square of the following form:*

$$\begin{array}{ccc} \mathbf{0} & \longrightarrow & \mathsf{Tm}^\bullet \\ \downarrow & & \downarrow \\ \mathbf{1} & \xrightarrow[\mathbf{Void}]{} & \mathsf{Ty} \end{array}$$

- *The gap map* $i : \mathbf{0} \longrightarrow \mathbf{Void}^* \mathsf{Tm}^\bullet$ *satisfies* $i \pitchfork \pi$.

6.3.4 *Natural numbers*

Just as in Section 2.5, the type of natural numbers proves to be more difficult than **Void**, **Bool**, or +. As before, the complexity is a result of the recursive nature of **Nat** which means we cannot consider just an orthogonality condition to describe **Nat**; we must also have some categorical account of (initial) algebras as introduced in Section 2.5.4.

We begin by recalling the specification of **Nat** in $\mathcal{M}$:

Structure 6.3.16. A natural number structure on a model $\mathcal{M}$ consists of the following:

- An operation $\mathbf{Nat} : \{\Gamma : \mathsf{Cx}\} \to \mathsf{Ty}(\Gamma)$.
- Equations $\mathbf{Nat}[\gamma] = \mathbf{Nat}$ for all $\gamma : \mathsf{Sb}(\Delta, \Gamma)$.

- An operation $\mathbf{zero} : \{\Gamma : \mathsf{Cx}\} \to \mathsf{Tm}(\Gamma, \mathbf{Nat})$.
- Equations $\mathbf{zero}[\gamma] = \mathbf{zero}$ for all $\gamma : \mathsf{Sb}(\Delta, \Gamma)$.
- An operation $\mathbf{suc} : \{\Gamma : \mathsf{Cx}\} \to \mathsf{Tm}(\Gamma, \mathbf{Nat}) \to \mathsf{Tm}(\Gamma, \mathbf{Nat})$.
- Equations $(\mathbf{suc}(n))[\gamma] = \mathbf{suc}(n[\gamma])$ for all $\gamma : \mathsf{Sb}(\Delta, \Gamma)$ and $n : \mathsf{Tm}(\Gamma, \mathbf{Nat})$.
- Given a type $A : \mathsf{Ty}(\Gamma.\mathbf{Nat})$ along with terms $a_z : \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{zero}])$ and $a_s : \mathsf{Tm}(\Gamma.\mathbf{Nat}.A, A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])])$, there is a unique term $a : \mathsf{Tm}(\Gamma.\mathbf{Nat}, A)$ satisfying the following two equations:

$$a[\mathbf{id}.\mathbf{zero}] = a_z$$
$$a[\mathbf{p}.\mathbf{suc}(\mathbf{q})] = a_s[\mathbf{id}.a]$$

As before, we refer to the final point as Property †.

The first six points can be compactly expressed using natural transformations in $\mathbf{Pr}(\mathsf{Cx})$ as we have seen already. They are precisely equivalent to the following two pieces of data:

- A morphism $\mathbf{Nat} : \mathbf{1} \longrightarrow \mathsf{Ty}$.
- A morphism $\alpha : \mathbf{1} \sqcup \mathbf{Nat}^*\mathsf{Tm}^\bullet \longrightarrow \mathbf{Nat}^*\mathsf{Tm}^\bullet$.

Initial algebras, categorically In fact, the morphism α can be said to shape $\mathbf{Nat}^*\mathsf{Tm}^\bullet$ into an algebra for a certain functor. To state this more precisely, we recall the definition of an algebra:

Definition 6.3.17. If $F : C \longrightarrow C$ is a functor, an F-algebra is an object C along with a morphism $a : F(C) \longrightarrow C$.

Definition 6.3.18. A homomorphism between F-algebras $a : F(C) \longrightarrow C$ and $b : F(D) \longrightarrow D$ is a morphism $f : C \longrightarrow D$ such that $f \circ a = b \circ F(f)$:

$$\begin{array}{ccc} F(C) & \xrightarrow{F(f)} & F(D) \\ \downarrow & & \downarrow \\ C & \xrightarrow[f]{} & D \end{array}$$

We write $\mathbf{Alg}(F)$ for the category of F-algebras.

With a category to hand, it is easy to define the initial F-algebra for any functor F: it is the initial object of $\mathbf{Alg}(F)$ provided such an object exists. Our goal shall be to use this definition to replay the intuition that **Nat** is an initial algebra of sorts. To this end, we shall eventually require the analog of a *dependent* algebra from Section 2.5.4 so we record a succinct definition of here:

Definition 6.3.19. The category of *dependent* F-algebras over an F-algebra $a : F(C) \longrightarrow C$ is the slice category $\mathbf{Alg}(F)_{/(C,a)}$.

Lemma 6.3.20. *Aside from Property †, a model supports a type of natural numbers precisely when there is a natural transformation* $\mathbf{Nat} : \mathbf{1} \longrightarrow \mathsf{Ty}$ *along with a choice of* α *of* $(- \sqcup \mathbf{1})$*-algebra structure on* $\mathbf{Nat}^*\mathsf{Tm}^\bullet$.

What remains, as ever, is to account for Property †. In this case, we do not require an orthogonality condition. We need to record the fact that $\mathbf{Nat}^*\mathsf{Tm}^\bullet$ is, in some sense, the initial $(- \sqcup \mathbf{1})$-algebra among types.

To begin with, we note the following:

Lemma 6.3.21. *If* $\Gamma : \mathsf{Cx}$ *then* $\mathbf{y}(\Gamma.\mathbf{Nat}) \cong \mathbf{y}(\Gamma) \times \mathbf{Nat}^*\mathsf{Tm}^\bullet$ *supports the structure of a* $(- \sqcup \mathbf{1})$*-algebra given up to isomorphism by* $\mathbf{y}(\Gamma) \times \alpha$.

Lemma 6.3.22. *If* $A : \mathsf{Ty}(\Gamma.\mathbf{Nat})$ *then* a_z*, and* a_s *as given in Property † are equivalent to structuring* $\mathbf{y}(\Gamma.\mathbf{Nat}.A)$ *as a dependent algebra over* $\mathbf{y}(\Gamma.\mathbf{Nat})$ *via a map:*

$$\chi_{a_z,a_s} : \mathbf{y}(\Gamma.\mathbf{Nat}.A) \sqcup \mathbf{1} \longrightarrow \mathbf{y}(\Gamma.\mathbf{Nat}.A)$$

Lemma 6.3.23. *If* $A : \mathsf{Ty}(\Gamma.\mathbf{Nat})$*,* a_z*, and* a_s *are as given in Property †, the unique existence of a term a corresponds to existence of a unique algebra homomorphism* $\mathbf{1} \longrightarrow \Gamma.\mathbf{Nat}.A$ *in* $\mathbf{Alg}(- \sqcup \mathbf{1})_{/\mathbf{y}(\Gamma.\mathbf{Nat})}$.

This suggests that $\mathbf{y}(\Gamma.\mathbf{Nat})$ ought to be the initial object in $\mathbf{Alg}(- \sqcup \mathbf{1})_{/\mathbf{y}(\Gamma.\mathbf{Nat})}$, but this is not quite correct. We only have initiality with respect to those dependent algebras of the form $\Gamma.\mathbf{Nat}.A \longrightarrow \Gamma.\mathbf{Nat}$ for some A.

Definition 6.3.24. Given an F-algebra (Y, α), a *representable* dependent F-algebra $X \longrightarrow Y$ is a dependent algebra over Y such that $X \longrightarrow Y$ is a pullback of π.

Lemma 6.3.25. *A natural number structure on a model of type theory is equivalent to a natural transformation* $\mathbf{Nat} : \mathbf{1} \longrightarrow \mathsf{Ty}$ *along with a* $(- \sqcup \mathbf{1})$*-algebra structure* α *on* $\mathbf{Nat}^*\pi$ *such that for all* $\Gamma : \mathsf{Cx}$*, if one restricts the category of dependent algebras over* $(\mathbf{y}(\Gamma) \times \mathbf{Nat}^*\pi, \mathbf{y}(\Gamma) \times \alpha)$ *to the full subcategory of representable dependent algebras,* $\mathbf{y}(\Gamma) \times \alpha$ *is initial.*

Can we simplify this further? Feels a little half-baked. Can we return to this with the internal language to give a slick definition that way?

6.3.5 *Weak orthogonality and types without η laws*

Recall that our official definition of ETT in Chapter 2 did not include η principles for inductive types. In particular, we chose to omit rules such as the following from our specification of *e.g.*, **Bool**:

$$\frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma.\mathbf{Bool} \vdash a : A}{\Gamma.\mathbf{Bool} \vdash a = \mathbf{if}\,(a[\mathbf{p}.\mathbf{true}], a[\mathbf{p}.\mathbf{false}], \mathbf{q}) : A}$$

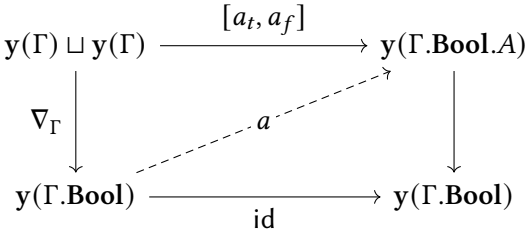
We justified this choice with two observations:

- These rules, much like equality reflection, make it vastly harder or even impossible to construct a normalization algorithm for type theory.
- All of these η principles are derivable from the corresponding β rules in the presence of equality reflection.

Accordingly, we reasoned that it was more efficient to have a single rule which compromised decidability of type-checking (equality reflection) to ensure that the transition from ETT to ITT was concentrated within a single connective (**Eq**).

In this subsection we pay attention to specifying mapping-out types *without* assuming an η law. If $\mathcal{M}$ supports **Eq**, these new descriptions are equivalent to those we have already given. However, if we wished to adapt this discussion from ETT to ITT, it is once again beneficial to specify mapping-out types without a unicity principle: the difference in models once more comes down to whether we include **Eq** or **Id** in the model. As a bonus, by investing some effort in describing mapping-out types without an η law, we are able to give a categorical description of when a model supports **Id** with no additional effort.

However, the inclusion of the η principles in our cwf reformulation of a model has actually allowed us to *simplify* various structures. In particular, the η rule ensures that the terms witnessing the elimination rules of various inductive connectives are actually unique. Accordingly, we were able to recast these elimination principles as various orthogonality properties: we showed that the elimination rule for *e.g.*, booleans could be recast as requiring a dotted map fitting into a commuting square:



The commutativity of this diagram corresponds to the β equalities of the elimination form: it states that when a is specialized to **true** or **false**, it collapses appropriately to a_t and a_f. The unicity of a accounts for the η law. If we remove the η law from booleans, therefore, we can no longer expect a to exist uniquely.

6.3.5.1 Booleans without a unicity principle

Let us recall the weakened notion of Property † used in Section 2.5.5. $\mathcal{M}$ supports booleans without the η law when in addition to the operations **Bool**, **true**, and **false**, it enjoys the following:

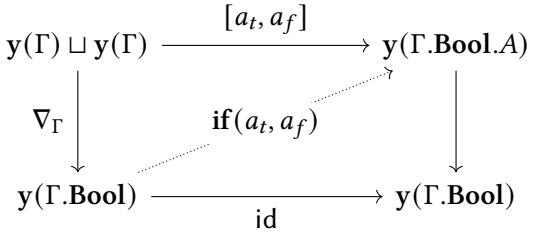
- An operation

$$\begin{aligned}\mathbf{if} : \{\Gamma : \mathsf{Cx}\}\{A : \mathsf{Ty}(\Gamma.\mathbf{Bool})\}\\ \to \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{true}]) \times \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{false}]) \to \mathsf{Tm}(\Gamma.\mathbf{Bool}, A)\end{aligned}$$

- Equations $\mathbf{if}(a_t, a_f)[\mathbf{id}.\mathbf{true}] = a_t$ and $\mathbf{if}(a_t, a_f)[\mathbf{id}.\mathbf{false}] = a_t$
- Equations $\mathbf{if}(a_t, a_f)[\gamma.\mathbf{Bool}] = \mathbf{if}(a_t[\gamma], a_f[\gamma])$ whenever $\gamma : \mathsf{Sb}_{\mathcal{M}}(\Delta, \Gamma)$.

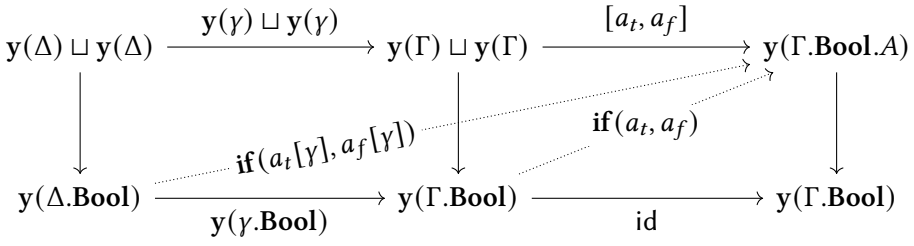
These properties combined are weaker than Property †, which essentially stated that **if** was *unique* among operations satisfying the second point (which, in particular, automatically causes it to satisfy the third point). Our goal is to discuss how this weaker set of properties can be recast categorically. Let us begin by fitting **if** into a lifting diagram.

Fixing $\Gamma : \mathsf{Cx}$, $A : \mathsf{Ty}(\Gamma)$, $a_t : \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{true}])$, and $a_f : \mathsf{Tm}(\Gamma, A[\mathbf{id}.\mathbf{false}])$, we see that the existence of **if** and the first pair of equations governing it can be summarized by the following commuting diagram:



However, we are no longer requiring that this diagonal lift exists *uniquely*, merely that some particular chosen lift exists. To integrate the third equation, suppose we are

given a substitution $\gamma : \mathrm{Sb}(\Delta, \Gamma)$. We require that the following diagram commute:

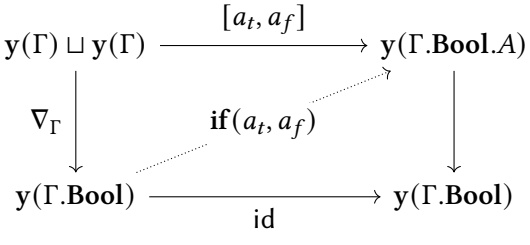


(6.7)

In particular, the third equation ensures that more than merely requiring that there are *some* collections of lifts to various commuting squares, the choice of lifts is suitably coherent: the chosen solution to lifting problem for a_t and a_f when restricted along $\mathbf{y}(\gamma.\mathbf{Bool})$ must match the solution to the lifting problem for $a_t[\gamma]$ and $a_f[\gamma]$.

We summarize this discussion with the following:

Lemma 6.3.26. *$\mathcal{M}$ supports* **if** *and β laws if there is a choice of lifting for all diagrams of the following shape:*

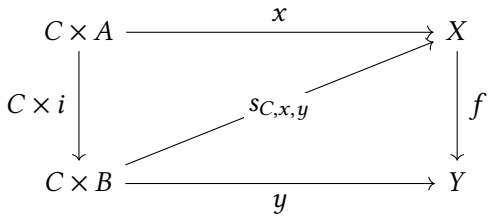


Furthermore, **if** *satisfies the final equation just when Diagram 6.7 commutes for all $\gamma : \Delta \longrightarrow \Gamma$.*

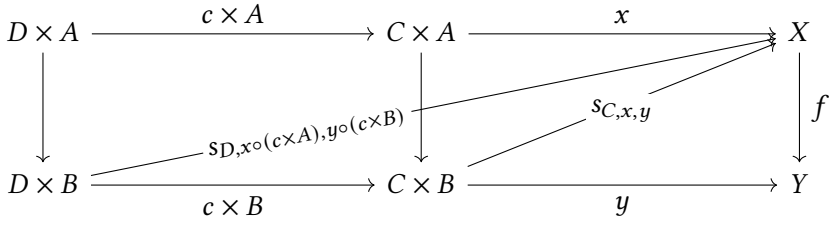
A digression: stable weak orthogonality structures This is a halfway point between "the lift is unique" and "there merely exists some lift". We have encountered the categorical incarnation of the former (orthogonality), while the latter is usually called *weak* orthogonality. To encode booleans, we need the midpoint of stable weak orthogonality. Note that unlike (weak) orthogonality, stable weak orthogonality is a *structure*: we must provide an explicit choice of maps which satisfy some properties. This is in contrast to (weak) orthogonality, where these maps are merely required to exist (uniquely or not).

Definition 6.3.27. An unstable weak orthogonality structure $s : (i : A \longrightarrow B) \pitchfork^{\mathsf{wk}} (f : X \longrightarrow Y)$ in a category C is an assignment of objects C and pairs of maps $x : C \times A \longrightarrow X$

and $y : C \times B \longrightarrow Y$ satisfying $f \circ x = y \circ (C \times i)$ to a map $s_{C,x,y}$ fitting into the following:



We say that s is stable—a *stable* weak orthogonality structure $s : i \pitchfork^{\mathrm{st}} f$—if it further satisfies the condition that for any $c : D \longrightarrow C$, the following diagram commutes:



We recall a characterization of stable orthogonality structures due to Awodey [Awo18]:

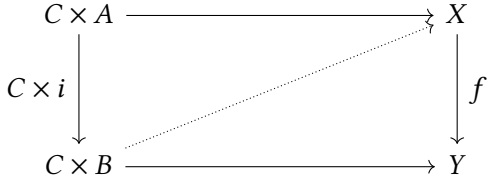
Lemma 6.3.28. *Supposing $\mathcal{C}$ has finite limits and exponentials, the stable weak orthogonality structure $i : A \longrightarrow B \pitchfork^{\mathrm{st}} f : X \longrightarrow Y$ is equivalent to a section to the canonical map $p : X^B \longrightarrow X^A \times_{Y^A} Y^B$.*

Proof. By the Yoneda lemma, to construct a map $s : X^A \times_{Y^A} Y^B \longrightarrow X^B$ such that $p \circ s = \mathrm{id}$, it suffices to construct a section $\mathbf{y}(X^A \times_{Y^A} Y^B) \longrightarrow \mathbf{y}(X^B)$ to $\mathbf{y}(p)$. Unfolding the data of a natural transformation in this case, for each $C : \mathcal{C}$, we must construct an assignment $\hom(C, X^A \times_{Y^A} Y^B) \longrightarrow \hom(C, X^B)$ which is natural in C. Let us use the universal properties of pullbacks and exponentials to simplify this:

$$\hom(C, X^B) \cong \hom(C \times B, X)$$

$$\hom(C, X^A \times_{Y^A} Y^B) \cong \hom(C \times A, X) \times_{\hom(C\times A, Y)} \hom(C \times B, Y)$$

In particular, an element of $\hom(C, X^A \times_{Y^A} Y^B)$ corresponds to a commuting square while elements of $\hom(C, X^B)$ correspond to commuting squares with a chosen lift:

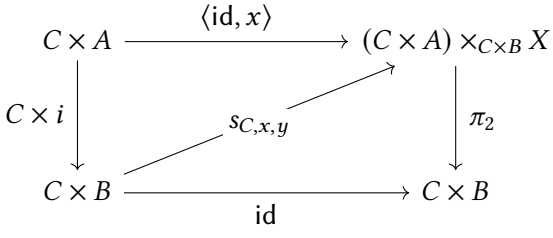


In other words, a section $\mathbf{y}(X^A \times_{Y^A} Y^B) \longrightarrow \mathbf{y}(X^B)$ corresponds precisely to an assignment of commuting squares to lifts and the naturality of this assignment is exactly the equation distinguishing a stable weak orthogonality structure from a weak orthogonality structure. □

Add something here about only needing to check at representables here. Tie in to section proof above and make things more related to the strict orthogonality case

By similar reasoning to Exercise 6.13, we obtain the following lemma:

Lemma 6.3.29. *An unstable weak orthogonality structure* $s : (i : A \longrightarrow B) \pitchfork^{\mathrm{wk}} (f : X \longrightarrow Y)$ *is equivalent to an assignment of objects* C *and maps* $y : C \times B \longrightarrow X$ *and* $x : C \times A \longrightarrow X \times_Y (C \times B)$ *satisfying* $\pi_2 \circ x = (C \times i)$ *to a map* $s_{C,x,y}$ *fitting into the following:*



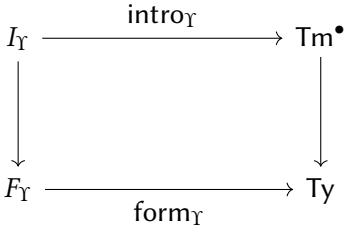
s is stable if for all $c : D \longrightarrow C$, $s_{C,x,y} \circ (c \times B) = ((i \times A) \times_{i \times B} X) \circ s_{D, x \circ (c \times A), y \circ (c \times B)}$.

Finally, just as done with orthogonality, we can combine Lemma 6.3.26 with the observations that (1) maps $\mathbf{y}(\Gamma.\mathbf{Bool}.A) \longrightarrow \mathbf{y}(\Gamma.\mathbf{Bool})$ are precisely the pullbacks of π along maps $\mathbf{y}(\Gamma.\mathbf{Bool}) \longrightarrow \mathsf{Ty}$ and (2) $\nabla_\Gamma \cong \mathbf{y}(\Gamma) \times i$ where i is the gap map $\mathbf{1} \sqcup \mathbf{1} \longrightarrow \mathbf{Bool}^* \mathsf{Tm}^\bullet$ to obtain the following:

Lemma 6.3.30. $\mathcal{M}$ *supports* **if** *and its attendant equations just when there is a stable weak orthogonality structure* $i \pitchfork^{\mathrm{st}} \pi$.

In total then, $\mathcal{M}$ supports booleans without an η law just when there is a commuting square Diagram 6.5 along with a stable weak orthogonality structure $i \pitchfork^{\mathrm{st}} \pi$. The revised version of Slogan 6.3.13 for types without an η law is given as follows:

Slogan 6.3.31. *The formation and introduction rules of a non-recursive inductive type* Υ *are specified by a commuting square:*



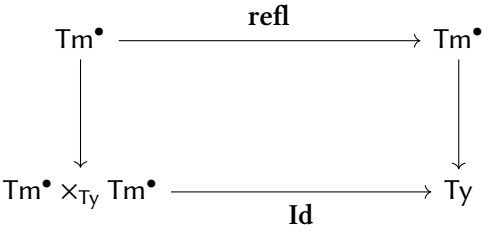
In the above, form *describes the formation operation and* intro *the introduction. The elimination rule* without *an* η *principle is given by the data of a stable weak orthogonality structure* $i \pitchfork^{\mathrm{st}} F_\Upsilon \times \pi$ *where* $i : I \longrightarrow \mathsf{form}_\Upsilon^* \mathsf{Tm}^\bullet$ *in* $\mathbf{Pr}(\mathsf{Cx})_{/F_\Upsilon}$ *is the gap map.*

Intensional identity types While strictly speaking our focus in this chapter is on the semantics of ETT, intensional identity types are too important to omit entirely. Moreover, they constitute an important addendum to Slogan 6.3.31 as the intensional identity type (more generally, *indexed* inductive types) does not fit the pattern described there. The problem lies in the elimination rule **j**, where the formation data of the type changes in one of the premises as it is fixed by **refl**. We already observed that this was necessary for **Id** when it was introduced, see Remark 4.2.2. The necessary alteration to Slogan 6.3.31 is relatively minor: instead of asking for a stable weak orthogonality structure in the slice over all the formation data, we work over only the portion of the formation data we wish to hold steady.

In the case of **Id**, the formation data is a type and two terms $\mathsf{Tm}^\bullet \times_{\mathsf{Ty}} \mathsf{Tm}^\bullet$, but the elimination rule changes those two terms in the premise for the base case—only the type is preserved. The categorical encoding of the intensional identity type is therefore given as follows:

Lemma 6.3.32. $\mathcal{M}$ *supports an intensional identity type just when it comes equipped with the following pieces of data:*

- *A commuting square of the following shape:*



- *A stable weak orthogonality structure* $(\mathsf{Tm}^\bullet \longrightarrow \mathbf{Id}^*\mathsf{Tm}^\bullet) \pitchfork^{\mathrm{st}} (\mathsf{Ty}^*\pi)$ *in* $\mathbf{Pr}(\mathsf{Cx})_{/\mathsf{Ty}}$.

We end our discussion here but note that Awodey [Awo18] specifically discusses the interpretation of **Id** in great detail. We refer the reader there for a detailed proof of the above.

6.4 *Cwf morphisms and* $\mathbf{U}_0, \mathbf{U}_1, \mathbf{U}_2, \ldots$

The final step in our process of converting Definition 3.4.2 to a more categorically acceptable form is to consider universes. We shall take this as an opportunity to also elaborate on the notion of a *homomorphism* of models (Definition 3.4.3) to give an slick—if indirect—characterization of universes as sub-models of type theory.

6.4.1 *Homomorphisms of models*

The definition of a homomorphism of models of type theory follows the same template as any algebraic structure: we have maps between all the (families of) sets which we require commute with all of the operations these sets are closed under.

Example 6.4.1. To see an example of this process in miniature, recall that a group $(G, 0, +, -)$ consists of (1) a set G and (2) three operations $0 : G$, $+ : G \times G \to G$ and $- : G \to G$ satisfying a handful of equations. We can 'read off' the definition of a morphism $f : (G, 0_G, +_G, -_G) \longrightarrow (H, 0_H, +_H, -_H)$ from this description. It consists of a function of sets $f_0 : G \longrightarrow H$ such that the following equations hold:

$$f_0(0_G) = 0_H \qquad f_0(a +_G b) = f_0(a) +_H f_0(b) \qquad f_0(-_G a) = -_H f_0(a)$$

We have already given a definition of morphisms of models in Definition 3.4.3 but since there are vastly more sets and operations for models of ETT than for groups, the definition is rather unwieldy. Our goal is to repackage this definition just as was done for that of models into a more concise and categorical framework.

Morphisms of models of base type theory

To this end, let us begin by considering type theory without any connectives and models consisting of only the operations described in Section 6.1 (*e.g.*, plain categories with families). Let us recall Definition 3.4.3 for this base type theory:

Definition 6.4.2. If $\mathcal{M}$ and $\mathcal{N}$ are models of base type theory, a homomorphism F from $\mathcal{M}$ to $\mathcal{N}$ consists of the following data:

- A function $F_{\mathsf{Cx}} : \mathsf{Cx}_{\mathcal{M}} \longrightarrow \mathsf{Cx}_{\mathcal{N}}$
- A family of functions $F_{\mathsf{Sb}(-,-)} : (\Delta, \Gamma : \mathsf{Cx}_{\mathcal{M}}) \to \mathsf{Sb}_{\mathcal{M}}(\Delta, \Gamma) \to \mathsf{Sb}_{\mathcal{N}}(F_{\mathsf{Cx}}(\Delta), F_{\mathsf{Cx}}(\Gamma))$
- A family of functions $F_{\mathsf{Ty}(-)} : (\Gamma : \mathsf{Cx}_{\mathcal{M}}) \to \mathsf{Ty}_{\mathcal{M}}(\Gamma) \to \mathsf{Ty}_{\mathcal{N}}(F_{\mathsf{Cx}}(\Gamma))$
- A family of functions

$$F_{\mathsf{Tm}(-,-)} : (\Gamma : \mathsf{Cx}_{\mathcal{M}})(A : \mathsf{Ty}_{\mathcal{M}}(\Gamma)) \to \mathsf{Tm}_{\mathcal{M}}(\Gamma, A) \to \mathsf{Tm}_{\mathcal{N}}(F_{\mathsf{Cx}}(\Gamma), F_{\mathsf{Ty}(\Gamma)}(A))$$

Moreover, we require that these functions commute with $\mathbf{1}$, $-.-$, $!$, $\mathbf{id}$, $\circ$, $\mathbf{p}$, $\mathbf{q}$, and substitution on types and terms. For instance, we have the following equations:

$$F_{\mathsf{Cx}}(\mathbf{1}_{\mathcal{M}}) = \mathbf{1}_{\mathcal{N}} \qquad F_{\mathsf{Sb}(\Gamma, \mathbf{1}_{\mathcal{M}})}(!_{\mathcal{M}}) = !_{\mathcal{N}}$$

We can reformulate homomorphisms using the description of models given in Definition 6.1.11. As a first step, we note the following:

Lemma 6.4.3. *If $F : \mathcal{M} \longrightarrow \mathcal{N}$ then the data of F_{Cx} and $F_{\mathsf{Sb}(-,-)}$ together with the requirements that these functions preserve $\circ$, $\mathbf{id}$, and $\mathbf{1}$ is equivalent to a functor $\mathsf{Cx}_{\mathcal{M}} \longrightarrow \mathsf{Cx}_{\mathcal{N}}$ which preserves the chosen terminal objects of these two categories.*

Lemma 6.4.4. *If $F : \mathcal{M} \longrightarrow \mathcal{N}$, the families of functions $F_{\mathsf{Ty}(-)}$ and $F_{\mathsf{Tm}(-,-)}$ together with the properties that they commute with substitution are equivalent to a choice of commuting square:*

$$\begin{array}{ccc} \mathsf{Tm}_{\mathcal{M}} & \xrightarrow{F_{\mathsf{Tm}}} & F^*\mathsf{Tm}_{\mathcal{N}} \\ \downarrow & & \downarrow \\ \mathsf{Ty}_{\mathcal{M}} & \xrightarrow[F_{\mathsf{Ty}}]{} & F^*\mathsf{Ty}_{\mathcal{N}} \end{array}$$

Here we denote the functor between categories of context induced by F as F.

Proof. Unfolding the definition of natural transformation and F^*, the conclusion follows immediately, *e.g.*, F_{Ty} sends an element $A \in \mathsf{Ty}_{\mathcal{M}}(\Gamma)$ to $F_{\mathsf{Ty}(\Gamma)}(A)$. □

These two requirements—a functor F between the categories of contexts preserving $\mathbf{1}$ and a commuting square between the presheaves of types and terms—record almost all of the requirements of Definition 6.4.2. The only outstanding requirement is the preservation of context extension. This is somewhat difficult to give a purely categorical phrasing of because it necessitates preserving *particular choices* of objects defined with universal properties.

Lemma 6.4.5. *A morphism of models $F : \mathcal{M} \longrightarrow \mathcal{N}$ consists of the following:*

- *A functor $F : \mathcal{M} \longrightarrow \mathcal{N}$ which preserves $\mathbf{1}$ on-the-nose.*
- *A commuting square of the following shape:*

$$\begin{array}{ccc} \mathsf{Tm}_{\mathcal{M}} & \xrightarrow{F_{\mathsf{Tm}}} & F^*\mathsf{Tm}_{\mathcal{N}} \\ \downarrow & & \downarrow \\ \mathsf{Ty}_{\mathcal{M}} & \xrightarrow[F_{\mathsf{Ty}}]{} & F^*\mathsf{Ty}_{\mathcal{N}} \end{array}$$

Such that for all $\Gamma : \mathsf{Cx}_{\mathcal{M}}$ and $A : \mathsf{Ty}_{\mathcal{M}}(\Gamma)$, we have $F(\Gamma.{}_{\mathcal{M}}A) = F(\Gamma).{}_{\mathcal{N}}F_{\mathsf{Ty}(\Gamma)}(A)$ along with $F(\mathbf{p}_{\mathcal{M}}) = \mathbf{p}_{\mathcal{N}}$ and $F_{\mathsf{Tm}(\Gamma.{}_{\mathcal{M}}A, A[\mathbf{p}_{\mathcal{M}}])}(\mathbf{q}_{\mathcal{M}}) = \mathbf{q}_{\mathcal{N}}$.

Remark 6.4.6. One could also imagine requiring that morphisms between cwfs preserve the empty context and context extension only up to canonical isomorphism. This viewpoint is systematically developed by *e.g.*, Clairambault and Dybjer [CD14]; Uemura [Uem21] constructs a further generalization of generalized algebraic theories which ensures that these morphisms are the default obtained by the logical framework. ◇

Say this defines a category

Dealing with connectives in morphisms of models

Thus far we have only discussed morphisms of type theory without any connectives. To extend our description of morphisms to full ETT, we must also specify how a morphism of models interacts with *e.g.*, $\mathbf{\Pi}$, Σ, and so on. Notably, since a connective extends the theory of type theory with new operations and equations but no new sorts, to extend our definition of morphism requires only that we add more conditions rather than imposing any new data.

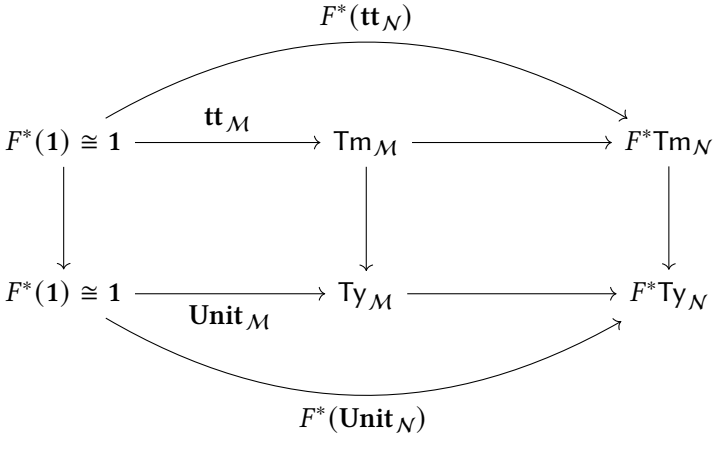
We once more recall a specialized version of Definition 3.4.3 dealing only with **Unit**:

Definition 6.4.7. A morphism $F : \mathcal{M} \longrightarrow \mathcal{N}$ of models of type theory with **Unit** consists of a morphism of models of base type theory of $F : \mathcal{M} \longrightarrow \mathcal{N}$ such that F satisfies the following equations:

$$F_{\mathsf{Ty}(\Gamma)}(\mathbf{Unit}_{\mathcal{M}}) = \mathbf{Unit}_{\mathcal{N}} \qquad F_{\mathsf{Tm}(\Gamma,\mathbf{Unit}_{\mathcal{M}})}(\mathbf{tt}_{\mathcal{M}}) = \mathbf{tt}_{\mathcal{N}}$$

The following is a direct rephrasing of these equations:

Lemma 6.4.8. *If $F : \mathcal{M} \longrightarrow \mathcal{N}$ is a morphism of models of base type theory and $\mathcal{M}$ and $\mathcal{N}$ are both equipped with a choice of unit types, F extends to a morphism of models with* **Unit** *just when the following diagram commutes:*



For a general connective Θ, we can specify the commutation of F with the operations of Θ using a diagram based on the commuting square specifying the formation and introduction data of a connective (Slogans 6.2.10 and 6.3.13). In particular, we have no need to specify that the elimination operator is also preserved, as this follows for free.

Remark 6.4.9. Note that if we instead used Slogan 6.3.31, we would have to impose additional requirements to make sure that F commuted appropriately with the chosen weak stable orthogonality structure. ◇

However, some care is required. In the case of **Unit**, we took advantage of the fact that F^* preserves $\mathbf{1}$ and therefore that we could relate the formation data for $\mathbf{Unit}_{\mathcal{M}}$ to that of $\mathbf{Unit}_{\mathcal{N}}$. We will not have an isomorphism $F^*(F_{\Theta_{\mathcal{N}}}) \cong F_{\Theta_{\mathcal{M}}}$ for each connective Θ, but we are always able to construct a canonical map $F_{\Theta_{\mathcal{M}}} \longrightarrow F^*(F_{\Theta_{\mathcal{N}}})$ for the connectives of ETT. For instance, since F^* preserves limits and colimits and there are maps $\mathsf{Ty}_{\mathcal{M}} \longrightarrow F^*\mathsf{Ty}_{\mathcal{N}}$ and $\mathsf{Tm}_{\mathcal{M}} \longrightarrow F^*\mathsf{Tm}_{\mathcal{N}}$, there are canonical (but non-invertible!) maps relating the formation data of **Eq**, **Bool**, and **Void**.

The cases of $\mathbf{\Pi}$ and $\mathbf{\Sigma}$ are slightly more complex, as they involve polynomial functors. We illustrate this principle for $\mathbf{\Pi}$ in detail and leave it to the reader to extrapolate the principle to other connectives.

Definition 6.4.10. A morphism of models of base type theory $F : \mathcal{M} \longrightarrow \mathcal{N}$ extends to a morphism of models of type theory with $\mathbf{\Pi}$ if it satisfies the following equations for all $\Gamma : \mathsf{Cx}_{\mathcal{M}}$, $A : \mathsf{Ty}_{\mathcal{M}}(\Gamma)$, $B : \mathsf{Ty}_{\mathcal{M}}(\Gamma.{}_{\mathcal{M}}A)$, and $b : \mathsf{Tm}_{\mathcal{M}}(\Gamma.{}_{\mathcal{M}}A, B)$:

$$F_{\mathsf{Ty}(\Gamma)}(\mathbf{\Pi}_{\mathcal{M}}(A,B)) = \mathbf{\Pi}_{\mathcal{M}}(F_{\mathsf{Ty}(\Gamma)}(A), F_{\mathsf{Ty}(\Gamma.{}_{\mathcal{M}}A)}(B))$$
$$F_{\mathsf{Tm}(\Gamma,\mathbf{\Pi}_{\mathcal{M}}(A,B))}(\lambda_{\mathcal{M}}(b)) = \lambda_{\mathcal{N}}(F_{\mathsf{Tm}(\Gamma.{}_{\mathcal{M}}A,B)}(b))$$

Notice that we have not included any equations governing **app**. This is because the desired equation holds *automatically* thanks to those equations governing λ along with the β and η laws for $\mathbf{\Pi}$-types:

Lemma 6.4.11. *If $F : \mathcal{M} \longrightarrow \mathcal{N}$ is a morphism of models with $\mathbf{\Pi}$-types then the following holds for all $\Gamma : \mathsf{Cx}_{\mathcal{M}}$, $A : \mathsf{Ty}_{\mathcal{M}}(\Gamma)$, $B : \mathsf{Ty}_{\mathcal{M}}(\Gamma.{}_{\mathcal{M}}A)$, $a : \mathsf{Tm}_{\mathcal{M}}(\Gamma, A)$, and $f : \mathsf{Tm}_{\mathcal{M}}(\Gamma, \mathbf{\Pi}_{\mathcal{M}}(A,B))$:*

$$F_{\mathsf{Tm}(\Gamma,B[\mathbf{id}_{\mathcal{M}}.{}_{\mathcal{M}}a]_{\mathcal{M}})}(\mathbf{app}_{\mathcal{M}}(f,a)) = \mathbf{app}_{\mathcal{M}}(F_{\mathsf{Tm}(\Gamma,\mathbf{\Pi}_{\mathcal{M}}(A,B))}(f), F_{\mathsf{Tm}(\Gamma,A)}(a))$$

Proof. This is a consequence of the β and η laws:

$$\begin{aligned}
&F_{\mathsf{Tm}(\Gamma,B[\mathbf{id}_{\mathcal{M}}.{}_{\mathcal{M}}a]_{\mathcal{M}})}(\mathbf{app}_{\mathcal{M}}(f,a)) \\
&= F_{\mathsf{Tm}(\Gamma,B[\mathbf{id}_{\mathcal{M}}.{}_{\mathcal{M}}a]_{\mathcal{M}})}(\mathbf{app}_{\mathcal{M}}(f[\mathbf{p}_{\mathcal{M}}], \mathbf{q}_{\mathcal{M}})[\mathbf{id}_{\mathcal{M}}.{}_{\mathcal{M}}a]) \\
&= F_{\mathsf{Tm}(\Gamma,B)}(\mathbf{app}_{\mathcal{M}}(f[\mathbf{p}_{\mathcal{M}}], \mathbf{q}_{\mathcal{M}}))[\mathbf{id}_{\mathcal{N}}.{}_{\mathcal{N}}F_{\mathsf{Tm}(\Gamma,A)}(a)] \\
&= \mathbf{app}_{\mathcal{N}}(\lambda_{\mathcal{N}}(F_{\mathsf{Tm}(\Gamma,B)}(\mathbf{app}_{\mathcal{M}}(f[\mathbf{p}_{\mathcal{M}}], \mathbf{q}_{\mathcal{M}}))), F_{\mathsf{Tm}(\Gamma,A)}(a)) \\
&= \mathbf{app}_{\mathcal{N}}(F_{\mathsf{Tm}(\Gamma,\mathbf{\Pi}_{\mathcal{M}}(A,B))}(\lambda_{\mathcal{M}}(\mathbf{app}_{\mathcal{M}}(f[\mathbf{p}_{\mathcal{M}}], \mathbf{q}_{\mathcal{M}}))), F_{\mathsf{Tm}(\Gamma,A)}(a)) \\
&= \mathbf{app}_{\mathcal{N}}(F_{\mathsf{Tm}(\Gamma,\mathbf{\Pi}_{\mathcal{M}}(A,B))}(f), F_{\mathsf{Tm}(\Gamma,A)}(a))
\end{aligned}$$

□

Remark 6.4.12. This proof is essentially a combination of the inter-derivability between **app** and λ^{-1} along with the observation that natural transformations which are pointwise isomorphisms are natural isomorphisms. ◇

We will now reformulate the equational presentation of Definition 6.4.10 into a less symbol-heavy diagrammatic formulation as was done for **Unit**. To start with, we must specify the canonical maps between the formation and introduction data of $\mathbf{\Pi}_{\mathcal{M}}$ and $\mathbf{\Pi}_{\mathcal{N}}$.

Lemma 6.4.13. *If $F : \mathcal{M} \longrightarrow \mathcal{N}$ then there is a canonical map $\alpha : \mathbf{P}_{\pi_{\mathcal{M}}}\mathsf{Ty}_{\mathcal{M}} \longrightarrow F^*(\mathbf{P}_{\pi_{\mathcal{N}}}\mathsf{Ty}_{\mathcal{N}})$.*

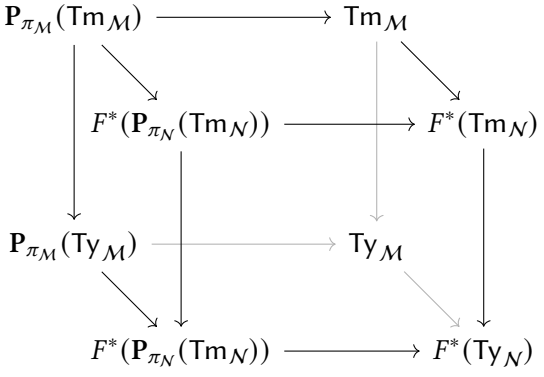
Proof. This is easiest to show using Lemma 6.2.15: if $\Gamma : \mathsf{Cx}_{\mathcal{M}}$ then $\mathbf{P}_{\pi_{\mathcal{M}}}\mathsf{Ty}_{\mathcal{M}}(\Gamma)$ consists of pairs $\sum_{A:\mathsf{Ty}_{\mathcal{M}}(\Gamma)} \mathsf{Ty}_{\mathcal{M}}(\Gamma.{}_{\mathcal{M}}A)$. Similarly, $F^*(\mathbf{P}_{\pi_{\mathcal{N}}}\mathsf{Ty}_{\mathcal{N}})(\Gamma) \cong \sum_{A:\mathsf{Ty}_{\mathcal{N}}(F(\Gamma))} \mathsf{Ty}_{\mathcal{M}}(F(\Gamma).{}_{\mathcal{N}}A)$. We now use F_{Ty} while taking advantage of the fact that $F(\Gamma.{}_{\mathcal{M}}A) = F(\Gamma).{}_{\mathcal{N}}F_{\mathsf{Ty}(\Gamma)}(A)$:

$$\alpha\,\Gamma\,(A, B) = (F_{\mathsf{Ty}(\Gamma)}(A), F_{\mathsf{Ty}(\Gamma.{}_{\mathcal{M}}A)}(B))$$

We leave it to the reader to check that this assignment is natural. □

Lemma 6.4.14. *If $F : \mathcal{M} \longrightarrow \mathcal{N}$ then there is a canonical map $\alpha : \mathbf{P}_{\pi_{\mathcal{M}}}\mathsf{Tm}_{\mathcal{M}} \longrightarrow F^*(\mathbf{P}_{\pi_{\mathcal{N}}}\mathsf{Tm}_{\mathcal{N}})$.*

Lemma 6.4.15. *If $F : \mathcal{M} \longrightarrow \mathcal{N}$ is a morphism of models of base type theory, F extends to a morphism of models of type theory with $\mathbf{\Pi}$ just when the following diagram commutes:*



Proof. Note that the front, back, left, and right faces commute for an arbitrary morphism of models of base type theory. It therefore suffices to show that extending to a morphism to support $\mathbf{\Pi}$ is equivalent to the commutation of the top and bottom squares. Unfolding, the commutation of the bottom square is equivalent to the following equation for all $\Gamma : \mathsf{Cx}_{\mathcal{M}}$, $A : \mathsf{Ty}_{\mathcal{M}}(\Gamma)$, and $B : \mathsf{Ty}_{\mathcal{M}}(\Gamma.{}_{\mathcal{M}}A)$:

$$F_{\mathsf{Ty}(\Gamma)}(\mathbf{\Pi}_{\mathcal{M}}(A, B)) = \mathbf{\Pi}_{\mathcal{N}}(F_{\mathsf{Ty}(\Gamma)}(A), F_{\mathsf{Ty}(\Gamma.{}_{\mathcal{M}}A)}(B))$$

Similarly, the bottom square is equivalent to the following equation for all $\Gamma : \mathsf{Cx}_{\mathcal{M}}$, $A : \mathsf{Ty}_{\mathcal{M}}(\Gamma)$, $B : \mathsf{Ty}_{\mathcal{M}}(\Gamma.{}_{\mathcal{M}}A)$, and $b : \mathsf{Tm}_{\mathcal{M}}(\Gamma.{}_{\mathcal{M}}A, B)$:

$$F_{\mathsf{Tm}(\Gamma,\mathbf{\Pi}_{\mathcal{M}}(A,B))}(\lambda_{\mathcal{M}}(A, B, b)) = \lambda_{\mathcal{N}}(F_{\mathsf{Ty}(\Gamma)}(A), F_{\mathsf{Ty}(\Gamma.{}_{\mathcal{M}}A)}(B), F_{\mathsf{Tm}(\Gamma.{}_{\mathcal{M}}A,B)}(b))$$

These exactly correspond to the requirements ensuring that F preserve $\mathbf{\Pi}$. □

Remark 6.4.16. We can re-express the above 3-dimensional diagram into a square in $\mathbf{Pr}(\mathsf{Cx}_{\mathcal{M}})^{\to}$:

$$\begin{array}{ccc} \mathbf{P}_{\pi_{\mathcal{M}}}(\pi_{\mathcal{M}}) & \longrightarrow & \pi_{\mathcal{M}} \\ \downarrow & & \downarrow \\ F^*(\mathbf{P}_{\pi_{\mathcal{N}}}(\pi_{\mathcal{N}})) & \longrightarrow & F^*\pi_{\mathcal{N}} \end{array}$$

◇

In total then, a morphism $F : \mathcal{M} \longrightarrow \mathcal{N}$ of models of type theory with some set of connectives consists of a morphism of base type theory which satisfies the additional properties required to commute with all relevant connectives.

6.4.2 *Universes as sub-models*

We now reap the rewards of our effort investigating morphisms of models of type theory, as it allows us to give a concise definition of when a model $\mathcal{M}$ supports a hierarchy of universes. For this subsection, let us fix a model $\mathcal{M}$ and we will once more suppress $\mathcal{M}$ as a subscript, instead simply writing *e.g.*, Ty or $\mathbf{\Pi}$.

Structure 6.4.17. A universe structure on a model of type theory $\mathcal{M}$ consists of the following:

- A type $\mathbf{U}_{0,\Gamma} : \mathsf{Ty}_{\mathcal{M}}(\Gamma)$ for every $\Gamma : \mathsf{Cx}_{\mathcal{M}}$ and a family of types $\mathbf{El}_{0,\Gamma} : \mathsf{Ty}_{\mathcal{M}}(\Gamma.\mathbf{U}_{0,\Gamma})$.
- Equations $\mathbf{U}_{0,\Gamma}[\gamma] = \mathbf{U}_{0,\Delta}$ and $\mathbf{El}_{0,\Gamma}(c)[\gamma] = \mathbf{El}_{0,\Delta}(c[\gamma])$ for every $\gamma : \mathsf{Sb}_{\mathcal{M}}(\Delta, \Gamma)$ and $c : \mathsf{Tm}_{\mathcal{M}}(\Gamma, \mathbf{U}_{0,\Gamma})$
- For each of $\mathbf{\Pi}$, Σ, **Eq**, **Unit**, **Bool**, $+$, **Void**, **Nat**, there is an operation **pi**, **sig**, **eq**, **unit**, **bool**, **coprod**, **void**, **nat** *e.g.*, $\mathbf{pi}(c_0, c_1) : \mathsf{Tm}_{\mathcal{M}}(\Gamma, \mathbf{U}_0)$ whenever $c_0 : \mathsf{Tm}_{\mathcal{M}}(\Gamma, \mathbf{U}_0)$ and $c_1 : \mathsf{Tm}_{\mathcal{M}}(\Gamma.\mathbf{El}_0(c_0), \mathbf{U}_0)$.
- For each of the connectives above, an equation stating that the operator commutes with substitution *e.g.*, $\mathbf{pi}(c_0, c_1)[\gamma] = \mathbf{pi}(c_0[\gamma], c_1[\gamma.\mathbf{El}(c_0)])$ whenever $\gamma : \mathsf{Sb}_{\mathcal{M}}(\Delta, \Gamma)$, $c_0 : \mathsf{Tm}_{\mathcal{M}}(\Gamma, \mathbf{U}_0)$ and $c_1 : \mathsf{Tm}_{\mathcal{M}}(\Gamma.\mathbf{El}_0(c_0), \mathbf{U}_0)$.
- For each of the connectives above, an equation stating that **El** commutes with the operation *e.g.*, $\mathbf{El}_0(\mathbf{pi}(c_0, c_1)) = \mathbf{pi}(\mathbf{El}_0(c_0), \mathbf{El}_0(c_1))$.

As is routine, the first two points are equivalent to a pair of natural transformations:

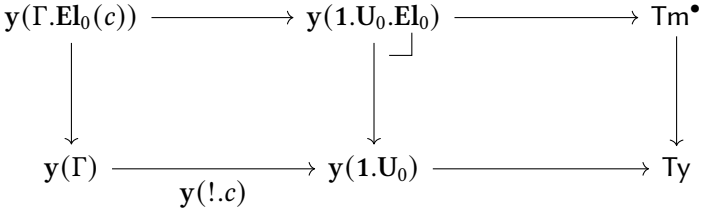
Lemma 6.4.18. *The operators* $\mathbf{U}_{0,\Gamma}$ *and* $\mathbf{El}_{0,\Gamma}$ *and the substitution equations on them are equivalent to a pair of natural transformations*

$$\mathbf{U}_0 : \mathbf{1} \longrightarrow \mathsf{Ty} \qquad \mathbf{El}_0 : \mathbf{U}_0^* \mathsf{Tm}^\bullet \longrightarrow \mathsf{Ty}$$

The challenge is to reformulate the final three points. While it is possible to specify operators such as **pi**, **sig**, and so on individually, this is rather laborious. Instead we opt for a different approach. We begin by observing the following:

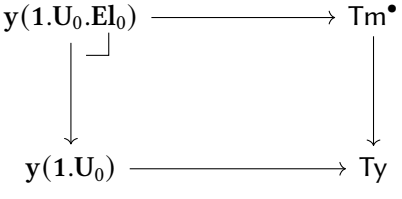
Lemma 6.4.19. *The projection* $\mathbf{y}(\mathbf{p}) : \mathbf{y}(\mathbf{1}.\mathbf{U}_0.\mathbf{El}_0) \longrightarrow \mathbf{y}(\mathbf{1}.\mathbf{U}_0)$ *obtains a canonical representability structure from* π*.*

Proof. Since $\mathbf{y}(\mathbf{p}) : \mathbf{y}(\mathbf{1}.\mathbf{U}_0.\mathbf{El}_0) \longrightarrow \mathbf{y}(\mathbf{1}.\mathbf{U}_0)$ is a pullback of π, the left-hand square in the following diagram is a pullback:



In particular, we may use $\mathbf{y}(\Gamma.\mathbf{El}_0(c))$ as the chosen pullback for the representability structure on $\mathbf{y}(\mathbf{p})$. □

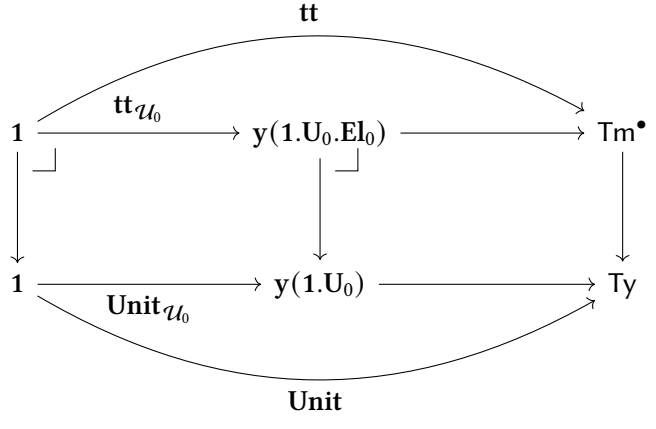
Corollary 6.4.20. $\mathsf{Cx}_{\mathcal{M}}$ *and* $\mathbf{y}(\mathbf{p}) : \mathbf{y}(\mathbf{1}.\mathbf{U}_0.\mathbf{El}_0) \longrightarrow \mathbf{y}(\mathbf{1}.\mathbf{U}_0)$ *form a model of base type theory* $\mathcal{U}_0$*. Moreover, the identity functor and the following commuting square then induce a morphism of models* $I : \mathcal{U}_0 \longrightarrow \mathcal{M}$*:*



Lemma 6.4.21. *The remaining structure specifying a universe in* $\mathcal{M}$ *is equivalent to the data equipping* $\mathcal{U}_0$ *with all the connectives of type theory such that* I *induces a morphism of models.*

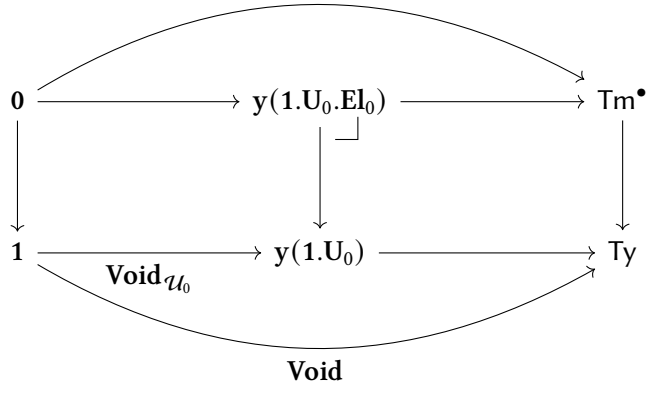
Proof. We describe this explicitly for **Void** and **Unit**, as the remaining connectives are identical but more notationally cumbersome. In the case of **Unit**, to equip $\mathcal{U}_0$ with a

unit type such that I is a morphism of models is equivalent to choosing a left-hand square in the following diagram:



Since both squares are required to be pullbacks, a choice of the left-hand diagram is fully determined by a morphism $\mathbf{Unit}_{\mathcal{U}_0} : \mathbf{1} \longrightarrow \mathbf{y}(\mathbf{1}.\mathbf{U}_0)$ such that the bottom triangle commutes. This precisely corresponds to the data closing $\mathbf{U}_0$ under **Unit** in $\mathcal{M}$.

For **Void**, the procedure is similar. Equipping $\mathbf{y}(\mathbf{p})$ with an interpretation of **Void** such that I is a morphism of models corresponds to picking a left-hand square in the following diagram, subject to an orthogonality condition:



The orthogonality condition states that the map $X \times \mathbf{0} \longrightarrow X \times \mathbf{Void}^*_{\mathcal{U}_0}\mathbf{y}(\mathbf{1}.\mathbf{U}_0.\mathbf{El}_0)$ is orthogonal to $\mathbf{y}(\mathbf{p})$. Since the right-hand square is a pullback, the left-hand map is equivalent to $X \times \mathbf{0} \longrightarrow X \times \mathbf{Void}^*\mathsf{Tm}^\bullet$ and since $\mathbf{y}(\mathbf{p})$ is a pullback of π, this condition is automatic.

In particular, the only requirement in the choice of such a left-hand square is the map $\mathbf{Void}_{\mathcal{U}_0} : \mathbf{1} \longrightarrow \mathbf{y}(\mathbf{1}.\mathbf{U}_0)$ subject to the commuting triangle above. This is equivalent to the data closing $\mathbf{U}_0$ under **Void** in $\mathcal{M}$ as required. □

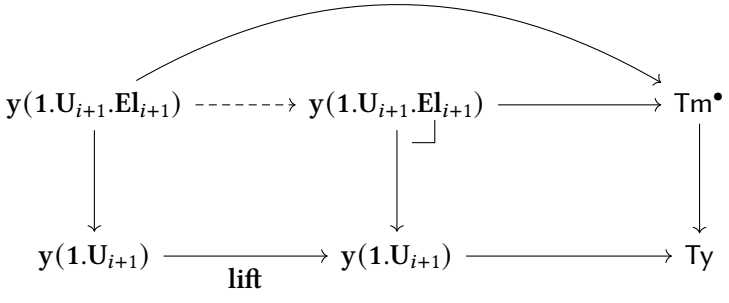
Theorem 6.4.22 (Categorical reformulation of **U**). *A universe structure on* $\mathcal{M}$ *is equivalent to the following:*

- *A choice of natural transformations* $\mathbf{U}_0 : \mathbf{1} \longrightarrow \mathsf{Ty}$ *and* $\mathbf{El}_0 : \mathbf{U}_0^* \mathsf{Tm}^\bullet \longrightarrow \mathsf{Ty}$
- *An interpretation of the connectives* $\mathbf{\Pi}$, $\mathbf{\Sigma}$, **Unit**, **Eq**, **Void**, **Bool**, **Nat**, *and* $+$ *into the model* $\mathcal{U}_0 = (\mathsf{Cx}_{\mathcal{M}}, \mathbf{y}(\mathbf{1}.\mathbf{U}_0.\mathbf{El}_0) \longrightarrow \mathbf{y}(\mathbf{1}.\mathbf{U}_0))$ *such that the canonical map* $I : \mathcal{U}_0 \longrightarrow \mathcal{M}$ *is a morphism of models with all of these connectives.*

Hierarchies of universes With Theorem 6.4.22, it is straightforward to describe the requirement that $\mathcal{M}$ supports a hierarchy of universes. Given the amount of data that is required to describe such a hierarchy in an unfolded fashion, we will present the categorical repackaging and leave it to the diligent reader to compare with Definition 3.4.2.

Lemma 6.4.23 (Categorical reformulation of a hierarchy). $\mathcal{M}$ *supports a cumulative hierarchy of universes just when it is equipped with the following:*

- *For each* $i : \mathbb{N}$, *a choice of natural transformations* $\mathbf{U}_i : \mathbf{1} \longrightarrow \mathsf{Ty}$ *and* $\mathbf{El}_i : \mathbf{U}_i^* \mathsf{Tm}^\bullet \longrightarrow \mathsf{Ty}$
- *For each* i, *an interpretation of the connectives* $\mathbf{\Pi}$, $\mathbf{\Sigma}$, **Unit**, **Eq**, **Void**, **Bool**, **Nat**, $+$, *and* $\mathbf{U}_j$ *for all* $j < i$ *into the model* $\mathcal{U}_i = (\mathsf{Cx}_{\mathcal{M}}, \mathbf{y}(\mathbf{1}.\mathbf{U}_i.\mathbf{El}_i) \longrightarrow \mathbf{y}(\mathbf{1}.\mathbf{U}_i))$ *such that the canonical map* $\mathcal{U}_i \longrightarrow \mathcal{M}$ *is a morphism of models.*
- *For each* i, *a natural transformation* $\mathbf{lift} : \mathbf{y}(\mathbf{1}.\mathbf{U}_i) \longrightarrow \mathbf{y}(\mathbf{1}.\mathbf{U}_{i+1})$ *such that the outer square commutes and the left-hand square in the following diagram is a pullback:*



Moreover, we require that the left-hand square induce a morphism of models $\mathcal{U}_i \longrightarrow \mathcal{U}_{i+1}$.

Exercise 6.20. Isolate the necessary operations and equations on a model for supporting a hierarchy of universes and argue that this structure is equivalent to the requirements of Lemma 6.4.23.

6.5 *Democratic models and local cartesian closure*

Thus far in this chapter, we have spent a considerable amount of effort investigating the definition of a model of type theory. Despite this effort, we have only met two examples of models: the syntactic model (Theorem 3.4.5) and the set model (Section 3.5). In general, constructing a model of type theory is hard work because of all the data that must be chosen and the properties that must be checked. Ideally, we would hope for a theorem which tells us that any suitably structured category can be upgraded automatically into a model of type theory. We would then be able to apply this theorem to *e.g.*, **Set** and automatically obtain the main results of Section 3.5.

This leaves an obvious question: what structure should a category possess to serve as input to this wondrous theorem? Scrutinizing our list of types in ETT, we can make some deductions. For instance, in order to interpret **Unit**, we should certainly require a terminal object. For extensional equality $\mathbf{Eq}(A, a, b)$, we will need equalizers. But other connectives are a bit less clear (what about natural numbers? $\mathbf{\Pi}$-types?).

In order to arrive at a plausible list of requirements, in this section we attack the question from the opposite direction: given a model of type theory $\mathcal{M}$, what properties does $\mathsf{Cx}_{\mathcal{M}}$ satisfy? We shall see that a number of type-theoretic connectives correspond directly to recognizable categorical structures. In particular, we shall show that for well-behaved models, the category of contexts is finitely complete, locally cartesian closed and possesses finite coproducts and a natural number object. We shall return to our original question in Section 6.6 where we shall see that, in good cases, this assignment $\mathcal{M} \mapsto \mathsf{Cx}_{\mathcal{M}}$ sending a model to a category satisfying the above list of requirements is essentially reversible.

6.5.1 *Democratic models*

In this subsection, we will fix a model of type theory $\mathcal{M}$ which we will assume to be *democratic*. Roughly, our goal is to analyze $\mathsf{Cx}_{\mathcal{M}}$ as a category and so it is useful to know that the behavior of $\mathsf{Cx}_{\mathcal{M}}$ is fully controlled by types. That is, to assume that every context is built from the empty context by repeatedly extending with types. Note that while this is true for the syntactic model $\mathcal{T}$, it need not hold in arbitrary models.

Definition 6.5.1. A model $\mathcal{M}$ is democratic if for every context $\Gamma : \mathsf{Cx}_{\mathcal{M}}$ there exists a type $A : \mathsf{Ty}_{\mathcal{M}}(\mathbf{1}_{\mathcal{M}})$ along with an isomorphism $\Gamma \cong \mathbf{1}_{\mathcal{M}} \cdot_{\mathcal{M}} A$.

Lemma 6.5.2. *The syntactic model $\mathcal{T}$ is democratic.*

Proof. While this may seem obvious, a modicum of effort is required to apply the induction principle for the syntactic model (Theorem 3.4.5) and we spell out some of the details here to illustrate the process.

We will construct a model of type theory $\mathcal{T}_0$ along with a homomorphism $i : \mathcal{T}_0 \longrightarrow \mathcal{T}$ and we will further arrange for contexts in $\mathcal{T}_0$ to be syntactic contexts Γ for which there exists a closed type A and isomorphism $\mathbf{1}.A \cong \Gamma$. The map i will then send Γ in $\mathcal{T}_0$ to Γ in T. By initiality, there is a unique model homomorphism $! : \mathcal{T} \longrightarrow \mathcal{T}_0$ and (by initiality once more) we must have $i \circ ! = \mathsf{id}$. Consequently, i is a split epimorphism and so every $\Gamma : \mathsf{Cx}_{\mathcal{T}}$ is in the image of i—precisely what we were attempting to prove.

It remains, therefore, only to construct $\mathcal{T}_0$ and i. Let us take the category of contexts $\mathsf{Cx}_{\mathcal{T}_0}$ for $\mathcal{T}_0$ to be the full subcategory of $\mathcal{T}$ spanned by contexts Γ for which there exists an isomorphism $\Gamma \cong \mathbf{1}.A$ for some closed type A. The chosen terminal object of $\mathsf{Cx}_{\mathcal{T}}$ lands in this full subcategory: $\mathbf{1} \cong \mathbf{1}.\mathbf{Unit}$. We write i for the inclusion functor $\mathcal{T}_0 \longrightarrow \mathcal{T}$. The presheaves of types and terms over $\mathsf{Cx}_{\mathcal{T}_0}$ are given by restricting those from $\mathcal{T}$:

$$\mathsf{Ty}_{\mathcal{T}_0} = i^*(\mathsf{Ty}_{\mathcal{T}}) = \mathsf{Ty}_{\mathcal{T}} \circ i \qquad \mathsf{Tm}^\bullet_{\mathcal{T}_0} = i^*(\mathsf{Tm}^\bullet_{\mathcal{T}}) = \mathsf{Tm}^\bullet_{\mathcal{T}} \circ i \qquad \pi_{\mathcal{T}_0} = i^*(\pi_{\mathcal{T}}) = \pi_{\mathcal{T}} \circ i$$

To show that $\pi_{\mathcal{T}_0}$ is representable, recall that (1) i^* preserves (co)limits and (2) $i^*\mathbf{y}(i(\Gamma)) \cong \mathbf{y}(\Gamma)$ since i is fully-faithful. It therefore suffices to show if $\Gamma : \mathsf{Cx}_{\mathcal{T}_0}$ and $A \in \mathsf{Ty}_{\mathcal{T}_0}(\Gamma)$ then $i(\Gamma).{}_{\mathcal{T}}A$ lies in the image of i. By construction, there must be some B such that $i(\Gamma) \cong \mathbf{1}.B$ and so $\mathbf{1}.\Sigma(B, A) \cong i(\Gamma).A$ as required.

Finally, we must show that $\mathcal{T}_0$ is closed under all the connectives of type theory and that i extends to a homomorphism of models. There is a conceptual reason for this: all connectives may be defined using finite limits and polynomial functors $\mathbf{P}_f$ where f is a morphism built from pullbacks, composites, and π. One may check that i^* preserves all of these operations—for polynomials, one uses Lemma 6.2.15—and therefore applying i^* to structure closing $\mathcal{T}$ under each connective yields the appropriate structure in $\mathcal{T}_0$. Moreover, one obtains a morphism of models extending i using $\mathsf{id} : \pi_{\mathcal{T}_0} \longrightarrow i^*\pi_{\mathcal{T}}$ (Lemma 6.4.5) which commutes with all connectives more-or-less tautologically.

However, a much less sophisticated though more tedious approach suffices: one may simply show that each operation listed in Definition 3.4.2 can be defined on $\mathcal{T}_0$ using the appropriate operation on $\mathcal{T}$. For instance, for $\mathbf{\Pi}$ we must define the following:

$$\mathbf{\Pi}_{\mathcal{T}_0} : (\Gamma : \mathsf{Cx}_{\mathcal{T}_0})(A : \mathsf{Ty}_{\mathcal{T}_0}(\Gamma))(B : \mathsf{Ty}_{\mathcal{T}_0}(\Gamma.{}_{\mathcal{T}_0}A)) \to \mathsf{Ty}_{\mathcal{T}_0}(\Gamma)$$

We choose $\mathbf{\Pi}_{\mathcal{T}_0} = \mathbf{\Pi}_{\mathcal{T}}$ which is well-formed because $i(\Gamma.{}_{\mathcal{T}_0}A) = i(\Gamma).{}_{\mathcal{T}}A$ by definition. The same procedure and argument works for every other operation. □

6.5.2 *The category of contexts and types in a democratic model*

We shall hereafter restrict our attention to a particular democratic model $\mathcal{M}$—we shall have the syntactic model $\mathcal{T}$ in mind, but there is no difference in working at this level of generality.

Notation 6.5.3. We shall once more begin suppressing the subscript $_{\mathcal{M}}$ in what follows to declutter the notation.

We have observed all the way back in Chapter 2 that the terms of a type $A \in \mathsf{Ty}(\Gamma)$ can be recovered from Cx through the weakening substitution $\mathbf{p} : \Gamma.A \longrightarrow \Gamma$. More generally, if $A, B \in \mathsf{Ty}(\Gamma)$ then there is an isomorphism between functions $\mathsf{Tm}(\Gamma, A \to B)$ and $\hom_{\mathsf{Cx}_{/\Gamma}}(\Gamma.A \longrightarrow \Gamma, \Gamma.B \longrightarrow \Gamma)$. We shall use this to produce a more convenient description of the slice category $\mathsf{Cx}_{/\Gamma}$ using terms of types and terms in context Γ. We begin by showing that each substitution is isomorphic to a weakening substitution:

Lemma 6.5.4. *If $\delta : \Gamma \longrightarrow \Delta$ then there exists $\mathbf{p}_A : \Delta.A \longrightarrow \Delta$ along with an isomorphism $\delta \cong \mathbf{p}_A$ in $\mathsf{Cx}_{/\Delta}$.*

Proof. By democracy, we know that $\Gamma \cong \mathbf{1}.A_0$ and $\Delta \cong \mathbf{1}.B$ for some B. Without loss of generality, we may replace Γ by $\mathbf{1}.A_0$ and Δ by $\mathbf{1}.B$ such that $\delta : \Gamma \longrightarrow \Delta$ is of the form $!.b$ where $b \in \mathsf{Tm}(\mathbf{1}.A_0, B[\mathbf{p}])$.

We then choose $A \in \mathsf{Ty}(\Delta)$ to be $\Sigma(A_0[!], \mathbf{Eq}(B[!], \mathbf{q}[\mathbf{p}], b[!.A_0]))$. In informal notation: $\mathbf{1}, x : B \vdash \sum_{a:A_0} \mathbf{Eq}(B, b(a), x)$ type. Next, we must construct an isomorphism $\delta \cong \mathbf{p}_A$. For this, we choose $f_0 = \delta.\mathbf{pair}(\mathbf{q}, \mathbf{refl}) : \delta \longrightarrow \mathbf{p}_A$ for one direction and $f_1 = !.\mathbf{fst}(\mathbf{q}) : \mathbf{p}_A \longrightarrow \delta$ for the other. For the latter, note that we must use equality reflection to ensure that $\delta \circ f_1 = \mathbf{p}$ as required of a morphism in $\mathsf{Cx}_{/\Delta}$. We leave it to the reader to check that these are inverses using the β and η laws for Σ and $\mathbf{Eq}$. □

Advanced Remark 6.5.5. Homotopy-theoretic readers may observe that there is some similarity between the replacement of $\Gamma \to \Delta$ by $\mathbf{p}_A$ and the factorization of a map of spaces $f : X \to Y$ into a trivial cofibration followed by a fibration $X \to X \times_Y Y^{[0,1]} \to Y$. This would be particularly evident if we replaced $\mathbf{Eq}$ with $\mathbf{Id}$ and used the dictionary between intensional type theory and homotopy theory explored in Chapter 5. In fact, this same factorization exists for intensional identity types and can be used to structure the category of contexts of a model of ITT into a *fibration category* [GG08; AKL15]. In the case of $\mathbf{Eq}$, the first map is a genuine isomorphism so this factorization system is trivial. ◇

Corollary 6.5.6. *There is an equivalence of categories between $\mathsf{Cx}_{/\Gamma}$ and the category of types $\mathsf{Ty}(\Gamma)$ whose morphisms $\hom(A, B)$ are given by functions $\mathsf{Tm}(\Gamma, A \to B)$.*

Remark 6.5.7. We emphasize that types in context Γ are viewed as maps *into* Γ. This is a curious reversal from both the notation $\Gamma \vdash A\ \mathsf{type}$ and Section 6.1 where Γ behaves like a domain of some function in both. This trick of viewing maps *into* an object as families indexed by that object is common in category theory and geometry; it allows us to define families even in the absence of an "object of objects". More concretely, $\mathsf{Ty} : \mathbf{Pr}(\mathsf{Cx})$ is not representable and so we must express dependent types (maps $\mathbf{y}(\Gamma) \longrightarrow \mathsf{Ty}$) more indirectly. We shall analyze the extent to which this process can be reversed in Section 6.6.1 ◇

The pullback functors $\gamma^* : \mathsf{Cx}_{/\Gamma} \longrightarrow \mathsf{Cx}_{/\Delta}$ for each $\gamma : \Delta \longrightarrow \Gamma$ also admit a familiar description when transported along the equivalence constructed in Lemma 6.5.4. By Exercise 6.4, pulling back $\Gamma.A \longrightarrow \Gamma$ along γ yields $\Delta.A[\gamma] \longrightarrow \Delta$ and the reader may compute that it sends a morphism $\mathbf{p}.b : \Gamma.A \longrightarrow \Gamma.B$ to $\mathbf{p}.b[\delta.A]$. In other words, when translating between slice categories and terms and types in context, the pullback operation between contexts is realized by substitution on terms and types.

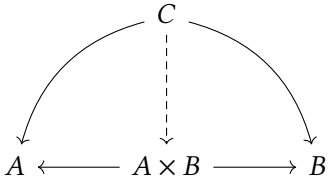
6.5.3 *Types with mapping-in properties*

With all of this effort, we can quickly rattle off a list of categorical properties satisfied by Cx by leveraging corresponding types. This discussion closely tracks the structure of Chapter 2: types with mapping-in properties correspond to limits and right adjoints, those with mapping out properties to left adjoints and colimits, and finally universes occupy an uneasy position of their own. As ever, we begin with those types with mapping-in characterizations.

Lemma 6.5.8. *Every slice category* $\mathsf{Cx}_{/\Gamma}$ *has finite products. Consequently,* Cx *has all finite limits.*

Proof. Every slice category has terminal objects—in the form of id_Γ—and so it suffices to show that $\mathsf{Cx}_{/\Gamma}$ has binary products. Passing to considering $\mathsf{Ty}(\Gamma)$ in context Γ, we claim the product of $A, B \in \mathsf{Ty}(\Gamma)$ is given by $A \times B$ (the non-dependent version of Σ).

To prove this, we must complete a programming exercise. We must argue that if $C \in \mathsf{Ty}(\Gamma)$ and $f \in \mathsf{Tm}(\Gamma, C \to A)$ and $g \in \mathsf{Tm}(\Gamma, C \to B)$ then there is a unique function $\langle f, g\rangle \in \mathsf{Tm}(\Gamma, C \to A \times B)$ such that $\mathbf{fst} \circ \langle f, g\rangle = f$ and $\mathbf{snd} \circ \langle f, g\rangle = g$:



We define $\langle f, g\rangle = \lambda c \to \mathbf{pair}(f(c), g(c))$ and the commutation of the diagram along with its uniqueness are then consequences of the β and η laws for Σ. □

Lemma 6.5.9. *If $\gamma : \Delta \longrightarrow \Gamma$ then $\gamma^* : \mathsf{Cx}_{/\Gamma} \longrightarrow \mathsf{Cx}_{/\Delta}$ commutes with finite products.*

Proof. It suffices to check this problem for $\mathsf{Ty}(\Gamma)$ and $\mathsf{Ty}(\Delta)$ where it is an immediate consequence of the stability of $\times$, **fst**, **snd**, and **pair** under substitution. □

Exercise 6.21. Show that $\mathsf{Cx}_{/\Gamma}$ has exponentials and these are preserved by γ^*.

Lemma 6.5.10. Cx *is locally cartesian closed.*

Proof. This is a general consequence of the observation that Cx has a terminal object and the fact that each slice category is cartesian closed and this structure is preserved by pullback functors. □

For the sake of completeness (and because the result is recognizable), we can give an explicit description of the right adjoint to pullback: $\gamma_* : \mathsf{Cx}_{/\Delta} \longrightarrow \mathsf{Cx}_{/\Gamma}$. We begin by replacing Δ and γ by $\mathbf{p}_A : \Gamma.A \longrightarrow \Gamma$. In this case, the right adjoint to $\mathbf{p}^*$ is given as follows:

$$B \in \mathsf{Ty}(\Gamma.A) \mapsto \mathbf{\Pi}(A, B)$$

To show this, it suffices to construct an isomorphism of the following shape natural in C:

$$\mathsf{Tm}(\Gamma, C \to \mathbf{\Pi}(A, B)) \cong \mathsf{Tm}(\Gamma.A, C[\mathbf{p}] \to B)$$

Using the mapping-in characterization of $\mathbf{\Pi}$, we may replace the left and right sides of this isomorphism with $\mathsf{Tm}(\Gamma.C.A[\mathbf{p}], B[\mathbf{p}.A])$ and $\mathsf{Tm}(\Gamma.A.C[\mathbf{p}], B[\mathbf{p}])$. These are naturally isomorphic by exchange.

Exercise 6.22. γ^* also has a left adjoint given by post-composition by γ (this holds whenever γ^* exists). Reformulate this left adjoint into another recognizable type-theoretic operation.

6.5.4 *Types with mapping-out properties*

Taking stock, thus far we have used $\mathbf{\Pi}$, Σ, **Eq**, and **Unit** (the last being a necessary consequence of democracy). What structure do the other connectives of dependent type theory induce? Following our noses from Section 6.3, we guess that the coproduct types + and the empty type **Void** suffice to close Cx under finite coproducts and **Nat** induces an initial algebra for $(\mathbf{1} \sqcup -)$.

Lemma 6.5.11. Cx *has finite coproducts.*

Proof. Considering the equivalent category $\mathsf{Ty}(\mathbf{1})$, we represent binary coproducts $A \sqcup B$ using the coproduct type, $A + B$. The rules governing this type are precisely those necessary for the universal property. Similarly, we realize the initial object with **Void**. □

Lemma 6.5.12. $\mathsf{Cx}_{/\Gamma}$ *has an initial* $(\mathbf{1} \sqcup -)$*-algebra (Definition 6.3.17) and pullback functors preserve this initial algebra.*

Remark 6.5.13. For the second claim to be well-formed, we must convince ourselves that $\gamma^* : \mathsf{Cx}_{/\Gamma} \longrightarrow \mathsf{Cx}_{/\Delta}$ induces a functor $\mathbf{Alg}(\mathbf{1}_{\mathsf{Cx}_{/\Gamma}} \sqcup -) \longrightarrow \mathbf{Alg}(\mathbf{1}_{\mathsf{Cx}_{/\Delta}} \sqcup -)$. This follows from the observation pullback commutes with both limits and colimits in a locally cartesian closed category. ◇

Proof. The initial $(\mathbf{1} \sqcup -)$-algebra in $\mathsf{Ty}(\Gamma)$ is given by **Nat** and the terms $\mathbf{zero} \in \mathsf{Tm}(\Gamma, \mathbf{Nat})$ and $\mathbf{suc} : \mathsf{Tm}(\Gamma, \mathbf{Nat} \to \mathbf{Nat})$. To prove initiality, let us fix A along with $a \in \mathsf{Tm}(\Gamma, A)$ and $s \in \mathsf{Tm}(\Gamma, A \to A)$. The unique algebra morphism $\alpha : \mathbf{Nat} \to A$ is given by the following function (written in informal notation for clarity): $\lambda n \to \mathbf{rec}(n, a, s)$. This organizes into an algebra morphism because of the β laws of **Nat** and it is unique with this property by the η law.

The commutation of these initial algebras with the pullback functor is then a consequence of the stability of **Nat** and the attendant operators under substitution. □

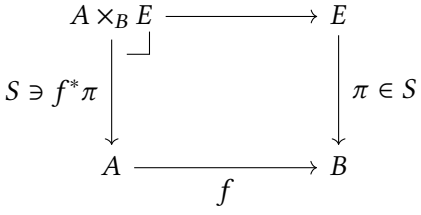
Clearly this collection of initial algebras is fully determined by the initial algebra for $\mathbf{1} \sqcup -$ in Cx. We shall call this algebra a *stably initial* $(\mathbf{1} \sqcup -)$-algebra.

Corollary 6.5.14. Cx *has a stably initial* $(\mathbf{1} \sqcup -)$*-algebra.*

6.5.5 *Identifying universes in the category of contexts*

It remains to discuss how the hierarchy of universes $\mathbf{U}_i$ fit into this story. Here the answer is somewhat messier because, unfortunately, universes in extensional type theory lack any clean description through universal properties. Indeed, we shall see in Section 6.6.4 that universe hierarchies can be a particular challenge when modeling type theory categorically. We shall roughly follow the approach proposed by Streicher [Str05]. To begin with, we recall the following definitions which roughly axiomatize the collection of maps isomorphic to $\mathbf{p} : \Gamma.\mathbf{El}(c) \longrightarrow \Gamma$ where $c \in \mathsf{Tm}(\Gamma, \mathbf{U})$:

Definition 6.5.15. If C is a category with finite limits a *bare universe* is a collection of morphisms S in C which is *stable under pullback*; if $\pi \in S$ then $f^*\pi \in S$ for all f:



Notation 6.5.16. Each universe induces a full subcategory $S_{/Y}$ of $C_{/Y}$ whose objects are those maps $f : X \longrightarrow Y \in S$. Closure under pullback ensures that pullback functors $y^* : C_{/Y_1} \longrightarrow C_{/Y_0}$ restrict to $S_{/Y_i}$. We say S contains an object C if $C \longrightarrow \mathbf{1} \in S$.

Obviously not much can be said about a bare categorical universe, but we can refine this definition to impose conditions matching the existence of **El** along with the closure properties of **U**. In other words, we insist that as a class of maps, S is generated by pulling back (applying a substitution to) a single map $(\mathbf{1}.\mathbf{U}.\mathbf{El}(\mathbf{q}) \longrightarrow \mathbf{1}.\mathbf{U})$ and is closed under all of the categorical structures induced by type-theoretic connectives.

Definition 6.5.17. Consider C is a locally cartesian closed category with finite coproducts and a stably initial $(\mathbf{1} \sqcup -)$-algebra and suppose further that S is a bare universe in C. We shall call a bare universe S a *universe* if it comes with a chosen map $\tau : U^\bullet \longrightarrow U \in S$ such that every map in $f : X \longrightarrow Y \in S$ can be presented $x^*\tau$ for some $x : Y \longrightarrow U$ (τ is *generic* for S) and such that it satisfies the following additional properties:

1. S contains all isomorphisms.
2. S is closed under composition.
3. If $f : X \longrightarrow Y$ and $g : Y \longrightarrow Z$ are in S then $f_*g \in S$.
4. If $f : X \longrightarrow Y \in S$ then $\Delta : X \longrightarrow X \times_Y X \in S$.
5. $S_{/Y}$ is closed under coproducts and contains the initial $(\mathbf{1} \sqcup -)$-algebras in $C_{/Y}$.

Lemma 6.5.18. *Each* $\mathbf{U}_i$ *induces a universe* $V_i = \{f \mid \exists c \in \mathsf{Tm}(\Gamma, \mathbf{U}_i).\, f \cong \mathbf{p} : \Gamma.\mathbf{El}(c) \longrightarrow \Gamma\}$.

Proof. As stated above, the generic map π is given by $\mathbf{1}.\mathbf{U}.\mathbf{El}(\mathbf{q}) \longrightarrow \mathbf{1}.\mathbf{U}$. To verify this, fix $\mathbf{p} : \Gamma.\mathbf{El}(c) \longrightarrow \Gamma$ with $\mathsf{Tm}(\Gamma, \mathbf{U}_i)$. The following diagram is a pullback:

$$\begin{array}{ccc} \Gamma.\mathbf{El}(c) & \longrightarrow & \mathbf{1}.\mathbf{U}.\mathbf{El}(\mathbf{q}) \\ \downarrow & & \downarrow \\ \Gamma & \xrightarrow[!.c]{} & \mathbf{1}.\mathbf{U} \end{array}$$

To verify properties (1–5), we use the closure of $\mathbf{U}_i$ under various connectives: **Unit** for (1), Σ for (2), $\mathbf{\Pi}$ for (3), **Eq** for (4), and $+$, **Void**, and **Nat** for (5). All of these properties are proven by essentially the same argument, so we illustrate the pattern by proving (3). Fix $f : \Gamma_0 \longrightarrow \Gamma_1$ and $g : \Gamma_1 \longrightarrow \Gamma_2$ such that $f, g \in S$. We must show that $g_*(f) \in S$.

First, since $f, g \in S$, we may replace them with isomorphic weakening maps and reduce to considering $f = \mathbf{p} : \Gamma.\mathbf{El}(c_0).\mathbf{El}(c_1) \longrightarrow \Gamma.\mathbf{El}(c_0)$ and $g = \mathbf{p} : \Gamma.\mathbf{El}(c_0) \longrightarrow \Gamma$. Above, we showed that if $\Gamma = \mathbf{1}$ then g_*f could be realized by $\Gamma.\mathbf{\Pi}(\mathbf{El}(c_0), \mathbf{El}(c_1)) \longrightarrow \Gamma$. The same argument applies to a general Γ and so it suffices to argue that

$$\Gamma.\mathbf{\Pi}(\mathbf{El}(c_0), \mathbf{El}(c_1)) \longrightarrow \Gamma \in S$$

Since $\mathbf{U}_i$ is closed under $\mathbf{\Pi}$, this map is equal to $\Gamma.\mathbf{El}(\mathbf{pi}(c_0, c_1)) \longrightarrow \Gamma$ and the conclusion is now immediate. □

Definition 6.5.19. A hierarchy of universes $S_0, S_1, \ldots$ in a category C consists of a collection of universes S_i such that $S_i \subseteq S_{i+1}$ and such that S_{i+1} contains U_i.

Lemma 6.5.20. *The collection $V_0, V_1, \cdots$ defined in Lemma 6.5.18 organizes into a hierarchy of universes.*

We summarize all of the insights of this discussion into the following theorem:

Theorem 6.5.21. *If $\mathcal{M}$ is a democratic model of type theory then $\mathsf{Cx}_{\mathcal{M}}$ is locally cartesian closed and has finite coproducts, a stably initial algebra for $\mathbf{1} \sqcup -$, and a hierarchy of universes.*

With additional effort, we could enhance Theorem 6.5.21 to the following theorem:

Theorem 6.5.22. *The map $\mathcal{M} \mapsto \mathsf{Cx}_{\mathcal{M}}$ induces a functor $\mathbf{CwF}_{\mathrm{dem}} \longrightarrow \mathbf{LCC}$ from the full subcategory of democratic models to the category of locally cartesian closed categories.*

We will not attempt to prove this theorem as we are not showing any sort of categorical equivalence between democratic models and locally cartesian closed categories (indeed, we would need to enhance locally cartesian closed categories to account for **Bool**, **Nat**, *etc.*). For further discussion on this point, see Clairambault and Dybjer [CD14].

6.6 *Strictification constructions*

As promised at the beginning of Section 6.5, this section explores the following question: $\mathcal{C}$ satisfying the conclusions of Theorem 6.5.21, can we find a (democratic) model of type theory $\mathcal{M}$ such that $\mathsf{Cx}_{\mathcal{M}} \simeq \mathcal{C}$. In fact, we shall see that such a model always exists, but the proof is not nearly so straightforward as one might hope.

We will find, in particular, a fundamental mismatch of strictness between locally cartesian closed categories and models of type theory. This *coherence problem* sets the stage for our main result: a strictification theorems which papers over the difference and shows that any category $\mathcal{C}$ satisfying these properties can be realized as the category of contexts of a model of type theory. In reality, even more is true: one can set up a (bi)-equivalence of (2-)categories showing that this procedures is a (bi)-inverse to the functor constructored in Theorem 6.5.22 [CD14].

6.6.1 *The coherence problem*

In this subsection, let us fix $\mathcal{C}$ to be a category satisfying the conclusions of Theorem 6.5.21: local cartesian closure, existence of finite coproducts, *etc.* Our goal is to study whether $\mathcal{C}$ can be realized as the category of contexts of some democratic model $\mathcal{M}$ of type theory. Running down the list of requirements of a model, we start with $\mathsf{Cx}_{\mathcal{M}} = \mathcal{C}$, and we see easily that $\mathcal{C}$ has a terminal object (it is locally cartesian closed). We run into trouble, however, with the very next piece of data: what should the presheaf of types $\mathsf{Ty}_{\mathcal{M}}$ be?

Our goal is to "reverse" Theorem 6.5.21 and so we can start by asking a related question. Suppose instead we were given a complete democratic model of type theory $\mathcal{N}$, how can one recover $\mathsf{Ty}_{\mathcal{N}}$ from $\mathsf{Cx}_{\mathcal{N}}$? One plausible approach is suggested by Lemma 6.5.4: since $\mathcal{N}$ is democratic, every substitution $\Delta \longrightarrow \Gamma$ is isomorphic to a weakening substitution $\Gamma.A \longrightarrow \Gamma$ with $A \in \mathsf{Ty}_{\mathcal{N}}(\Gamma)$. Consequently, there is a tight relationship between $\mathsf{Ty}_{\mathcal{N}}(\Gamma)$ and $\mathsf{Cx}_{\mathcal{N}/\Gamma}$ given by sending $A \in \mathsf{Ty}_{\mathcal{N}}(\Gamma)$ to $\mathbf{p} : \Gamma.A \longrightarrow \Gamma$. Accordingly, returning to our original question, we begin to search for a suitable definition of $\mathsf{Ty}_{\mathcal{M}}(\Gamma)$ within the category $\mathcal{C}_{/\Gamma}$.

Care is required, however, as the aforementioned correspondence is not a bijection. In fact, it is neither necessarily injective nor surjective! In a democratic model distinct types can be sent to the same context and there is no guarantee that every morphism $\Delta \longrightarrow \Gamma$ is *equal* to weakening map. What is present is an equivalence of groupoids:

Notation 6.6.1. Write $\mathcal{C}^{\cong}$ for the *groupoid core* of $\mathcal{C}$: the wide subcategory which discards all non-invertible morphisms.

Lemma 6.6.2. *For a democratic model $\mathcal{N}$, there is an equivalence* $\mathsf{Ty}_{\mathcal{N}}(\Gamma)^{\cong} \simeq \mathsf{Cx}_{\mathcal{N}/\Gamma}^{\cong}$.

Proof. The equivalence of categories restricts to an equivalence of groupoids as every functor preserves isomorphisms. □

Exercise 6.23. Show that if $F : \mathcal{C} \longrightarrow \mathcal{D}$ is an equivalence of groupoids, then F induces a bijection of sets $\mathcal{C}/\sim \longrightarrow \mathcal{D}/\sim$ where $C_0 \sim C_1$ if there exists an isomorphism $C_0 \cong C_1$. What does this imply in the case of Lemma 6.6.2?

Exercise 6.24. Show that while $\mathsf{Cx}_{\mathcal{N}}$ does not suffice to recover $\mathsf{Ty}_{\mathcal{N}}$, both of them together fully determine $\mathsf{Tm}_{\mathcal{N}}$. In other words, once $\mathsf{Ty}_{\mathcal{M}}$ is chosen $\mathsf{Tm}_{\mathcal{M}}$ is forced.

Fortunately, this complication is not as major a problem as it might seem. After all, our goal in the section was only to define $\mathcal{M}$ such that $\mathsf{Cx}_{\mathcal{M}} = \mathcal{C}$. We are therefore not overly concerned with whether $\mathsf{Ty}_{\mathcal{M}}$ is determined uniquely, just with whether there is any $\mathsf{Ty}_{\mathcal{M}}$ such that the induced groupoid $\mathsf{Ty}_{\mathcal{M}}(C)^{\cong}$ is $\mathcal{C}_{/C}^{\cong}$. Motivated by this line of thought, we therefore arrive at the following guess for a "functor" $\mathsf{Ty}_{\mathcal{M}}$:

$$\mathsf{Ty}_{\mathcal{M}}(C) = \mathsf{Ob}(\mathcal{C}_{/C}) \qquad (?!)$$

Unfortunately, this definition fails even the most basic test: this is not even a functor! Indeed, while each $f : C \longrightarrow D$ induces a pullback function $f^* : \mathrm{Ob}(\mathcal{C}_{/D}) \longrightarrow \mathrm{Ob}(\mathcal{C}_{/C})$, these are only truly well-defined up to isomorphism. Once we choose particular representatives, we cannot expect that $\mathrm{id}^* = \mathrm{id}$ or that $f^* \circ g^* = (g \circ f)^*$. In fact, it is not guaranteed that such a choice is even possible: Lumsdaine [Lum17] shows that certain subcategories of **Set** can fail to have this property.

Exercise 6.25. Recall the standard explicit description of pullbacks $A \times_C B$ in **Set** as subsets of the cartesian product $A \times B$, convince yourself that the maps $\mathbf{Set}_{/Y} \longrightarrow \mathbf{Set}_{/X}$ induced by this realization of pullbacks are not functorial.

Once more, the situation improves slightly if we consider categories (or even groupoids) rather than just sets: one can show that $\mathcal{C}_{/-} : \mathcal{C}^{\mathrm{op}} \to \mathbf{CAT}$ is a *pseudofunctor*; $f^* \circ g^* \cong (g \circ f)^*$ and these isomorphisms are suitably coherent. The same is true of the restriction of this functor to groupoids $\mathcal{C}_{/-}^{\cong} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Grpd}$.

This mismatch of equality versus coherent isomorphism is commonly referred to as the *coherence problem* for dependent type theory and was famously overlooked by Seely [See84]. Our task is then to find a suitable functor which approximates the merely pseudo-functorial $\mathcal{C}_{/-}^{\cong}$. There are two distinct approaches to this problem:

1. We can capitalize on some special feature of $\mathcal{C}$ which enables us to give a functorial presentation of $\mathcal{C}_{/-}^{\cong}$ to bypass this issue.

2. We can give a much more involved replacement of this pseudofunctor which uses comparatively minimal information about C but then work harder to build the rest of the model with this more intricate definition of $\mathsf{Ty}_{\mathcal{M}}$.

We will focus on the second approach: more complicated replacements for $\mathrm{Ob}(C_{/-})$ which apply with fewer assumptions based on C. We will discuss two such constructions in the following two subsections.

Remark 6.6.3. For completeness, we note an important example of (1). Recall from Section 6.1 that there is a canonical equivalence $\mathbf{Pr}(C_0)_{/X} \simeq \mathbf{Pr}(\int X)$. While we do not prove it, this equivalence is pseudofunctorial in X such that the following diagrams commute up to (coherent) isomorphisms for all $f : X \longrightarrow Y$:

$$\begin{array}{ccc} \mathbf{Pr}(C_0)_{/Y} & \longrightarrow & \mathbf{Pr}(C_0)_{/X} \\ \downarrow & & \downarrow \\ \mathbf{Pr}(\int Y) & \longrightarrow & \mathbf{Pr}(\int X) \end{array}$$

Moreover, the assignment of $X \mapsto \int X$ and $C \mapsto \mathbf{Pr}(C)$ are both functorial and so the following gives a functorial replacement of $\mathrm{Ob}(C_{/-})$ when $C = \mathbf{Pr}(C_0)$:

$$\mathsf{Ty}_{\mathcal{M}}(X) = \mathrm{Ob}(\mathbf{Pr}(\textstyle\int X))$$

This definition is used by Hofmann [Hof97] to give an interpretation of type theory into $\mathbf{Pr}(C_0)$ and we refer the reader there for more information on this model. ◇

6.6.2 *The universe construction*

In this subsection, we present our first and simplest solution to the coherence problem, modulo the additional assumption that our input category C has an additional universe. We refer to this model as $\mathcal{U}(C)$.

The core idea behind the construction of $\mathcal{U}(C)$ is simple enough: we will take the additional universe V in C (Definition 6.5.17) and use it as the basis for a workable approximation of $\mathrm{Ob}(C_{/-})$. More specifically, V must come equipped with a generic map $\pi : E \longrightarrow B$ and we argue that $\mathbf{y}(B)$ is a sufficient definition for $\mathsf{Ty}_{\mathcal{U}(C)}$. We emphasize that this is a necessarily imperfect approximation: $\mathbf{y}(B)(C)$ consists of maps $C \longrightarrow B$ which, by assumption, correspond to V-small families over C. This is only a subset of $\mathrm{Ob}(C_{/C})$, but the raison d'être of universes was that this subset of families was closed under all the operations of type theory so that we could pretend it was complete.

Warning 6.6.4. Strictly speaking this coherence construction does not meet our goals: the model induced on $\mathcal{C}$ is not democratic. By choosing V to be a sufficiently large universe, however, this has little impact in practice.

The astute reader might recognize both this argument and this idea from Section 3.5. Indeed, while we motivated our use of Grothendieck universes purely in terms of size considerations, it was also used to give a definition of presheaves of types and terms. This construction is more-or-less a reprise of the set model construction from earlier but with the salient properties of **Set** now axiomatized. To that end, let us fix a category $\mathcal{C}$ and assume the following properties:

- $\mathcal{C}$ is locally cartesian closed,
- has finite coproducts,
- has a stably initial algebra for $\mathbf{1} \sqcup -$,
- and $\mathcal{C}$ has an $(\omega + 1)$-indexed hierarchy of universes $S_0, \ldots, S_\omega$.

In particular, we assume that $\mathcal{C}$ has an additional universe compared with Theorem 6.5.21 which contains the hierarchy of universes already specified. We will not use this largest universe to interpret $\mathbf{U}_i$ for some universe level i. Instead, this final universe will serve form the basis for our strictly functorial $\mathsf{Ty}_{\mathcal{U}(\mathcal{C})}$:

$$\begin{aligned}
\mathsf{Cx}_{\mathcal{U}(\mathcal{C})} &= \mathcal{C} \\
\mathsf{Sb}_{\mathcal{U}(\mathcal{C})}(\Gamma, \Delta) &= \hom(\Gamma, \Delta) \\
\mathsf{Ty}_{\mathcal{U}(\mathcal{C})}(\Gamma) &= \hom(\Gamma, U_\omega) \\
\mathsf{Tm}_{\mathcal{U}(\mathcal{C})}(\Gamma, A : \Gamma \longrightarrow U_\omega) &= \{a : \Gamma \longrightarrow U_\omega^\bullet \mid \pi_\omega \circ a = A\}
\end{aligned}$$

In other words, we take $\pi_{\mathcal{U}(\mathcal{C})} : \mathsf{Tm}^\bullet_{\mathcal{U}(\mathcal{C})} \longrightarrow \mathsf{Ty}_{\mathcal{U}(\mathcal{C})}$ to be $\mathbf{y}(\tau_\omega) : \mathbf{y}(U_\omega^\bullet) \longrightarrow \mathbf{y}(U_\omega)$. Compare these definitions to Section 3.5 to see how Definition 6.5.17 serves as our replacement for Grothendieck universes.

Exercise 6.26. Show that $\mathbf{y}(\tau_\omega)$ is a representable natural transformation.

What remains is to show that $\pi_{\mathcal{U}(\mathcal{C})}$ is closed under the various connectives. One might fear that this process will be difficult. Fortunately, that difficulty has been shifted into showing that $\mathcal{C}$ has an $(\omega + 1)$-indexed hierarchy of universes. Having assumed this, the requirement that $\pi_{\mathcal{U}(\mathcal{C})}$ is closed under all the connectives of type theory is more-or-less true by definition. In particular, (1) ensures that $\pi_{\mathcal{U}(\mathcal{C})}$ can be equipped with the requisite structure for **Unit**, (2) handles Σ, (3) handles Π, (4) handles **Eq**, and (5) handles $+$, **Bool**, and **Nat**. We will go through the details for Π and **Bool** for completeness.

Lemma 6.6.5. *There exists a pullback square of the following shape in* $\mathbf{Pr}(\mathrm{Cx}_{\mathcal{U}(C)})$*:*

$$\begin{array}{ccc} \mathbf{P}_\pi \mathsf{Tm}^\bullet & \longrightarrow & \mathsf{Ty} \\ \downarrow & & \downarrow \\ \mathbf{P}_\pi \mathsf{Ty} & \longrightarrow & \mathsf{Tm}^\bullet \end{array}$$

Proof. We will construct this pullback square in two steps. First, we will construct the corresponding square in C itself and second we will argue that $\mathbf{y}$ commutes will all the relevant operations and functors involved. Accordingly, since the Yoneda embedding preserves pullback square (along with all other limits) the desired square in $\mathbf{Pr}(\mathrm{Cx}_{\mathcal{U}(C)})$ arises from the C version.

In more detail, recall that $\mathbf{P}_\pi$ was defined as the composite of three functors:

$$\mathbf{Pr}(C) \xrightarrow{(\mathsf{Tm}^\bullet)^*} \mathbf{Pr}(C)_{/\mathsf{Tm}^\bullet} \xrightarrow{\pi_*} \mathbf{Pr}(C)_{/\mathsf{Ty}} \xrightarrow{\mathsf{Ty}_!} \mathbf{Pr}(C)$$

All three of these categories and functors have counterparts in C and the Yoneda embedding then induces the following commutative diagram of functors where each square commutes up to isomorphism:

$$\begin{array}{ccccccc} \mathbf{Pr}(C) & \xrightarrow{(\mathsf{Tm}^\bullet)^*} & \mathbf{Pr}(C)_{/\mathsf{Tm}^\bullet} & \xrightarrow{\pi_*} & \mathbf{Pr}(C)_{/\mathsf{Ty}} & \xrightarrow{\mathsf{Ty}_!} & \mathbf{Pr}(C) \\ \uparrow{\scriptstyle \mathbf{y}} & & \uparrow{\scriptstyle \mathbf{y}} & & \uparrow{\scriptstyle \mathbf{y}} & & \uparrow{\scriptstyle \mathbf{y}} \\ C & \xrightarrow[(U_\omega^\bullet)^*]{} & C_{/U_\omega^\bullet} & \xrightarrow[(\tau_\omega)_*]{} & C_{/U_\omega} & \xrightarrow[(U_\omega)_!]{} & C \end{array}$$

The main thing that must be checked in this diagram is the commutativity of the inner square. This is a consequence of the more general fact that $\mathbf{y} \circ f_* \cong \mathbf{y}(f)_* \circ \mathbf{y}$ whose verification we leave to the reader—it is a slightly more complex version of the argument that the Yoneda embedding preserves exponentials. This shows that $\mathbf{P}_\pi(\mathbf{y}(X)) \cong \mathbf{y}(\mathbf{P}_{\tau_\omega}(X))$ and so we are reduced to constructing the following square in C:

$$\begin{array}{ccc} \mathbf{P}_{\tau_\omega} U_\omega^\bullet & \longrightarrow & U_\omega^\bullet \\ \downarrow & & \downarrow \\ \mathbf{P}_\pi U_\omega & \longrightarrow & U_\omega \end{array}$$

Since τ_ω is generic for S_ω, to construct this pullback square it suffices to show $\mathbf{P}_{\tau_\omega}(\tau_\omega) \in S_\omega$. Examining the definition of $\mathbf{P}_{\tau_\omega}$, we note that all three of the relevant functors preserve elements of S_ω and so the conclusion follows. □

This proof methodology is a useful trick: each of the operations involved in defining various connectives (*e.g.*, those given by Slogans 6.2.10 and 6.3.13) are available in any locally cartesian closed category and are therefore preserved by any locally cartesian closed functor. In particular, the Yoneda embedding commutes with all of these operations and so we can transfer these structures from C to $\mathbf{Pr}(C)$ using $\mathbf{y}$. We go through another example of this with **Bool**. Here we must work slightly harder to rephrase our requirements in the language of locally cartesian closed categories.

Lemma 6.6.6. *There exists a commutative square of the following form in* $\mathbf{Pr}(C)$*:*

$$\begin{array}{ccc} \mathbf{1} \sqcup \mathbf{1} & \longrightarrow & \mathsf{Tm}^\bullet \\ \downarrow & & \downarrow \\ \mathbf{1} & \xrightarrow[\mathbf{Bool}_{\mathcal{U}(C)}]{} & \mathsf{Ty} \end{array}$$

Moreover, the gap map $g : \mathbf{1} \sqcup \mathbf{1} \longrightarrow \mathsf{Tm}^\bullet \times_{\mathsf{Ty}} \mathbf{1} = \mathbf{Bool}^*_{\mathcal{U}(C)}\pi$ *is left orthogonal to* π*.*

Proof. For the sake of readability, let us write $B = \mathbf{Bool}^*_{\mathcal{U}(C)}\pi$—the presheaf of boolean terms in $\mathcal{U}(C)$. Recall that $g \pitchfork \pi$ is equivalent to requiring that the following canonical map is an isomorphism:

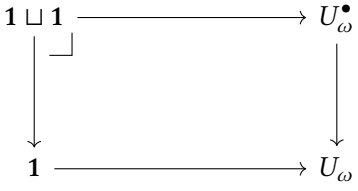
$$(\mathsf{Tm}^\bullet)^B \longrightarrow (\mathsf{Tm}^\bullet)^{\mathbf{1} \coprod \mathbf{1}} \times_{\mathsf{Ty}^{\mathbf{1} \coprod \mathbf{1}}} \mathsf{Ty}^B$$

As it stands, neither this requirement nor the commuting diagram above are formulated in the language of locally cartesian closed categories as both mention a coproduct: $\mathbf{1} \sqcup \mathbf{1}$. In particular, even if we formulate such a square in C, it is not automatic that it will be preserved by the Yoneda embedding. Fortunately, we can replace all occurrences of $\sqcup$ with appropriate products as $\mathbf{1} \sqcup \mathbf{1}$ is used only in the domains of various functions.

We may reformulate our goal as constructing (1) a map $\mathbf{Bool}_{\mathcal{U}(C)} : \mathbf{1} \longrightarrow \mathsf{Ty}$ and (2) a pair of maps $\mathbf{true}_{\mathcal{U}(C)}, \mathbf{false}_{\mathcal{U}(C)} : \mathbf{1} \longrightarrow \mathbf{Bool}^*_{\mathcal{U}(C)}\mathsf{Tm}^\bullet$ such that the following canonical map is an isomorphism:

$$(\mathsf{Tm}^\bullet)^B \longrightarrow (\mathsf{Tm}^\bullet \times \mathsf{Tm}^\bullet) \times_{\mathsf{Ty} \times \mathsf{Ty}} \mathsf{Ty}^B$$

This can then be recast into C. By assumption, S_ω is closed under isomorphisms and coproducts and so we obtain a pullback square of the following shape:



From this, this contains the required maps and the induced gap map is invertible by construction. □

We may stitch these two lemmas, along with other similar arguments, together to conclude the following:

Theorem 6.6.7. *C supports a model of type theory $\mathcal{U}(C)$ with all connectives except universes.*

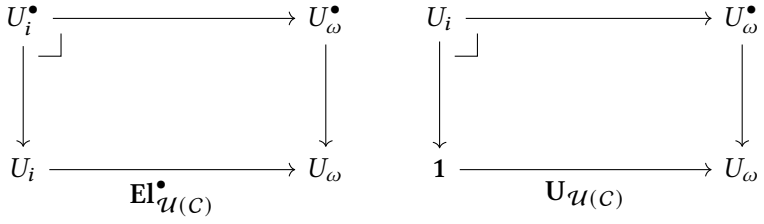
The poor behavior of universe hierarchies Unfortunately, the story around universes is not nearly so simple. While a version of a universe hierarchy may be interpreted into this model, it will not satisfy cumulativity nor any of the other definitional equalities imposed on codes in Section 6.4. For example, neither the equations $\mathbf{lift}(\mathbf{pi}(c_0, c_1)) = \mathbf{pi}(\mathbf{lift}(c_0), \mathbf{lift}(c_1))$ nor $\mathbf{El}(\mathbf{pi}(c_0, c_1)) = \mathbf{\Pi}(\mathbf{El}(c_0), \mathbf{El}(c_1))$ will automatically hold. The latter can be replaced with an isomorphism *i.e.*, there is a pair of mutually inverse functions between these types in the model (Remark 2.6.6). Unfortunately, the first class of equations requiring that **lift** commute with various codes may simply fail to hold.

Current solutions to this problem are sufficiently technical that we will avoid discussing them in full [GSS26]. We will content ourselves with discussing how one interprets *non-cumulative* universes where the equation $\mathbf{El}(\mathbf{pi}(c_0, c_1)) = \mathbf{\Pi}(\mathbf{El}(c_0), \mathbf{El}(c_1))$ is weakened to an isomorphism (*i.e.*, the universe is *weakly* closed under $\mathbf{\Pi}$-types).

Lemma 6.6.8. *For every i, $\mathcal{U}(C)$ supports a universe which is* weakly *closed under all connectives (*$\mathbf{\Pi}$, Σ, **Bool**, $\mathbf{U}_j$ *where $j < i$,* etc.*).*

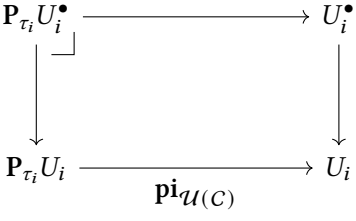
Proof. We have assumed that C has a hierarchy of universes $S_0 \subseteq \cdots \subseteq S_\omega$. The codomain of the generic map $\tau_i : U_i^\bullet \longrightarrow U_i$ of S_i will serve as our interpretation of the universe. In particular, since we have assumed this is a hierarchy of universes, we

have a pair of pullback squares:



The bottom map $\mathbf{U}_{\mathcal{U}(C)}$ is the required element of $\mathsf{Ty}_{\mathcal{U}(C)}(\mathbf{1}_{\mathcal{U}(C)})$ and map $\mathbf{El}_{\mathcal{U}(C)}$ is the required element of $\mathsf{Ty}_{\mathcal{U}(C)}(\mathbf{1}_{\mathcal{U}(C)} \cdot_{\mathcal{U}(C)} \mathbf{U}_{\mathcal{U}(C)})$.

We detail the process of weakly closing $\mathbf{U}_{\mathcal{U}(C)}$ under $\mathbf{\Pi}$-types, as the remaining cases are similar. We note that we have a pullback square of the following shape:



Finally, by pasting pullbacks we see that the pullback of τ_ω along $\mathbf{El}_{\mathcal{U}(C)} \circ \mathbf{pi}_{\mathcal{U}(C)}$ and along $\mathbf{\Pi}_{\mathcal{U}(C)} \circ \mathbf{P}_{\mathbf{El}_{\mathcal{U}(C)}} \mathbf{El}^\bullet_{\mathcal{U}(C)}$ are both canonically isomorphic to $\mathbf{P}_{\tau_i} U_i^\bullet$. This yields the desired isomorphism $\mathbf{\Pi}(\mathbf{El}(c_0), \mathbf{El}(c_1)) \cong \mathbf{El}(\mathbf{pi}(c_0, c_1))$. □

Lemma 6.6.9. *For every pair i, the universe model $\mathcal{U}(C)$ supports an interpretation of $\mathbf{lift}_i$ but it may not validate any of the equations commuting $\mathbf{lift}_i$ past codes.*

Proof. By assumption, there is a pullback square of the following shape:

After applying the Yoneda embedding, this is precisely the data required of the interpretation of $\mathbf{lift}_i$. □

6.6.3 *Presheaf models of type theory*

While not strictly speaking necessary, it is too tempting to not go through the construction of a hierarchy of universes in $\mathbf{Pr}(C)$ due to Hofmann and Streicher [HS97]. In light of the previous subsection, this construction also yields a model of type theory in arbitrary presheaf topoi. The goal of this section is to prove the following:

Theorem 6.6.10. *If V is a Grothendieck universe (Definition 3.5.1) and C is a V-small category, then the following set of morphisms in $\mathbf{Pr}(C)$ forms a universe:*

$$S = \{f : X \longrightarrow Y \mid \forall C : \mathcal{C}, y \in Y(C).\ f^{-1}(y) \text{ is } V\text{-small}\}$$

We say that f is fiberwise V*-small.*

Exercise 6.27. Show that S is a bare universe *i.e.* that fiberwise small morphisms are stable under pullback.

The heart of Theorem 6.6.10 is to construct the generic map for S, so we will begin by showing that S satisfies (1–5) of Definition 6.5.17.

Lemma 6.6.11. *S contains all isomorphisms, is closed under composition, pushforward, diagonals, coproducts, and contains a stably initial $(1 \sqcup -)$-algebra.*

Proof. All of these calculations are of a similar flavor. For instance, to show that S is stable under composition, it suffices to show that if $f : X \longrightarrow Y$ and $g : Y \longrightarrow Z$ are both fiberwise V-small then so too is $g \circ f$. Fix $z \in Z(C)$ for some C such that it now suffices to argue that $X = (g \circ f)^{-1}(z)$ is V-small. We may decompose X into the disjoint union $\sum_{x_0 \in g^{-1}(z)} f^{-1}(x_0)$. The conclusion then follows by assumption together with the observation that V-small sets are closed under V-small indexed disjoint unions.

Note that for dependent products, matters are slightly complicated by the fact that if $C : \mathcal{C}$ and $z : Z(C)$ then $(g_* f)^{-1}(z)$ is realized as follows:

$$(g_* f)^{-1}(z) = \prod_{c : C' \to C} \prod_{y \in g^{-1}(z)} f^{-1}(y)$$

Here we must use the fact that $\mathcal{C}$ is V-small to ensure that $\prod_{c : C' \to C}$ is not too large. □

Corollary 6.6.12. *Given a hierarchy of Grothendieck universes $V_0 \in V_1 \ldots$, the presheaf category $\mathbf{Pr}(C)$ satisfies the conclusion of Theorem 6.5.21. If there is an additional Grothendieck universe V_ω containing all V_i, then $\mathcal{U}(\mathbf{Pr}(C))$ is a model of type theory closed under all connectives.*

Proof of Theorem 6.6.10. It remains only to show that S has a generic family. Let us write $\mathbf{Pr}_V(\mathcal{D})$ for the full subcategory of $\mathbf{Pr}(\mathcal{D})$ spanned by those objects $X : \mathbf{Pr}(\mathcal{D})$

such that $X(D) \in V$ for every $D : \mathcal{D}$. It is important here that we have required that $X(D)$ is literally a member of V, rather than merely being V small as it ensures that $\mathbf{Pr}_V(\mathcal{D})$ is a small category whenever $\mathcal{D}$ is small. We then consider the following presheaf:

$$B(C) = \mathrm{Ob}(\mathbf{Pr}_V(\mathcal{C}_{/C}))$$

B is strictly functorial: we send $f : C_0 \longrightarrow C_1$ to the action on objects associated with the functor $F^* : \mathbf{Pr}(\mathcal{C}_{/C_1}) \longrightarrow \mathbf{Pr}(\mathcal{C}_{/C_0})$ where $F = f_! : \mathcal{C}_{/C_0} \longrightarrow \mathcal{C}_{/C_1}$. This presheaf will serve as the base of our generic family. The total family is given as follows:

$$E : \mathbf{Pr}(\mathcal{C})$$
$$E(C) = \textstyle\sum_{X : \mathrm{Ob}(\mathbf{Pr}_V(\mathcal{C}_{/C}))} X(C, \mathrm{id})$$

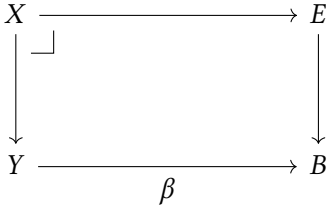
$$\pi : E \to B$$
$$\pi_C(X, _) = X$$

As an aside, both π and E can be specified as a presheaf over $\int_{\mathcal{C}} B$ (Theorem 6.1.7):

$$\tilde{E}(C, X) = X(C, \mathrm{id})$$

Note that $\pi \in S$ as $\pi^{-1}(C, X) = X^{-1}(C, \mathrm{id})$ is V-small because $X : \mathbf{Pr}_V(\mathcal{C}_{/C})$.

Fix $f : X \longrightarrow Y \in S$. It remains to show that there exists a pullback of the following shape:



Since f is fiberwise V-small, for each $C : \mathcal{C}$ and $y \in Y(C)$ there exists an element $v \in V$ such that $v \cong f^{-1}(y)$. Using the axiom of choice, we assemble these into a function $\tilde{f}$ and we define a natural transformation $\beta : Y \longrightarrow B$ as follows:

$$\beta_C(y) = \lambda c : C' \to C.\, \tilde{f}(C', y \cdot c)$$

We leave it to the reader to verify that this is indeed natural. Moreover, if $C : \mathcal{C}$ then $(Y \times_B E)(C)$ is then equivalent to $\sum_{y : Y(c)} \tilde{f}(C, y)$ which, in turn, is equivalent to $\sum_{y : Y(C)} f^{-1}(y)$. It follows that β then fits into the required pullback diagram. □

Remark 6.6.13. Hofmann-Streicher universes share many characteristics with the Grothendieck universes used in their construction. In particular, just as was done in Section 3.5, one can arrange for strictly cumulative universe hierarchies by carefully choosing the interpretations of codes in these universes. We again refer the reader to Gratzer, Shulman, and Sterling [GSS26] for discussion on this point. ◇

6.6.4 *The local universes construction*

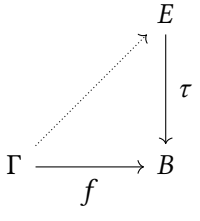
We now turn to the coherence construction introduced by Lumsdaine and Warren [LW15] and Awodey [Awo18]. For the sake of expediency, we present only a special case of this construction and refer the reader to Awodey [Awo18] and Shulman [Shu19, Appendix A] for more thorough treatments which deal with *e.g.*, intensional type theory and the non-democratic models one frequently encounters in the semantics of homotopy type theory.

As in Section 6.6.2, let us fix a locally cartesian closed category $\mathcal{C}$ equipped with coproducts, a stably initial $(1 \sqcup -)$-algebra, a hierarchy of universes, *etc.* Unlike previously, however, we do not insist on a top universe U_ω and instead we work a bit harder to define $\mathsf{Ty}_{\mathcal{L}(\mathcal{C})}$ and $\mathsf{Tm}_{\mathcal{L}(\mathcal{C})}$.

The key idea of the *local universes construction* is to compensate for the lack of U_ω by choosing $\pi_{\mathcal{L}(\mathcal{C})}$ to be the sum of all possible choices; no single choice of universe will necessarily suffice for every situation, but we shall show that in every situation there is at least one suitable universe:

$$\pi_{\mathcal{L}(\mathcal{C})} = \textstyle\sum_{\tau : E \to B} \mathsf{y}(\tau) : \mathsf{Tm}_{\mathcal{L}(\mathcal{C})} \longrightarrow \mathsf{Ty}_{\mathcal{L}(\mathcal{C})}$$

Explicitly, a type $A \in \mathsf{Ty}_{\mathcal{L}(\mathcal{C})}(\Gamma)$ is a pair of (1) a 'local universe' $\tau : E \longrightarrow B$ and (2) a 'type in this universe' $f : \Gamma \longrightarrow B$. A term of type A in context Γ then consists of a section of A:



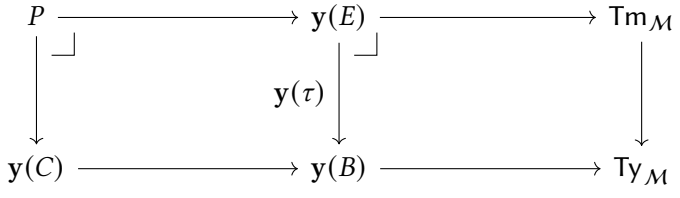
Notation 6.6.14. We have deliberately chosen to use E, B rather than Δ, Γ for the (co)domain of a local universe in an attempt to disambiguate between morphisms in $\mathcal{C}$ qua universes versus morphisms qua substitutions.

If we imagine that there is a single master universe then this definition collapses to that of Section 6.6.2, but this definition allows the universe of types to change between types. Before giving further intuition, we note that π is a representable natural transformation.

Lemma 6.6.15. *$\pi_{\mathcal{L}(\mathcal{C})}$ is a representable natural transformation.*

Proof. Consider the following pullback diagram:

By the Yoneda lemma, $\hom(\mathbf{y}(\Gamma), \sum_{\tau:E\to B} \mathbf{y}(B))$ is equivalent to $\sum_{\tau:E\to B} \hom(\mathbf{y}(\Gamma), \mathbf{y}(B))$, so we may factor the above diagram into two pullback squares for some $\tau : E \longrightarrow B$:



In particular, $P \cong \mathbf{y}(E \times_B X)$ and so π is representable. □

Let us suppose Γ is a context in this nascent model (an object of $\mathcal{C}$) and $A = (\tau : E \longrightarrow B, f : \Gamma \longrightarrow B) \in \mathsf{Ty}_{\mathcal{L}(\mathcal{C})}(C)$. Unfolding the above proof, we see that $\Gamma.A$ is given by $B \times_E \Gamma$ and the $\mathbf{p}$ is the projection $B \times_E \Gamma \longrightarrow \Gamma$. Consequently, there are many distinct $A = (\tau, f)$ which give rise to isomorphic maps $B \times_E \Gamma \longrightarrow \Gamma$ and therefore many distinct types $A, B \in \mathsf{Ty}_{\mathcal{L}(\mathcal{C})}(\Gamma)$ such that $\Gamma.A \cong \Gamma.B$.

Our earlier observation was that the groupoid $\mathsf{Ty}_{\mathcal{L}(\mathcal{C})}(\Gamma)$ ought to be equivalent to $\mathcal{C}^{\cong}_{/\Gamma}$, but this redundancy tells us that $\mathsf{Ty}_{\mathcal{L}(\mathcal{C})}$ as a set is very far from being in bijection with $\mathrm{Ob}(\mathcal{C}_{/C})$. This is vital: the many distinct representations of a given type is what ensures that $\mathsf{Ty}_{\mathcal{L}(\mathcal{C})}$ is strictly functorial.

For instance, we can construct two types which give rise to the same extended context by taking a type A realized by (τ, f) in context Γ and a substitution $\gamma : \Gamma_0 \longrightarrow \Gamma$. The type $A[\gamma] = (\tau, f \circ \gamma)$ induces an isomorphic context to the distinct type $(f^*\tau, \gamma)$. Intuitively, the local universes model 'delays' implementing substitution by pullback to ensure functoriality at the cost of many redundant representations of each types. Fortunately, this duplication does not really impact the construction. All that matters is that every such family $f \in \mathcal{C}^{\cong}_{/C}$ can be realized by at least one type (say, (f, id)) and that $\mathsf{Ty}_{\mathcal{L}(\mathcal{C})}$ is strictly functorial.

Exercise 6.28. Show that $\pi_{\mathcal{L}(\mathcal{C})}$ is democratic (Definition 6.5.1).

Closure under type connectives The heart of the local universes construction is to close $\pi_{\mathcal{L}(C)}$ under the operations of type theory. This is more difficult than Section 6.6.2 because we must describe how to form a **Π**-type when, for instance, the two types are drawn from separate universes. Many of these arguments are formally similar and so we shall detail only three connectives: **Unit**, **Bool**, and **Π**. We refer the reader to Awodey [Awo18] or Lumsdaine and Warren [LW15] for other basic types and to Appendix A of Shulman [Shu19] for universes.[3]

Lemma 6.6.16. *$\pi_{\mathcal{L}(C)}$ supports unit types* i.e.*, there exists a pullback square of the following shape:*

$$\begin{array}{ccc} \mathbf{1} & \longrightarrow & \mathsf{Tm}_{\mathcal{L}(C)} \\ \downarrow & & \downarrow \\ \mathbf{1} & \xrightarrow[\mathbf{Unit}_{\mathcal{L}(C)}]{} & \mathsf{Ty}_{\mathcal{L}(C)} \end{array}$$

Proof. A map $\mathbf{Unit}_{\mathcal{L}(C)} : \mathbf{1} \longrightarrow \mathsf{Ty}_{\mathcal{L}(C)}$ consists of a local universe $\tau : E \longrightarrow B$ and a map $f : \mathbf{1} \longrightarrow B$. We take $\tau = \mathsf{id} : \mathbf{1} \longrightarrow \mathbf{1}$ and $f = \mathsf{id} : \mathbf{1} \longrightarrow \mathbf{1}$. By our earlier discussion, we know that the pullback $\mathbf{Unit}^*_{\mathcal{L}(C)}\mathsf{Tm}_{\mathcal{M}}$—the extension of the empty context by $\mathbf{Unit}_{\mathcal{L}(C)}$—is given by $E \times_B \mathbf{1}$ *i.e.* $\mathbf{1}$ as required. □

Lemma 6.6.17. *$\pi_{\mathcal{L}(C)}$ supports booleans* i.e.*, there exists a square of the following shape whose gap map is orthogonal to $\pi_{\mathcal{L}(C)}$:*

$$\begin{array}{ccc} \mathbf{1} \sqcup \mathbf{1} & \longrightarrow & \mathsf{Tm}_{\mathcal{L}(C)} \\ \downarrow & & \downarrow \\ \mathbf{1} & \xrightarrow[\mathbf{Bool}_{\mathcal{L}(C)}]{} & \mathsf{Ty}_{\mathcal{L}(C)} \end{array}$$

Proof. We start by defining $\mathbf{Bool}_{\mathcal{L}(C)} : \mathbf{1} \longrightarrow \mathsf{Ty}_{\mathcal{L}(C)}$ as the local universe $\mathbf{1} \sqcup \mathbf{1} \longrightarrow \mathbf{1}$ together with type id. Direct calculation then shows that the pullback $\mathbf{Bool}^*_{\mathcal{L}(C)}\mathsf{Tm}_{\mathcal{L}(C)}$ is given by $\mathbf{y}(\mathbf{1} \sqcup \mathbf{1})$. It then suffices to show that $\mathbf{1} \sqcup \mathbf{1} \longrightarrow \mathbf{y}(\mathbf{1} \sqcup \mathbf{1})$ is orthogonal to π. This follows from the representability of $\pi_{\mathcal{L}(C)}$ and we leave this calculation to the reader. □

[3] Just as with the universes construction, the universes obtained in this manner satisfy fewer equations than the theory described Chapter 2.

Lemma 6.6.18. *$\pi_{\mathcal{L}(C)}$ is closed under $\mathbf{\Pi}$* i.e.*, there exists a pullback square of the following shape:*

$$\begin{array}{ccc} \mathbf{P}_{\pi_{\mathcal{L}(C)}}(\mathsf{Tm}_{\mathcal{L}(C)}) & \longrightarrow & \mathsf{Tm}_{\mathcal{L}(C)} \\ \downarrow & & \downarrow \\ \mathbf{P}_{\pi_{\mathcal{L}(C)}}(\mathsf{Ty}_{\mathcal{L}(C)}) & \xrightarrow[\mathbf{\Pi}_{\mathcal{L}(C)}]{} & \mathsf{Ty}_{\mathcal{L}(C)} \end{array}$$

Proof. We begin by defining $\mathbf{\Pi}_{\mathcal{L}(C)}$. The input to $\mathbf{\Pi}_{\mathcal{L}(C)}$ consists of the following:

- a context $\Gamma : C$,
- a type $A \in \mathsf{Ty}_{\mathcal{L}(C)}(\Gamma)$ given by a local universe $\tau_A : E_A \longrightarrow B_A$ and a map $f_A : \Gamma \longrightarrow B_A$,
- a type $B \in \mathsf{Ty}_{\mathcal{L}(C)}(\Gamma.A)$ given by a local universe $\tau_B : E_B \longrightarrow B_B$ and a map $f_B : \Gamma \times_{B_A} E_A \longrightarrow B_B$,

We must construct a local universe along with a map into this universe. Just as with the prior two examples, we choose a local universe which suitably 'encodes' the dependent product. Drawing inspiration from Section 6.2, we define $\tau : E \longrightarrow B$ to be

$$\mathbf{P}_{\tau_A}(\tau_B) : \mathbf{P}_{\tau_A}(E_B) \longrightarrow \mathbf{P}_{\tau_A}(B_B)$$

Under this definition, a map $C \longrightarrow B$ consists of (1) a map $C \longrightarrow B_A$ along with (2) a map $E_A \times_{B_A} C \longrightarrow B_B$. We therefore obtain the required map $f : \Gamma \longrightarrow B$ precisely from (f_A, f_B). The reader may verify directly that $\mathbf{\Pi}_{\mathcal{L}(C)}((\tau_A, f_A), (\tau_B, f_B)) = (\tau, f)$ assembles into the required natural transformation.

It remains to show that $\mathbf{\Pi}_{\mathcal{L}(C)}$ fits into the desired pullback square. We begin by calculating a term of $\mathbf{\Pi}_{\mathcal{L}(C)}(A, B)$ with A, B as above. Unfolding definitions, a term is a map $t : \Gamma \longrightarrow \mathbf{P}_{\tau_A}(E_B)$ fitting into the appropriate commuting triangle:

$$\begin{array}{ccc} & & \mathbf{P}_{\tau_A}(E_B) \\ & \nearrow & \downarrow \mathbf{P}_{\tau_A}(\tau_B) \\ \Gamma & \xrightarrow[(f_A, f_B)]{} & \mathbf{P}_{\tau_A}(B_B) \end{array}$$

By universal properties of $\mathbf{P}_{\tau_A}(E_B)$ and $\mathbf{P}_{\tau_A}(\tau_B)$, t corresponds to (1) a map $t_0 : \Gamma \longrightarrow B_A$ and (2) a map $t_1 : E_A \times_{B_A} \Gamma \longrightarrow E_B$. The commuting triangle above forces $\Gamma \longrightarrow B_A$ to be f_A and further ensures that $\tau_B \circ t_1 = f_B$. In other words, an element of $\mathbf{\Pi}_{\mathcal{L}(C)}(A, B)$ is precisely determined by an element of B in the context $\Gamma.{}_{\mathcal{L}(C)}A$. The reader may check that this equivalence is natural in order to obtain the required pullback square. □

The final result One can proceed as we have done in Lemmas 6.6.16 to 6.6.18 to show that the model based on local universes is closed under all the connectives of type theory (sans universes). With further effort, one can also account universes [Shu19, Appendix A] to some extent in this theory, though as of writing this construction is not known to support *cumulative* universes.

Putting these pieces together, one arrives at the following result:

Theorem 6.6.19. *If $\mathcal{C}$ satisfies the conclusion of Theorem 6.5.21 then $\mathcal{L}(\mathcal{C})$ is a democratic model of type theory with all connectives—though whose universe hierarchy is not cumulative—whose category of contexts is precisely $\mathcal{C}$.*

6.6.5 *Strictification constructions in action*

To close out this section, let us list a few potential applications of Theorem 6.6.19 and, more generally, the connection it implies between locally cartesian closed categories and type theory.

Broadly, there are two classes of applications:

1. We can now use locally cartesian closed categories to construct exotic models of type theory
2. We can now use type theory to reason about exotic locally cartesian closed categories.

We content ourselves with only a few examples in the literature of each, as these two classes of applications contain a large swathe of modern type theory.

For the first application, a number of independence results are now readily available and, in particular, we may use Theorem 6.6.19 with various topoi to deliver on some of the independence results promised in Section 2.7. One may use the model of type theory in $\mathbf{Pr}(\{0 \leq 1\}) = \mathbf{Set}^{\rightarrow}$ to show the independence of both the law of the excluded middle and the axiom of choice from ETT. Exchanging presheaf topoi for sheaf topoi, one

can falsify Markov's principle [CM16][4] and various other constructive taboos. Using instead various realizability topoi [vOos08], one can show the consistency of Church's thesis with extensional type theory. More recently, Andrew Swan has announced a proof that not all quotient types are definable in ETT using similar methods [Swa25].

In the second direction, one may use the model of extensional type theory available in $\mathbf{Pr}(C)$ to give a succinct account of all of the structures defined in Sections 6.1 to 6.4. In particular, the interpretation of dependent products in $\mathbf{Pr}(C)$ yields a semantic version of *higher-order abstract syntax* [Hof99] and this maneuver is already present in Awodey [Awo18]. More strikingly, the same model of type theory in cubical sets can be used to succinctly *construct* a model of cubical type theory [OP16]. The same approach applied to categories arising from Artin gluing may be used to give conceptual arguments for the normalization of various type theories [SA21; Ste21; Gra22].

This is a very random assortment of references. Try and systematize this (even when limited to working with extensional type theory which pars down the list quite a lot).

6.7 *Canonicity via gluing*

In Section 3.4, we discussed how various metatheorems of type theory can be reduced to questions about models of dependent type theory. In this section, we follow one such reduction to deliver on a result promised earlier. We will construct a particular model of extensional type theory and from it conclude that extensional type theory satisfies *canonicity* (Definition 3.4.9). In addition to showing the validity of a crucial property, this argument exemplifies the style of *gluing* argument which has become widely used technique in dependent type theory.

The basic outline for all arguments of this style breaks down into three steps:

1. We construct a particular model $\mathcal{G}$.
2. We exhibit a morphism of models $\pi : \mathcal{G} \longrightarrow \mathcal{T}$ from this freshly constructed model to the syntactic model.
3. Using the initiality of syntax, we conclude that π has a section $i : \mathcal{T} \longrightarrow \mathcal{G}$ where initiality tells in particular that $\pi \circ i = \mathsf{id}$.

[4] Coquand and Mannaa opt for a more elaborate approach to deal with the relatively poor behavior (particularly in constructive metatheory) of hierarchies of universes which we have largely ignored in this section. See Gratzer, Shulman, and Sterling [GSS26] for more discussion on this point.

connect this to discussions of displayed algebras in Sections 2.5.4 and 5.2.3 and Appendix B.2

The goal is to arrange the first and second steps in such a way that a section to $\mathcal{G} \longrightarrow \mathcal{T}$ yields the desired theorem. For instance, in this section we wish to prove canonicity at **Bool**. Accordingly, we will arrange matters such that $\mathsf{Tm}_{\mathcal{G}}(\mathbf{1}_{\mathcal{G}}, \mathbf{Bool}_{\mathcal{G}}) \cong \{0, 1\}$ and the map $\pi_{\mathsf{Tm}_{\mathcal{G}}(\mathbf{1}_{\mathcal{G}}, \mathbf{Bool}_{\mathcal{G}})}$ sends 0 to **true** and 1 to **false**. With these two assumptions to hand, it is only a short argument to conclude canonicity. Indeed, we note that if $b \in \mathsf{Tm}_{\mathcal{T}}(\mathbf{1}, \mathbf{Bool})$ the equation $\pi \circ i = \mathrm{id}$ ensures the following:

$$b = \pi_{\mathsf{Tm}_{\mathcal{G}}(\mathbf{1}_{\mathcal{G}}, \mathbf{Bool}_{\mathcal{G}})}(i_{\mathsf{Tm}_{\mathcal{T}}(\mathbf{1}, \mathbf{Bool})}(b))$$

By assumption, the image of $\pi_{\mathsf{Tm}_{\mathcal{G}}(\mathbf{1}_{\mathcal{G}}, \mathbf{Bool}_{\mathcal{G}})}$ is $\{\mathbf{true}, \mathbf{false}\}$ and so we know that $b =$ **true** or $b =$ **false**, as required.

Warning 6.7.1. The reader will potentially wonder to which model the un-subscripted **true** and **false** above belong. In accordance with our convention (implicitly dating back to Chapter 2), we write elements of the syntactic model $\mathcal{T}$ without a subscript, so $\mathbf{true} \in \mathsf{Tm}_{\mathcal{T}}(\mathbf{1}, \mathbf{Bool})$. The constructions of $\mathcal{G}$ will always be subscripted as such.

The heart of the argument is therefore contained in the first two steps. In fact, they often happen somewhat simultaneously—one usually constructs the model $\mathcal{G}$ in such a way that the definition of π is immediate.

The canonicity model This leads us to the next question: where does $\mathcal{G}$ come from? We know what this model must provide: we must have $\mathsf{Tm}_{\mathcal{G}}(\mathbf{1}_{\mathcal{G}}, \mathbf{Bool}_{\mathcal{G}}) \cong \{0, 1\}$. With just this constraint, however, it is far from obvious what the rest of such a model ought to be. We have met one model with $\mathsf{Tm}_{\mathcal{G}}(\mathbf{1}_{\mathcal{G}}, \mathbf{Bool}_{\mathcal{G}}) \cong \{0, 1\}$: the set model $\mathcal{S}$ (Section 3.5). However, there is no suitable morphism $\pi : \mathcal{S} \longrightarrow \mathcal{T}$, so we cannot simply take the set model off-the-shelf. We have something of the opposite problem with syntactic model $\mathcal{T}$, where it is easy to obtain a morphism π but where we cannot directly establish $\mathsf{Tm}_{\mathcal{T}}(\mathbf{1}, \mathbf{Bool}) \cong \{0, 1\}$.

In fact, $\mathcal{G}$ will be, in some formal sense, a mixing of both $\mathcal{T}$ and $\mathcal{S}$ (see Section 6.7.4). Each sort in $\mathcal{G}$ will be interpreted as a Σ type pairing an element X° of the appropriate sort from $\mathcal{T}$ with some additional data $X^\bullet$ specific to proving canonicity. Projecting out the first component X° will then give rise to the homomorphism $\mathcal{G} \longrightarrow \mathcal{T}$. The complexity of dependent type theory means that the additional data $X^\bullet$ is hard to summarize concisely, but its role is to ensure that an element of $\mathsf{Tm}_{\mathcal{G}}(\mathbf{1}_{\mathcal{G}}, \mathbf{Bool}_{\mathcal{G}})$ will consist of a closed boolean term b° along with a proof that either $b^\circ =$ **true** or $b^\circ =$ **false**. The data contained in $X^\bullet$ becomes more complex to account for open terms and terms of other types, but this complexity is only what is required to show that every closed boolean can be appropriately equipped with canonicity data.

Gluing (specifically, *Artin gluing* [AGV72]) enters the picture as a conceptual way to structure all of this additional data. It is a categorical construction which pastes together two categories along a functor and, in this case, we shall find we can glue together the syntactic model and the set model appropriately to obtain the model. While geometrical considerations motivated the original construction, its application to proving metatheorems can be traced to Freyd [Fre78]. Since then, the methodology has been applied and adapted to a wide variety of different systems and metatheorems.

We begin with the definition of Artin gluing in its most general form:

Definition 6.7.2. If $\mathcal{C}$ and $\mathcal{D}$ are categories and $F : \mathcal{C} \longrightarrow \mathcal{D}$ then the gluing category $\mathbf{Gl}(F)$ is defined such that:

- an object $X : \mathbf{Gl}(F)$ consists of a triple $(D : \mathcal{D}, C : \mathcal{C}, f : D \longrightarrow F(C))$,
- a morphism $g : (D_0, C_0, f_0) \longrightarrow (D_1, C_1, f_1)$ consists of a pair of morphisms $g_0 : D_0 \longrightarrow D_1$ and $g_1 : C_0 \longrightarrow C_1$ fitting into the following commutative diagram:

$$\begin{array}{ccc} D_0 & \xrightarrow{g_0} & D_1 \\ {\scriptstyle f_0}\downarrow & & \downarrow{\scriptstyle f_1} \\ F(C_0) & \xrightarrow[F(g_1)]{} & F(C_1) \end{array}$$

Generally, if $\mathcal{C}$ and $\mathcal{D}$ enjoy some categorical properties (*e.g.*, $\mathcal{C}$ and $\mathcal{D}$ are locally cartesian closed) and F is sufficiently well-behaved (*e.g.*, preserves finite limits) then (1) $\mathbf{Gl}(F)$ will inherit these properties and (2) $\pi_1 : \mathbf{Gl}(F) \longrightarrow \mathcal{C}$ will preserve them. For instance, Freyd [Fre78] capitalize on the fact that $\mathbf{Gl}(F)$ forms an (elementary) topos in certain situations and π_1 is a logical morphism in those cases. See Lambek and Scott [LS88] for a textbook account of this proof together with the argument connecting it to Freyd's result.

We shall be interested in the case where $\mathcal{C}$ and $\mathcal{D}$ are the categories of contexts associated to a pair of models and F preserves "enough" of this structure—in particular, when F is a pseudo-morphism of models [KHS19]. Under these assumptions, we may equip $\mathbf{Gl}(F)$ with the structure of a model of type theory and upgrade the functor $\pi_1 : \mathbf{Gl}(F) \longrightarrow \mathcal{C}$ to a morphism of models. By carefully choosing $\mathcal{C}$, $\mathcal{D}$, and F, we may use this procedure to procure the required model $\mathcal{G}$ and homomorphism π discussed above. While one could construct gluing models at this of generality, we will focus on the more concrete case arising from proving canonicity and return to the general result only after carrying out this proof.

In particular, we shall focus on case where $\mathcal{C}$ is the syntactic model, $\mathcal{D}$ is the set model, and F is the global sections functor $\Gamma_{\mathcal{C}} = \hom(\mathbf{1}, -) : \mathcal{C} \longrightarrow \mathbf{Set}$.[5] In this situation, we may unfold to see that objects of $\mathbf{Gl}(\Gamma)$ are pairs of (1) an object Γ° of Cx along with (2) a family of sets X_γ indexed by $\gamma \in \hom(\mathbf{1}, \Gamma^\circ)$. We shall view these families of sets as *proof-relevant* predicates on global elements of Γ° and often write the collection as $\Gamma^\bullet : \hom(\mathbf{1}, \Gamma^\circ) \to \mathbf{Set}$.

The definition of $\mathrm{Ty}_{\mathcal{G}}$ we build on top of $\mathbf{Gl}(\Gamma)$ and the interpretations of each connective shall ensure that whenever $\gamma^\circ \in \mathrm{Sb}(\mathbf{1}, \Gamma^\circ)$, the elements of $\Gamma^\bullet(\gamma^\circ)$ provide information on how to place each term 'contained' within γ into canonical form. For instance, if $\Gamma^\circ = \Gamma_0^\circ.A$ then each $\gamma^\bullet \in \Gamma^\bullet(\gamma^\circ)$ will (1) describe how to place $\mathbf{q}[\gamma^\circ]$ into canonical form as well as (2) contain an element of $\Gamma^\bullet(\mathbf{p} \circ \gamma^\circ)$.

Substitutions in this nascent model are ordinary syntactic substitutions $\gamma^\circ \in \mathrm{Sb}(\Delta^\circ, \Gamma^\circ)$ together with a function assigning to each pair $\delta^\circ \in \mathrm{Sb}(\mathbf{1}, \Delta^\circ)$ and $\delta^\bullet \in \Delta^\bullet(\delta^\circ)$ an element of $\Gamma^\bullet(\gamma^\circ \circ \delta^\circ)$. In other words, a syntactic substitution together with a procedure ensuring that this substitution preserves canonicity evidence.

Remark 6.7.3. To those familiar with logical relations, we note that models based on gluing are a method of categorically reconstructing this technique. See, for instance, Mitchell and Scedrov [MS93]. ◇

We have thus far been deliberately vague about what canonicity evidence shall mean precisely and this vagueness shall persist for a while longer yet; it will be formally specified only when we close $\mathcal{G}$ under various connectives. We note, however, we shall require more information than just the fact that a term can be placed into canonical form. For instance, for elements of the universe c we shall require not just the canonical form c but also a description of the canonicity evidence associated with $\mathbf{El}(c)$. In particular, accounting for the universe makes it vital for us to consider general proof-relevant data for canonicity rather than merely having a predicate.

Axiom 6.7.4. We shall assume a hierarchy of Grothendieck universes $V_0 \in V_1 \cdots \in V_\omega$ as in Section 3.5.

6.7.1 *The bare cwf structure of the gluing model*

We begin by constructing a model of bare type theory $\mathcal{G}$ expanding on the earlier intuition given above. As mentioned above, we shall take $\mathrm{Cx}_{\mathcal{G}}$ to be $\mathbf{Gl}(\Gamma_{\mathrm{Cx}})$, so it

[5] Note that we have not said "global sections homomorphism", as Γ is not a homomorphism of models. It does not, for instance, preserve dependent products.

remains only to define the empty context, $\mathsf{Ty}_{\mathcal{G}}$, $\mathsf{Tm}_{\mathcal{G}}$, and to define context extension and the attendant structure.

The empty context—the terminal object in $\mathsf{Cx}_{\mathcal{G}}$—may be constructed directly:

$$\mathbf{1}_{\mathcal{G}} = (\mathbf{1}_{\mathcal{T}}, \lambda_ \to \{\star\})$$

Exercise 6.29. Prove that $\mathbf{1}_{\mathcal{G}}$ is actually a terminal object.

We now turn to the construction of terms and types. Prior to doing so, it pays to start fixing some notation to allow definitions to be more compact and readable.

Notation 6.7.5. By convention, we shall superscript variables ranging over elements of $\mathcal{T}$ by $\circ$ and families of proof-relevant predicates or witnesses by $\bullet$. Accordingly, a $\mathcal{G}$-context Γ is a pair $(\Gamma^\circ, \Gamma^\bullet)$. We shall also use overload these superscripts to denote projections; if $\Gamma : \mathsf{Cx}_{\mathcal{G}}$ we write $\Gamma^\bullet$ for $\pi_2(\Gamma)$ and likewise with Γ°.

With all of this notation, we may now define the presheaf of types in $\mathcal{G}$:

$$\mathsf{Ty}_{\mathcal{G}} : \mathbf{Pr}(\mathsf{Cx}_{\mathcal{G}})$$
$$\mathsf{Ty}_{\mathcal{G}}(\Gamma) = \textstyle\sum_{A^\circ \in \mathsf{Ty}(\Gamma^\circ)} \prod_{\gamma^\circ \in \mathsf{Sb}(\mathbf{1}, \Gamma^\circ)} \Gamma^\bullet(\gamma^\circ) \to \mathsf{Tm}(\mathbf{1}, A^\circ[\gamma^\circ]) \to V_\omega$$

Exercise 6.30. We have specified the action of $\mathsf{Ty}_{\mathcal{G}}$ on objects of $\mathsf{Cx}_{\mathcal{G}}$ but not morphisms. Give the functorial action and check that it satisfies the functor laws.

In other words, a $\mathcal{G}$-type consists of a syntactic type A together with a family of proof-relevant predicates. We would like to consider a single proof-relevant predicate on the closed elements of A but this makes no sense in dependent type theory since A itself may be open. Accordingly, we must instead consider the more complex family of proof-relevant predicates indexed by closing substitutions $\gamma^\circ \in \mathsf{Sb}(\mathbf{1}, \Gamma^\circ)$ paired with $\gamma^\bullet \in \Gamma^\bullet(\gamma^\circ)$.

This is our first encounter with an unfortunate truth: the amount of indices and dependence in gluing models can be quite overwhelming (already we have three indices to the predicate). Let us take a moment to compress this definition slightly. If $\Gamma : \mathsf{Cx}_{\mathcal{G}}$ we observe that the dependent sum $\sum_{\gamma^\circ \in \mathsf{Sb}(\mathbf{1}, \Gamma^\circ)} \Gamma^\bullet(\gamma^\circ)$ is equivalent to $\hom(\mathbf{1}_{\mathcal{G}}, \Gamma)$. We shall often suppress this isomorphism, such that $\mathsf{Ty}_{\mathcal{G}}(\Gamma)$ may be written as follows:

$$\textstyle\sum_{A \in \mathsf{Ty}(\Gamma^\circ)} \prod_{\gamma \in \hom(\mathbf{1}_{\mathcal{G}}, \Gamma)} \mathsf{Tm}(\mathbf{1}, A[\gamma^\circ]) \to V_\omega$$

Just as with contexts, we shall use A° and $A^\bullet$ to denote the first and second projections from an $\mathcal{G}$ type and we shall continue to use these superscripts as part

of a variable names on occasion to emphasize that a particular variable ranges over syntactic types or proof-relevant predicates. We now define the presheaf of terms:

$$\begin{aligned}&\mathsf{Tm}_{\mathcal{G}} : \mathbf{Pr}(\textstyle\int_{\mathsf{Cx}_{\mathcal{G}}} \mathsf{Ty}_{\mathcal{G}})\\&\mathsf{Tm}_{\mathcal{G}}(\Gamma, A) = \textstyle\sum_{a^\circ \in \mathsf{Tm}(\Gamma^\circ, A^\circ)} \prod_{\gamma \in \hom(\mathbf{1}_{\mathcal{G}}, \Gamma)} A^\bullet(\gamma, a^\circ[\gamma^\circ])\end{aligned}$$

Finally, context extension is defined (as always) by something akin to a Σ-type. As will become common, the first component of $\Gamma.{}_{\mathcal{G}}A$ is realized by the syntactic version of context extension. The associated proof-relevant predicate is less obvious, but pairs together a witness for predicate $\Gamma^\bullet$ with one for $A^\bullet$ more or less as promised earlier:

$$\begin{aligned}&- .{}_{\mathcal{G}} - : (\Gamma \in \mathsf{Cx}_{\mathcal{G}}) \to \mathsf{Ty}_{\mathcal{G}}(\Gamma) \to \mathsf{Cx}_{\mathcal{G}}\\&\Gamma.{}_{\mathcal{G}}A = (\Gamma^\circ.A^\circ, \lambda\gamma^\circ \to \textstyle\sum_{\gamma_0^\bullet \in \Gamma^\bullet(\mathbf{p} \circ \gamma^\circ)} A^\bullet(\mathbf{p} \circ \gamma^\circ, \gamma_0^\bullet, \mathbf{q}[\gamma]))\end{aligned}$$

While both the syntactic half and family of proof-relevant predicates were relatively short in this case, later as both components become larger it will be convenient to specify them separately. Accordingly, we will frequently specify an $\mathcal{G}$ operation by writing two lines: one defining the syntactic half and one defining the predicate:

$$\begin{aligned}&- .{}_{\mathcal{G}} - : \{\Gamma \in \mathsf{Cx}_{\mathcal{G}}\}\{A \in \mathsf{Ty}_{\mathcal{G}}(\Gamma)\} \to \mathsf{Cx}_{\mathcal{G}}\\&\pi_1(\Gamma.{}_{\mathcal{G}}A) = \Gamma^\circ.A^\circ\\&\pi_2(\Gamma.{}_{\mathcal{G}}A) = \lambda\gamma^\circ \to \textstyle\sum_{\gamma_0^\bullet \in \Gamma^\bullet(\mathbf{p} \circ \gamma^\circ)} A^\bullet(\mathbf{p} \circ \gamma, \gamma_0^\bullet, \mathbf{q}[\gamma])\end{aligned}$$

We now turn to the weakening substitution and variable terms for this operation:

$$\begin{aligned}&\mathbf{p}_{\mathcal{G}} : \{\Gamma \in \mathsf{Cx}_{\mathcal{G}}\}\{A \in \mathsf{Ty}_{\mathcal{G}}(\Gamma)\} \to \mathsf{Sb}_{\mathcal{G}}(\Gamma.{}_{\mathcal{G}}A, \Gamma)\\&\pi_1(\mathbf{p}_{\mathcal{G}}) = \mathbf{p}\\&\pi_2(\mathbf{p}_{\mathcal{G}}) = \lambda(\gamma^\circ, \gamma^\bullet) \to \pi_1(\gamma^\bullet)\\&\mathbf{q}_{\mathcal{G}} : \{\Gamma \in \mathsf{Cx}_{\mathcal{G}}\}\{A \in \mathsf{Ty}_{\mathcal{G}}(\Gamma)\} \to \mathsf{Tm}_{\mathcal{G}}(\Gamma.{}_{\mathcal{G}}A, A[\mathbf{p}_{\mathcal{G}}])\\&\pi_1(\mathbf{q}_{\mathcal{G}}) = \mathbf{q}\\&\pi_2(\mathbf{q}_{\mathcal{G}}) = \lambda(\gamma^\circ, \gamma^\bullet) \to \pi_2(\gamma^\bullet)\end{aligned}$$

Lemma 6.7.6. *$\mathbf{q}_{\mathcal{G}}$ and $\mathbf{p}_{\mathcal{G}}$ induce an isomorphism of the following shape:*

$$\lambda\gamma \to (\mathbf{p}_{\mathcal{G}} \circ \gamma, \mathbf{q}_{\mathcal{G}}[\gamma]) : \mathsf{Sb}_{\mathcal{G}}(\Delta, \Gamma.{}_{\mathcal{G}}A) \cong \textstyle\sum_{\gamma \in \mathsf{Sb}_{\mathcal{G}}(\Delta, \Gamma)} \mathsf{Tm}_{\mathcal{G}}(\Delta, A[\gamma])$$

Proof. We construct an inverse to this operation. Given $(\gamma, a) \in \sum_{\gamma_0 \in \mathsf{Sb}_{\mathcal{G}}(\Delta,\Gamma)} \mathsf{Tm}_{\mathcal{G}}(\Delta, A[\gamma])$, we must construct a unique $\bar{\gamma} \in \mathsf{Sb}_{\mathcal{G}}(\Delta, \Gamma.{}_{\mathcal{G}}A)$ such that $\mathbf{p}_{\mathcal{G}} \circ \bar{\gamma} = \gamma_0$ and $\mathbf{q}_{\mathcal{G}}[\bar{\gamma}] = a$. Let us observe that, by definition, a substitution $\mathsf{Sb}_{\mathcal{G}}(\Delta, \Gamma.{}_{\mathcal{G}}A)$ which lies over γ_0 and a consists of the following data:

1. A (syntactic) substitution $\bar{\gamma}^\circ \in \mathrm{Sb}(\Delta^\circ, \Gamma^\circ.A^\circ)$ such that $\mathbf{p} \circ \bar{\gamma}^\circ = \gamma_0^\circ$ and $\mathbf{q}[\bar{\gamma}^\circ] = a^\circ$,
2. A set-theoretic function of the following type:

$$\bar{\gamma}^\bullet : (\delta \in \hom(\mathbf{1}, \Delta)) \to \textstyle\sum_{\gamma^\bullet \in \Gamma^\bullet(\gamma_0^\circ \circ \delta^\circ)} A^\bullet((\gamma_0^\circ \circ \delta^\circ, \gamma^\bullet), \mathbf{q}[\bar{\gamma}^\circ])$$

   such that $\pi_1 \circ \bar{\gamma}^\bullet = \gamma_0^\bullet$ and $\pi_2 \circ \bar{\gamma}^\bullet = a^\bullet$.

Examining these constraints, we find that $\bar{\gamma}^\circ$ and $\bar{\gamma}^\bullet$ are both uniquely determined (as $\gamma_0^\circ.a^\circ$ and $\lambda\delta \to (\gamma_0^\bullet(\delta), a^\bullet(\delta))$). The conclusion then follows. □

Theorem 6.7.7. *With the above definitions, $\mathcal{G}$ is a model of type theory without connectives.*

Before moving on to close $\mathcal{G}$ under connectives, let us recall that we also must construct a homomorphism of models $\pi : \mathcal{G} \longrightarrow \mathcal{T}$. We already have the functor between categories of contexts defined by projection $\mathsf{Cx}_\mathcal{G} = \mathbf{Gl}(\Gamma) \longrightarrow \mathsf{Cx}$. We may extend this to a morphism of models as follows:

$$\pi_{\mathsf{Ty}_\mathcal{G}(\Gamma)}(A^\circ, A^\bullet) = A^\circ$$
$$\pi_{\mathsf{Tm}_\mathcal{G}(\Gamma,A)}(a^\circ, a^\bullet) = a^\circ$$

We have defined operations like $\mathbf{p}_\mathcal{G}$ and $-.\mathcal{G}-$ so that these operations strictly preserve the operations of bare type theory, so we arrive at the following:

Theorem 6.7.8. *There is a morphism of models $\pi : \mathcal{G} \longrightarrow \mathcal{T}$ given by projecting out the syntactic components of each sort of $\mathcal{G}$.*

As we proceed to close $\mathcal{G}$ under various connectives, we shall also ensure that π extends to a homomorphism of models closed under these connectives (recalling that this is merely a property from now on).

6.7.2 *Closing the gluing model under connectives*

The majority of the work in constructing $\mathcal{G}$—indeed, constructing any model—is showing that it is closed under all the connectives of dependent type theory. We have already encountered this process several times now (Sections 3.5 and 6.6) and, just as with those cases, the process quickly becomes repetitive. Accordingly, we shall focus on a few representative connectives and leave it to the reader to extrapolate the process to the other connectives of dependent type theory. In particular, we shall deal with **Unit**, **Π**, **Eq**, $\mathbf{U}_0$, and (of course), **Bool**. Since the heart of the canonicity theorem centers around **Bool**, let us start with this case.

Lemma 6.7.9. *$\mathcal{G}$ is closed under booleans and π preserves them.*

Proof. Let us recall from Structure 6.3.2 that closing $\mathcal{G}$ under booleans requires providing several pieces of data. First, and most important, we must construct an $\mathbf{Bool}_{\mathcal{G}} : \{\Gamma \in \mathsf{Cx}_{\mathcal{G}}\} \to \mathsf{Ty}_{\mathcal{G}}(\Gamma)$:

$$\begin{aligned}
&\mathbf{Bool}_{\mathcal{G}} : \{\Gamma \in \mathsf{Cx}_{\mathcal{G}}\} \to \mathsf{Ty}_{\mathcal{G}}(\Gamma)\\
&\pi_1(\mathbf{Bool}_{\mathcal{G}}) = \mathbf{Bool}\\
&\pi_2(\mathbf{Bool}_{\mathcal{G}}) = \lambda\gamma\, b \to \{0 \mid b = \mathbf{true}\} \cup \{1 \mid b = \mathbf{false}\}
\end{aligned}$$

We must check that $\mathbf{Bool}_{\mathcal{G}}$ is stable under substitution. That is, if $\gamma \in \mathsf{Sb}_{\mathcal{G}}(\Delta, \Gamma)$ then $\mathbf{Bool}_{\mathcal{G}}[\gamma] = \mathbf{Bool}_{\mathcal{G}}$. It suffices to check that $\pi_i(\mathbf{Bool}_{\mathcal{G}}[\gamma]) = \pi_i(\mathbf{Bool}_{\mathcal{G}})$ with $i \in \{1, 2\}$. For $i = 1$, this is immediate from the stability of $\mathbf{Bool}$ under substitution in $\mathcal{T}$. For $i = 2$, one calculates to see that these two predicates agree for each $\delta \in \mathsf{Sb}_{\mathcal{G}}(\mathbf{1}, \Delta)$ and $b \in \mathsf{Tm}(\Delta^\circ, \mathbf{Bool})$:

$$\begin{aligned}
&\pi_2(\mathbf{Bool}_{\mathcal{G}}[\gamma])(\delta, b)\\
&= \{0 \mid b = \mathbf{true}\} \cup \{1 \mid b = \mathbf{false}\}\\
&= \pi_2(\mathbf{Bool}_{\mathcal{G}})(\delta, b)
\end{aligned}$$

It remains to construct $\mathbf{true}_{\mathcal{G}}$ and $\mathbf{false}_{\mathcal{G}}$ and to verify the orthogonality condition as well as the naturality equations. We begin with the two additional operations:

$$\begin{aligned}
\pi_1(\mathbf{true}_{\mathcal{G}}) &= \mathbf{true}\\
\pi_2(\mathbf{true}_{\mathcal{G}}) &= 0\\
\pi_1(\mathbf{false}_{\mathcal{G}}) &= \mathbf{false}\\
\pi_2(\mathbf{false}_{\mathcal{G}}) &= 1
\end{aligned}$$

We leave it to the reader to check that these two operations are natural in Γ. To verify the orthogonality condition, let us fix $A \in \mathsf{Ty}_{\mathcal{G}}(\Gamma.{}_{\mathcal{G}}\mathbf{Bool}_{\mathcal{G}})$. We may break apart the map $\mathsf{Tm}_{\mathcal{G}}(\Gamma.{}_{\mathcal{G}}\mathbf{Bool}_{\mathcal{G}}, A)$ to $\mathsf{Tm}_{\mathcal{G}}(\Gamma, A[\mathbf{id}.\mathbf{true}_{\mathcal{G}}]) \times \mathsf{Tm}_{\mathcal{G}}(\Gamma, A[\mathbf{id}.\mathbf{false}_{\mathcal{G}}])$ into several

steps, each of which is an isomorphism:

$$
\begin{aligned}
&\mathsf{Tm}_{\mathcal{G}}(\Gamma.{}_{\mathcal{G}}\mathbf{Bool}_{\mathcal{G}}, A)\\
&\cong \textstyle\sum_{a^\circ \in \mathsf{Tm}(\Gamma^\circ.\mathbf{Bool}, A^\circ)} \prod_{\gamma \in \mathsf{Sb}_{\mathcal{G}}(\mathbf{1}, \Gamma.{}_{\mathcal{G}}\mathbf{Bool}_{\mathcal{G}})} A^\bullet(\gamma, a^\circ[\gamma^\circ])\\
&\cong \textstyle\sum_{a^\circ \in \mathsf{Tm}(\Gamma^\circ.\mathbf{Bool}, A^\circ)} \prod_{\gamma_0 \in \mathsf{Sb}_{\mathcal{G}}(\mathbf{1}, \Gamma), b \in \mathsf{Tm}_{\mathcal{G}}(\mathbf{1}, \mathbf{Bool}_{\mathcal{G}})} A^\bullet(\gamma_0 .{}_{\mathcal{G}} b, a^\circ[\gamma^\circ.b^\circ])\\
&\cong \textstyle\sum_{a^\circ \in \mathsf{Tm}(\Gamma^\circ.\mathbf{Bool}, A^\circ)} \prod_{\gamma_0 \in \mathsf{Sb}_{\mathcal{G}}(\mathbf{1}, \Gamma), b^\circ \in \mathsf{Tm}(\mathbf{1}, \mathbf{Bool}), b^\bullet \in \mathbf{Bool}^\bullet_{\mathcal{G}}(\gamma_0, b)}\\
&\qquad A^\bullet(\gamma_0 .{}_{\mathcal{G}} (b^\circ, b^\bullet), a^\circ[\gamma^\circ.b^\circ])\\
&\cong \textstyle\sum_{a^\circ \in \mathsf{Tm}(\Gamma^\circ.\mathbf{Bool}, A^\circ)} \prod_{\gamma_0 \in \mathsf{Sb}_{\mathcal{G}}(\mathbf{1}, \Gamma)}\\
&\qquad A^\bullet(\gamma_0 .{}_{\mathcal{G}} \mathbf{true}_{\mathcal{G}}, a^\circ[\gamma^\circ.\mathbf{true}]) \times A^\bullet(\gamma_0 .{}_{\mathcal{G}} \mathbf{false}_{\mathcal{G}}, a^\circ[\gamma^\circ.\mathbf{false}])\\
&\cong \textstyle\sum_{a_t^\circ \in \mathsf{Tm}(\Gamma^\circ, A^\circ[\mathbf{id}.\mathbf{true}])} \sum_{a_f^\circ \in \mathsf{Tm}(\Gamma^\circ, A^\circ[\mathbf{id}.\mathbf{false}])} \prod_{\gamma_0 \in \mathsf{Sb}_{\mathcal{G}}(\mathbf{1}, \Gamma)}\\
&\qquad A^\bullet(\gamma_0 .{}_{\mathcal{G}} \mathbf{true}_{\mathcal{G}}, a_t^\circ) \times A^\bullet(\gamma_0 .{}_{\mathcal{G}} \mathbf{false}_{\mathcal{G}}, a_f^\circ)\\
&\cong \mathsf{Tm}_{\mathcal{G}}(\Gamma, A[\mathbf{id}.\mathbf{true}_{\mathcal{G}}]) \times \mathsf{Tm}_{\mathcal{G}}(\Gamma, A[\mathbf{id}.\mathbf{false}_{\mathcal{G}}])
\end{aligned}
$$

Finally, we must check that π preserves booleans. That is, we must show that $\pi_{\mathsf{Ty}_{\mathcal{G}}(\Gamma)}(\mathbf{Bool}_{\mathcal{G}}) = \mathbf{Bool}$ along with similar equations for **true** and **false**. However, since π acts on types and terms by projecting out the syntactic half of each, these equations are immediate consequences of our definitions of these operations. □

Lemma 6.7.10. *We can close $\mathcal{G}$ under unit types such that π extends to a morphism of models of type theory with* **Unit**.

Proof. We must construct two pieces of data to close $\mathcal{G}$ under **Unit**:

$$
\begin{aligned}
&\mathbf{Unit}_{\mathcal{G}} : \{\Gamma \in \mathsf{Cx}_{\mathcal{G}}\} \to \mathsf{Ty}_{\mathcal{G}}(\Gamma)\\
&\iota_{\mathbf{Unit}_{\mathcal{G}}} : \{\Gamma \in \mathsf{Cx}_{\mathcal{G}}\} \to \mathsf{Tm}_{\mathcal{G}}(\Gamma, \mathbf{Unit}_{\mathcal{G}}) \cong \{\star\}
\end{aligned}
$$

In addition, we must verify that both of these operations are suitably natural in Γ (a condition which trivializes for $\iota_{\mathbf{Unit}_{\mathcal{G}}}$).

We begin by defining $\mathbf{Unit}_{\mathcal{G}}$. As with contexts, it is convenient to break this down into specifying the syntactic component first and the family of proof-relevant predicates after the fact:

$$
\begin{aligned}
&\mathbf{Unit}_{\mathcal{G}} : \{\Gamma \in \mathsf{Cx}_{\mathcal{G}}\} \to \mathsf{Ty}_{\mathcal{G}}(\Gamma)\\
&\pi_1(\mathbf{Unit}_{\mathcal{G}}) = \mathbf{Unit}\\
&\pi_2(\mathbf{Unit}_{\mathcal{G}}) = \lambda \gamma\, t \to \{\star\}
\end{aligned}
$$

In other words, the syntactic component is **Unit** and the proof-relevant predicates are all trivial. Strictly speaking, one must verify that $\mathbf{Unit}_{\mathcal{G}}$ is stable under substitution but we leave this calculation to the reader.

Next, we construct the required isomorphism. For this, let us calculate:

$$\begin{aligned}
&\mathsf{Tm}_{\mathcal{G}}(\Gamma, \mathbf{Unit}_{\mathcal{G}}) \\
&\cong \textstyle\sum_{t \in \mathsf{Tm}(\Gamma^\circ, \mathbf{Unit})} \prod_{\gamma \in \mathsf{Sb}_{\mathcal{G}}(\mathbf{1}, \Gamma)} \mathbf{Unit}^\bullet_{\mathcal{G}}(\gamma, t) \\
&\cong \mathsf{Tm}(\Gamma^\circ, \mathbf{Unit}) && \mathbf{Unit}^\bullet_{\mathcal{G}} \text{ is trivial.} \\
&\cong \{\star\} && \text{Using the } \eta \text{ law on } \mathcal{T}.
\end{aligned}$$

In particular, the unique map $\mathsf{Tm}_{\mathcal{G}}(\Gamma, \mathbf{Unit}_{\mathcal{G}}) \to \{\star\}$ is a bijection as required.

Finally, we must check that π commutes with all of this new structure. For $\mathbf{Unit}_{\mathcal{G}}$, this is immediate by construction: π projects out the syntactic component and we have defined this to be $\mathbf{Unit}$. For ι it is automatic; there is only one map from $\mathsf{Tm}(\Gamma^\circ, \mathbf{Unit}) \to \{\star\}$ and inverses are unique when they exist, so both must be preserved by π. □

Lemma 6.7.11. *$\mathcal{G}$ is closed under extensional equality types and π preserves them.*

Proof. As with $\mathbf{Unit}$, we have two pieces of data to implement:

$$\begin{aligned}
&\mathbf{Eq}_{\mathcal{G}} : \{\Gamma \in \mathsf{Cx}_{\mathcal{G}}\}(A \in \mathsf{Ty}_{\mathcal{G}}(\Gamma)) \to \mathsf{Tm}_{\mathcal{G}}(\Gamma, A) \to \mathsf{Tm}_{\mathcal{G}}(\Gamma, A) \to \mathsf{Ty}_{\mathcal{G}}(\Gamma) \\
&\iota_{\mathbf{Eq}_{\mathcal{G}}} : \{\Gamma \in \mathsf{Cx}_{\mathcal{G}}\}(A \in \mathsf{Ty}_{\mathcal{G}}(\Gamma))(a\, b \in \mathsf{Tm}_{\mathcal{G}}(\Gamma, A)) \\
&\qquad \to \mathsf{Tm}_{\mathcal{G}}(\Gamma, \mathbf{Eq}_{\mathcal{G}}(A, a, b)) \cong \{\star \mid a = b\}
\end{aligned}$$

As in Lemma 6.7.10, we must check that $\mathbf{Eq}_{\mathcal{G}}$ is natural and preserved by π, while both of these conditions hold automatically for ι.

We once more break down the construction of $\mathbf{Eq}_{\mathcal{G}}$ into its syntactic component and family of proof-relevant predicates:

$$\begin{aligned}
\pi_1(\mathbf{Eq}_{\mathcal{G}}(A, a, b)) &= \mathbf{Eq}(A, a, b) \\
\pi_2(\mathbf{Eq}_{\mathcal{G}}(A, a, b)) &= \lambda \gamma\ z \to \{\star \mid a^\bullet(\gamma) = b^\bullet(\gamma)\}
\end{aligned}$$

For the canonicity data, we note that to compare $a^\bullet(\gamma)$ to $b^\bullet(\gamma)$, we must know that $a^\circ[\gamma^\circ] = b^\circ[\gamma^\circ]$. However, we know that z is an inhabitant of $\mathbf{Eq}(A^\circ[\gamma^\circ], a^\circ[\gamma^\circ], b^\circ[\gamma^\circ])$ and so, by equality reflection in $\mathcal{T}$, we have $a^\circ[\gamma^\circ] = b^\circ[\gamma^\circ]$. The reader may directly check that $\mathbf{Eq}_{\mathcal{G}}$ is natural and that it is preserved by π.

To substantiate the necessary isomorphism, we once more calculate:

$$\begin{aligned}
&\mathsf{Tm}_{\mathcal{G}}(\Gamma, \mathbf{Eq}_{\mathcal{G}}(A, a, b)) \\
&\cong \textstyle\sum_{t \in \mathsf{Tm}(\Gamma^\circ, \mathbf{Eq}(A^\circ, a^\circ, b^\circ))} \prod_{\gamma \in \mathsf{Sb}_{\mathcal{G}}(\mathbf{1}, \Gamma)} \mathbf{Eq}^\bullet_{\mathcal{G}}(\gamma, t) \\
&\cong \mathsf{Tm}(\Gamma^\circ, \mathbf{Eq}(A^\circ, a^\circ, b^\circ)) \times \{\star \mid a^\bullet = b^\bullet\} \\
&\cong \{\star \mid a^\circ = b^\circ\} \times \{\star \mid a^\bullet = b^\bullet\} && \text{Using the } \eta \text{ law on } \mathcal{T}. \\
&\cong \{\star \mid a = b\} && \square
\end{aligned}$$

Lemma 6.7.12. *$\mathcal{G}$ is closed under dependent products and π preserves them.*

Proof. To close $\mathcal{G}$ under dependent products (Structure 6.2.18), it suffices to exhibit operations pieces (satisfying suitable naturality equations):

$$\begin{aligned}
&\mathbf{\Pi}_{\mathcal{G}} : \{\Gamma \in \mathsf{Cx}_{\mathcal{G}}\}(A \in \mathsf{Ty}_{\mathcal{G}}(\Gamma)) \to \mathsf{Ty}_{\mathcal{G}}(\Gamma.{}_{\mathcal{G}}A) \to \mathsf{Ty}_{\mathcal{G}}(\Gamma)\\
&\iota_{\mathbf{\Pi}_{\mathcal{G}}} : \{\Gamma \in \mathsf{Cx}_{\mathcal{G}}\}(A \in \mathsf{Ty}_{\mathcal{G}}(\Gamma))(B \in \mathsf{Ty}_{\mathcal{G}}(\Gamma.{}_{\mathcal{G}}A))\\
&\qquad \to \mathsf{Tm}_{\mathcal{G}}(\Gamma, \mathbf{\Pi}_{\mathcal{G}}(A,B)) \cong \mathsf{Tm}_{\mathcal{G}}(\Gamma.{}_{\mathcal{G}}A, B)
\end{aligned}$$

We begin by defining $\mathbf{\Pi}_{\mathcal{G}}$:

$$\begin{aligned}
&\pi_1(\mathbf{\Pi}_{\mathcal{G}}(A,B)) = \mathbf{\Pi}(A^\circ, B^\circ)\\
&\pi_2(\mathbf{\Pi}_{\mathcal{G}}(A,B)) = \lambda\gamma f \to \textstyle\prod_{a \in \mathsf{Tm}_{\mathcal{G}}(\mathbf{1}, A[\gamma])} B^\bullet(\gamma.{}_{\mathcal{G}}a, \mathbf{app}(f, a^\circ))
\end{aligned}$$

We leave the routine calculations that this is natural to the reader along with the proof that this is preserved by π. It remains to construct ι. For this, we calculate:

$$\begin{aligned}
&\mathsf{Tm}_{\mathcal{G}}(\Gamma, \mathbf{\Pi}_{\mathcal{G}}(A,B))\\
&\cong \textstyle\sum_{f^\circ \in \mathsf{Tm}(\Gamma^\circ, \mathbf{\Pi}(A^\circ, B^\circ))} \prod_{\gamma \in \mathsf{Sb}_{\mathcal{G}}(\mathbf{1}, \Gamma)} \mathbf{\Pi}_{\mathcal{G}}(A,B)^\bullet(\gamma, f^\circ[\gamma^\circ])\\
&\cong \textstyle\sum_{f^\circ \in \mathsf{Tm}(\Gamma^\circ, \mathbf{\Pi}(A^\circ, B^\circ))} \prod_{\gamma \in \mathsf{Sb}_{\mathcal{G}}(\mathbf{1}, \Gamma), a \in \mathsf{Tm}_{\mathcal{G}}(\mathbf{1}, A[\gamma])} B^\bullet(\gamma.{}_{\mathcal{G}}a, \mathbf{app}(f, a^\circ))\\
&\cong \textstyle\sum_{f^\circ \in \mathsf{Tm}(\Gamma^\circ.A^\circ, B^\circ)} \prod_{\gamma \in \mathsf{Sb}_{\mathcal{G}}(\mathbf{1}, \Gamma.{}_{\mathcal{G}}A)} B^\bullet(\gamma.{}_{\mathcal{G}}a, f^\circ[\gamma^\circ.a^\circ])\\
&\cong \mathsf{Tm}_{\mathcal{G}}(\Gamma.{}_{\mathcal{G}}A, B)
\end{aligned}$$

The reader may check directly that this chain of isomorphisms is natural and that it is sent by π to the natural bijection given as part of $\mathcal{T}$. □

Lemma 6.7.13. *$\mathcal{G}$ is closed under $\mathbf{U}_0$ and its attendant operations and π preserves them.*

Proof. Closing $\mathcal{G}$ under a universe is a somewhat arduous process as universes (in extensional type theory) do not have a simple universal property. For this reason, we shall describe the interpretation of **U** and **El** carefully but content ourselves with merely sketching how to close it under all connectives.

The crucial idea of $\mathbf{U}_{\mathcal{G}}$ is to store not just the canonical form of an element in the associated canonicity data, but also the canonicity data of the small type encoded by that element. Accordingly, we first introduce an auxiliary definition stating that $c \in \mathsf{Tm}(\mathbf{1}, \mathbf{U})$ is in canonical form:

$$\begin{aligned}
&\mathsf{Canonical}(c) =\\
&\quad \{0 \mid c = \mathbf{unit}\} \cup \{1 \mid c = \mathbf{void}\} \cup \{2 \mid c = \mathbf{bool}\} \cup \{3 \mid c = \mathbf{nat}\}\\
&\quad \cup \{4 \mid \exists c', a, b.\ c = \mathbf{eq}(c', a, b)\}\\
&\quad \cup \{5 \mid \exists c_0, c_1.\ c = \mathbf{pi}(c_0, c_1)\}\\
&\quad \cup \{5 \mid \exists c_0, c_1.\ c = \mathbf{sig}(c_0, c_1)\}
\end{aligned}$$

We now define $\mathbf{U}_{\mathcal{G}}$ as follows:

$$\pi_1(\mathbf{U}_{\mathcal{G}}) = \mathbf{U}_0$$
$$\pi_2(\mathbf{U}_{\mathcal{G}}) = \lambda\gamma\ c \to \mathsf{Canonical}(c) \times (\mathsf{Tm}(\mathbf{1}, \mathbf{El}(c^\circ)) \to V_0)$$

Notice that $\pi_2(\mathbf{U}_{\mathcal{G}})$ is our first predicate which is necessarily proof-relevant. Previously one could get by with $A^\bullet(\gamma, t)$ being a mere proposition for each γ and t, but the same is not true for $\mathbf{U}_{\mathcal{G}}$. In particular, since $\pi_2(\mathbf{U}_{\mathcal{G}})(\gamma, c)$ contains an element of $\mathsf{Tm}(\mathbf{1}, \mathbf{El}(c^\circ)) \to V_0$ it will frequently contain at least as many elements as V_0! In fact, the universe is the sole connective we encounter in extensional type theory with this property. Note too that we have used the fact that V_ω (used to define $\mathsf{Ty}_{\mathcal{G}}(\Gamma)$) contains the Grothendieck universe V_0. This is similar to the situation encountered in Section 3.5 where an $(n+1)$-hierarchy of Grothendieck universes was used to interpret a hierarchy of n-universes.

We note, however, that the other half of the computability data for $\mathbf{U}$ is somewhat spurious. Since there is no elimination form for $\mathbf{U}_0$, we do not technically need to specify $\mathsf{Canonical}(c)$ for closed elements. Its inclusion, however, will allow us to prove a slightly stronger canonicity theorem that also specifies elements of the universe. The second component of the computability data $(\mathsf{Tm}(\mathbf{1}, \mathbf{El}(c^\circ)) \to V_0)$, the one that actually forces this to be proof-relevant, is mandatory; we need it to define $\mathbf{El}_{\mathcal{G}}$:

$$\pi_1(\mathbf{El}_{\mathcal{G}}(c)) = \mathbf{El}(c^\circ)$$
$$\pi_2(\mathbf{El}_{\mathcal{G}}(c)) = \lambda\gamma\ a \to \pi_2(c^\bullet(\gamma))(a)$$

We once more leave the naturality conditions for both $\mathbf{U}_{\mathcal{G}}$ and $\mathbf{El}_{\mathcal{G}}$ to the reader.

Closing this universe under various connectives is a procedure analogous to the problem faced by the set model (Section 3.5). Since V_0 is large enough to be closed in V_ω under all relevant operations, we can replay the construction of *e.g.*, $\mathbf{Bool}_{\mathcal{G}}$ to define $\mathbf{bool}_{\mathcal{G}}$:

$$\pi_1(\mathbf{bool}_{\mathcal{G}}) = \mathbf{bool}$$
$$\pi_2(\mathbf{bool}_{\mathcal{G}}) = \lambda\gamma \to (2, \lambda b \to \{0 \mid b = \mathbf{true}\} \cup \{1 \mid b = \mathbf{false}\})$$

Routine calculation then shows that $\mathbf{El}_{\mathcal{G}}(\mathbf{bool}_{\mathcal{G}}) = \mathbf{Bool}_{\mathcal{G}}$. The remaining connectives follow the same pattern, so we leave the reader to handle those cases. □

We may collect this series of lemmas into the main result of this section:

Theorem 6.7.14. *$\mathcal{G}$ is a model of type theory closed under all the connectives and π is a morphism of such models.*

6.7.3 Deriving canonicity

With Theorem 6.7.14 to hand, we can prove canonicity for extensional type theory following exactly the argument outlined at the beginning of this section.

Theorem 6.7.15. *Suppose* $b \in \mathsf{Tm}(\mathbf{1}, \mathbf{Bool})$ *then* $b = \mathbf{true}$ *or* $b = \mathbf{false}$.

Proof. By Theorem 3.4.5 and Theorem 6.7.14, we know that there is a morphism of models $i : \mathcal{T} \longrightarrow \mathcal{G}$ and that $\pi \circ i = \mathsf{id}$. Consequently, we know that $b = \pi_{\mathsf{Tm}_{\mathcal{G}}(\mathbf{1},\mathbf{Bool}_{\mathcal{G}})}(i_{\mathsf{Tm}(\mathbf{1},\mathbf{Bool})}(b))$. Let us then consider what data is contained in $\bar{b} = i_{\mathsf{Tm}(\mathbf{1},\mathbf{Bool})}(b) \in \mathsf{Tm}_{\mathcal{G}}(\mathbf{1}, \mathbf{Bool}_{\mathcal{G}})$. As a term in $\mathcal{G}$, we know that $\bar{b}$ is a pair. We analyze the two components separately.

The first component $\pi_1(\bar{b})$ is an element of $\mathsf{Tm}(\mathbf{1}, \mathbf{Bool})$. Moreover, since we have already noted that $b = \pi_{\mathsf{Tm}_{\mathcal{G}}(\mathbf{1},\mathbf{Bool}_{\mathcal{G}})}(\bar{b})$, we in fact know that $\pi_1(\bar{b}) = b$. The second component $\pi_2(\bar{b})$ is an element of the following set:

$$\textstyle\prod_{\gamma \in \mathsf{Sb}_{\mathcal{G}}(\mathbf{1},\mathbf{1})} \mathbf{Bool}^{\bullet}_{\mathcal{G}}(\gamma, b[\gamma^{\circ}])$$

However, since $\mathbf{1}$ is terminal, we must know that $\mathsf{Sb}_{\mathcal{G}}(\mathbf{1}, \mathbf{1})$ consists of a single element: the identity substitution. Consequently, the data in $\pi_2(\bar{b})$ collapses to an element of $\mathbf{Bool}^{\bullet}_{\mathcal{G}}(\mathsf{id}, b[\gamma^{\circ}])$. Unfolding the definition of $\mathbf{Bool}^{\bullet}_{\mathcal{G}}$ from Lemma 6.7.9, we see that this amounts to an element of $b = \mathbf{true} \sqcup b = \mathbf{false}$. The conclusion is now immediate. □

In fact, the additional properties of $\mathcal{G}$ allow us to deduce a more refined result than merely characterizing closed booleans.

Exercise 6.31. Modify the argument given in Theorem 6.7.15 to prove the following: if $c \in \mathsf{Tm}(\mathbf{1}, \mathbf{U}_0)$ then one of the following conditions must hold:

- $c = \mathbf{unit}$, $c = \mathbf{void}$, or $c = \mathbf{bool}$,
- there exists $c' \in \mathsf{Tm}(\mathbf{1}, \mathbf{U}_0)$ and $a, b \in \mathsf{Tm}(\mathbf{1}, \mathbf{El}(c'))$ such that $c = \mathbf{eq}(c', a, b)$,
- there exists $c_0 \in \mathsf{Tm}(\mathbf{1}, \mathbf{U}_0)$ and $c_1 \in \mathsf{Tm}(\mathbf{1}.\mathbf{El}(c_0), \mathbf{U}_0)$ such that $c = \mathbf{pi}(c_0, c_1)$,
- or there exists $c_0 \in \mathsf{Tm}(\mathbf{1}, \mathbf{U}_0)$ and $c_1 \in \mathsf{Tm}(\mathbf{1}.\mathbf{El}(c_0), \mathbf{U}_0)$ such that $c = \mathbf{sig}(c_0, c_1)$,

Similar statements may be proven for **Void** and **Nat** (that there are no closed elements or a closed element is equal to $\mathbf{suc}^m(\mathbf{zero})$ for some numeral m, respectively). The reader may wish to carefully work out the definitions of $\mathbf{Nat}_{\mathcal{M}}$ for themselves to confirm that this is indeed the case. We note that doing so for **Void** provides another proof of the consistency of ETT.

6.7.4 *Variations on gluing arguments*

In this section, we have constructed $\mathcal{G}$ to prove canonicity. Along the way, the reader may have noticed similarities in how various operations were defined and wondered if there might be a more conceptual argument available. For instance, how much of this construction really depends on our choice to use $\mathcal{T}$ instead of some other model? To what extent was the model we constructed in Section 3.5 really used? Could we have hidden some of the indices in this proof which caused so much bureaucracy in this proof? Fortunately, the answer to all of these questions is affirmative. In fact, these questions have motivated a great deal of recent work in dependent type theory [AK16; Shu15; KHS19; Coq19]. Roughly, this work attempts to improve on the style of gluing argument we have detailed above in two distinct ways:

- They axiomitize the precise requirements needed on $\mathcal{T}$, $\mathcal{S}$, and Γ to allow this argument to be repurposed for closely related systems and proofs.
- They capitalize on the models of type theory in the various categories being manipulated (such as $\mathbf{Pr}(\mathsf{Cx})$) to alleviate the bookkeeping and ensure that these constructions are easier to follow.

For the first point, for instance, Kaposi, Huber, and Sattler [KHS19] introduce the notion of a *pseudo*-morphism of models of type theory. Roughly, while a morphism of models insists that every connective and operation of type theory be preserved, a pseudo-morphism of models requires only that context extension, the empty context and their attendant substitutions are preserved up to (canonical) isomorphism. In particular, there is no requirement that various type formers be preserved. The authors show that this suffices to carry out a version of the gluing model we described above. For instance, if $\mathcal{M}$ and $\mathcal{N}$ are models and $F : \mathcal{M} \longrightarrow \mathcal{N}$ is a pseudo-morphism between them, one can define a model $\mathcal{G}(F)$ whose category of contexts is again $\mathbf{Gl}(F)$ and whose presheaf of types is given as follows:

$$\mathsf{Ty}_{\mathcal{G}(F)}((\Gamma_0, \Gamma_1, \gamma) : \mathbf{Gl}(F)) = \textstyle\sum_{A \in \mathsf{Ty}_{\mathcal{M}}(\Gamma_0)} \mathsf{Ty}_{\mathcal{N}}(\Gamma_1 . {}_{\mathcal{N}}F_{\mathsf{Ty}_{\mathcal{M}}(\Gamma_0)}(A)[\gamma])$$

In other words, a type in the glued model is a type A from $\mathcal{M}$ together with a family of types in $\mathcal{N}$ parameterized by elements of $F(A)$. The reader may confirm that these agrees with our definition of $\mathsf{Ty}_{\mathcal{G}}(\Gamma)$ in the case where $\mathcal{M} = \mathcal{T}$, $\mathcal{N} = \mathcal{S}$ and $F = \Gamma$. In fact, all of our definitions could have been replayed in this more general setting and we could have derived the following result:

Theorem 6.7.16 (Kaposi, Huber, and Sattler [KHS19])**.** *If $\mathcal{M}$ and $\mathcal{N}$ are models of type theory closed under all connectives and F is a pseudo-morphism between them, then* $\mathbf{Gl}(F)$

supports a model of type theory closed under all connectives and $\pi_1 : \mathbf{Gl}(F) \longrightarrow \mathsf{Cx}_{\mathcal{M}}$ *extends to a homomorphism of models.*

Even with this additional generality, for certain gluing models (*e.g.*, those arising in proofs of normalization) the model itself is complex enough to warrant more abstract arguments in its construction. To this end, Sterling and Harper [SH21] and Sterling and Angiuli [SA21] observed that much of the construction of $\mathcal{G}$ could be done fully within dependent type theory by extending extensional type theory with a handful of constants and interpreting the theory into a certain presheaf category. Working with dependent type theory to carry out these constructions meant that many of the naturality obligations we left to the reader would become automatic. Even better, for a number of constructions *e.g.*, the construction of $\mathbf{\Pi}$-types, one many constructions become remarkably short and simple. While for canonicity proofs this is less of an issue, as the gluing model becomes more complex these naturality requirements become increasingly tedious and difficult to manage. We refer the reader to Sterling [Ste21] for an exposition of these style of gluing arguments (as well as a complete proof of normalization for cubical type theory as introduced in Section 5.3).

Further reading

In general, after working through the material of this chapter, a great deal of literature is now within reach of the reader. From this, we isolate a handful of topics the reader might find interesting.

Alternatives to categories with families While we have focused on categories with families in this chapter, other styles of semantic models are considered by type theorists and the reader may find it interesting to explore them. For instance, Jacobs [Jac99] explores categorical semantics in exceptional generality (including but not limited to dependent type theory) through the framework of Grothendieck fibrations and comprehension categories. Cartmell's [Car86] contextual categories, also known as C-systems [Voe17], have been used in various important models such as the simplicial model of type theory [KL21]. In general, while quite a lot of work uses categories with families—often in its natural model guise, as we have done in this chapter—alternative styles of model are not uncommon and we recommend Ahrens, Lumsdaine, and North [ALN24] as a guide to comparing these different styles of models.

Concrete models of type theory While Section 6.6 has provided the reader with a good supply of models in general, they may well wish to explore particular models more carefully. We have already mentioned Hofmann and Streicher's groupoid model [HS98] in Section 4.3.2 as well as von Glehn's polynomial endofunctor model [vGle14] for illustrating the limitations of the intensional identity type. The reader should become also familiar with presheaf models [Hof97] and realizability models [Reu99] of type theory, as they both provide new insights on type theory and can be used to prove a variety of consistency and independence results. Finally, the more ambitious reader who is also familiar with Chapter 5 may find the simplicial sets model of HoTT [KL21] as well as the cubical sets model [BCH14; OP16] interesting to study.

Metatheory through models Section 6.7 highlighted how models of type theory arising from Artin gluing may be used to prove some of the metatheorems discussed in Chapter 3. For reasons of space, we gave only the proof of canonicity, but the reader is encouraged to study similarly semantic proofs of normalization [Coq18; Ste21; BKS23]. Furthermore, as previously mentioned, semantic proofs of conservativity results (such as Theorem 4.3.18) are also possible [KL25].

This material will be published by Cambridge University Press as *Principles of Dependent Type Theory* by Carlo Angiuli and Daniel Gratzer. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © Carlo Angiuli and Daniel Gratzer 2024

Martin-Löf type theory

A

This appendix presents a substitution calculus [Mar92; Tas93; Dyb96] for several variants of Martin-Löf's dependent type theory. Martin-Löf type theories are systems admitting the rules in section *Contexts and substitutions*; the rules specific to extensional type theory, those axiomatizing extensional equality types, are marked (ETT); the rules specific to intensional type theory, those axiomatizing *intensional identity types*, are marked (ITT).

Judgments

Martin-Löf type theory has four basic judgments:

1. $\vdash \Gamma\ \mathsf{cx}$ asserts that Γ is a context.
2. $\Delta \vdash \gamma : \Gamma$, presupposing $\vdash \Delta\ \mathsf{cx}$ and $\vdash \Gamma\ \mathsf{cx}$, asserts that γ is a substitution from Δ to Γ (*i.e.*, assigns a term in Δ to each variable in Γ).
3. $\Gamma \vdash A\ \mathsf{type}$, presupposing $\vdash \Gamma\ \mathsf{cx}$, asserts that A is a type in context Γ.
4. $\Gamma \vdash a : A$, presupposing $\vdash \Gamma\ \mathsf{cx}$ and $\Gamma \vdash A\ \mathsf{type}$, asserts that a is an element/term of type A in context Γ.

The *presuppositions* of a judgment are its meta-implicit-arguments, so to speak. For instance, the judgment $\Gamma \vdash A\ \mathsf{type}$ is sensible to write (is meta-well-typed) only when the judgment $\vdash \Gamma\ \mathsf{cx}$ holds. We adopt the convention that asserting the truth of a judgment implicitly asserts its well-formedness; thus asserting $\Gamma \vdash A\ \mathsf{type}$ also asserts $\vdash \Gamma\ \mathsf{cx}$.

As we assert the existence of various contexts, substitutions, types, and terms, we will simultaneously need to assert that some of these (already introduced) objects are equal to other (already introduced) objects of the same kind.

1. $\Delta \vdash \gamma = \gamma' : \Gamma$, presupposing $\Delta \vdash \gamma : \Gamma$ and $\Delta \vdash \gamma' : \Gamma$, asserts that γ, γ' are equal substitutions from Δ to Γ.
2. $\Gamma \vdash A = A'\ \mathsf{type}$, presupposing $\Gamma \vdash A\ \mathsf{type}$ and $\Gamma \vdash A'\ \mathsf{type}$, asserts that A, A' are equal types in context Γ.

3. $\Gamma \vdash a = a' : A$, presupposing $\Gamma \vdash a : A$ and $\Gamma \vdash a' : A$, asserts that a, a' are equal elements of type A in context Γ.

Two types (*resp.*, contexts, substitutions, terms) being equal has the force that it does in standard mathematics: any expression can be replaced silently by an equal expression without affecting the meaning or truth of the statement in which it appears. One important example of this principle is the "conversion rule" which states that if $\Gamma \vdash A = A'$ type and $\Gamma \vdash a : A$, then $\Gamma \vdash a : A'$.

In the rules that follow, some arguments of substitution, type, and term formers are typeset as gray subscripts; these are arguments that we will often omit because they can be inferred from context and are tedious and distracting to write.

Contexts and substitutions

$$\frac{}{\vdash \mathbf{1}\ \mathsf{cx}}\ \textsc{cx/emp} \qquad \frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma \vdash A\ \mathsf{type}}{\vdash \Gamma.A\ \mathsf{cx}}\ \textsc{cx/ext}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{id}_\Gamma : \Gamma}\ \textsc{sb/id} \qquad \frac{\Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0}{\Gamma_2 \vdash \gamma_0 \circ_{\Gamma_2,\Gamma_1,\Gamma_0} \gamma_1 : \Gamma_0}\ \textsc{sb/comp}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{id}_\Gamma \circ \gamma = \gamma : \Gamma} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \gamma \circ \mathbf{id}_\Delta = \gamma : \Gamma}$$

$$\frac{\Gamma_3 \vdash \gamma_2 : \Gamma_2 \qquad \Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0}{\Gamma_3 \vdash \gamma_0 \circ (\gamma_1 \circ \gamma_2) = (\gamma_0 \circ \gamma_1) \circ \gamma_2 : \Gamma_0}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\ \mathsf{type}}{\Delta \vdash A[\gamma]_{\Delta,\Gamma}\ \mathsf{type}}\ \textsc{ty/sb} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A}{\Delta \vdash a[\gamma]_{\Delta,\Gamma} : A[\gamma]}\ \textsc{tm/sb}$$

$$\frac{\Gamma \vdash A\ \mathsf{type}}{\Gamma \vdash A[\mathbf{id}_\Gamma] = A\ \mathsf{type}} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash a[\mathbf{id}_\Gamma] = a : A}$$

$$\frac{\Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0 \qquad \Gamma_0 \vdash A\ \mathsf{type}}{\Gamma_2 \vdash A[\gamma_0 \circ \gamma_1] = A[\gamma_0][\gamma_1]\ \mathsf{type}}$$

$$\frac{\Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0 \qquad \Gamma_0 \vdash a : A}{\Gamma_2 \vdash a[\gamma_0 \circ \gamma_1] = a[\gamma_0][\gamma_1] : A[\gamma_0 \circ \gamma_1]}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash !_\Gamma : \mathbf{1}}\ \textsc{sb/emp} \qquad \frac{\Gamma \vdash \delta : \mathbf{1}}{\Gamma \vdash !_\Gamma = \delta : \mathbf{1}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Delta \vdash a : A[\gamma]}{\Delta \vdash \gamma.{}_{\Delta,\Gamma,A}a : \Gamma.A}\ \textsc{sb/ext} \qquad \frac{\Gamma \vdash A\,\mathsf{type}}{\Gamma.A \vdash \mathbf{p}_{\Gamma,A} : \Gamma}\ \textsc{sb/wk}$$

$$\frac{\Gamma \vdash A\,\mathsf{type}}{\Gamma.A \vdash \mathbf{q}_{\Gamma,A} : A[\mathbf{p}_{\Gamma,A}]}\ \textsc{var} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Delta \vdash a : A[\gamma]}{\Delta \vdash \mathbf{p}_{\Gamma,A} \circ_{\Gamma.A} (\gamma.a) = \gamma : \Gamma}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Delta \vdash a : A[\gamma]}{\Delta \vdash \mathbf{q}_{\Gamma,A}[\gamma.a] = a : A[\gamma]} \qquad \frac{\Delta \vdash \gamma : \Gamma.A}{\Delta \vdash \gamma = (\mathbf{p}_{\Gamma,A} \circ_{\Gamma.A} \gamma).(\mathbf{q}_{\Gamma,A}[\gamma]) : \Gamma.A}$$

$\mathbf{\Pi}$-*types*

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type}}{\Gamma \vdash \mathbf{\Pi}_\Gamma(A,B)\,\mathsf{type}}\ \textsc{pi/form} \qquad \frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash b : B}{\Gamma \vdash \lambda_{\Gamma,A,B}(b) : \mathbf{\Pi}(A,B)}\ \textsc{pi/intro}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash f : \mathbf{\Pi}(A,B)}{\Gamma \vdash \mathbf{app}_{\Gamma,A,B}(f,a) : B[\mathbf{id}_\Gamma.a]}\ \textsc{pi/elim}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type}}{\Delta \vdash \mathbf{\Pi}_\Gamma(A,B)[\gamma] = \mathbf{\Pi}_\Delta(A[\gamma], B[(\gamma \circ \mathbf{p}_{\Delta,A[\gamma]}).\mathbf{q}_{\Delta,A[\gamma]}])\,\mathsf{type}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash b : B}{\Delta \vdash \lambda(b)[\gamma] = \lambda(b[(\gamma \circ \mathbf{p}).\mathbf{q}]) : \mathbf{\Pi}(A,B)[\gamma]}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash f : \mathbf{\Pi}(A,B)}{\Delta \vdash \mathbf{app}(f,a)[\gamma] = \mathbf{app}(f[\gamma], a[\gamma]) : B[(\mathbf{id}_\Gamma.a) \circ \gamma]}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash b : B}{\Gamma \vdash \mathbf{app}(\lambda(b), a) = b[\mathbf{id}.a] : B[\mathbf{id}.a]}$$

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash f : \mathbf{\Pi}(A,B)}{\Gamma \vdash f = \lambda(\mathbf{app}(f[\mathbf{p}_{\Gamma,A}], \mathbf{q}_{\Gamma,A})) : \mathbf{\Pi}(A,B)}$$

Σ-types

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type}}{\Gamma \vdash \Sigma_\Gamma(A,B)\,\mathsf{type}}\ \textsc{sigma/form}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}_\Gamma.a]}{\Gamma \vdash \mathbf{pair}_{\Gamma,A,B}(a,b) : \Sigma(A,B)}\ \textsc{sigma/intro}$$

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash \mathbf{fst}_{\Gamma,A,B}(p) : A}\ \textsc{sigma/elim/fst}$$

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash \mathbf{snd}_{\Gamma,A,B}(p) : B[\mathbf{id}_\Gamma.\mathbf{fst}(p)]}\ \textsc{sigma/elim/snd}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type}}{\Delta \vdash \Sigma_\Gamma(A,B)[\gamma] = \Sigma_\Delta(A[\gamma], B[(\gamma \circ \mathbf{p}).\mathbf{q}])\,\mathsf{type}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}.a]}{\Delta \vdash \mathbf{pair}(a,b)[\gamma] = \mathbf{pair}(a[\gamma], b[\gamma]) : \Sigma(A,B)[\gamma]}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Delta \vdash \mathbf{fst}(p)[\gamma] = \mathbf{fst}(p[\gamma]) : A[\gamma]}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Delta \vdash \mathbf{snd}(p)[\gamma] = \mathbf{snd}(p[\gamma]) : B[(\mathbf{id}.\mathbf{fst}(p)) \circ \gamma]}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}.a]}{\Gamma \vdash \mathbf{fst}(\mathbf{pair}(a,b)) = a : A}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash b : B[\mathbf{id}.a]}{\Gamma \vdash \mathbf{snd}(\mathbf{pair}(a,b)) = b : B[\mathbf{id}.a]}$$

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma.A \vdash B\,\mathsf{type} \qquad \Gamma \vdash p : \Sigma(A,B)}{\Gamma \vdash p = \mathbf{pair}(\mathbf{fst}(p), \mathbf{snd}(p)) : \Sigma(A,B)}$$

Extensional equality types

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash \mathbf{Eq}_\Gamma(A, a, b)\ \mathrm{type}}\ \textsc{eq/form}\ (\textsc{ett}) \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{refl}_{\Gamma,A,a} : \mathbf{Eq}(A, a, a)}\ \textsc{eq/intro}\ (\textsc{ett})$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Delta \vdash \mathbf{Eq}_\Gamma(A, a, b)[\gamma] = \mathbf{Eq}_\Delta(A[\gamma], a[\gamma], b[\gamma])\ \mathrm{type}}\ (\textsc{ett})$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A}{\Delta \vdash \mathbf{refl}[\gamma] = \mathbf{refl} : \mathbf{Eq}(A, a, a)[\gamma]}\ (\textsc{ett})$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : \mathbf{Eq}(A, a, b)}{\Gamma \vdash a = b : A}\ (\textsc{ett})$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : \mathbf{Eq}(A, a, b)}{\Gamma \vdash p = \mathbf{refl} : \mathbf{Eq}(A, a, b)}\ (\textsc{ett})$$

Unit type

$$\frac{\vdash \Gamma\ \mathrm{cx}}{\Gamma \vdash \mathbf{Unit}_\Gamma\ \mathrm{type}}\ \textsc{unit/form} \qquad \frac{\vdash \Gamma\ \mathrm{cx}}{\Gamma \vdash \mathbf{tt}_\Gamma : \mathbf{Unit}}\ \textsc{unit/intro}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Unit}_\Gamma[\gamma] = \mathbf{Unit}_\Delta\ \mathrm{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{tt}_\Gamma[\gamma] = \mathbf{tt}_\Delta : \mathbf{Unit}} \qquad \frac{\Gamma \vdash a : \mathbf{Unit}}{\Gamma \vdash a = \mathbf{tt} : \mathbf{Unit}}$$

Empty type

$$\frac{\vdash \Gamma\ \mathrm{cx}}{\Gamma \vdash \mathbf{Void}_\Gamma\ \mathrm{type}}\ \textsc{empty/form} \qquad \frac{\Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash A\ \mathrm{type}}{\Gamma \vdash \mathbf{absurd}_{\Gamma,A}(b) : A[\mathbf{id}.b]}\ \textsc{empty/elim}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Void}_\Gamma[\gamma] = \mathbf{Void}_\Delta\ \mathrm{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash b : \mathbf{Void} \qquad \Gamma.\mathbf{Void} \vdash A\ \mathrm{type}}{\Delta \vdash \mathbf{absurd}(b)[\gamma] = \mathbf{absurd}(b[\gamma]) : A[\gamma.b[\gamma]]}$$

Boolean type

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{Bool}_\Gamma\ \mathsf{type}}\ \textsc{bool/form}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{true}_\Gamma : \mathbf{Bool}}\ \textsc{bool/intro/true} \qquad \frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{false}_\Gamma : \mathbf{Bool}}\ \textsc{bool/intro/false}$$

$$\frac{\Gamma \vdash b : \mathbf{Bool} \qquad \Gamma.\mathbf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]}{\Gamma \vdash \mathbf{if}_{\Gamma,A}(a_t, a_f, b) : A[\mathbf{id}.b]}\ \textsc{bool/elim}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Bool}_\Gamma[\gamma] = \mathbf{Bool}_\Delta\ \mathsf{type}}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{true}_\Gamma[\gamma] = \mathbf{true}_\Delta : \mathbf{Bool}} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{false}_\Gamma[\gamma] = \mathbf{false}_\Delta : \mathbf{Bool}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash b : \mathbf{Bool} \qquad \Gamma.\mathbf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]}{\Delta \vdash \mathbf{if}\,(a_t, a_f, b)[\gamma] = \mathbf{if}\,(a_t[\gamma], a_f[\gamma], b[\gamma]) : A[\gamma.b[\gamma]]}$$

$$\frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma.\mathbf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]}{\Gamma \vdash \mathbf{if}\,(a_t, a_f, \mathbf{true}) = a_t : A[\mathbf{id}.\mathbf{true}]}$$

$$\frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma.\mathbf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_t : A[\mathbf{id}.\mathbf{true}] \qquad \Gamma \vdash a_f : A[\mathbf{id}.\mathbf{false}]}{\Gamma \vdash \mathbf{if}\,(a_t, a_f, \mathbf{false}) = a_f : A[\mathbf{id}.\mathbf{false}]}$$

Coproduct types

$$\frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash B\,\mathsf{type}}{\Gamma \vdash A +_\Gamma B\,\mathsf{type}}\ \textsc{plus/form}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash B\,\mathsf{type}}{\Gamma \vdash \mathbf{inl}_{\Gamma,A,B}(a) : A + B}\ \textsc{plus/intro/inl} \qquad \frac{\Gamma \vdash b : B \qquad \Gamma \vdash A\,\mathsf{type}}{\Gamma \vdash \mathbf{inr}_{\Gamma,A,B}(b) : A + B}\ \textsc{plus/intro/inr}$$

$$\frac{\begin{array}{c}\Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash B\,\mathsf{type} \qquad \Gamma.(A+B) \vdash C\,\mathsf{type} \\ \Gamma.A \vdash c_l : C[\mathbf{p}.\mathbf{inl}(\mathbf{q})] \qquad \Gamma.B \vdash c_r : C[\mathbf{p}.\mathbf{inr}(\mathbf{q})] \qquad \Gamma \vdash p : A + B\end{array}}{\Gamma \vdash \mathbf{case}_{\Gamma,A,B,C}(c_l, c_r, p) : C[\mathbf{id}.p]}\ \textsc{plus/elim}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash B\,\mathsf{type}}{\Delta \vdash (A +_\Gamma B)[\gamma] = A[\gamma] +_\Delta B[\gamma]\,\mathsf{type}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma \vdash B\,\mathsf{type}}{\Delta \vdash \mathbf{inl}(a)[\gamma] = \mathbf{inl}(a[\gamma]) : A[\gamma] + B[\gamma]}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash b : B \qquad \Gamma \vdash A\,\mathsf{type}}{\Delta \vdash \mathbf{inr}(b)[\gamma] = \mathbf{inr}(b[\gamma]) : A[\gamma] + B[\gamma]}$$

$$\frac{\begin{array}{c}\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash B\,\mathsf{type} \qquad \Gamma.(A+B) \vdash C\,\mathsf{type} \\ \Gamma.A \vdash c_l : C[\mathbf{p}.\mathbf{inl}(\mathbf{q})] \qquad \Gamma.B \vdash c_r : C[\mathbf{p}.\mathbf{inr}(\mathbf{q})] \qquad \Gamma \vdash p : A + B\end{array}}{\Delta \vdash \mathbf{case}(c_l, c_r, p)[\gamma] = \mathbf{case}(c_l[\gamma.A], c_r[\gamma.B], p[\gamma]) : C[\gamma.p[\gamma]]}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : A \qquad \Gamma \vdash B\,\mathsf{type} \\ \Gamma.(A+B) \vdash C\,\mathsf{type} \qquad \Gamma.A \vdash c_l : C[\mathbf{p}.\mathbf{inl}(\mathbf{q})] \qquad \Gamma.B \vdash c_r : C[\mathbf{p}.\mathbf{inr}(\mathbf{q})]\end{array}}{\Gamma \vdash \mathbf{case}(c_l, c_r, \mathbf{inl}(a)) = c_l[\mathbf{id}.a] : C[\mathbf{id}.\mathbf{inl}(a)]}$$

$$\frac{\begin{array}{c}\Gamma \vdash b : B \qquad \Gamma \vdash A\,\mathsf{type} \\ \Gamma.(A+B) \vdash C\,\mathsf{type} \qquad \Gamma.A \vdash c_l : C[\mathbf{p}.\mathbf{inl}(\mathbf{q})] \qquad \Gamma.B \vdash c_r : C[\mathbf{p}.\mathbf{inr}(\mathbf{q})]\end{array}}{\Gamma \vdash \mathbf{case}(c_l, c_r, \mathbf{inr}(b)) = c_r[\mathbf{id}.b] : C[\mathbf{id}.\mathbf{inr}(b)]}$$

Natural number type

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{Nat}_\Gamma\ \mathsf{type}}\ \textsc{nat/form}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{zero}_\Gamma : \mathbf{Nat}}\ \textsc{nat/intro/zero} \qquad \frac{\Gamma \vdash n : \mathbf{Nat}}{\Gamma \vdash \mathbf{suc}_\Gamma(n) : \mathbf{Nat}}\ \textsc{nat/intro/suc}$$

$$\frac{\begin{array}{c}\Gamma.\mathbf{Nat} \vdash A\ \mathsf{type} \\ \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])] \qquad \Gamma \vdash n : \mathbf{Nat}\end{array}}{\Gamma \vdash \mathbf{rec}_{\Gamma,A}(a_z, a_s, n) : A[\mathbf{id}.n]}\ \textsc{nat/elim}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{Nat}_\Gamma[\gamma] = \mathbf{Nat}_\Delta\ \mathsf{type}}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{zero}_\Gamma[\gamma] = \mathbf{zero}_\Delta : \mathbf{Nat}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash n : \mathbf{Nat}}{\Delta \vdash \mathbf{suc}_\Gamma(n)[\gamma] = \mathbf{suc}_\Delta(n[\gamma]) : \mathbf{Nat}}$$

$$\frac{\begin{array}{c}\Delta \vdash \gamma : \Gamma \qquad \Gamma.\mathbf{Nat} \vdash A\ \mathsf{type} \\ \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])] \qquad \Gamma \vdash n : \mathbf{Nat}\end{array}}{\Delta \vdash \mathbf{rec}(a_z, a_s, n)[\gamma] = \mathbf{rec}(a_z[\gamma], a_s[(\gamma \circ \mathbf{p}^2).\mathbf{q}[\mathbf{p}].\mathbf{q}], n[\gamma]) : A[\gamma.n[\gamma]]}$$

$$\frac{\Gamma.\mathbf{Nat} \vdash A\ \mathsf{type} \qquad \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])]}{\Gamma \vdash \mathbf{rec}(a_z, a_s, \mathbf{zero}) = a_z : A[\mathbf{id}.\mathbf{zero}]}$$

$$\frac{\begin{array}{c}\Gamma.\mathbf{Nat} \vdash A\ \mathsf{type} \\ \Gamma \vdash a_z : A[\mathbf{id}.\mathbf{zero}] \qquad \Gamma.\mathbf{Nat}.A \vdash a_s : A[\mathbf{p}^2.\mathbf{suc}(\mathbf{q}[\mathbf{p}])] \qquad \Gamma \vdash n : \mathbf{Nat}\end{array}}{\Gamma \vdash \mathbf{rec}(a_z, a_s, \mathbf{suc}(n)) = a_s[\mathbf{id}.n.\mathbf{rec}(a_z, a_s, n)] : A[\mathbf{id}.\mathbf{suc}(n)]}$$

Intensional identity types

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash \mathbf{Id}_\Gamma(A,a,b)\ \mathsf{type}}\ \textsc{id/form (itt)} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{refl}_{\Gamma,A,a} : \mathbf{Id}(A,a,a)}\ \textsc{id/intro (itt)}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : \mathbf{Id}(A,a,b) \\ \Gamma.A.A[\mathbf{p}].\mathbf{Id}(A[\mathbf{p}^2],\mathbf{q}[\mathbf{p}],\mathbf{q}) \vdash C\ \mathsf{type} \qquad \Gamma.A \vdash c : C[\mathbf{p}.\mathbf{q}.\mathbf{q}.\mathbf{refl}]\end{array}}{\Gamma \vdash \mathbf{j}_{\Gamma,A,a,b,C}(c,p) : C[\mathbf{id}.a.b.p]}\ \textsc{id/elim (itt)}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Delta \vdash \mathbf{Id}_\Gamma(A,a,b)[\gamma] = \mathbf{Id}_\Delta(A[\gamma],a[\gamma],b[\gamma])\ \mathsf{type}}\ \text{(itt)}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A}{\Delta \vdash \mathbf{refl}[\gamma] = \mathbf{refl} : \mathbf{Id}(A[\gamma],a[\gamma],a[\gamma])}\ \text{(itt)}$$

$$\frac{\begin{array}{c}\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : \mathbf{Id}(A,a,b) \\ \Gamma.A.A[\mathbf{p}].\mathbf{Id}(A[\mathbf{p}^2],\mathbf{q}[\mathbf{p}],\mathbf{q}) \vdash C\ \mathsf{type} \qquad \Gamma.A \vdash c : C[\mathbf{p}.\mathbf{q}.\mathbf{q}.\mathbf{refl}]\end{array}}{\Delta \vdash \mathbf{j}(c,p)[\gamma] = \mathbf{j}(c[(\gamma \circ \mathbf{p}).\mathbf{q}],p[\gamma]) : C[\gamma.a[\gamma].b[\gamma].p[\gamma]]}\ \text{(itt)}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma.A.A[\mathbf{p}].\mathbf{Id}(A[\mathbf{p}^2],\mathbf{q}[\mathbf{p}],\mathbf{q}) \vdash C\ \mathsf{type} \qquad \Gamma.A \vdash c : C[\mathbf{p}.\mathbf{q}.\mathbf{q}.\mathbf{refl}]}{\Gamma \vdash \mathbf{j}(c,\mathbf{refl}) = c[\mathbf{id}.a] : C[\mathbf{id}.a.a.\mathbf{refl}]}\ \text{(itt)}$$

Universes

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{U}_{\Gamma,i}\ \mathsf{type}}\ \textsc{uni/form} \qquad \frac{\Gamma \vdash a : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}_{\Gamma,i}(a)\ \mathsf{type}}\ \textsc{el/form}$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma.\mathbf{El}_i(a) \vdash b : \mathbf{U}_i}{\Gamma \vdash \mathbf{pi}_{i,\Gamma}(a,b) : \mathbf{U}_i}\ \textsc{pi/code} \qquad \frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma.\mathbf{El}_i(a) \vdash b : \mathbf{U}_i}{\Gamma \vdash \mathbf{sig}_{i,\Gamma}(a,b) : \mathbf{U}_i}\ \textsc{sig/code}$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma \vdash x : \mathbf{El}_i(a) \qquad \Gamma \vdash y : \mathbf{El}_i(a)}{\Gamma \vdash \mathbf{eq}_{i,\Gamma}(a,x,y) : \mathbf{U}_i}\ \textsc{eq/code (ett)}$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma \vdash x : \mathbf{El}_i(a) \qquad \Gamma \vdash y : \mathbf{El}_i(a)}{\Gamma \vdash \mathbf{id}_{i,\Gamma}(a,x,y) : \mathbf{U}_i}\ \textsc{id/code (itt)}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{unit}_{i,\Gamma} : \mathbf{U}_i}\ \textsc{unit/code} \qquad \frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{void}_{i,\Gamma} : \mathbf{U}_i}\ \textsc{empty/code}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{bool}_{i,\Gamma} : \mathbf{U}_i}\ \textsc{bool/code} \qquad \frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma \vdash b : \mathbf{U}_i}{\Gamma \vdash \mathbf{coprod}_{i,\Gamma}(a,b) : \mathbf{U}_i}\ \textsc{plus/code}$$

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathbf{nat}_{i,\Gamma} : \mathbf{U}_i}\ \textsc{nat/code} \qquad \frac{\vdash \Gamma\ \mathsf{cx} \qquad j < i}{\Gamma \vdash \mathbf{uni}_{\Gamma,i,j} : \mathbf{U}_i}\ \textsc{uni/code} \qquad \frac{\Gamma \vdash a : \mathbf{U}_i}{\Gamma \vdash \mathbf{lift}_{i,\Gamma}(a) : \mathbf{U}_{i+1}}$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{U}_{\Gamma,i}[\gamma] = \mathbf{U}_{\Delta,i}\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}_i}{\Delta \vdash \mathbf{El}_i(a)[\gamma] = \mathbf{El}_i(a[\gamma])\ \mathsf{type}}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}_i \qquad \Gamma.\mathbf{El}_i(a) \vdash b : \mathbf{U}_i}{\Delta \vdash \mathbf{pi}(a,b)[\gamma] = \mathbf{pi}(a[\gamma], b[(\gamma \circ \mathbf{p}).\mathbf{q}]) : \mathbf{U}_i}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}_i \qquad \Gamma.\mathbf{El}_i(a) \vdash b : \mathbf{U}_i}{\Delta \vdash \mathbf{sig}(a,b)[\gamma] = \mathbf{sig}(a[\gamma], b[(\gamma \circ \mathbf{p}).\mathbf{q}]) : \mathbf{U}_i}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}_i \qquad \Gamma \vdash x : \mathbf{El}_i(a) \qquad \Gamma \vdash y : \mathbf{El}_i(a)}{\Delta \vdash \mathbf{eq}(a,x,y)[\gamma] = \mathbf{eq}(a[\gamma], x[\gamma], y[\gamma]) : \mathbf{U}_i}\ (\textsc{ett})$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}_i \qquad \Gamma \vdash x : \mathbf{El}_i(a) \qquad \Gamma \vdash y : \mathbf{El}_i(a)}{\Delta \vdash \mathbf{id}(a,x,y)[\gamma] = \mathbf{id}(a[\gamma], x[\gamma], y[\gamma]) : \mathbf{U}_i}\ (\textsc{itt})$$

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{unit}[\gamma] = \mathbf{unit} : \mathbf{U}_i} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{void}[\gamma] = \mathbf{void} : \mathbf{U}_i} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{bool}[\gamma] = \mathbf{bool} : \mathbf{U}_i}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}_i \qquad \Gamma \vdash b : \mathbf{U}_i}{\Delta \vdash \mathbf{coprod}(a,b)[\gamma] = \mathbf{coprod}(a[\gamma], b[\gamma]) : \mathbf{U}_i} \qquad \frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \mathbf{nat}[\gamma] = \mathbf{nat} : \mathbf{U}_i}$$

$$\frac{\Delta \vdash \gamma : \Gamma \qquad j < i}{\Delta \vdash \mathbf{uni}_j[\gamma] = \mathbf{uni}_j : \mathbf{U}_i} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash a : \mathbf{U}_i}{\Delta \vdash \mathbf{lift}(a)[\gamma] = \mathbf{lift}(a[\gamma]) : \mathbf{U}_{i+1}}$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma.\mathbf{El}_i(a) \vdash b : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}_i(\mathbf{pi}(a,b)) = \mathbf{\Pi}(\mathbf{El}_i(a), \mathbf{El}_i(b))\ \mathsf{type}}$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma.\mathbf{El}_i(a) \vdash b : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}_i(\mathbf{sig}(a,b)) = \Sigma(\mathbf{El}_i(a), \mathbf{El}_i(b))\ \mathsf{type}}$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma \vdash x : \mathbf{El}_i(a) \qquad \Gamma \vdash y : \mathbf{El}_i(a)}{\Gamma \vdash \mathbf{El}_i(\mathbf{eq}(a,x,y)) = \mathbf{Eq}(\mathbf{El}_i(a), x, y)\ \mathsf{type}}\ (\text{ETT})$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma \vdash x : \mathbf{El}_i(a) \qquad \Gamma \vdash y : \mathbf{El}_i(a)}{\Gamma \vdash \mathbf{El}_i(\mathbf{id}(a,x,y)) = \mathbf{Id}(\mathbf{El}_i(a), x, y)\ \mathsf{type}}\ (\text{ITT}) \qquad \frac{}{\Gamma \vdash \mathbf{El}_i(\mathbf{unit}) = \mathbf{Unit}\ \mathsf{type}}$$

$$\frac{}{\Gamma \vdash \mathbf{El}_i(\mathbf{void}) = \mathbf{Void}\ \mathsf{type}} \qquad \frac{}{\Gamma \vdash \mathbf{El}_i(\mathbf{bool}) = \mathbf{Bool}\ \mathsf{type}}$$

$$\frac{\Gamma \vdash a : \mathbf{U}_i \qquad \Gamma \vdash b : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}_i(\mathbf{coprod}(a,b)) = \mathbf{El}_i(a) + \mathbf{El}_i(b)\ \mathsf{type}} \qquad \frac{}{\Gamma \vdash \mathbf{El}_i(\mathbf{nat}) = \mathbf{Nat}\ \mathsf{type}}$$

$$\frac{j < i}{\Gamma \vdash \mathbf{El}_i(\mathbf{uni}_j) = \mathbf{U}_j\ \mathsf{type}} \qquad \frac{\Gamma \vdash a : \mathbf{U}_i}{\Gamma \vdash \mathbf{El}_{i+1}(\mathbf{lift}(a)) = \mathbf{El}_i(a)\ \mathsf{type}}$$

This material will be published by Cambridge University Press as *Principles of Dependent Type Theory* by Carlo Angiuli and Daniel Gratzer. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © Carlo Angiuli and Daniel Gratzer 2024

B *Principles of dependent sort theory*

Throughout the book, we have presented type theories as collections of judgments and inference rules, which we regard as defining families of sets (Cx, $\mathsf{Ty}(\Gamma)$, etc.) and functions between them, subject to various equations. As discussed in Section 2.2, the well-definedness of these inference rules is quite subtle; many rules only make sense in light of equations imposed by earlier rules, leading us to introduce the discipline of presuppositions and "meta-well-typedness." Then, starting in Section 3.4, we further associated to each type theory the notions of model and homomorphism of models, and asserted without proof that the aforementioned sets and functions assemble into the *initial model* of each type theory (Theorem 3.4.5).

The goal of this appendix is to substantiate these claims, namely, to (1) provide a precise mathematical account of the method of judgments and inference rules which can account for the type theories defined in this book, and (2) establish initiality theorems for all type theories defined in that manner. We achieve these aims via Cartmell's [Car86] notion of *generalized algebraic theories* (GATs),[1] a dependently-sorted generalization of algebraic theories, although other approaches (e.g., those of Uemura [Uem21] or Haselwarter [Has21]) are also possible. The main results in this appendix were first established by Cartmell [Car86]; we adopt a more modern presentation which integrates ideas of Castellan, Clairambault, and Dybjer [CCD17], Kaposi, Kovács, and Altenkirch [KKA19], and Bezem et al. [Bez+21].

Warning B.0.1. We assume the reader is comfortable with Chapters 2 and 3, including the set model of extensional type theory (Section 3.5).

In this appendix In Appendix B.1 we introduce generalized algebraic theories, and define their signatures as contexts in a specialized type theory known as a logical framework. In Appendix B.2 we associate to each signature a category of models with an initial object, establishing Theorem 3.4.5. Finally, in Appendix B.3 we construct the initial model of our logical framework, tying up a loose end from Appendix B.1.

[1] As is now customary, we deviate from Cartmell's original notion by omitting sort equations; the resulting GATs are also known as *quotient inductive-inductive types* [KKA19].

B.1 Signatures of generalized algebraic theories

Abstracting away from the specifics of dependent type theory, our definitions of type theories consist of three kinds of data: sorts, operations, and equations. *Sorts* are the grammatical categories of a theory; the sorts of extensional type theory are the contexts, the substitutions, the types, and the terms. Sorts are populated by *operations*, which are constants or functions between sorts. Extensional type theory has many operations, including the empty context, substitution extension, and every type and term former. Finally, *equations* assert that certain sequences of operations produce the same results; for example, composition of substitutions is associative, and β-laws hold.

Many familiar mathematical objects—the *algebraic theories*—can be presented as a single sort equipped with operations and equations [Grä79]. A simple but representative example is the theory of monoids, which consists of a single sort X, two operations $e \in X$ and $\cdot : X \times X \to X$, and three equations $e \cdot x = x$, $x \cdot e = x$, and $(x \cdot y) \cdot z = x \cdot (y \cdot z)$. Monoids themselves are models of the theory of monoids, in the same sense as Definition 3.4.2: a set M, $e_M \in M$, and $\cdot_M : M \times M \to M$ such that $e_M \cdot_M x = x$, $x \cdot_M e_M = x$, and $(x \cdot_M y) \cdot_M z = x \cdot_M (y \cdot_M z)$. Monoid homomorphisms are likewise homomorphisms of models in the same sense as Definition 3.4.3: a function $f : M \to M'$ such that $f(e_M) = e_{M'}$ and $f(x \cdot_M y) = f(x) \cdot_{M'} f(y)$.

Extensional type theory is not an algebraic theory because it has multiple sorts, including the *dependent sorts* $\mathsf{Sb}(\Delta, \Gamma)$, $\mathsf{Ty}(\Gamma)$, and $\mathsf{Tm}(\Gamma, A)$ indexed by elements of other sorts $\Delta, \Gamma \in \mathsf{Cx}$ and $A \in \mathsf{Ty}(\Gamma)$. Instead, ETT and all of the other type theories discussed in this book fall into a broader class of dependently-sorted algebraic theories known as *generalized algebraic theories* or GATs [Car86]. In this section, we will consider the question of whether a collection of dependent sorts, operations, and equations constitutes a well-formed presentation, or *signature*, of a generalized algebraic theory.

Signature well-formedness is an abstraction of the meta-well-typedness issues we have already discussed extensively in the setting of ETT in Sections 2.2 and 2.3. Recall the formation and introduction rules for $\mathbf{\Pi}$-types, which stipulate operations:

$$\mathbf{\Pi} : \textstyle\prod_{\Gamma \in \mathsf{Cx}} (\sum_{A \in \mathsf{Ty}(\Gamma)} \mathsf{Ty}(\Gamma.A)) \to \mathsf{Ty}(\Gamma)$$
$$\lambda : \textstyle\prod_{\Gamma \in \mathsf{Cx}} \prod_{A \in \mathsf{Ty}(\Gamma)} \prod_{B \in \mathsf{Ty}(\Gamma.A)} \mathsf{Tm}(\Gamma.A, B) \to \mathsf{Tm}(\Gamma, \mathbf{\Pi}_\Gamma(A, B))$$

The declarations of $\mathbf{\Pi}$ and λ make no sense unless we have already declared the sorts of contexts, types, and terms, much like the declaration $e \in X$ in the signature of monoids. But the output sort of λ also mentions the $\mathbf{\Pi}$ *operation*, and thus only makes sense after $\mathbf{\Pi}$ has been declared. Moreover, the input sorts of both $\mathbf{\Pi}$ and λ refer to the context extension operation $\Gamma.A$ and likewise only make sense after its declaration.

Equations pose similar issues. Consider the following substitution calculus rules:

$$\frac{\Gamma \vdash A\,\mathsf{type}}{\Gamma \vdash A[\mathbf{id}] = A\,\mathsf{type}} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash a[\mathbf{id}] = a : A}$$

These equations reference the operations $\mathbf{id}$, $A[\gamma]$, and $a[\gamma]$, much like the monoid equation $e \cdot x = x$ references the operations $\cdot$ and e. However, in the second equation, the left-hand side appears to have sort $\mathsf{Tm}(\Gamma, A[\mathbf{id}])$, whereas the right-hand side has sort $\mathsf{Tm}(\Gamma, A)$; it is only by virtue of the first equation $A[\mathbf{id}] = A$ that these two sorts are equal and thus that the equation makes sense.

Zooming out, a well-formed GAT signature is one in which the "meta-dependent-type" of every sort, operation, and equation is well-formed. As we have seen, this intuitive idea is really quite complex: each operation creates more valid sorts for subsequent operations to populate, and each equation allows more subsequent operations and equations to be meta-well-typed. There are several ways to turn this idea into a precise mathematical definition [Car86; KKA19; Bez+21]; we will expound the *theory of signatures* approach due to Kaposi, Kovács, and Altenkirch [KKA19].

A logical framework In short, our solution is to use dependent type theory to axiomatize what it means for a GAT signature to be meta-well-typed! A type theory whose purpose is to specify the syntax of other theories is known as a *logical framework* (LF), a methodology which traces back to de Bruijn [dBru73], Martin-Löf [NPS90], and Harper, Honsell, and Plotkin [HHP93], among others.

To understand how the LF approach works, let us walk through a simple example.

Example B.1.1. The *GAT of categories* can be presented by the following signature:

$$\begin{aligned}
\Sigma_{\mathrm{cat}} \coloneqq\ & \mathsf{Ob} : \mathbf{Sort},\\
& \mathsf{Hom} : \mathsf{Ob} \to \mathsf{Ob} \to \mathbf{Sort},\\
& \mathsf{id} : (a : \mathsf{Ob}) \to \mathbf{El}(\mathsf{Hom}\ a\ a),\\
& _\circ_ : (a\ b\ c : \mathsf{Ob}) \to \mathsf{Hom}\ b\ c \to \mathsf{Hom}\ a\ b \to \mathbf{El}(\mathsf{Hom}\ a\ c),\\
& \mathsf{idl} : (a\ b : \mathsf{Ob})\ (f : \mathsf{Hom}\ a\ b) \to \mathbf{Eq}(\mathsf{Hom}\ a\ b, f, \mathsf{id}(b) \circ_{a,b,b} f),\\
& \mathsf{idr} : (a\ b : \mathsf{Ob})\ (f : \mathsf{Hom}\ a\ b) \to \mathbf{Eq}(\mathsf{Hom}\ a\ b, f, f \circ_{a,a,b} \mathsf{id}(a)),\\
& \mathsf{assoc} : (a\ b\ c\ d : \mathsf{Ob})\ (f : \mathsf{Hom}\ a\ b)\ (g : \mathsf{Hom}\ b\ c)\ (h : \mathsf{Hom}\ c\ d)\\
& \quad \to \mathbf{Eq}(\mathsf{Hom}\ a\ d, (h \circ_{b,c,d} g) \circ_{a,b,d} f, h \circ_{a,c,d} (g \circ_{a,b,c} f))
\end{aligned}$$

The GAT of categories has two sorts Ob and Hom, two operations id and _∘_, and three equations. In Example B.1.1, this signature is rendered as a context in a minimal dependent type theory with four type formers: **Sort**, **El**(−), **Π**-types, and **Eq**-types.

The context Σ_{cat} begins by postulating a variable Ob : **Sort**. Terms of type **Sort** are sorts of the GAT in question, so this variable effectively adjoins a new sort Ob of objects to the theory. Because sorts themselves have elements, **Sort** is a *universe* in the LF: for every s : **Sort** there is an LF type $\mathbf{El}(s)$ whose terms are the elements of sort s.

Next, Hom : Ob → Ob → **Sort** adjoins a dependent sort Hom indexed twice over the sort Ob; that is, Hom is not itself a sort, but for any two elements a, b of sort Ob there is a sort Hom a b : **Sort**. The rules for the LF's **Π**-types are almost the usual formation and elimination rules for **Π**-types, except that the domain is required to be a sort s : **Sort**, and application takes arguments of type $\mathbf{El}(s)$. This restriction prevents us from declaring sorts, operations, or equations indexed by anything other than the elements of a sort, and explains why the type of Hom mentions Ob and not $\mathbf{El}(\text{Ob})$.

The declarations of the operations id and _∘_ follow a similar pattern, except that they adjoin elements of sorts rather than sorts themselves. Note that their types are well-formed only in light of the previous declarations of Ob and Hom; the LF's rules for **Sort**, $\mathbf{El}(-)$, and **Π**-types ensure that these declarations are sensible.

Finally, idl, idr, and assoc adjoin equations between elements of the sort of morphisms by postulating functions into **Eq**-types. The rules for the LF's **Eq**-types are almost the usual formation and elimination rules for extensional equality types, except that as with the **Π**-types, we restrict equality to the elements of a sort s : **Sort**. Note that these **Eq**-types have equality reflection; if Example B.1.1 included any sorts indexed by morphisms, declarations in those sorts would consider $\text{id}(b) \circ_{a,b,b} f$ and f identical.

To summarize, GAT signatures are contexts in a Martin-Löf type theory with:

- a type universe **Sort** with a decoding function $\mathbf{El}(-)$,
- dependent product types $\mathbf{\Pi}(s, J)$ for s : **Sort** and J a family of types over $\mathbf{El}(s)$, with **Π**-elimination (function application), and
- extensional equality types $\mathbf{Eq}(s, x, y)$ for s : **Sort** and x, y : $\mathbf{El}(s)$, with **Eq**-elimination (equality reflection).

Note that we do not ask for **Sort** to contain any codes (besides those constructed with variables), nor do we include introduction forms (or β- or η-laws) for **Π**-types or **Eq**-types; none of these features are needed in order to specify GAT signatures.

Since our primary motivation for introducing GATs and the LF is to define the syntax of dependent type theory, it is prudent to rename the LF's contexts, types, and terms to more evocative words that avoid confusion with the sorts of ETT and ITT.

Type theory	**LF**
Contexts Γ	Signatures Σ
Substitutions γ	Substitutions σ
Types A	Judgments J
Terms a	Derivations j

Now that we are thoroughly unconfused, we can observe that all the rules considered in Chapters 2 and 4, and indeed all of the rules of any Martin-Löf type theory, can be straightforwardly translated into declarations within GAT signatures.

Example B.1.2. The *GAT of extensional type theory* can be presented by the following signature, where $\Sigma_{\mathrm{cat}}[\mathsf{cx}/\mathsf{Ob}, \mathsf{sb}/\mathsf{Hom}]$ refers to the signature of categories (Example B.1.1) with the sorts Ob and Hom renamed cx and sb respectively:

$$
\begin{aligned}
\Sigma_{\mathrm{ETT}} \coloneqq\ & \Sigma_{\mathrm{cat}}[\mathsf{cx}/\mathsf{Ob}, \mathsf{sb}/\mathsf{Hom}], \\
& \mathsf{ty} : \mathsf{cx} \to \mathbf{Sort}, \\
& \mathsf{tm} : (\Gamma : \mathsf{cx}) \to \mathsf{ty}\ \Gamma \to \mathbf{Sort}, \\
& \mathsf{tySb} : (\Gamma\ \Delta : \mathsf{cx}) \to \mathsf{sb}\ \Delta\ \Gamma \to \mathsf{ty}\ \Gamma \to \mathbf{El}(\mathsf{ty}\ \Delta), \\
& \mathsf{tmSb} : (\Gamma\ \Delta : \mathsf{cx})\ (\gamma : \mathsf{sb}\ \Delta\ \Gamma)\ (A : \mathsf{ty}\ \Gamma) \to \mathsf{tm}\ \Gamma\ A \\
& \quad \to \mathbf{El}(\mathsf{tm}\ \Delta\ (\mathsf{tySb}\ \Gamma\ \Delta\ \gamma\ A)), \\
& \mathsf{cxEmp} : \mathbf{El}(\mathsf{cx}), \\
& \mathsf{cxExt} : (\Gamma : \mathsf{cx}) \to \mathsf{ty}\ \Gamma \to \mathbf{El}(\mathsf{cx}), \\
& \mathsf{p} : (\Gamma : \mathsf{cx})\ (A : \mathsf{ty}\ \Gamma) \to \mathbf{El}(\mathsf{sb}\ (\mathsf{cxExt}\ \Gamma\ A)\ \Gamma), \\
& \mathsf{q} : (\Gamma : \mathsf{cx})\ (A : \mathsf{ty}\ \Gamma) \\
& \quad \to \mathbf{El}(\mathsf{tm}\ (\mathsf{cxExt}\ \Gamma\ A)\ (\mathsf{tySb}\ \Gamma\ (\mathsf{cxExt}\ \Gamma\ A)\ (\mathsf{p}\ \Gamma\ A)\ A)), \\
& \ldots
\end{aligned}
$$

From the GAT perspective, the set of contexts of ETT, Cx, is the set of LF derivations of the judgment $\mathbf{El}(\mathsf{cx})$ in the signature Σ_{ETT}; the set of ETT substitutions $\mathrm{Sb}(\Delta, \Gamma)$ is the set of LF derivations of $\mathbf{El}(\mathsf{sb}\ \Delta\ \Gamma)$ in the signature Σ_{ETT}; and so forth.

Exercise B.1. Choose a type former of ETT or ITT and translate its rules from Appendix A into declarations extending the GAT signature started in Example B.1.2.

Remark B.1.3. Many GAT signatures may present the same generalized algebraic theory. In Example B.1.1, it is clear that swapping the order of id and _∘_ results in a well-formed signature whose sets of judgments and derivations are identical to those of Σ_{cat}. For a more interesting example, recall from Section 2.4.2 that **Π**-types can be equivalently axiomatized via **app** or via λ^{-1}. ◇

Back where we started? Our strategy of using a dependent type theory to explain the syntax of dependent type theory is quite economical, but the reader may reasonably object that we have not actually made any progress. We stress that *this strategy is not circular*: it reduces the problem of defining ETT to the problem of defining the syntax of the LF itself, which is a different (and smaller!) type theory. It remains, however, to actually define the syntax of the LF without presupposing its own existence.

Just as the syntax of ETT can be understood as the initial model of ETT (Theorem 3.4.5), we will proceed by explicitly defining the notions of *LF algebra*, or model of the LF (analogously to Definition 3.4.2), and *homomorphism of LF algebras* (analogously to Definition 3.4.3). We will then manually construct the initial LF algebra, thereby completing not only our definition of GAT signatures—as the signatures of the initial LF algebra—but also of the syntax of extensional type theory—as derivations of the initial LF algebra in the signature Σ_{ETT}, as described in Example B.1.2.

The benefit of this approach is that it will allow us to prove the initiality theorem "once and for all": by the end of this appendix we will have proven that *every* GAT signature, and thus every Martin-Löf type theory, has an initial model.

Definition B.1.4. An *LF algebra* X consists of four families of sets:

1. a set Sig_X of *X-signatures*,
2. for each $\Sigma, \Sigma' \in \mathrm{Sig}_X$, a set $\mathrm{Sb}_X(\Sigma, \Sigma')$ of *X-substitutions from Σ to Σ'*,
3. for each $\Sigma \in \mathrm{Sig}_X$, a set $\mathrm{Jdg}_X(\Sigma)$ of *X-judgments in Σ*, and
4. for each $\Sigma \in \mathrm{Sig}_X$ and $J \in \mathrm{Jdg}_X(\Sigma)$, a set $\mathrm{Deriv}_X(\Sigma, J)$ of *X-derivations of J in Σ*,

along with two X-signature-forming operations, exactly as in the substitution calculus:

5. an *empty X-signature* $\mathbf{1}_X \in \mathrm{Sig}_X$, and
6. for each $\Sigma \in \mathrm{Sig}_X$ and $J \in \mathrm{Jdg}_X(\Sigma)$, an *$X$-signature extension* $\Sigma.{}_X J \in \mathrm{Sig}_X$,

many X-substitution-forming operations and equations:

7. every substitution-forming operation and equation of the substitution calculus, including for each $\Sigma \in \mathrm{Sig}_X$, an *identity X-substitution* $\mathbf{id}_{X,\Sigma} \in \mathrm{Sb}_X(\Sigma, \Sigma)$,

many X-judgment-forming operations and equations, including several new ones:

8. for each $\Sigma, \Sigma' \in \mathrm{Sig}_X$, $\sigma \in \mathrm{Sb}_X(\Sigma, \Sigma')$, and $J \in \mathrm{Jdg}_X(\Sigma')$, a *substituted X-judgment* $J[\sigma]_X \in \mathrm{Jdg}_X(\Sigma)$ along with functoriality equations, exactly as with types in the substitution calculus,

9. for each $\Sigma \in \mathsf{Sig}_X$, a *universe of sorts* $\mathbf{Sort}_X \in \mathsf{Jdg}_X(\Sigma)$,
10. for each $\Sigma \in \mathsf{Sig}_X$ and sort $s \in \mathsf{Deriv}_X(\Sigma, \mathbf{Sort})$, a judgment $\mathbf{El}_X(s) \in \mathsf{Jdg}_X(\Sigma)$ for the *elements of sort* s,
11. for each $\Sigma \in \mathsf{Sig}_X$, sort $s \in \mathsf{Deriv}_X(\Sigma, \mathbf{Sort})$, and $x, y \in \mathsf{Deriv}_X(\Sigma, \mathbf{El}(s))$ of sort s, an *equality judgment* $\mathbf{Eq}_{X,\Sigma}(s, x, y) \in \mathsf{Jdg}_X(\Sigma)$,
12. for each $\Sigma \in \mathsf{Sig}_X$, sort $s \in \mathsf{Deriv}_X(\Sigma, \mathbf{Sort})$, and judgment $J \in \mathsf{Jdg}_X(\Sigma.\mathbf{El}(s))$ indexed over elements of sort s, a *hypothetical judgment* $\mathbf{\Pi}_{X,\Sigma}(s, J) \in \mathsf{Jdg}_X(\Sigma)$,
13. and the expected naturality equations governing substitution into the judgments $\mathbf{Sort}_X$, $\mathbf{El}_X(s)$, $\mathbf{Eq}_X(s, x, y)$, and $\mathbf{\Pi}_X(s, J)$,

and many X-derivation-forming operations and equations, including several new ones:

14. every term-forming operation and equation of the substitution calculus, including for each $\Sigma \in \mathsf{Sig}_X$ and $J \in \mathsf{Jdg}_X(\Sigma)$, a *variable X-derivation* $\mathbf{q}_{X,\Sigma,J} \in \mathsf{Deriv}_X(\Sigma.J, J[\mathbf{p}])$,
15. for each $\Sigma \in \mathsf{Sig}_X$, sort $s \in \mathsf{Deriv}_X(\Sigma, \mathbf{Sort})$, judgment $J \in \mathsf{Jdg}_X(\Sigma.\mathbf{El}(s))$, hypothetical derivation $j \in \mathsf{Deriv}_X(\Sigma, \mathbf{\Pi}(s, J))$, and element $x \in \mathsf{Deriv}_X(\Sigma, \mathbf{El}(s))$ of sort s, an *instantiated X-derivation* $\mathbf{app}(j, x) \in \mathsf{Deriv}_X(\Sigma, J[\mathbf{id}.x])$ along with the expected naturality equation, and
16. for each $\Sigma \in \mathsf{Sig}_X$, sort $s \in \mathsf{Deriv}_X(\Sigma, \mathbf{Sort})$, $x, y \in \mathsf{Deriv}_X(\Sigma, \mathbf{El}(s))$ of sort s, and equality derivation $j \in \mathsf{Deriv}_X(\Sigma, \mathbf{Eq}(s, x, y))$, the *equality reflection* equation asserting $x = y$.

Definition B.1.5. Given two LF algebras X, Y, a *homomorphism of LF algebras* $F : X \to Y$ consists of the following data:

1. a function $\mathsf{Sig}_F : \mathsf{Sig}_X \to \mathsf{Sig}_Y$,
2. for each $\Sigma, \Sigma' \in \mathsf{Sig}_X$, a function
$$\mathsf{Sb}_F(\Sigma, \Sigma') : \mathsf{Sb}_X(\Sigma, \Sigma') \to \mathsf{Sb}_Y(\mathsf{Sig}_F(\Sigma), \mathsf{Sig}_F(\Sigma'))$$
3. for each $\Sigma \in \mathsf{Sig}_X$, a function $\mathsf{Jdg}_F(\Sigma) : \mathsf{Jdg}_X(\Sigma) \to \mathsf{Jdg}_Y(\mathsf{Sig}_F(\Sigma))$, and
4. for each $\Sigma \in \mathsf{Sig}_X$ and $J \in \mathsf{Jdg}_X(\Sigma)$, a function
$$\mathsf{Deriv}_F(\Sigma, J) : \mathsf{Deriv}_X(\Sigma, J) \to \mathsf{Deriv}_Y(\mathsf{Sig}_F(\Sigma), \mathsf{Jdg}_F(\Sigma)(J))$$

5. such that the four functions above send every LF algebra operation of X to the corresponding LF algebra operation of Y.

Notation B.1.6. When unambiguous, we will write F instead of Sig_F, $\mathsf{Sb}_F(\Sigma, \Sigma')$, $\mathsf{Jdg}_F(\Sigma)$, or $\mathsf{Deriv}_F(\Sigma, J)$.

LF algebras and homomorphisms assemble into a *category of LF algebras* **LFAlg**.

Theorem B.1.7. *There exists an* initial LF algebra $\mathcal{I}$ *in* **LFAlg***; that is, there is an LF algebra $\mathcal{I}$ such that for any LF algebra X, there is a unique homomorphism $\mathcal{I} \to X$.*

The proof of Theorem B.1.7 is somewhat involved, so we will defer it to Appendix B.3 and continue our discussion of GATs under the assumption that $\mathcal{I}$ exists, starting with, at long last, a precise definition of GAT signature.

Definition B.1.8. A *GAT signature* is an element of $\mathsf{Sig}_{\mathcal{I}}$. A *homomorphism of GAT signatures* $\Sigma \to \Sigma'$ is an element of $\mathsf{Sb}_{\mathcal{I}}(\Sigma, \Sigma')$.

B.2 *The category of models of a GAT*

Now that we have a definition of generalized algebraic theories, we can turn to studying their properties. In this section, we will show that every GAT signature Σ induces a category of models $\mathsf{Mod}(\Sigma)$—and in particular, notions of model and homomorphism of models—containing an initial object $\mathsf{Syn}(\Sigma)$. In the case of Σ_{cat}, one obtains the 1-category of small categories; in the case of Σ_{ETT}, we recover exactly our earlier definitions of model (Definition 3.4.2), homomorphism of models (Definition 3.4.3), and syntactic model (Definition 3.4.4) along with its initiality property (Theorem 3.4.5).

The results in this section all proceed by constructing (non-initial) LF algebras. Why? Although we have not phrased it as such, the collection of GAT signatures is "inductively defined" in the sense of being a carrier set for an initial algebra, much like the natural numbers. As a result, maps out of that set are defined "by induction," or equivalently—as in Section 2.5.4—by constructing (displayed) algebras.

Many of these constructions are rather lengthy and tedious. We focus on the most salient points and leave the reader to work out the remaining details by themselves or by consulting Kaposi, Kovács, and Altenkirch [KKA19] or Kovács [Kov22].

B.2.1 Models of a signature

We begin by defining a map sending each GAT signature Σ to its set of models. Informally, a model of Σ consists of an (indexed) set for each sort of Σ and a function for each operation of Σ, validating each equation of Σ. Because GAT signatures are the signatures of the initial LF algebra $\mathcal{I}$, we will obtain the desired map by first defining an LF algebra $\mathcal{M}od$ and then taking the signature component of the unique LF algebra homomorphism $!_{\mathcal{M}od} : \mathcal{I} \to \mathcal{M}od$, so that the set of models of $\Sigma \in \mathsf{Sig}_{\mathcal{I}}$ is $\mathsf{Sig}_{!_{\mathcal{M}od}}(\Sigma)$.

As in Section 3.5, we will encounter size issues. Consider the simplest nontrivial GAT signature $(X : \mathbf{Sort})$, which presents the theory of a single sort. Models of this signature are exactly sets, so its set of models must be the "set of all (small) sets." We once again circumvent this issue by working relative to a choice of Grothendieck universes (Section 3.5.1), although in this case we will only need two.

Axiom B.2.1. We assume a pair of Grothendieck universes $\mathcal{V}_0 \in \mathcal{V}_1$.

To define the LF algebra $\mathcal{M}od$, we must first choose its sets of signatures, substitutions, judgments, and derivations. Recalling that we want $\mathsf{Sig}_{!_{\mathcal{M}od}} : \mathsf{Sig}_{\mathcal{I}} \to \mathsf{Sig}_{\mathcal{M}od}$ to send signatures to their sets of models, and that the set of models of $(X : \mathbf{Sort})$ must be a Grothendieck universe, we choose the larger of our two Grothendieck universes $\mathcal{V}_1$ to be the set of $\mathcal{M}od$-signatures.

$$\mathsf{Sig}_{\mathcal{M}od} \coloneqq \mathcal{V}_1$$

The entire remainder of the substitution calculus of $\mathcal{M}od$ is identical to the substitution calculus of the set model defined in Section 3.5.2, except that $\mathcal{V}_\omega$ is replaced by $\mathcal{V}_1$: $\mathcal{M}od$-substitutions are functions, $\mathcal{M}od$-judgments are indexed families of $\mathcal{V}_1$-small sets, $\mathcal{M}od$-derivations are indexed families of elements, the empty $\mathcal{M}od$-signature is $\{\star\}$, $\mathcal{M}od$-signature extension $X.J$ is $\sum_{x \in X} J(x)$, and so forth.

$$\begin{aligned}
&\mathsf{Sb}_{\mathcal{M}od}(X, Y) \coloneqq X \to Y && (X, Y \in \mathsf{Sig}_{\mathcal{M}od}) \\
&\mathsf{Jdg}_{\mathcal{M}od}(X) \coloneqq X \to \mathcal{V}_1 && (X \in \mathsf{Sig}_{\mathcal{M}od}) \\
&\mathsf{Deriv}_{\mathcal{M}od}(X, J) \coloneqq \textstyle\prod_{x \in X} J(x) && (X \in \mathsf{Sig}_{\mathcal{M}od}, J \in \mathsf{Jdg}_{\mathcal{M}od}(X))
\end{aligned}$$

It remains to interpret the connectives $\mathbf{Sort}$, $\mathbf{El}(s)$, $\mathbf{\Pi}(s, J)$, and $\mathbf{Eq}(s, x, y)$, but these also directly mirror the set-theoretic interpretations of universes, $\mathbf{\Pi}$-types, and $\mathbf{Eq}$-types detailed in Section 3.5.3. We define $\mathbf{Sort}_{\mathcal{M}od}$ to be the smaller of our two

Grothendieck universes $\mathcal{V}_0$, and $\mathbf{El}_{\mathcal{M}od}$ to be the inclusion $\mathcal{V}_0 \subseteq \mathcal{V}_1$.

$$\begin{aligned}
&\mathbf{Sort}_{\mathcal{M}od} : \textstyle\prod_{X \in \mathcal{V}_1} \mathsf{Jdg}_{\mathcal{M}od}(X) \\
&\mathbf{Sort}_{\mathcal{M}od}\ X\ x := \mathcal{V}_0 \\
&\mathbf{El}_{\mathcal{M}od} : \textstyle\prod_{X \in \mathcal{V}_1} \mathsf{Deriv}_{\mathcal{M}od}(X, \mathbf{Sort}_{\mathcal{M}od}) \to \mathsf{Jdg}_{\mathcal{M}od}(X) \\
&\mathbf{El}_{\mathcal{M}od}\ X\ s := s
\end{aligned}$$

We leave the remaining details of $\mathbf{\Pi}(s, J)$ and $\mathbf{Eq}(s, x, y)$ to the reader.

Lemma B.2.2. *$\mathcal{M}od$ is an LF algebra.*

By Theorem B.1.7 we obtain a unique homomorphism of LF algebras $!_{\mathcal{M}od} : \mathcal{I} \to \mathcal{M}od$, whose signature component assigns to each $\Sigma \in \mathsf{Sig}_{\mathcal{I}}$ its set of models.

Definition B.2.3. Given a GAT signature Σ, its *set of models* $\mathsf{Mod}(\Sigma)$ is $\mathsf{Sig}_{!_{\mathcal{M}od}}(\Sigma)$.

Example B.2.4. We may compute $\mathsf{Mod}(\Sigma_{\mathsf{cat}})$ by replacing every LF algebra operation of $\mathcal{I}$ in Σ_{cat} (Example B.1.1) by the corresponding operation of $\mathcal{M}od$, obtaining the following iterated indexed coproduct (up to isomorphism, for readability):

$$\begin{aligned}
\mathsf{Mod}(\Sigma_{\mathsf{cat}}) \cong\ & \textstyle\sum_{\mathsf{Ob} \in \mathcal{V}_0} \\
& \textstyle\sum_{\mathsf{Hom} \in \mathsf{Ob} \to \mathsf{Ob} \to \mathcal{V}_0} \\
& \textstyle\sum_{\mathsf{id} \in (\prod_{a \in \mathsf{Ob}} \mathsf{Hom}\ a\ a)} \\
& \textstyle\sum_{_\circ_ \in (\prod_{a,b,c \in \mathsf{Ob}} \mathsf{Hom}\ b\ c \to \mathsf{Hom}\ a\ b \to \mathsf{Hom}\ a\ c)} \\
& \quad (\textstyle\prod_{a,b \in \mathsf{Ob}} \prod_{f \in \mathsf{Hom}\ a\ b} \{\star \mid f = \mathsf{id}(b) \circ_{a,b,b} f\}) \\
& \times (\textstyle\prod_{a,b \in \mathsf{Ob}} \prod_{f \in \mathsf{Hom}\ a\ b} \{\star \mid f = f \circ_{a,a,b} \mathsf{id}(a)\}) \\
& \times (\textstyle\prod_{a,b,c,d \in \mathsf{Ob}} \prod_{f \in \mathsf{Hom}\ a\ b} \prod_{g \in \mathsf{Hom}\ b\ c} \prod_{h \in \mathsf{Hom}\ c\ d} \\
& \qquad \{\star \mid (h \circ_{b,c,d} g) \circ_{a,b,d} f = h \circ_{a,c,d} (g \circ_{a,b,c} f)\})
\end{aligned}$$

Thus $\mathsf{Mod}(\Sigma_{\mathsf{cat}})$ is the set of all $\mathcal{V}_0$-small categories.

B.2.2 Homomorphisms of models

Next we associate to every pair of models $X, Y \in \mathsf{Mod}(\Sigma)$ a set $\mathsf{hom}_{\mathsf{Mod}(\Sigma)}(X, Y)$ of homomorphisms of models, thereby upgrading $\mathsf{Mod}(\Sigma)$ to a *category* of models of Σ.

Once again we obtain $\mathsf{hom}(-,-)$ via an LF algebra $\mathcal{H}om$. However, whereas $\mathsf{Mod}(-)$ was a non-dependent function $\mathsf{Sig}_{\mathcal{I}} \to \mathcal{V}_1$, $\mathsf{hom}_{\mathsf{Mod}(-)}(-,-)$ is a *dependent* function $\prod_{\Sigma \in \mathsf{Sig}_{\mathcal{I}}} \mathsf{Mod}(\Sigma) \times \mathsf{Mod}(\Sigma) \to \mathcal{V}_1$, forcing $\mathcal{H}om$ to be a *displayed* LF algebra

over $\mathcal{I}$. We represent this formally as an LF algebra each of whose carrier sets is an indexed coproduct over the corresponding carrier set of $\mathcal{I}$, and each of whose operations has the corresponding $\mathcal{I}$ operation as its first projection; we have previously encountered such encodings in Sections 2.5.4 and 6.7.

In particular, we will eventually define $\hom_{\mathsf{Mod}(\Sigma)}(-,-)$ as the second projection of the image of $\Sigma \in \mathsf{Sig}_{\mathcal{I}}$ in $\mathcal{H}om$, so the collection of $\mathcal{H}om$-signatures must be:

$$\mathsf{Sig}_{\mathcal{H}om} := \textstyle\sum_{\Sigma \in \mathsf{Sig}_{\mathcal{I}}} \mathsf{Mod}(\Sigma) \times \mathsf{Mod}(\Sigma) \to \mathcal{V}_1$$

The $\mathcal{H}om$-substitutions from (Σ, H) to (Σ', H') over the $\mathcal{I}$-substitution $\sigma \in \mathsf{Sb}_{\mathcal{I}}(\Sigma, \Sigma')$ are then functorial actions sending homomorphisms of Σ-models $f \in H(X, Y)$ to homomorphisms of the Σ'-models induced by $\mathsf{Sb}_{!_{\mathcal{M}od}}(\Sigma, \Sigma')(\sigma)$, which we abbreviate $!_{\mathcal{M}od}(\sigma)$ in accordance with Notation B.1.6.

$$\begin{aligned}&\mathsf{Sb}_{\mathcal{H}om}((\Sigma, H), (\Sigma', H')) := \\ &\quad \textstyle\sum_{\sigma \in \mathsf{Sb}_{\mathcal{I}}(\Sigma,\Sigma')} \prod_{X,Y \in \mathsf{Mod}(\Sigma)} H(X, Y) \to H'(!_{\mathcal{M}od}(\sigma)(X), !_{\mathcal{M}od}(\sigma)(Y))\end{aligned}$$

Finally, the $\mathcal{H}om$-judgments in (Σ, H) over the $\mathcal{I}$-judgment $J \in \mathsf{Jdg}_{\mathcal{I}}(\Sigma)$ are families of sets indexed by homomorphisms $H(X, Y)$ and the interpretations of J in X and Y, and $\mathcal{H}om$-derivations are indexed families of elements of those sets.

$$\begin{aligned}&\mathsf{Jdg}_{\mathcal{H}om}(\Sigma, H) := \\ &\quad \textstyle\sum_{J \in \mathsf{Jdg}_{\mathcal{I}}(\Sigma)} \prod_{X,Y \in \mathsf{Mod}(\Sigma)} H(X, Y) \to !_{\mathcal{M}od}(J)(X) \times !_{\mathcal{M}od}(J)(Y) \to \mathcal{V}_1 \\ &\mathsf{Deriv}_{\mathcal{H}om}((\Sigma, H), (J, R)) := \\ &\quad \textstyle\sum_{j \in \mathsf{Deriv}_{\mathcal{I}}(\Sigma,J)} \prod_{X,Y \in \mathsf{Mod}(\Sigma)} \prod_{f \in H(X,Y)} R(f)(!_{\mathcal{M}od}(j)(X), !_{\mathcal{M}od}(j)(Y))\end{aligned}$$

We now turn our attention to some of the LF algebra operations of $\mathcal{H}om$.

Notation B.2.5. The inputs and outputs of these LF algebra operations are dependent pairs of data from $\mathcal{I}$ and displayed data over $\mathcal{I}$. For readability we will define the two projections of each operation separately; as in Notation 6.7.5, we write $-^\circ$ for the first projection (the $\mathcal{I}$ component) and $-^\bullet$ for the second projection.

The signature-forming operations of $\mathcal{H}om$ form the core of our definition of homomorphisms. The first two clauses below state that the empty $\mathcal{H}om$-signature over the empty $\mathcal{I}$-signature is the constant function $(X, Y) \mapsto \{\star\}$; in short, there is exactly one homomorphism $\hom_{\mathsf{Mod}(\mathbf{1})}(X, Y)$. The next two clauses define homomorphisms $\hom_{\mathsf{Mod}(\Sigma.J)}((X, A), (Y, B))$ as pairs of $f \in \hom_{\mathsf{Mod}(\Sigma)}(X, Y)$ along with an element of

the relevant $\mathcal{H}om$-judgment associated to J, over f and between A and B.

$$\begin{aligned}
&\mathbf{1}_{\mathcal{H}om}{}^{\circ} := \mathbf{1}_{\mathcal{I}} \\
&\mathbf{1}_{\mathcal{H}om}{}^{\bullet}(X, Y) := \{\star\} \\
&(\Sigma._{\mathcal{H}om} J)^{\circ} := \Sigma^{\circ}._{\mathcal{I}} J^{\circ} \\
&(\Sigma._{\mathcal{H}om} J)^{\bullet}((X, A), (Y, B)) := \textstyle\sum_{f \in \Sigma^{\bullet}(X,Y)} J^{\bullet}(f)(A, B)
\end{aligned}$$

But what is the set $J^{\bullet}(f)(A, B)$? Suppose that $J = \mathbf{Sort}_{\mathcal{H}om}$, in which case we are extending the GAT signature Σ by a new sort, and A, B are sets modeling this sort. Elements of $\hom_{\mathsf{Mod}(\Sigma.\mathbf{Sort})}((X, A), (Y, B))$ should be pairs of a homomorphism $f \in \hom_{\mathsf{Mod}(\Sigma)}(X, Y)$ and a function $A \to B$, and indeed we define $J^{\bullet}(f)(A, B)$ in this case to be the set of functions $A \to B$.

$$\begin{aligned}
&\mathbf{Sort}_{\mathcal{H}om}{}^{\circ} := \mathbf{Sort}_{\mathcal{I}} \\
&\mathbf{Sort}_{\mathcal{H}om}{}^{\bullet}(f)(A, B) := A \to B
\end{aligned}$$

If instead $J = \mathbf{El}_{\mathcal{H}om}(s)$, then we are extending the GAT signature Σ by an element of sort s°, and A and B are elements of the X and Y interpretations of that sort respectively. Elements of $\hom_{\mathsf{Mod}(\Sigma.\mathbf{El}(s^{\circ}))}((X, A), (Y, B))$ should in this case be pairs of a homomorphism $f \in \hom_{\mathsf{Mod}(\Sigma)}(X, Y)$ and an equation stating that f sends A to B.

$$\begin{aligned}
&\mathbf{El}_{\mathcal{H}om}(s)^{\circ} := \mathbf{El}_{\mathcal{I}}(s^{\circ}) \\
&\mathbf{El}_{\mathcal{H}om}(s)^{\bullet}(f)(x, y) := \{\star \mid s^{\bullet}(f)(x) = y\}
\end{aligned}$$

For hypothetical judgments $\mathbf{\Pi}_{\mathcal{H}om}(s, J)$, the set is defined pointwise by recursively applying $J^{\bullet}$ after applying the appropriate functorial actions, producing either a family of functions between sets (in the case of dependent sorts) or a family of equations (in the case of dependent operations). Finally, equations in GAT signatures impose no additional requirements on homomorphisms.

$$\begin{aligned}
&\mathbf{\Pi}_{\mathcal{H}om}(s, J)^{\circ} := \mathbf{\Pi}_{\mathcal{I}}(s^{\circ}, J^{\circ}) \\
&\mathbf{\Pi}_{\mathcal{H}om}(s, J)^{\bullet}(f)(h, k) := \textstyle\prod_{x \in !_{\mathcal{M}od(s^{\circ})}(X)} J^{\bullet}(f, \star)(h(x), k(s^{\bullet}(f)(x))) \quad (f \in H(X, Y)) \\
&\mathbf{Eq}_{\mathcal{H}om}(s, x, x')^{\circ} := \mathbf{Eq}_{\mathcal{I}}(s^{\circ}, x^{\circ}, x'^{\circ}) \\
&\mathbf{Eq}_{\mathcal{H}om}(s, x, x')^{\bullet}(f)(_, _) := \{\star\}
\end{aligned}$$

The above clauses suffice to reconstruct the remaining LF algebra structure, which can also be found in Section 5 of Kaposi, Kovács, and Altenkirch [KKA19]. Because each component of $\mathcal{H}om$ is a pair whose first projection is the corresponding component of $\mathcal{I}$, we obtain not only an LF algebra $\mathcal{H}om$ but a projection homomorphism $\mathcal{H}om \to \mathcal{I}$.

Lemma B.2.6. *$\mathcal{H}om$ is an LF algebra equipped with a homomorphism $-^\circ : \mathcal{H}om \to \mathcal{I}$.*

As before, we obtain a unique homomorphism of LF algebras $!_{\mathcal{H}om} : \mathcal{I} \to \mathcal{H}om$. Unfolding the definition of $\mathsf{Sig}_{\mathcal{H}om}$, for any $\Sigma \in \mathsf{Sig}_{\mathcal{I}}$,

$$\mathsf{Sig}_{!_{\mathcal{H}om}}(\Sigma) \in \textstyle\sum_{\Sigma' \in \mathsf{Sig}_{\mathcal{I}}} \mathsf{Mod}(\Sigma') \times \mathsf{Mod}(\Sigma') \to \mathcal{V}_1$$

We have $\mathsf{Sig}_{!_{\mathcal{H}om}}(\Sigma)^\circ = \Sigma$ because $-^\circ \circ !_{\mathcal{H}om}$ is an LF algebra homomorphism $\mathcal{I} \to \mathcal{I}$ and thus, by Theorem B.1.7, the identity homomorphism. Thus $\mathsf{Sig}_{!_{\mathcal{H}om}}(\Sigma)^\bullet : \mathsf{Mod}(\Sigma) \times \mathsf{Mod}(\Sigma) \to \mathcal{V}_1$ and we can define homomorphisms of models of Σ as follows:

Definition B.2.7. Given a GAT signature Σ and models $X, Y \in \mathsf{Mod}(\Sigma)$, the set of *homomorphisms of models* $\hom_{\mathsf{Mod}(\Sigma)}(X, Y)$ is $\mathsf{Sig}_{!_{\mathcal{H}om}}(\Sigma)^\bullet(X, Y)$.

With more work, one can show that models and homomorphisms form a category.

Lemma B.2.8. *Given a GAT signature Σ, the sets $\mathsf{Mod}(\Sigma)$ and $\hom_{\mathsf{Mod}(\Sigma)}(-,-)$ form its* category of models*, which we also write $\mathsf{Mod}(\Sigma)$.*

Proof sketch. We construct the identity and composition operations as displayed LF algebras, much like $\mathcal{H}om$ itself. We verify the associativity and unit laws by induction, which is to say, also by constructing displayed LF algebras. We refer the reader to Kaposi, Kovács, and Altenkirch [KKA19] for details. □

Example B.2.9. In Example B.2.4, we saw that $\mathsf{Mod}(\Sigma_{\mathsf{cat}})$ is the set of $\mathcal{V}_0$-small categories. For any $C, D \in \mathsf{Mod}(\Sigma_{\mathsf{cat}})$, if we compute $\hom_{\mathsf{Mod}(\Sigma_{\mathsf{cat}})}(C, D)$ by replacing each LF algebra operation in Σ_{cat} with the corresponding operation of $\mathcal{H}om$, we will see that $\hom_{\mathsf{Mod}(\Sigma_{\mathsf{cat}})}(C, D)$ is the set of functors $C \to D$.

B.2.3 The initiality theorem

Our last obligation is to show that the category $\mathsf{Mod}(\Sigma)$ has an initial object, i.e., to construct a *syntactic model* $\mathsf{Syn}(\Sigma)$ with the property that for any $X \in \mathsf{Mod}(\Sigma)$ there exists a unique homomorphism of models $\mathsf{Syn}(\Sigma) \to X$. (We obtain Theorem 3.4.5, the initiality theorem for ETT, as an immediate corollary.) We have already seen the main idea of our construction in Example B.1.2: $\mathsf{Syn}(\Sigma)$ will interpret each closed sort s as the set of derivations $\mathsf{Deriv}_{\mathcal{I}}(\Sigma, \mathbf{El}(s))$ in the initial LF algebra.

Remark B.2.10. In what sense is $\mathsf{Syn}(\Sigma)$ the "syntactic" model of Σ? In an abstract sense, initiality is what it means to be syntactic: as discussed in Section 3.4, the syntax of any theory, by definition, is the "least" model generated by the rules of the theory. In a more literal sense, the carrier sets of $\mathsf{Syn}(\Sigma)$ are defined as sets of $\mathcal{I}$-derivations, which we will in turn define in Appendix B.3 as equivalence classes of terms. ◇

As in the previous section, we start by defining a displayed LF algebra $\mathcal{S}yn$ whose signatures contain the data of interest. One new complication is that the signature Σ appears twice in the types of our construction, in $\mathsf{Mod}(\Sigma)$ and in $\mathsf{Deriv}_{\mathcal{I}}(\Sigma, \mathbf{El}(s))$. In order to proceed, we generalize the situation by fixing $\Sigma \in \mathsf{Sig}_{\mathcal{I}}$ and defining $\mathcal{S}yn$-signatures as pairs of an $\mathcal{I}$-signature $\Upsilon \in \mathsf{Sig}_{\mathcal{I}}$ together with a function sending each $\upsilon \in \mathsf{Sb}_{\mathcal{I}}(\Sigma, \Upsilon)$ to an element of $\mathsf{Mod}(\Upsilon)$. At the end of the construction we will obtain a $\mathcal{S}yn$-signature for which $\Upsilon = \Sigma$, allowing us to define $\mathsf{Syn}(\Sigma)$ by setting $\upsilon = \mathbf{id}_{\Sigma}$.

$$\mathsf{Sig}_{\mathcal{S}yn} := \textstyle\sum_{\Upsilon \in \mathsf{Sig}_{\mathcal{I}}} \mathsf{Sb}_{\mathcal{I}}(\Sigma, \Upsilon) \to \mathsf{Mod}(\Upsilon)$$

$$\mathsf{Sb}_{\mathcal{S}yn}((\Upsilon, \Phi), (\Upsilon', \Phi')) :=$$
$$\textstyle\sum_{\upsilon' \in \mathsf{Sb}_{\mathcal{I}}(\Upsilon, \Upsilon')} \prod_{\upsilon \in \mathsf{Sb}_{\mathcal{I}}(\Sigma, \Upsilon)} \{\star \mid {!_{\mathcal{M}od}}(\upsilon')(\Phi(\upsilon)) = \Phi'(\upsilon' \circ \upsilon)\}$$

$$\mathsf{Jdg}_{\mathcal{S}yn}(\Upsilon, \Phi) := \textstyle\sum_{J \in \mathsf{Jdg}_{\mathcal{I}}(\Upsilon)} \prod_{\upsilon \in \mathsf{Sb}_{\mathcal{I}}(\Sigma, \Upsilon)} \mathsf{Deriv}_{\mathcal{I}}(\Sigma, J[\upsilon]) \to {!_{\mathcal{M}od}}(J)(\Phi(\upsilon))$$

$$\mathsf{Deriv}_{\mathcal{S}yn}((\Upsilon, \Phi), (J, \Psi)) :=$$
$$\textstyle\sum_{j \in \mathsf{Deriv}_{\mathcal{I}}(\Upsilon, J)} \prod_{\upsilon \in \mathsf{Sb}_{\mathcal{I}}(\Sigma, \Upsilon)} \{\star \mid \Psi(\upsilon)(j[\upsilon]) = {!_{\mathcal{M}od}}(j)(\Phi(\upsilon))\}$$

The first projections of the LF algebra operations of $\mathcal{S}yn$ are, once again, the corresponding operations of $\mathcal{I}$. Unfolding definitions, $\mathbf{Sort}^{\bullet}_{\mathcal{S}yn}$ must be a function which assigns a $\mathcal{V}_0$-small set to each $\upsilon \in \mathsf{Sb}_{\mathcal{I}}(\Sigma, \Upsilon)$ and $s \in \mathsf{Deriv}_{\mathcal{I}}(\Sigma, \mathbf{Sort})$; as described above, we choose the set of $\mathcal{I}$-derivations of $\mathbf{El}(s)$ in Σ.

$$\mathbf{Sort}_{\mathcal{S}yn}{}^{\circ} := \mathbf{Sort}_{\mathcal{I}}$$
$$\mathbf{Sort}_{\mathcal{S}yn}{}^{\bullet}(\upsilon)(s) := \mathsf{Deriv}_{\mathcal{I}}(\Sigma, \mathbf{El}(s)) \qquad (s \in \mathsf{Deriv}_{\mathcal{I}}(\Sigma, \mathbf{Sort}))$$

As for $\mathbf{El}_{\mathcal{S}yn}(s)^{\bullet}$ where $s \in \mathsf{Deriv}_{\mathcal{S}yn}((\Upsilon, \Phi), \mathbf{Sort})$, this is a function which given $\upsilon \in \mathsf{Sb}_{\mathcal{I}}(\Sigma, \Upsilon)$ and $x \in \mathsf{Deriv}_{\mathcal{I}}(\Sigma, \mathbf{El}(s^{\circ}[\upsilon]))$ must produce an element of ${!_{\mathcal{M}od}}(s^{\circ})(\Phi(\upsilon))$; x is such an element in light of $s^{\bullet}(\upsilon) \in \{\star \mid \mathsf{Deriv}_{\mathcal{I}}(\Sigma, \mathbf{El}(s^{\circ}[\upsilon])) = {!_{\mathcal{M}od}}(s^{\circ})(\Phi(\upsilon))\}$.

$$\mathbf{El}_{\mathcal{S}yn}(s)^{\circ} := \mathbf{El}_{\mathcal{I}}(s^{\circ})$$
$$\mathbf{El}_{\mathcal{S}yn}(s)^{\bullet}(\upsilon)(x) := x$$

We refer the reader to Kaposi, Kovács, and Altenkirch [KKA19] for the remaining LF algebra structure, which once again admits a projection homomorphism to $\mathcal{I}$.

Lemma B.2.11. *$\mathcal{S}yn$ is an LF algebra equipped with a homomorphism $-^{\circ} : \mathcal{S}yn \to \mathcal{I}$.*

We again obtain a unique homomorphism of LF algebras ${!_{\mathcal{S}yn}} : \mathcal{I} \to \mathcal{S}yn$, and

$$\mathsf{Sig}_{{!_{\mathcal{S}yn}}}(\Sigma) \in \textstyle\sum_{\Upsilon \in \mathsf{Sig}_{\mathcal{I}}} \mathsf{Sb}_{\mathcal{I}}(\Sigma, \Upsilon) \to \mathsf{Mod}(\Upsilon)$$

Once again, $\mathsf{Sig}_{{!_{\mathcal{S}yn}}}(\Sigma)^{\circ} = \Sigma$ because $-^{\circ} \circ {!_{\mathcal{S}yn}}$ is necessarily the identity homomorphism $\mathcal{I} \to \mathcal{I}$, and thus $\mathsf{Sig}_{{!_{\mathcal{S}yn}}}(\Sigma)^{\bullet} : \mathsf{Sb}_{\mathcal{I}}(\Sigma, \Sigma) \to \mathsf{Mod}(\Sigma)$.

Definition B.2.12. Given a GAT signature Σ, its *syntactic model* $\mathsf{Syn}(\Sigma) \in \mathsf{Mod}(\Sigma)$ is $\mathsf{Sig}_{!_{\mathcal{S}yn}}(\Sigma)^\bullet(\mathbf{id}_\Sigma)$.

Example B.2.13. The *GAT of natural numbers* can be presented by the signature $\Sigma_{\mathrm{nat}} := X : \mathbf{Sort}, z : \mathbf{El}(X), s : X \to \mathbf{El}(X)$. Unfolding the definitions in this section, models of Σ_{nat} are triples of a $\mathcal{V}_0$-small set N along with $z_N \in N$ and $s_N : N \to N$, and its syntactic model is the triple $(\mathsf{Deriv}_{\mathcal{I}}(\Sigma_{\mathrm{nat}}, \mathbf{El}(X)), z, \mathbf{app}(s, -))$.

Remark B.2.14. In Section 3.6, we used a similar idea to define a context $\Gamma_{\mathrm{SK}} := A : \mathbf{U}, \ldots$ in which ETT terms $\mathsf{Tm}(\Gamma_{\mathrm{SK}}, A)$ are in bijection with SK combinator terms. ⋄

It remains to show that $\mathsf{Syn}(\Sigma)$ is the initial model of Σ. Here we depart from the strategy of Kaposi, Kovács, and Altenkirch [KKA19]. Instead, we simply:

1. define $\mathrm{rec}_X \in \hom_{\mathsf{Mod}(\Sigma)}(\mathsf{Syn}(\Sigma), X)$ for every $X \in \mathsf{Mod}(\Sigma)$, and
2. prove that for every $f, g \in \hom_{\mathsf{Mod}(\Sigma)}(\mathsf{Syn}(\Sigma), X)$ we have $f = g$.

Both steps once again involve constructing displayed LF algebras over $\mathcal{I}$. In step (1), as in $\mathcal{S}yn$, we fix $\Sigma \in \mathsf{Sig}_{\mathcal{I}}$ and $X \in \mathsf{Mod}(\Sigma)$ and generalize the situation over $\Upsilon \in \mathsf{Sig}_{\mathcal{I}}$ and $v \in \mathsf{Sb}_{\mathcal{I}}(\Sigma, \Upsilon)$. In particular, we define $\mathcal{R}ec$-signatures as pairs of an Υ and a function sending each v to a homomorphism between the Υ-models obtained by applying $!_{\mathcal{M}od}(v) : \mathsf{Mod}(\Sigma) \to \mathsf{Mod}(\Upsilon)$ to $\mathsf{Syn}(\Sigma)$ and X.

$$\mathsf{Sig}_{\mathcal{R}ec} := \textstyle\sum_{\Upsilon \in \mathsf{Sig}_{\mathcal{I}}} \prod_{v \in \mathsf{Sb}_{\mathcal{I}}(\Sigma,\Upsilon)} \hom_{\mathsf{Mod}(\Upsilon)}(!_{\mathcal{M}od}(v)(\mathsf{Syn}(\Sigma)), !_{\mathcal{M}od}(v)(X))$$

$$\mathsf{Sb}_{\mathcal{R}ec}((\Upsilon, \Phi), (\Upsilon', \Phi')) :=$$
$$\textstyle\sum_{v' \in \mathsf{Sb}_{\mathcal{I}}(\Upsilon,\Upsilon')} \prod_{v \in \mathsf{Sb}_{\mathcal{I}}(\Sigma,\Upsilon)} \{\star \mid !_{\mathcal{H}om}(v')^\bullet(\Phi(v)) = \Phi'(v' \circ v)\}$$

$$\mathsf{Jdg}_{\mathcal{R}ec}(\Upsilon, \Phi) := \textstyle\sum_{J \in \mathsf{Jdg}_{\mathcal{I}}(\Upsilon)} \prod_{v \in \mathsf{Sb}_{\mathcal{I}}(\Sigma,\Upsilon)} \prod_{j \in \mathsf{Deriv}_{\mathcal{I}}(\Sigma, J[v])}$$
$$!_{\mathcal{H}om}(J)^\bullet(\Phi(v))(!_{\mathcal{M}od}(j)(\mathsf{Syn}(\Sigma)), !_{\mathcal{M}od}(j)(X))$$

$$\mathsf{Deriv}_{\mathcal{R}ec}((\Upsilon, \Phi), (J, \Psi)) :=$$
$$\textstyle\sum_{j \in \mathsf{Deriv}_{\mathcal{I}}(\Upsilon, J)} \prod_{v \in \mathsf{Sb}_{\mathcal{I}}(\Sigma,\Upsilon)} \{\star \mid \Psi(v)(j[v]) = !_{\mathcal{H}om}(j)^\bullet(\Phi(v))\}$$

We note that the above rely heavily on equations validated by the substitution calculus of sets (and thus of $\mathcal{M}od$), which imply $!_{\mathcal{M}od}(v' \circ v) = !_{\mathcal{M}od}(v') \circ !_{\mathcal{M}od}(v)$, $!_{\mathcal{M}od}(J[v])(X) = !_{\mathcal{M}od}(J)(!_{\mathcal{M}od}(v)(X))$, and $!_{\mathcal{M}od}(j[v])(X) = !_{\mathcal{M}od}(j)(!_{\mathcal{M}od}(v)(X))$.

Equipping these sets with the relevant operations produces an LF algebra $\mathcal{R}ec$ and homomorphisms $-^\circ : \mathcal{R}ec \to \mathcal{I}$ and $!_{\mathcal{R}ec} : \mathcal{I} \to \mathcal{R}ec$ with $-^\circ \circ !_{\mathcal{R}ec}$ the identity, from which we conclude $\mathsf{Sig}_{!_{\mathcal{R}ec}}(\Sigma)^\circ = \Sigma$. We may therefore complete step (1) by defining:

$$\mathrm{rec}_X := \mathsf{Sig}_{!_{\mathcal{R}ec}}(\Sigma)^\bullet(\mathbf{id}_\Sigma)$$

In step (2), we fix $\Sigma \in \mathsf{Sig}_{\mathcal{I}}$, $X \in \mathsf{Mod}(\Sigma)$, and $f, g \in \hom_{\mathsf{Mod}(\Sigma)}(\mathsf{Syn}(\Sigma), X)$, and generalize over $\Upsilon \in \mathsf{Sig}_{\mathcal{I}}$ and $v \in \mathsf{Sb}_{\mathcal{I}}(\Sigma, \Upsilon)$. Our final LF algebra $\mathcal{E}qn$ encodes an inductive proof that the images of f and g under every v are equal homomorphisms of Υ-models from $!_{\mathcal{S}yn}(\Upsilon)^\bullet(v)$ to $!_{\mathcal{M}od}(v)(X)$.

$$\mathsf{Sig}_{\mathcal{E}qn} := \textstyle\sum_{\Upsilon \in \mathsf{Sig}_{\mathcal{I}}} \prod_{v \in \mathsf{Sb}_{\mathcal{I}}(\Sigma,\Upsilon)} \{\star \mid !_{\mathcal{H}om}(v)^\bullet(f) = !_{\mathcal{H}om}(v)^\bullet(g)\}$$

$$\mathsf{Sb}_{\mathcal{E}qn}((\Upsilon, \Phi), (\Upsilon', \Phi')) := \textstyle\sum_{v' \in \mathsf{Sb}_{\mathcal{I}}(\Upsilon,\Upsilon')} \{\star\}$$

$$\mathsf{Jdg}_{\mathcal{E}qn}(\Upsilon, \Phi) :=$$
$$\textstyle\sum_{J \in \mathsf{Jdg}_{\mathcal{I}}(\Upsilon)} \prod_{v \in \mathsf{Sb}_{\mathcal{I}}(\Sigma,\Upsilon)} \prod_{j \in \mathsf{Deriv}_{\mathcal{I}}(\Sigma, J[v])} \{\star \mid !_{\mathcal{H}om}(j)^\bullet(f) = !_{\mathcal{H}om}(j)^\bullet(g)\}$$

$$\mathsf{Deriv}_{\mathcal{E}qn}((\Upsilon, \Phi), (J, \Psi)) := \textstyle\sum_{j \in \mathsf{Deriv}_{\mathcal{I}}(\Upsilon, J)} \{\star\}$$

After equipping $\mathcal{E}qn$ with the remaining LF algebra structure—the only interesting case being $\mathbf{Sort}_{\mathcal{E}qn}$—we obtain $\mathsf{Sig}_{!_{\mathcal{E}qn}(\Sigma)^\bullet(\mathbf{id}_\Sigma)} \in \{\star \mid f = g\}$, completing step (2).

We may thus sum up Appendix B.2 in the following theorem, whose $\Sigma = \Sigma_{\mathrm{ETT}}$ case is the initiality theorem for extensional type theory (Theorem 3.4.5).

Theorem B.2.15. *For every GAT signature* Σ*, there is a* category of models $\mathsf{Mod}(\Sigma)$ *with an initial object* $\mathsf{Syn}(\Sigma)$.

B.3 The initial LF algebra

We conclude our treatment of generalized algebraic theories by constructing the initial LF algebra $\mathcal{I}$ asserted in Theorem B.1.7, whose existence is essential to our definition of GAT signature (Definition B.1.8) and all of the model theory of Appendix B.2.

There are many ways to see that $\mathcal{I}$ exists. For example, the existence of an initial object $\mathcal{I}$ follows immediately from the fact that the category **LFAlg** is locally finitely presentable [AR94] and thus both complete and cocomplete. This in turn follows from the fact that LF algebras are the models of an essentially algebraic theory [AR94, Section 3.D], or of a finite limit sketch [Joh02, Chapter D2], or of a partial Horn theory [PV07]; or alternatively, by the argument in Section 5.2.1 of Uemura [Uem21].

We will instead outline an explicit construction of $\mathcal{I}$ as equivalence classes of syntax, closely following Castellan, Clairambault, and Dybjer [CCD17] and de Boer [dBoe20]. (There are also multiple syntactic constructions in the literature, including those of Cartmell [Car86] and Palmgren and Vickers [PV07].) Specifically, we define each carrier set of $\mathcal{I}$ as a set of well-formed, fully-annotated terms quotiented by judgmental equality. The construction is somewhat delicate, due (as usual) to the

mutual and quotiented nature of signatures, substitutions, judgments, and derivations, so we break it down into three steps:

1. First, we define the "pre-syntax" of LF signatures, substitutions, judgments, and derivations, which omits all equations and well-formedness conditions.
2. Next, we define several families of partial equivalence relations (PERs) over the pre-syntax, specifying when two pieces of pre-syntax are well-formed and equal according to the rules of LF algebras.
3. Finally, we show that the subquotients of pre-syntax induced by these PERs produce the initial LF algebra $\mathcal{I}$.

Readers may consult Castellan, Clairambault, and Dybjer [CCD17] for more details.

B.3.1 Pre-syntax

Our first step is to generate sets of pre-signatures, pre-substitutions, pre-judgments, and pre-derivations with a simple context-free grammar.

Pre-signatures	$S, T ::=$	$\mathtt{Emp} \mid \mathtt{Ext}(S, J)$
Pre-substitutions	$s, t ::=$	$\mathtt{emp}_S \mid \mathtt{ext}_{S,T,J}(s, a) \mid \mathtt{wk}_{S,J} \mid \mathtt{id}_S \mid s \circ_{S_2,S_1,S_0} t$
Pre-judgments	$J, K ::=$	$\mathtt{Sort}_S \mid \mathtt{El}_S(u) \mid \mathtt{Eq}_S(u, a, b) \mid \mathtt{Pi}_S(u, J) \mid \mathtt{Sub}_{S,T}(J, s)$
Pre-derivations	$u, a, b ::=$	$\mathtt{var}_{S,J} \mid \mathtt{app}_{S,u,J}(a, b) \mid \mathtt{sub}_{S,T,J}(a, s)$

The resulting sets PreSig, PreSb, PreJdg, and PreDeriv are the raw material of the corresponding carrier sets of $\mathcal{I}$: they contain fully-annotated versions of each LF algebra operation, but with no well-formedness conditions and no equations. To pass from these sets to $\mathcal{I}$, we must first carve out the subset of well-formed expressions—which involves (e.g.) determining the signature at which each well-formed pre-judgment lives—and then quotient them by the LF algebra equations. More precisely, because equations affect well-formedness, we must perform these two steps *simultaneously*.

Partial equivalence relations, or PERs, are a convenient technical device for simultaneously specifying a subset and an equivalence relation on that subset.

Definition B.3.1. Given a set X, a *partial equivalence relation* (PER) is a binary relation $R \subseteq X \times X$ on X satisfying the following properties:

- *Symmetry:* If $(x, y) \in R$ then $(y, x) \in R$.
- *Transitivity:* If $(x, y) \in R$ and $(y, z) \in R$ then $(x, z) \in R$.

Note that we do *not* ask for R to be reflexive; however, if we restrict attention to the subset of elements related to themselves, $\mathrm{dom}(R) := \{x \in X \mid (x, x) \in R\}$, then the restriction of R to $\mathrm{dom}(R)$, $R_{|\mathrm{dom}(R)}$, is an *equivalence relation* (reflexive, symmetric, and transitive) on $\mathrm{dom}(R)$.

Definition B.3.2. Given a set X and PER R on X, the *subquotient* of X by R, X/R, is the quotient of the set $\mathrm{dom}(R)$ by the equivalence relation $R_{|\mathrm{dom}(R)}$. We write $[-] : \mathrm{dom}(R) \to X/R$ for the canonical quotient map.

B.3.2 Well-formedness and equations

Our next step is thus to define PERs on our sets of pre-syntax which simultaneously capture the well-formedness conditions and equations of LF algebras. For example, we will define a PER $\mathcal{R}_{\mathsf{Sig}} \subseteq \mathsf{PreSig} \times \mathsf{PreSig}$ such that $\mathrm{dom}(\mathcal{R}_{\mathsf{Sig}})$ is the unquotiented set of well-formed pre-signatures, $(\mathcal{R}_{\mathsf{Sig}})_{|\mathrm{dom}(\mathcal{R}_{\mathsf{Sig}})}$ is the equivalence relation induced by the axioms of LF algebras, and $\mathsf{PreSig}/\mathcal{R}_{\mathsf{Sig}}$ is exactly the set of $\mathcal{I}$-signatures $\mathsf{Sig}_{\mathcal{I}}$.

Unfortunately, the remaining carrier sets $\mathsf{Sb}_{\mathcal{I}}(\Sigma, \Sigma')$, $\mathsf{Jdg}_{\mathcal{I}}(\Sigma)$, and $\mathsf{Deriv}_{\mathcal{I}}(\Sigma, J)$ introduce the further complication of being indexed, requiring us to consider relations of higher arity. We start by defining the following four relations by mutual induction:

$$\mathcal{R}_{\mathsf{Sig}} \subseteq \mathsf{PreSig} \times \mathsf{PreSig}$$
$$\mathcal{R}_{\mathsf{Sb}} \subseteq \mathsf{PreSig} \times \mathsf{PreSb} \times \mathsf{PreSb} \times \mathsf{PreSig}$$
$$\mathcal{R}_{\mathsf{Jdg}} \subseteq \mathsf{PreSig} \times \mathsf{PreJdg} \times \mathsf{PreJdg}$$
$$\mathcal{R}_{\mathsf{Deriv}} \subseteq \mathsf{PreSig} \times \mathsf{PreDeriv} \times \mathsf{PreDeriv} \times \mathsf{PreJdg}$$

We have already explained that $(S, T) \in \mathcal{R}_{\mathsf{Sig}}$ should hold if and only if the pre-signatures S and T are well-formed and equal according to the definition of LF algebras. As for the remaining relations, $(S, s, t, T) \in \mathcal{R}_{\mathsf{Sb}}$ holds if and only if S and T are well-formed pre-signatures, and s and t are well-formed and equal pre-substitutions from S to T; $(S, J, K) \in \mathcal{R}_{\mathsf{Jdg}}$ holds when S is a well-formed pre-signature and J and K are well-formed and equal pre-judgments in S; and $(S, a, b, J) \in \mathcal{R}_{\mathsf{Deriv}}$ holds when S is a well-formed pre-signature, J is a well-formed pre-judgment in S, and a and b are well-formed and equal pre-derivations of J in S.

Warning B.3.3. Well-formed pre-signatures are not $\mathcal{I}$-signatures; recall that we will eventually define $\mathcal{I}$-signatures as *equivalence classes* of well-formed pre-signatures.

The definitions of $\mathcal{R}_{\mathsf{Sig}}$, $\mathcal{R}_{\mathsf{Sb}}$, $\mathcal{R}_{\mathsf{Jdg}}$, and $\mathcal{R}_{\mathsf{Deriv}}$ look very similar to the rules of the substitution calculus (Section 2.3) extended by the type formers described in

Appendix B.1, with congruence and conversion principles carefully incorporated throughout. For example, the two clauses of $\mathcal{R}_{\mathsf{Sig}}$ are:

$$\frac{}{(\mathtt{Emp}, \mathtt{Emp}) \in \mathcal{R}_{\mathsf{Sig}}} \qquad \frac{(S,T) \in \mathcal{R}_{\mathsf{Sig}} \qquad (S,J,K) \in \mathcal{R}_{\mathsf{Jdg}}}{(\mathtt{Ext}(S,J), \mathtt{Ext}(T,K)) \in \mathcal{R}_{\mathsf{Sig}}}$$

Remark B.3.4. We should pause to remark on our use of inference rule notation here, given that the purpose of this entire appendix is to formally justify the inference rules used throughout the book. Unlike the latter inference rules, the rules in this section can be trivially translated into an ordinary inductive definition. That is, we are defining $\mathcal{R}_{\mathsf{Sig}}$, $\mathcal{R}_{\mathsf{Sb}}$, $\mathcal{R}_{\mathsf{Jdg}}$, and $\mathcal{R}_{\mathsf{Deriv}}$ as the smallest subsets satisfying the system of equations:

$$\begin{aligned}
\mathcal{R}_{\mathsf{Sig}} &= \{(\mathtt{Emp}, \mathtt{Emp})\} \cup \{(\mathtt{Ext}(S,J), \mathtt{Ext}(T,K)) \mid (S,T) \in \mathcal{R}_{\mathsf{Sig}} \wedge (S,J,K) \in \mathcal{R}_{\mathsf{Jdg}}\} \\
\mathcal{R}_{\mathsf{Jdg}} &= \{(S, \mathtt{Sort}_S, \mathtt{Sort}_T) \mid (S,T) \in \mathcal{R}_{\mathsf{Sig}}\} \cup \cdots \\
&\vdots
\end{aligned}$$

To see why this system of equations admits a (smallest) solution, one may appeal to classical results such as the Knaster–Tarski fixed point theorem, which also justify the existence of the sets of pre-syntax defined earlier. With this in mind, we will continue using inference rule notation for readability. ◇

The defining clauses of these relations are quite tedious, so we only include a representative sample. For example, the following rules for $\mathcal{R}_{\mathsf{Sb}}$ ensure that it respects the relation $\mathcal{R}_{\mathsf{Sig}}$ in both of its pre-signature arguments, and that $\mathcal{R}_{\mathsf{Sb}}(S,-,-,T) := \{(s,t) \in \mathsf{PreSb} \times \mathsf{PreSb} \mid (S,s,t,T) \in \mathcal{R}_{\mathsf{Sb}}\}$ is a PER for every $S, T \in \mathsf{PreSig}$.

$$\frac{(S,S') \in \mathcal{R}_{\mathsf{Sig}} \qquad (S,s,t,T) \in \mathcal{R}_{\mathsf{Sb}}}{(S',s,t,T) \in \mathcal{R}_{\mathsf{Sb}}} \qquad \frac{(T,T') \in \mathcal{R}_{\mathsf{Sig}} \qquad (S,s,t,T) \in \mathcal{R}_{\mathsf{Sb}}}{(S,s,t,T') \in \mathcal{R}_{\mathsf{Sb}}}$$

$$\frac{(S,s,t,T) \in \mathcal{R}_{\mathsf{Sb}}}{(S,t,s,T) \in \mathcal{R}_{\mathsf{Sb}}} \qquad \frac{(S,t,t',T) \in \mathcal{R}_{\mathsf{Sb}} \qquad (S,t',t'',T) \in \mathcal{R}_{\mathsf{Sb}}}{(S,t,t'',T) \in \mathcal{R}_{\mathsf{Sb}}}$$

Other rules ensure the well-formedness of LF algebra operations, and others yet correspond to LF algebra equations:

$$\frac{(S,T) \in \mathcal{R}_{\mathsf{Sig}}}{(S, \mathsf{emp}_S, \mathsf{emp}_T, \mathtt{Emp}) \in \mathcal{R}_{\mathsf{Sb}}} \qquad \frac{(S,s,t,T) \in \mathcal{R}_{\mathsf{Sb}}}{(S,s,\mathtt{id}_T \circ_{S,T,T} t, T) \in \mathcal{R}_{\mathsf{Sb}}}$$

$$\frac{(S,_,_,\mathtt{Eq}_S(u,a,b)) \in \mathcal{R}_{\mathsf{Deriv}}}{(S,a,b,\mathtt{El}_S(u)) \in \mathcal{R}_{\mathsf{Deriv}}}$$

Many of these rules explicitly "build in" the fact that equality is a congruence. The first rule above ensures not only that emp_S is a well-formed pre-substitution from S to Emp whenever S is a well-formed pre-signature, but also that emp_S and emp_T are *equal* such well-formed pre-substitutions when $(S,T) \in \mathcal{R}_{\mathsf{Sig}}$.

Once these relations are defined, we obtain the carrier sets of $\mathcal{I}$ as subquotients:

$$\begin{aligned}
&\mathsf{Sig}_{\mathcal{I}} := \mathsf{PreSig}/\mathcal{R}_{\mathsf{Sig}} \\
&\mathsf{Sb}_{\mathcal{I}}([S],[T]) := \mathsf{PreSb}/\mathcal{R}_{\mathsf{Sb}}(S,-,-,T) && (S,T \in \mathrm{dom}(\mathcal{R}_{\mathsf{Sig}})) \\
&\mathsf{Jdg}_{\mathcal{I}}([S]) := \mathsf{PreJdg}/\mathcal{R}_{\mathsf{Jdg}}(S,-,-) && (S \in \mathrm{dom}(\mathcal{R}_{\mathsf{Sig}})) \\
&\mathsf{Deriv}_{\mathcal{I}}([S],[J]) := \mathsf{PreDeriv}/\mathcal{R}_{\mathsf{Deriv}}(S,-,-,J) && ((S,J,J) \in \mathcal{R}_{\mathsf{Jdg}})
\end{aligned}$$

Note that these definitions are only well-formed if each of the above relations is a PER, and if the latter three clauses are invariant under choices of representative, e.g., $\mathcal{R}_{\mathsf{Jdg}}(S,-,-)$ and $\mathcal{R}_{\mathsf{Jdg}}(T,-,-)$ must be identical PERs for $(S,T) \in \mathcal{R}_{\mathsf{Sig}}$ (and moreover $\mathcal{R}_{\mathsf{Jdg}}(S,J,K)$ must hold only for $S \in \mathrm{dom}(\mathcal{R}_{\mathsf{Sig}})$). These properties hold more or less by construction, and can be established by rule induction.

Theorem B.3.5. *The sets* $\mathsf{Sig}_{\mathcal{I}}$, $\mathsf{Sb}_{\mathcal{I}}(\Sigma,\Sigma')$, $\mathsf{Jdg}_{\mathcal{I}}(\Sigma)$, *and* $\mathsf{Deriv}_{\mathcal{I}}(\Sigma,J)$ *support the structure of an LF algebra, which we call* $\mathcal{I}$.

Proof sketch. We must equip these sets with LF algebra operations and verify that those operations satisfy the required equations. We implement each operation by the obvious pre-operation, e.g., $!_{[S]} := [\mathsf{emp}_S]$, which is well-defined on the subquotient because we have ensured congruence principles such as $(S, \mathsf{emp}_S, \mathsf{emp}_T, \mathsf{Emp}) \in \mathcal{R}_{\mathsf{Sb}}$ when $(S,T) \in \mathcal{R}_{\mathsf{Sig}}$. The required LF algebra equations are easy to verify because they are defining clauses of $\mathcal{R}_{\mathsf{Sig}}$, $\mathcal{R}_{\mathsf{Sb}}$, $\mathcal{R}_{\mathsf{Jdg}}$, and $\mathcal{R}_{\mathsf{Deriv}}$. □

B.3.3 Initiality

Now that we have defined $\mathcal{I}$, our final step is to prove that it is initial in **LFAlg**: for every LF algebra X, there is a unique homomorphism of LF algebras $\mathrm{rec}_X : \mathcal{I} \to X$. We start by defining four *partial functions* out of pre-syntax:

$$\begin{aligned}
&\mathrm{rec}_X^{\mathsf{Sig}} : \mathsf{PreSig} \rightharpoonup \mathsf{Sig}_X \\
&\mathrm{rec}_X^{\mathsf{Sb}} : \mathsf{PreSb} \rightharpoonup \textstyle\sum_{\Sigma,\Sigma' \in \mathsf{Sig}_X} \mathsf{Sb}_X(\Sigma,\Sigma') \\
&\mathrm{rec}_X^{\mathsf{Jdg}} : \mathsf{PreJdg} \rightharpoonup \textstyle\sum_{\Sigma \in \mathsf{Sig}_X} \mathsf{Jdg}_X(\Sigma) \\
&\mathrm{rec}_X^{\mathsf{Deriv}} : \mathsf{PreDeriv} \rightharpoonup \textstyle\sum_{\Sigma \in \mathsf{Sig}_X} \sum_{J \in \mathsf{Jdg}_X(\Sigma)} \mathsf{Deriv}_X(\Sigma,J)
\end{aligned}$$

These partial functions will send well-formed pre-operations to their corresponding operations in X. For instance, $\mathrm{rec}^{\mathsf{Sb}}_X(s \circ_{S_2,S_1,S_0} t)$ is defined exactly when $\mathrm{rec}^{\mathsf{Sig}}_X$ and $\mathrm{rec}^{\mathsf{Sb}}_X$ are defined on all subterms, and the source and target signatures line up:

$$\begin{aligned}
&\mathrm{rec}^{\mathsf{Sb}}_X(s \circ_{S_2,S_1,S_0} t) = (\Sigma_2, \Sigma_0, \sigma_0 \circ_X \sigma_1) \\
&\quad \textbf{where} \\
&\quad \Sigma_2, \Sigma_1, \Sigma_0 = \mathrm{rec}^{\mathsf{Sig}}_X(S_2), \mathrm{rec}^{\mathsf{Sig}}_X(S_1), \mathrm{rec}^{\mathsf{Sig}}_X(S_0) \\
&\quad (\Sigma_1, \Sigma_0, \sigma_0) = \mathrm{rec}^{\mathsf{Sb}}_X(s) \\
&\quad (\Sigma_2, \Sigma_1, \sigma_1) = \mathrm{rec}^{\mathsf{Sb}}_X(t)
\end{aligned}$$

One can then verify that these partial functions are defined on all well-formed pre-syntax, and send equal well-formed pre-syntax to equal and type-correct results.

Lemma B.3.6.

1. *If* $(S, T) \in \mathcal{R}_{\mathsf{Sig}}$, *then* $\mathrm{rec}^{\mathsf{Sig}}_X(S)$ *and* $\mathrm{rec}^{\mathsf{Sig}}_X(T)$ *are defined and equal.*
2. *If* $(S, s, t, T) \in \mathcal{R}_{\mathsf{Sb}}$, *then* $\mathrm{rec}^{\mathsf{Sb}}_X(s)$ *and* $\mathrm{rec}^{\mathsf{Sb}}_X(t)$ *are defined and equal, and their first two projections are* $\mathrm{rec}^{\mathsf{Sig}}_X(S)$ *and* $\mathrm{rec}^{\mathsf{Sig}}_X(T)$.
3. *If* $(S, J, K) \in \mathcal{R}_{\mathsf{Jdg}}$, *then* $\mathrm{rec}^{\mathsf{Jdg}}_X(J)$ *and* $\mathrm{rec}^{\mathsf{Jdg}}_X(K)$ *are defined and equal, and their first projection is* $\mathrm{rec}^{\mathsf{Sig}}_X(S)$.
4. *If* $(S, a, b, J) \in \mathcal{R}_{\mathsf{Deriv}}$, *then* $\mathrm{rec}^{\mathsf{Deriv}}_X(a)$ *and* $\mathrm{rec}^{\mathsf{Deriv}}_X(b)$ *are defined and equal, and their first two projections are* $\mathrm{rec}^{\mathsf{Sig}}_X(S)$ *and* $\pi_2(\mathrm{rec}^{\mathsf{Jdg}}_X(J))$.

Corollary B.3.7. *There is a homomorphism of LF algebras* $\mathrm{rec}_X : \mathcal{I} \to X$ *which sends signatures* $[S] \in \mathsf{Sig}_{\mathcal{I}}$ *to* $\mathrm{rec}^{\mathsf{Sig}}_X(S)$, *substitutions* $[s]$ *to* $\pi_3(\mathrm{rec}^{\mathsf{Sb}}_X(s))$, *judgments* $[J]$ *to* $\pi_2(\mathrm{rec}^{\mathsf{Jdg}}_X(J))$, *and derivations* $[a]$ *to* $\pi_3(\mathrm{rec}^{\mathsf{Deriv}}_X(a))$.

We can show that $f = g$ for any pair of $f, g \in \hom_{\mathbf{LFAlg}}(X, Y)$ by an inductive argument similar to Lemma B.3.6. We conclude that $\mathcal{I}$ is initial in **LFAlg**, thereby completing our proof of Theorem B.1.7.

This material will be published by Cambridge University Press as *Principles of Dependent Type Theory* by Carlo Angiuli and Daniel Gratzer. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. © Carlo Angiuli and Daniel Gratzer 2024

Bibliography

[AB04] Steve Awodey and Andrej Bauer. "Propositions as [Types]". In: *Journal of Logic and Computation* 14.4 (Aug. 2004), pp. 447–471. ISSN: 0955-792X. DOI: 10.1093/logcom/14.4.447.

[Abe13] Andreas Abel. "Normalization by Evaluation: Dependent Types and Impredicativity". Habilitation thesis. Ludwig-Maximilians-Universität München, 2013. URL: http://www2.tcs.ifi.lmu.de/~abel/habil.pdf.

[ACD07] Andreas Abel, Thierry Coquand, and Peter Dybjer. "Normalization by Evaluation for Martin-Löf Type Theory with Typed Equality Judgements". In: *22nd Annual IEEE Symposium on Logic in Computer Science*. LICS 2007. July 2007, pp. 3–12. DOI: 10.1109/LICS.2007.33.

[AFH17] Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. *Computational Higher Type Theory III: Univalent Universes and Exact Equality*. Preprint. Dec. 2017. arXiv: 1712.01800 [cs.LO].

[AFH18] Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. "Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities". In: *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*. Ed. by Dan Ghica and Achim Jung. Vol. 119. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018, 6:1–6:17. ISBN: 978-3-95977-088-0. DOI: 10.4230/LIPIcs.CSL.2018.6.

[Agda] The Agda Development Team. *The Agda Programming Language*. 2020. URL: http://wiki.portal.chalmers.se/agda/pmwiki.php.

[AGV72] Michael Artin, Alexander Grothendieck, and Jean-Louis Verdier. *Théorie des topos et cohomologie étale des schémas*. Séminaire de Géométrie Algébrique du Bois-Marie 1963–1964 (SGA 4), Dirigé par M. Artin, A. Grothendieck, et J.-L. Verdier. Avec la collaboration de N. Bourbaki, P. Deligne et B. Saint-Donat, Lecture Notes in Mathematics, Vol. 269, 270, 305. Berlin: Springer-Verlag, 1972.

[Ahr+25] Benedikt Ahrens, Paige Randall North, Michael Shulman, and Dimitris Tsementzis. *The Univalence Principle*. Vol. 305. Memoirs of the American Mathematical Society 1541. American Mathematical Society, 2025. ISBN: 978-1-4704-7269-6. DOI: 10.1090/memo/1541.

[AK16] Thorsten Altenkirch and Ambrus Kaposi. "Type theory in type theory using quotient inductive types". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 2016. New York, NY, USA: ACM, 2016, pp. 18–29. ISBN: 9781450335492. DOI: 10.1145/2837614.2837638.

[AKL15] Jeremy Avigad, Krzysztof Kapulkin, and Peter LeFanu Lumsdaine. "Homotopy limits in type theory". In: *Mathematical Structures in Computer Science* 25.5 (2015). Special issue: From type theory and homotopy theory to Univalent Foundations of Mathematics, pp. 1040–1070. ISSN: 1469-8072. DOI: 10.1017/s0960129514000498.

[ALN24] Benedikt Ahrens, Peter LeFanu Lumsdaine, and Paige Randall North. "Comparing Semantic Frameworks for Dependently-Sorted Algebraic Theories". In: *Programming Languages and Systems*. Springer Nature Singapore, Oct. 2024, pp. 3–22. ISBN: 9789819789436. DOI: 10.1007/978-981-97-8943-6_1.

[Alt+01] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. "Normalization by evaluation for typed lambda calculus with coproducts". In: *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*. LICS 2001. 2001, pp. 303–310. DOI: 10.1109/LICS.2001.932506.

[Alt+19] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. "Setoid Type Theory—A Syntactic Translation". In: *Mathematics of Program Construction*. Ed. by Graham Hutton. Cham: Springer International Publishing, 2019, pp. 155–196. ISBN: 978-3-030-33636-3.

[Alt23] Thorsten Altenkirch. "Should Type Theory Replace Set Theory as the Foundation of Mathematics?" In: *Global Philosophy* 33.21 (2023). DOI: 10.1007/s10516-023-09676-0.

[Alt99] Thorsten Altenkirch. "Extensional equality in intensional type theory". In: *Proceedings. 14th Symposium on Logic in Computer Science*. 1999, pp. 412–420. DOI: 10.1109/LICS.1999.782636.

[AM06] Thorsten Altenkirch and Conor McBride. *Towards observational type theory*. Unpublished draft. 2006. URL: http://strictlypositive.org/ott.pdf.

[AMB13] Guillaume Allais, Conor McBride, and Pierre Boutillier. "New equations for neutral terms: a sound and complete decision procedure, formalized". In: *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-Typed Programming*. DTP 2013. New York, NY, USA: ACM, 2013, pp. 13–24. DOI: 10.1145/2502409.2502411.

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "Observational Equality, Now!" In: *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*. PLPV 2007. New York, NY, USA: ACM, 2007, pp. 57–68. ISBN: 978-1-59593-677-6. DOI: 10.1145/1292597.1292608.

[Ane19] Mathieu Anel. *Descent & Univalence*. Slides from HoTTEST seminar. May 2019. URL: https://www.math.uwo.ca/faculty/kapulkin/seminars/hottestfiles/Anel-2019-05-2-HoTTEST.pdf.

[Ang+21] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. "Syntax and models of Cartesian cubical type theory". In: *Mathematical Structures in Computer Science* 31.4 (2021). Special issue on Homotopy Type Theory and Univalent Foundations, pp. 424–468. DOI: 10.1017/S0960129521000347.

[Ang19] Carlo Angiuli. "Computational Semantics of Cartesian Cubical Type Theory". PhD thesis. Carnegie Mellon University, Sept. 2019. URL: http://reports-archive.adm.cs.cmu.edu/anon/2019/CMU-CS-19-127.pdf.

[AÖV17] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. "Decidability of conversion for type theory in type theory". In: *Proceedings of the ACM on Programming Languages* 2.POPL (Dec. 2017), 23:1–23:29. DOI: 10.1145/3158111.

[AR94] Jiří Adámek and Jiří Rosický. *Locally Presentable and Accessible Categories*. Vol. 189. London Mathematical Society Lecture Note Series. Cambridge University Press, 1994. ISBN: 9780521422611. DOI: 10.1017/CBO9780511600579.

[Asp95] David Aspinall. "Subtyping with singleton types". In: *Computer Science Logic (CSL 1994)*. Ed. by Leszek Pacholski and Jerzy Tiuryn. Vol. 933. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–15. ISBN: 978-3-540-60017-6. DOI: 10.1007/BFb0022243.

[Aug99] Lennart Augustsson. "Cayenne — A Language with Dependent Types". In: *Advanced Functional Programming (AFP 1998)*. Ed. by S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques. Vol. 1608. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 240–267. ISBN: 978-3-540-66241-9. DOI: 10.1007/10704973_6.

[AW09] Steve Awodey and Michael A. Warren. "Homotopy theoretic models of identity types". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 146.1 (Jan. 2009), pp. 45–55. ISSN: 0305-0041. DOI: 10.1017/S0305004108001783.

[Awo10] Steve Awodey. *Category Theory*. Second Edition. Oxford Logic Guides 52. Oxford University Press, 2010. ISBN: 9780199587360.

[Awo18] Steve Awodey. "Natural models of homotopy type theory". In: *Mathematical Structures in Computer Science* 28.2 (2018), pp. 241–286. DOI: 10.1017/S0960129516000268.

[Awo26] Steve Awodey. *Cartesian Cubical Model Categories*. Springer Nature Switzerland, 2026. ISBN: 9783032087300. DOI: 10.1007/978-3-032-08730-0.

[Bar+26] Reid Barton, Axel Ljungström, Owen Milner, and Anders Mörtberg. *A computer formalisation of the Serre finiteness theorem*. To appear in LICS'26. 2026. URL: https://github.com/CMU-HoTT/serre-finiteness.

[Bar91] Henk Barendregt. "Introduction to generalized type systems". In: *Journal of Functional Programming* 1.2 (Apr. 1991), pp. 125–154. DOI: 10.1017/S0956796800020025.

[Bau+17] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. "The HoTT Library: A Formalization of Homotopy Type Theory in Coq". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. CPP 2017. New York, NY, USA: ACM, 2017, pp. 164–172. ISBN: 978-1-4503-4705-1. DOI: 10.1145/3018610.3018615.

[Bau+21] Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Anja Petković, Matija Pretnar, and Chris Stone. *Andromeda: Your type theory à la Martin-Löf*. 2021. URL: https://www.andromeda-prover.org/.

[BCH14] Marc Bezem, Thierry Coquand, and Simon Huber. "A Model of Type Theory in Cubical Sets". In: *19th International Conference on Types for Proofs and Programs (TYPES 2013)*. Ed. by Ralph Matthes and Aleksy Schubert. Vol. 26. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik,

2014, pp. 107–128. ISBN: 978-3-939897-72-9. DOI: 10.4230/LIPIcs.TYPES.2013.107.

[Bez+21] Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Escardó. "On generalized algebraic theories and categories with families". In: *Mathematical Structures in Computer Science* 31.9 (2021). Special issue in homage to Martin Hofmann, pp. 1006–1023. DOI: 10.1017/S0960129521000268.

[Bis67] Errett Bishop. *Foundations of constructive analysis*. McGraw-Hill Book Company, 1967.

[BKS23] Rafaël Bocquet, Ambrus Kaposi, and Christian Sattler. *For the Metatheory of Type Theory, Internal Sconing Is Enough*. 2023. arXiv: 2302.05190 [cs.LO]. URL: https://arxiv.org/abs/2302.05190.

[Bor94] Francis Borceux. *Handbook of Categorical Algebra*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1994.

[Bra13] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.5 (2013), pp. 552–593. DOI: 10.1017/S095679681300018X.

[Bra17] Edwin Brady. *Type-Driven Development with Idris*. Manning Publications, 2017. ISBN: 9781617293023.

[Bru16] Guillaume Brunerie. "On the homotopy groups of spheres in homotopy type theory". PhD thesis. Université Nice Sophia Antipolis, 2016. URL: http://arxiv.org/abs/1606.05916.

[Bru18] Guillaume Brunerie. "The James Construction and $\pi_4(\mathbb{S}^3)$ in Homotopy Type Theory". In: *Journal of Automated Reasoning* 63.2 (June 2018), pp. 255–284. ISSN: 1573-0670. DOI: 10.1007/s10817-018-9468-2.

[BV73] J. M. Boardman and R. M. Vogt. *Homotopy Invariant Algebraic Structures on Topological Spaces*. Vol. 347. Lecture Notes in Mathematics. Springer Berlin Heidelberg, 1973. ISBN: 978-3-540-06479-4. DOI: 10.1007/bfb0068547.

[Car86] John Cartmell. "Generalised algebraic theories and contextual categories". In: *Annals of Pure and Applied Logic* 32 (1986), pp. 209–243. ISSN: 0168-0072. DOI: 10.1016/0168-0072(86)90053-9.

[CCD17] Simon Castellan, Pierre Clairambault, and Peter Dybjer. "Undecidability of Equality in the Free Locally Cartesian Closed Category (Extended version)". In: *Logical Methods in Computer Science* 13.4 (Nov. 2017). DOI: 10.23638/LMCS-13(4:22)2017.

[CCHM18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom". In: *21st International Conference on Types for Proofs and Programs (TYPES 2015)*. Ed. by Tarmo Uustalu. Vol. 69. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018, 5:1–5:34. ISBN: 978-3-95977-030-9. DOI: 10.4230/LIPIcs.TYPES.2015.5.

[CD13] Thierry Coquand and Nils Anders Danielsson. "Isomorphism is equality". In: *Indagationes Mathematicae* 24.4 (2013). In memory of N.G. (Dick) de Bruijn (1918–2012), pp. 1105–1120. ISSN: 0019-3577. DOI: 10.1016/j.indag.2013.09.002.

[CD14] Pierre Clairambault and Peter Dybjer. "The biequivalence of locally cartesian closed categories and Martin-Löf type theories". In: *Mathematical Structures in Computer Science* 24.6 (2014). DOI: 10.1017/S0960129513000881.

[CDP14] Jesper Cockx, Dominique Devriese, and Frank Piessens. "Pattern Matching without K". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2014. New York, NY, USA: ACM, 2014, pp. 257–268. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628139.

[CH19] Evan Cavallo and Robert Harper. "Higher Inductive Types in Cubical Computational Type Theory". In: *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019), 1:1–1:27. ISSN: 2475-1421. DOI: 10.1145/3290314.

[CH88] Thierry Coquand and Gérard Huet. "The Calculus of Constructions". In: *Information and Computation* 76.2 (1988), pp. 95–120. ISSN: 0890-5401. DOI: 10.1016/0890-5401(88)90005-3.

[Chr19] David Thrane Christiansen. *Checking Dependent Types with Normalization by Evaluation: A Tutorial*. Online. 2019. URL: https://davidchristiansen.dk/tutorials/nbe/.

[Chr23] David Thrane Christiansen. *Functional Programming in Lean*. Online, 2023. URL: https://lean-lang.org/functional_programming_in_lean/.

[CM16] Thierry Coquand and Bassel Mannaa. "The Independence of Markov's Principle in Type Theory". In: *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. Ed. by Delia Kesner and Brigitte Pientka. Vol. 52. Leibniz International Proceedings

in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, 17:1–17:18. ISBN: 978-3-95977-010-1. DOI: 10.4230/LIPIcs.FSCD.2016.17.

[Con+85] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development Environment*. Prentice-Hall, 1985. URL: http://www.nuprl.org/book/.

[Coq+09] Thierry Coquand, Yoshiki Kinoshita, Bengt Nordström, and Makoto Takeyama. "A simple type-theoretic language: Mini-TT". In: *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*. Ed. by Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin. Cambridge University Press, 2009. Chap. 6, pp. 139–164. DOI: 10.1017/CBO9780511770524.007.

[Coq+13] Thierry Coquand, Nils Anders Danielsson, Martín Escardó, Ulf Norell, and Chuangjie Xu. "Negative consistent axioms can be postulated without loss of canonicity". Unpublished note. 2013. URL: https://martinescardo.github.io/papers/negative-axioms.pdf.

[Coq13] Thierry Coquand. "Presheaf model of type theory". Unpublished note. 2013. URL: http://www.cse.chalmers.se/~coquand/presheaf.pdf.

[Coq14] Thierry Coquand. "A remark on singleton types". Unpublished note. Mar. 2014. URL: https://www.cse.chalmers.se/~coquand/singl.pdf.

[Coq18] Thierry Coquand. *Quillen model structure*. Email to Homotopy Type Theory mailing list. June 2018. URL: https://groups.google.com/d/msg/homotopytypetheory/RQkLWZ_83kQ/tAyb3zYTBQAJ.

[Coq19] Thierry Coquand. "Canonicity and normalization for dependent type theory". In: *Theoretical Computer Science* 777 (2019). In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I, pp. 184–191. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2019.01.015.

[Coq86] Thierry Coquand. "An Analysis of Girard's Paradox". In: *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science*. LICS 1986. IEEE Computer Society Press, June 1986, pp. 227–236. URL: https://inria.hal.science/inria-00076023.

[Coq91] Thierry Coquand. "An algorithm for testing conversion in type theory". In: *Logical Frameworks*. Ed. by Gérard Huet and Gordon Plotkin. Proceedings of the first international workshop on Logical Frameworks. Cambridge University Press, 1991, pp. 255–279. ISBN: 9780521413008. DOI: 10.1017/CBO9780511569807.011.

[Coq92] Thierry Coquand. "Pattern Matching with Dependent Types". In: *Proceedings of the 1992 Workshop on Types for Proofs and Programs*. Ed. by Bengt Nordström, Kent Petersson, and Gordon Plotkin. 1992, pp. 66–79. URL: https://www.lfcs.inf.ed.ac.uk/research/types-bra/proc/proc92.ps.gz.

[Coq96] Thierry Coquand. "An algorithm for type-checking dependent types". In: *Science of Computer Programming* 26.1 (1996), pp. 167–177. ISSN: 0167-6423. DOI: 10.1016/0167-6423(95)00021-6.

[CP90] Thierry Coquand and Christine Paulin. "Inductively defined types". In: *COLOG-88*. Ed. by Per Martin-Löf and Grigori Mints. Vol. 417. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 50–66. ISBN: 978-3-540-52335-2. DOI: 10.1007/3-540-52335-9_47.

[Cro94] Roy L. Crole. *Categories for Types*. Cambridge Mathematical Textbooks. Cambridge University Press, 1994. ISBN: 978-0521457019. DOI: 10.1017/CBO9781139172707.

[dBoe20] Menno de Boer. "A Proof and Formalization of the Initiality Conjecture of Dependent Type Theory". Licentiate thesis. Stockholm University, 2020. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:su:diva-181640.

[dBru72] N. G. de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: 10.1016/1385-7258(72)90034-0.

[dBru73] Nicolaas Govert de Bruijn. *Automath: a language for mathematics*. Les Presses de L'Université de Montréal, 1973. ISBN: 0840502230.

[Dij17] Gabe Dijkstra. "Quotient inductive-inductive definitions". PhD thesis. University of Nottingham, 2017. URL: https://eprints.nottingham.ac.uk/42317/.

[dMU21] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 625–635. ISBN: 978-3-030-79876-5. DOI: 10.1007/978-3-030-79876-5_37.

[Dow93] Gilles Dowek. "The undecidability of typability in the Lambda-Pi-calculus". In: *Typed Lambda Calculi and Applications (TLCA 1993)*. Ed. by Marc Bezem and Jan Friso Groote. Vol. 664. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 139–145. ISBN: 978-3-540-56517-8. DOI: 10.1007/BFb0037103.

[Dyb00] Peter Dybjer. "A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory". In: *The Journal of Symbolic Logic* 65.2 (2000), pp. 525–549. ISSN: 00224812. DOI: 10.2307/2586554.

[Dyb94] Peter Dybjer. "Inductive families". In: *Formal Aspects of Computing* 6.4 (July 1994), pp. 440–465. ISSN: 0934-5043. DOI: 10.1007/BF01211308.

[Dyb96] Peter Dybjer. "Internal type theory". In: *Types for Proofs and Programs (TYPES 1995)*. Ed. by Stefano Berardi and Mario Coppo. Vol. 1158. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 120–134. ISBN: 978-3-540-70722-6. DOI: 10.1007/3-540-61780-9_66.

[Esc+10] Martín H. Escardó and contributors. *TypeTopology*. Agda development. 2010. URL: https://github.com/martinescardo/TypeTopology.

[Esc14] Martín Hötzel Escardó. *Comment on "Generalize 7.2.2 and simplify encode-decode"*. GitHub comment. Dec. 2014. URL: https://github.com/HoTT/book/issues/718#issuecomment-65378867.

[FAM23] Kuen-Bang Hou (Favonia), Carlo Angiuli, and Reed Mullanix. "An Order-Theoretic Analysis of Universe Polymorphism". In: *Proceedings of the ACM on Programming Languages* 7.POPL (Jan. 2023). DOI: 10.1145/3571250.

[FC18] Daniel P. Friedman and David Thrane Christiansen. *The Little Typer*. The MIT Press, 2018. ISBN: 9780262536431.

[Fio02] Marcelo Fiore. "Semantic Analysis of Normalisation by Evaluation for Typed Lambda Calculus". In: *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP 2002. ACM, 2002, pp. 26–37. ISBN: 1-58113-528-9. DOI: 10.1145/571157.571161.

[Fre78] Peter Freyd. "On proving that **1** is an indecomposable projective in various free categories". Unpublished note. 1978.

[GG08] Nicola Gambino and Richard Garner. "The identity type weak factorisation system". In: *Theoretical Computer Science* 409.1 (2008), pp. 94–109. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2008.08.030.

[Gir99] Jean-Yves Girard. "On the Meaning of Logical Rules I: Syntax Versus Semantics". In: *Computational Logic*. Ed. by Ulrich Berger and Helmut Schwichtenberg. Vol. 165. NATO ASI Series F: Computer and Systems Sciences. Springer Berlin Heidelberg, 1999, pp. 215–272. ISBN: 9783642586224. DOI: 10.1007/978-3-642-58622-4_7.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989. ISBN: 0521371813. URL: https://www.paultaylor.eu/stable/Proofs+Types.

[GMW79] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Vol. 78. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 1979. ISBN: 978-3-540-09724-2. DOI: 10.1007/3-540-09724-4.

[Gra+21] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. "Multimodal Dependent Type Theory". In: *Logical Methods in Computer Science* 17.3 (July 2021). DOI: 10.46298/lmcs-17(3:11)2021.

[Gra+22] Daniel Gratzer, Evan Cavallo, G. A. Kavvos, Adrien Guatto, and Lars Birkedal. "Modalities and Parametric Adjoints". In: *ACM Transactions on Computational Logic* 23.3 (Apr. 2022). ISSN: 1529-3785. DOI: 10.1145/3514241.

[Gra22] Daniel Gratzer. "Normalization for Multimodal Type Theory". In: *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS 2022. New York, NY, USA: ACM, 2022. ISBN: 9781450393515. DOI: 10.1145/3531130.3532398.

[Gra23] Daniel Gratzer. "Syntax and semantics of modal type theory". PhD thesis. Aarhus University, 2023. URL: https://pure.au.dk/portal/en/publications/syntax-and-semantics-of-modal-type-theory(694f77d2-47d3-4986-bb82-129b8d96206e).html.

[Grä79] George Grätzer. *Universal Algebra*. Second Edition. Springer, 1979. ISBN: 978-0387903552. DOI: 10.1007/978-0-387-77487-9.

[GSS26] Daniel Gratzer, Michael Shulman, and Jonathan Sterling. "Strict universes for Grothendieck topoi". In: *Theory and Application of Categories* 45.30 (2026), pp. 1227–1279. URL: http://www.tac.mta.ca/tac/volumes/45/30/45-30abs.html.

[Har09] John Harrison. "HOL Light: An Overview". In: *Theorem Proving in Higher Order Logics (TPHOLs 2009)*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 60–66. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_4.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Second Edition. Cambridge University Press, 2016. ISBN: 9781107150300. DOI: 10.1017/CBO9781316576892.

[Has21] Philipp Haselwarter. "Effective Metatheory for Type Theory". PhD thesis. University of Ljubljana, 2021. URL: https://repozitorij.uni-lj.si/IzpisGradiva.php?id=134439.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. "A Framework for Defining Logics". In: *Journal of the ACM* 40.1 (Jan. 1993), pp. 143–184. ISSN: 0004-5411. DOI: 10.1145/138027.138060.

[HM95] Robert Harper and Greg Morrisett. "Compiling Polymorphism Using Intensional Type Analysis". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 1995. New York, NY, USA: ACM, 1995, pp. 130–141. ISBN: 0897916921. DOI: 10.1145/199448.199475.

[Hof95a] Martin Hofmann. "Extensional concepts in intensional type theory". PhD thesis. University of Edinburgh, July 1995. URL: http://www.lfcs.inf.ed.ac.uk/reports/95/ECS-LFCS-95-327/.

[Hof95b] Martin Hofmann. "On the interpretation of type theory in locally cartesian closed categories". In: *8th Workshop, Computer Science Logic (CSL 1994)*. Ed. by Leszek Pacholski and Jerzy Tiuryn. Vol. 933. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 427–441. ISBN: 978-3-540-49404-1. DOI: 10.1007/BFb0022273.

[Hof97] Martin Hofmann. "Syntax and Semantics of Dependent Types". In: *Semantics and Logics of Computation*. Ed. by Andrew M. Pitts and P. Dybjer. Publications of the Newton Institute. Cambridge University Press, 1997, pp. 79–130. DOI: 10.1017/CBO9780511526619.004.

[Hof99] Martin Hofmann. "Semantical analysis of higher-order abstract syntax". In: *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*. LICS 1999. Washington, DC, USA: IEEE Computer Society, 1999. ISBN: 0-7695-0158-3. DOI: 10.1109/LICS.1999.782616.

[How80] William A. Howard. "The formulae-as-types notion of construction". In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Ed. by Jonathan P. Seldin and J. Roger Hindley. Academic Press, 1980, pp. 479–490. ISBN: 978-0-12-349050-6.

[HS97] Martin Hofmann and Thomas Streicher. "Lifting Grothendieck Universes". Unpublished note. 1997. URL: https://www2.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf.

[HS98] Martin Hofmann and Thomas Streicher. "The groupoid interpretation of type theory". In: *Twenty-Five Years of Constructive Type Theory*. Ed. by Giovanni Sambin and Jan Smith. Vol. 36. Oxford Logic Guides. Oxford University Press, 1998, pp. 83–111.

[Hub18] Simon Huber. "Canonicity for Cubical Type Theory". In: *Journal of Automated Reasoning* (June 2018). ISSN: 1573-0670. DOI: 10.1007/s10817-018-9469-1.

[Hur95] Antonius J. C. Hurkens. "A simplification of Girard's paradox". In: *Typed Lambda Calculi and Applications*. Ed. by Mariangiola Dezani-Ciancaglini and Gordon Plotkin. TLCA 1995. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 266–278. ISBN: 978-3-540-49178-1. DOI: 10.1007/BFb0014058.

[Hyl82] J. M. E. Hyland. "The Effective Topos". In: *The L. E. J. Brouwer Centenary Symposium*. Ed. by A.S. Troelstra and D. van Dalen. Vol. 110. Studies in Logic and the Foundations of Mathematics. Elsevier, 1982, pp. 165–216. DOI: 10.1016/s0049-237x(09)70129-6.

[Jac99] Bart Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics 141. North Holland, 1999. ISBN: 9780444539427.

[Joh02] Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium, Volume 2*. Vol. 44. Oxford Logic Guides. Clarendon Press, 2002. ISBN: 9780198515982.

[KHS19] Ambrus Kaposi, Simon Huber, and Christian Sattler. "Gluing for Type Theory". In: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. by Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Ger-

many: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019, 25:1–25:19. ISBN: 978-3-95977-107-8. DOI: 10.4230/LIPIcs.FSCD.2019.25.

[KKA19] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. "Constructing quotient inductive-inductive types". In: *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019), 2:1–2:24. DOI: 10.1145/3290315.

[KL21] Krzysztof Kapulkin and Peter LeFanu Lumsdaine. "The simplicial model of Univalent Foundations (after Voevodsky)". In: *Journal of the European Mathematical Society* 23.6 (2021), pp. 2071–2126. DOI: 10.4171/JEMS/1050.

[KL25] Krzysztof Kapulkin and Yufeng Li. "Extensional concepts in intensional type theory, revisited". In: *Theoretical Computer Science* 1029 (Mar. 2025). ISSN: 0304-3975. DOI: 10.1016/j.tcs.2024.115051.

[Kle50] S. C. Kleene. "A symmetric form of Gödel's theorem". In: *Koninklijke Nederlandse Akademie van Wetenschappen, Proceedings of the section of sciences* 53 (1950), pp. 800–802. URL: https://dwc.knaw.nl/DL/publications/PU00014670.pdf.

[Kov16] András Kovács. `elaboration-zoo`. 2016. URL: https://github.com/AndrasKovacs/elaboration-zoo.

[Kov22] András Kovács. "Type-Theoretic Signatures for Algebraic Theories and Inductive Types". PhD thesis. Eötvös Loránd University, 2022. DOI: 10.15476/ELTE.2022.070.

[KS15] Nicolai Kraus and Christian Sattler. "Higher Homotopies in a Hierarchy of Univalent Universes". In: *ACM Transactions on Computational Logic* 16.2 (Apr. 2015), 18:1–18:12. ISSN: 1529-3785. DOI: 10.1145/2729979.

[LH12] Daniel R. Licata and Robert Harper. "Canonicity for 2-Dimensional Type Theory". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 2012. New York, NY, USA: ACM, 2012, pp. 337–348. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103697.

[Lic+18] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. "Internal Universes in Models of Homotopy Type Theory". In: *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*. Ed. by Hélène Kirchner. Vol. 108. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018, 22:1–22:17. ISBN: 978-3-95977-077-4. DOI: 10.4230/LIPIcs.FSCD.2018.22.

[Lic16] Dan Licata. *Weak univalence with "beta" implies full univalence*. Email to Homotopy Type Theory mailing list. Sept. 2016. URL: https://groups.google.com/d/msg/homotopytypetheory/j2KBIvDw53s/YTDK4D0NFQAJ.

[LMS10] Andres Löh, Conor McBride, and Wouter Swierstra. "A Tutorial Implementation of a Dependently Typed Lambda Calculus". In: *Fundamenta Informaticae* 102.2 (2010). Special issue: Dependently Typed Programming, pp. 177–207. DOI: 10.3233/FI-2010-304.

[LS13] Daniel R. Licata and Michael Shulman. "Calculating the Fundamental Group of the Circle in Homotopy Type Theory". In: *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS 2013. Washington, DC, USA: IEEE Computer Society, 2013, pp. 223–232. ISBN: 978-0-7695-5020-6. DOI: 10.1109/LICS.2013.28.

[LS19] Peter LeFanu Lumsdaine and Michael Shulman. "Semantics of higher inductive types". In: *Mathematical Proceedings of the Cambridge Philosophical Society* (2019). DOI: 10.1017/S030500411900015X.

[LS88] Joachim Lambek and Philip J. Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge Studies in Advanced Mathematics 7. Cambridge University Press, 1988. ISBN: 9780521356534.

[Lum11] Peter LeFanu Lumsdaine. *Strong functional extensionality from weak*. Blog post. Dec. 2011. URL: https://homotopytypetheory.org/2011/12/19/strong-funext-from-weak/.

[Lum17] Peter LeFanu Lumsdaine. *Answer to "Can we always make a strictly functorial choice of pullbacks/re-indexing?"* MathOverflow answer. 2017. URL: https://mathoverflow.net/q/279985.

[LW15] Peter LeFanu Lumsdaine and Michael A. Warren. "The Local Universes Model: An Overlooked Coherence Construction for Dependent Type Theories". In: *ACM Transactions on Computational Logic* 16.3 (2015). DOI: 10.1145/2754931.

[Mar71] Per Martin-Löf. "An intuitionistic theory of types". Unpublished preprint. 1971.

[Mar75] Per Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Logic Colloquium '73*. Ed. by H. E. Rose and J. C. Shepherdson. Vol. 80. Studies in Logic and the Foundations of Mathematics. North-Holland, 1975, pp. 73–118. DOI: 10.1016/S0049-237X(08)71945-1.

[Mar82] Per Martin-Löf. "Constructive mathematics and computer programming". In: *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science, Hannover 1979*. Ed. by L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski. Vol. 104. Studies in Logic and the Foundations of Mathematics. North-Holland, 1982, pp. 153–175. DOI: 10.1016/S0049-237X(09)70189-2.

[Mar84a] Per Martin-Löf. "Constructive mathematics and computer programming". In: *Philosophical Transactions of the Royal Society of London A* 312.1522 (Oct. 1984), pp. 501–518. ISSN: 0080-4614. DOI: 10.1098/rsta.1984.0073.

[Mar84b] Per Martin-Löf. *Intuitionistic type theory. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980*. Vol. 1. Studies in Proof Theory. Bibliopolis, 1984. ISBN: 88-7088-105-9.

[Mar87] Per Martin-Löf. "Truth of a Proposition, Evidence of a Judgement, Validity of a Proof". In: *Synthese* 73.3 (1987), pp. 407–420. DOI: 10.1007/bf00484985.

[Mar92] Per Martin-Löf. *Substitution calculus*. Notes from a lecture given in Göteborg, Sweden. Sept. 1992. URL: https://raw.githubusercontent.com/michaelt/martin-lof/master/pdfs/Substitution-calculus-1992.pdf.

[Mar96] Per Martin-Löf. "On the meanings of the logical constants and the justifications of the logical laws". In: *Nordic Journal of Philosophical Logic* 1.1 (May 1996), pp. 11–60. URL: https://www.hf.uio.no/ifikk/english/research/publications/journals/njpl/files/vol1no1/meaning.pdf.

[McB02] Conor McBride. "Elimination with a Motive". In: *Types for Proofs and Programs (TYPES 2000)*. Ed. by Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack. Vol. 2277. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 197–216. DOI: 10.1007/3-540-45842-5_13.

[McB18] Conor McBride. *Basics of bidirectionalism*. Blog post. Aug. 2018. URL: https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionalism/.

[McB19] Conor McBride. "The types who say 'ni'". Draft paper. Feb. 2019. URL: https://github.com/pigworker/TypesWhoSayNi/blob/master/tex/TypesWhoSayNi.pdf.

[McB99] Conor McBride. "Dependently Typed Functional Programs and their Proofs". PhD thesis. University of Edinburgh, 1999. URL: https://era.ed.ac.uk/bitstream/id/600/ECS-LFCS-00-419.pdf.

[Mim20] Samuel Mimram. *PROGRAM = PROOF*. Independently published, 2020. ISBN: 979-8615591839. URL: http://www.lix.polytechnique.fr/Labo/Samuel.Mimram/teaching/INF551/course.pdf.

[MS93] John C. Mitchell and Andre Scedrov. "Notes on sconing and relators". In: *Computer Science Logic (CSL 1992)*. Ed. by E. Börger, G. Jäger, H. Kleine Büning, S. Martini, and M. M. Richter. Vol. 702. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 1993, pp. 352–378. DOI: 10.1007/3-540-56992-8_21.

[NPS90] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990. URL: http://www.cse.chalmers.se/research/group/logic/book/.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. ISBN: 978-3-540-43376-7. DOI: 10.1007/3-540-45949-9.

[NS12] Fredrik Nordvall Forsberg and Anton Setzer. "A Finite Axiomatisation of Inductive-Inductive Definitions". In: *Logic, Construction, Computation*. Ed. by Ulrich Berger, Hannes Diener, Peter Schuster, and Monika Seisenberger. Vol. 3. Ontos Mathematical Logic. Ontos Verlag, 2012, pp. 259–288. ISBN: 9783110324532. DOI: 10.1515/9783110324921.259.

[OP16] Ian Orton and Andrew M. Pitts. "Axioms for Modelling Cubical Type Theory in a Topos". In: *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*. Ed. by Jean-Marc Talbot and Laurent Regnier. Vol. 62. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, 24:1–24:19. ISBN: 978-3-95977-022-4. DOI: 10.4230/LIPIcs.CSL.2016.24.

[Pal22] Erik Palmgren. "From type theory to setoids and back". In: *Mathematical Structures in Computer Science* 32.10 (Nov. 2022), pp. 1283–1312. ISSN: 1469-8072. DOI: 10.1017/s0960129521000189.

[Pau93] Christine Paulin-Mohring. "Inductive definitions in the system Coq: rules and properties". In: *Typed Lambda Calculi and Applications (TLCA 1993)*. Ed. by Marc Bezem and Jan Friso Groote. Vol. 664. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 328–345. ISBN: 978-3-540-56517-8. DOI: 10.1007/bfb0037116.

[PD01] Frank Pfenning and Rowan Davies. "A judgmental reconstruction of modal logic". In: *Mathematical Structures in Computer Science* 11.4 (2001), pp. 511–540. DOI: 10.1017/S0960129501003322.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN: 0-262-16209-1.

[PLT25] Loïc Pujet, Yann Leray, and Nicolas Tabareau. "Observational Equality Meets CIC". In: *ACM Trans. Program. Lang. Syst.* 47.2 (Apr. 2025). DOI: 10.1145/3719342.

[Pol02] Robert Pollack. "Dependently Typed Records in Type Theory". In: *Formal Aspects of Computing* 13.3–5 (July 2002), pp. 386–402. ISSN: 0934-5043. DOI: 10.1007/s001650200018.

[Por21] Timothy Porter. "Spaces as Infinity-Groupoids". In: *New Spaces in Mathematics: Formal and Conceptual Reflections*. Ed. by Mathieu Anel and Gabriel Catren. Vol. 1. Cambridge University Press, 2021. Chap. 5, pp. 258–321. DOI: 10.1017/9781108854429.008.

[PP90] Frank Pfenning and Christine Paulin-Mohring. "Inductively defined types in the Calculus of Constructions". In: *Mathematical Foundations of Programming Semantics (MFPS 1989)*. Ed. by M. Main, A. Melton, M. Mislove, and D. Schmidt. Vol. 442. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 1990, pp. 209–228. ISBN: 978-0-387-97375-3. DOI: 10.1007/bfb0040259.

[PT00] Benjamin C. Pierce and David N. Turner. "Local type inference". In: *ACM Transactions on Programming Languages and Systems* 22.1 (2000), pp. 1–44. DOI: 10.1145/345099.345100.

[PT22] Loïc Pujet and Nicolas Tabareau. "Observational Equality: Now for Good". In: *Proceedings of the ACM on Programming Languages* 6.POPL (Jan. 2022). DOI: 10.1145/3498693.

[PT23] Loïc Pujet and Nicolas Tabareau. "Impredicative Observational Equality". In: *Proceedings of the ACM on Programming Languages* 7.POPL (Jan. 2023). DOI: 10.1145/3571739.

[PV07] E. Palmgren and S. J. Vickers. "Partial Horn logic and cartesian categories". In: *Annals of Pure and Applied Logic* 145.3 (2007), pp. 314–353. ISSN: 0168-0072. DOI: 10.1016/j.apal.2006.10.001.

[Reu99] Bernhard Reus. "Realizability Models for Type Theories". In: *Electronic Notes in Theoretical Computer Science* 23.1 (1999). Tutorial Workshop on Realizability Semantics and Applications (associated to FLoC'99, the 1999 Federated Logic Conference), pp. 128–158. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)00108-2.

[Rie16] Emily Riehl. *Category Theory in Context*. Aurora: Dover Modern Math Originals. Dover Publications, 2016. ISBN: 978-0486809038. URL: https://emilyriehl.github.io/files/context.pdf.

[Rie24] Emily Riehl. "On the ∞-topos semantics of homotopy type theory". In: *Bulletin of the London Mathematical Society* 56.2 (Feb. 2024), pp. 461–517. ISSN: 1469-2120. DOI: 10.1112/blms.12997.

[Rij+21] Egbert Rijke, Elisabeth Stenholm, Jonathan Prieto-Cubides, Fredrik Bakke, et al. *The* agda-unimath *library*. 2021. URL: https://github.com/UniMath/agda-unimath/.

[Rij25] Egbert Rijke. *Introduction to Homotopy Type Theory*. Vol. 219. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2025. ISBN: 9781108844161. DOI: 10.1017/9781108933568.

[Rocq] The Rocq Team. *The Rocq Prover*. Formerly known as the Coq proof assistant. 2025. URL: https://rocq-prover.org/.

[Ros36] Barkley Rosser. "Extensions of Some Theorems of Gödel and Church". In: *The Journal of Symbolic Logic* 1.3 (Sept. 1936), pp. 87–91. DOI: 10.2307/2269028.

[RSS20] Egbert Rijke, Michael Shulman, and Bas Spitters. "Modalities in homotopy type theory". In: *Logical Methods in Computer Science* Volume 16, Issue 1, 2 (Jan. 2020). ISSN: 1860-5974. DOI: 10.23638/LMCS-16(1:2)2020.

[Rus03] Bertrand Russell. *The Principles of Mathematics*. 1903. URL: https://people.umass.edu/klement/pom/.

[SA21] Jonathan Sterling and Carlo Angiuli. "Normalization for Cubical Type Theory". In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS 2021. 2021, pp. 1–15. DOI: 10.1109/LICS52264.2021.9470719.

[SAG22] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. "A Cubical Language for Bishop Sets". In: *Logical Methods in Computer Science* 18.1 (Mar. 2022). DOI: 10.46298/lmcs-18(1:43)2022.

[Sal84] Anne B. Salvesen. "Typeteori - en studie". Master's thesis. University of Oslo, 1984.

[Sco18] Dana S. Scott. *Looking backward; looking forward.* Invited Talk at the Workshop in honour of Dana Scott's 85th birthday and 50 years of domain theory. July 2018. URL: https://www.youtube.com/watch?v=uS9InrmPIoc.

[See84] R. A. G. Seely. "Locally cartesian closed categories and type theory". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 95.1 (1984), pp. 33–48. DOI: 10.1017/S0305004100061284.

[Ser53] Jean-Pierre Serre. "Cohomologie modulo 2 des complexes d'Eilenberg-MacLane". In: *Commentarii Mathematici Helvetici* 27.1 (Dec. 1953), pp. 198–232. ISSN: 1420-8946. DOI: 10.1007/bf02564562.

[SH06] Christopher A. Stone and Robert Harper. "Extensional equivalence and singleton types". In: *Transactions on Computational Logic* 7.4 (2006), pp. 676–722. DOI: 10.1145/1183278.1183281.

[SH21] Jonathan Sterling and Robert Harper. "Logical Relations as Types: Proof-Relevant Parametricity for Program Modules". In: *Journal of the ACM* 68.6 (Oct. 2021). ISSN: 0004-5411. DOI: 10.1145/3474834.

[Shu08] Michael A. Shulman. *Set theory for category theory.* Preprint. Oct. 2008. arXiv: 0810.1279 [math.CT].

[Shu15] Michael Shulman. "Univalence for inverse diagrams and homotopy canonicity". In: *Mathematical Structures in Computer Science* 25.5 (2015), pp. 1203–1277. DOI: 10.1017/S0960129514000565.

[Shu19] Michael Shulman. *All $(\infty, 1)$-toposes have strict univalent universes.* Preprint. Apr. 2019. arXiv: 1904.07004 [math.AT].

[Shu21] Michael Shulman. "Homotopy Type Theory: The Logic of Space". In: *New Spaces in Mathematics: Formal and Conceptual Reflections.* Ed. by Mathieu Anel and Gabriel Catren. Vol. 1. Cambridge University Press, 2021. Chap. 6, pp. 322–404. DOI: 10.1017/9781108854429.009.

[Shu22] Michael Shulman. *Towards third generation HOTT.* Joint work with Thorsten Altenkirch and Ambrus Kaposi. Slides from CMU HoTT seminar. Apr. 2022. URL: https://home.sandiego.edu/~shulman/papers/hott-cmu-day1.pdf.

[Smi88] Jan M. Smith. "The Independence of Peano's Fourth Axiom from Martin-Löf's Type Theory Without Universes". In: *The Journal of Symbolic Logic* 53.3 (1988), pp. 840–845. ISSN: 00224812. DOI: 10.2307/2274575.

[Smi89] Jan M. Smith. "Propositional functions and families of types". In: *Notre Dame Journal of Formal Logic* 30.3 (1989), pp. 442–458. ISSN: 0029-4527. DOI: 10.1305/ndjfl/1093635159.

[SP94] Paula Severi and Erik Poll. "Pure Type Systems with Definitions". In: *Logical Foundations of Computer Science (LFCS 1994)*. Ed. by Anil Nerode and Yu. V. Matiyasevich. Vol. 813. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 1994, pp. 316–328. ISBN: 3540581405. DOI: 10.1007/3-540-58140-5_30.

[Ste21] Jonathan Sterling. "First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory". PhD thesis. Carnegie Mellon University, 2021. DOI: 10.5281/zenodo.5709838.

[Str05] Thomas Streicher. "Universes in Toposes". In: *From Sets and Types to Topology and Analysis: Towards practicable foundations for constructive mathematics*. Ed. by Laura Crosilla and Peter Schuster. Oxford University Press, Oct. 2005, pp. 78–90. ISBN: 9780198566519. DOI: 10.1093/acprof:oso/9780198566519.003.0005.

[Str93] Thomas Streicher. "Investigations Into Intensional Type Theory". Habilitation thesis. Ludwig-Maximilians-Universität München, 1993. URL: https://www2.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf.

[Stu16] Aaron Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, 2016. ISBN: 9781970001273. DOI: 10.1145/2841316.

[Swa25] Andrew W. Swan. "Irregular models of type theory". Abstract at *31st International Conference on Types for Proofs and Programs (TYPES 2025)*. 2025. URL: https://msp.cis.strath.ac.uk/types2025/abstracts/TYPES2025_paper40.pdf.

[Tas93] Álvaro Tasistro. "Formulation of Martin-Löf's theory of types with explicit substitutions". Licentiate thesis. Chalmers University of Technology and University of Göteborg, 1993.

[Tra53] B. A. Trakhtenbrot. "On Recursive Separability". In: *Doklady Akademii Nauk SSSR* 88.6 (1953), pp. 953–956.

[Tse17] Dimitris Tsementzis. "Univalent foundations as structuralist foundations". In: *Synthese* 194.9 (2017), pp. 3583–3617. DOI: 10.1007/s11229-016-1109-x.

[Tur85] D. A. Turner. "Miranda: A non-strict functional language with polymorphic types". In: *Functional Programming Languages and Computer Architecture (FPCA 1985)*. Ed. by Jean-Pierre Jouannaud. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 1–16. ISBN: 978-3-540-39677-2.

[Tur89] David Turner. "A new formulation of constructive type theory". In: *Proceedings of the Workshop on Programming Logic*. 1989.

[Uem19] Taichi Uemura. "Cubical Assemblies, a Univalent and Impredicative Universe and a Failure of Propositional Resizing". In: *24th International Conference on Types for Proofs and Programs (TYPES 2018)*. Ed. by Peter Dybjer, José Espírito Santo, and Luís Pinto. Vol. 130. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019, 7:1–7:20. ISBN: 978-3-95977-106-1. DOI: 10.4230/LIPIcs.TYPES.2018.7.

[Uem21] Taichi Uemura. "Abstract and Concrete Type Theories". PhD thesis. Institute for Logic, Language and Computation, University of Amsterdam, 2021. URL: https://hdl.handle.net/11245.1/41ff0b60-64d4-4003-8182-c244a9afab3b.

[UF13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Self-published, 2013. URL: https://homotopytypetheory.org/book/.

[VAG+20] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. *UniMath — a computer-checked library of univalent mathematics*. 2020. URL: https://github.com/UniMath/UniMath.

[Vaz+14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. "Refinement Types for Haskell". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2014. New York, NY, USA: ACM, 2014, pp. 269–282. ISBN: 9781450328739. DOI: 10.1145/2628136.2628161.

[vGle14] Tamara von Glehn. "Polynomials and Models of Type Theory". PhD thesis. University of Cambridge, 2014. DOI: 10.17863/CAM.16245.

[Voe10] Vladimir Voevodsky. *Univalent Foundations Project*. A modified version of an NSF grant application. Oct. 2010. URL: https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/univalent_foundations_project.pdf.

[Voe14] Vladimir Voevodsky. "Univalent Foundations – new type-theoretic foundations of mathematics". Talk at IHP, Paris. Apr. 2014. URL: http://www.math.ias.edu/vladimir/Lectures.

[Voe17] Vladimir Voevodsky. "The C-systems defined by universe categories: presheaves". In: *Theory and Applications of Categories* 32 (Jan. 2017), pp. 53–112. URL: http://www.tac.mta.ca/tac/volumes/32/3/32-03abs.html.

[vOos08] Jaap van Oosten. *Realizability: An Introduction to its Categorical Side*. Vol. 152. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 2008. ISBN: 9780444515841.

[War08] Michael A. Warren. "Homotopy theoretic aspects of constructive type theory". PhD thesis. Carnegie Mellon University, Aug. 2008. URL: http://mawarren.net/papers/phd.pdf.

[Wer97] Benjamin Werner. "Sets in types, types in sets". In: *Theoretical Aspects of Computer Software (TACS 1997)*. Ed. by Martín Abadi and Takayasu Ito. Vol. 1281. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 1997, pp. 530–546. ISBN: 978-3-540-63388-4. DOI: 10.1007/bfb0014566.

[Win20] Théo Winterhalter. "Formalisation and Meta-Theory of Type Theory". PhD thesis. Université de Nantes, 2020. URL: https://theowinterhalter.github.io/#phd.

[WKS22] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. Online, Aug. 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.

[Xi07] Hongwei Xi. "Dependent ML: An approach to practical programming with dependent types". In: *Journal of Functional Programming* 17.2 (2007), pp. 215–286. DOI: 10.1017/S0956796806006216.