Lecture Notes 6
Sum types

Carlo Angiuli

B522: PL Foundations
February 23, 2026

In this lecture, we further extend the STLC with an *empty type* and *sum types* (or *coproduct types*), which are in a precise sense the opposites of unit types and product types respectively. After defining empty types and sum types, we prove type safety for the resulting system, consider generalizations of product and sum types, and discuss what it means for two types to be the same.

These lecture notes correspond to Chapter 11 of Harper [Har16], although the section on type isomorphisms is not discussed in Harper [Har16].

1 Syntax

As remarked in the previous lecture, we can generally add or remove type formers from the STLC in a modular fashion. So although we will consider empty and sum types as an extension to our previous simply-typed language (with unit, function, and product types), we will not revisit unit, function, or product types until the end of this lecture when we consider applications of empty and sum types.

Types	$\tau ::=$	$\vdots$	$\vdots$	$\vdots$
		$\mathtt{void}$	$\mathtt{void}$	empty type
		$\mathtt{sum}(\tau_1, \tau_2)$	$\tau_1 + \tau_2$	sum type
Terms	$e ::=$	$\vdots$	$\vdots$	$\vdots$
		$\mathtt{abort}(\tau, e)$	$\mathtt{abort}_\tau(e)$	nullary case
		$\mathtt{inl}(\tau_1, \tau_2, e)$	$\mathtt{inl}_{\tau_1+\tau_2}(e)$	left injection
		$\mathtt{inr}(\tau_1, \tau_2, e)$	$\mathtt{inr}_{\tau_1+\tau_2}(e)$	right injection
		$\mathtt{case}(e, x_1.e_1, x_2.e_2)$	$\mathtt{case}\ e\ [\mathtt{inl}(x_1) \to e_1]\ [\mathtt{inr}(x_2) \to e_2]$	$\mathtt{case}$ analysis

Note that $\mathtt{case}$ analyses are an additional source of binding in this language.

2 Type system

We extend the STLC's type system with the following rules.

Definition 6.1 (Type system)**.** For x_1 tm, …, x_n tm $\vdash e$ tm and τ ty, we define the judgment $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$ ("e has type τ in context $x_1 : \tau_1, \ldots, x_n : \tau_n$") by the following inference rules:

$$\ldots \qquad \frac{\Gamma \vdash e : \texttt{void}}{\Gamma \vdash \texttt{abort}_\tau(e) : \tau}\ \texttt{void-ELIM}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{inl}_{\tau_1+\tau_2}(e) : \tau_1 + \tau_2}\ +\text{-INTRO}_1 \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{inr}_{\tau_1+\tau_2}(e) : \tau_1 + \tau_2}\ +\text{-INTRO}_2$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}\ e\ [\texttt{inl}(x_1) \to e_1]\ [\texttt{inr}(x_2) \to e_2] : \tau}\ +\text{-ELIM}$$

The idea behind the type $\tau_1 + \tau_2$ is that its terms are either terms of τ_1 ("tagged" with `inl`), or terms of τ_2 ("tagged" with `inr`). Given a term of type $\tau_1 + \tau_2$, we can `case` (or "`match`") on whether it evaluates to an `inl` (resp., `inr`), and then use the contained term of type τ_1 (resp., τ_2) in a further computation. Because the type system does not know whether $\Gamma \vdash e : \tau_1 + \tau_2$ evaluates to `inl` or `inr`, the two branches of the `case` must have the same type τ in order for the entire `case` expression to have a determinate type τ.

Remark 6.2. The type $\tau_1 + \tau_2$ is also often called a *tagged union* or *disjoint union* of τ_1 and τ_2. In Haskell, this type is written `Either` τ_1 τ_2. As we will discuss in Section 5, many languages allow users to choose the *names* of the tags in a tagged union, but for now we are considering "generic" tags `inl` and `inr` in the same way that we previously considered tuples with "generic" projections `fst` and `snd`.

At the start of the lecture, we said that sum types are the "opposite" of product types. Comparing the rules for sum and product types—

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}\ \times\text{-INTRO}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}(e) : \tau_1}\ \times\text{-ELIM}_1 \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}(e) : \tau_2}\ \times\text{-ELIM}_2$$

—we see that $\tau_1 \times \tau_2$ has one introduction form which takes two terms of type τ_1 and τ_2 respectively, and two elimination forms producing terms of type τ_1 and τ_2

respectively. In contrast, $\tau_1 + \tau_2$ has two introduction forms taking terms of type τ_1 and τ_2 respectively, and one elimination form with two additional premises which are respectively given variables of type τ_1 or τ_2.

The `void` type is more confusing at first glance. Just as `unit` is a nullary product type (a pair of zero things), `void` is a nullary sum type (a tagged union of zero types). There are zero introduction rules for `void`, just as there are two introduction rules for the binary sum $\tau_1 + \tau_2$. There is one elimination rule called `abort`, which can be thought of as a zero-way `case` analysis on a tagged union of zero types, just as `case` analysis on $\tau_1 + \tau_2$ is a two-way `case` analysis. All of the zero branches of `abort` have the type τ, as does the `abort` itself.

Comparing to the `unit` type, `unit` has one introduction form with no premises and zero elimination forms; `void` has zero introduction forms and one elimination form with no additional premises.

$$\frac{}{\Gamma \vdash () : \mathtt{unit}}\ \textsc{unit-intro}$$

Exercise 6.3. Although `void` has no introduction rules, there are terms e satisfying $\Gamma \vdash e : \mathtt{void}$. List several different such e.

Our type system still assigns a unique type to every term. (We still require that all variables in Γ are distinct.)

Lemma 6.4 (Uniqueness of types)**.** *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e : \tau'$ *then* $\tau = \tau'$.

Exercise 6.5. The subscripts on $\mathtt{inl}_{\tau_1+\tau_2}(e)$, $\mathtt{inr}_{\tau_1+\tau_2}(e)$, and $\mathtt{abort}_{\tau}(e)$ are needed for uniqueness of types to hold. Why?

The type system still satisfies the structural properties of hypothetical judgments: reflexivity, exchange, substitution (to be discussed), and weakening.

Lemma 6.6 (Weakening)**.** *If* $\Gamma \vdash e : \tau$ *and* τ' ty *then* $\Gamma, x : \tau' \vdash e : \tau$.

3 Operational semantics

We extend the STLC's operational semantics with the following rules.

Definition 6.7 (Values)**.** For $\cdot \vdash e$ tm, we define the judgment e val ("e is a value") by the following inference rules.

$$\ldots \qquad \frac{v\ \mathsf{val}}{\mathtt{inl}_{\tau_1+\tau_2}(v)\ \mathsf{val}} \qquad \frac{v\ \mathsf{val}}{\mathtt{inr}_{\tau_1+\tau_2}(v)\ \mathsf{val}}$$

There are two new value forms because there are two new introduction forms.

Definition 6.8 (Small-step operational semantics)**.** For $\cdot \vdash e\ \mathsf{tm}$, we define the judgment $e \longmapsto e'$ ("e steps to e'") by the following inference rules.

$$\frac{e \longmapsto e'}{\mathtt{abort}_\tau(e) \longmapsto \mathtt{abort}_\tau(e')} \qquad \frac{e \longmapsto e'}{\mathtt{inl}_{\tau_1+\tau_2}(e) \longmapsto \mathtt{inl}_{\tau_1+\tau_2}(e')}$$

$$\frac{e \longmapsto e'}{\mathtt{inr}_{\tau_1+\tau_2}(e) \longmapsto \mathtt{inr}_{\tau_1+\tau_2}(e')} \qquad \frac{e \longmapsto e'}{\begin{array}{l}\mathtt{case}\ e\ [\mathtt{inl}(x_1) \to e_1]\ [\mathtt{inr}(x_2) \to e_2] \longmapsto \\ \quad \mathtt{case}\ e'\ [\mathtt{inl}(x_1) \to e_1]\ [\mathtt{inr}(x_2) \to e_2]\end{array}}$$

$$\frac{v\ \mathsf{val}}{\mathtt{case}\ \mathtt{inl}_{\tau_1+\tau_2}(v)\ [\mathtt{inl}(x_1) \to e_1]\ [\mathtt{inr}(x_2) \to e_2] \longmapsto e_1[v/x_1]}$$

$$\frac{v\ \mathsf{val}}{\mathtt{case}\ \mathtt{inr}_{\tau_1+\tau_2}(v)\ [\mathtt{inl}(x_1) \to e_1]\ [\mathtt{inr}(x_2) \to e_2] \longmapsto e_2[v/x_2]}$$

Remark 6.9. Just as the type annotations on `lambda`s do not play a role in their operational semantics, neither do the type annotations above.

Remark 6.10. One could again consider a call-by-name operational semantics.

The usual properties hold.

Lemma 6.11 (Finality of values)**.** *If $v\ \mathsf{val}$ then there is no e' such that $v \longmapsto e'$.*

Lemma 6.12 (Determinacy)**.** *If $e \longmapsto e'$ and $e \longmapsto e''$ then $e' = e''$.*

Lemma 6.13. *If $\cdot \vdash e\ \mathsf{tm}$ and $e \longmapsto e'$ then $\cdot \vdash e'\ \mathsf{tm}$.*

4 Type safety

Well-typed terms still don't get stuck. We once again prove canonical forms (Lemma 6.16), progress (Lemma 6.17), substitution (Lemma 6.18), and preservation (Lemma 6.19). Although the shape of the proof is unchanged, some of the new cases are interesting; see for instance the `void` clause of Lemma 6.16.

Theorem 6.14 (Type safety)**.**

1. *If $\cdot \vdash e : \tau$ and $e \longmapsto e'$ then $\cdot \vdash e' : \tau$.*
2. *If $\cdot \vdash e : \tau$ then either $e\ \mathsf{val}$ or $e \longmapsto e'$.*

Remark 6.15. In the following proofs, we are going to skip over the cases relating to unit, function, and product types from the previous lecture. This is **potentially erroneous** because some of those cases relied on *inversion* to say that the only rule that can apply is such-and-such, and inversion lemmas can be disrupted whenever we add new rules. You can trust me that the previous proof cases continue to work—I am a doctor—or you can check them yourself. At a high level, the reason is that none of our new rules mention the old type or term constructors.

Lemma 6.16 (Canonical forms)**.** *Suppose* $\cdot \vdash v : \tau$ *and* v val. *Then:*

1. *If* $\tau = \mathtt{unit}$, $\tau = \tau_1 \to \tau_2$, *or* $\tau = \tau_1 \times \tau_2$, *refer to the previous lecture.*
2. *It is impossible that* $\tau = \mathtt{void}$.
3. *If* $\tau = \tau_1 + \tau_2$, *then either:*
   (a) $v = \mathtt{inl}_{\tau_1+\tau_2}(v_1)$ *where* v_1 val *and* $\cdot \vdash v_1 : \tau_1$, *or*
   (b) $v = \mathtt{inr}_{\tau_1+\tau_2}(v_2)$ *where* v_2 val *and* $\cdot \vdash v_2 : \tau_2$.

Proof. By inversion on $\cdot \vdash v : \tau$ and v val. □

Lemma 6.17 (Progress)**.** *If* $\cdot \vdash e : \tau$ *then either* e val *or* $e \longmapsto e'$ *for some* e'.

Proof. By rule induction on $\cdot \vdash e : \tau$, focusing on the new cases.

- Case $\dfrac{\cdot \vdash e : \mathtt{void}}{\cdot \vdash \mathtt{abort}_\tau(e) : \tau}$ void-ELIM :

  $\mathtt{abort}_\tau(e)$ is never a value, so we show that it takes a step. By the IH, either e val or $e \longmapsto e'$. Lemma 6.16 says that terms of type void cannot be values, so the only possibility is $e \longmapsto e'$, in which case $\mathtt{abort}_\tau(e) \longmapsto \mathtt{abort}_\tau(e')$.

- Cases $\dfrac{\cdot \vdash e : \tau_1}{\cdot \vdash \mathtt{inl}_{\tau_1+\tau_2}(e) : \tau_1 + \tau_2}$ +-INTRO$_1$, $\dfrac{\cdot \vdash e : \tau_2}{\cdot \vdash \mathtt{inr}_{\tau_1+\tau_2}(e) : \tau_1 + \tau_2}$ +-INTRO$_2$:

  By the IH, either e val or $e \longmapsto e'$. If e val then this term is also a value; if $e \longmapsto e'$ then this term also takes a step.

- Case $\dfrac{\cdot \vdash e : \tau_1 + \tau_2 \qquad x_1 : \tau_1 \vdash e_1 : \tau \qquad x_2 : \tau_2 \vdash e_2 : \tau}{\cdot \vdash \mathtt{case}\ e\ [\mathtt{inl}(x_1) \to e_1]\ [\mathtt{inr}(x_2) \to e_2] : \tau}$ +-ELIM :

  By the IH on e, either e val or $e \longmapsto e'$. (Note that we don't have IHs for e_1 or e_2, because they are not closed!) If $e \longmapsto e'$, then the entire case steps to $\mathtt{case}\ e'\ [\mathtt{inl}(x_1) \to e_1]\ [\mathtt{inr}(x_2) \to e_2]$. If e val, then by Lemma 6.16 there are two possibilities:

– $e = \mathtt{inl}_{\tau_1+\tau_2}(v_1)$ where v_1 val and $\cdot \vdash v_1 : \tau_1$, in which case the following step occurs:

$$\frac{v_1 \text{ val}}{\mathtt{case}\ \mathtt{inl}_{\tau_1+\tau_2}(v_1)\ [\mathtt{inl}(x_1) \to e_1]\ [\mathtt{inr}(x_2) \to e_2] \longmapsto e_1[v_1/x_1]}$$

– $e = \mathtt{inr}_{\tau_1+\tau_2}(v_2)$ where v_2 val and $\cdot \vdash v_2 : \tau_2$, in which case the following step occurs:

$$\frac{v_2 \text{ val}}{\mathtt{case}\ \mathtt{inr}_{\tau_1+\tau_2}(v_2)\ [\mathtt{inl}(x_1) \to e_1]\ [\mathtt{inr}(x_2) \to e_2] \longmapsto e_2[v_2/x_2]}$$

□

Lemma 6.18 (Substitution)**.** *If* $\Gamma, x : \tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$ *then* $\Gamma \vdash e[e'/x] : \tau$.

Proof. By rule induction on $\Gamma, x : \tau' \vdash e : \tau$, where nothing different happens in the new cases. □

Lemma 6.19 (Preservation)**.** *If* $\cdot \vdash e : \tau$ *and* $e \longmapsto e'$ *then* $\cdot \vdash e' : \tau$.

Proof. By rule induction on $e \longmapsto e'$, focusing on the new cases.

- The four congruence reductions (for `abort`, `inl`, `inr`, and `case`) follow the same pattern as in the previous lecture.
- Case $\dfrac{v \text{ val}}{\mathtt{case}\ \mathtt{inl}_{\tau_1+\tau_2}(v)\ [\mathtt{inl}(x_1) \to e_1]\ [\mathtt{inr}(x_2) \to e_2] \longmapsto e_1[v/x_1]}$:

  We must show $\cdot \vdash e_1[v/x_1] : \tau$. By successive inversion on the typing judgment $\cdot \vdash \mathtt{case} \ldots : \tau$,

  $$\frac{\dfrac{\cdot \vdash v : \tau_1}{\cdot \vdash \mathtt{inl}_{\tau_1+\tau_2}(v) : \tau_1 + \tau_2} \qquad x_1 : \tau_1 \vdash e_1 : \tau \qquad x_2 : \tau_2 \vdash e_2 : \tau}{\cdot \vdash \mathtt{case}\ \mathtt{inl}_{\tau_1+\tau_2}(v)\ [\mathtt{inl}(x_1) \to e_1]\ [\mathtt{inr}(x_2) \to e_2] : \tau}$$

  The result follows by Lemma 6.18, $x_1 : \tau_1 \vdash e_1 : \tau$, and $\cdot \vdash v : \tau_1$.
- Case $\dfrac{v \text{ val}}{\mathtt{case}\ \mathtt{inr}_{\tau_1+\tau_2}(v)\ [\mathtt{inl}(x_1) \to e_1]\ [\mathtt{inr}(x_2) \to e_2] \longmapsto e_2[v/x_2]}$:

  Similar to previous case. □

Recall that type safety tells us that any closed term of type τ must either evaluate (in some finite number of steps) to a value of type τ, or else diverge (fail to terminate). Because there are no values of type `void`, we conclude from type safety that all closed terms of type `void` must diverge.

Remark 6.20. In fact, because termination holds for this language, there are *no* closed terms of type `void`, although this fact is not a corollary of type safety.

Theorem 6.21 (Termination)**.** *If* $\cdot \vdash e : \tau$ *then* $e \longmapsto^* v$ *where* v val.

Exercise 6.22. Using the typing rules from the previous lecture as well as this lecture, attempt to prove "directly" that it is impossible for $\cdot \vdash e : \texttt{void}$ to hold. What goes wrong?

5 Named finite sums and products

We have covered the nullary and binary cases of product and sum types, but they generalize directly to any finite arity:

- If we primitively added a type constructor $\tau_1 \times \cdots \times \tau_n$ (where n is any natural number), it would have one "n-tuple" introduction form and n "ith projection" elimination forms.
- If we primitively added a type constructor $\tau_1 + \cdots + \tau_n$ (where n is any natural number), it would have n "ith inclusion" introduction forms and one "$n$ary case split" elimination form.

Harper [Har16, Chapters 10–11] lists rules for finite product and sum types.

Most programming languages have finite product types in one guise or another; fewer programming languages have finite sum types, although they are common in functional programming languages. Generally, these finite product/sum types are *named*, in the sense that programmers can give a custom name to a particular finite product/sum type and its projections/inclusions.

5.1 Named products: records/structs

Named product types are usually called *record types* or *structs*, and their components are usually called *fields*. Even C has structs:

```
struct point_t {
  int x;
  int y;
};
```

```
struct point_t p = { 1, 2 };
int px = p.x; // 1
int py = p.y; // 2
```

So does Racket (although they are not typed, of course):

```
(struct point (x y))

(define p (point 1 2))
(point-x p)
(point-y p)
```

5.2 Named sums: tagged unions

Named sums are most common in typed functional programming languages. In Haskell, one might write:

```
data Value = Str String | Bln Bool

show :: Value -> String
show x = case x of
           Str s -> s
           Bln b -> if b then "true" else "false"
```

In Racket, you have used the idea of tagged unions (without types) many times:

```
(define (expr->string e)
  (match e
    [`(int ,x) (number->string x)]
    [`(str ,y) y]))
```

5.3 Nesting products and sums

It is particularly profitable to *combine* named finite products and sums. As a first example, the type of booleans can be recovered as a named sum of products:

$$\begin{aligned} \mathsf{bool} &\coloneqq \mathsf{unit} + \mathsf{unit} \\ \mathsf{true} &\coloneqq \mathsf{inl}_{\mathsf{unit+unit}}(()) \\ \mathsf{false} &\coloneqq \mathsf{inr}_{\mathsf{unit+unit}}(()) \\ \mathsf{if}(e, e', e'') &\coloneqq \mathsf{case}\ e\ [\mathsf{inl}(\_) \to e']\ [\mathsf{inr}(\_) \to e''] \end{aligned}$$

Note that the variables are not used in the case, nor are they particularly useful because they have type unit.

Remark 6.23. Booleans also showcase the fact sum types are useful even when the two types are the same, and even when the two types both seem useless!

Exercise 6.24. For any types τ_1 and τ_2, define functions

$$\texttt{inl?} : (\tau_1 + \tau_2) \to \texttt{bool}$$
$$\texttt{inr?} : (\tau_1 + \tau_2) \to \texttt{bool}$$

that return `true` iff their inputs have the specified form.

Similarly, C-style *enumerations* are named n-ary sums of `unit` types:

```
enum suit_t {
  CLUBS,
  DIAMONDS,
  HEARTS,
  SPADES
};
```

Another example is the *maybe* or *option* type.

$$\begin{aligned}
\texttt{maybe}(\tau) &\coloneqq \texttt{unit} + \tau \\
\texttt{none} &\coloneqq \texttt{inl}_{\texttt{unit}+\tau}(()) \\
\texttt{just}(e) &\coloneqq \texttt{inr}_{\texttt{unit}+\tau}(e) \\
\texttt{fromMaybe}(e, e_n, x.e_j) &\coloneqq \texttt{case}\ e\ [\texttt{inl}(_) \to e_n]\ [\texttt{inr}(x) \to e_j]
\end{aligned}$$

Note that to use a term of type $\texttt{maybe}(\tau)$ one must be prepared for the possibility that it turns out to be `none`.

6 Type isomorphisms

Rather than directly defining $n$ary product and sum types for any natural number n, we could simply iterate the binary product/sum, defining $A \times B \times C \coloneqq A \times (B \times C)$. This immediately raises a few questions:

- Does it matter whether we write $A \times (B \times C)$ or $(A \times B) \times C$?
- How do we define the $n$ary product for $n = 0$ or $n = 1$? We have said that `unit` is a "nullary product," but does this work formally? Does that mean the "unary product" should be $A \times \texttt{unit}$ (or $\texttt{unit} \times A$)?

It turns out that the types $A \times (B \times C)$ and $(A \times B) \times C$ are the "same" in a certain sense, although we have to be careful what we mean by this. By Lemma 6.16, they clearly do not have the same values: the former has values $(a, (b, c))$ where a, b, c are values of the appropriate types, whereas the latter has values $((a, b), c)$. In fact, by Lemma 6.4, no term has both types. On the other hand, there is clearly a strong relationship between these types' values.

Although these types are different, there are functions back and forth:

$$\texttt{to} : (A \times (B \times C)) \to ((A \times B) \times C)$$
$$\texttt{to} \coloneqq \lambda x : A \times (B \times C).((\texttt{fst}(x), \texttt{fst}(\texttt{snd}(x))), \texttt{snd}(\texttt{snd}(x)))$$
$$\texttt{from} : ((A \times B) \times C) \to (A \times (B \times C))$$
$$\texttt{from} \coloneqq \lambda y : (A \times B) \times C.(\texttt{fst}(\texttt{fst}(y)), (\texttt{snd}(\texttt{fst}(y)), \texttt{snd}(y)))$$

and these functions intuitively "cancel out" in the sense that

$$\texttt{fromTo} \coloneqq \lambda x : A \times (B \times C).\texttt{from}\ (\texttt{to}\ x)$$

is the same as the identity function $\mathbf{I}$ on $A \times (B \times C)$, and

$$\texttt{toFrom} \coloneqq \lambda y : (A \times B) \times C.\texttt{to}\ (\texttt{from}\ y)$$

is the same as the identity function $\mathbf{I}$ on $(A \times B) \times C$.

However, it is not clear what we mean by "the same." Neither `fromTo` nor `toFrom` *evaluates* to $\mathbf{I}$: in fact, all three of these are already values. What we *can* prove is that if we apply `fromTo` and $\mathbf{I}$ to a term $\cdot \vdash p : A \times (B \times C)$, then they will produce the same results. We say that `fromTo` and $\mathbf{I}$ are *extensionally equal*.

Lemma 6.25. *For any types A, B, C and any $\cdot \vdash p : A \times (B \times C)$, `fromTo` p and $\mathbf{I}\ p$ either both diverge or both evaluate to the same value.*

Proof. By Theorem 6.14 and Lemma 6.16, p either diverges or evaluates to $(a, (b, c))$ where a, b, c are values, $\cdot \vdash a : A$, $\cdot \vdash b : B$, and $\cdot \vdash c : C$. If p diverges, then `fromTo` p and $\mathbf{I}$ both diverge because our operational semantics evaluates function arguments until they reach a value. If instead $p \longmapsto^* (a, (b, c))$, then so does $\mathbf{I}\ p$, and:

$$\begin{aligned}
& \texttt{fromTo}\ p \\
\longmapsto^* \ & \texttt{fromTo}\ (a, (b, c)) \\
\longmapsto \ & \texttt{from}\ (\texttt{to}\ (a, (b, c))) \\
\longmapsto \ & \texttt{from}\ ((\texttt{fst}((a, (b, c))), \texttt{fst}(\texttt{snd}((a, (b, c))))), \texttt{snd}(\texttt{snd}((a, (b, c)))))) \\
\longmapsto^* \ & \texttt{from}\ ((a, b), c) \\
\longmapsto \ & (\texttt{fst}(\texttt{fst}(((a, b), c))), (\texttt{snd}(\texttt{fst}(((a, b), c))), \texttt{snd}(((a, b), c)))) \\
\longmapsto^* \ & (a, (b, c))
\end{aligned}$$

□

Remark 6.26. Given Theorem 6.21 we can of course rule out the possibility that both diverge; however, it is instructive to see what we can prove without termination.

Remark 6.27. This proof only works in the by-value operational semantics. In a by-name operational semantics, it is *a priori* possible for p to terminate but for $\mathtt{fst}(p)$ and hence $\mathtt{fromTo}\ p$ to diverge; thus we cannot prove the by-name version of Lemma 6.25 without assuming termination.

If two types A, B have functions going back and forth that cancel up to extensional equality, we say that those types are *isomorphic* and write $A \cong B$. We can prove various laws of "type arithmetic": *commutativity*, *associativity*, *distributivity*, and *unit laws* where $\mathtt{unit}$ plays the role of 1 and $\mathtt{void}$ plays the role of 0:

$$\begin{aligned}
A \times B &\cong B \times A \\
A + B &\cong B + A \\
A \times (B \times C) &\cong (A \times B) \times C \\
A + (B + C) &\cong (A + B) + C \\
(A + B) \times C &\cong (A \times C) + (B \times C) \\
A \times \mathtt{void} &\cong \mathtt{void} \\
A \times \mathtt{unit} &\cong A \\
A + \mathtt{void} &\cong A
\end{aligned}$$

We can even prove arithmetic laws involving function types, where $A \to B$ behaves like exponentiation B^A:

$$\begin{aligned}
\mathtt{unit} \to C &\cong C \\
\mathtt{void} \to C &\cong \mathtt{unit} \\
(A \times B) \to C &\cong A \to (B \to C) \\
(A + B) \to C &\cong (A \to C) \times (B \to C)
\end{aligned}$$

This latter group of isomorphisms is more subtle because they involve nested function types. Let's look at the first isomorphism as an example:

$$\begin{aligned}
\mathtt{to} &: (\mathtt{unit} \to C) \to C \\
\mathtt{to} &\coloneqq \lambda x : \mathtt{unit} \to C.x\ () \\
\mathtt{from} &: C \to (\mathtt{unit} \to C) \\
\mathtt{from} &\coloneqq \lambda y : C.\lambda x : \mathtt{unit}.y
\end{aligned}$$

Let us try to prove that

$$\mathtt{fromTo} \coloneqq \lambda x : \mathtt{unit} \to C.\mathtt{from}\ (\mathtt{to}\ x))$$

is extensionally equal to $\lambda x : \mathtt{unit} \to C.x$. Suppose we are given $\cdot \vdash f : \mathtt{unit} \to C$ with $f \longmapsto^* \lambda z : \mathtt{unit}.e$. Then $(\lambda x : \mathtt{unit} \to C.x)\ f \longmapsto^* \lambda z : \mathtt{unit}.e$, but:

$$\begin{aligned} & \mathtt{fromTo}\ f \\ \longmapsto^* & \mathtt{fromTo}\ (\lambda z : \mathtt{unit}.e) \\ \longmapsto & \ \mathtt{from}\ (\mathtt{to}\ (\lambda z : \mathtt{unit}.e)) \\ \longmapsto^* & \mathtt{from}\ (e[()/z]) \end{aligned}$$

Without Theorem 6.21 we cannot be sure that $e[()/z]$ terminates. But let us proceed under the assumption that $e[()/z] \longmapsto^* v$ where v val. Then:

$$\begin{aligned} & \mathtt{from}\ (e[()/z]) \\ \longmapsto^* & \mathtt{from}\ v \\ \longmapsto & \ \lambda x : \mathtt{unit}.v \neq \lambda z : \mathtt{unit}.e \end{aligned}$$

In this case, `fromTo` and **I** are *not* extensionally equal—even assuming Theorem 6.21—because they produce different values $\lambda x : \mathtt{unit}.v$ and $\lambda z : \mathtt{unit}.e$. On the other hand, those values are *extensionally equal*, because whenever we apply both to a term of type `unit`, that term (by Theorem 6.21 and Lemma 6.16) will evaluate to $()$ and thus both sides will evaluate to v. That is, our two functions are extensionally equal up to extensional equality.

Lemma 6.28. *For any type C, if we apply $\cdot \vdash \mathtt{fromTo}, \mathbf{I} : (\mathtt{unit} \to C) \to (\mathtt{unit} \to C)$ to a term $\cdot \vdash f : \mathtt{unit} \to C$, and then apply each result to a term $\cdot \vdash e : \mathtt{unit}$, then we will obtain equal values of type C.*

Remark 6.29. As we will see later, the correct fully general notion of extensional equality is given by structural recursion on types: two functions are extensionally equal at type $A \to B$ if, when given two extensionally equal terms at type A, they produce two extensionally equal terms at type B.

remark about denotational semantics here (cardinality)

References

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Second Edition. Cambridge University Press, 2016. ISBN: 9781107150300. DOI: 10.1017/CBO9781316576892.