Lecture Notes 4
Binding

Carlo Angiuli

B522: PL Foundations
February 5, 2025

In this lecture, we extend our notion of inductive definition to account for *variable binding*; the resulting notion of "inductive definition with binding" is often known as *abstract binding trees* (ABTs) or *abstract syntax with binding*.

Note that our textbook introduces the notions of abstract syntax and binding simultaneously. These lecture notes correspond to Chapters 1, 2, and 3 of Harper [Har16]; my explanation of structural recursion and rule induction for ABTs follows that of Pitts [Pit06]. We will resume our discussion of type systems in subsequent lectures.

1 Untyped λ-calculus (first try)

Virtually every reasonable programming language includes *variables* in its syntax, at minimum as the "formal parameters" of function definitions. Since you are already familiar with Racket and `lambda`, our first example of a language with variables will be the famed and minimalist *untyped λ-calculus*.

Remark 4.1. The untyped λ-calculus was invented in the 1930s by Alonzo Church as a notation (on paper) for computable (i.e., partial recursive) functions; it subsequently influenced the development of the Lisp programming language in the 1950s, from which Racket descends. Perhaps surprisingly, despite having no obvious "data," the untyped λ-calculus is a universal model of computation equivalent to Turing machines, which are complex in comparison.

The point of today's lecture will be to introduce the subtleties of variables and to see why we should extend our notion of inductive definitions to support binding natively. But let's start by **temporarily** considering an encoding of the λ-calculus as an ordinary inductive definition:

$$\begin{array}{lll} \textit{Variables?} & v ::= & x \mid y \mid z \mid \cdots \\ \textit{Terms?} & e ::= & v \mid \texttt{lambda}(v, e) \mid \texttt{app}(e, e) \end{array}$$

Remark 4.2. In Racket, `lambda`(v, e) is written `(lambda (`v`)` e`)` and `app`(e, e') is written `(`e e'`)`. (For us, function application is "just" another binary term former, so we need a notation for it.) Unlike in Racket, all `lambda`s in the untyped λ-calculus (and in this class generally) have exactly one argument. We can represent multi-argument functions by *currying*, or nesting `lambda`s and applying them multiple times: `(lambda (x) (lambda (y) (+ x y)))`.

There is already a lot to unpack in our inductive definition above:

- Every variable should be a term, but terms and variables are not interchangeable. Inside of `lambda`$(x, \dots)$ we can use x anywhere a term could go. But there are some places where only a variable is permitted, in particular the first argument to `lambda`. (You can't write `(lambda ((+ x y)) ...)`.)

- We want an infinite number of variables. To see why, suppose there were only two variables, x and y, and suppose we are writing a two-argument function that returns a function:

  ```
  (lambda (x) (lambda (y) (lambda (?) (... x y ...))))
  ```

  The formal parameter of the innermost `lambda` has to be either x or y, and in either case we lose access to one of the two existing parameters. Having an infinite number of variables lets us avoid worrying about "running out" of variables in a sufficiently large term.

- In the same way that e is a metavariable ranging over "actual" terms, v is a *metavariable* ranging over "actual" *variables*. The difference between variables and metavariables is that variables are actually part of the programming language we're specifying, whereas metavariables are just a notation or technical device used in writing the specification.

  Unfortunately, it's common to use x, y, z for variables *and* metavariables ranging over variables, and for the collection of variables to be left implicit. This leads to the following commonly-seen notation:

$$\textit{Terms??} \quad e ::= \quad x \mid \texttt{lambda}(x, e) \mid \texttt{app}(e, e)$$

All that said, there are three important aspects of variables that our inductive definition above does not capture (yet, at least): context-sensitivity, α-equivalence, and substitution.

- **Context-sensitivity**: Our grammar says that every variable is always a valid term, but a well-formed program should only contain variables that are already "in scope": $\texttt{lambda}(x, x)$ is a valid term, but x isn't (at top-level). The variable x should be a term if and only if there is an enclosing $\texttt{lambda}(x, \ldots)$. As a result, the λ-calculus is not context-free and is thus not expressible as a BNF grammar. In Section 2 we will show how to define a *context-sensitive* term well-formedness judgment.

- α-**equivalence**: Variables have names because we need to be able to tell them apart; inside $\texttt{lambda}(x, \texttt{lambda}(y, \ldots))$, x and y refer to the first and second inputs to this two-argument function respectively. However, the names of variables should carry no other (formal) meaning: this function should behave identically if its variables were instead named z and w.

  In Section 3 we will define *α-equivalence*, an equivalence relation on terms which makes this intuition precise. In programming language theory, we insist that α-equivalent terms are always treated identically.

- **Substitution**: Variables are formal stand-ins for arbitrary (or "varying") terms, often function arguments. As soon as we receive those inputs, we will want to replace, or *substitute*, the variable with the given term.

2 Judgments in context

As we mentioned above, the variable x should not be considered a term in general, but it *should* be considered a term inside of an enclosing $\texttt{lambda}(x, \ldots)$. To model this, we consider the grammar of terms to have no variables, but to be *extended* by the variable x whenever we go inside $\texttt{lambda}(x, \ldots)$:

- $\texttt{lambda}(x, \texttt{lambda}(y, \texttt{app}(x, y)))$ is a term because, assuming x is a term,
  - $\texttt{lambda}(y, \texttt{app}(x, y))$ is a term because, assuming y is a term,
    - $\texttt{app}(x, y)$ is a term because
      - x is a term
      - y is a term

These "assumptions" are localized to all the bullet points contained (immediately or transitively) inside the `lambda` bullet point in which they are made.

Exercise 4.3. Is $\texttt{lambda}(x, \texttt{app}(x, y))$ a term under no assumptions?

Exercise 4.4. Is $\texttt{lambda}(x, \texttt{app}(\texttt{lambda}(y, x), y))$ a term under no assumptions?

We extend the judgmental machinery from the previous lecture to keep track of the (unordered) set of currently-active assumptions $x\ \mathsf{tm}, y\ \mathsf{tm}, \dots$, which is known as the *context* and written Γ. We write

$$\Gamma \vdash e\ \mathsf{tm}$$

for the assertion that e is a term, under the set of hypotheses Γ.

Remark 4.5. Judgments that are relative to a context are called *hypothetical judgments*; judgments without a context are called *categorical judgments.* The $\vdash$ symbol (`\vdash`) is called a *turnstile*,[5] and the empty context is often written $\cdot$ (`\cdot`).

We define $\Gamma \vdash e\ \mathsf{tm}$ by the following inference rules. In these rules, Γ is a metavariable ranging over all possible contexts, which in this system are unordered lists of the form $x\ \mathsf{tm}, y\ \mathsf{tm}, \dots$.

$$\frac{}{\Gamma, x\ \mathsf{tm} \vdash x\ \mathsf{tm}} \qquad \frac{\Gamma, x\ \mathsf{tm} \vdash e\ \mathsf{tm}}{\Gamma \vdash \mathtt{lambda}(x, e)\ \mathsf{tm}} \qquad \frac{\Gamma \vdash e\ \mathsf{tm} \qquad \Gamma \vdash e'\ \mathsf{tm}}{\Gamma \vdash \mathsf{app}(e, e')\ \mathsf{tm}}$$

We can now rewrite the bulleted list above as a derivation tree:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{x\ \mathsf{tm}, y\ \mathsf{tm} \vdash x\ \mathsf{tm}} \qquad \cfrac{}{x\ \mathsf{tm}, y\ \mathsf{tm} \vdash y\ \mathsf{tm}}}{x\ \mathsf{tm}, y\ \mathsf{tm} \vdash \mathsf{app}(x, y)\ \mathsf{tm}}}{x\ \mathsf{tm} \vdash \mathtt{lambda}(y, \mathsf{app}(x, y))\ \mathsf{tm}}}{\cdot \vdash \mathtt{lambda}(x, \mathtt{lambda}(y, \mathsf{app}(x, y)))\ \mathsf{tm}}$$

Let's examine each rule carefully. The first rule, called the *variable rule*, says that if our context (collection of hypotheses) includes $x\ \mathsf{tm}$, then $x\ \mathsf{tm}$. Note that contexts are unordered, so when we write "$\Gamma, x\ \mathsf{tm}$" we really mean that the context is of the form *[some possibly-empty collection of hypotheses], and* $x\ \mathsf{tm}$.

Remark 4.6. Sometimes people write the variable rule in one of these styles:

$$\frac{x\ \mathsf{tm} \in \Gamma}{\Gamma, x\ \mathsf{tm} \vdash x\ \mathsf{tm}} \qquad \frac{}{\Gamma, x\ \mathsf{tm}, \Gamma' \vdash x\ \mathsf{tm}}$$

The second rule says that in order to show that $\mathtt{lambda}(x, e)$ is a term in context Γ, we must show that e is a term in context Γ extended with the additional hypothesis $x\ \mathsf{tm}$. The third rule says that in order to show that $\mathsf{app}(e, e')$ is a term in context Γ, we must show that e and e' are both terms in context Γ.

[5] Some students may have previously encountered turnstiles in the film *Tenet* (2020).

Remark 4.7. The phrase "e is a term" is now ambiguous, because this depends on which variables are in the context. ($\mathtt{lambda}(x, x)$ is a term in any context, but x or $\mathtt{lambda}(y, x)$ are only terms in contexts containing x tm.) We say that e is a *closed term* if it is a term in the empty context. We will try to reserve the ambiguous phrase "e is a term" for situations in which the intended context (empty or otherwise) is clear from…context.

Remark 4.8. The horizontal line and the turnstile can both can be pronounced "if…then…" but they have different meanings. We write $\mathcal{J}_1 \vdash \mathcal{J}_2$ only when $\mathcal{J}_1$ is a *categorical judgment about a variable*, which is being *locally* hypothesized in order to derive $\mathcal{J}_2$. In contrast, writing $\frac{\mathcal{J}_1}{\mathcal{J}_2}$ means that whenever $\mathcal{J}_1$ is *globally* true, $\mathcal{J}_2$ is also globally true; here $\mathcal{J}_1, \mathcal{J}_2$ are usually hypothetical judgments.

2.1 Structural properties

We have said that the meaning of a hypothetical judgment $\mathcal{J}_1, \dots, \mathcal{J}_n \vdash \mathcal{J}$ is "assuming that $\mathcal{J}_1, \dots, \mathcal{J}_n$ hold, then $\mathcal{J}$ holds." There are certain *structural properties* that every[6] hypothetical judgment ought to satisfy in order for this interpretation to make sense:

- **Reflexivity**: $\mathcal{J} \vdash \mathcal{J}$. ("If $\mathcal{J}$ then $\mathcal{J}$.") We typically include a variable rule which states this explicitly.
- **Weakening**: If $\mathcal{J}_1, \dots, \mathcal{J}_n \vdash \mathcal{J}$ then $\mathcal{J}_1, \dots, \mathcal{J}_n, \mathcal{J}_{n+1} \vdash \mathcal{J}$. ("You can add unnecessary hypotheses.") We typically check that this principle is admissible.
- **Exchange**: The order of $\mathcal{J}_1, \dots, \mathcal{J}_n$ does not affect the truth of the judgment. We have asserted that contexts are unordered, although sometimes people say that contexts are ordered and check that this principle is admissible.
- **Substitution**: See Section 4 and Lemma 4.27.

Theorem 4.9 (Weakening)**.** *If* $\Gamma \vdash e\ \mathsf{tm}$ *then* $\Gamma, y\ \mathsf{tm} \vdash e\ \mathsf{tm}$.

Proof. We prove that weakening is admissible by rule induction.

- Case $\dfrac{}{\Gamma, x\ \mathsf{tm} \vdash x\ \mathsf{tm}}$:

  We must show $\Gamma, x\ \mathsf{tm}, y\ \mathsf{tm} \vdash x\ \mathsf{tm}$, which follows from the variable rule.

[6]There are some exceptions, known as *substructural judgments*.

- Case $\dfrac{\Gamma, x \ \mathsf{tm} \vdash e \ \mathsf{tm}}{\Gamma \vdash \mathtt{lambda}(x, e) \ \mathsf{tm}}$:

  We must show $\Gamma, y \ \mathsf{tm} \vdash \mathtt{lambda}(x, e) \ \mathsf{tm}$. Our inductive hypothesis is $\Gamma, x \ \mathsf{tm}, y \ \mathsf{tm} \vdash e \ \mathsf{tm}$, and the result follows by applying the `lambda` rule.

- Case $\dfrac{\Gamma \vdash e \ \mathsf{tm} \qquad \Gamma \vdash e' \ \mathsf{tm}}{\Gamma \vdash \mathsf{app}(e, e') \ \mathsf{tm}}$:

  We must show $\Gamma, y \ \mathsf{tm} \vdash \mathsf{app}(e, e') \ \mathsf{tm}$. Our inductive hypotheses are $\Gamma, y \ \mathsf{tm} \vdash e \ \mathsf{tm}$ and $\Gamma, y \ \mathsf{tm} \vdash e' \ \mathsf{tm}$. We apply the app rule. □

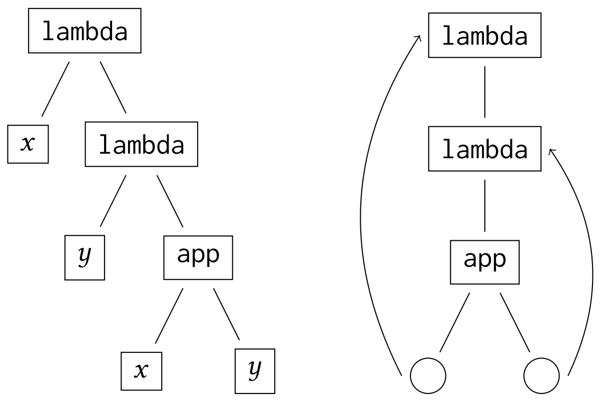
3 α-equivalence

Every `lambda` gives its formal parameter a name (such as x) which we may then use within the body of the `lambda` in order to refer to that particular formal parameter. Programming language theorists regard the name x as merely a *notation* linking the formal parameter to its mentions; from this perspective, $\mathtt{lambda}(x, x)$ and $\mathtt{lambda}(y, y)$ are two different notations for the exact same term.

Put another way, we should think of variables not as names but as *pointers* to the `lambda`s that introduce them. Below are two depictions of the term

$$\mathtt{lambda}(x, \mathtt{lambda}(y, \mathsf{app}(x, y)))$$

first as an abstract syntax tree with named variables, and second as what is sometimes called an *abstract binding tree*, where variables are pointers to `lambda`s.



Two terms are *α-equivalent* if they differ only by the names of their bound variables: that is, if they pictorially correspond to the same abstract binding trees.

Remark 4.10. By the rules in Section 2, variable nodes must point to a `lambda` node that is the variable's ancestor.

Remark 4.11. We say that the term $\texttt{lambda}(x, e)$ *binds* the variable x, that x is a *binder*, and occurrences of x inside e are *bound*. A variable that is not bound anywhere is called *free*. For example, in the subterm $\texttt{lambda}(y, \texttt{app}(x, y))$ above, the leftmost y is a binder binding the rightmost y, and the x is free.

Now that we have the idea of α-equivalence, let's make it precise.

Definition 4.12. Write $\mathbb{V}$ for the set of variables. For any $x, y \in \mathbb{V}$ and $\Gamma \vdash e \ \texttt{tm}$ we inductively define the operation of *swapping x and y in e*, written $e[x \leftrightarrow y]$:

$$z[x \leftrightarrow y] = \begin{cases} y & \text{if } z = x \\ x & \text{if } z = y \\ z & \text{if } z \neq x, z \neq y \end{cases}$$

$$\texttt{lambda}(z, e)[x \leftrightarrow y] = \texttt{lambda}(z[x \leftrightarrow y], e[x \leftrightarrow y])$$

$$\texttt{app}(e, e')[x \leftrightarrow y] = \texttt{app}(e[x \leftrightarrow y], e'[x \leftrightarrow y])$$

Definition 4.13. For any $\Gamma \vdash e \ \texttt{tm}$ and $\Gamma \vdash e' \ \texttt{tm}$, we inductively define the relation $e =_\alpha e'$ stating that *e and e' are α-equivalent*, as follows:

- $x =_\alpha x$ for every variable x,
- $\texttt{lambda}(x, e) =_\alpha \texttt{lambda}(x', e')$ if for some $z \in \mathbb{V}$ that appears nowhere in x, e, x', e', we have $e[x \leftrightarrow z] =_\alpha e'[x' \leftrightarrow z]$, and
- $\texttt{app}(e_1, e_1') =_\alpha \texttt{app}(e_2, e_2')$ if $e_1 =_\alpha e_1'$ and $e_2 =_\alpha e_2'$.

Note that Definition 4.12 is totally oblivious to the binding structure of terms, but Definition 4.13 is not: the second clause of Definition 4.13 tells us that in the term $\texttt{lambda}(x, e)$, occurrences of x inside of e refer to *this* x.

Example 4.14. We derive an α-equivalence using Definitions 4.12 and 4.13:

$$\begin{aligned} \texttt{lambda}(x, \texttt{lambda}(x, x)) &=_\alpha \texttt{lambda}(x, \texttt{lambda}(y, y)) \\ \texttt{lambda}(x, x)[x \leftrightarrow z] &=_\alpha \texttt{lambda}(y, y)[x \leftrightarrow z] \\ \texttt{lambda}(z, z) &=_\alpha \texttt{lambda}(y, y) \\ x[x \leftrightarrow w] &=_\alpha y[y \leftrightarrow w] \\ w &=_\alpha w \end{aligned}$$

Remark 4.15. The second clause applies regardless of whether x and x' are equal. We can always find some $z \in \mathbb{V}$ that appears nowhere in x, e, x', e' because $\mathbb{V}$ is infinite and e, e' contain only finitely many variables.

Remark 4.16. α-equivalence is an equivalence relation on terms: it is reflexive, symmetric, and transitive.

Commandment 4.17. Thou shalt treat α-equivalent terms in the same way.

Commandment 4.17 is more subtle than it may appear at first glance.

Definition 4.18. For any $\Gamma \vdash e\ \mathsf{tm}$ we define the set of *free variables* of e (also known as the *support* of e) $\mathsf{fv}(e)$ inductively as follows:

$$\begin{aligned} \mathsf{fv}(x) &= \{x\} \\ \mathsf{fv}(\texttt{lambda}(x, e)) &= \mathsf{fv}(e) \setminus \{x\} \\ \mathsf{fv}(\mathsf{app}(e, e')) &= \mathsf{fv}(e) \cup \mathsf{fv}(e') \end{aligned}$$

Definition 4.18 respects α-equivalence. I'm not going to prove this, but working through the following examples may help illustrate why it's true:

$$\begin{aligned} \mathsf{fv}(\texttt{lambda}(x, x)) &= \emptyset \\ \mathsf{fv}(\texttt{lambda}(y, \mathsf{app}(x, y))) &= \{x\} \\ \mathsf{fv}(\texttt{lambda}(z, \mathsf{app}(x, z))) &= \{x\} \end{aligned}$$

On the other hand, it would violate Commandment 4.17 to define an operation $\mathsf{vars}(e)$ which returns the set of all variables contained in a term,

$$\begin{aligned} \mathsf{vars}(x) &= \{x\} \\ \mathsf{vars}(\texttt{lambda}(x, e)) &= \{x\} \cup \mathsf{vars}(e) \\ \mathsf{vars}(\mathsf{app}(e, e')) &= \mathsf{vars}(e) \cup \mathsf{vars}(e') \end{aligned}$$

because vars behaves differently on α-equivalent terms: $\mathsf{vars}(\texttt{lambda}(x, x)) = \{x\}$ and $\mathsf{vars}(\texttt{lambda}(y, y)) = \{y\}$. (Nor can we define an operation computing the set of *bound* variables in a term.)

We will get the hang of this throughout the semester, but one way to avoid making mistakes is to follow what is known as the *Barendregt convention*: when writing down terms, always give every binder a name that is different from every other binder in the term, and also from every free variable.

Remark 4.19. Why can't we just insist that every binder always has a different name? Unfortunately, once we start talking about operational semantics we will see that higher-order functions can duplicate `lambda` terms and thus binders.

4 Substitution

If binders are just notation for the formal parameters of functions, then variables are fundamentally *placeholders* for the concrete inputs that those functions eventually receive. When we eventually apply $\texttt{lambda}(x, e_1)$ to an input e, the result should be e_1 but with every x inside it *substituted* with e.

The subtlety here is that a purely textual substitution would be the wrong notion of substitution: we want to replace every occurrence of x that points to this formal parameter x, skipping any occurrence of x that refers to something else. On the flip side, we also want to ensure that we do not accidentally change the referent of any other variables in e or e_1. In other words, this operation must respect α-equivalence in all its inputs.

Definition 4.20 (Capture-avoiding substitution)**.** Given terms e_1 and e and a variable x, we define inductively an operation $e_1[e/x]$ ("e_1 with e for x") that replaces occurrences of x in e_1 with e:

$$\begin{aligned}
y[e/x] &= \begin{cases} e & \text{if } y = x \\ x & \text{if } y \neq x \end{cases} \\
\texttt{lambda}(y, e_1)[e/x] &= \texttt{lambda}(y, e_1[e/x]) \quad \text{if } y \neq x \text{ and } y \notin \mathsf{fv}(e) \\
\mathsf{app}(e_1, e_2)[e/x] &= \mathsf{app}(e_1[e/x], e_2[e/x])
\end{aligned}$$

Remark 4.21. Although capture-avoiding substitution is one of the most important operations in programming language theory, nobody can agree on a notation for it. Some notations include $[e/x]e_1$, $e_1[x/e]$, $e_1[x \coloneqq e]$, $(x \coloneqq e)e_1$, $[x \mapsto e]e_1$, …

The first and third cases are relatively straightforward: we replace x with e, leave non-x variables alone, and recur into applications. The second case raises several questions: why are these two side conditions necessary, and how do we define substitution into a $\texttt{lambda}$ in the case that these conditions don't hold?

To answer the second question first, these three cases do actually cover all possible terms, because every $\texttt{lambda}$ is α-equivalent to one whose binder is not in $\{x\} \cup \mathsf{fv}(e)$ (because $\mathbb{V}$ is infinite). This definition of $e_1[e/x]$ may be called *α-structurally recursive* [Pit06] on e_1, a variation of structural recursion that takes α-equivalence into account.

Remark 4.22. Analogously, one can fully define the addition of fractions as:

$$\frac{a}{b} + \frac{a'}{b} = \frac{a + a'}{b}$$

In order to add two fractions whose denominators differ, one must first convert them into equivalent fractions with a common denominator.

The two side conditions themselves are necessary to ensure that the definition respects α-equivalence. The first one is straightforward: it ensures that we stop at any subterm of e_1 that "shadows" or "re-binds" x:

$$\mathtt{app}(\mathtt{lambda}(x, \mathtt{lambda}(x, x)), e) \text{ “=” } \mathtt{lambda}(x, x)[e/x] \neq \mathtt{lambda}(x, e)$$

Intuitively, our substitution is meant to replace every x that refers to the first/outer x in the original term, but the third x refers to the second/inner x so it should not be replaced. Another way to look at it is that the original term is α-equivalent to a term which clearly should evaluate to the identity function, not $\mathtt{lambda}(x, e)$:

$$\begin{aligned}\mathtt{app}(\mathtt{lambda}(x, \mathtt{lambda}(y, y)), e) &\text{ “=” } \mathtt{lambda}(y, y)[e/x] \\ &= \mathtt{lambda}(y, y[e/x]) \\ &= \mathtt{lambda}(y, y)\end{aligned}$$

The second side condition is trickier: it ensures that free variables in e do not accidentally become "captured" by binders in e_1.

$$\mathtt{app}(\mathtt{lambda}(x, \mathtt{lambda}(z, x)), z) \text{ “=” } \mathtt{lambda}(z, x)[z/x] \neq \mathtt{lambda}(z, z)$$

Here, the second/argument z starts out as a free variable (perhaps it is bound by some `lambda` on the outside) but it ends up pointing instead to the first z. The original term is again α-equivalent to a term which clearly should evaluate to $\mathtt{lambda}(y, z)$, not the identity function.

$$\begin{aligned}\mathtt{app}(\mathtt{lambda}(x, \mathtt{lambda}(y, x)), z) &\text{ “=” } \mathtt{lambda}(y, x)[z/x] \\ &= \mathtt{lambda}(y, x[z/x]) \\ &= \mathtt{lambda}(y, z)\end{aligned}$$

Remark 4.23. The second side condition ($y \notin \mathsf{fv}(e)$) is the *capture-avoiding* part of capture-avoiding substitution. It is vacuously satisfied whenever e is closed.

The "=" relation mentioned throughout this section has a name: it is called β-equivalence of untyped λ-terms.

Definition 4.24. For any $\Gamma \vdash e\ \mathsf{tm}$ and $\Gamma \vdash e'\ \mathsf{tm}$, we inductively define the relation $e =_\beta e'$ stating that *e and e' are β-equivalent*, as follows:

1. $x =_\beta x$ for every variable x,
2. $\mathtt{lambda}(x, e) =_\beta \mathtt{lambda}(x, e')$ if $e =_\beta e'$,
3. $\mathtt{app}(e_1, e_1') =_\beta \mathtt{app}(e_2, e_2')$ if $e_1 =_\beta e_1'$ and $e_2 =_\beta e_2'$, and

4. $\mathtt{app}(\mathtt{lambda}(x, e_1), e) =_\beta e_1[e/x]$.

Remark 4.25. In accordance with Commandment 4.17, β-equivalence is defined over α-equivalence classes of terms. In particular, in clause (2) of Definition 4.24 we choose the same bound variable name for both `lambda`s.

Remark 4.26. Clause (4) of Definition 4.24 is the interesting one; clauses (1–3) simply state that β-equivalence is a *congruence relation*, i.e. that it is respected by all term-formers. We could more tersely define β-equivalence as the least congruence generated by clause (4).

Substitution is the final structural property of hypothetical judgments, allowing us to discharge hypotheses that we can prove: if $\mathcal{J}_1, \dots, \mathcal{J}_n, \mathcal{J}_{n+1} \vdash \mathcal{J}$ and $\mathcal{J}_1, \dots, \mathcal{J}_n \vdash \mathcal{J}_{n+1}$, then $\mathcal{J}_1, \dots, \mathcal{J}_n \vdash \mathcal{J}$. In this case, if e_1 is a term under the assumption that $x\ \mathsf{tm}$ (and possibly some other assumptions Γ), then we can remove that hypothesis by replacing x with an actual term (in context Γ).

Lemma 4.27 (Substitution)**.** *If* $\Gamma, x\ \mathsf{tm} \vdash e_1\ \mathsf{tm}$ *and* $\Gamma \vdash e\ \mathsf{tm}$ *then* $\Gamma \vdash e_1[e/x]\ \mathsf{tm}$.

5 Abstract binding trees

Add more words in this section.

The untyped λ-calculus is extremely minimalist, and we will want to consider programming languages with additional operators that bind variables. Accordingly, we extend our notion of inductive definition so that each term-former can bind zero or more variables in each of its subterms.

Before seeing the general case, let us give several examples. First, we might imagine adding `let` to our language, written $\mathtt{let}\ x = e\ \mathtt{in}\ e'$. (In Racket, `(let ([x e]) e')`.) In this term, x is a binder and x is bound in e' but not in e:

$$\frac{\Gamma \vdash e\ \mathsf{tm} \qquad \Gamma, x\ \mathsf{tm} \vdash e'\ \mathsf{tm}}{\Gamma \vdash (\mathtt{let}\ x = e\ \mathtt{in}\ e')\ \mathsf{tm}}$$

Compare this to $\mathtt{letrec}\ x = e\ \mathtt{in}\ e'$, or `(letrec ([x e]) e')`, in which x is bound in both e' *and* e:

$$\frac{\Gamma, x\ \mathsf{tm} \vdash e\ \mathsf{tm} \qquad \Gamma, x\ \mathsf{tm} \vdash e'\ \mathsf{tm}}{\Gamma \vdash (\mathtt{letrec}\ x = e\ \mathtt{in}\ e')\ \mathsf{tm}}$$

Pattern-matching constructs like `(match e1 ['() e2] [(cons x l) e3])` commonly bind multiple variables at once:

$$\frac{\Gamma \vdash e_1\ \mathsf{tm} \qquad \Gamma \vdash e_2\ \mathsf{tm} \qquad \Gamma, x\ \mathsf{tm}, \ell\ \mathsf{tm} \vdash e_3\ \mathsf{tm}}{\Gamma \vdash (\mathtt{match}\ e_1\ [\mathtt{empty} \longrightarrow e_2]\ [\mathtt{cons}(x, \ell) \longrightarrow e_2])\ \mathsf{tm}}$$

In the same way that we use BNF notation to abbreviate a collection of inference rules defining a e tm judgment, Harper [Har16] extends BNF notation to primitively support *abstract binding trees* (ABTs), or inductive definitions with variable binding that are considered modulo α-equivalence.

In ABT notation, each subterm e with a bound variable x is prefixed by x.; for example, we write $\texttt{lambda}(x.e)$ instead of $\texttt{lambda}(x, e)$, and we write $\texttt{let}(e, x.e')$ instead of $\texttt{let}(x, e, e')$ or $\texttt{let}\ x = e\ \texttt{in}\ e'$.

$$\textit{Terms} \quad e ::= \quad \texttt{lambda}(x.e) \mid \mathsf{app}(e, e)$$

This grammar is precisely a shorthand for the three rules we wrote earlier:

$$\frac{}{\Gamma, x\ \mathsf{tm} \vdash x\ \mathsf{tm}} \qquad \frac{\Gamma, x\ \mathsf{tm} \vdash e\ \mathsf{tm}}{\Gamma \vdash \texttt{lambda}(x, e)\ \mathsf{tm}} \qquad \frac{\Gamma \vdash e\ \mathsf{tm} \qquad \Gamma \vdash e'\ \mathsf{tm}}{\Gamma \vdash \mathsf{app}(e, e')\ \mathsf{tm}}$$

Note that the contexts are modified in accordance with the binding structure indicated by the ABT, and that the the variable rule appears "automatically." We also *automatically* derive notions of α-equivalence and substitution for any ABT definition, and we agree to follow Commandment 4.17.

Example 4.28. If we add a third clause $\texttt{let}(e, x.e')$ to the untyped λ-calculus, we extend the definition of $e[x \leftrightarrow y]$ with one additional clause:

$$\texttt{let}(e, z.e')[x \leftrightarrow y] = \texttt{let}(e[x \leftrightarrow y], z[x \leftrightarrow y].e[x \leftrightarrow y])$$

the definition of $e =_\alpha e'$ with one additional clause:

- $\texttt{let}(e_1, x.e_1') =_\alpha \texttt{let}(e_2, y.e_2')$ if $e_1 =_\alpha e_2$ and for some $z \in \mathbb{V}$ that appears nowhere in x, e_1', y, e_2', we have $e_1'[x \leftrightarrow z] =_\alpha e_2'[y \leftrightarrow z]$,

the definition of $\mathsf{fv}(e)$ with one additional clause:

$$\mathsf{fv}(\texttt{let}(e, x.e')) = \mathsf{fv}(e) \cup (\mathsf{fv}(e') \setminus \{x\})$$

and the definition of $e[e''/x]$ with one additional clause:

$$\texttt{let}(e, y.e')[e''/x] = \texttt{let}(e[e''/x], y.e'[e''/x]) \quad \text{if } y \neq x \text{ and } y \notin \mathsf{fv}(e'')$$

The benefit of the ABT notation is that it is highly systematic and can be mechanically translated into inference rules; the drawback is that it can lead to very unintuitive notations like $\texttt{let}(e, x.e')$ for $\texttt{let}\ x = e\ \texttt{in}\ e'$. For this reason, Harper [Har16] introduces *syntax charts*, which for each term give not only a formal ABT notation (and thus a binding structure) but also an informal notation and a brief English description.

Terms	$e ::=$	$\texttt{lambda}(x.e)$	$\lambda x.e$	λ-abstraction
		$\texttt{app}(e, e')$	$e\ e'$	application
		$\texttt{let}(e, x.e')$	$\texttt{let}\ x = e\ \texttt{in}\ e'$	`let`-binding
		$\texttt{letrec}(x.e, x.e')$	$\texttt{letrec}\ x = e\ \texttt{in}\ e'$	`letrec`-binding
		$\texttt{if}(e_1, e_2, e_3)$	$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	`if-then-else`
		$\vdots$	$\vdots$	$\vdots$

6 α-rule induction

Add more words in this section.

Hopefully Example 4.28 convinced you that extending α-equivalence and substitution to arbitrary programming languages with binding is both mechanically straightforward and quite tedious. An additional wrinkle that we haven't really dwelled on yet is that every time we perform a structural recursion or rule induction over a language with binding, we need to ensure that the definition/property in question respects α-equivalence.

Although these problems have been "solved" for decades, researchers continue to investigate alternative approaches with various benefits and drawbacks, especially in light of the increasing prevalence of mechanized proofs in programming language metatheory. (Indeed, there are distinguished papers at POPL 2024 [Pop24] and POPL 2025 [vBrü+25] about binding!) Some approaches include:

- using ABTs (and formalizing them: `jsiek/abstract-binding-trees`)
- nominal sets [Pit13]
- second-order [Fio08] and higher-order [HHP93] abstract syntax
- de Bruijn indices

Sadly we will not have time to get into these approaches in this class; instead, my emphasis will be on making sure you understand how to write correct but informal paper proofs by induction about languages with binding. We will see examples of this in the next lecture (and every subsequent lecture).

Nevertheless, I will close this lecture by *precisely* stating two α-equivalence-respecting rule induction principles for the $\Gamma \vdash e\ \mathsf{tm}$ judgment of the untyped λ-calculus [CST21]. As before, $\mathbb{V}$ is a countably infinite set of variables.

Theorem 4.29 (α-rule induction for terms I)**.** *To prove that a property $P(e)$ holds for all terms e, it suffices to show that:*

- *P respects α-equivalence: that is, if $P(e)$ and $e =_\alpha e'$ then $P(e')$,*

- *$P(x)$ for every $x \in \mathbb{V}$,*
- *for all e and e', if $P(e)$ and $P(e')$ then $P(\texttt{app}(e, e'))$, and*
- *there exists a finite set $V \subset \mathbb{V}$ such that for all $x \in \mathbb{V} \setminus V$ and all e, if $P(e)$ then $P(\texttt{lambda}(x.e))$.*

Definition 4.30. A *finite permutation of variables* $\pi : \mathbb{V} \leftrightarrow \mathbb{V}$ is a permutation $\pi : \mathbb{V} \to \mathbb{V}$ for which there exists a finite set $V \subset \mathbb{V}$ such that $\pi(x) = x$ for all $x \notin V$. We write $e[\pi]$ for the operation that permutes *all* variables in e according to π (directly generalizing Definition 4.12).

Theorem 4.31 (α-rule induction for terms II)**.** *To prove that a property $P(e)$ holds for all terms e, it suffices to show that:*

- *$P(x)$ for every $x \in \mathbb{V}$,*
- *for all e and e', if $P(e)$ and $P(e')$ then $P(\texttt{app}(e, e'))$, and*
- *for all x and e, if for all finite permutations $\pi : \mathbb{V} \leftrightarrow \mathbb{V}$ we have $P(e[\pi])$, then $P(\texttt{lambda}(x.e))$.*

talk about α-structural recursion?

References

[CST21] Ernesto Copello, Nora Szasz, and Álvaro Tasistro. "Formalization of metatheory of the Lambda Calculus in constructive type theory using the Barendregt variable convention". In: *Mathematical Structures in Computer Science* 31.3 (2021), pp. 341–360. DOI: 10.1017/S0960129521000335.

[Fio08] Marcelo Fiore. "Second-Order and Dependently-Sorted Abstract Syntax". In: *23rd Annual IEEE Symposium on Logic in Computer Science (LICS 2008)*. 2008, pp. 57–68. DOI: 10.1109/LICS.2008.38.

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

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. "A framework for defining logics". In: *J. ACM* 40.1 (Jan. 1993), pp. 143–184. ISSN: 0004-5411. DOI: 10.1145/138027.138060.

[Pit06] Andrew M. Pitts. “Alpha-structural recursion and induction”. In: *Journal of the ACM* 53.3 (May 2006), pp. 459–506. ISSN: 0004-5411. DOI: 10.1145/1147954.1147961.

[Pit13] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Vol. 57. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2013. ISBN: 9781107017788. DOI: 10.1017/CBO9781139084673.

[Pop24] Andrei Popescu. “Nominal Recursors as Epi-Recursors”. In: *Proceedings of the ACM on Programming Languages* 8.POPL (Jan. 2024). DOI: 10.1145/3632857.

[vBrü+25] Jan van Brügge, James McKinna, Andrei Popescu, and Dmitriy Traytel. “Barendregt Convenes with Knaster and Tarski: Strong Rule Induction for Syntax with Bindings”. In: *Proceedings of the ACM on Programming Languages* 9.POPL (Jan. 2025). DOI: 10.1145/3704893.