Foundations of Programming Languages

Paul Downen

June 3–8, 2024

Contents

Chapter 1

Operational Semantics

It's turtles all the way down...

The foundation of programming languages is fundamentally built on just one idea: induction. Everything we do is induction. We define grammars and the representation of valid syntax trees by induction. We specify the behavior of what a program is supposed to do by induction. We define operations on programs and their complex relationships by induction. We prove properties about individual programs and entire languages by induction. Compilers can be organized as a series of steps translating between lower and lower levels of code, each translation being defined by induction on its input. Our crown jewel – mechanized proof assistants — are effectively big, fancy induction engines.

If you *really* understand this one idea, and apply it to its fullest potential, you can go far in the field of programming languages.

1.1 Syntax

A grammar for the abstract syntax of λ-calculus with booleans, written in BNF:

$$\begin{aligned} \mathit{Variable} \ni x, y, z &::= \mathit{foo} \mid \mathit{bar} \mid \mathit{baz} \mid \ldots \\ \mathit{Constant} \ni \quad c &::= \texttt{true} \mid \texttt{false} \\ \mathit{Term} \ni M, N &::= c \mid x \mid M\ N \mid \lambda x.M \mid \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \end{aligned}$$

A BNF grammar definition is always signaled by ::=. The left-hand side of the ::= defines the name of the defined set(s) and/or one or more standard *meta-variables* which stand for elements of that set. The right-hand side gives a list of *all* the possible ways to form trees separated by a | divider.

The above BNF grammar is the same as the following, more verbose, definition of the set named *Term*.

Definition 1.1 (λ Terms)**.** Suppose that $\mathit{Constant} = \{\texttt{true}, \texttt{false}\}$ is the set of known constants and *Variable* is a set of all the (infinitely many) possible

identifier names. The set *Term* is *inductively defined* as the *smallest set* which is closed under the following formations of *distinct elements*:

- $c \in \mathit{Term}$ for any $c \in \mathit{Constant}$,
- $x \in \mathit{Term}$ for any $x \in \mathit{Variable}$,
- $M\ N \in \mathit{Term}$ for any $M, N \in \mathit{Term}$, and
- $\lambda x.M \in \mathit{Term}$ for any $x \in \mathit{Variable}$ and $M \in \mathit{Term}$.
- $\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \in \mathit{Term}$ for any $M, N_1, N_2 \in \mathit{Term}$,

In the following, we will implicitly let c stand for the elements of *Constant*, x, y, z stand for elements of *Variable*, and let M, N stand for elements of *Term*.

The inductive definition of a set is based on three facts that work in concert to describe all the (infinitely many) possible syntax trees:

1. The set is defined in terms of itself (i.e. the formation rules build on top of existing elements we already know must be the set).
2. The final set is the smallest possible one that contains all the elements defined by the formation rules.
3. Each formation rule creates an element that is always distinct from elements made by any other formation rule.

Property (1) is a key aspect to the expressive power of self-reference in an inductive definition to recursively identify an infinite number of things. The other two properties give a "Goldilocks" bound on the set that is *just right* — not too big and not small. Property (2) says that the rules provided give an *exhaustive* description of every element, which eliminates the possibility of exotic elements that come from something else that wasn't mentioned explicitly in the definition. Property (3) says that there is no *redundancy* in the rules, so that no two rules can make the same element. The combination of properties (2) and (3) means that we can reverse the formation rules, and use them to *match* on the structure of an *arbitrary* element in the set.

In contrast, here is some syntactic sugar:

$$(\texttt{let}\ x = M\ \texttt{in}\ N) = (\lambda x.N)\ M$$

Because $\texttt{let}\ x = M\ \texttt{in}\ N$ is not its own formation rule — it is defined in terms of the rules for $M\ N$ and $\lambda x.M$ — the two sides are *the same* tree.

1.2 Static Scope

Static variables should obey the following two (for now informal) laws:

1. The names of local variables don't matter (they can be renamed without affecting the meaning of an term).

2. The free variables of a term remain are still free after substitution (they cannot be *captured* by local bindings).

The only form of syntax that binds a variable is $\lambda x.M$, which defines a function that takes one parameter (named x) and returns the result calculated by M (which is allowed to refer to that x during calculation).

The *bound variables* of a term are any variable which is introduced by a λ inside that term.

$$\begin{aligned} BV &: \mathit{Term} \to \wp(\mathit{Variable}) \\ BV(c) &= \{\} \\ BV(x) &= \{\} \\ BV(M\ N) &= BV(M) \cup BV(N) \\ BV(\lambda x.M) &= BV(M) \cup \{x\} \\ BV(\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2) &= BV(M) \cup BV(N_1) \cup BV(N_2) \end{aligned}$$

The *free variables* of a term are any variable which that the term makes reference to *without* being inside of a λ that binds variables of that name.[1]

$$\begin{aligned} FV &: \mathit{Term} \to \wp(\mathit{Variable}) \\ FV(c) &= \{\} \\ FV(x) &= \{x\} \\ FV(M\ N) &= FV(M) \cup FV(N) \\ FV(\lambda x.M) &= FV(M) \setminus \{x\} \\ FV(\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2) &= FV(M) \cup FV(N_1) \cup FV(N_2) \end{aligned}$$

Example 1.1. In $\lambda x.x\ y$, the use of x refers to the locally-bound parameter of the λ, whereas the use of y refers to something else that must come from the larger context of the term. Thus, $FV(\lambda x.x\ y) = \{y\}$ — even though the term refers to x, that reference stays internal to the term — and $BV(\lambda x.x\ y)$.

Note that there can be some tricky cases. A variable might be both bound and free in the same term, such as $x\ \lambda x.x$ for which $FV(x\ \lambda x.x) = \{x\}$ and $BV(x\ \lambda x.x) = \{x\}$. A variable might also be bound but never used, such as $\lambda y.x$ who has one bound variable $BV(\lambda y.x) = \{y\}$ that is never referenced.

[1]The set operation $X \setminus Y$ means to take the set consisting of everything in X that *does not* appear in Y, i.e. the subtraction of Y's elements from X's elements. Sometimes this operation is written as $X - Y$.

1.3 Substitution

The capture-avoiding substitution operation $M[N/x]$ means to replace every *free* occurrence of x appearing inside M with the term N.

$$
\begin{aligned}
c\,[N/x] &= c \\
x\,[N/x] &= N \\
y\,[N/x] &= y && (x \neq y) \\
(M_1\ M_2)\,[N/x] &= (M_1\,[N/x])\ (M_2\,[N/x]) \\
(\lambda x.M)\,[N/x] &= \lambda x.M \\
(\lambda y.M)\,[N/x] &= \lambda y.(M\,[N/x]) && (x \neq y) \text{ and } y \notin FV(N) \\
\begin{pmatrix} \texttt{if}\ M_1 \\ \quad \texttt{then}\ M_2 \\ \quad \texttt{else}\ M_3 \end{pmatrix}[N/x] &= \begin{array}{l} \texttt{if}\ M_1\,[N/x] \\ \quad \texttt{then}\ M_2\,[N/x] \\ \quad \texttt{else}\ M_3\,[N/x] \end{array}
\end{aligned}
$$

Note that substitution is a *partial* function, because it might not be defined when substituting into a λ: if the replacement term M for x happens to contain a free variable y, then N cannot be substituted into a λ-abstraction that binds y because that would *capture* the free y found in N.

Example 1.2. $(x\ (\lambda x.(x\ y)))\,[y/x] = y\ (\lambda x.(x\ y))$ but $(x\ (\lambda x.(x\ y)))\,[x/y]$ is undefined.

The partiality of capture-avoiding substitution is expressed in the above equations by the following implicit convention: the particular case is only defined when each recursive call is also defined. This implicit convention can be made more explicit by the use of inference rules as an alternative definition of substitution. Focusing on the λ-calculus portion, we have:

$$
\frac{}{x\,[N/x] = N} \qquad \frac{y \neq x}{y\,[N/x] = y} \qquad \frac{M_1\,[N/x] = M_1' \quad M_2\,[N/x] = M_2'}{(M_1\ M_2)\,[N/x] = M_1'\ M_2'}
$$

$$
\frac{}{(\lambda x.M)\,[N/x] = \lambda x.M} \qquad \frac{x \neq y \quad y \notin FV(N) \quad M\,[N/x] = M'}{(\lambda y.M)\,[N/x] = \lambda y.M'}
$$

Both styles definitions should be seen as two different ways of expressing exactly the same operation.

Exercise 1.1. Finish the above set of inference rules defining substitution for booleans (substitution into constants as well as `if`-expressions).

Lemma 1.1. *For all terms M and N and variables x, if $BV(M) \cap FV(N) = \{\}$ then $M[N/x]$ is defined.*

Proof. By induction on the syntax of the term M.

- y: By definition $y[N/x]$ is always defined by one of two possible sub-cases, depending on the comparison between x and y:

  - If $x = y$, then $y\,[N/x] = x\,[N/x] = N$.
  - If $x \neq y$, then $y\,[N/x] = y$.

- $\lambda y.M$: The inductive hypothesis we get for M is

  Inductive Hypothesis. *If* $BV(M) \cap FV(N) = \{\}$ *then* $M[N/x]$ *is defined.*

  Note that $BV(\lambda y.M) = \{y\} \cup BV(M)$ by definition, so

  $$\begin{aligned} &BV(\lambda y.M) \cap FV(N) \\ &= (\{y\} \cup BV(M)) \cap FV(N) \\ &= (\{y\} \cap FV(N)) \cup (BV(M) \cap FV(N)) \end{aligned}$$

  It follows that $BV(\lambda y.M) \cap FV(N) = \{\}$ exactly when $\{y\} \cap FV(N) = \{\}$ (i.e. $y \notin FV(N)$)*and* $BV(M) \cap FV(N) = \{\}$ for both $i = 1$ and $i = 2$. Since we assumed $BV(\lambda y.M) \cap FV(N) = \{\}$, it must be that $y \notin FV(N)$ and $BV(M) \cap FV(N) = \{\}$.

  There are now two cases to consider, depending on whether or not x and y are equal.

  - $y = x$: The substitution $(\lambda y.M)\,[N/x] = (\lambda x.M)\,[N/x] = \lambda x.M$ is defined.
  - $y \neq x$: The substitution $(\lambda y.M)\,[N/x] = \lambda y.(M\,[N/x])$ is defined only when both $M\,[N/x]$ is defined *and* when the side condition $y \notin FV(N)$ is met. We already derived the fact that $y \notin FV(N)$ from the assumption $BV(\lambda y.M) \cap FV(N) = \{\}$ above, and applying the inductive hypothesis to the other derived fact that $BV(M) \cap FV(N) = \{\}$ ensures that $M\,[N/x]$ is defined. Thus, $(\lambda y.M)\,[N/x] = \lambda y.(M\,[N/x])$ is also defined.

- $M_1\ M_2$: The *two* inductive hypotheses we get for the sub-terms M_1 and M_2 are

  Inductive Hypothesis.

  a) If $BV(M_1) \cap FV(N) = \{\}$ *then* $M_1\,[N/x]$ *is defined.*

  b) If $BV(M_2) \cap FV(N) = \{\}$ *then* $M_2\,[N/x]$ *is defined.*

  Now, note that $BV(M_1\ M_2) = BV(M_1) \cup BV(M_2)$ by definition, so

  $$\begin{aligned} BV(M_1\ M_2) \cap FV(N) &= (BV(M_1) \cup BV(M_2)) \cap FV(N) \\ &= (BV(M_1) \cap FV(N)) \cup (BV(M_2) \cap FV(N)) \end{aligned}$$

  So the assumption that $BV(M_1\ M_2) \cap FV(N) = \{\}$ implies that $BV(M_i) \cap FV(N) = \{\}$ for both $i = 1$ and $i = 2$.

The substitution

$$(M_1\ M_2)[N/x] = M_1[N/x]\ M_2[N/x])$$

is defined exactly when $M_i[N/x]$ is defined for both $i = 1$ and $i = 2$. By applying the inductive hypothesis to the fact $BV(M_i) \cap FV() = \{\}$ derived above, we learn that $(M_1\ M_2)[N/x]$ is defined.

- The remaining cases for constants and $\texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3$ are left as an exercise to the reader. ■

Lemma 1.2. *For all terms M and N and all variables x, if $x \notin FV(M)$ then $M[N/x] = M$ when $M[N/x]$ is defined.*

Proof. By induction on the syntax of M. The proof is left as an exercise to the reader. ■

1.4 Renaming: α equivalence

The *renaming* operation—replacing all occurrences of a free variable with another variable—can be derived from capture-avoiding substitution. That is, the renaming operation $M[y/x]$ is just a special case of the more general substitution operation $M[N/x]$ since the variable y is an instance of a term.

In general, the particular choice of variable (i.e. name) of a bound variable should not matter: two terms where the bound variables have been renamed should be the same. This idea is captured for the only binder in our little language (λ) with the following α *equivalence* law

$$(\alpha\rightarrow) \qquad \lambda x.M =_\alpha \lambda y.(M\,[y/x]) \qquad (\text{if } y \notin FV(M))$$

The caveat of this rule makes sure that you don't accidentally capture a free variable of M, causing a formerly free variable to be bound by this λ.

Example 1.3. The function $\lambda x.x\ y$ takes one parameter (x) and applies it to y. The reference to y in the term is free (it is not bound locally in this term).

However, trying to α-rename this function to $\lambda y.y\ y$ is *wrong*, because this is a totally different function: it takes one parameter (now named y) and applies it to *itself*. The reason why $(\lambda x.x\ y) \neq_\alpha (\lambda y.y\ y)$ is because the choice of y for the parameter clashes with the existing free variables of the function's body ($y \in FV(x\ y)$).

If two terms M and N can be related by any number of applications of this α equivalence rule to any sub-term, then those terms are considered α-equivalent, which is written as $M =_\alpha N$. This can be formalized with inference rules. The main rule is

$$\frac{M\,[z/x] =_\alpha N\,[z/y] \quad z \notin FV(M) \cup FV(N)}{\lambda x.M =_\alpha \lambda y.N}\ \alpha\rightarrow$$

And the other rules just apply α-equivalence within sub-terms

$$\frac{}{x =_\alpha x} \qquad \frac{}{c =_\alpha c} \qquad \frac{M =_\alpha M' \quad N =_\alpha N'}{M\ N =_\alpha M'\ N'}$$

$$\frac{M =_\alpha M' \quad N_1 =_\alpha N_1' \quad N_2 =_\alpha N_2'}{\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 =_\alpha \texttt{if}\ M'\ \texttt{then}\ N_1'\ \texttt{else}\ N_2'}$$

The importance of α equivalence is not just so that we can ignore the superfluous choice of bound variable names. It means that capture-avoiding substitution — which is technically a partial operation from a surface-level reading — can always be done without restrictions so long as some convenient renaming is done first.

Lemma 1.3. *For any term M and variables x and y, $BV(M\,[y/x]) = BV(M)$ if $M\,[y/x]$ is defined.*

Proof. By induction on the syntax of the term M. The proof is left as an exercise to the reader. ■

Lemma 1.4. *For any term M and set of variables X, there is an α-equivalent term M' such that $M =_\alpha M'$ and $X \cap BV(M') = \{\}$.*

Proof. By induction on the syntax of M. The proof is left as an exercise to the reader. ■

Theorem 1.5. *For any terms M and N and any variable x, there is an α-equivalent M' such that $M =_\alpha M'$ and $M'[N/x]$ is defined.*

Proof. We can find such an α-equivalent M' by renaming M via Lemma 1.4 so that $BV(M') \cap FV(N) = \{\}$, which implies that $M'\,[N/x]$ is defined by Lemma 1.1. □

Theorem 1.6. *For any terms M, M', and N and any variable x, if $M =_\alpha M'$ then $M\,[N/x] =_\alpha M'\,[N/x]$ whenever both $M\,[N/x]$ and $M'\,[N/x]$ are defined.*

Proof. By induction on the derivation of derivation $M =_\alpha M'$. The proof is left as an exercise to the reader. ■

Because substitution is well-defined only up to α-equivalence, from now on, we will never distinguish between two α-equivalent terms. In other words, we will implicitly consider α-equivalent syntax trees as the same terms as needed.

When you need to be persnikety about the choice of local variable names (for example, when implementing a programming language or mechanizing its properties in a proof assistant), then it can help to keep λ-terms in a special form that avoids issues of variable shadowing and free versus bound variable confusion.

Theorem 1.7. *Every term M is α-equivalent to another term M' with the properties that:*

a) the free variables and bound variables of M' are distinct from one another ($FV(M') \cap BV(M') = \{\}$), and

b) no two bound variables can have the same name in M' (i.e. a variable name can be introduced by only one binder in M').

This is called the Barendregt [1985] convention.

Proof. By induction on the syntax of M. Left as an exercise to the reader. ■

Working with λ-terms following Barendregt's convention can have some benefits. For example, substituting a closed term for a variable inside such a term can be correctly implemented as a naïve search-and-replace without worrying about any side conditions, renaming, or checking the names introduced by λs. (Why?) The same also works if both the substitutee and substituter are sub-terms of some larger term: given $(\lambda x.M)\ N$ follows Barendregt's convention, then $M\,[N/x]$ is always defined and is the same as merely replacing *all* occurrences of x in M with N without checking side-conditions. (Why?)

1.5 Small-Step Operational Semantics: β reduction

The small-step operational semantics is defined in terms of a reduction relation written $M \mapsto M'$, and pronounced as "M steps to M'."

The basic steps for reducing a term:[2]

$$(\beta\!\to) \qquad (\lambda x.M)\ N \mapsto_\beta M\,[N/x]$$

$$(\beta\texttt{bool}_1) \qquad \texttt{if true then}\ N_1\ \texttt{else}\ N_2 \mapsto_\beta N_1$$

$$(\beta\texttt{bool}_2) \qquad \texttt{if false then}\ N_1\ \texttt{else}\ N_2 \mapsto_\beta N_2$$

These are *axioms*; they apply exactly as-is to a term. But they are not enough to reduce most terms down to an answer (a boolean literal or λ)!

Example 1.4. Using only the rules above,

$$\mathit{not} = \lambda x.\,\texttt{if}\ x\ \texttt{then false else true}$$

$$\texttt{if}\ \mathit{not}\ \texttt{false then}\ \mathit{yay}\ \texttt{else}\ \mathit{boo} \not\mapsto_\beta$$

You cannot yet reduce the outer `if`-expression because *not* `false` is not one of the two canonical boolean cases (`true` or `false`). Instead, we must reduce the call *not* `false` *first*, and only *after* can the `if` decide what to return based on the eventual result `true`.

[2] Note that, due to the use of substitution, renaming to an α-equivalent term may be needed to apply $\beta\!\to$ in certain cases.

Hence, we often need to reduce sub-terms. Which sub-term to reduce in these cases is formalized by the following inference rules for applying reductions inside certain contexts:

$$\frac{M \mapsto_\beta M'}{\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \mapsto_\beta \texttt{if}\ M'\ \texttt{then}\ N_1\ \texttt{else}\ N_2} \qquad \frac{M \mapsto_\beta M'}{M\ N \mapsto_\beta M'\ N}$$

Combined with the above axioms, these rules are now enough to reduce terms.

Example 1.5. Using the extra inference rules above, we can now take a step in

$$\frac{\mathit{not}\ \texttt{false} \mapsto_\beta \texttt{if false then false else true}}{\begin{array}{lll}\texttt{if}\ \mathit{not}\ \texttt{false} & & \texttt{if}\ (\texttt{if false then false else true}) \\ \texttt{then}\ \mathit{yay} & \mapsto_\beta & \texttt{then}\ \mathit{yay} \\ \texttt{else}\ \mathit{boo} & & \texttt{else}\ \mathit{boo}\end{array}}$$

Theorem 1.8 (Determinism)**.** *If $M \mapsto_\beta M_1$ and $M \mapsto_\beta M_2$ then $M_1 =_\alpha M_2$.*

As the name suggests, each step only does a little bit of work. The vast majority of terms will take many steps to fully reduce to their final answer.

The multi-step operational semantics chains together multiple (zero or more) small reduction steps, and is defined as the smallest binary relation $M \mapsto\!\!\!\!\mapsto M'$ between terms closed under the following:

- *Inclusion*: $M \mapsto\!\!\!\!\mapsto M'$ if $M \mapsto M'$,
- *Reflexivity*: $M \mapsto\!\!\!\!\mapsto M$, and
- *Transitivity*: $M \mapsto\!\!\!\!\mapsto M''$ if $M \mapsto\!\!\!\!\mapsto M'$ and $M' \mapsto\!\!\!\!\mapsto M''$ for some M'.

Rephrased in terms of inference rules, these closure properties of multi-step reduction are:

$$\frac{M \mapsto M'}{M \mapsto\!\!\!\!\mapsto M'}\ \mathit{Incl.} \qquad \frac{}{M \mapsto\!\!\!\!\mapsto M}\ \mathit{Refl.} \qquad \frac{M \mapsto\!\!\!\!\mapsto M' \quad M' \mapsto\!\!\!\!\mapsto M''}{M \mapsto\!\!\!\!\mapsto M''}\ \mathit{Trans.}$$

1.6 Evaluation Contexts

But these inference rules are awfully repetitive and writing the whole derivation tree for a single step quickly becomes unwieldy. And their cumbersome nature is compounded as we extend the language with more features.

A more concise presentation of exactly the same thing is to define a grammar of *evaluation contexts* (a subset of all contexts around a term) like so:

$$\mathit{EvalCxt} \ni E ::= \square \mid E\ N \mid \texttt{if}\ E\ \texttt{then}\ N_1\ \texttt{else}\ N_2$$

Now, *all* of the above inference rules for evaluating certain sub-terms are expressed by the *one* inference rule:

$$\frac{M \mapsto M'}{E[M] \mapsto E[M']}$$

Where $E[M]$ is the notation for plugging M in for the $\Box$ inside the evaluation context E, defined as:

$$\begin{aligned}\Box[M] &= M\\ (E\ N)[M] &= E[M]\ N\\ (\texttt{if}\ E\ \texttt{then}\ N_1\ \texttt{else}\ N_2)[M] &= \texttt{if}\ E[M]\ \texttt{then}\ N_1\ \texttt{else}\ N_2\end{aligned}$$

Note that, unlike with substitution, there is no issues involving capture when plugging a term into a context.[3]

Example 1.6. The term $\texttt{if}\ \mathit{not}\ \texttt{false}\ \texttt{then}\ \mathit{yay}\ \texttt{else}\ \mathit{boo}$ can be fully reduced using evaluation contexts like so:

$$\begin{aligned}&\texttt{if}\ \boxed{\mathit{not}\ \texttt{false}}\ \texttt{then}\ \mathit{yay}\ \texttt{else}\ \mathit{boo}\\ \mapsto\ &\texttt{if}\ \boxed{\texttt{if false then false else true}}\ \texttt{then}\ \mathit{yay}\ \texttt{else}\ \mathit{boo}\\ \mapsto\ &\boxed{\texttt{if true then}\ \mathit{yay}\ \texttt{else}\ \mathit{boo}}\\ \mapsto\ &\mathit{yay}\end{aligned}$$

Going the other way, we can *always* describe every reduction step as a decomposition into a *single* evaluation context surrounding a *reducible expression* (called a "redex" for short).

$$\mathit{Redex} \ni R ::= (\lambda x.M)\ N \mid \texttt{if}\ c\ \texttt{then}\ N_1\ \texttt{else}\ N_2$$

Lemma 1.9. *If $M \mapsto_\beta M'$, then there is an evaluation context E, a redex R, and a term N such that*

a) $M = E[R]$,

b) $R \mapsto_\beta N$, *and*

c) $M' = E[N]$

Proof. By induction on the derivation of $M \mapsto_\beta M'$. The proof is left as an exercise to the reader. ■

In general, is usually many ways to decompose a term M into an evaluation context surrounding some sub-term $E[N]$. For example, decomposing M into $\Box[M]$ is always valid. However, *at most one* of these decompositions will identify a redex as the sub-term inside the evaluation context, giving a unique way to point out the next step of reduction.

Lemma 1.10 (Unique Decomposition)**.** *If $M = E_1[R_1]$ and $M = E_2[R_2]$, then $E_1 = E_2$ and $R_1 = R_2$.*

[3] In fact, the convention is that plugging a term into a more general kind of context (which might place the hole $\Box$ under a binder like $\lambda x.\Box$) will *intentionally* capture free variables of the expression that's replacing $\Box$.

Proof. By induction on the syntax of M. The proof is left as an exercise to the reader. ■

Theorem 1.11 (Determinism)**.** *If $M \mapsto_\beta M_1$ and $M \mapsto_\beta M_2$ then $M_1 =_\alpha M_2$.*

Proof. Applying Lemma 1.9 to the reduction steps $M \mapsto_\beta M_1$ and $M \mapsto_\beta M_2$, we have the decomposition.

$$\frac{R_1 \mapsto_\beta N_1}{M = E_1[R_1] \mapsto_\beta E_1[N_1] = M_1} \qquad \frac{R_2 \mapsto_\beta N_2}{M = E_2[R_2] \mapsto_\beta E_2[N_2] = M_2}$$

And since the decomposition into an evaluation context and redex is unique (Lemma 1.10), it must be that

$$M_1 = E_1[R_1] = E_2[R_2] = M_2 \qquad \square$$

1.7 Call-by-Value & Call-By-Name

The operational semantics we have seen so far specified *call-by-name* evaluation, where the unevaluated code describing the argument to a function call is passed directly to the function.

Example 1.7.

$$\begin{aligned} \mathit{not} &= \lambda x.\, \texttt{if}\ x\ \texttt{then false else true} \\ \mathit{and} &= \lambda x.\lambda y.\, \texttt{if}\ x\ \texttt{then}\ y\ \texttt{else false} \end{aligned}$$

$$\begin{aligned} &\mathit{and}\ (\mathit{not}\,\texttt{true})\ (\mathit{not}(\mathit{not}\,\texttt{true})) \\ &\mapsto (\lambda y.\, \texttt{if}\ \mathit{not}\,\texttt{true}\,\texttt{then}\ y\,\texttt{else false})\ (\mathit{not}(\mathit{not}\,\texttt{true})) \\ &\mapsto \texttt{if}\ \mathit{not}\,\texttt{true}\,\texttt{then}\,\mathit{not}(\mathit{not}\,\texttt{true})\,\texttt{else false} \\ &\mapsto \texttt{if}\,(\texttt{if true then false else true})\,\texttt{then}\,\mathit{not}(\mathit{not}\,\texttt{true})\,\texttt{else false} \\ &\mapsto \texttt{if false then}\,\mathit{not}(\mathit{not}\,\texttt{true})\,\texttt{else false} \\ &\mapsto \texttt{false} \end{aligned}$$

Functions arguments are evaluated *only* when the body of the function uses a parameter (and re-evaluated every time). This is caused by the substitution of unevaluated terms during β-reduction. If the parameter is used more than one, then the term is duplicated on each use, $(\lambda x. \ldots x \ldots x \ldots)\ (f\ y) \mapsto_\beta \ldots f\ y \ldots f\ y \ldots$. In certain cases, this can cause unwanted duplication of the same steps.

Exercise 1.2. Use the call-by-name small-step operational semantics given in Sections 1.5 and 1.6 to reduce the following term to a boolean constant:

$$\begin{aligned} &\texttt{let}\ x = \mathit{not}\,\texttt{false}\,\texttt{in} \\ &\texttt{let}\ y = \mathit{and}\ x\ x\,\texttt{in} \\ &\mathit{and}\ y\ y \end{aligned}$$

This is very different from the vast majority of practical programming languages, which evaluate arguments first and pass their value during a function call. This commonly-used evaluation order is referred to as *call-by-value* evaluation.

We can give an alternate semantics to our λ-calculus with booleans that describes call-by-value evaluation by changing some of the definitions. First, we need to restrict the reduction rule for function calls so that only *values* can be substituted for the function's parameter:

$$\mathit{Value} \ni V, W ::= c \mid \lambda x.M \mid x$$

$$\begin{array}{rr}
(\beta{\to_V}) & (\lambda x.M)\ V \mapsto_\beta M\,[V/x] \\
(\beta\texttt{bool}_1) & \texttt{if true then}\ N_1\ \texttt{else}\ N_2 \mapsto_\beta N_1 \\
(\beta\texttt{bool}_2) & \texttt{if else then}\ N_1\ \texttt{else}\ N_2 \mapsto_\beta N_2
\end{array}$$

Now, we need to point the evaluator in the direction of the argument when encountering a function call like $(\lambda x.x)\ (f\ y)$. To do so, we must expand the evaluation contexts to work on *both* the function and argument side of an application. The following expansion of evaluation contexts is enough to evaluate all sensible terms, but still keeps determinism:

$$\mathit{EvalCxt} \ni E ::= \Box \mid E\ N \mid V\ E \mid \texttt{if}\ E\ \texttt{then}\ N_1\ \texttt{else}\ N_2$$

To keep determinism, we need to pick which side goes first. The above definition specifies that functions should be evaluated before arguments, since $M\ N$ can always be decomposed into the sub-term M surrounded by the evaluation context $\Box\ N$. Only when the function has been reduced to a value, $M \mapsto\!\!\!\!\to V$, we have $M\ N \mapsto\!\!\!\!\to V\ N$, which can now be decomposed into the sub-term N surrounded by the evaluation context $V\ \Box$.

Our notion of reducible expression has now also changed, since function calls can only be resolved with values as arguments. The new definition of call-by-value redexes is refined to:

$$\mathit{Redex} \ni R ::= (\lambda x.M)\ V \mid \texttt{if}\ c\ \texttt{then}\ N_1\ \texttt{else}\ N_2$$

Exercise 1.3. Use the call-by-value small-step operational semantics given above to reduce the same term as Exercise 1.2 to a boolean constant:

$$\begin{array}{l}
\texttt{let}\ x = \mathit{not}\ \texttt{false in} \\
\texttt{let}\ y = \mathit{and}\ x\ x\ \texttt{in} \\
\mathit{and}\ y\ y
\end{array}$$

Chapter 2

Type Systems & Safety

The syntax trees have some structure — especially up to α-equivalence — compared to raw strings. However, real programs still have more structure to them not expressed in the grammar of syntax.

Nonsense terms, like $\lambda x.(\texttt{true}\, x)$ or $\texttt{if}\, \lambda x.x\, \texttt{then true else false}$, are syntactically correct, but don't mean anything. When run, they will *go wrong* — crash with an error, cause unpredictable behavior, or otherwise get stuck.

Type systems are tools for predicting some properties of programs before they are run, so that we may fix them or throw them out ASAP. The type safety motto is: *well-typed programs don't go wrong.*

2.1 "Simple" Types

$$\begin{aligned} \textit{Type} \ni A, B &::= \texttt{bool} \mid A \to B \\ \textit{Environment} \ni \quad \Gamma &::= x_1 : A_1, \ldots, x_n : A_n \\ \textit{Judgement} \quad &::= \Gamma \vdash M : A \end{aligned}$$

Specific forms of types are booleans ($\texttt{bool}$) or functions ($A \to B$). Typing judgements are *hypothetical* — to check that a term M has a type A (written $M : A$), we need to assign an assumed type for each free variable that might appear in M. This assumed list of type assignments for free variables is stored in an environment Γ which is separated from the main conclusion by a "turn-style" ($\vdash$). We won't worry about the order of Γ (you can rearrange the variable assignments as you wish), but we will stipulate that Γ can only contain *at most one* type assignment for any particular variable. So $x : A, y : B$ is considered the same as $y : B, x : A$, and both are different from $x : A, y : A$. But the environment $x : A, x : B$ is illegal.

$$\frac{}{\Gamma, x : A \vdash x : A}\ \mathit{Var}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ {\to}I \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ {\to}E$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \texttt{bool}I_1 \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ \texttt{bool}I_2$$

$$\frac{\Gamma \vdash M : \texttt{bool} \quad \Gamma \vdash N_1 : A \quad \Gamma \vdash N_2 : A}{\Gamma \vdash \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 : A}\ \texttt{bool}E$$

2.2 Type Safety

Definition 2.1 (Stuck)**.** A term M is *stuck* if it is not a value (i.e. $M \notin \mathit{Value}$) and cannot take a step (i.e. there is no M' such that $M \mapsto_\beta M'$).

Definition 2.2 (Closed vs Open)**.** A term M is *closed* if it has no free variables (i.e. $FV(M) = \{\}$), and *open* otherwise.

The "big-step" type safety theorem.

Theorem 2.1 (Type Safety)**.** *No well-typed, closed term M ever gets stuck (i.e. for any $M \mapsto\!\!\!\!\to_\beta M'$, either M' is a value or $M' \mapsto_\beta$).*

Proof. Type safety can be proved in terms of two small-step properties by Wright and Felleisen [1994]:

- *Progress*: No well-typed, closed term $\bullet \vdash M : A$ is stuck.
- *Preservation*: The reduct ($M \mapsto_\beta M'$) of every well-typed term ($\Gamma \vdash M : A$) has the same type ($\Gamma \vdash M' : A$).

A multiple-step reduction $M \mapsto\!\!\!\!\to_\beta M'$ is a finite sequence of individual reduction steps $M \mapsto_\beta M_1 \mapsto_\beta M_2 \mapsto_\beta \ldots \mapsto_\beta M'$. Given that the starting point is well-typed as $\bullet \vdash M : A$, we can proceed by induction on the reduction sequence from left-to-right to apply preservation to each individual step to conclude $\bullet \vdash M' : A$ because each intermediate step is also well-typed ($\bullet \vdash M_i : A$ for each i). Then, progress applied to $\bullet \vdash M' : A$ ensures M' is not stuck. □

2.2.1 Progress

Lemma 2.2 (Canonical Forms)**.**

- *If $\bullet \vdash V : \texttt{bool}$ then $V = \texttt{true}$ or $V = \texttt{false}$.*
- *If $\bullet \vdash V : A \to B$ then $V = \lambda x.M$ for some $x : A \vdash M : B$.*

Proof. By induction on the possible derivations concluding $\bullet \vdash V : \texttt{bool}$ and $\bullet \vdash V : A \to B$. Left as an exercise to the reader. ■

Lemma 2.3 (Progress)**.** *If $\bullet \vdash M : A$ then M is not stuck: either M is a value ($M \in$ Value) or takes a step ($M \mapsto_\beta M'$ for some M').*

Proof. By induction on the given derivation $\mathcal{D}$ of $\bullet \vdash M : A$,

$$\begin{array}{c} \vdots\ \mathcal{D} \\ \bullet \vdash M : A \end{array}$$

- (*Var*) The bottom inference cannot possibly be the axiom for variables

$$\frac{}{\Gamma, x : A \vdash x : A}\ \mathit{Var}$$

since there is no Γ which makes $\Gamma, x : A = \bullet$.

- ($\mathtt{bool}I$) If the bottom inference is a boolean introduction rule

$$\frac{}{\bullet \vdash \mathtt{true} : \mathtt{bool}}\ \mathtt{bool}I_1 \qquad \frac{}{\bullet \vdash \mathtt{false} : \mathtt{bool}}\ \mathtt{bool}I_1$$

then the term is a value since $\mathtt{true}, \mathtt{false} \in$ *Value*.

- ($\mathtt{bool}E$) If the bottom inference is the boolean elimination rule

$$\frac{\begin{array}{c}\vdots\ \mathcal{D} \\ \bullet \vdash M : \mathtt{bool}\end{array} \quad \begin{array}{c}\\ \bullet \vdash N_1 : A\end{array}\ \mathcal{E}_1 \quad \begin{array}{c}\\ \bullet \vdash N_2 : A\end{array}\ \mathcal{E}_2}{\bullet \vdash \mathtt{if}\, M\, \mathtt{then}\, N_1\, \mathtt{else}\, N_2 : A}\ \mathtt{bool}E$$

then we have these three inductive hypotheses from sub-derivations $\mathcal{D}$, $\mathcal{E}_1$, and $\mathcal{E}_2$:

Inductive Hypothesis. *a) M is not stuck,*

b) N_1 is not stuck, and

c) N_2 is not stuck.

We can then proceed by the reason *why* M is not stuck:

– If $M \mapsto_\beta M'$ for some M', then

$$\mathtt{if}\, M\, \mathtt{then}\, N_1\, \mathtt{else}\, N_2 \mapsto_\beta \mathtt{if}\, M'\, \mathtt{then}\, N_1\, \mathtt{else}\, N_2$$

because $\mathtt{if}\, \square\, \mathtt{then}\, N_1\, \mathtt{else}\, N_2$ is an evaluation context.

– If $M \in$ *Value*, then it must be one of the two known canonical forms $M = \mathtt{true}$ or $M = \mathtt{false}$ of type $\mathtt{bool}$ (Lemma 2.2). In either case, one of the two possible $\beta\,\mathtt{bool}$ reductions apply:

$$\begin{array}{ll} \mathtt{if}\, M\, \mathtt{then}\, N_1\, \mathtt{else}\, N_2 \mapsto_{\beta\,\mathtt{bool}_1} N_1 & (\text{if } M = \mathtt{true}) \\ \mathtt{if}\, M\, \mathtt{then}\, N_1\, \mathtt{else}\, N_2 \mapsto_{\beta\,\mathtt{bool}_2} N_2 & (\text{if } M = \mathtt{false}) \end{array}$$

So no matter the reason why M is not stuck, $\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2$ always takes a step.

- ($\to I$) If the bottom inference is the function introduction rule

$$\frac{\begin{array}{c}\vdots\ \mathcal{D} \\ x : A \vdash M : B\end{array}}{\bullet \vdash \lambda x.M : A \to B}\ \to I$$

then we have no inductive hypothesis (because the premise $x : A \vdash M : B$ has a non-empty environment), but the term is always a value since $\lambda x.M \in \mathit{Value}$ for any sub-term M.

- ($\to E$) If the bottom inference is the function elimination rule

$$\frac{\begin{array}{c}\vdots\ \mathcal{D} \\ \bullet \vdash M : A \to B\end{array} \quad \begin{array}{c}\vdots\ \mathcal{E} \\ \bullet \vdash N : A\end{array}}{\bullet \vdash M\ N : B}\ \to E$$

then we have these two inductive hypotheses from sub-derivations $\mathcal{D}$ and $\mathcal{E}$:

Inductive Hypothesis. *a) M is not stuck, and*

b) N is not stuck.

We can then proceed by the reason *why* M is not stuck:

- If $M \mapsto_\beta M'$ for some M', then $M\ N \mapsto_\beta M'\ N$ because $\square\ N$ is an evaluation context.
- If $M \in \mathit{Value}$, then it must be a canonical form $M = \lambda x.M'$ of type $A \to B$ (Lemma 2.2). Thus, the $\beta \to$ reduction applies:

$$M\ N \mapsto_{\beta\to} M'\,[N/x] \qquad (\text{if } M = \lambda x.M')$$

So no matter the reason why M is not stuck, $M\ N$ always takes a step. □

Exercise 2.1. Rephrase and redo the proof of the progress lemma for the alternate call-by-value semantics in Section 1.7.

2.2.2 Preservation

Lemma 2.4 (Typed Substitution)**.** *If $\Gamma, x : A \vdash M : B$ and $\Gamma \vdash N : A$ then $\Gamma \vdash M\,[N/x] : B$.*

Proof. By induction on the given derivation of $\Gamma, x : A \vdash M : B$. Left as an exercise to the reader. ■

Lemma 2.5 (Preservation)**.** *If $\Gamma \vdash M : A$ and $M \mapsto_\beta M'$ then $\Gamma \vdash M' : A$.*

Proof. Applying Lemma 1.9 to $M \mapsto_\beta M'$, we have $M = E[R] \mapsto_\beta E[N] = M'$ because $R \mapsto_\beta N$. We can then proceed by induction on the typing derivation of $\Gamma \vdash E[R] : A$. For the base cases where $E = \Box$, we have a reduction step applied directly in the conclusion of the typing derivation:

- ($\beta\,\texttt{bool}$) where $R = \texttt{if}\,c\,\texttt{then}\,N_1\,\texttt{else}\,N_2$. The typing derivation of $\Gamma \vdash \texttt{if}\,c\,\texttt{then}\,N_1\,\texttt{else}\,N_2 : A$ must conclude with

$$\frac{\begin{array}{ccc}\vdots\,\mathcal{D} & \vdots\,\mathcal{E}_1 & \vdots\,\mathcal{E}_2 \\ \Gamma \vdash c : \texttt{bool} & \Gamma \vdash N_1 : A & \Gamma \vdash N_2 : A\end{array}}{\Gamma \vdash \texttt{if}\,c\,\texttt{then}\,N_1\,\texttt{else}\,N_2 : A}\ \texttt{bool}E$$

  where the derivation $\mathcal{D}$ of the boolean constant must be either

$$\mathcal{D} = \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \texttt{bool}I_1 \qquad (\text{where } c = \texttt{true})$$

  or

$$\mathcal{D} = \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ \texttt{bool}I_2 \qquad (\text{where } c = \texttt{false})$$

  There are two possible reduction steps depending the typing derivation $\mathcal{D}$ of $\Gamma \vdash c : \texttt{bool}$:

  - If $c = \texttt{true}$ then $\texttt{if}\,c\,\texttt{then}\,N_1\,\texttt{else}\,N_2 \mapsto_{\beta\,\texttt{bool}_1} N_1$, and $\mathcal{E}_1$ is the derivation proving $\Gamma \vdash N_1 : A$.
  - If $c = \texttt{false}$ then $\texttt{if}\,c\,\texttt{then}\,N_1\,\texttt{else}\,N_2 \mapsto_{\beta\,\texttt{bool}_2} N_2$, and $\mathcal{E}_2$ is the derivation proving $\Gamma \vdash N_2 : A$.

- ($\beta\ \to$) where $R = (\lambda x.M)\ N$. The derivation of $\Gamma \vdash M\ N : A$ must conclude with

$$\frac{\dfrac{\vdots\,\mathcal{D}}{\Gamma, x : B \vdash M : A}}{\Gamma \vdash \lambda x.M : B \to A}\ {\to}I \qquad \frac{\vdots\,\mathcal{E}}{\Gamma \vdash N : B}}{\Gamma \vdash (\lambda x.M)\ N : A}\ {\to}E$$

  The only possible reduction step is $(\lambda x.M)\ N \mapsto_{\beta\to} M\,[N/x]$. Typed substitution (Lemma 2.4) applied to $\mathcal{D}$ and $\mathcal{E}$ gives a new derivation proving $\Gamma \vdash M\,[N/x] : A$.

The remaining cases apply a reduction step inside a non-empty evaluation context:

- ($\texttt{bool}E$) Given a derivation concluding

$$\frac{\begin{array}{ccc}\vdots\,\mathcal{D} & \vdots\,\mathcal{E}_1 & \vdots\,\mathcal{E}_2 \\ \Gamma \vdash E[R] : \texttt{bool} & \Gamma \vdash N_1 : A & \Gamma \vdash N_2 : A\end{array}}{\Gamma \vdash \texttt{if}\,E[R]\,\texttt{then}\,N_1\,\texttt{else}\,N_2 : A}\ \texttt{bool}E$$

for the left-hand side of the step

$$\begin{array}{r}\texttt{if}\, E[R]\, \texttt{then}\, N_1 \\ \texttt{else}\, N_2\end{array} \mapsto_\beta \begin{array}{r}\texttt{if}\, E[M']\, \texttt{then}\, N_1 \\ \texttt{else}\, N_2\end{array} \qquad (\text{where } R \mapsto_\beta M')$$

Inductive Hypothesis. *There is a derivation* $\mathcal{D}'$ *of* $\Gamma \vdash E[M'] : \texttt{bool}$.

The right-hand side than has the same type by the derivation

$$\frac{\begin{array}{ccc}\vdots\, \mathcal{D}' & \vdots\, \mathcal{E}_1 & \vdots\, \mathcal{E}_2 \\ \Gamma \vdash E[M'] : \texttt{bool} & \Gamma \vdash N_1 : A & \Gamma \vdash N_2 : A\end{array}}{\Gamma \vdash \texttt{if}\, E[M']\, \texttt{then}\, N_1\, \texttt{else}\, N_2 : A}\ \texttt{bool}E$$

- $(\to E)$ Given a derivation concluding

$$\frac{\begin{array}{cc}\vdots\, \mathcal{D} & \vdots\, \mathcal{E} \\ \Gamma \vdash E[R] : B \to A & \Gamma \vdash N : B\end{array}}{\Gamma \vdash M\ N : A}\ \to E$$

for the left-hand side of the step

$$E[R]\ N \mapsto_\beta \texttt{if}\, E[M']\ N \qquad (\text{where } R \mapsto_\beta M')$$

Inductive Hypothesis. *There is a derivation* $\mathcal{D}'$ *of* $\Gamma \vdash E[M'] : B \to A$.

The right-hand side than has the same type by the derivation

$$\frac{\begin{array}{cc}\vdots\, \mathcal{D}' & \vdots\, \mathcal{E} \\ \Gamma \vdash E[M'] : B \to A & \Gamma \vdash N : A\end{array}}{\Gamma \vdash E[M']\ N : A}\ \to E$$

□

Corollary 2.6 (Preservation*)**.** *If* $\Gamma \vdash M : A$ *and* $M \mapsto\!\!\!\!\to_\beta M'$ *then* $\Gamma \vdash M' : A$.

Proof. Follows from Lemma 2.5 by induction on the multi-step reduction sequence. Left as an exercise to the reader. ■

2.3 Sums

Sum types are like booleans, but where the two different values are not just constants, but carry along some other data with them.

Extended syntax:

$$\mathit{Term} \ni M, N ::= \ldots \mid \texttt{inl}\, M \mid \texttt{inr}\, M \mid \texttt{case}\, M\, \texttt{of}\, \{\, \texttt{inl}\, x \Rightarrow N_1 \mid \texttt{inr}\, y \Rightarrow N_2 \,\}$$

Extended operational semantics:

$$\mathit{EvalCxt} \ni E ::= \ldots \mid \texttt{case}\, E\, \texttt{of}\, \{\, \texttt{inl}\, x \Rightarrow N_1 \mid \texttt{inr}\, y \Rightarrow N_2 \,\}$$

$$(\beta{+}_1) \qquad \texttt{case inl}\, M\, \texttt{of}\, \{\, \texttt{inl}\, x \Rightarrow N_1 \mid \texttt{inr}\, y \Rightarrow N_2 \,\} \mapsto_\beta N_1\,[M/x]$$

$$(\beta{+}_2) \qquad \texttt{case inr}\, M\, \texttt{of}\, \{\, \texttt{inl}\, x \Rightarrow N_1 \mid \texttt{inr}\, y \Rightarrow N_2 \,\} \mapsto_\beta N_2\,[M/y]$$

Exercise 2.2. Give an alternate call-by-value operational semantics for sum types.

Typing rules:

$$\mathit{Type} \ni A, B ::= \ldots \mid A + B$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathtt{inl}\, M : A + B}\ {+I_1} \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathtt{inr}\, M : A + B}\ {+I_2}$$

$$\frac{\Gamma \vdash M : A + B \quad \Gamma, x : A \vdash N_1 : B' \quad \Gamma, y : B \vdash N_2 : B'}{\Gamma \vdash \mathtt{case}\, M\, \mathtt{of}\, \{\, \mathtt{inl}\, x \Rightarrow N_1 \mid \mathtt{inr}\, y \Rightarrow N_2 \,\} : B'}\ {+E}$$

Exercise 2.3. Extend the proofs of progress and preservation for sum types, using a call-by-name and/or call-by-value operational semantics.

2.4 Products

Product types combine together two things into one.

Extended syntax:

$$\mathit{Term} \ni M, N ::= \ldots \mid (M, N) \mid \mathtt{fst}\, M \mid \mathtt{snd}\, M$$

Extended operational semantics:

$$\mathit{EvalCxt} \ni E ::= \ldots \mid \mathtt{fst}\, E \mid \mathtt{snd}\, E$$

$$(\beta{\times_1}) \qquad \mathtt{fst}(M, N) \mapsto_\beta M$$
$$(\beta{\times_2}) \qquad \mathtt{snd}(M, N) \mapsto_\beta N$$

Exercise 2.4. Give an alternate call-by-value operational semantics for product types.

Typing rules:

$$\mathit{Type} \ni A, B ::= \ldots \mid A \times B$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B}\ {+I}$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathtt{fst}\, M : A}\ {\times E_1} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathtt{snd}\, M : B}\ {\times E_1}$$

Exercise 2.5. Instead of `fst` and `snd` projections, products can be taken apart by pattern-matching similar to sum types:

$$\mathtt{case}\, M\, \mathtt{of}\, \{(x, y) \Rightarrow N\}$$

Specify the semantics of this case expression for products by extending the operational semantics and type system.

Exercise 2.6. Extend the proofs of progress and preservation for product types, using a call-by-name and/or call-by-value operational semantics.

2.5 Recursive Types

Many types are defined recursively in terms of themselves. For example, natural numbers, lists, and binary trees are defined in the ML and Haskell family of languages as:

$$\begin{aligned}\texttt{data nat} &= \texttt{zero} \mid \texttt{succ nat}\\ \texttt{data list}\,\alpha &= \texttt{nil} \mid \texttt{cons}\,\alpha\ (\texttt{list}\,\alpha)\\ \texttt{data tree}\,\alpha &= \texttt{leaf}\,\alpha \mid \texttt{branch}\ (\texttt{tree}\,\alpha)\ (\texttt{tree}\,\alpha)\end{aligned}$$

Because the definitions are recursive, we can't treat them as merely shorthand (they would expand forever!). To break the recursive loop, we can use recursive types $\mu\alpha.B$ which represent the repetitive infinite cycle with a finite description. First, encode the types to just one option using product and sum types (replace the alternative | bar with the sum $+$, and combine multiple arguments of a constructor with a product $\times$):

$$\begin{aligned}\texttt{nat} &= 1 + \texttt{nat}\\ \texttt{list}\,\alpha &= 1 + (\alpha \times \texttt{list}\,\alpha)\\ \texttt{tree}\,\alpha &= \alpha + (\texttt{tree}\,\alpha \times \texttt{tree}\,\alpha)\end{aligned}$$

Then, we can abstract over the recursive self-reference with the recursive μ type:

$$\begin{aligned}\texttt{nat} &= \mu\alpha.1 + \alpha\\ \texttt{list}\,\alpha &= \mu\beta.1 + (\alpha \times \beta)\\ \texttt{tree}\,\alpha &= \mu\beta.\alpha + (\beta \times \beta)\end{aligned}$$

Extended syntax:

$$\mathit{Term} \ni M, N ::= \ldots \mid \texttt{fold}\,M \mid \texttt{unfold}\,M$$

Typing rules:

$$\begin{aligned}\mathit{TypeVar} &\ni \alpha, \beta ::= \ldots\\ \mathit{Type} &\ni A, B ::= \ldots \mid \mu\alpha.B \mid \alpha\end{aligned}$$

$$\frac{\Gamma \vdash M : B\,[\mu\alpha.B/\alpha]}{\Gamma \vdash \texttt{fold}\,M : \mu\alpha.B}\ \mu I \qquad \frac{\Gamma \vdash M : \mu\alpha.B}{\Gamma \vdash \texttt{unfold}\,M : B\,[\mu\alpha.B/\alpha]}\ \mu E$$

Extended operational semantics:

$$\mathit{EvalCxt} \ni E ::= \ldots \mid \texttt{unfold}\,E$$

$$(\beta\mu) \qquad \texttt{unfold}(\texttt{fold}\,M) \mapsto_\beta M$$

Exercise 2.7. Give an alternate call-by-value operational semantics for recursive types.

Exercise 2.8. Extend the proofs of progress and preservation for recursive types, using a call-by-name and/or call-by-value operational semantics.

Chapter 3

Equational & Rewriting Theories

When do we know that two programs are interchangeable?

$$\texttt{true} \stackrel{?}{=} \texttt{false}$$
$$\mathit{not}\,\texttt{true} \stackrel{?}{=} \texttt{false}$$
$$\lambda x.\mathit{and}\,\texttt{false}\,x \stackrel{?}{=} \lambda x.\,\texttt{false}$$
$$\lambda x.\mathit{not}(\mathit{not}\ x) \stackrel{?}{=} \lambda x.x$$

The "gold star" definition of (untyped) program equivalence is *observational equivalence* (also known as *contextual equivalence*), which only checks that programs give "similar" results when run in any closing context. These contexts let you put the hole $\Box$ *anywhere* in a term, without restriction.

$$\begin{aligned}\mathit{Context} \ni C ::= \Box \mid C\ N \mid M\ C \mid \lambda x.C \\ \mid \texttt{if}\,C\,\texttt{then}\,N_1\,\texttt{else}\,N_2 \\ \mid \texttt{if}\,M\,\texttt{then}\,C\,\texttt{else}\,N_2 \\ \mid \texttt{if}\,M\,\texttt{then}\,N_1\,\texttt{else}\,C\end{aligned}$$

Definition 3.1 (Observational Equivalence (*a.k.a.* Contextual Equivalence))**.** Untyped *Observational (*a.k.a. *contextual) approximation*, written $M \preccurlyeq N$, means that N always evaluates to a similar value as M when they are both plugged into any closing context C:

$$\begin{aligned}M \preccurlyeq N \iff & \forall C \in \mathit{Context}.\, FV(C[M]) = FV(C[N]) = \{\} \implies \\ & \forall V \in \mathit{Value}.\, C[M] \mapsto\!\!\!\!\to_\beta V \implies \\ & \exists V \sim W.\, C[N] \mapsto\!\!\!\!\to_\beta W\end{aligned}$$

Untyped *Observational (*a.k.a. *contextual) equivalence*, written $M \approx N$, means that both terms approximate each other:

$$M \approx N \iff M \preccurlyeq N \text{ and } N \preccurlyeq M$$

Similarity of values, written $V \sim W$ is inductively defined as the smallest relation admitting these rules:

$$\begin{array}{ll} \texttt{true} \sim \texttt{true} & \\ \texttt{false} \sim \texttt{false} & \\ \lambda x.M \sim \lambda y.N & (\text{for } \textit{arbitrary } M, N \in \textit{Term}) \end{array}$$

3.1 Intensional Equality

3.1.1 Reduction theory as rewriting

The operational semantics only lets you step in certain contexts:

$$\frac{M \mapsto_\beta M'}{E[M] \mapsto_\beta E[M']}$$

In contrast, the reduction theory allows you to reduce in *any* context. In other words, the general reduction of a term, written as $M \to_\beta N$, is *compatible* with all contexts of the language:

$$\frac{M \to_\beta M'}{C[M] \to_\beta C[M']}\ \textit{Compatibility}$$

This is also known as *congruence*. In the base case, all of the specific steps of the operational semantics can be used in the reduction theory:

$$\frac{M \mapsto_\beta M'}{M \to_\beta M'}$$

The multi-step reduction relation, $M \twoheadrightarrow_\beta N$, is the reflexive, transitive closure of $M \to N$:

$$\frac{M \to_\beta M'}{M \twoheadrightarrow_\beta M'} \qquad \frac{}{M \twoheadrightarrow_\beta M}\ \textit{Refl.} \qquad \frac{M \twoheadrightarrow_\beta M' \quad M' \twoheadrightarrow_\beta M''}{M \twoheadrightarrow_\beta M''}\ \textit{Trans.}$$

Example 3.1. Using the ordinary (call-by-name *or* call-by-value) operational semantics, we can show that

$$\textit{and}\,\texttt{false}\,x \mapsto\!\!\twoheadrightarrow_\beta \texttt{false}$$

Since $\to_\beta$ is compatible with arbitrary contexts, that means we get:

$$\dfrac{\dfrac{\vdots\ \text{Left as exercise}}{\textit{and}\,\texttt{false}\,x \mapsto\!\!\twoheadrightarrow_\beta \texttt{false}}}{\textit{and}\,\texttt{false}\,x \twoheadrightarrow_\beta \texttt{false}}}{\lambda x.\textit{and}\,\texttt{false}\,x \twoheadrightarrow_\beta \lambda x.\texttt{false}}\ \textit{Compatibility}$$

3.1.2 Equational theory as rewriting

Reduction can be generalized to equality by letting you apply the rules forward (left-to-right) and backward (right-to-left).

The β-equality relation, $M =_\beta N$, is the reflexive, transitive, and *symmetric* closure of $M \to N$:

$$\frac{M \to_\beta M'}{M =_\beta M'} \qquad \frac{}{M =_\beta M}\ \textit{Refl.}$$

$$\frac{M =_\beta M' \quad M' \twoheadrightarrow_\beta M''}{M =_\beta M''}\ \textit{Trans.} \qquad \frac{M =_\beta M'}{M' =_\beta M}\ \textit{Symm.}$$

Example 3.2. Another function that always returns false is $\lambda x.\mathit{not}\,\texttt{true}$. We can prove this by reducing under the λ:

$$\cfrac{\cfrac{\cfrac{\vdots\ \text{Left as exercise}}{\mathit{not}\,\texttt{true} \mapsto_\beta \texttt{false}}}{\mathit{not}\,\texttt{true} \twoheadrightarrow_\beta \texttt{false}}}{\lambda x.\mathit{not}\,\texttt{true} \twoheadrightarrow_\beta \lambda x.\,\texttt{false}}$$

So how are $\lambda x.\mathit{not}\,\texttt{true}$ and $\lambda x.\mathit{and}\,\texttt{false}\,x$ related? They both reduce to a common function, $\lambda x.\,\texttt{false}$, but neither one reduces directly to the other. That's where β-equality can help: the two can be rewritten into one another by applying a chain of β-reduction steps forward *and* backward. Using what we already know:

$$\cfrac{\cfrac{\vdots}{\lambda x.\mathit{not}\,\texttt{true} \twoheadrightarrow_\beta \lambda x.\,\texttt{false}}}{\lambda x.\mathit{not}\,\texttt{true} =_\beta \lambda x.\,\texttt{false}} \quad \cfrac{\cfrac{\vdots}{\lambda x.\mathit{and}\,\texttt{false}\,x \twoheadrightarrow_\beta \lambda x.\,\texttt{false}}}{\lambda x.\mathit{and}\,\texttt{false}\,x =_\beta \lambda x.\,\texttt{false}}\ \textit{Symm.}}{\lambda x.\mathit{not}\,\texttt{true} =_\beta \lambda x.\mathit{and}\,\texttt{false}\,x}\ \textit{Trans.}$$

3.2 Soundness of Intensional Equality

Goal: to show soundness of equational theory with respect to observational equivalence — $M =_\beta N$ implies $M \approx N$.

3.2.1 Confluence: Relating equations & reductions

Because $M \to M'$ can apply a reduction in *any* context, there are often many choices of steps that can be done depending on where we choose to put the $\Box$. In other words, $M \to M'$ *non-deterministic*, in contrast to $M \mapsto M'$ which is deterministic. Given a starting M, it may reduce in multiple different ways $M_1 \leftarrow M \to M_2$ such that $M_1 \neq_\alpha M_2$.

But does this difference really matter, or do we have an illusion of choice that always leads to the same final conclusion. That is, if M splits in two paths as $M_1 \leftarrow M \rightarrow M_2$, does it always join back together at M' as $M_1 \rightarrow M \leftarrow M_2$?

Sort of. In λ-calculus (and many of its extensions), you can always join back together, but it can take multiple steps. Why? Substitution can duplicates a term, so it can duplicate reduction steps.

Lemma 3.1. *If $N \rightarrow_\beta N'$, then $M\,[N/x] \twoheadrightarrow_\beta M\,[N'/x]$.*

Proof. By induction on the syntax of M. Left as an exercise to the reader. ■

Example 3.3. This split from $(\lambda x.M)\ N$

$$(\lambda x.M)\ N' \leftarrow_\beta (\lambda x.M)\ N \rightarrow_\beta M\,[N/x]$$

can be joined to $M\,[N'/x]$ in multiple steps by Lemma 3.1:

$$(\lambda x.M)\ N' \rightarrow_\beta M\,[N'/x] \twoheadleftarrow_\beta M\,[N/x]$$

Weak confluence

Definition 3.2 (Diamond Property)**.** A reduction relation has the *diamond property* if every one-step split $M_1 \leftarrow M \rightarrow M_2$ can always be joined by multiple steps $M_1 \rightarrow M' \leftarrow M_2$.

Definition 3.3 (Weak Confluence)**.** A reduction relation is *weakly confluent* if every one-step split $M_1 \leftarrow M \rightarrow M_2$ can always be joined by multiple steps $M_1 \twoheadrightarrow M' \twoheadleftarrow M_2$.

Definition 3.4 (Strong Confluence)**.** A reduction relation is *strongly confluent* if every multi-step split $M_1 \twoheadleftarrow M \twoheadrightarrow M_2$ can always be joined by multiple steps $M_1 \twoheadrightarrow M' \twoheadleftarrow M_2$.

Lemma 3.2. *β-reduction of the λ-calculus is weakly confluent.*

Proof. Left as an exercise to the reader. ■

In certain circumstances, weak confluence can imply strong confluence (e.g. for strongly normalizing systems), but not always.

Example 3.4. Consider this little artificial rewriting system for coin flips:

$$\begin{aligned} \texttt{flip}\,\texttt{head} &\rightarrow \texttt{flip}\,\texttt{tail} \\ \texttt{flip}\,\texttt{tail} &\rightarrow \texttt{flip}\,\texttt{head} \\ \texttt{flip}\,M &\rightarrow M \end{aligned}$$

This system is weakly confluent (try to prove it), but not strongly confluent. Consider this multi-step split:

$$\texttt{head} \leftarrow \texttt{flip}\,\texttt{head} \rightarrow \texttt{flip}\,\texttt{tail} \rightarrow \texttt{tail}$$

There is no way to join $\texttt{head} \rightarrow \not M' \leftarrow \texttt{tail}$.

Critical pairs

The easiest method to prove strong confluence is to check for certain properties of the rewriting axioms themselves.

Definition 3.5 (Orthogonal Term Rewriting System)**.** A term rewriting system is *orthogonal* if:

1. The rules are all left-linear, i.e. the left-hand sides never mention the same meta-variable more than once. For example, $eq\ M\ M \to \texttt{true}$ is *not* left-linear because M appears twice on the left-hand side.
2. There are no critical pairs, i.e. the left-hand sides of any two rules never overlap.

Theorem 3.3. *All orthogonal term rewriting systems are strongly confluent.*

Proof. By Klop [1993]. Term Rewriting Systems. □

Corollary 3.4 (Strong Confluence)**.** *β-reduction of the λ-calculus is strongly confluent.*

Proof. Follows from Theorem 3.3 since all β-reduction rules are left-linear and have no critical pairs. □

Parallel reduction

Instead, we can bridge the gap between one-step and multi-step via *parallel reduction*, which allows you to reduce multiple different independent, parallel sub-terms "at once," but cannot chain together a sequence of steps that depend on one another. This notion of parallel reduction is nicely behaved in a certain way:

Property 3.6 (Parallel Reduction)**.** Given a single-step reduction relation $M \to N$, parallel reduction, written $M \Rightarrow N$, has the following properties:

a) $M \Rightarrow M$,

b) if $M \to M'$ then $M \Rightarrow M'$,

c) if $M \Rightarrow M'$ then $M \twoheadrightarrow M'$, and

d) if $M \Rightarrow M'$ and $N \Rightarrow N'$ then $M\,[N/x] \Rightarrow M'\,[N'/x]$.

Theorem 3.5 (Strong Confluence)**.** *If a parallel reduction relation has the diamond property, then the underlying reduction relation is strongly confluent.*

Proof. By induction on both reduction sequences of the split, using the properties of parallel reduction in Property 3.6. Left as an exercise to the reader. ■

Challenge 3.1. Try defining a β-parallel reduction relation with these properties. Giving an inductive definition directly on the syntax of terms can be more challenging, but also more rewarding in the following.

Theorem 3.6 (Diamond Property)**.** *Parallel β-reduction of the λ-calculus has the diamond property.*

Proof. Left as a challenge to the reader. ■

Convertibility of equality

Confluence is useful, because it says that every equality can be decided by just reducing both sides to some common reduct.

Theorem 3.7. *If a reduction relation $\to_R$ is strongly confluent, then $M =_R N$ implies $M \twoheadrightarrow_R M' \twoheadleftarrow_R N$ for some M'.*

Proof. Every rewriting equality $M =_R N$ can be expressed as a sequence of alternating reductions $M \twoheadleftarrow_R M_1 \twoheadrightarrow_R M_2 \twoheadleftarrow_R \ldots \twoheadrightarrow_R N$. The proof proceeds by induction on the number of alternations, using confluence to join together every split. □

Corollary 3.8 (Convertibility)**.** *$M_1 =_\beta M_2$ if and only if $M_1 \twoheadrightarrow_\beta M' \twoheadleftarrow_\beta M_2$ for some M'.*

3.2.2 Standardization: Relating reductions & operational semantics

Property 3.7 (Internal reduction)**.** Given a standard reduction relation $\mapsto$ and its compatible closure $\to$, an *internal reduction*, written $M \rightarrowtail M'$, has the properties:

a) if $M \rightarrowtail M'$ then $M \to M'$,

b) both $M \rightarrowtail M'$ and $M \mapsto M'$ is impossible, and

c) if $M \to M'$ then either $M \mapsto M'$ or $M \rightarrowtail M'$.

We write $\twoheadrightarrow$ to denote the reflexive, transitive closure of $\rightarrowtail$.

Lemma 3.9 (Postponement)**.** *If $M \twoheadrightarrow_\beta M_1 \mapsto_\beta M'$ then $M \mapsto\!\!\!\!\twoheadrightarrow_\beta M_2 \twoheadrightarrow_\beta M'$ for some M_2.*

Proof. Left as a challenge to the reader. ■

Hint. Try proving this simpler property first: If $M \twoheadrightarrow_\beta M_1 \mapsto_\beta M'$ then $M \mapsto\!\!\!\!\twoheadrightarrow_\beta M_2 \twoheadrightarrow_\beta M'$ for some M_2.

How can you generalize this weaker property — that starts with only one $\mapsto_\beta$ step after any number of $\twoheadrightarrow_\beta$ steps — to the full form in Lemma 3.9?

Lemma 3.10 (Standard Order)**.** *If $M \twoheadrightarrow_\beta N$ then $M \mapsto\!\!\!\!\twoheadrightarrow_\beta M' \twoheadrightarrow_\beta N$ for some M'.*

Proof. Every reduction sequence can be written as an alternation of internal and standard reduction steps, $M \mapsto\!\!\!\!\rightarrow_\beta M_1 \rightarrowtail\!\!\!\!\rightarrow_\beta M_2 \mapsto\!\!\!\!\rightarrow_\beta \ldots \rightarrowtail\!\!\!\!\rightarrow_i N$. Swapping each alternation to postpone internal reduction after standard reduction reduces the number of alternations by 1. Thus, we get $M \mapsto\!\!\!\!\rightarrow_\beta M' \rightarrowtail\!\!\!\!\rightarrow_\beta N$ from Lemma 3.9 by induction on the number of alternations. □

Lemma 3.11. *If $M \rightarrowtail_\beta V$, then M is a value.*

Proof. By induction on the possible internal reductions. Left as an exercise to the reader. ■

Theorem 3.12 (Standardization)**.** *If $M \twoheadrightarrow_\beta V$ then $M \mapsto\!\!\!\!\rightarrow_\beta W \twoheadrightarrow_\beta V$ for some W.*

Proof. From Lemma 3.10, we get $M \mapsto\!\!\!\!\rightarrow_\beta M' \rightarrowtail\!\!\!\!\rightarrow_\beta V$, and we know M' must be a value from Lemma 3.11. □

3.2.3 Confluence & Standardization $\Longrightarrow$ Soundness

Lemma 3.13. *Values are closed under reduction: if $V \twoheadrightarrow_\beta M$ then M is a value.*

Proof. By induction on the possible reductions from V. Left as an exercise to the reader. ■

Lemma 3.14. *If $V_1 \twoheadrightarrow_\beta V' \twoheadleftarrow_\beta V_2$, then $V_1 \sim V_2$.*

Proof. By induction on the possible reductions from V_1 and V_2. Left as an exercise to the reader. ■

Theorem 3.15 (Soundness)**.** *If $M =_\beta N$ then $M \approx N$.*

Proof. We first show $M =_\beta N$ implies $M \preccurlyeq N$, so we need to demonstrate that $C[M] \mapsto\!\!\!\!\rightarrow_\beta V$ implies $V \sim W \twoheadleftarrow\!\!\!\!\dashv_\beta C[N]$ for every possible closing context C around M and N.

Let C be any context such that both $C[M]$ and $C[N]$ are closed. Because β-equality is compatible with arbitrary contexts by definition, $M =_\beta N$ implies $C[M] =_\beta C[N]$.

First, suppose that $C[M] \mapsto\!\!\!\!\rightarrow_\beta V$. We know that $C[N] =_\beta V$ by symmetry and transitivity of equality. Applying Corollary 3.8 to $C[N] =_\beta V$ gives $C[N] \twoheadrightarrow_\beta W \twoheadleftarrow_\beta V$ and we know W must be a value because values are closed under reduction (Lemma 3.13). Applying Theorem 3.12 to $C[N] \twoheadrightarrow_\beta W$ gives

$$C[N] \mapsto\!\!\!\!\rightarrow_\beta W' \twoheadrightarrow_\beta W \twoheadleftarrow_\beta V$$

Applying Lemma 3.14 to $W' \twoheadrightarrow_\beta W \twoheadleftarrow_\beta V$ gives $W' \sim V$. Thus, $M \preccurlyeq N$.

Since equality is symmetric, $M =_\beta N$ implies $N =_\beta M$ which implies $N \preccurlyeq M$ as above. Thus $M =_\beta N$ implies $M \approx N$. □

Example 3.5. Before, we used the syntactic β rule from the ordinary (call-by-value or call-by-name) operational semantics to relate two functions by rewriting them into a common form of a function $\lambda x.\,\texttt{false}$ that always returns false:

$$\lambda x.\mathit{and}\,\texttt{false}\,x =_\beta \lambda x.\,\texttt{false} =_\beta \lambda x.\mathit{not}\,\texttt{true}$$

Soundness of intensional equality (Theorem 3.15) then tells us that these purely syntactic rewritings actually prove a property about how the functions will behave in arbitrary programs:

$$\lambda x.\mathit{and}\,\texttt{false}\,x \approx \lambda x.\,\texttt{false} \approx \lambda x.\mathit{not}\,\texttt{true}$$

which means that all three functions can be exchanged in *any* context without changing the result of the program.

3.3 Extensional Equational Theory: η equality

Despite being able to prove many programs are observationally equivalent, still some obvious equalities elude us. For example, $\lambda x.\mathit{not}(\mathit{not}\ x)$ should be the same as the identity function $\lambda x.x$ on booleans, but trying to reduce inside the body of the function as it stands quickly gets stuck.

Intuitively, we know that x should only stand for $\texttt{true}$ or $\texttt{false}$, and in both cases simplify to themselves: $\mathit{not}(\mathit{not}\,\texttt{true}) \mapsto\!\!\!\!\rightarrow_\beta \texttt{true}$ and $\mathit{not}(\mathit{not}\,\texttt{false}) \mapsto\!\!\!\!\rightarrow_\beta \texttt{false}$. However, this doesn't always work in *arbitrary* contexts, because we might plug in something else, like a λ, and $\mathit{not}(\mathit{not}(\lambda x.M))$ gets stuck instead of simplifying down to $\lambda x.M$.

So $\lambda x.\mathit{not}(\mathit{not}\ x) \not\approx \lambda x.x$ in the sense of Definition 3.1 that lets the closing context do *anything*. However, if we restrict the scope of contexts to only those that make sense — in the sense of type checking — then maybe there is still hope?

Definition 3.8 (Observational Equivalence (*a.k.a.* Contextual Equivalence))**.** Typed *observational (*a.k.a. *contextual) equivalence*, written $\Gamma \vdash M \approx N : A$, means that M and N always evaluate to similar values when they are both plugged into any closing context C that takes an input of type A to an output of the "ground" type $\texttt{bool}$:

$$\begin{aligned}\Gamma \vdash M \approx N : A \iff\ & \Gamma \vdash M : A \text{ and } \Gamma \vdash N : A \text{ and} \\ & \forall C \in \mathit{Context}.\ \bullet \vdash C[M] : \texttt{bool} \text{ and } \bullet \vdash C[N] : \texttt{bool} \implies \\ & \exists V \sim W : \texttt{bool}\,.\ C[M] \mapsto\!\!\!\!\rightarrow_\beta V \sim W \mathrel{\leftarrow\!\!\!\!\mapsto_\beta} C[N]\end{aligned}$$

Similarity of typed ground values, written $V \sim W : A$ is inductively defined as the smallest relation admitting these rules:

$$\texttt{true} \sim \texttt{true} : \texttt{bool} \qquad\qquad \texttt{false} \sim \texttt{false} : \texttt{bool}$$

Now, we will define a typed equality relation $\Gamma \vdash M = N : A$ which asserts that two terms M and N of type A are equal. Similar to before, this equality relation extends the operational semantics (every β step *is* an equality), and is reflexive, transitive, and symmetric.

$$\frac{\Gamma \vdash M : A \quad M \mapsto_\beta M' \quad \Gamma \vdash M' = N : A}{\Gamma \vdash M = N : A}\ \textit{Step} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash M = M : A}\ \textit{Refl.}$$

$$\frac{\Gamma \vdash M = N : A \quad \Gamma \vdash N = M' : A}{\Gamma \vdash M = M' : A}\ \textit{Trans.} \qquad \frac{\Gamma \vdash M = N : A}{\Gamma \vdash N = M : A}\ \textit{Symm.}$$

3.3.1 Extensionality rules & η axioms

To give some extra *oomph* to typed equality, we will add some extra equations that use typing information to rule out nonsensical possibilities and push computation forward. This gives us a notion of *extensional* that is only concerned with external input-output behavior, and isn't as sensitive to internal details of how programs are exactly written.

Extensionality of functions — two functions are equal when they give equal answers to *all* possible equal inputs:

$$\frac{\Gamma, x : A \vdash M\ x = N\ x : B \quad x \notin FV(M) \cup FV(N)}{\Gamma \vdash M = N : A \to B} \to X$$

Extensionality of booleans — every use-case of equivalent booleans can assume it is either `true` or `false`:

$$\frac{\Gamma \vdash M = N : \texttt{bool} \quad \Gamma \vdash E[\texttt{true}] = E'[\texttt{true}] : A \quad \Gamma \vdash E[\texttt{false}] = E'[\texttt{false}] : A}{\Gamma \vdash E[M] = E'[N] : A}\ \texttt{bool}X$$

Alternatively, we can capture this notion of extensionality in the form of rewriting axioms that operate over typed terms:

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

$$\frac{\Gamma \vdash M : \texttt{bool} \quad \Gamma, x : \texttt{bool} \vdash E[x] : A \quad x \notin FV(E)}{\Gamma \vdash \texttt{if}\ M\ \texttt{then}\ E[\texttt{true}]\ \texttt{else}\ E[\texttt{false}] = E[M] : A}\ \eta\texttt{bool}$$

The above axioms can be written in a more familiar way by presenting the typing premises as side conditions:

$$(\eta\to) \qquad (\lambda x.M\ x) = M \quad : A \to B \quad (\text{if } x \notin FV(M))$$

$$(\eta\texttt{bool}) \quad \texttt{if}\ M\ \texttt{then}\ E[\texttt{true}]\ \texttt{else}\ E[\texttt{false}] = E[M] : A \qquad (\text{if } M : \texttt{bool})$$

Exercise 3.2. Prove that the η axioms can be derived from the extensionality inference rules

1. Use the extensionality rule $\to X$ and the call-by-name operational semantics to conclude the equality of the $\eta\to$ axiom applied to a typed term $M : A \to B$.

2. Use the extensionality rule $\texttt{bool}X$ and the call-by-name operational semantics to conclude the equality of the $\eta\texttt{bool}$ axiom applied to a typed term $M : \mathit{Bool}$ and evaluation context $E[x] : A$ (assuming $x : \texttt{bool}$).

3.3.2 Typed congruence

Typed equality should be congruent, i.e. compatible with all (type-preserving) contexts. We could do this in one rule:

$$\frac{\Gamma \vdash M = N : A \quad \Gamma' \vdash C[M] : B \quad \Gamma' \vdash C[N] : B}{\Gamma' \vdash C[M] = C[N] : B}\ \mathit{Compat}.$$

But this rule can be a little unwieldy to reason about and formalize.

Instead, we can simplify the specification of congruence by just "doubling up" all the ordinary typing rules. Each rule building single well-typed term out of well-typed sub-terms can be generalized to stating that two typed terms are equal if they are made of equal sub-terms. This looks like:

$$\frac{}{\Gamma, x : A \vdash x = x : A}\ \mathit{Var}^2$$

$$\frac{\Gamma, x : A \vdash M = M' : B}{\Gamma \vdash \lambda x.M = \lambda x.M' : A \to B} \to I^2 \qquad \frac{\Gamma \vdash M = M' : A \to B \quad \Gamma \vdash N = N' : A}{\Gamma \vdash M\ N = M'\ N' : B} \to E^2$$

$$\frac{}{\Gamma \vdash \texttt{true} = \texttt{true} : \texttt{bool}}\ \texttt{bool}I_1^2 \qquad \frac{}{\Gamma \vdash \texttt{false} = \texttt{false} : \texttt{bool}}\ \texttt{bool}I_2^2$$

$$\frac{\Gamma \vdash M = M' : \texttt{bool} \quad \Gamma \vdash N_1 = N_1' : A \quad \Gamma \vdash N_2 = N_2' : A}{\Gamma \vdash \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 = \texttt{if}\ M'\ \texttt{then}\ N_1'\ \texttt{else}\ N_2' : A}\ \texttt{bool}E^2$$

Exercise 3.3. Show that the doubled-up congruence rules (Var^2, $\to I^2$, $\to E^2$, ...) make both reflexivity and compatibility redundant:

1. Given an arbitrary typing derivation of $\Gamma \vdash M : A$, prove that $\Gamma \vdash M = M : A$ is derivable without using *Refl*.

2. Given a derivation that two terms are equal $\Gamma \vdash M = M' : A$ as well as a proof that a context is well typed $\Gamma, x : A \vdash C[x] : B$ for a fresh variable x (i.e. $x \notin FV(C)$) prove that $\Gamma \vdash C[M] = C[M'] : B$ is derivable without using *Compat*.

Exercise 3.4. Prove that the extensionality inference rules can be derived from the η axioms:

1. Use the $\eta\to$ axiom to derive the $\to X$ rule.

2. Use the $\eta\texttt{bool}$ axiom to derive the $\texttt{bool}X$ rule.

Hint. You may need to use compatibility.

3.3.3 Logical relation of typed equivalence

Now, how do we show that the syntactic, rewriting-based notion of extensional equality ($\Gamma \vdash M = N : A$) is a sound approximation of the more behavioral notion of typed observational equivalence ($\Gamma \vdash M \approx N : A$)? Logical relations!

First, specify how programs of each type are expected to interact with equivalence as a binary relation between terms.[1]

$$\begin{aligned} [\![\_]\!] &: \mathit{Type} \to \mathrm{Rel}(\mathit{Term}, \mathit{Term}) \\ M\ [\![\texttt{bool}]\!]\ M' &\iff M \mapsto\!\!\!\!\to_\beta \texttt{true} \mathrel{\reflectbox{$\mapsto\!\!\!\!\to$}}_\beta M' \text{ or } M \mapsto\!\!\!\!\to_\beta \texttt{false} \mathrel{\reflectbox{$\mapsto\!\!\!\!\to$}}_\beta M' \\ M\ [\![A \to B]\!]\ M' &\iff \forall N\ [\![A]\!]\ N'.\ (M\ N)\ [\![B]\!]\ (M'\ N') \end{aligned}$$

Notice that reflexivity of $\mapsto\!\!\!\!\to_\beta$ means the boolean equivalence relationship includes both $\texttt{true}\ [\![\texttt{bool}]\!]\ \texttt{true}$ and $\texttt{false}\ [\![\texttt{bool}]\!]\ \texttt{false}$, as expected. You might think that these two facts are enough, but we also want to be able to relate boolean programs that return similar results. For example, we should be able to say $(\mathit{not}\,\texttt{true})\ [\![\texttt{bool}]\!]\ \texttt{false}$, too, because $\mathit{not}\,\texttt{true}$ returns $\texttt{false}$. The use of reduction in the definition of $[\![\texttt{bool}]\!]$ expands the relationship to include these other programs which will eventually return a $\texttt{true}$ or $\texttt{false}$ after some work.

Exercise 3.5. Show that the relationship $(\mathit{not}\,\texttt{true})\ [\![\texttt{bool}]\!]\ (\mathit{and}\,\texttt{false}\,x)$ holds.

Typing environments specify the valid substitutions into an open term. An environment Γ is interpreted as a relationship between two substitutions that plug in related inputs according to the types assigned to each free variable. To do this, we generalize from single substitution to simultaneous substitution, $M\,[N_1/x_1, \ldots, N_n/x_n]$, defined similarly to avoid capturing free variables.

$$\mathit{Substitution} \ni \sigma ::= M_1/x_1, \ldots, M_n/x_n$$

$$\begin{aligned} [\![\_]\!] &: \mathit{Environment} \to \mathrm{Rel}(\mathit{Substitution}, \mathit{Substitution}) \\ \sigma\ [\![\Gamma]\!]\ \sigma' &\iff \forall (x : A) \in \Gamma.\ x\,[\sigma]\ \ [\![A]\!]\ x\,[\sigma'] \end{aligned}$$

Now, finally, typed equality judgements are interpreted as the statement that the supposedly equal terms at type A are actually related by $[\![A]\!]$ for *all* possible substitutions allowed by the typing environment:[2]

$$\begin{aligned} [\![\_]\!] &: \mathit{Judgement} \to \mathrm{Prop} \\ [\![\Gamma \vdash M = M' : A]\!] &\iff \forall \sigma\ [\![\Gamma]\!]\ \sigma'.\ M\,[\sigma]\ \ [\![A]\!]\ M'\,[\sigma'] \end{aligned}$$

The *fundamental* property of the logical relation is that the inductively-defined derivations of syntactic equality $\Gamma \vdash M = N : A$ can always be transformed into

[1] A *binary relation* between two sets A and B, written $\mathrm{Rel}(A, B)$, can be represented as a subset of all possible pairs of A and B elements, $\mathrm{Rel}(A, B) = \wp(A \times B)$. We will leave the specific encoding of relations as abstract in the presentation.

[2] A *proposition*, written Prop, is a statement that could be true or false. An individual proposition can be represented by the two-element set $\{tt, ff\}$ deciding its truth value. Note that the mapping $A \to \mathrm{Prop}$ decides a truth value for each element of A, and is equivalent to a predicate on A that is sometimes true and sometimes false.

their interpretation as a behavioral proposition $[\![\Gamma \vdash M = N : A]\!]$. To prove this, we need to use a special property that is true of the interpretation for every type: they are interpreted as relations that are *closed under expansion*.

Lemma 3.16 (Closure Under Expansion)**.**
For all types A, if $M \mapsto\!\!\!\!\rightarrow_\beta M'$ $[\![A]\!]$ $N' \twoheadleftarrow_\beta N$ then M $[\![A]\!]$ N.

Proof. By induction on the syntax of A. Left as an exercise to the reader. ■

As a warm up, you can just try proving the reflexive case:

Lemma 3.17. *If $\Gamma \vdash M : A$ is derivable then $[\![\Gamma \vdash M = M : A]\!]$ is true.*

Proof. By induction on the derivation of $\Gamma \vdash M : A$. Left as an exercise to the reader. ■

To prove soundness of typed $\beta\eta$-equality using syntactic rewriting rules, we also need to handle transitivity (combining together many steps) and symmetry (reversing the direction of steps). This can be

Lemma 3.18 (Partial Equivalence)**.** *For all types A, $[\![A]\!]$ is a* partial equivalence relation*, i.e. it is*

a) Symmetric*: if M $[\![A]\!]$ N then N $[\![A]\!]$ M, and*

b) Transitive*: if M $[\![A]\!]$ N and N $[\![A]\!]$ M' then M $[\![A]\!]$ M'.*

Proof. By induction on the syntax of A. Left as an exercise to the reader. ■

Lemma 3.19 (Fundamental Property of the Logical Relation)**.**
If $\Gamma \vdash M = N : A$ is derivable then $[\![\Gamma \vdash M = N : A]\!]$ is true.

Proof. By induction on the derivation of $\Gamma \vdash M = N : A$. Left as a challenge to the reader. ■

Hint. You may assume that the given derivation of $\Gamma \vdash M = N : A$ does *not* use the *Refl.* or *Compat.* rules, based on Exercise 3.3. In place of these general rules, you can assume the derivation only uses the "doubled-up" versions of the regular typing rules, along with the *Step*, *Symm.*, and *Trans.*, as well as either the η axioms or extensionality X rules for every type (your choice).

Hint. It can help to also consider the interpretation of inference rules themselves. Given an inference rule of the form (where J and H_i are judgements)

$$\frac{H_1 \quad H_2 \quad \dots \quad H_n}{J}\ \mathit{Rule}$$

its interpretation as a logical proposition, $[\![\mathit{Rule}]\!]$ is

$$\left[\!\!\left[\frac{H_1 \quad H_2 \quad \dots \quad H_n}{J}\ \mathit{Rule}\right]\!\!\right]$$

$\iff$ If $[\![H_1]\!]$ and $[\![H_2]\!]$ and $\dots$ and $[\![H_n]\!]$ are all true, then $[\![J]\!]$ is true.

You can then justify *Rule* is sound by proving that $[\![\mathit{Rule}]\!]$ proposition is always true. A derivation made up of several of these rules — all proved sound in this way — then proves the conclusion is true without any open premises.

Lemma 3.20. *Given a derivation $\mathcal{D}$ concluding a judgement J, if $\mathcal{D}$ is built by rules R_i such that $[\![R_i]\!]$ are all true, then $[\![J]\!]$ is true.*

Proof. By induction on the derivation $\mathcal{D}$, using the assumption that $[\![R_i]\!]$ is true in each case R_i. Left as an exercise for the reader. ■

You can then simplify the proof of Lemma 3.19 to sequence shorter lemmas showing why the logical interpretation of each inference rule — $[\![\mathit{Var}^2]\!]$, $[\![{\to}I^2]\!]$, $[\![{\to}E^2]\!]$, $[\![{\to}X]\!]$, $[\![\mathtt{bool}I^2]\!]$, $[\![\mathtt{bool}E^2]\!]$, $[\![\mathtt{bool}X]\!]$, $[\![\mathit{Step}]\!]$, $[\![\mathit{Symm.}]\!]$, $[\![\mathit{Trans.}]\!]$ — is true.

Hint. Notice that, using Lemma 3.16 makes it easy to justify this equality rule:

$$\frac{M \mapsto\!\!\!\!\rightarrow_\beta M' \quad \Gamma \vdash M' = N' : A \quad N' \twoheadleftarrow_\beta N}{\Gamma \vdash M = N : A}\ \mathit{Expand}$$

In other words, $[\![\mathit{Expand}]\!]$ is true, and the proof follows immediately from Lemma 3.16 and the fact that reduction is closed under substitution — if $M \mapsto_\beta M'$ then $M\,[N/x] \mapsto_\beta M'\,[N/x]$ (by cases on the $\mapsto_\beta$ rules).

A key difference between *Expand* versus *Step* is that *Expand* doesn't check the types of the starting terms M and N. In other words, *Expand* potentially lets you equate two terms M and N at a type A even if M and N themselves don't appear to have type A! This goes against a design goal that typed equality only relates terms that syntactically belong to that type. However, bending this rule has a huge advantage: it lets us cut down on the number of equality rules we have to consider.

Of course *Step* can be derived in terms of *Expand*. But what is more interesting is that both ${\to}I^2$ and $\mathtt{bool}E^2$ can *also* be derived from *Expand* along with their respective extensionality rules.

As an exercise, see if you can derive proofs of

$$\begin{array}{c}\Gamma, x : A \vdash M = M' : B \\ \vdots\ \mathit{Expand}, {\to}X \\ \Gamma \vdash \lambda x.M = \lambda x.M' : A \to B\end{array}$$

$$\begin{array}{c}\Gamma \vdash M = M : \mathtt{bool} \quad \Gamma \vdash N_1 = N_1' : A \quad \Gamma \vdash N_2 = N_2' : A \\ \vdots\ \mathit{Expand}, \mathtt{bool}X \\ \Gamma \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 = \mathtt{if}\ M'\ \mathtt{then}\ N_1'\ \mathtt{else}\ N_2' : A\end{array}$$

using *Expand* along with ${\to}X$ and $\mathtt{bool}X$, respectively. Doing so means all proofs of equality can done without using the *Step*, ${\to}I^2$, and $\mathtt{bool}E^2$ rules if we add *Expand* to the system, so we wouldn't need to address these cases in our inductive proof of Lemma 3.19 (as they have been removed already and replaced by other rules) as long as we have a case for *Expand*. Looking at this

fact another way, proving $[\![\mathit{Expand}]\!]$, $[\![{\to}X]\!]$, and $[\![\mathtt{bool}X]\!]$ are true automatically entails $[\![{\to}I^2]\!]$ and $[\![\mathtt{bool}E^2]\!]$ are true by just following the logical interpretation of inference rules.

From this fundamental property, we can prove that the inductively-defined, rewriting-based, extensional equality is sound (i.e. approximates) the behavior-based, typed observational equivalence.

Theorem 3.21 (Soundness)**.** *If* $\Gamma \vdash M = N : A$ *then* $\Gamma \vdash M \approx N : A$.

Proof. To demonstrate the observational equivalence, we have to show that M and N reduce to the same boolean value when plugged into any type-preserving, closing context with the return type `bool`. So let C be any context such that $\bullet \vdash C[M] : \mathtt{bool}$ and $\bullet \vdash C[N] : \mathtt{bool}$.

From compatibility (either the *Compat.* rule or its derivation from doubled-up congruence rules Var^2, ${\to}I^2$, ...as in Exercise 3.3), we can build a derivation of $\bullet \vdash C[M] = C[N] : \mathtt{bool}$. Applying Lemma 3.19, $[\![\bullet \vdash C[M] = C[N] : \mathtt{bool}]\!]$ must be true. Note that the proposition $[\![\bullet \vdash C[M] = C[N] : \mathtt{bool}]\!]$ is defined to mean the same thing as:

$$C[M] \mapsto\!\!\!\!\to_\beta c \twoheadleftarrow_\beta C[N]$$

where c is either `true` or `false`, as required by typed observational equivalence.

Therefore, $\Gamma \vdash M \approx N : A$. □

3.3.4 Evaluation order and extensionality

The extensionality rules presented only really work for call-by-name evaluation, particularly when we have general loops or side effects. For example, $y{+}{+}; f \neq \lambda x.(y{+}{+}; f)\ x$.

So for call-by-value, we have to restrict extensionality of functions to only function *values*, and not arbitrary function-producing code.

$$(\eta{\to}) \qquad \lambda x.V\ x = V : A \to B \qquad (x \notin FV(V))$$

$$\frac{\Gamma, x : A \vdash V\ x = W\ x : B \quad x \notin FV(V) \cup FV(W)}{\Gamma \vdash V = W : A \to B}\ {\to}X$$

Happily, the extensionality rules presented here for booleans in call-by-name still apply in call-by-value. But notice that call-by-value has a different definition of evaluation contexts, E, which adds more cases. What impact does that have on the notion of boolean extensionality?

Exercise 3.6. Interpret the $\eta\mathtt{bool}$ and/or $\mathtt{bool}X$ rules with call-by-value operational semantics (the definition of *EvalCxt* and reduction steps) to prove the following more general extensionality principles of booleans:

$$\frac{\Gamma \vdash M\,[\mathtt{true}/x] = N\,[\mathtt{true}/x] : A \quad \Gamma \vdash M\,[\mathtt{false}/x] = N\,[\mathtt{false}/x] : A}{\Gamma, x : \mathtt{bool} \vdash M = N : A}\ \mathtt{bool}X_V$$

$$(\eta\texttt{bool}_V) \quad \texttt{if}\, V\, \texttt{then}\, N\,[\texttt{true}/x]\, \texttt{else}\, N\,[\texttt{false}/x] = N\,[V/x] : A \qquad (\text{if } V : \texttt{bool}, x \notin FV(N))$$

Exercise 3.7. Come up with a counter-example where the call-by-value extensionality principle for booleans ($\texttt{bool}X_{CBV}$ or $\eta\texttt{bool}_{CBV}$) *breaks* observational equivalence under a call-by-name operational semantics. To come up with your counter-example, you may assume that the language has some form of looping construct (e.g. recursive functions or recursive types) such that the non-terminating $\Omega \mapsto_\beta \Omega$ can be written.

Chapter 4

Polymorphism & Modularity

"Simple" types, the ones we have seen thus far, are extremely limited in practice. Consider just the basic identity function

$$id(x) = x$$

represented in λ-calculus as $\lambda x.x$. What is it's type? It could be `bool` $\to$ `bool`, but it just as well could be `int` $\to$ `int` or (`bool` $\to$ `bool`) $\to$ (`bool` $\to$ `bool`). It would be a shame — as well as intractable — to copy and paste the *exact* same code for every single type we want to use it at. That's why cool languages give us *parametric polymorphism* (*a.k.a. generics*), to capture all these instances in a single type for generic code. In ML- and Haskell-like languages, this looks like

$$id : \alpha \to \alpha$$

where α stands for a generic, unknown type that could be instantiated with any specific type like `bool` or `int` or `list nat`. To be more explicit about this generalization over the type α, we can add a quantifier $\forall$ meaning "for all" that introduces and abstracts over the type variable:

$$id : \forall\alpha.\ \alpha \to \alpha$$

The exciting thing is: the generic type abstraction $\forall$ corresponds exactly to the universal quantifier $\forall$ from logic! Thanks Girard [1972] and Reynolds [1974]!

Another issue in practical programming is modularity. I might implement first-in-first-out queues with the usual *enqueue* and *dequeue* operations. Of course, queues need *some* sort of specific representation for me to implement those operations. But I don't want you to exploit that choice of representation in the code that imports it. Maybe I will change the representation at some point to optimize or improve queues somehow, and your code that uses my queues should still work fine as long as I implement *enqueue* and *dequeue* correctly.

This problem is solved by *modules* which set up boundaries that protect certain information during cross-code linking to help improve maintainability. A module that implements queues might have a signature like this:

$$\begin{aligned} \mathit{queue} : \{\, & \texttt{type}\,\alpha; \\ & \mathit{empty} : \alpha \\ & \mathit{enqueue} : \texttt{int} \to \alpha \to \alpha \\ & \mathit{dequeue} : \alpha \to \texttt{maybe}\,\texttt{int} \\ \} & \end{aligned}$$

As you may have guessed, this corresponds to another logical quantifier, $\exists$, which says *there exists* some type α such that the following type makes sense. The queue module signature can be represented by $\exists$ as

$$\begin{aligned} \mathit{queue} : \exists\alpha.\ & \alpha \\ & \times (\texttt{int} \to \alpha \to \alpha) \\ & \times (\alpha \to 1 + \texttt{int}) \end{aligned}$$

4.1 Type Abstraction

Start with the typing rules first, since the whole point is about types. We add two new types for $\forall$ and $\exists$ quantifiers which abstract over type variables (α, β, ...). So like with recursive types, we need to be able to use type variables as another form of type.

$$\begin{aligned} \mathit{TypeVar} \ni\ & \alpha, \beta ::= \ldots \\ \mathit{Type} \ni\ & A, B ::= \ldots \mid \alpha \mid \forall\alpha.\tau \mid \exists\alpha.\tau \end{aligned}$$

Typing judgements are generalized to also include an environment of free type variables $\Theta = \alpha, \beta, \ldots$, and are written as $\Theta\ ;\ \Gamma \vdash M : A$. Θ is a set (an unordered list, with at most one copy of any given type variable). We also have a judgement for checking that types are well-formed, where a derivation of $\Theta \vdash A : \star$ implies $FV(A) \subseteq \Theta$.

$$\begin{aligned} \mathit{Generics} \ni \Theta &::= \alpha_1, \ldots, \alpha_n \\ \mathit{Judgement} &::= \Theta\ ;\ \Gamma \vdash M : A \\ &\mid \Theta \vdash A : \star \end{aligned}$$

$$\frac{}{\Theta, \alpha \vdash \alpha : \star}\ \mathit{VarT} \qquad \frac{}{\Theta \vdash \texttt{bool} : \star}\ \texttt{bool}T \qquad \frac{\Theta \vdash A : \star \quad \Theta \vdash B : \star}{\Theta \vdash A \to B : \star}\ {\to}T$$

$$\frac{\Theta, \alpha \vdash \tau : \star}{\Theta \vdash \forall\alpha.\tau : \star}\ \forall T \qquad \frac{\Theta, \alpha \vdash \tau : \star}{\Theta \vdash \exists\alpha.\tau : \star}\ \exists T$$

Note that we only consider typing judgements $\Theta\,;\, x_1 : A_1, \ldots, x_n : A_n \vdash M : B$ *well-formed* when $\Theta \vdash B : \star$ and $\Theta \vdash A_i : \star$ for $1 \leq i \leq n$.

$$\frac{\Theta, \alpha\,;\, \Gamma \vdash M : \tau}{\Theta\,;\, \Gamma \vdash \Lambda\alpha.M : \forall\alpha.\tau}\ \forall I \qquad \frac{\Theta\,;\, \Gamma \vdash M : \forall\alpha.B \quad \Theta \vdash A : \star}{\Theta\,;\, \Gamma \vdash M\ A : B\,[A/\alpha]}\ \forall E$$

$$\frac{\Theta \vdash A : \star \quad \Theta\,;\, \Gamma \vdash M : B\,[A/\alpha]}{\Theta\,;\, \Gamma \vdash (A, M) : \exists\alpha.B}\ \exists I$$

$$\frac{\Theta\,;\, \Gamma \vdash M : \exists\alpha.B \quad \Theta, \alpha\,;\, \Gamma, x : B \vdash M : A \quad \Theta \vdash A : \star}{\Gamma \vdash \texttt{case}\, M\, \texttt{of}\ \{\, (\alpha, x{:}B) \Rightarrow M \,\} : A}\ \exists E$$

Note, the $\Theta \vdash A : \star$ premise of $\exists E$ ensures that local type variable α in the pattern-match doesn't accidentally escape into the return type A.

These local type variables (introduced by big lambdas Λ or a `case` matching on an existential package) can be used in type annotations. To help a computer check the types of functions, we need to give a hint that says what is the type of the argument. The easiest place to insert this annotation is on the places where variables are introduced, like in the `case` expression above or a λ-abstraction itself:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A \to B}\ \to I$$

All of these type annotations can be omitted from specific examples if they are obvious from the context.

4.2 Syntax and Operational Semantics

Extended syntax of terms

$$\begin{aligned} \mathit{Term} \ni M, N ::= {} & \ldots \mid \lambda x{:}A.M \\ & \mid \Lambda\alpha.M \mid M\ A \\ & \mid (A, M) \mid \texttt{case}\, M\, \texttt{of}\ \{\, (\alpha, x : B) \Rightarrow N \,\} \end{aligned}$$

Extended operational semantics

$$\mathit{EvalCxt} \ni E ::= \ldots \mid E\ A \mid \texttt{case}\, E\, \texttt{of}\ \{\, (\alpha, x{:}B) \Rightarrow N \,\}$$

$$\begin{aligned} (\beta\to) \qquad & (\lambda x{:}A.M)\ N \mapsto M\,[N/x] \\ (\beta\forall) \qquad & (\Lambda\alpha.M)\ A \mapsto M\,[A/\alpha] \\ (\beta\exists) \qquad & \texttt{case}\ (A, M)\ \texttt{as}\ (\alpha, x{:}B) \Rightarrow N \mapsto N\,[A/\alpha]\,[M/x] \end{aligned}$$

Notice: the typing annotations are totally ignored by the operational semantics. Annotations are *irrelevant* at run-time, and all types can be *erased* at compile-time.

Exercise 4.1. Give an alternate call-by-value semantics to $\forall$ and $\exists$ types.

4.3 Extensional Equational Theory

Congruence of type abstraction:

$$\frac{\Theta, \alpha \,;\, \Gamma \vdash M = M' : B}{\Theta \,;\, \Gamma \vdash \Lambda\alpha.M = \Lambda\alpha.M' : \forall\alpha.B}\ \forall I^2 \qquad \frac{\Theta \,;\, \Gamma \vdash M = M' : \forall\alpha.B}{\Theta \,;\, \Gamma \vdash M\ A = M'\ A : B\,[A/\alpha]}\ \forall E^2$$

$$\frac{\Theta \,;\, \Gamma \vdash M = M' : B\,[A/\alpha]}{\Theta \,;\, \Gamma \vdash (A, M) = (A, M') : \exists\alpha.B}\ \exists I^2$$

$$\frac{\Theta \,;\, \Gamma \vdash M = M' : \exists\alpha.B \quad \Theta, \alpha \,;\, \Gamma, x : B \vdash N = N' : A \quad \alpha \notin FV(A)}{\Theta \,;\, \Gamma \vdash \mathtt{case}\, M\, \mathtt{of}\, \{\, (\alpha, x{:}B) \Rightarrow N \,\} = \mathtt{case}\, M'\, \mathtt{of}\, \{\, (\alpha, x{:}B) \Rightarrow N' \,\} : A}\ \exists E^2$$

η axioms:

$$(\eta\forall) \qquad (\Lambda\alpha.M\ \alpha) = M : \forall\alpha.B \quad (\text{if } \alpha \notin FV(M))$$

$$(\eta\exists) \quad \mathtt{case}\, M\, \mathtt{of}\, \{\, (\alpha, x{:}B) \Rightarrow E[(\alpha, x)] \,\} = E[M] : A \quad (\text{if } M : \exists\alpha.B)$$

Extensionality rules:

$$\frac{\Theta, \alpha \,;\, \Gamma \vdash M\ \alpha = M'\ \alpha : B}{\Theta \,;\, \Gamma \vdash M = M' : \forall\alpha.B}\ \forall X$$

$$\frac{\Theta \,;\, \Gamma \vdash M = M' : \exists\alpha.B \quad \Theta, \alpha \,;\, \Gamma, x : B \vdash E[(\alpha, x)] = E'[(\alpha, x)] : A}{\Theta \,;\, \Gamma \vdash E[M] = E'[M'] : A}\ \exists X$$

Note: to keep the typing judgements in the conclusions well-formed, need to make sure that $\alpha \notin FV(M) \cup FV(M') \cup FV(B)$ in the $\forall X$ rule, and that $x \notin FV(E) \cup FV(E')$ and $\alpha \notin FV(E) \cup FV(E') \cup FV(A)$ in the $\exists X$ rule.

Exercise 4.2. Give a revised version of the $\eta\forall$ axiom and $\forall X$ rule that are sound in call-by-value. Specifically, these rules should not be able to equate values with non-value terms on their own without the help of β-reduction.

Exercise 4.3. Interpret the $\eta\exists$ and/or $\exists X$ rules with call-by-value operational semantics to prove the following more general extensionality principles of existential types:

$$(\eta\exists_{CBV}) \quad \mathtt{case}\, V\, \mathtt{of}\, \{\, (\alpha, x) \Rightarrow N\,[(\alpha, x)/z] \,\} = N\,[V/z] \quad (\text{if } V : \exists\alpha.B, z \notin FV(N))$$

$$\frac{\Theta, \alpha \,;\, \Gamma, y : B \vdash M\,[(\alpha, y)/x] = M'\,[(\alpha, y)/x] : A \quad x \notin FV(M) \cup FV(M')}{\Theta \,;\, \Gamma, x : \exists\alpha.B \vdash M = M' : A}\ \exists X_C$$

4.3.1 Parametricity

We saw how to justify that extensional equality is sound w.r.t. typed observational equivalence in Section 3.3.3. The trick was to use a logical relation, i.e. a relation

between terms that depends on types specifying the terms expected run-time behavior.

Let's just try to extend the interpretation of types to include the quantifiers $\forall$ and $\exists$ in the most obvious way. The first obstacle is how to handle type variables. We'll use the usual trick of threading an environment through the interpreter of types which maps type variables to term relations.

$$
\begin{aligned}
[\![\_]\!]\_ &: \mathit{Type} \to (\mathit{TypeVar} \to \mathrm{Rel}(\mathit{Term}, \mathit{Term})) \to \mathrm{Rel}(\mathit{Term}, \mathit{Term}) \\
M\ [\![\texttt{bool}]\!]_\tau\ M' &\iff M \mapsto\!\!\!\!\rightarrow_\beta \texttt{true} \twoheadleftarrow\!\!\!\!\mapsto_\beta M' \text{ or } M \mapsto\!\!\!\!\rightarrow_\beta \texttt{false} \twoheadleftarrow\!\!\!\!\mapsto_\beta M' \\
M\ [\![A{\to}B]\!]_\tau\ M' &\iff \forall N\ [\![A]\!]_\tau\ N'.\ (M\ N)\ [\![B]\!]_\tau\ (M'\ N') \\
M\ [\![\alpha]\!]_\tau\ M' &\iff M\ \tau(\alpha)\ M' \\
M\ [\![\forall\alpha.B]\!]_\tau\ M' &\iff \forall A \in \mathit{Type}.\ (M\ A)\ [\![B]\!]_{\tau,[\![A]\!]_\tau/\alpha}\ (M'\ A) \\
M\ [\![\exists\alpha.B]\!]_\tau\ M' &\iff M \mapsto\!\!\!\!\rightarrow_\beta (A, N) \text{ and} M' \mapsto\!\!\!\!\rightarrow_\beta (A', N') \text{ and} N\ [\![B]\!]_{\tau,[\![A?]\!]_\tau/\alpha}\ N'
\end{aligned}
$$

There's lots of problems with this definition! Worst of all, it is not a well-founded definition by induction like before. Previously, $[\![A]\!]$ was defined only in terms of smaller types found inside A, but now $[\![\forall\alpha.B]\!]$ and $[\![\forall\alpha.B]\!]$ depend on the meaning $[\![A]\!]$ of *all* other types, including much larger ones that have nothing to do with B. Besides that, the condition of $M\ [\![\exists\alpha.B]\!]_\tau\ M'$ seems bogus. Running both M and M' give two different packages, with two different implementation types A and A'. There's no reason to assume that A and A' are related in any standard way, so how do we compare the contents of the package?

It turns out that the interpretation of all types as relations share one key fact: they are *closed under expansion*. This was a key fact (Lemma 3.16) that is at the heart of the fundamental lemma (Lemma 3.19). It is true by definition for $[\![\texttt{bool}]\!]$, and $[\![A \to B]\!]$ inherits this property from $[\![B]\!]$. Therefore, we can circumscribe interpretation of all the types we know about now — and *all* the possible future types we might add to the language later — as term relations with this special property.

Definition 4.1 (Equivalence Candidate)**.** A *relation candidate* $\mathbb{A}$ is any binary term relation that is *closed under expansion*: if $M \mapsto\!\!\!\!\rightarrow_\beta M'\ \mathbb{A}\ N' \twoheadleftarrow\!\!\!\!\mapsto_\beta N$ then $M\ \mathbb{A}\ N$. Furthermore, an *equivalence candidate* is any relation candidate $\mathbb{A}$ that is also a partial equivalence relation (i.e. $\mathbb{A}$ is symmetric and transitive). The set of all relation candidates is written as RC and the set of all equivalence candidates is written as EC.

These are called *candidates* because some relations in RC or even EC may not correspond to any type in our actual language. These are only candidate relations that *might* correspond to some type. A good place to read more about circumscribing a "candidate" pool of possible type interpretations is Girard et al. [1989].

We can now fix up the problems in our definition to make it a real inductive definition by generalizing beyond the specific problem cases to just try *any* possible candidate from a candidate pool $C \subseteq \mathrm{Rel}(\mathit{Term}, \mathit{Term})$ — such as

$C = EC$ or $C = RC$ — that might potentially fit in that position.

$$\begin{aligned}
[\![_]\!]^{_}_{_} &: \mathit{Type} \to \wp(\mathrm{Rel}(\mathit{Term}, \mathit{Term})) \\
&\to (\mathit{TypeVar} \to \mathrm{Rel}(\mathit{Term}, \mathit{Term})) \\
&\to \mathrm{Rel}(\mathit{Term}, \mathit{Term})
\end{aligned}$$

$$\begin{aligned}
M\ [\![\texttt{bool}]\!]^C_\tau\ M' &\iff M \mapsto\!\!\!\!\to_\beta \texttt{true} \mathrel{\reflectbox{$\mapsto\!\!\!\!\to$}}_\beta M' \text{ or } M \mapsto\!\!\!\!\to_\beta \texttt{false} \mathrel{\reflectbox{$\mapsto\!\!\!\!\to$}}_\beta M' \\
M\ [\![A \to B]\!]^C_\tau\ M' &\iff \forall N\ [\![A]\!]^C_\tau\ N'.\ (M\ N)\ [\![B]\!]^C_\tau\ (M'\ N') \\
M\ [\![\forall\alpha.B]\!]^C_\tau\ M' &\iff \forall A, A' \in \mathit{Type}, \mathbb{A} \in C.\ (M\ A)\ [\![B]\!]^C_{\tau,\mathbb{A}/\alpha}\ (M'\ A') \\
M\ [\![\exists\alpha.B]\!]^C_\tau\ M' &\iff M \mapsto\!\!\!\!\to_\beta (A, N) \text{ and } M' \mapsto\!\!\!\!\to_\beta (A', N') \text{ and} \\
&\qquad \exists \mathbb{A} \in C.\ N\ [\![B]\!]^C_{\tau,\mathbb{A}/\alpha}\ N'
\end{aligned}$$

This is now well-defined... But does it really work?? Well, at least we should be sure that the interpretation of every type has the relation candidate properties.

$$\begin{aligned}
\tau \in [\![\Theta]\!]^C &\iff \forall \alpha \in \Theta.\ \tau(\alpha) \in C \\
[\![\Theta \vdash A : \star]\!]^C &\iff \forall \tau \in [\![\Theta]\!]^C.\ [\![A]\!]^C_\tau \in C
\end{aligned}$$

Lemma 4.1 (Fundamental Property of Types)**.** *If $\Theta \vdash A : \star$ is derivable then both $[\![\Theta \vdash A : \star]\!]^{RC}$ and $[\![\Theta \vdash A : \star]\!]^{EC}$ are true.*

In other words, if τ maps all of A's free type variables to relation candidates (they are closed under expansion) then $[\![A]\!]^{RC}_\tau$ is also a relation candidate (it is closed under expansion). Furthermore, if τ maps all of A's free type variables to equivalence candidates (they are also symmetric and transitive) then $[\![A]\!]^{EC}_\tau$ is also an equivalence candidate (it is also symmetric and transitive).

Proof. By induction on the derivation of $\Theta \vdash A : \star$. Left as an exercise to the reader. ■

Let's finish threading type variables through the rest of our interpretation. Since terms can have free type variables in them, syntactic substitutions can also plug in types for generic type variables in addition to plugging in terms for regular variables.

$$\mathit{Substitution} \ni \sigma ::= \bullet \mid M/x, \sigma \mid A/\alpha, \sigma$$

$$\begin{aligned}
\sigma\ [\![\Gamma]\!]^C_\tau\ \sigma' &\iff \forall (x : A) \in \Gamma.\ x\,[\sigma]\ \ [\![A]\!]^C_\tau\ x\,[\sigma'] \\
[\![\Theta\ ;\ \Gamma \vdash M : A]\!] &\iff \forall \tau \in [\![\Theta]\!]^{RC}, \sigma\ [\![\Gamma]\!]^{RC}_\tau\ \sigma'.\ M\,[\sigma]\ \ [\![A]\!]^{RC}_\tau\ M\,[\sigma'] \\
[\![\Theta\ ;\ \Gamma \vdash M = M' : A]\!] &\iff \forall \tau \in [\![\Theta]\!]^{EC}, \sigma\ [\![\Gamma]\!]^{EC}_\tau\ \sigma'.\ M\,[\sigma]\ \ [\![A]\!]^{EC}_\tau\ M'\,[\sigma']
\end{aligned}$$

Note that the definition of the $[\![\Gamma]\!]^C_\tau$ relation between substitutions is defined to be *as permissive as possible* while still respecting the explicit type assignments listed in Γ. That means term variables — and more importantly, generic type variables — that aren't assigned a type in Γ can be replaced by anything or nothing on either side without changing the $[\![\Gamma]\!]^C_\tau$ relationship.

Lemma 4.2. *a)* $\sigma\ [\![\Gamma]\!]^C_\tau\ \sigma'$ *iff* $(A/\alpha, \sigma)\ [\![\Gamma]\!]^C_\tau\ \sigma'$ *iff* $\sigma\ [\![\Gamma]\!]^C_\tau\ (B/\alpha, \sigma')$.

b) If x *is not assigned a type in* Γ*, then* $\sigma\ [\![\Gamma]\!]^C_\tau\ \sigma'$ *iff* $(M/x, \sigma)\ [\![\Gamma]\!]^C_\tau\ \sigma'$ *iff* $\sigma\ [\![\Gamma]\!]^C_\tau\ (N/x, \sigma')$.

Now the challenge is to prove the updated fundamental property. To deal with the substitution of types for generic type variables, we need to relate syntactic substitution $B\,[A/\alpha]$ used by the type system with the semantic substitution in τ that is carried out by the interpretation $[\![A]\!]^C_\tau$.

Lemma 4.3. *For all types* A *and* B, $[\![B\,[A/\alpha]]\!]^C_\tau = [\![B]\!]^C_{\tau, [\![A]\!]^C_\tau/\alpha}$.

Proof. By induction on the syntax of B. Left as an exercise to the reader. ■

As before, it can help to understand the meaning of each inference rule in isolation, before getting lost in a big inductive proof.

Lemma 4.4. *The logical interpretation of the simply-typed inference rules for type checking and equality still hold when generalized over generic type variables. Furthermore, the logical interpretation of these new inference rules also hold:*

a) $[\![\forall I]\!]$ *and* $[\![\forall E]\!]$

b) $\left[\!\left[\forall I^2\right]\!\right]$ *and* $\left[\!\left[\forall E^2\right]\!\right]$ *and* $[\![\forall X]\!]$.

c) $\left[\!\left[\exists I^2\right]\!\right]$ *and* $\left[\!\left[\exists E^2\right]\!\right]$ *and* $[\![\exists X]\!]$.

Proof. By the definition of $[\![\_]\!]$. Left as a challenge to the reader. ■

And some cases can even be eliminated via the "semi-typed" *Expand* rule.

Lemma 4.5. *The* $\forall I^2$ *and* $\exists E^2$ *rules can be derived from Expand,* $\forall X$ *and* $\exists X$*:*

$$\dfrac{\begin{array}{c}\Theta, \alpha\ ;\ \Gamma \vdash M = M' : B \\ \vdots\ \mathit{Expand}, \forall X \\ \Theta\ ;\ \Gamma \vdash \Lambda\alpha.M = \Lambda\alpha.M' : \forall\alpha.B\end{array}}{}$$

$$\dfrac{\begin{array}{c}\Theta\ ;\ \Gamma \vdash M = M' : \exists\alpha.B \quad \Theta, \alpha\ ;\ \Gamma, x : B \vdash N = N' : A \quad \alpha \notin FV(A) \\ \vdots\ \mathit{Expand}, \exists X \\ \Theta\ ;\ \Gamma \vdash \mathtt{case}\ M\ \mathtt{of}\ \{\,(\alpha, x{:}B) \Rightarrow N\,\} = \mathtt{case}\ M'\ \mathtt{of}\ \{\,(\alpha, x{:}B) \Rightarrow N'\,\} : A\end{array}}{}$$

Proof. Left as an exercise to the reader. ■

Lemma 4.6 (Fundamental Property of the Logical Relation)**.**

a) If $\Theta\ ;\ \Gamma \vdash M : A$ *is derivable then* $[\![\Theta\ ;\ \Gamma \vdash M : A]\!]$ *is true.*

b) If $\Theta\ ;\ \Gamma \vdash M = N : A$ *is derivable then* $[\![\Theta\ ;\ \Gamma \vdash M = N : A]\!]$ *is true.*

Proof. By induction on the derivation of $\Theta\ ;\ \Gamma \vdash M : A$ (for the first part) and $\Theta\ ;\ \Gamma \vdash M = N : A$ (for the second part). Left as a challenge to the reader. ■

Theorem 4.7 (Soundness)**.** *If* $\Theta\ ;\ \Gamma \vdash M = N : A$ *then* $\Theta\ ;\ \Gamma \vdash M \approx N : A$.

Proof. Analogous to Theorem 3.21. □

Chapter 5

Compilation & Abstract Machines

So far, we have been using a small-step operational semantics as the main tool to explain how a program can make its way to computing its answer. This is great for clearly specifying what that answer should be for *all* programs that may be written, and also as a building block to build other higher-level theories — like equational theories and logical relations — for understanding the properties of programs. But a straightforward interpretation of small-step operational semantics is a *terribly inefficient* to the point of being useless for any practical application.

What's the problem? There are two sources of wasted, redundant effort that get in the way of a real implementation. The first, most obvious problem is that substitution can duplicate work an arbitrary number of times. Every time we substitute, as in $\beta\!\rightarrow$ reduction, the result looks like:

$$(\lambda x.M)\ N \mapsto_\beta M\,[N/x]$$

How long does it take to calculate the right-hand side? Substitution has to dig through the entire syntax tree of M. So if M is program with a hundred thousand "lines" of code, then a single $M\,[N/x]$ will traverse the whole thing. If a program has side m and takes n $\beta\!\rightarrow$ steps, then the real cost of running this substitution model literally takes can take $O(mn)$ time (each $\beta\!\rightarrow$ step causes a substitution which incurs a search and replace over m "lines"). Oof.

The situation is actually even more dire. Substitution can actually make the program much *bigger*, not smaller, by coping a subterm a large number of times:

$$(\lambda x.(\ldots x \ldots x \ldots x \ldots))\ \mathit{big} \mapsto_\beta \ldots \mathit{big} \ldots \mathit{big} \ldots \mathit{big} \ldots$$

where *big* is some syntactically large term. So using substitution to calculate the answer of a program can be *worse* than $O(mn)$, where m is the size of the term and n the number of steps, because the m can *grow* as the program "reduces."

To see the second, consider what a literal interpretation of this rule really tells us to do:

$$\frac{R \mapsto_\beta N}{E[R] \mapsto_\beta E[N]}$$

where $R \mapsto_\beta N$ is an application of an actual reduction step directly to the top of the redex R. In the conclusion of the rule, the $E[R]$ is actually a complex form of pattern match, which starts looking at some arbitrary term M and digs through the syntax tree following a specific path described by the grammar of evaluation contexts until it finally discovers a reducible expression R hidden inside. Hypothetically, R might be very deep — e.g. hundreds of thousands of "lines" of code — from the top of the whole program. Then, the computer performs a single, tiny step $R \mapsto_\beta N$ to simplify just the top part of the redex. The reduct N then gets plugged back into the original evaluation context $E[N]$, which might mean "zipping" back up the whole tree (another hundred thousand lines) from the point of the reduction to the top of the program.

Abstract machines are an alternate form of operational semantics that makes lower-level concerns more explicit, and lets us address these concerns over cost. The first, most apparent, way that abstract machines differ from what we've seen so far is that they build in the search for the next redex directly into the individual small steps of the semantics itself.

5.1 Introducing Continuations

A call-by-name abstract machine:

$$\begin{aligned}
\langle M\ N \| K\rangle &\mapsto \langle M \| N \cdot K\rangle \\
\langle \texttt{if}\, M\, \texttt{then}\, N_1\, \texttt{else}\, N_2 \| K\rangle &\mapsto \langle M \| \texttt{if then}\, N_1\, \texttt{else}\, N_2; K\rangle \\
\langle \lambda x.M \| N \cdot K\rangle &\mapsto \langle M\,[N/x] \| K\rangle \\
\langle \texttt{true} \| \texttt{if then}\, N_1\, \texttt{else}\, N_2; K\rangle &\mapsto \langle N_1 \| K\rangle \\
\langle \texttt{false} \| \texttt{if then}\, N_1\, \texttt{else}\, N_2; K\rangle &\mapsto \langle N_2 \| K\rangle
\end{aligned}$$

Some rules build up *continuations* K while looking for the next step to perform. Other rules use that continuation together with some value to do a "real" step.

$$\mathit{Cont} \ni K ::= N \cdot K \mid \texttt{if then}\, N_1\, \texttt{else}\, N_2; K \mid \texttt{ret}$$

The final continuation (i.e. the bottom of the call stack) is written `ret` for "return," since at that point there is nothing more to do any value it receives is returned as the final result.

The machine keeps running until they reach a final state, in which a value (a constant c or some abstraction $\lambda x.M$) is returned.

Definition 5.1 (Final & Stuck)**.** *Final* statements have the form $\langle c \| \texttt{ret}\rangle$ or $\langle \lambda x.M \| \texttt{ret}\rangle$. A statement S is *stuck* if it is not final and there is no S' such that $S \mapsto S'$.

5.2 Intermezzo: Environments & Closures

To solve both inefficiencies of substitution — it traverses a huge sub-tree and it copies subterms that might make the result even bigger than the source program — we can use *environments* (L) that stores a map from local variables to what they stand for. Intuitively, the environment is a data structure that represents all the delayed substitution that we *should have* already performed, but have not finished yet. The environment gets carried around by the machine so that we essentially perform a big simultaneous substitution at the same time as searching for a redex and performing reduction steps.

$$\begin{aligned}
\langle M\ N \mid L \mid K\rangle &\mapsto \langle M \mid L \mid N\,\{L\} \cdot K\rangle \\
\langle \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \mid L \mid K\rangle &\mapsto \langle M \mid L \mid \texttt{if}\ \texttt{then}\ N_1\ \texttt{else}\ N_2\,\{L\}\,; K\rangle \\
\langle \lambda x.M \mid L \mid N\left\{L'\right\} \cdot K\rangle &\mapsto \langle M \mid x := N\left\{L'\right\}, L \mid K\rangle \\
\langle \texttt{true} \mid L \mid \texttt{if}\ \texttt{then}\ N_1\ \texttt{else}\ N_2\left\{L'\right\}; K\rangle &\mapsto \langle N_1 \mid L' \mid K\rangle \\
\langle \texttt{false} \mid L \mid \texttt{if}\ \texttt{then}\ N_1\ \texttt{else}\ N_2\left\{L'\right\}; K\rangle &\mapsto \langle N_2 \mid L' \mid K\rangle \\
\langle x \mid L \mid K\rangle &\mapsto \langle N \mid L' \mid K\rangle \qquad (\text{if } x := N\left\{L'\right\} \in L)
\end{aligned}$$

$$\begin{aligned}
\mathit{Cont} \ni K &::= N\,\{L\} \cdot K \mid \texttt{if}\ \texttt{then}\ N_1\ \texttt{else}\ N_2\,\{L\}\,; K \mid \texttt{ret} \\
\mathit{LocalEnv} \ni\ L &::= x_1 := N_1\,\{L_1\}, \ldots, x_n := N_n\,\{L_n\}
\end{aligned}$$

Note that some steps create *closures*, $N\,\{L\}$, which is a pair of an (open) term and its environment of local variable bindings. A good rule of thumb is that in the machine statement $\langle M \mid L \mid K\rangle$, the term M can refer to bindings in L, but K is "closed" and cannot access L. So whenever an unevaluated, open term moves outside of an L's bindings, then it needs to carry with it a snap-shot of that L to remember what variable bindings were active so they can be reinstated when that code might eventually be run.

You can check that closures are introduced and used correctly from the invariant that substituting the local environment *now* gives the same result as running the machine as-is.

Lemma 5.1.

a) If $\langle M \mid L \mid K\rangle \mapsto\!\!\!\!\rightarrow_\beta \langle V_1 \mid L_1 \mid \texttt{ret}\,\rangle$ *then* $\langle M\,[L] \mid \bullet \mid K\rangle \mapsto\!\!\!\!\rightarrow_\beta \langle V_2 \mid L_2 \mid \texttt{ret}\,\rangle$ *for some* $V_1 \sim V_2$.

b) If $\langle M\,[L] \mid \bullet \mid K\rangle \mapsto\!\!\!\!\rightarrow_\beta \langle V_1 \mid L_1 \mid \texttt{ret}\,\rangle$ *then* $\langle M \mid L \mid K\rangle \mapsto\!\!\!\!\rightarrow_\beta \langle V_2 \mid L_2 \mid \texttt{ret}\,\rangle$ *for some* $V_1 \sim V_2$.

Proof. Left as a challenge to the reader. ■

Exercise 5.1. Define an alternate machine with environments, like above, that follows the call-by-value operational semantics, rather than call-by-name.

5.3 Compilation

I have two gripes with the abstract machine(s) we've seen so far:

1. There is a lot of redundancy: e.g. I can apply M to N either as $\langle M\ N \| K\rangle$ or $\langle M \| N \cdot K\rangle$, and the two mean the same thing.
2. I could use the operational semantics on plain terms to derive all sorts of equational theories between open terms that are fragments of whole programs. I can only use the abstract machine to reason about a complete program; it is helpless to say anything interesting about individual program fragments.

Both of these problems can be fixed by compiling source code to an appropriate abstract machine code!

Abstract machine code:

$$\begin{aligned} \mathit{Term} \ni M, N &::= x \mid c \mid \lambda x.M \mid \mathtt{run}\, S \\ \mathit{Cont} \ni \quad K &::= N \cdot K \mid \mathtt{if\ then}\, S_1\, \mathtt{else}\, S_2 \mid \mathtt{ret} \\ \mathit{State} \ni \quad S &::= \langle M \| K\rangle \end{aligned}$$

Compilation from source functional programs to machine code:

$$\begin{aligned} [\![M\ N]\!] &= \mathtt{run}\ \langle M \| N \cdot \mathtt{ret}\,\rangle \\ [\![\mathtt{if}\, M\, \mathtt{then}\, N_1\, \mathtt{else}\, N_2]\!] &= \mathtt{run}\ \langle M \| \mathtt{if\ then}\ \langle N_1 \| \mathtt{ret}\,\rangle\, \mathtt{else}\ \langle N_2 \| \mathtt{ret}\,\rangle\rangle \\ [\![x]\!] &= x \\ [\![c]\!] &= c \\ [\![\lambda x.M]\!] &= \lambda x.[\![M]\!] \end{aligned}$$

Steps of the abstract machine:

$$\begin{aligned} (\beta\!\to) \qquad & \langle \lambda x.M \| N \cdot K\rangle \mapsto \langle M\,[N/x] \| K\rangle \\ (\beta\mathtt{bool}_1) \qquad & \langle \mathtt{true} \| \mathtt{if\ then}\, N_1\, \mathtt{else}\, N_2; K\rangle \mapsto \langle N_1 \| K\rangle \\ (\beta\mathtt{bool}_1) \qquad & \langle \mathtt{false} \| \mathtt{if\ then}\, N_1\, \mathtt{else}\, N_2; K\rangle \mapsto \langle N_2 \| K\rangle \\ (\beta\mathtt{run}) \qquad & \langle \mathtt{run}\, S \| K\rangle \mapsto S\,[K/\mathtt{ret}\,] \end{aligned}$$

Plugging a continuation K in for the final "return" pointer.

$$\begin{aligned} \langle M \| K'\rangle\,[K/\mathtt{ret}\,] &= \langle M \| K'\,[K/\mathtt{ret}\,]\rangle \\ \mathtt{ret}\,[K/\mathtt{ret}\,] &= K \\ (N \cdot K')\,[K/\mathtt{ret}\,] &= N \cdot (K'\,[K/\mathtt{ret}\,]) \\ (\mathtt{if\ then}\, S_1\, \mathtt{else}\, S_2)\,[K/\mathtt{ret}\,] &= \mathtt{if\ then}\, S_1\,[K/\mathtt{ret}\,]\, \mathtt{else}\, S_2\,[K/\mathtt{ret}\,] \end{aligned}$$

5.3.1 Intensional Machine Equality

Now we have a meaningful reduction theory with reduction in any context:

$$\frac{S \mapsto S'}{C[S] \to C[S']}\ \textit{Compat.}$$

General reduction ($\twoheadrightarrow$) is the reflexive, transitive closure of $\to$, and β-equivalence ($=_\beta$) is the reflexive, transitive, symmetric closure of $\to$.

In addition, we can add some extensionality to the equational theory. Here is a presentation in the form of η-axioms:

$$\begin{array}{rcll}
(\eta\mu) & \mu\tilde{x}.\langle M \| \tilde{x}\rangle & =_\eta M : A & (\text{if } \tilde{x} \notin FV(M)) \\
(\eta\to) & \lambda x.\mu\tilde{y}.\langle M \| x \cdot \tilde{y}\rangle & =_\eta M : A \to B & (\text{if } x, \tilde{y} \notin FV(M)) \\
(\eta\texttt{bool}) & \begin{array}{l}\texttt{if then } \langle \texttt{true} \| K\rangle \\ \quad \texttt{else } \langle \texttt{false} \| K\rangle\end{array} & =_\eta K : \texttt{bool} &
\end{array}$$

Note that the $\eta\mu$ axiom can apply to a term of *any* type, so it works just as well in an untyped setting.

Theorem 5.2 (Soundness)**.** *If $M =_\beta N$ then $[\![M]\!] =_\beta [\![N]\!]$ in the machine.*

Proof. By induction on the derivation of $M =_\beta N$. Left as an exercise to the reader. ■

Hint. The main thing to check is that translating both sides of each source rewriting rule (*a.k.a.* axiom) gives you two target expressions that you can equate using the target machine's rewriting rules.

To complete soundness, you also have to justify that other rules map from the source to the target: reflexivity, symmetry, transitivity, and compatibility. The first three are immediate, but compatibility can take a lot of work to show all the details (by induction on the context it introduces).

However, the compilation translation is defined in a nicely-organized way. Namely, it is *compositional*: the compilation of each syntax node is defined only in terms of the translation of its immediate sub-terms. In other words, you can infer from the given definition that there is *also* a translation of *arbitrary* contexts, $C \in \mathit{Context}$, such that $[\![C[M]]\!] = [\![C]\!][[\![M]\!]]$. (If this is not entirely obvious, try defining some of the cases for $[\![C]\!]$ and checking the compositionality property.) From this property, compatibility follows immediately without having to look at C in any more detail. (How come?)

Compilation is also *hygienic*, in the sense that C and $[\![C]\!]$ introduce *exactly* the same bound variables around the hole $\Box$. Together with compositionality, we can automatically prove another important lemma, about how substitution commutes with translation, that is necessary for showing soundness:

Lemma 5.3. *For any compositional and hygienic $[\![\_]\!]$ such that $[\![x]\!] = x$,* $[\![M]\!]\,[[\![N]\!]/x] =_\alpha [\![M\,[N/x]]\!]$

Proof. Left as a hint for the reader. Why should this always work if you only know that $[\![C[M]]\!] = [\![C]\!][[\![M]]\!]$, the variable bindings in scope around $\Box$ in C is the same as in $[\![C]\!]$, and that $[\![x]\!] = x$? ■

More detail about these two shortcuts for proving equational correspondence of compositional and hygienic can be found in [Downen and Ariola, 2014a].

Theorem 5.4 (Soundness & Completeness)**.** $[\![M]\!] =_\beta [\![N]\!]$ *in the machine iff* $M =_{\beta\kappa} N$*, where the* κ *reduction rules (*a.k.a. commuting conversions*) are:*

$$(\kappa\,\texttt{bool}\to) \qquad \begin{pmatrix}\texttt{if}\,M\\ \texttt{then}\,N_1\\ \texttt{else}\,N_2\end{pmatrix}\ N' \to \begin{array}{l}\texttt{if}\,M\\ \texttt{then}\,(N_1\ N')\\ \texttt{else}\,(N_2\ N')\end{array}$$

$$(\kappa\,\texttt{bool}\,\texttt{bool}) \qquad \begin{array}{l}\texttt{if}\,(\texttt{if}\,M\,\texttt{then}\,N_1\,\texttt{else}\,N_2)\\ \texttt{then}\,N_1'\\ \texttt{else}\,N_2'\end{array} \to \begin{array}{l}\texttt{if}\,M\\ \texttt{then}\,(\texttt{if}\,N_1\,\texttt{then}\,N_1'\,\texttt{else}\,N_2')\\ \texttt{else}\,(\texttt{if}\,N_2\,\texttt{then}\,N_1'\,\texttt{else}\,N_2')\end{array}$$

Proof. Left as a challenge to the reader. ■

Hint. It might be helpful to define a decompilation translation $[\![\_]\!]^{-1}$ from abstract machine terms, continuations, and statements back to the source language. If both forward and backward translations are sound, and if both round-trips are equal to the starting point (using rewriting in the equational theory), then you can derive completeness. How can you show this?

Hint. Even if you define the decompilation translation $[\![\_]\!]^{-1}$, you may quickly run into the problem in showing how the $\beta\,\texttt{run}$ reduction can be simulated in the source language. From the perspective of the source language, $\beta\,\texttt{run}$ will seemingly inline evaluation contexts into various places (which are written as the places invoking `ret` in the machine). This is captured by the one generic κ axiom that commutes evaluation contexts E with *tail contexts* T that point out *all* the places that a program returns from; in our case, this will include all the places that a program returns from chains of if-then-else.

$$\mathit{TailCxt} \ni T ::= \Box \mid \texttt{if}\,M\,\texttt{then}\,T\,\texttt{else}\,T'$$

$$(\kappa) \qquad E[T[M_1,\ldots,M_n]] \to T[E[M_1],\ldots,E[M_n]]$$

Note that one tail context L may have multiple different holes $\Box$ in it, hence filling an L may require more than one term. Furthermore, the result $T[E[M_1],\ldots,E[M_n]]$ will replicate an evaluation context E multiple different times (one for each $\Box$ in T). How do you derive this general κ rule from the specific cases $\kappa\,\texttt{bool}\to$ and $\kappa\,\texttt{bool}\,\texttt{bool}$ written in Theorem 5.4.

Exercise 5.2. Prove that β-reduction of compiled abstract machine expressions is confluent: if $S_1 \twoheadleftarrow_\beta S \twoheadrightarrow_\beta S_2$ then there is some S' such that $S_1 \twoheadrightarrow_\beta S' \twoheadleftarrow_\beta S_2$, and similarly for terms (M) and continuations (K).

Challenge 5.3. Prove that β-reduction of compiled abstract machine expressions enjoys the standardization property: if $S \twoheadrightarrow_\beta S_1$ for some final state S_1, then there is another final state S_2 such that $S \mapsto\!\!\!\!\twoheadrightarrow_\beta S_2 \twoheadrightarrow_\beta S_1$.

Challenge 5.4. Define a notion of untyped contextual equivalence ($S \approx S'$) for compiled abstract machine expressions, and prove that β-equivalence is sound with respect to that untyped contextual equivalence (if $S =_\beta S'$ then $S \approx S'$).

5.4 Machine Types

There are now three different typing judgements for the three different syntactic sorts:

$$\begin{aligned} \mathit{Judgement} ::= {} & (\Gamma \vdash M : A) \\ \mid {} & (\Gamma \mid K : A \vdash \mathtt{ret} : B) \\ \mid {} & S : (\Gamma \vdash \mathtt{ret} : B) \end{aligned}$$

Typing rules for the abstract machine:

$$\frac{\Gamma \vdash M : A \quad \Gamma \mid K : A \vdash \mathtt{ret} : B}{\langle M \| K \rangle : (\Gamma \vdash \mathtt{ret} : B)}\ \mathit{Cut}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ {\to}R \qquad \frac{\Gamma \vdash N : A \quad \Gamma \mid K : B \vdash \mathtt{ret} : B'}{\Gamma \mid N \cdot K : A \to B \vdash \mathtt{ret} : B'}\ {\to}L$$

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}}\ \mathtt{bool}R_1 \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}}\ \mathtt{bool}R_2$$

$$\frac{S_1 : (\Gamma \vdash \mathtt{ret} : B) \quad S_2 : (\Gamma \vdash \mathtt{ret} : B)}{\Gamma \mid \mathtt{if\ then}\ S_1\ \mathtt{else}\ S_2 : \mathtt{bool} \vdash \mathtt{ret} : B}\ \mathtt{bool}L$$

$$\frac{}{\Gamma, x : A \vdash x : A}\ \mathit{Var} \qquad \frac{}{\Gamma \mid \mathtt{ret} : A \vdash \mathtt{ret} : A}\ \mathit{Ret}$$

$$\frac{S : (\Gamma \vdash \mathtt{ret} : A)}{\Gamma \vdash \mathtt{run}\, S : A}\ \mathit{Run}$$

If you erase all the program bits (M, K, S), you get a form of logic called the *sequent calculus*. In particular, it is equivalent to the LJ system of intuitionistic logic by Gentzen [1935].

More down to earth, this type system lets us prove type safety directly on machine statements.

Theorem 5.5 (Type Preservation)**.** *If* $\Gamma \vdash M : A$ *then* $\Gamma \vdash [\![M]\!] : A$.

Proof. By induction on the derivation of $\Gamma \vdash M : A$. Left as an exercise to the reader. ∎

Lemma 5.6 (Typed Substitution)**.**

a) If $S : (\Gamma, x : A \vdash \texttt{ret} : B)$ *and* $\Gamma \vdash M : A$ *then* $S\,[M/x] : (\Gamma \vdash \texttt{ret} : B)$.

b) If $S : (\Gamma \vdash \texttt{ret} : A)$ *and* $\Gamma \mid K : A \vdash \texttt{ret} : B$ *then* $S\,[K/\texttt{ret}\,] : (\Gamma \vdash \texttt{ret} : B)$.

And similarly for substitution into terms and continuations.

Lemma 5.7 (Progress)**.** *If* $S : (\bullet \vdash \texttt{ret} : A)$ *then either* S *is final or* $S \mapsto S'$ *for some* S'.

Proof. By induction on the derivation of $S : (\bullet \vdash \texttt{ret} : A)$. Left as an exercise to the reader. ■

Lemma 5.8 (Preservation)**.** *If* $S : (\Gamma \vdash \texttt{ret} : A)$ *and* $S \mapsto S'$ *then* $S' : (\Gamma \vdash \texttt{ret} : A)$.

Proof. By induction on the derivation of $S : (\Gamma \vdash \texttt{ret} : A)$ and the reduction step $S \mapsto S'$. Left as an exercise to the reader. ■

5.4.1 Extensional Machine Equality

Since we now have typing information directly in the machine, we can now also specify a syntactic notion of extensionality on machine terms and continuations. For our two types, functions and booleans, we can define extensionality like so:

$$\frac{\langle M \| x \cdot \texttt{ret}\,\rangle = \langle M' \| x \cdot \texttt{ret}\,\rangle : (\Gamma, x : A \vdash \texttt{ret} : B) \quad x \notin FV(M) \cup FV(M')}{\Gamma \vdash M = M' : A \to B} \to X$$

$$\frac{\langle \texttt{true} \| K\rangle = \langle \texttt{true} \| K'\rangle : (\Gamma \vdash \texttt{ret} : B) \quad \langle \texttt{false} \| K\rangle = \langle \texttt{false} \| K'\rangle : (\Gamma \vdash \texttt{ret} : B)}{\Gamma \mid K = K' : \texttt{bool} \vdash \texttt{ret} : B} \texttt{bool}X$$

The $\to X$ rule says that two functions of type $A \to B$ are equal if they form equal statements for any generic call stack of the form $x \cdot \texttt{ret}$. The $\texttt{bool}X$ says that two continuations taking type `bool` are equal if they form equal statements when both are given `true` or both are given `false`.

The form of these rules suggest more generic forms that do no separate the sub-cases of `bool` or sub-structure of $A \to B$ like so:

$$\frac{\langle M \| \texttt{ret}\,\rangle = \langle M' \| \texttt{ret}\,\rangle : (\Gamma \vdash \texttt{ret} : A)}{\Gamma \vdash M = M' : A} XR$$

$$\frac{\langle x \| K\rangle = \langle x \| K'\rangle : (\Gamma, x : A \vdash \texttt{ret} : B) \quad F \notin FV(K) \cup FV(K')}{\Gamma \mid K = K' : A \vdash \texttt{ret} : B} XL$$

With this, we can also relate the typed, extensional equational theory of the source λ-calculus with the abstract machine.

Theorem 5.9 (Soundness)**.** *If* $\Gamma \vdash M = N : A$ *then* $\Gamma \vdash [\![M]\!] = [\![N]\!] : A$ *in the machine.*

Proof. By induction on the derivation of $\Gamma \vdash M = N : A$, using the fact that type checking in preserved (Theorem 5.5). Left as an exercise to the reader. ■

Theorem 5.10 (Completeness)**.** *If* $\Gamma \vdash [\![M]\!] = [\![N]\!] : A$ *in the machine then* $\Gamma \vdash M = N : A$.

Proof. Left as a challenge to the reader. ■

Hint. Consider using a similar strategy as Theorem 5.4, where instead you define a decompilation translation $[\![\_]\!]^{-1}$ that is sound and forms a round-trip (up to β-equality) with compilation $[\![\_]\!]$ in either direction. From there, completeness follows in the same way as before.

Hint. Note that we are missing the commuting conversions κ that were needed before in Theorem 5.4 to prove soundness of $\beta\,\texttt{ret}$. Can you somehow derive the κ rules from just β and extensionality?

Exercise 5.5. Add product ($A \times B$) and sum ($A + B$) types to the abstract machine, giving the syntax of terms and continuations, operational reduction rules, compilation, typing rules, and extensionality rules.

5.5 First-Class Control: Classical Logic

Why just one return pointer? Let's have lots of return pointers! In the program, they are represented as *continuation variables* (or just *covariables* for short), written with a tilde over them like $\tilde{x}, \tilde{y}, \tilde{z}$.

$$\begin{aligned}
\mathit{Variable} \ni\ & x, y, z ::= \dots \\
\mathit{CoVariable} \ni\ & \tilde{x}, \tilde{y}, \tilde{z} ::= \dots \\
\mathit{Term} \ni\ & M, N ::= x \mid \lambda(x \cdot \tilde{y}).S \mid \mu\tilde{x}.S \\
\mathit{Cont} \ni\ & K ::= \tilde{x} \mid N \cdot K \\
\mathit{State} \ni\ & S ::= \langle M \| K \rangle
\end{aligned}$$

This calculus is based on the $\lambda\mu\tilde{\mu}$-calculus by Curien and Herbelin [2000]. The main difference is that a function can now give a name ($\tilde{y}$) to its return pointer of the form $\lambda(x \cdot \tilde{y}).S$ [Johnson-Freyd et al., 2015]. The entire continuation can be bound to a covariable using $\mu\tilde{x}.S$.[1] This new syntax for functions is equivalent to the familiar form:

$$\begin{aligned}
\lambda x.M &\approx \lambda(x \cdot \tilde{y}).\, \langle M \| \tilde{y} \rangle \\
\lambda(x \cdot \tilde{y}).S &\approx \lambda x.\mu\tilde{y}.S
\end{aligned}$$

Covariables enjoy similar α-equivalence properties as regular variables.

$$\begin{array}{lcll}
(\alpha\mu) & \mu\tilde{x}.S =_\alpha \mu\tilde{y}.S\,[\tilde{y}/\tilde{x}] & & (\text{if } \tilde{y} \notin FV(S)) \\
(\alpha\to) & \lambda(x \cdot \tilde{y}).S =_\alpha \lambda(x' \cdot \tilde{y}').S\,[x'/x]\,[\tilde{y}'/\tilde{y}] & & (\text{if } x', \tilde{y}' \notin FV(S))
\end{array}$$

[1] Apologies for the notation clash with recursive types. The choice of μ for this binder comes from the $\lambda\mu$-calculus by Parigot [1992] and has nothing to do with recursion, but is instead about setting up labels and jumps for a natural deduction logic with multiple conclusions.

Compilation in the presence of multiple covariables:

$$\begin{aligned}[\![x]\!] &= x \\ [\![M\ N]\!] &= \mu\tilde{x}.\langle M \| N \cdot \tilde{x}\rangle \\ [\![\lambda x.M]\!] &= \lambda(x \cdot \tilde{y}).\langle [\![M]\!] \| \tilde{y}\rangle\end{aligned}$$

Machine reduction steps for functions and μ-abstractions:

$$\begin{aligned}(\beta\!\rightarrow) \qquad & \langle \lambda(x \cdot \tilde{y}).S \| N \cdot K\rangle \mapsto S\,[N/x, K/\tilde{y}] \\ (\beta\mu) \qquad & \langle \mu\tilde{x}.S \| K\rangle \mapsto S\,[K/\tilde{x}]\end{aligned}$$

Now judgements can have *multiple consequences* which corresponds to a choice of *multiple outputs*. Each named covariable in Δ represents a different output channel (i.e. sinks) that can receive a result, similar to the way that each named variable in Γ represents a different input channel (i.e. sources) that stand for incoming values.

$$\begin{aligned}\mathit{InEnvironment} \ni \Gamma &::= x_1 : A_1, \ldots, x_n : A_n \\ \mathit{OutEnvironment} \ni \Delta &::= \tilde{x}_1 : A_1, \ldots, \tilde{x}_n : A_n \\ \mathit{Judgement} &::= (\Gamma \vdash M : A \mid \Delta) \\ &\quad \mid (\Gamma \mid K : A \vdash \Delta) \\ &\quad \mid S : (\Gamma \vdash \Delta)\end{aligned}$$

The updated typing rules with multiple consequences:

$$\frac{\Gamma \vdash M : A \mid \Delta \quad \Gamma \mid K : A \vdash \Delta}{\langle M \| K\rangle : (\Gamma \vdash \Delta)}\ \mathit{Cut}$$

$$\frac{S : (\Gamma, x : A \vdash \tilde{y} : B, \Delta)}{\Gamma \vdash \lambda(x \cdot \tilde{y}).S : A \rightarrow B \mid \Delta}\ {\rightarrow}R \qquad \frac{\Gamma \vdash N : A \mid \Delta \quad \Gamma \mid K : B \vdash \Delta}{\Gamma \mid N \cdot K : A \rightarrow B \vdash \Delta}\ {\rightarrow}L$$

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Delta}\ \mathit{Var} \qquad \frac{}{\Gamma \mid \tilde{x} : A \vdash \tilde{x} : A, \Delta}\ \mathit{CoVar}$$

$$\frac{S : (\Gamma \vdash \tilde{x} : A, \Delta)}{\Gamma \vdash \mu\tilde{x}.S : A \mid \Delta}\ \mathit{Act}$$

This type system now corresponds to LK [Gentzen, 1935] by erasing all the program bits (M, K, S), which is a system of *classical logic*. That means the addition of multiple consequences to the right — along with a μ-abstraction that lets us assign a name a term's output channel so it can be used as many times (zero or more) as you want — can let us write *new* programs that inhabit types that used to be empty.

Exercise 5.6. Try writing terms (and their typing derivations) of these types for an arbitrary unknown A and B:

1. $A \to ((A \to B) \to B)$
2. $((A \to B) \to A) \to A$

Exercise 5.7. Use the product and sum types from Exercise 5.5 to write terms (and their typing derivations) of these types for an arbitrary unknown A_1, A_2, and B:

1. $((A_1 \to B) \times (A_2 \to B)) \to ((A_1 + A_2) \to B)$
2. $((A_1 + A_2) \to B) \to ((A_1 \to B) \times (A_2 \to B))$
3. $((A_1 \to B) + (A_2 \to B)) \to ((A_1 \times A_2) \to B)$
4. $((A_1 \times A_2) \to B) \to ((A_1 \to B) + (A_2 \to B))$
5. $A + (A \to B)$

Challenge 5.8. The extension from a single `ret` to multiple covariables $\tilde{x}$ is a *conservative extension*, in the sense that any typing derivation, operational step, or $\beta\eta$ equality that was valid before is still valid after. That means Theorems 5.2, 5.5 and 5.9, *etc.,* still hold when the λ-calculus is compiled to the extended machine.

However, the other direction of completeness, Theorems 5.4 and 5.10, is not so easy. We can now write *strictly more* programs in the machine language compared to the source λ-calculus language. What would you need to add to the λ-calculus to translate back *all* multi-covariable machine code to the source? Can you extend the λ-calculus with this new feature, and show how to complete completeness with classical machines?

5.6 Call-by-Value is Dual to Call-By-Name

Abstract machines have an inherent symmetry between producers (terms M returning results) and consumers (continuations K taking results). Variables are exactly dual to covariables. To complete the symmetry, we need to have a mirror image of functions and call stacks, as well as μ-abstractions. Lets just add some syntax that copies things from one side to the other without thinking too hard.

$$\begin{aligned}
\mathit{Variable} \ni x, y, z &::= \dots \\
\mathit{CoVariable} \ni \tilde{x}, \tilde{y}, \tilde{z} &::= \dots \\
\mathit{Term} \ni M, N &::= x \mid \mu\tilde{x}.S \mid \lambda(x \cdot \tilde{y}).S \mid K \cdot N \\
\mathit{Cont} \ni K &::= \tilde{x} \mid \tilde{\mu}x.S \mid N \cdot K \mid \tilde{\lambda}(\tilde{y} \cdot x).S \\
\mathit{State} \ni S &::= \langle M \| K \rangle
\end{aligned}$$

As a form of syntactic salt (some explicit, perhaps annoying, annotations to make things more explicit or obvious), the dual of something we already have is going to have a tilde on top of it, like $\tilde{\mu}$ is the dual of the μ we already

had. Traditionally, the opposite of functions (written as a $\tilde{\lambda}$ continuation that consumes a pair $K \cdot N$ of a term and continuation) is called a *subtraction* type, written $A - B$.

Generalized semantics [Downen and Ariola, 2014b]: defined in terms of values (V) and covalues (E) corresponding to evaluation contexts ... but don't worry about how they are defined yet.

$$\begin{aligned} \mathit{Value} \ni V, W &::= \dots \\ \mathit{CoValue} \ni E, F &::= \dots \end{aligned}$$

$$\begin{array}{rc} (\beta\tilde{\mu}) & \langle V \| \tilde{\mu} x.S \rangle \mapsto S\,[V/x] \\ (\beta\mu) & \langle \mu\tilde{x}.S \| E \rangle \mapsto S\,[E/\tilde{x}] \\ (\beta{\to}) & \langle \lambda(x \cdot \tilde{y}).S \| V \cdot E \rangle \mapsto S\,[V/x, E/\tilde{y}] \\ (\beta{-}) & \left\langle E \cdot V \middle\| \tilde{\lambda}(k \cdot x).S \right\rangle \mapsto S\,[V/x, E/\tilde{y}] \end{array}$$

The new rules involving $\tilde{\mu}$ and $\tilde{\lambda}$ are just mirror images of μ and λ.

For call-by-value semantics, define

$$\begin{aligned} \mathit{Value} \ni V, W &::= x \mid \lambda(x \cdot \tilde{y}).S \mid E \cdot V \\ \mathit{CoValue} \ni E, F &::= K \end{aligned}$$

For call-by-value semantics, define

$$\begin{aligned} \mathit{Value} \ni V, W &::= M \\ \mathit{CoValue} \ni E, F &::= k \mid V \cdot E \mid \tilde{\lambda}(\tilde{y} \cdot x).S \end{aligned}$$

Challenge 5.9. Add a new "subtraction" type $A - B$ with left and right inference rules to the type system to type-check the new program constructs $\tilde{\lambda}(\tilde{y} \cdot x).S$ and $K \cdot M$, and prove Progress and Preservation for them. The intuition is that a pair $K \cdot M$ has type $A - B$ when the term M has type A and the continuation expects a B (i.e. it has an A but is lacking a B). That means the matching abstraction $\tilde{\lambda}(\tilde{y} \cdot x).S$ expects an $A - B$ when its body can run under the scope of the variable $x : A$ and covariable $\tilde{y} : B$.

We also have some η-axioms that work for *any* evaluation strategy:

$$\begin{array}{rcl} (\eta\mu) & \mu\tilde{x}.\langle M \| \tilde{x} \rangle =_\eta M : A & (\text{if } \tilde{x} \notin FV(M)) \\ (\eta\tilde{\mu}) & \tilde{\mu}x.\langle x \| K \rangle =_\eta K : A & (\text{if } x \notin FV(K)) \\ (\eta{\to}) & \lambda(x \cdot \tilde{y}).\langle z \| x \cdot \tilde{y} \rangle =_\eta z : A \to B & \\ (\eta{-}) & \tilde{\lambda}(\tilde{y} \cdot x).\langle \tilde{y} \cdot x \| \tilde{z} \rangle =_\eta \tilde{z} : A - B & \end{array}$$

Duality — swapping machine states reverses the flow of information between

producers and consumers:

$$\langle M \| K \rangle^\perp = \langle K^\perp \| M^\perp \rangle$$

$$x^\perp = \tilde{x} \qquad \tilde{x}^\perp = x$$

$$(\mu\tilde{x}.S)^\perp = \tilde{\mu}x.S^\perp \qquad (\tilde{\mu}x.S)^\perp = \mu\tilde{x}.S^\perp$$

$$(\lambda(x \cdot \tilde{y}).S)^\perp = \tilde{\lambda}(\tilde{x} \cdot y).S^\perp \quad (\tilde{\lambda}(\tilde{x} \cdot y).S)^\perp = \lambda(x \cdot \tilde{y}).S^\perp$$

$$(K \cdot M)^\perp = K^\perp \cdot M^\perp \qquad (M \cdot K)^\perp = M^\perp \cdot K^\perp$$

This mirroring of producers and consumers formally expresses the fact that call-by-value is dual to call-by-name [Curien and Herbelin, 2000, Wadler, 2003].

Theorem 5.11 (Involutive Duality)**.** *a)* $S^{\perp\perp} = S$

b) $M^{\perp\perp} = M$

c) $K^{\perp\perp} = K$

Proof. By induction on the structure of syntax. Left as an exercise to the reader. ■

Theorem 5.12 (Operational Duality)**.** $S \mapsto_\beta S'$ *under the call-by-value semantics if and only if* $S^\perp \mapsto_\beta S'^\perp$ *under the call-by-name semantics.*

Proof. By induction on the given reduction step $\mapsto_\beta$. Left as an exercise to the reader. ■

Challenge 5.10. Extend the definition of duality to the type level ($A^\perp$) using your typing rules for subtraction types from Challenge 5.9 to prove the following theorem:

Theorem 5.13 (Type Duality). *a) If* $S : (\Gamma \vdash \Delta)$ *then* $S^\perp : (\Delta^\perp \vdash \Gamma^\perp)$.

b) If $\Gamma \vdash M : A \mid \Delta$ *then* $\Delta^\perp \mid M^\perp : A^\perp \vdash \Gamma^\perp$

c) If $\Gamma \mid K : A \vdash \Delta$ *then* $\Delta^\perp \vdash K^\perp : A^\perp \mid \Gamma^\perp$

where $\Gamma^\perp$ and $\Delta^\perp$ are defined point-wise:

$$(x_1 : A_1, \ldots, x_n : A_n)^\perp = \tilde{x}_1 : A_1^\perp, \ldots, \tilde{x}_n : A_n^\perp$$

$$(\tilde{x}_1 : A_1, \ldots, \tilde{x}_n : A_n)^\perp = x_1 : A_1^\perp, \ldots, x_n : A_n^\perp$$

Expressing logical duality as a computational duality is useful for all sorts of applications [Downen and Ariola, 2021]! Examples include induction and coinduction [Downen et al., 2015], compiling programs [Downen et al., 2016, Downen and Ariola, 2019], and abstracting over a large class of logical relations using the "bi-orthogonality" (*a.k.a.* $\top\top$-closure) technique [Downen et al., 2020] that uses subtyping to describe (co)recursive programs [Downen and Ariola, 2023] and intersection and union types [Downen et al., 2019]. To learn more about the bi-orthogonal types, check the previous OPLSS 2022 lecture series[2] and the accompanying notes [Downen, 2022, 2014].

[2] `https://www.cs.uoregon.edu/research/summerschool/summer22/topics.php#Downen`

Appendix A

Encodings

A.1 Untyped Encodings

A.1.1 Booleans

$$IfThenElse = \lambda x.\lambda t.\lambda f.x\ t\ f$$

$$True = \lambda t.\lambda f.t$$
$$False = \lambda t.\lambda f.f$$

$$And = \lambda x.\lambda y.IfThenElse\ x\ y\ False$$
$$Or = \lambda x.\lambda y.IfThenElse\ x\ True\ y$$
$$Not = \lambda x.IfThenElse\ x\ False\ True$$

A.1.2 Sums

$$Case = \lambda i.\lambda l.\lambda r.i\ l\ r$$

$$Inl = \lambda x.\lambda l.\lambda r.l\ x$$
$$Inr = \lambda x.\lambda l.\lambda r.r\ x$$

A.1.3 Products

$$Pair = \lambda x.\lambda y.\lambda p.p\ x\ y$$

$$Fst = \lambda x.\lambda y.x$$
$$Snd = \lambda x.\lambda y.y$$

A.1.4 Numbers

$$Iter = \lambda n.\lambda z.\lambda s.n\ z\ s$$

$$Zero = \lambda z.\lambda s.z$$
$$Suc = \lambda n.\lambda z.\lambda s.s\ (n\ z\ s)$$

$$\begin{aligned} One &= Suc\ Zero &&=_\beta \lambda z.\lambda s.s\ z \\ Two &= Suc\ One &&=_\beta \lambda z.\lambda s.s\ (s\ z) \\ Three &= Suc\ Two &&=_\beta \lambda z.\lambda s.s\ (s\ (s\ z)) \\ Four &= Suc\ Three &&=_\beta \lambda z.\lambda s.s\ (s\ (s\ (s\ z))) \\ Five &= Suc\ Four &&=_\beta \lambda z.\lambda s.s\ (s\ (s\ (s\ (s\ z)))) \end{aligned}$$

A.1.5 Lists

$$Fold = \lambda l.\lambda n.\lambda c.l\ n\ c$$

$$Nil = \lambda n.\lambda c.n$$
$$Cons = \lambda x.\lambda l.\lambda n.\lambda c.c\ x\ (l\ n\ c)$$

A.2 Typed Encodings

A.2.1 Booleans

$$Bool = \forall\delta.\delta \to \delta \to \delta$$

$$IfThenElse : \forall\delta.Bool \to \delta \to \delta \to \delta$$
$$IfThenElse = \Lambda\delta.\lambda x{:}Bool.\lambda t{:}\delta.\lambda f{:}\delta.x\ \delta\ t\ f$$

$$
\begin{aligned}
True, False &: Bool \\
True &= \Lambda\delta.\lambda t{:}\delta.\lambda f{:}\delta.t \\
False &= \Lambda\delta.\lambda t{:}\delta.\lambda f{:}\delta.f
\end{aligned}
$$

$$
\begin{aligned}
And, Or &: Bool \to Bool \to Bool \\
And &= \lambda x{:}Bool.\lambda y{:}Bool.IfThenElse\ Bool\ x\ y\ False \\
Or &= \lambda x{:}Bool.\lambda y{:}Bool.IfThenElse\ Bool\ x\ True\ y
\end{aligned}
$$

$$
\begin{aligned}
Not &: Bool \to Bool \\
Not &= \lambda x{:}Bool.IfThenElse\ Bool\ x\ False\ True
\end{aligned}
$$

A.2.2 Sums

$$Sum\ A\ B = \forall\delta.(A \to \delta) \to (B \to \delta) \to \delta$$

$$
\begin{aligned}
Case &: \forall\alpha.\forall\beta.\forall\delta.Sum\ \alpha\ \beta \to (\alpha \to \delta) \to (\beta \to \delta) \to \delta \\
Case &= \Lambda\alpha.\Lambda\beta.\Lambda\delta.\lambda i{:}Sum\ \alpha\ \beta.\lambda l{:}\alpha \to \delta.\lambda r{:}\beta \to \delta.i\ \delta\ l\ r
\end{aligned}
$$

$$
\begin{aligned}
Inl &: \forall\alpha.\forall\beta.\alpha \to Sum\ \alpha\ \beta \\
Inl &= \Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\Lambda\delta.\lambda l{:}\alpha \to \delta.\lambda r{:}\beta \to \delta.l\ x \\
Inr &: \forall\alpha.\forall\beta.\beta \to Sum\ \alpha\ \beta \\
Inr &= \Lambda\alpha.\Lambda\beta.\lambda x{:}\beta.\Lambda\delta.\lambda l{:}\alpha \to \delta.\lambda r{:}\beta \to \delta.r\ x
\end{aligned}
$$

A.2.3 Products

$$Prod\ A\ B = \forall\delta.(A \to B \to \delta) \to \delta$$

$$
\begin{aligned}
Pair &: \forall\alpha.\forall\beta.\alpha \to \beta \to Prod\ \alpha\ \beta \\
Pair &= \Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.\Lambda\delta.\lambda p{:}\alpha \to \beta \to \delta.p\ x\ y
\end{aligned}
$$

$$
\begin{aligned}
Fst &: \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha \\
Fst &= \lambda x{:}\alpha.\lambda y{:}\beta.x \\
Snd &: \forall\alpha.\forall\beta.\alpha \to \beta \to \beta \\
Snd &= \lambda x{:}\alpha.\lambda y{:}\beta.y
\end{aligned}
$$

A.2.4 Existentials

To properly encode existential types with universal types, we should use type functions which, which is like pushing a second level of λ-calculus (λs and applications) into types:

$$\begin{aligned} \mathit{Type} \ni A, B &::= \ldots \mid \lambda\alpha.B \mid A\ B \\ \mathit{Kind} \ni \quad k &::= \star \mid k \to k' \end{aligned}$$

Note that this means we now have *other* kinds of "types" that do different things that types did before. There are the old $\star$ kind of types that classify terms, but also function ($k \to k'$) kinds of types. For example, a type $A : \star \to \star$ does not classify a term, instead it transforms one $\star$ into another $\star$, so that $A\ B : \star$ (when $A : \star$) can classify terms but not just τ. This is the difference between *List Bool* (a list of booleans) versus just *List* (the list type constructor).

Because there are different kinds of types serving different roles (like different kinds of terms serving different roles), the well-formedness rules for types are more serious, and look like "type-checking the types." For type functions, we have the inference rules:

$$\frac{\Gamma, \alpha : k \vdash B : k'}{\Gamma \vdash \lambda\alpha.B : k \to k'} \to I^2 \qquad \frac{\Gamma \vdash A : k \to k' \quad \Gamma \vdash B : k}{\Gamma \vdash A\ B : k'} \to E^2$$

The addition of type-level λs and applications makes deciding the equality of to types (for the purpose of type-checking and unification) tricky. When we just had simple types built from $\to$, $\times$, and $+$, type equality was strict syntactic equality. When we added type variable binders like $\mu\alpha.\tau$, $\forall\alpha.B$ and $\exists\alpha.B$ then type equality incorporates α-equivalence which is easily decidable. When $\lambda\alpha.B$ and $A\ B$ then type equality should incorporate at least β-equivalence (and possibly η-equivalence) which requires more care and effort than just α.

Because there are now several kinds of types, it makes sense to annotate the bound type variables (introduced by the $\forall$s) with their kind as in the following encoding of existential types.

$$\mathit{Exists}\ \alpha{:}\star.\tau = \forall\delta{:}\star.(\forall\alpha{:}\star.\tau \to \delta) \to \delta$$

$$\begin{aligned} &\mathit{Open} : \forall\phi{:}\star \to \star.\forall\delta{:}\star.(\mathit{Exists}\ \alpha{:}\star.\phi\ \delta) \to (\forall\alpha{:}\star.\phi\ \alpha \to \delta) \to \delta \\ &\mathit{Open} = \Lambda\phi{:}\star \to \star.\Lambda\delta{:}\star.\lambda p{:}\mathit{Exists}\ \alpha : \star.\phi\ \alpha.\lambda f{:}\forall\alpha : \star.\phi\ \alpha \to \delta.p\ \delta\ f \end{aligned}$$

$$\begin{aligned} &\mathit{Pack} : \forall\phi{:}\star \to \star.\forall\alpha{:}\star.\phi\ \alpha \to \mathit{Exists}\ \alpha{:}\star.\phi\ \alpha \\ &\mathit{Pack} = \Lambda\phi{:}\star \to \star.\Lambda\alpha{:}\star.\lambda y{:}\phi\ \alpha.\Lambda\delta{:}\star.\lambda f{:}\forall\alpha{:}\star.\phi\ \alpha \to \delta.f\ \alpha\ y \end{aligned}$$

A.2.5 Numbers

$$Nat = \forall\delta.\delta \to (\delta \to \delta) \to \delta$$

$$\begin{aligned} &Iter : \forall\delta.Nat \to \delta \to (\delta \to \delta) \to \delta \\ &Iter = \lambda n{:}Nat.\Lambda\delta.\lambda z{:}\delta.\lambda s{:}\delta \to \delta.n\ \delta\ z\ s \end{aligned}$$

$$\begin{aligned} &Zero : Nat \\ &Zero = \Lambda\delta.\lambda z{:}\delta.\lambda s{:}\delta \to \delta.z \\ &Suc : Nat \to Nat \\ &Suc = \lambda n{:}Nat.\Lambda\delta.\lambda z{:}\delta.\lambda s{:}\delta \to \delta.s\ (n\ \delta\ z\ s) \end{aligned}$$

A.2.6 Lists

$$List\ A = \forall\delta.\delta \to (A \to \delta \to \delta) \to \delta$$

$$\begin{aligned} &Fold : \forall\alpha.\forall\delta.List\ \alpha \to \delta \to (\alpha \to \delta \to \delta) \to \delta \\ &Fold = \Lambda\alpha.\Lambda\delta.\lambda l{:}List\ \alpha.\lambda n{:}\delta.\lambda c{:}\alpha \to \delta \to \delta.l\ \delta\ n\ c \end{aligned}$$

$$\begin{aligned} &Nil : \forall\alpha.List\ \alpha \\ &Nil = \Lambda\alpha.\Lambda\delta.\lambda n{:}\delta.\lambda c{:}\alpha \to \delta \to \delta.n \\ &Cons : \forall\alpha.\alpha \to List\ \alpha \to List\ \alpha \\ &Cons = \Lambda\alpha.\lambda x{:}\alpha.\lambda l{:}List\ \alpha.\Lambda\delta.\lambda n{:}\delta.\lambda c{:}\alpha \to \delta \to \delta.c\ x\ (l\ \delta\ n\ c) \end{aligned}$$

A.3 Intermezzo: Russel's Paradox

An implementation of sets where

$$M \in N = M\ N$$

Example A.1.

$$\begin{aligned} M \cup N &= \lambda x.or\ (M\ x)\ (N\ x) \\ M \cap N &= \lambda x.and\ (M\ x)\ (N\ x) \end{aligned}$$

Russel's set

$$R = \{M \mid M \notin M\}$$

is then written as

$$R = \lambda x.not\ (x\ x)$$

Is Russel's set in Russel's set? $R \in R$?

$$\begin{aligned} R\ R &\mapsto (not\ (x\ x))[R/x] = not\ (R\ R) \\ &\mapsto not\ (not\ (R\ R)) \mapsto \dots \\ &\mapsto not\ (not\ (not\ \dots)) \end{aligned}$$

A.4 Untyped λ-Calculus: Recursion

Perhaps surprisingly, not every λ-calculus term reaches an answer.

Example A.2.

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$

Notice that

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x) \mapsto (x\ x)[(\lambda x.x\ x)/x] = (\lambda x.x\ x)\ (\lambda x.x\ x) = \Omega$$

That means

$$\Omega \mapsto \Omega \mapsto \Omega \mapsto \dots$$

In other words, some terms of the λ-calculus will never reach an answer; sometimes you might spin forever without getting any closer to a result.

Ω isn't very useful; a reduction step just regenerates the same Ω again. But what happens if we have something like Ω that changes a bit every step.[1]

$$Y\ f = (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

Now what happens when $Y\ f$ takes a step?

$$\begin{aligned} Y\ f &= (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) \\ &\mapsto (f\ (x\ x))[(\lambda x.f\ (x\ x))/x] \\ &= f\ ((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))) \\ &= f\ (Y\ f) \end{aligned}$$

That means

$$Y\ f =_\beta f\ (Y\ f)$$

[1] Notice that $\Omega =_\beta Y\ (\lambda x.x)$ and the encoding of Russel's set is $R =_\beta Y\ not$.

In other words, $Y\ f$ is a *fixed point* of the function f.

Why is a the fixed-point generator Y useful? Because it can be used to implement *recursion*, even though there is no recursion in the λ-calculus to begin with. For example, this recursive definition of multiplication

$$times = \lambda x.\lambda y.\, \texttt{if}\ x \leq 0\ \texttt{then}\ 0\ \texttt{else}\ y + (times\ (x-1)\ y)$$

can instead be written non-recursively by using Y like so:

$$\begin{aligned} timesish &= \lambda next.\lambda x.\lambda y.\, \texttt{if}\ x \leq 0\ \texttt{then}\ 0\ \texttt{else}\ y + (next\ (x-1)\ y) \\ times &= Y\ timesish \end{aligned}$$

Now check that *times* does the same thing as the recursive definition above:

$$\begin{aligned} times\ 0\ y &= Y\ timesish\ 0\ y \\ &\mapsto timesish\ (Y\ timesish)\ 0\ y \\ &\mapsto\!\!\!\!\mapsto \texttt{if}\ 0 \leq 0\ \texttt{then}\ 0\ \texttt{else}\ y + (Y\ timesish\ (0-1)\ y) \\ &\mapsto 0 \\ times\ (x+1)\ y &= Y\ timesish\ (x+1)\ y \\ &\mapsto timesish\ (Y\ timesish)\ (x+1)\ y \\ &\mapsto\!\!\!\!\mapsto \texttt{if}\ (x+1) \leq 0\ \texttt{then}\ 0\ \texttt{else}\ y + (Y\ timesish\ (x+1-1)\ y) \\ &\mapsto\!\!\!\!\mapsto y + (Y\ timesish\ (x+1-1)\ y) \\ &\twoheadrightarrow y + (times\ x\ y) \end{aligned}$$

The fact that the untyped λ-calculus can express the Y combinator as-is — just using higher-order functions and nothing else — is the main ingredient that makes the λ-calculus *Turing complete*. Furthermore, it is the fundamental insight behind the *Church-Turing thesis*: not only are Turing machines and λ-calculus equivalent, but they can encode *every* computable function.[2]

[2]The other insight is about the other direction — implementing λ-calculus in a Turing machine — and essentially boils down to the fact that program source code can be written as a sequence of finite characters (e.g. ASCII byte strings) and interpreters for those encoded programs can be written on conventional computer processors (e.g. `python` exists).

Appendix B

Free Theorems

B.1 Logical Operators

As useful shorthand, define some "logical" operations on binary term relations from a candidate pool C (e.g. where $C = RC$ or $C = EC$).

$$
\begin{aligned}
(\_ \Rightarrow^C \_) &: C \times C \to C \\
M\ (\mathbb{A} \Rightarrow \mathbb{B})\ M' &\iff \forall N\ \mathbb{A}\ N'.\ (M\ N)\ \mathbb{B}\ (M'\ N') \\
\forall^C &: (C \to C) \to C \\
M\ \forall^C(\mathbb{F})\ M' &\iff \forall A, A' \in \mathit{Type}, \mathbb{A} \in C.\ (M\ A)\ \mathbb{F}(\mathbb{A})\ (M'\ A') \\
\exists^C &: (C \to C) \to C \\
M\ \exists(\mathbb{F})\ M' &\iff M \mapsto\!\!\!\!\rightarrow_\beta (A, N) \text{ and } M' \mapsto\!\!\!\!\rightarrow_\beta (A', N') \text{ and} \\
&\qquad \exists \mathbb{A} \in RC.\ N\ \mathbb{F}(\mathbb{A})\ N'
\end{aligned}
$$

We omit the candidate pool C when it is clear from context. And notice that these operators are the essential meaning of the reducibility interpretation of types:

$$
\begin{aligned}
[\![\alpha]\!]^C_\tau &= \tau(\alpha) \\
[\![A \to B]\!]^C_\tau &= [\![A]\!]^C_\tau \Rightarrow^C [\![B]\!]^C_\tau \\
[\![\forall\alpha.B]\!]^C_\tau &= \forall^C(\lambda\mathbb{A} : C.\ [\![B]\!]^C_{\tau,\mathbb{A}/\alpha}) \\
[\![\exists\alpha.B]\!]^C_\tau &= \exists^C(\lambda\mathbb{A} : C.\ [\![B]\!]^C_{\tau,\mathbb{A}/\alpha})
\end{aligned}
$$

B.2 Polymorphic Absurdity (Void Type)

Theorem B.1. *There is no term M such that $\bullet\ ;\ \bullet \vdash M : \forall\alpha.\alpha$ is derivable.*

Proof. Suppose that we had some M and a derivation of $\bullet\ ;\ \bullet \vdash M : \forall\alpha.\alpha$. From Lemma 4.6, we would know that $M\ [\![\forall\alpha.\alpha]\!]^{RC}\ M$, i.e. $M\ \forall^{RC}(\lambda\mathbb{A}.\ \mathbb{A})\ M$. In

other words, for any types A, A' and reducibility candidate $\mathbb{A} \in RC$, it must be that $(M\ A)\ \mathbb{A}\ (M'\ A')$. So let's choose $A = A' = \alpha$ and the empty relation candidate

$$M\ \mathbb{A}\ M' \iff \text{never}$$

which is vacuously closed under expansion. It follows that $(M\ \alpha)\ \mathbb{A}\ (M\ \alpha)$ which is a contradiction. Therefore, there is no such $\bullet\ ;\ \bullet \vdash M : \forall\alpha.\alpha$. □

B.3 Polymorphic Identity (Unit Type)

Theorem B.2. *If* $\bullet\ ;\ \bullet \vdash M : \forall\alpha.\alpha \to \alpha$ *then* $\bullet\ ;\ \bullet \vdash M = \Lambda\alpha.\lambda x{:}\alpha.x : \forall\alpha.\alpha \to \alpha$.

Proof. Lemma 4.6 ensures $M\ [\![\forall\alpha.\alpha \to \alpha]\!]^{RC}\ M$, i.e. $M\ \forall^{RC}(\lambda\mathbb{A}.\ \mathbb{A} \Rightarrow \mathbb{A})\ M$. In other words, for any types A, A', relation candidate $\mathbb{A} \in CR$, and related terms $N\ \mathbb{A}\ N'$, it must be that $M\ A\ N\ \mathbb{A}\ M\ A'\ N'$. So let's choose the types $A = A' = \alpha$ and relation candidate

$$M\ \mathbb{A}\ M' \iff M \mapsto\!\!\!\!\rightarrow_\beta x \twoheadleftarrow\!\!\!\!\mapsto_\beta M'$$

so the variable x is related to itself by $\mathbb{A}$ by definition ($x\ \mathbb{A}\ x$ holds by reflexivity). It follows that $M\ \alpha\ (\mathbb{A} \Rightarrow \mathbb{A})\ M\ \alpha$ and thus $M\ \alpha\ x\ \mathbb{A}\ M\ \alpha\ x$ as well, meaning $M\ \alpha\ x \mapsto\!\!\!\!\rightarrow_\beta x$. Therefore,

$$M =_\eta \Lambda\alpha.(M\ \alpha) =_\eta \Lambda\alpha.\lambda x{:}\alpha.(M\ \alpha\ x) =_\beta \Lambda\alpha.\lambda x{:}\alpha.x$$ □

B.4 Encodings

B.4.1 Booleans

Recall that

$$\begin{aligned} \mathit{Bool} &= \forall\delta.\delta \to \delta \to \delta \\ \mathit{True} &: \mathit{Bool} \\ \mathit{True} &= \Lambda\delta.\lambda x{:}\delta.\lambda y{:}\delta.x \\ \mathit{False} &: \mathit{Bool} \\ \mathit{False} &= \Lambda\delta.\lambda x{:}\delta.\lambda y{:}\delta.y \end{aligned}$$

Theorem B.3 (Canonicity)**.** *If* $\bullet\ ;\ \bullet \vdash M : \mathit{Bool}$ *then* $\bullet\ ;\ \bullet \vdash M = \mathit{True} : \mathit{Bool}$ *or* $\bullet\ ;\ \bullet \vdash M = \mathit{False} : \mathit{Bool}$.

Proof. Using Lemma 4.6. Left as an exercise to the reader. ■

Hint. An example equivalence candidate with two elements can be defined as

$$M\ \mathbb{A}\ M' \iff M \mapsto\!\!\!\!\rightarrow_\beta x \twoheadleftarrow\!\!\!\!\mapsto_\beta M' \text{ or } M \mapsto\!\!\!\!\rightarrow_\beta y \twoheadleftarrow\!\!\!\!\mapsto_\beta M'$$

where x and y are two arbitrary, different free variables.

B.4.2 Sums

Recall that

$$\begin{aligned} \mathit{Sum}\ A\ B &= \forall\delta.(A \to \delta) \to (B \to \delta) \to \delta \\ \mathit{Left} &: \forall\alpha.\forall\beta.\alpha \to \mathit{Sum}\ \alpha\ \beta \\ \mathit{Left} &= \Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\Lambda\delta.\lambda l{:}\alpha \to \delta.\lambda r{:}\beta \to \delta.l\ x \\ \mathit{Right} &: \forall\alpha.\forall\beta.\beta \to \mathit{Sum}\ \alpha\ \beta \\ \mathit{Right} &= \Lambda\alpha.\Lambda\beta.\lambda x{:}\beta.\Lambda\delta.\lambda l{:}\alpha \to \delta.\lambda r{:}\beta \to \delta.r\ x \end{aligned}$$

Theorem B.4 (Canonicity)**.** *If* $\bullet\ ;\ \bullet \vdash M : \mathit{Sum}\ A\ B$ *then either*

a) $\bullet\ ;\ \bullet \vdash M = \mathit{Left}\ A\ B\ N' : \mathit{Sum}\ A\ B$ *for some* $N\ [\![A]\!]^{RC}\ N'$ *or*

b) $\bullet\ ;\ \bullet \vdash M = \mathit{Right}\ A\ B\ N' : \mathit{Sum}\ A\ B$ *for some* $N\ [\![B]\!]^{RC}\ N'$.

Proof. Using Lemma 4.6. Left as an exercise to the reader. ■

Hint. We can generalize the simple "boolean" two-element relation candidate and assume the free variables act like functions that can be applied to arguments with certain properties like so:

$$\begin{aligned} M\ \mathbb{C}\ M' \iff &\exists N_1\ [\![A]\!]^{RC}\ N_1'.\ M \mapsto\!\!\!\!\to_\beta x\ N_1 \text{ and } M' \mapsto\!\!\!\!\to_\beta x\ N_1' \\ &\text{or } \exists N_2\ [\![B]\!]^{RC}\ N_2'.\ M \mapsto\!\!\!\!\to_\beta y\ N_2 \text{ and } M' \mapsto\!\!\!\!\to_\beta y\ N_2' \end{aligned}$$

where x and y are two arbitrary, different free variables.

B.4.3 Products

Recall that

$$\begin{aligned} \mathit{Prod}\ A\ B &= \forall\delta.(A \to B \to \delta) \to \delta \\ \mathit{Pair} &: \forall\alpha.\forall\beta.\alpha \to \beta \to \mathit{Prod}\ \alpha\ \beta \\ \mathit{Pair} &= \Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.\Lambda\delta.\lambda p{:}\alpha \to \beta \to \delta.p\ x\ y \end{aligned}$$

Theorem B.5 (Canonicity)**.** *If* $\bullet\ ;\ \bullet \vdash M : \mathit{Prod}\ A\ B$ *then* $\bullet\ ;\ \bullet \vdash M = \mathit{Pair}\ A\ B\ N_1'\ N_2' : \mathit{Prod}\ A\ B$ *for some* $N_1\ [\![A]\!]^{RC}\ N_1'$ *and* $N_2\ [\![B]\!]^{RC}\ N_2'$.

Proof. Using Lemma 4.6. Left as an exercise to the reader. ■

Hint. Instead of defining a relation candidate that pretends two different free variables act like one-argument functions, a relation candidate can pretend one free variable x acts like a two-function argument expects its arguments have two different properties like so:

$$\begin{aligned} M\ \mathbb{C}\ M' \iff &\exists N_1\ [\![A]\!]^{RC}\ N_1'.\ \exists N_2\ [\![B]\!]^{RC}\ N_2'. \\ &M \mapsto\!\!\!\!\to_\beta x\ N_1\ N_2 \text{ and } M' \mapsto\!\!\!\!\to_\beta x\ N_1'\ N_2' \end{aligned}$$

B.4.4 Numbers

Recall that

$$\begin{aligned}
&Nat = \forall\delta.\delta \to (\delta \to \delta) \to \delta \\
&Zero : Nat \\
&Zero = \Lambda\delta.\lambda z{:}\delta.\lambda s{:}\delta \to \delta.z \\
&Succ : Nat \to Nat \\
&Succ = \lambda n{:}Nat.\Lambda\delta.\lambda z{:}\delta.\lambda s{:}\delta \to \delta.s\ (n\ \delta\ z\ s)
\end{aligned}$$

Define the n^{th} iteration of s as:

$$\begin{aligned}
s^0\ z &= z \\
s^{n+1}\ z &= s\ (s^n\ z)
\end{aligned}$$

Theorem B.6 (Numericity)**.** *If* $\bullet\ ;\ \bullet \vdash M : Nat$ *then*

$$\alpha\ ;\ z : \alpha, s : \alpha \to \alpha \vdash M\ \alpha\ z\ s = s^n\ z : \alpha$$

for some natural number $n \in \mathbb{N}$.

Proof. Using Lemma 4.6. Left as a challenge to the reader. ■

Theorem B.7 (Canonicity)**.** *If* $\bullet\ ;\ \bullet \vdash M : Nat$ *then either* $\bullet\ ;\ \bullet \vdash M = Zero : Nat$ *or* $\bullet\ ;\ \bullet \vdash M = Succ\ N : Nat$ *for some* $\bullet\ ;\ \bullet \vdash N : Nat$.

Proof. Using Theorem B.6. Left as an exercise to the reader. ■

Hint. These relation candidates can also be defined inductively as well. For example, this relation candidate is defined in terms of itself:

$$\begin{aligned}
M\ \mathbb{C}\ M' \iff &M \mapsto\!\!\!\!\twoheadrightarrow_\beta z \text{ and } M' \mapsto\!\!\!\!\twoheadrightarrow_\beta z \\
&\text{or } \exists N\ \mathbb{C}\ N'.\ M \mapsto\!\!\!\!\twoheadrightarrow_\beta s\ N \text{ and } M' \mapsto\!\!\!\!\twoheadrightarrow_\beta s\ N'
\end{aligned}$$

where z and s are two arbitrary, different free variables. This circular definition is well-defined, because it is the limit of this step-based, inductive definition of an increasingly-expanding sequence of relations $\mathbb{C}_n$:

$$\begin{aligned}
M\ \mathbb{C}_0\ M' &\iff \text{never} \\
M\ \mathbb{C}_{n+1}\ M' &\iff M \mapsto\!\!\!\!\twoheadrightarrow_\beta z \text{ and } M' \mapsto\!\!\!\!\twoheadrightarrow_\beta z \\
&\quad\ \text{or } \exists N\ \mathbb{C}_n\ N'.\ M \mapsto\!\!\!\!\twoheadrightarrow_\beta s\ N \text{ and } M' \mapsto\!\!\!\!\twoheadrightarrow_\beta s\ N'
\end{aligned}$$

Where $\mathbb{C} = \bigcup_{n=0}^{\infty} \mathbb{C}_n$.

B.5 Relational Parametricity

Let's add consider products and strings, and add them to our logical relation interpretation of types:

$$M\ [\![A \times B]\!]^C_\tau\ M' \iff (\texttt{fst}\, M)\ [\![A]\!]^C_\tau\ (\texttt{fst}\, M') \text{ and } (\texttt{snd}\, M)\ [\![B]\!]^C_\tau\ (\texttt{snd}\, M')$$
$$M\ [\![\texttt{string}]\!]^C_\tau\ M' \iff M \mapsto\!\!\!\!\gg_\beta c \ll\!\!\!\!\mapsto_\beta M \text{ and } c \text{ is a string literal}$$

Now we can model small modules, like a enumeration between red and blue, with a printing function:

$$\begin{aligned} \mathit{red\_v\_blue} : \{\, & \texttt{type}\, \alpha; \\ & \mathit{red} : \alpha; \\ & \mathit{blue} : \alpha; \\ & \mathit{print} : \alpha \to \texttt{string} \\ \} & \end{aligned}$$

such that $\mathit{print}(\mathit{red}) \mapsto\!\!\!\!\gg_\beta$ "`red`" and $\mathit{print}(\mathit{blue}) \mapsto\!\!\!\!\gg_\beta$ "`blue`". This module signature can be encoded as the following existential type with two different implementations:

$$\begin{aligned} &\mathit{red\_v\_blue}_{1,2} : \exists\alpha.(\alpha \times (\alpha \times (\alpha \to \texttt{string}))) \\ &\quad \mathit{red\_v\_blue}_1 = (\texttt{string}, (\text{“}\texttt{red}\text{”}, (\text{“}\texttt{blue}\text{”}, (\lambda x.x)))) \\ &\quad \mathit{red\_v\_blue}_2 = (\texttt{bool}, (\texttt{true}, (\texttt{false}, (\lambda x.\, \texttt{if}\, x\, \texttt{then}\, \text{“}\texttt{red}\text{”}\, \texttt{else}\, \text{“}\texttt{blue}\text{”})))) \end{aligned}$$

We want to show that $\mathit{red\_v\_blue}_1 \approx \mathit{red\_v\_blue}_2$ in some appropriate sense, even though their internal representation is incomparable.

Theorem B.8. $\mathit{red\_v\_blue}_1\ [\![\exists\alpha.(\alpha \times (\alpha \times (\alpha \to \texttt{string})))]\!]^{RC}\ \mathit{red\_v\_blue}_2$

Proof. Using Lemma 4.6. Left as a challenge to the reader. ■

Hint. The two different implementations use totally different internal representations, `string` versus `bool`. How do you relate `true` to "`red`" and `false` to "`blue`"??

Bibliography

Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985. ISBN 978-0-444-86748-3.

Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *International Conference on Functional Programming (ICFP)*, 2000. URL https://doi.org/10.1145/351240.351262.

P. Downen and Zena M. Ariola. Compiling with classical connectives. *Logical Methods in Computer Science*, 16, 2019. URL https://api.semanticscholar.org/CorpusID:199000843.

Paul Downen. Bi-orthogonality. https://pauldownen.com/notes/biorthogonality_notes.pdf, 2014.

Paul Downen. Abstract machines and classical realizability. OPLSS lecture notes, https://pauldownen.com/notes/amcr.pdf, 2022.

Paul Downen and Zena M. Ariola. Compositional semantics for composable continuations: from abortive to delimited control. In *International conference on Functional programming (ICFP)*, 2014a. doi: 10.1145/2628136.2628147. URL https://doi.org/10.1145/2628136.2628147.

Paul Downen and Zena M. Ariola. The duality of construction. In *European Symposium on Programming (ESOP)*, volume 8410, 2014b. URL https://doi.org/10.1007/978-3-642-54833-8_14.

Paul Downen and Zena M. Ariola. Duality in action. In *Formal Structures for Computation and Deduction (FSCD)*, volume 195, 2021. URL https://doi.org/10.4230/LIPIcs.FSCD.2021.1.

Paul Downen and Zena M. Ariola. Classical (co)recursion: Mechanics. *Journal of Functional Programming*, 33:e4, 2023. URL https://doi.org/10.1017/S0956796822000168.

Paul Downen, Philip Johnson-Freyd, and Zena M. Ariola. Structures for structural recursion. In *International Conference on Functional Programming (ICFP)*, 2015. URL https://doi.org/10.1145/2784731.2784762.

Paul Downen, Luke Maurer, Zena M. Ariola, and Simon Peyton Jones. Sequent calculus as a compiler intermediate language. In *International Conference on Functional Programming (ICFP)*, 2016. URL https://doi.org/10.1145/2951913.2951931.

Paul Downen, Zena M. Ariola, and Silvia Ghilezan. The duality of classical intersection and union types. *Fundamenta Informaticae*, 170(1-3), 2019. URL https://doi.org/10.3233/FI-2019-1855.

Paul Downen, Philip Johnson-Freyd, and Zena M. Ariola. Abstracting models of strong normalization for classical calculi. *Journal of Logical and Algebraic Methods in Programming*, 111, 2020. URL https://doi.org/10.1016/j.jlamp.2019.100512.

Gerhard Gentzen. Untersuchungen über das logische schließen. I. *Mathematische Zeitschrift*, 39(1), 1935. URL http://dx.doi.org/10.1007/BF01201353.

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, USA, 1989. URL https://doi.org/10.2307/2274726.

Philip Johnson-Freyd, Paul Downen, and Zena M. Ariola. First class call stacks: Exploring head reduction. In *WoC*, volume 212 of *EPTCS*, 2015. URL https://doi.org/10.4204/EPTCS.212.2.

J. W. Klop. *Term rewriting systems*, pages 1–116. Oxford University Press, Inc., USA, 1993. ISBN 0198537611.

Michel Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning (LPAR)*, volume 624, 1992. URL https://doi.org/10.1007/BFb0013061.

John C. Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Programming Symposium, Proceedings Colloque Sur La Programmation*, volume 19, 1974.

Philip Wadler. Call-by-value is dual to call-by-name. In *International Conference on Functional Programming (ICFP)*, 2003. URL https://doi.org/10.1145/944705.944723.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1), 1994. URL https://doi.org/10.1006/inco.1994.1093.