Foundations of Programming Languages

Paul Downen

July 3–8, 2018

Contents

Chapter 1

Static and Dynamic Semantics of a Little Language

1.1 Syntax

$$x, y, z \in \mathit{Variable} ::= \dots$$

A grammar for abstract syntax:

$$\begin{aligned}
n \in \mathit{Num} &::= 0 \mid 1 \mid 2 \mid \dots \\
b \in \mathit{Bool} &::= \texttt{true} \mid \texttt{false} \\
e \in \mathit{Expr} &::= x \mid \texttt{num}[n] \mid \texttt{bool}[b] \mid \texttt{plus}(e_1; e_2) \mid \texttt{less}(e_1; e_2) \mid \texttt{if}\,(e_1; e_2; e_3) \mid \texttt{let}\,(e_1; x.e_2)
\end{aligned}$$

A more "user-friendly" presentation of the same syntax:

$$\begin{aligned}
n \in \mathit{Num} &::= 0 \mid 1 \mid 2 \mid \dots \\
b \in \mathit{Bool} &::= \texttt{true} \mid \texttt{false} \\
e \in \mathit{Expr} &::= x \mid n \mid b \mid e_1 + e_2 \mid e_1 \le e_2 \mid \texttt{if}\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \mid \texttt{let}\, x = e_1 \,\texttt{in}\, e_2
\end{aligned}$$

The two grammars contain exactly the same information, but one is more explicit and regular at the cost of being more verbose:

$$\begin{aligned}
x &\equiv x \\
n &\equiv \texttt{num}[n] \\
b &\equiv \texttt{bool}[b] \\
e_1 + e_2 &\equiv \texttt{plus}(e_1; e_2) \\
e_1 \le e_2 &\equiv \texttt{less}(e_1; e_2) \\
\texttt{if}\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 &\equiv \texttt{if}\,(e_1; e_2; e_3) \\
\texttt{let}\, x = e_1 \,\texttt{in}\, e_2 &\equiv \texttt{let}\,(e_1; x.e_2)
\end{aligned}$$

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 expression).
2. The free variables of an expression remain the same after substitution (they cannot be *captured* by local bindings).

$$\begin{aligned}
BV(x) &= \{\} \\
BV(\texttt{num}[n]) &= \{\} \\
BV(\texttt{bool}[b]) &= \{\} \\
BV(\texttt{plus}(e_1; e_2)) &= BV(e_1) \cup BV(e_2) \\
BV(\texttt{less}(e_1; e_2)) &= BV(e_1) \cup BV(e_2) \\
BV(\texttt{if}\,(e_1; e_2; e_3)) &= BV(e_1) \cup BV(e_2) \cup BV(e_3) \\
BV(\texttt{let}\,(e_1; x.e_2)) &= BV(e_1) \cup (BV(e_2) \cup \{x\})
\end{aligned}$$

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(\texttt{num}[n]) &= \{\} \\
FV(\texttt{bool}[b]) &= \{\} \\
FV(\texttt{plus}(e_1; e_2)) &= FV(e_1) \cup FV(e_2) \\
FV(\texttt{less}(e_1; e_2)) &= FV(e_1) \cup FV(e_2) \\
FV(\texttt{if}\,(e_1; e_2; e_3)) &= FV(e_1) \cup FV(e_2) \cup FV(e_3) \\
FV(\texttt{let}\,(e_1; x.e_2)) &= FV(e_1) \cup (FV(e_2) \setminus \{x\})
\end{aligned}$$

1.3 Substitution

The capture-avoiding substitution operation $e_1[e/x]$ means to replace every *free* occurence of x appearing inside e_1 with the expression e.

$$\begin{aligned}
x[e/x] &= e \\
y[e/x] &= y && \text{(if } y \neq x\text{)} \\
\texttt{num}[n][e/x] &= \texttt{num}[n] \\
\texttt{bool}[b][e/x] &= \texttt{bool}[b] \\
\texttt{plus}(e_1; e_2)[e/x] &= \texttt{plus}(e_1[e/x]; e_2[e/x]) \\
\texttt{less}(e_1; e_2)[e/x] &= \texttt{less}(e_1[e/x]; e_2[e/x]) \\
\texttt{if}\,(e_1; e_2; e_3)[e/x] &= \texttt{if}\,(e_1[e/x]; e_2[e/x]; e_3[e/x]) \\
\texttt{let}\,(e_1; x.e_2)[e/x] &= \texttt{let}\,(e_1[e/x]; x.e_2) \\
\texttt{let}\,(e_1; y.e_2)[e/x] &= \texttt{let}\,(e_1; y.e_2[e/x]) && \text{(if } y \neq x \text{ and } y \notin FV(e)\text{)}
\end{aligned}$$

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

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:

$$\frac{}{x[e/x] = x} \qquad \frac{y \neq x}{y[e/x] = y} \qquad \frac{}{\mathtt{num}[n][e/x] = \mathtt{num}[n]} \qquad \frac{}{\mathtt{bool}[b][e/x] = \mathtt{bool}[b]}$$

$$\frac{e_1[e/x] = e_1' \quad e_2[e/x] = e_2'}{\mathtt{plus}(e_1; e_2)[e/x] = \mathtt{plus}(e_1'; e_2')} \qquad \frac{e_1[e/x] = e_1' \quad e_2[e/x] = e_2'}{\mathtt{less}(e_1; e_2)[e/x] = \mathtt{less}(e_1'; e_2')}$$

$$\frac{e_1[e/x] = e_1' \quad e_2[e/x] = e_2' \quad e_3[e/x] = e_3'}{\mathtt{if}\,(e_1; e_2; e_3)[e/x] = \mathtt{if}\,(e_1'; e_2'; e_3')}$$

$$\frac{e_1[e/x] = e_1'}{\mathtt{let}\,(e_1; x.e_2)[e/x] = \mathtt{let}\,(e_1'; x.e_2)} \qquad \frac{e_1[e/x] = e_1' \quad y \neq x \quad y \notin FV(e) \quad e_2[e/x] = e_2'}{\mathtt{let}\,(e_1; y.e_2)[e/x] = \mathtt{let}\,(e_1'; y.e_2')}$$

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

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

Proof. By induction on the syntax of the expression e'.

- y: Note that $BV(y) \cap FV(e) = \{\} \cap FV(e) = \{\}$. By definition $y[e/x]$ is defined for any variable y, with $y[e/x] = e$ when $y = x$ and $y[e/x] = y$ when $y \neq x$.

- $\mathtt{num}[n]$: Note that $BV(\mathtt{num}[n]) \cap FV(e) = \{\} \cap FV(e) = \{\}$. By definition, $\mathtt{num}[n][e/x] = \mathtt{num}[n]$, which is defined.

- $\mathtt{bool}[b]$: follows by definition similarly to the case for $\mathtt{num}[n]$.

- $\mathtt{plus}(e_1; e_2)$: The inductive hypothesis we get for e_1 and e_2 are

  Inductive Hypothesis. *For both $i = 1$ and $i = 2$, if $BV(e_i) \cap FV(e) = \{\}$ then $e_i[e/x]$ is defined.*

  Now, note that $BV(\mathtt{plus}(e_1; e_2)) = BV(e_1) \cup BV(e_2)$ by definition, so

$$\begin{aligned} BV(\mathtt{plus}(e_1; e_2)) \cap FV(e) &= (BV(e_1) \cup BV(e_2)) \cap FV(e) \\ &= (BV(e_1) \cap FV(e)) \cup (BV(e_2) \cap FV(e)) \end{aligned}$$

It follows that $BV(\texttt{plus}(e_1; e_2)) \cap FV(e) = \{\}$ exactly when $BV(e_i) \cap FV(e) = \{\}$ for both $i = 1$ and $i = 2$. Similarly, the substitution

$$\texttt{plus}(e_1; e_2)[e/x] = \texttt{plus}(e_1[e/x]; e_2[e/x])$$

is defined exactly when $e_i[e/x]$ is defined for both $i = 1$ and $i = 2$. By applying the inductive hypothesis to the assumption that $BV(\texttt{plus}(e_1; e_2)) \cap FV(e) = \{\}$, we learn that $\texttt{plus}(e_1; e_2)[e/x]$ is defined.

- $\texttt{less}(e_1; e_2)$ and $\texttt{if}\,(e_1; e_2; e_3)$: follows from the inductive hypothesis similarly to the case for $\texttt{plus}(e_1; e_2)$.
- $\texttt{let}\,(e_1; y.e_2)$: The inductive hypotheses we get for e_1 and e_2 are

  Inductive Hypothesis. *For both $i = 1$ and $i = 2$, if $BV(e_i) \cap FV(e) = \{\}$ then $e_i[e/x]$ is defined.*

  Note that $BV(\texttt{let}\,(e_1; y.e_2)) = BV(e_1) \cup (\{y\} \cup BV(e_2))$ by definition, so

  $$\begin{aligned} &BV(\texttt{let}\,(e_1; y.e_2)) \cap FV(e) \\ &= (BV(e_1) \cup (\{y\} \cup BV(e_2))) \cap FV(e) \\ &= (BV(e_1) \cap FV(e)) \cup (\{y\} \cap FV(e)) \cup (BV(e_2) \cap FV(e)) \end{aligned}$$

  It follows that $BV(\texttt{let}\,(e_1; y.e_2)) \cap FV(e) = \{\}$ exactly when $y \notin FV(e)$ *and* $BV(e_i) \cap FV(e) = \{\}$ for both $i = 1$ and $i = 2$. There are now two cases to consider, depending on whether or not x and y are equal.

  - $y = x$: By definition, $\texttt{let}\,(e_1; x.e_2)[e/x] = \texttt{let}\,(e_1[e/x]; x.e_2)$, which is defined only when $e_1[e/x]$ is defined. By applying the inductive hypothesis to the fact that $BV(e_1) \cap FV(e) = \{\}$ as implied by the assumption that $BV(\texttt{let}\,(e_1; x.e_2)) \cap FV(e)$, we learn that $e_1[e/x]$ must be defined, so $\texttt{let}\,(e_1; x.e_2)[e/x]$ is defined, too.
  - $y \neq x$: By definition, $\texttt{let}\,(e_1; x.e_2)[e/x] = \texttt{let}\,(e_1[e/x]; x.e_2)$, which is defined only when both $e_1[e/x]$ and $e_2[e/x]$ are defined *and* when the side condition $y \notin FV(e)$ is met. By applying the inductive hypothesis to the facts that $BV(e_1) \cap FV(e) = \{\}$, $BV(e_2) \cap FV(e) = \{\}$ as implied by the assumption that $BV(\texttt{let}\,(e_1; y.e_2)) \cap FV(e) = \{\}$, we learn that both $e_1[e/x]$ and $e_2[e/x]$ are defined. Furthermore, the same assumption also implies that $y \notin FV(e)$, meeting the side condition so that $\texttt{let}\,(e_1; y.e_2)[e/x]$ is defined. □

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

Proof. By induction on the syntax of e':

- y: Since $FV(y) = \{y\}$, the assumption that $x \notin FV(y)$ implies that $x \neq y$. Therefore, $y[e/x] = y$.

- $\texttt{num}[n]$: By definition, $\texttt{num}[n][e/x] = \texttt{num}[n]$.
- $\texttt{bool}[b]$: follows by definition similarly to the case for $\texttt{num}[n]$.
- $\texttt{plus}(e_1; e_2)$: The inductive hypotheses we get for e_1 and e_2 are

  Inductive Hypothesis. *For both $i = 1$ and $i = 2$, if $x \notin FV(e_i)$ then $e_i[e/x] = e$ when $e_i[e/x]$ is defined.*

  Now, note that $FV(\texttt{plus}(e_1; e_2)) = FV(e_1) \cup FV(e_2)$, so the assumption that $x \notin FV(\texttt{plus}(e_1; e_2))$ implies that $x \notin FV(e_1)$ and $x \notin FV(e_2)$. From the inductive hypothesis, we then learn that $e_1[e/x] = e_1$ and $e_2[e/x] = e_2$ are defined. So by definition of substitution

  $$\texttt{plus}(e_1; e_2)[e/x] = \texttt{plus}(e_1[e/x]; e_2[e/x]) = \texttt{plus}(e_1; e_2)$$

  when it is defined.
- $\texttt{less}(e_1; e_2)$ and $\texttt{if}\,(e_1; e_2; e_3)$: follows from the inductive hypothesis similarly to the case for $\texttt{plus}(e_1; e_2)$.
- $\texttt{let}\,(e_1; y.e_2)$: The inductive hypotheses we get for e_1 and e_2 are

  Inductive Hypothesis. *For both $i = 1$ and $i = 2$, if $x \notin FV(e_i)$ then $e_i[e/x] = e$ when $e_i[e/x]$ is defined.*

  Now, note that $FV(\texttt{let}\,(e_1; y.e_2)) = FV(e_1) \cup (FV(e_2) \setminus \{y\})$, so the assumption that $x \notin FV(\texttt{let}\,(e_1; y.e_2))$ implies that $x \notin FV(e_1)$ and that either $x = y$ or $x \notin FV(e_2)$. By applying the inductive hypothesis to the fact that $x \notin FV(e_1)$, we learn that $e_1[e/x] = e_1$ when it is defined. There are now two cases, depending on whether or not x and y are equal.

  - $y = x$: From the above fact that $e_1[e/x] = e_1$, we get

    $$\texttt{let}\,(e_1; x.e_2)[e/x] = \texttt{let}\,(e_1[e/x]; x.e_2) = \texttt{let}\,(e_1; x.e_2)$$

    when it is defined.
  - $y \neq x$: It must be that $x \notin FV(e_2)$, so by applying the inductive hypothesis we learn that $e_2[e/x] = e_2$ when it is defined. Therefore,

    $$\texttt{let}\,(e_1; y.e_2)[e/x] = \texttt{let}\,(e_1[e/x]; y.e_2[e/x]) = \texttt{let}\,(e_1; y.e_2)$$

    when it is defined. □

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 $e[y/x]$ is just a special case of the more general substitution operation $e[e'/x]$ since the variable y is an instance of an expression.

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

$$(\alpha) \qquad \texttt{let}\,(e_1; x.e_2) =_\alpha \texttt{let}\,(e_1; y.e_2[y/x]) \qquad (\text{if } y \notin FV(e_2))$$

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

$$\frac{e_1 =_\alpha e_1' \quad e_2[z/x] =_\alpha e_2'[z/y] \quad z \notin FV(e_1) \cup FV(e_2)}{\texttt{let}\,(e_1; x.e_2) =_\alpha \texttt{let}\,(e_1'; y.e_2')}\ \alpha$$

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

$$\frac{}{x =_\alpha x} \qquad \frac{}{\texttt{num}[n] =_\alpha \texttt{num}[n]} \qquad \frac{}{\texttt{bool}[b] =_\alpha \texttt{bool}[b]}$$

$$\frac{e_1 =_\alpha e_1' \quad e_2 =_\alpha e_2'}{\texttt{plus}(e_1; e_2) =_\alpha \texttt{plus}(e_1'; e_2')} \qquad \frac{e_1 =_\alpha e_1' \quad e_2 =_\alpha e_2'}{\texttt{less}(e_1; e_2) =_\alpha \texttt{less}(e_1'; e_2')}$$

$$\frac{e_1 =_\alpha e_1' \quad e_2 =_\alpha e_2' \quad e_3 =_\alpha e_3'}{\texttt{if}\,(e_1; e_2; e_3) =_\alpha \texttt{if}\,(e_1'; e_2'; e_3')}$$

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 expression e and variables x and y, $BV(e[y/x]) = BV(e)$ if $e[y/x]$ is defined.*

Proof. By induction on the syntax of the expression e. □

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

Proof. By induction on the syntax of e.

- y: Note that $FV(y) = \{\}$, so $FV(y) \cap X = \{\} \cap X = \{\}$ already for any set of variables X and no renaming is needed.
- $\texttt{num}[n]$ and $\texttt{bool}[b]$: these expressions have no free variables so no renaming is needed similar to the case for a variable y.
- $\texttt{plus}(e_1; e_2)$: The inductive hypotheses for the sub-expressions e_1 and e_2 are

Inductive Hypothesis. *For both $i = 1$ and $i = 2$, and for any set of variables X, there is an α-equivalent e'_i such that $e_1 =_\alpha e'_1$ and $X \cap BV(e'_i) = \{\}$.*

Now, since $BV(\texttt{plus}(e'_1; e'_2)) = BV(e'_1) \cup BV(e'_2)$, it follows that for any set of variables X,

$$\begin{aligned} BV(\texttt{plus}(e'_1; e'_2)) \cap X &= (BV(e'_1) \cup BV(e'_2)) \cap X \\ &= (BV(e'_1) \cap X) \cup (BV(e'_2) \cap X) \\ &= \{\} \cup \{\} = \{\} \end{aligned}$$

Furthermore, this expression is α-equivalent to the original one, by applying the α rule in both sub-expressions like so:

$$\texttt{plus}(e_1; e_2) =_\alpha \texttt{plus}(e'_1; e_2) =_\alpha \texttt{plus}(e'_1; e'_2)$$

- $\texttt{less}(e_1; e_2)$ and $\texttt{if}\,(e_1; e_2; e_3)$: follows from the inductive hypothesis similar to the case.
- $\texttt{let}\,(e_1; x.e_2)$: The inductive hypotheses for the sub-expressions e_1 and e_2 are

  Inductive Hypothesis. *For both $i = 1$ and $i = 2$, and for any set of variables X, there is an α-equivalent e'_i such that $e_1 =_\alpha e'_1$ and $X \cap BV(e'_i) = \{\}$.*

  By applying the inductive hypothesis to any set of variables X, we find a new pair of sub-expressions e'_1 and e'_2 such that $e_i =_\alpha e'_i$ and $X \cap BV(e'_i) = \{\}$ for both $i = 1$ and $i = 2$. It may be possible that $x \in X$, so that the bound variable x would need to be renamed. We can handle this worst-case scenario by choosing an arbitrary new variable $y \notin X \cup FV(e'_2) \cup BV(e'_2)$ for any set of variables X. Now, observe that we have the α-equivalent $\texttt{let}$-expression

$$\texttt{let}\,(e_1; x.e_2) =_\alpha \texttt{let}\,(e'_1; x.e_2) =_\alpha \texttt{let}\,(e'_1; x.e'_2) =_\alpha \texttt{let}\,(e'_1; y.e'_2[y/x])$$

  where the substitution $e'_2[y/x]$ is defined because $y \notin BV(e'_2)$. Furthermore, the bound variables of this new expression are distinct from X because

$$\begin{aligned} BV(\texttt{let}\,(e'_1; y.e'_2[y/x])) &= BV(e'_1) \cup (BV(e'_2[y/x]) \cup \{y\}) \\ &= BV(e'_1) \cup (BV(e'_2) \cup \{y\}) \end{aligned}$$

  and so

$$\begin{aligned} &BV(\texttt{let}\,(e'_1; y.e'_2[y/x])) \cap X \\ &= (BV(e'_1) \cap X) \cup (BV(e'_2) \cap X) \cup (\{y\} \cap X) \\ &= \{\} \cup \{\} \cup \{\} = \{\} \end{aligned}$$

□

Theorem 1.1. *For any expressions e_1 and e and any variable x, there is an α-equivalent e_2 such that $e_1 =_\alpha e_2$ and $e_2[e/x]$ is defined.*

Proof. We can find such an e_2 by renaming e_1 so that $BV(e_2) \cup FV(e) = \{\}$, which implies that $e_2[e/x]$ is defined. □

Theorem 1.2. *For any expressions e_1, e_2, and e and any variable x, if $e_1 =_\alpha e_2$ then $e_1[e/x] =_\alpha e_2[e/x]$ whenever both $e_1[e/x]$ and $e_2[e/x]$ are both defined.*

Proof. By induction on the derivation of derivation $e_1 =_\alpha e_2$. □

1.5 Static Semantics: Types

$$\tau \in \mathit{Type} ::= \mathtt{num} \mid \mathtt{bool}$$

The type system for our little language consists of inference rules for concluding *hypothetical judgements* of the form $\Gamma \vdash e : \tau$, which can be read as "in the environment Γ, e has type τ." The environment Γ is a list of assumed types for free variables as shown in the following grammar

$$\Gamma \in \mathit{Environment} ::= \bullet \mid \Gamma, x : \tau$$

Note that we will impose an extra side condition on valid Γs so that a variable can only appear *at most* once in Γ. For example, $x : \tau_1, y : \tau_2, x : \tau_3$ is not allowed. In contrast, a type may appear many times in Γ if multiple variables have the same type, like in this valid environment $x : \mathtt{num}, y : \mathtt{num}, z : \mathtt{num}$. Furthermore, the order of Γ does not matter, so that the following two environments are considered equal: $x : \mathtt{num}, y : \mathtt{bool} = y : \mathtt{bool}, x : \mathtt{num}$.

The inference rules for typing expressions of our little language are as follows:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \mathtt{num}[n] : \mathtt{num}} \qquad \frac{}{\Gamma \vdash \mathtt{bool}[b] : \mathtt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{num} \quad \Gamma \vdash e_2 : \mathtt{num}}{\Gamma \vdash \mathtt{plus}(e_1; e_2) : \mathtt{num}} \qquad \frac{\Gamma \vdash e_1 : \mathtt{num} \quad \Gamma \vdash e_2 : \mathtt{num}}{\Gamma \vdash \mathtt{less}(e_1; e_2) : \mathtt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathtt{if}\,(e_1; e_2; e_3) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau' \quad \Gamma, x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \mathtt{let}\,(e_1; x.e_2) : \tau}$$

Note that the typing rule for variables can be read as "x has type τ in any environment where x is assumed to have type τ." Since the order of Γ does not matter, the assumption $x : \tau$ can appear anywhere to the left of $\vdash$ to use this rule.

Furthermore, the requirement that Γ does not have multiple assumptions for the same variable means that the typing rule for `let`-expressions only applies when the bound variable x does not already appear in Γ. This means that an expression like

$$\mathtt{let}\, x = 2\, \mathtt{in}\, \mathtt{let}\, x = 1\, \mathtt{in}\, x$$

which *should* be well-typed actually isn't, because the inner `let` attempts to "shadow" the binding of x given by the outer `let`. This problem can be avoided by renaming away all shadowing, as in

$$\texttt{let}\,x = 2\,\texttt{in}\,\texttt{let}\,x = 1\,\texttt{in}\,x =_\alpha \texttt{let}\,x = 2\,\texttt{in}\,\texttt{let}\,y = 1\,\texttt{in}\,y$$

where there is a typing derivation of $\bullet \vdash \texttt{let}\,x = 2\,\texttt{in}\,\texttt{let}\,y = 1\,\texttt{in}\,y : \texttt{num}$. And since we always consider α-equivalent expressions to be the "same" expression, that means that $\texttt{let}\,x = 2\,\texttt{in}\,\texttt{let}\,x = 1\,\texttt{in}\,x$ is also well-typed up to α-equivalence.

1.6 Dynamic Semantics: Behavior

Small-step, operational reduction relation $e \mapsto e'$, "e steps to e'." Multi-step operational semantics is the smallest binary relation $e \mapsto^* e'$ between terms closed under the following:

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

The basic steps for reducing an expression:

$$\begin{aligned}
\texttt{plus}(\texttt{num}[n_1];\texttt{num}[n_2]) &\mapsto \texttt{num}[n_1 + n_2]\\
\texttt{less}(\texttt{num}[n_1];\texttt{num}[n_2]) &\mapsto \texttt{bool}[n_1 = n_2]\\
\texttt{if}\,(\texttt{bool}[\texttt{true}];e_1;e_2) &\mapsto e_1\\
\texttt{if}\,(\texttt{bool}[\texttt{false}];e_1;e_2) &\mapsto e_2\\
\texttt{let}\,(e_1;x.e_2) &\mapsto e_2[e_1/x]
\end{aligned}$$

These are *axioms*; they apply exactly as-is to an expression. But they are not enough to reduce expressions down to an answer (a number or boolean literal)! What about

$$(1 + 2) + 3 = \texttt{plus}(\texttt{plus}(\texttt{num}[1];\texttt{num}[2]);\texttt{num}[4])$$

You cannot yet add $1 + 2$ to 4 because $1 + 2$ is not a number literal. Rather, you want to add 1 and 2 *first*, and *then* add that result to 4. Hence, the need to reduce sub-expressions, as expressed by the following inference rules:

$$\frac{}{\texttt{num}[n]\,\texttt{val}} \qquad \frac{}{\texttt{bool}[b]\,\texttt{val}}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{plus}(e_1; e_2) \mapsto \texttt{plus}(e_1'; e_2)} \qquad \frac{e_1\,\texttt{val} \quad e_2 \mapsto e_2'}{\texttt{less}(e_1; e_2) \mapsto \texttt{less}(e_1; e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{less}(e_1; e_2) \mapsto \texttt{less}(e_1'; e_2)} \qquad \frac{e_1\,\texttt{val} \quad e_2 \mapsto e_2'}{\texttt{less}(e_1; e_2) \mapsto \texttt{less}(e_1; e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{if}\,(e_1; e_2; e_3) \mapsto \texttt{if}\,(e_1'; e_2; e_3)}$$

Combined with the above axioms, these rules are now enough to reduce expressions. For example, we can take a step in

$$\frac{\texttt{plus}(\texttt{num}[1]; \texttt{num}[2]) \mapsto \texttt{num}[3]}{\texttt{plus}(\texttt{plus}(\texttt{num}[1]; \texttt{num}[2]); \texttt{num}[4]) \mapsto \texttt{plus}(\texttt{num}[3]; \texttt{num}[4])}$$

Chaining together multiple (zero or more) reduction steps is done by

$$\frac{}{e \mapsto^* e} \qquad \frac{e \mapsto^* e' \quad e' \mapsto^* e''}{e \mapsto^* e''} \qquad \frac{e \mapsto e'}{e \mapsto^* e'}$$

But these inference rules are awfully repetitive. A more concise presentation of exactly the same thing is to define a grammar of *values* (a subset of expressions) and *evaluation contexts* (a subset of expression contexts) like so:

$$\begin{aligned} V \in \mathit{Value} &::= \texttt{num}[n] \mid \texttt{bool}[b] \\ E \in \mathit{EvalCxt} &::= \square \mid \texttt{plus}(E; e) \mid \texttt{plus}(V; E) \mid \texttt{less}(E; e) \mid \texttt{less}(V; E) \mid \texttt{if}\,(E; e_1; e_2) \end{aligned}$$

Now, the judgement $e\,\texttt{val}$ is derivable exactly when $e \in \mathit{Value}$, and *all* of the above inference rules for evaluating certain sub-expressions are expressed by the *one* inference rule:

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

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

$$\begin{aligned} \square[e] &= e \\ \texttt{plus}(E; e')[e] &= \texttt{plus}(E[e]; e') \\ \texttt{plus}(V; E)[e] &= \texttt{plus}(V; E[e]) \\ \texttt{less}(E; e')[e] &= \texttt{less}(E[e]; e') \\ \texttt{less}(V; E)[e] &= \texttt{less}(V; E[e]) \\ \texttt{if}\,(E; e_1'; e_2')[e] &= \texttt{if}\,(E[e]; e_1'; e_2') \end{aligned}$$

Note that, unlike with substitution, there is no issues involving capture when plugging a term into a context. In fact, the convention is that plugging a term into a more general kind of context (which might place the hole $\square$ under a binder like $\texttt{let}\,(e_1; x.\square)$) will *intentionally* capture free variables of the expression that's replacing $\square$. For example, reducing the expression $\texttt{plus}(\texttt{plus}(\texttt{num}[1];\texttt{num}[2]);\texttt{num}[4])$ goes like

$$\begin{array}{ll}
\texttt{plus}(\texttt{plus}(\texttt{num}[1];\texttt{num}[2]);\texttt{num}[4]) & \\
\mapsto \texttt{plus}(\texttt{num}[3];\texttt{num}[4]) & (E = \texttt{plus}(\square;\texttt{num}[4]), \texttt{plus}(\texttt{num}[1];\texttt{num}[2]) \mapsto \texttt{num}[3]) \\
\mapsto \texttt{num}[7] & (E = \square, \texttt{plus}(\texttt{num}[3];\texttt{num}[4]) \mapsto \texttt{num}[7])
\end{array}$$

1.7 Statics & Dynamics: Type Safety

1.7.1 Progress

Lemma 1.5 (Canonical Forms)**.**

- *If* $\bullet \vdash V : \texttt{num}$ *then there is an* n *such that* $V = \texttt{num}[n]$.
- *If* $\bullet \vdash V : \texttt{bool}$ *then there is a* b *such that* $V = \texttt{bool}[b]$.

Proof. By inversion on the possible derivations of $\bullet \vdash V : \texttt{num}$ and $\bullet \vdash V : \texttt{bool}$. $\square$

Lemma 1.6 (Progress)**.** *If* $\bullet \vdash e : \tau$ *then either* $e\,\texttt{val}$ *or there exists an* e' *such that* $e \mapsto e'$.

Proof. By induction on the derivation $\mathcal{D}$ of $\bullet \vdash e : \tau$,

$$\begin{array}{c} \vdots\ \mathcal{D} \\ \bullet \vdash e : \tau \end{array}$$

- The bottom inference cannot possibly be the axiom for variables

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

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

- If the bottom inference is the axiom for numbers or booleans

$$\frac{}{\bullet \vdash \texttt{num}[n] : \texttt{num}} \qquad \frac{}{\bullet \vdash \texttt{bool}[n] : \texttt{bool}}$$

  then the expression is a value since $\texttt{num}[n]\,\texttt{val}$ and $\texttt{bool}[b]\,\texttt{val}$ are both axioms.

- If the bottom inference is the rule for $\texttt{plus}$

$$\frac{\begin{array}{c}\vdots\ \mathcal{D}_1 \\ \bullet \vdash e_1 : \texttt{num}\end{array} \quad \begin{array}{c}\vdots\ \mathcal{D}_2 \\ \bullet \vdash e_2 : \texttt{num}\end{array}}{\bullet \vdash \texttt{plus}(e_1; e_2) : \texttt{num}}$$

  then proceed by induction on the derivations $\mathcal{D}_i$ of $\bullet \vdash e_i : \texttt{num}$:

Inductive Hypothesis. *For $i = 1$ and $i = 2$, either e_i `val` or there is an e'_i such that $e_i \mapsto e'_i$.*

There are three cases to consider depending on the results of the inductive hypothesis; whether e_i is a value or takes a step:

– If $e_1 \mapsto e'_1$ for some e'_1, then

$$\frac{e_1 \mapsto e'_1}{\texttt{plus}(e_1; e_2) \mapsto \texttt{plus}(e'_1; e_2)}$$

– If e_1 `val` and $e_2 \mapsto e'_2$ for some e'_2, then

$$\frac{e_1\,\texttt{val} \quad e_2 \mapsto e'_2}{\texttt{plus}(e_1; e_2) \mapsto \texttt{plus}(e_1; e'_2)}$$

– If both e_1 `val` and e_2 `val`, then there must be numbers n_1 and n_2 such that $e_i = \texttt{num}[n_i]$ (canonical forms). Therefore,

$$\texttt{plus}(e_1; e_2) = \texttt{plus}(\texttt{num}[n_1]; \texttt{num}[n_2]) \mapsto \texttt{num}[n_1 + n_2]$$

- If the bottom inference is the rule for `less`, then progress follows similarly to the above case for `plus`.
- If the bottom inference is the rule for `if`

$$\frac{\begin{array}{c}\vdots\,\mathcal{D}_1\\ \bullet \vdash e_1 : \texttt{bool}\end{array} \quad \begin{array}{c}\vdots\,\mathcal{D}_2\\ \bullet \vdash e_2 : \tau\end{array} \quad \begin{array}{c}\vdots\,\mathcal{D}_3\\ \bullet \vdash e_3 : \tau\end{array}}{\bullet \vdash \texttt{if}\,(e_1; e_2; e_3) : \tau}$$

then proceed by induction on the derivation $\mathcal{D}_1$ of $\bullet \vdash e_1 : \texttt{bool}$:

Inductive Hypothesis. *Either e_1 `val` or there is an e'_1 such that $e_1 \mapsto e'_1$.*

There are two cases depending on the result of the inductive hypothesis:

– If $e_1 \mapsto e'_1$ for some e'_1, then

$$\frac{e_1 \mapsto e'_1}{\texttt{if}\,(e_1; e_2; e_3) \mapsto \texttt{if}\,(e'_1; e_2; e_3)}$$

– If e_1 `val`, then there must be a boolean b such that $e_1 = \texttt{bool}[b]$ (canonical forms). Therefore,

$$\texttt{if}\,(e_1; e_2; e_3) = \texttt{if}\,(\texttt{bool}[b]; e_2; e_3) \mapsto e'$$

where $e' = e_2$ when $b = \texttt{true}$ and $e' = e_3$ when $b = \texttt{false}$.

- If the bottom inference is the rule for `let`

$$\dfrac{\overset{\vdots\ \mathcal{D}_1}{\bullet \vdash e_1 : \tau'} \quad \overset{\vdots\ \mathcal{D}_2}{x : \tau' \vdash e_2 : \tau}}{\bullet \vdash \texttt{let}\,(e_1; x.e_2) : \tau}$$

  then there is always the reduction step

$$\texttt{let}\,(e_1; x.e_2) \mapsto e_2[e_1/x]$$

□

1.7.2 Preservation

Lemma 1.7 (Typed Substitution). *If $\Gamma, x : \tau' \vdash e : \tau$ and $\Gamma \vdash e' : \tau'$ then $\Gamma \vdash e[e'/x] : \tau$.*

Proof. By induction on the derivation $\mathcal{D}$ of $\Gamma, x : \tau' \vdash e : \tau$. □

Lemma 1.8 (Preservation). *If $\Gamma \vdash e : \tau$ and $e \mapsto e'$ then $\Gamma \vdash e' : \tau$.*

Proof. By induction on the derivation $\mathcal{E}$ of $e \mapsto e'$ and inversion on the derivation $\mathcal{D}$ of $\Gamma \vdash e : \tau$. First we have the cases where $\mathcal{E}$ is one of the reduction step axioms:

- If $\mathcal{E} = \overline{\texttt{plus}(\texttt{num}[n_1];\texttt{num}[n_2]) \mapsto \texttt{num}[n_1 + n_2]}$, then it must be that $\tau = \texttt{num}$, and observe that $\overline{\Gamma \vdash \texttt{num}[n_1 + n_2] : \texttt{num}}$.

- If $\mathcal{E} = \overline{\texttt{less}(\texttt{num}[n_1];\texttt{num}[n_2]) \mapsto \texttt{bool}[n_1 = n_2]}$, then it must be that $\tau = \texttt{bool}$, and observe that $\overline{\Gamma \vdash \texttt{bool}[n_1 = n_2] : \texttt{bool}}$.

- If $\mathcal{E} = \overline{\texttt{if}\,(\texttt{bool}[b]; e_1; e_2) \mapsto e'}$, where $e' = e_1$ or $e' = e_2$, then it must be that the derivation $\mathcal{D}$ concludes with

$$\dfrac{\overline{\Gamma \vdash \texttt{bool}[b] : \texttt{bool}} \quad \overset{\vdots\ \mathcal{D}_1}{\Gamma \vdash e_1 : \tau} \quad \overset{\vdots\ \mathcal{D}_2}{\Gamma \vdash e_2 : \tau}}{\Gamma \vdash \texttt{if}\,(\texttt{bool}[b]; e_1; e_2) : \tau}$$

  In either case of $e' = e_i$, we get that

$$\overset{\vdots\ \mathcal{D}_\rangle}{\Gamma \vdash e_i : \tau}$$

- If $\mathcal{E} = \overline{\texttt{let}\,(e_1; x.e_2) \mapsto e_2[e_1/x]}$, then it must be that the derivation $\mathcal{D}$ concludes with

$$\dfrac{\overset{\vdots\ \mathcal{D}_1}{\Gamma \vdash e_1 : \tau'} \quad \overset{\vdots\ \mathcal{D}_2}{\Gamma, x : \tau' \vdash e_2 : \tau}}{\Gamma \vdash \texttt{let}\,(e_1; x.e_2) : \tau}$$

  By typed substitution, it follows that $\Gamma \vdash e_2[e_1/x] : \tau$.

The remaining cases are for where a reduction step is applied to a sub-expression (i.e. within an evaluation context). These cases all follow from the inductive hypothesis. For example, in the case where

$$\mathcal{E} = \dfrac{\begin{matrix} \vdots\ \mathcal{E}' \\ e_1 \mapsto e_1' \end{matrix}}{\texttt{plus}(e_1; e_2) \mapsto \texttt{plus}(e_1'; e_2)}$$

then the derivation $\mathcal{D}$ must conclude with

$$\mathcal{D} = \dfrac{\begin{matrix} \vdots\ \mathcal{D}_1 \\ \Gamma \vdash e_1 : \texttt{num} \end{matrix} \quad \begin{matrix} \vdots\ \mathcal{D}_2 \\ \Gamma \vdash e_2 : \texttt{num} \end{matrix}}{\Gamma \vdash \texttt{plus}(e_1; e_2) : \texttt{num}}$$

By applying the induction hypothesis to the derivations $\mathcal{E}'$ and $\mathcal{D}_1$, we learn that there is a derivation $\mathcal{D}_1'$ of $\Gamma \vdash e_1' : \texttt{num}$, and so

$$\dfrac{\begin{matrix} \vdots\ \mathcal{D}_1' \\ \Gamma \vdash e_1' : \texttt{num} \end{matrix} \quad \begin{matrix} \vdots\ \mathcal{D}_2 \\ \Gamma \vdash e_2 : \texttt{num} \end{matrix}}{\Gamma \vdash \texttt{plus}(e_1; e_2) : \texttt{num}}$$

The other cases for evaluation inside of a `plus`, `less`, and `if` follow similarly. □

Corollary 1.1 (Preservation*). *If* $\Gamma \vdash e : \tau$ *and* $e \mapsto^* e'$ *then* $\Gamma \vdash e' : \tau$.

Proof. Follows from the single-step preservation lemma by induction on the derivation of $e \mapsto^* e'$. □

1.7.3 Type Safety

Definition 1.1 (Stuck). An expression e is *stuck* if e `val` is not derivable and there is no e' such that $e \mapsto e'$ is derivable.

Theorem 1.3 (Type Safety). *If* $\bullet \vdash e : \tau$ *and* $e \mapsto^* e'$ *then* e' *is not stuck.*

Proof. From preservation, we know that $\bullet \vdash e' : \tau$. Then from progress we know that either e' `val` or $e' \mapsto e''$ (for some e'') is derivable. So e' cannot be stuck. □

Chapter 2

Lambda Calculus

2.1 Syntax

$$e \in \mathit{Expr} ::= x \mid \texttt{lam}(x.e) \mid \texttt{app}(e_1; e_2)$$

$$e \in \mathit{Expr} ::= x \mid \lambda x.e \mid e_1\ e_2$$

$$\lambda x.e = \texttt{lam}(x.e) \qquad\qquad e_1\ e_2 = \texttt{app}(e_1; e_2)$$

Some syntactic sugar

$$\texttt{let}\, x = e_1 \,\texttt{in}\, e_2 = (\lambda x.e_2)\ e_1$$

2.2 Substitution and Scope

$$\begin{aligned} BV(x) &= \{\} & FV(x) &= \{x\} \\ BV(\lambda x.e) &= BV(e) \cup \{x\} & FV(\lambda x.e_1) &= FV(e_1) \setminus \{x\} \\ BV(e_1\ e_2) &= BV(e_1) \cup BV(e_2) & FV(e_1\ e_2) &= FV(e_1) \cup FV(e_2) \end{aligned}$$

$$\begin{aligned} x\,[e/x] &= e & & \\ y\,[e/x] &= x & & (x \neq y) \\ (\lambda x.e')\,[e/x] &= \lambda x.e' & & \\ (\lambda y.e')\,[e/x] &= \lambda y.(e'\,[e/x]) & & (x \neq y) \text{ and } y \notin FV(e) \\ (e_1\ e_2)\,[e/x] &= (e_1\,[e/x])\ (e_2\,[e/x]) & & \end{aligned}$$

Lemma 2.1. *For all terms e', e, if $BV(e') \cap FV(e) = \{\}$ then $e'\,[e/x]$ is defined.*

Proof. By induction on the syntax of the term e'. □

2.3 Laws

2.3.1 Alpha

$$(\alpha) \qquad \lambda x.e =_\alpha \lambda y.(e\,[y/x])$$

Lemma 2.2. *For any term e and set of variables X, there exists a term e' such that $e =_\alpha e'$ and $X \cap BV(e') = \{\}$.*

Proof. By induction on the syntax of the term e. □

Theorem 2.1. *For all terms e_1, e_2, there exists a term e'_1 such that $e_1 =_\alpha e'_1$ and $e'_1\,[e_2/x]$ is defined.*

Theorem 2.2. *If $e_1 =_\alpha e_2$ then $e_1[e/x] =_\alpha e_2[e/x]$ when both are defined.*

2.3.2 Beta

$$(\beta) \qquad (\lambda x.e_1)\ e_2 =_\beta e_1\,[e_2/x]$$

2.3.3 Eta

$$(\eta) \qquad \lambda x.(e\ x) =_\eta e : \tau_1 \to \tau_2 \qquad \texttt{if}\, x \notin FV(e)$$

2.4 Dynamic Semantics: Call-by-Name vs Call-by-Value

2.4.1 Call-by-Name

Operational reduction rules (axioms):

$$(\lambda x.e_1)\ e_2 \mapsto e_1\,[e_2/x]$$

Evaluation context rules (inferences):

$$\frac{e_1 \mapsto e'_1}{e_1\ e_2 \mapsto e'_1\ e_2}$$

$$E \in \mathit{EvalCxt} ::= \square \mid E\ e$$

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

2.4.2 Call-by-Value

$$V \in \mathit{Value} ::= x \mid \lambda x.e$$

$$(\lambda x.e)\ V \mapsto e\,[V/x]$$

$$E \in \mathit{EvalCxt} ::= \Box \mid E\ e \mid V\ E$$

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

$$\frac{e_1 \mapsto e_2'}{e_1\ e_2 \mapsto e_1'\ e_2} \qquad \frac{e \mapsto e'}{V\ e \mapsto V\ e'}$$

2.5 Some Encodings

$$\begin{aligned} \texttt{if}\,e\,\texttt{then}\,e_1\,\texttt{else}\,e_2 &= (e\ e_1)\ e_2 \\ \texttt{true} &= \lambda x.\lambda y.x \\ \texttt{false} &= \lambda x.\lambda y.y \end{aligned}$$

$$\begin{aligned} \texttt{if true then}\,e_1\,\texttt{else}\,e_2 &= ((\lambda x.\lambda y.x)\ e_1)\ e_2 \\ &\mapsto (\lambda y.e_1) && (E = \Box\ e_2) \\ &\mapsto e_1 && (E = \Box) \end{aligned}$$

$$\begin{aligned} \texttt{if false then}\,e_1\,\texttt{else}\,e_2 &= ((\lambda x.\lambda y.y)\ e_1)\ e_2 \\ &\mapsto (\lambda y.y) && (E = \Box\ e_2) \\ &\mapsto e_2 && (E = \Box) \end{aligned}$$

2.6 Intermezzo: Russel's Paradox

An implementation of sets where

$$e \in e' = e'\ e$$

Example:

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

Russel's set

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

is then written as

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

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

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

2.7 Untyped λ-Calculus: Recursion

Unlike the little language from Chapter 1, not every expression reaches an answer. For example:

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

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, not every expression in the λ-calculus actually leads to 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.

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

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

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

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

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

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

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

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

$$\begin{aligned}\mathit{times}\ 0\ y &= Y\ \mathit{timesish}\ 0\ y\\ &\mapsto \mathit{timesish}\ (Y\ \mathit{timesish})\ 0\ y\\ &\mapsto^* \texttt{if}\,0 \leq 0\,\texttt{then}\,0\,\texttt{else}\,y + (Y\ \mathit{timesish}\ (0-1)\ y)\\ &\mapsto 0\\ \mathit{times}\ (x+1)\ y &= Y\ \mathit{timesish}\ (x+1)\ y\\ &\mapsto \mathit{timesish}\ (Y\ \mathit{timesish})\ (x+1)\ y\\ &\mapsto^* \texttt{if}\,(x+1) \leq 0\,\texttt{then}\,0\,\texttt{else}\,y + (Y\ \mathit{timesish}\ (x+1-1)\ y)\\ &\mapsto^* y + (Y\ \mathit{timesish}\ (x+1-1)\ y)\\ &\to^* y + (\mathit{times}\ x\ y)\end{aligned}$$

2.8 Static Semantics: "Simple" Types

$$\begin{aligned}\alpha, \beta \in \mathit{TypeVariable} &::= \ldots\\ \tau \in \mathit{Type} &::= \alpha \mid \tau_1 \to \tau_2\end{aligned}$$

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

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ {\to}I \qquad\qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}\ {\to}E$$

2.9 Simply-Typed λ-Calculus: Termination

Theorem 2.3 (Termination)**.** *If $\Gamma \vdash e : \tau$ then there is an expression e' such that $e \mapsto^* e' \not\mapsto$.*

Corollary 2.1. *If $\bullet \vdash e : \texttt{num}$ then there is a number n such that $e \mapsto^* \texttt{num}[n]$.*

Chapter 3

Products and Sums

3.1 Syntax

$$\begin{aligned} e \in \mathit{Expr} &::= \ldots \mid \langle e_1, e_2 \rangle \mid \langle\rangle \mid e\cdot\pi_1 \mid e\cdot\pi_2 \\ &\quad \mid \iota_1\cdot e \mid \iota_2\cdot e \mid \texttt{case}\, e\, \texttt{of}\, \{\iota_1\cdot x \Rightarrow e_1 \mid \iota_2\cdot y \Rightarrow e_2\} \mid \texttt{case}\, e\, \texttt{of}\, \{\} \\ \tau \in \mathit{Type} &::= \ldots \mid \tau_1 \times \tau_2 \mid \texttt{unit} \mid \tau_1 + \tau_2 \mid \texttt{void} \end{aligned}$$

$$\begin{aligned} BV(\langle e_1, e_2 \rangle) &= BV(e_1) \cup BV(e_2) \\ BV(\langle\rangle) &= \{\} \\ BV(e\cdot\pi_i) &= BV(e) \end{aligned}$$

$$\begin{aligned} FV(\langle e_1, e_2 \rangle) &= FV(e_1) \cup FV(e_2) \\ FV(\langle\rangle) &= \{\} \\ FV(e\cdot\pi_i) &= FV(e) \end{aligned}$$

$$\begin{aligned} \langle e_1, e_2 \rangle\, [e/x] &= \langle e_1\, [e/x], e_2\, [e/x] \rangle \\ (e'\cdot\pi_i)\, [e/x] &= (e'\, [e/x])\cdot\pi_i \end{aligned}$$

$$\begin{aligned} BV(\iota_i\cdot e) &= BV(e) \\ BV(\texttt{case}\, e\ \{\iota_1\cdot x \Rightarrow e_1 \mid \iota_2\cdot y \Rightarrow e_2\}) &= BV(e) \cup (BV(e_1') \cup \{x\}) \cup (BV(e_2') \cup \{y\}) \\ BV(\texttt{case}\, e\ \{\}) &= BV(e) \end{aligned}$$

$$\begin{aligned} FV(\iota_i{\cdot}e) &= FV(e) \\ FV(\texttt{case}\,e\ \{\iota_1{\cdot}x \Rightarrow e_1 \mid \iota_2{\cdot}y \Rightarrow e_2\}) &= FV(e) \cup (FV(e_1) \setminus \{x\}) \cup (FV(e_2) \setminus \{y\}) \\ FV(\texttt{case}\,e\ \{\}) &= FV(e) \end{aligned}$$

$$\begin{aligned} (\iota_i{\cdot}e')\,[e/x] &= \iota_i{\cdot}(e'\,[e/x]) \\ \begin{pmatrix} \texttt{case}\,e'\,\texttt{of} \\ \quad \iota_1{\cdot}y_1 \Rightarrow e_1 \\ \quad \iota_2{\cdot}y_2 \Rightarrow e_2 \end{pmatrix} [e/x] &= \begin{pmatrix} \texttt{case}\,(e'\,[e/x])\,\texttt{of} \\ \quad \iota_1{\cdot}y_1 \Rightarrow e'_1 \\ \quad \iota_2{\cdot}y_2 \Rightarrow e'_2 \end{pmatrix} \\ \texttt{where}\ e'_1 &= e_1 \qquad\quad\ \texttt{if}\,x = y_1 \\ e'_1 &= e_1\,[e/x] \quad \texttt{if}\,x \neq y_1 \text{ and } y_1 \notin FV(e) \\ e'_2 &= e_2 \qquad\quad\ \texttt{if}\,x = y_2 \\ e'_2 &= e_2\,[e/x] \quad \texttt{if}\,x \neq y_2 \text{ and } y_2 \notin FV(e) \\ (\texttt{case}\,e'\,\texttt{of}\,\{\})[e/x] &= \texttt{case}\,e'[e/x]\,\texttt{of}\,\{\} \end{aligned}$$

3.2 Laws

3.2.1 Product Laws

$$(\beta) \qquad \langle e_1, e_2\rangle{\cdot}\pi_1 =_\beta e_1 \qquad\qquad \langle e_1, e_2\rangle{\cdot}\pi_2 =_\beta e_2$$

$$\begin{aligned} (\eta) \qquad \langle e{\cdot}\pi_1, e{\cdot}\pi_2\rangle &=_\eta e : \tau_1 \times \tau_2 \\ (\eta) \qquad \langle\rangle &=_\eta e : \texttt{unit} \end{aligned}$$

3.2.2 Sum Laws

$$(\alpha) \qquad \begin{array}{l} \texttt{case}\,e\,\texttt{of} \\ \quad \iota_1{\cdot}x_1 \Rightarrow e_1 \\ \quad \iota_2{\cdot}x_2 \Rightarrow e_2 \end{array} \quad =_\alpha \quad \begin{array}{l} \texttt{case}\,e\,\texttt{of} \\ \quad \iota_1{\cdot}y_1 \Rightarrow (e_1\,[y_1/x_1]) \\ \quad \iota_2{\cdot}y_2 \Rightarrow (e_2\,[y_2/x_2]) \end{array}$$

$$(\beta) \qquad \begin{array}{l} \texttt{case}\,\iota_1{\cdot}e\,\texttt{of} \\ \quad \iota_1{\cdot}x \Rightarrow e_1 \\ \quad \iota_2{\cdot}y \Rightarrow e_2 \end{array} =_\beta e_1\,[e/x] \qquad\qquad \begin{array}{l} \texttt{case}\,\iota_2{\cdot}e\,\texttt{of} \\ \quad \iota_1{\cdot}x \Rightarrow e_1 \\ \quad \iota_2{\cdot}y \Rightarrow e_2 \end{array} =_\beta e_2\,[e/y]$$

$$(\eta) \qquad \begin{array}{l} \texttt{case}\,e\,\texttt{of} \\ \quad \iota_1{\cdot}x \Rightarrow \iota_1{\cdot}x \\ \quad \iota_2{\cdot}y \Rightarrow \iota_1{\cdot}x \end{array} \quad =_\eta \quad e : \tau_1 + \tau_2$$

$$(\eta) \qquad \texttt{case}\,e\,\texttt{of}\,\{\} \quad =_\eta \quad e : \texttt{void}$$

3.3 Static Semantics

$$\tau \in \mathit{Type} ::= \ldots \mid \tau_1 \times \tau_2 \mid \texttt{unit} \mid \tau_1 + \tau_2 \mid \texttt{void}$$

3.3.1 Product Types

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \times I$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e{\cdot}\pi_1 : \tau_1} \times E_1 \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e{\cdot}\pi_2 : \tau_2} \times E_2$$

$$\frac{}{\Gamma \vdash \langle\rangle : \texttt{unit}} \; 1I$$

3.3.2 Sum Types

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \iota_1{\cdot}e : \tau_1 + \tau_2} +I_1 \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \iota_2{\cdot}e : \tau_1 + \tau_2} +I_2$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \quad \Gamma, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}\, e \,\texttt{of}\, \iota_1{\cdot}x \Rightarrow e_1 \mid \iota_2{\cdot}y \Rightarrow e_2 : \tau} +E$$

$$\frac{\Gamma \vdash e : \texttt{void}}{\Gamma \vdash \texttt{case}\, e \,\texttt{of}\, \{\} : \tau} \; 0E$$

3.4 Dynamic Semantics

3.4.1 Call-by-Name

$$E \in \mathit{EvalCxt} ::= \Box \mid \ldots \mid E{\cdot}\pi_i \mid \texttt{case}\, E \,\texttt{of}\, \{\iota_1{\cdot}x \Rightarrow e_1 \mid \iota_2{\cdot}y \Rightarrow e_2\}$$

$$\langle e_1, e_2 \rangle{\cdot}\pi_i \mapsto e_i$$
$$\texttt{case}\, \iota_i{\cdot}e \,\texttt{of}\, \{\iota_1{\cdot}x \Rightarrow e_1 \mid \iota_2{\cdot}y \Rightarrow e_2\} \mapsto e_i[e/x]$$

3.4.2 Call-by-Value

$$V \in \mathit{Value} ::= \ldots \mid \langle V_1, V_2 \rangle \mid \langle\rangle \mid \iota_i{\cdot}V$$
$$E \in \mathit{EvalCxt} ::= \Box \mid \ldots \mid E{\cdot}\pi_i \mid \langle E, e_2 \rangle \mid \langle V_1, E \rangle \mid \texttt{case}\, E \,\texttt{of}\, \{\iota_1{\cdot}x \Rightarrow e_1 \mid \iota_2{\cdot}y \Rightarrow e_2\} \mid \iota_i{\cdot}E$$

$$\begin{aligned}\langle V_1, V_2\rangle\cdot\pi_i &\mapsto V_i\\ \texttt{case}\,\iota_i\cdot V\,\texttt{of}\,\{\iota_1\cdot x \Rightarrow e_1 \mid \iota_2\cdot y \Rightarrow e_2\} &\mapsto e_i[V/x]\end{aligned}$$

Chapter 4

Primitive Recursion

4.1 Syntax

$$e \in \mathit{Expr} ::= \ldots \mid \mathtt{Z} \mid \mathtt{S}(e) \mid \mathtt{rec}\,(e; e_0; x, y.e_1)$$

$$e \in \mathit{Expr} ::= \ldots \mid \mathtt{Z} \mid \mathtt{S}\,e \mid \mathtt{rec}\,e\,\mathtt{as}\,\{\mathtt{Z} \Rightarrow e_0 \mid \mathtt{S}\,x\,\mathtt{with}\,y \Rightarrow e_1\}$$

4.2 Examples

$$\begin{aligned} \mathit{add}\;\; \mathtt{Z}\;\; n &= n \\ \mathit{add}\;\; (\mathtt{S}\,m)\;\; n &= \mathtt{S}(\mathit{add}\;\; m\;\; n) \end{aligned}$$

$$\begin{aligned} \mathit{add} = \lambda m{:}\,\mathtt{nat}.\lambda n{:}\,\mathtt{nat}.\,&\mathtt{rec}\,m\,\mathtt{as} \\ &\mathtt{Z} \qquad\qquad\quad \Rightarrow n \\ &\mathtt{S}\,m'\,\mathtt{with}\,r \Rightarrow \mathtt{S}\,r \end{aligned}$$

$$\begin{aligned} \mathit{mult}\;\; \mathtt{Z}\;\; n &= \mathtt{Z} \\ \mathit{mult}\;\; (\mathtt{S}\,m)\;\; n &= \mathit{add}\;\; n\;\; (\mathit{mult}\;\; m\;\; n) \end{aligned}$$

$$\begin{aligned} \mathit{mult} = \lambda m{:}\,\mathtt{nat}.\lambda n{:}\,\mathtt{nat}.\,&\mathtt{rec}\,m\,\mathtt{as} \\ &\mathtt{Z} \qquad\qquad\quad \Rightarrow \mathtt{Z} \\ &\mathtt{S}\,m'\,\mathtt{with}\,r \Rightarrow \mathit{add}\;\; n\;\; r \end{aligned}$$

$$\begin{aligned} \mathit{pred}\;\; \mathtt{Z} &= \mathtt{Z} \\ \mathit{pred}\;\; (\mathtt{S}\,n) &= n \end{aligned}$$

$$\begin{array}{l} \mathit{pred} = \lambda n{:}\,\mathtt{nat}.\,\mathtt{rec}\,n\,\mathtt{as} \\ \qquad\qquad \begin{array}{ll} \mathtt{Z} & \Rightarrow \mathtt{Z} \\ \mathtt{S}\,n'\,\mathtt{with}\,r & \Rightarrow n' \end{array} \end{array}$$

4.3 Laws

$$(\alpha)\qquad \begin{array}{l} \mathtt{rec}\,e\,\mathtt{as} \\ \quad \begin{array}{ll} \mathtt{Z} & \Rightarrow e_0 \\ \mathtt{S}\,x\,\mathtt{with}\,y & \Rightarrow e_1 \end{array} \end{array} =_\alpha \begin{array}{l} \mathtt{rec}\,e\,\mathtt{as} \\ \quad \begin{array}{ll} \mathtt{Z} & \Rightarrow e_0 \\ \mathtt{S}\,x'\,\mathtt{with}\,y' & \Rightarrow (e_1\,[x'/x, y'/y]) \end{array} \end{array}$$

$$(\beta)\qquad \begin{array}{l} \mathtt{rec}\ \mathtt{Z}\,\mathtt{as} \\ \quad \begin{array}{ll} \mathtt{Z} & \Rightarrow e_0 \\ \mathtt{S}\,x\,\mathtt{with}\,y & \Rightarrow e_1 \end{array} \end{array} =_\beta e_0$$

$$(\beta)\qquad \begin{array}{l} \mathtt{rec}\ \mathtt{S}\,e\,\mathtt{as} \\ \quad \begin{array}{ll} \mathtt{Z} & \Rightarrow e_0 \\ \mathtt{S}\,x\,\mathtt{with}\,y & \Rightarrow e_1 \end{array} \end{array} =_\beta e_1 \left[e/x, \left(\begin{array}{l} \mathtt{rec}\,e\,\mathtt{as} \\ \quad \begin{array}{ll} \mathtt{Z} & \Rightarrow e_0 \\ \mathtt{S}\,x\,\mathtt{with}\,y & \Rightarrow e_1 \end{array} \end{array} \right) /y \right]$$

$$(\eta)\qquad \begin{array}{l} \mathtt{rec}\,e\,\mathtt{as} \\ \quad \begin{array}{ll} \mathtt{Z} & \Rightarrow \mathtt{Z} \\ \mathtt{S}\,x\,\mathtt{with}\,y & \Rightarrow \mathtt{S}\,x \end{array} \end{array} =_\eta e : \mathtt{nat}$$

$$(\eta)\qquad \begin{array}{l} \mathtt{rec}\,e\,\mathtt{as} \\ \quad \begin{array}{ll} \mathtt{Z} & \Rightarrow \mathtt{Z} \\ \mathtt{S}\,x\,\mathtt{with}\,y & \Rightarrow \mathtt{S}\,y \end{array} \end{array} =_\eta e : \mathtt{nat}$$

4.4 Static Semantics

$$\tau \in \mathit{Type} ::= \ldots \mathtt{nat}$$

$$\frac{}{\Gamma \vdash \mathtt{Z} : \mathtt{nat}}\ \mathtt{nat}I_1 \qquad\qquad \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{S}\,e : \mathtt{nat}}\ \mathtt{nat}I_2$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}\,e\,\mathtt{as}\,\{\mathtt{Z} \Rightarrow e_0 \mid \mathtt{S}\,x\,\mathtt{with}\,y \Rightarrow e_1\} : \tau}\ \mathtt{nat}E$$

4.5 Dynamic Semantics

$$E \in \mathit{EvalCxt} ::= \ldots \mid \mathtt{rec}\,E\,\mathtt{as}\,\{\mathtt{Z} \Rightarrow e_0 \mid \mathtt{S}\,x\,\mathtt{with}\,y \Rightarrow e_1\}$$

$$\begin{array}{l}\texttt{rec}\;\texttt{Z}\;\texttt{as}\\ \quad \begin{array}{ll}\texttt{Z} & \Rightarrow e_0\\ \texttt{S}\,x\,\texttt{with}\,y & \Rightarrow e_1\end{array}\end{array} \mapsto e_0$$

$$\begin{array}{l}\texttt{rec}\;\texttt{S}\,e\;\texttt{as}\\ \quad \begin{array}{ll}\texttt{Z} & \Rightarrow e_0\\ \texttt{S}\,x\,\texttt{with}\,y & \Rightarrow e_1\end{array}\end{array} \mapsto e_1 \left[e/x, \left(\begin{array}{l}\texttt{rec}\,e\,\texttt{as}\\ \quad \begin{array}{ll}\texttt{Z} & \Rightarrow e_0\\ \texttt{S}\,x\,\texttt{with}\,y & \Rightarrow e_1\end{array}\end{array} \right) /y \right]$$

Chapter 5

General Recursion

5.1 Untyped Recursion

Recall

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

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

Some syntactic sugar:

$$\begin{aligned} \texttt{rec}\, x = e &= Y\ (\lambda x.e) \\ \texttt{let rec}\, x = e\, \texttt{in}\, e' &= (\lambda x.e')\ (Y\ (\lambda x.e)) \end{aligned}$$

But remember, Y is not well-typed in the simply typed lambda calculus!

Solution: If you wanted recursion in the STLC, then add it explicitly as part of your language.

$$e \in \mathit{Expr} ::= \ldots \mid \texttt{rec}\, x = e$$

$$\begin{array}{lll} (\alpha) & (\texttt{rec}\, x = e) =_\alpha (\texttt{rec}\, y = e[y/x]) & (y \notin FV(e)) \\ (\beta) & (\texttt{rec}\, x = e) \mapsto_\beta e[(\texttt{rec}\, x = e)/x] & \end{array}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \texttt{rec}\, x = e : \tau}$$

5.2 Recursive Types

$$\tau \in \mathit{Type} ::= \ldots \mid \mu\alpha.\tau$$

Equi-recursive: $\mu\alpha.\tau = \tau[(\mu\alpha.\tau)/\alpha]$.
Iso-recursive: $\mu\alpha.\tau \approx \tau[(\mu\alpha.\tau)/\alpha]$. Need these rules.

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

5.3 Syntax

$$e \in \mathit{Expr} ::= \ldots \mid \texttt{fold}\,e \mid \texttt{unfold}\,e$$

$$\begin{aligned} BV(\texttt{fold}\,e) &= BV(e) = BV(\texttt{unfold}\,e) \\ FV(\texttt{fold}\,e) &= FV(e) = FV(\texttt{unfold}\,e) \end{aligned}$$

$$\begin{aligned} (\texttt{fold}\,e')\,[e/x] &= \texttt{fold}\,(e'\,[e/x]) \\ (\texttt{unfold}\,e')\,[e/x] &= \texttt{unfold}\,(e'\,[e/x]) \end{aligned}$$

5.4 Laws

$$\begin{array}{ll} (\beta) & \texttt{unfold}\,(\texttt{fold}\,e) \mapsto_\beta e \\ (\eta) & \texttt{fold}\,(\texttt{unfold}\,e) =_\eta e : \mu\alpha.\tau \end{array}$$

5.5 Encodings

$$\begin{aligned} \texttt{nat} &\approx \texttt{unit} + \texttt{nat} \\ \texttt{nat} &= \mu\alpha.\,\texttt{unit} + \alpha \\ \texttt{Z} &= \texttt{fold}(\texttt{left}\,\langle\rangle) \\ \texttt{S}\,e &= \texttt{fold}(\texttt{right}\,e) \end{aligned}$$

$$\begin{aligned} \texttt{list}\,\tau &\approx \texttt{unit} + (\tau \times \texttt{list}\,\tau) \\ \texttt{list}\,\tau &= \mu\alpha.\,\texttt{unit} + (\tau \times \alpha) \\ \texttt{nil} &= \texttt{fold}(\texttt{left}\,\langle\rangle) \\ \texttt{cons}\,e\ e' &= \texttt{fold}(\texttt{right}(e, e')) \end{aligned}$$

$$\begin{aligned} \texttt{tree}\,\tau &\approx \tau + (\texttt{tree}\,\tau \times \texttt{tree}\,\tau) \\ \texttt{tree}\,\tau &= \mu\alpha.\tau + (\alpha \times \alpha) \\ \texttt{leaf}\,e &= \texttt{fold}(\texttt{left}\,e) \\ \texttt{branch}\,e_1\ e_2 &= \texttt{fold}(\texttt{right}(e_1, e_2)) \end{aligned}$$

5.6 Typed Recursion

$$\begin{aligned}
&\omega : (\mu\alpha.(\alpha \to \tau)) \to \tau\\
&\omega = \lambda x.(\texttt{unfold}\,x)\ x\\
&\Omega : \tau\\
&\Omega = \omega\ (\texttt{fold}\,\omega)
\end{aligned}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mu\alpha.\alpha \to A \vdash x : \mu\alpha.\alpha \to A}\,\textit{Var}}{x : \mu\alpha.\alpha \to A \vdash \texttt{unfold}\,x : (\mu\alpha.\alpha \to A) \to A}\,\mu E \qquad \dfrac{}{x : \mu\alpha.\alpha \to A \vdash x : \mu\alpha.\alpha \to A}\,\textit{Var}}{x : \mu\alpha.\alpha \to A \vdash (\texttt{unfold}\,x)\ x : A}\,{\to}E}{\bullet \vdash \lambda x.(\texttt{unfold}\,x)\ x : (\mu\alpha.\alpha \to A) \to A}\,{\to}I$$

$$\dfrac{\begin{array}{c}\vdots\\ \bullet \vdash \omega : (\mu\alpha.\alpha \to A) \to A\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \bullet \vdash \omega : (\mu\alpha.\alpha \to A) \to A\end{array}}{\bullet \vdash \texttt{fold}\,\omega : \mu\alpha.\alpha \to A}\,\mu I}{\bullet \vdash \omega\ (\texttt{fold}\,\omega) : A}\,{\to}E$$

$$\begin{aligned}
&Y : (\tau \to \tau) \to \tau\\
&Y = \lambda f.(\lambda x.f\ (\texttt{unfold}\,x\ x))\ (\texttt{fold}\,(\lambda x.f\ (\texttt{unfold}\,x\ x)))
\end{aligned}$$

$$\dfrac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \texttt{let}\,x = e\,\texttt{in}\,e' : \tau'}\,\textit{Let} \qquad\qquad \dfrac{\Gamma, x : \tau \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \texttt{let rec}\,x = e\,\texttt{in}\,e' : \tau'}\,\textit{LetRec}$$

$$\dfrac{\dfrac{\Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \lambda x.e' : \tau \to \tau'}\,{\to}I \qquad \Gamma \vdash e : \tau}{\Gamma \vdash (\lambda x.e')\ e : \tau'}\,{\to}E$$

$$\dfrac{\dfrac{\Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \lambda x.e' : \tau \to \tau'}\,{\to}I \qquad \dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash Y : (\tau \to \tau) \to \tau\end{array} \quad \dfrac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau \to \tau}\,{\to}I}{\Gamma \vdash Y\ (\lambda x.e) : \tau}\,{\to}E}{\Gamma \vdash (\lambda x.e')\ (Y\ (\lambda x.e)) : \tau'}\,{\to}E$$

Chapter 6

Polymorphism

6.1 Polymorphic Types

$$\tau \in \mathit{Type} ::= \ldots \mid \forall\alpha.\tau \mid \exists\alpha.\tau$$

Hypothetical judgements are generalized to also include an environment of free type variables $\Theta = \alpha, \beta, \ldots$, and are written as $\Theta; \Gamma \vdash e : \tau$. Δ 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 $\Theta; \Gamma \vdash \tau : \star$ means $FV(\tau) \subseteq \Theta$.

$$\frac{}{\Theta, \alpha \vdash \alpha : \star}\ \mathit{Var} \qquad \frac{\Theta \vdash \tau_1 : \star \quad \Theta \vdash \tau_2 : \star}{\Theta \vdash \tau_1 \to \tau_2 : \star}\ {\to}T$$

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

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

$$\frac{\Theta \vdash \tau' : \star \quad \Theta; \Gamma \vdash e : \tau\,[\tau'/\alpha]}{\Theta; \Gamma \vdash \langle \tau', e \rangle : \exists\alpha.\tau}\ \exists I$$

$$\frac{\Theta; \Gamma \vdash e : \exists\alpha.\tau \quad \Theta, \alpha; \Gamma, x : \tau \vdash e' : \tau' \quad \Theta \vdash \tau' : \star}{\Gamma \vdash \mathtt{open}\, e\, \mathtt{as}\ \langle \alpha, x \rangle \Rightarrow e' : \tau'}\ \exists E$$

$$\frac{\Theta \vdash \Gamma, x : \tau}{\Theta; \Gamma, x : \tau \vdash x : \tau}\ \mathit{Var}$$

$$\frac{}{\Theta \vdash \bullet} \qquad \frac{\Theta \vdash \Gamma \quad \Theta \vdash \tau : \star}{\Theta \vdash \Gamma, x : \tau}$$

Lemma 6.1 (Scope of Types)**.** *If $\Theta \vdash \tau : \star$ is derivable, then $FV(\tau) \subseteq \Theta$. If $\Theta; \Gamma \vdash e : \tau$ is derivable then both $\Theta \vdash \Gamma$ and $\Theta \vdash \tau : \star$ are, too.*

Lemma 6.2. *By induction on the derivations of $\Theta; \Gamma \vdash e : \tau$ and $\Theta \vdash \tau : \star$, respectively.*

6.2 Syntax

$$\begin{aligned} e \in \mathit{Expr} ::= {} & \dots \\ & \mid \Lambda\alpha.e \mid e\ \tau \\ & \mid \langle \tau, e \rangle \mid \texttt{open}\, e\, \texttt{as}\ \langle \alpha, x \rangle \Rightarrow e' \end{aligned}$$

6.3 Laws

$$\begin{array}{lll} (\alpha) & \forall\alpha.\tau =_\alpha \forall\beta.(\tau\,[beta/\alpha]) & (\beta \notin FV(\tau)) \\ (\alpha) & \exists\alpha.\tau =_\alpha \exists\beta.(\tau\,[\beta/\alpha]) & (\beta \notin FV(\tau)) \end{array}$$

$$\begin{array}{lll} (\alpha) & \Lambda\alpha.e =_\alpha \Lambda\beta.(e\,[\beta/\alpha]) & (\beta \notin FV(e)) \\ (\beta) & (\Lambda\alpha.e)\ \tau =_\beta e\,[\tau/\alpha] & \\ (\eta) & \Lambda\alpha.(e\ \alpha) =_\eta e : \forall\alpha.\tau & \texttt{if}\ \alpha \notin FV(e) \end{array}$$

$$\begin{array}{ll} (\alpha) & \texttt{open}\, e\, \texttt{as}\ \langle \alpha, x \rangle \Rightarrow e' =_\alpha \texttt{open}\, e\, \texttt{as}\ \langle \alpha', x' \rangle \Rightarrow (e'\,[\alpha'/\alpha]\,[x'/x]) \\ (\beta) & \texttt{open}\ \langle \tau, e \rangle\ \texttt{as}\ \langle \alpha, x \rangle \Rightarrow e' =_\beta e'\,[\tau/\alpha]\,[e/x] \\ (\eta) & \texttt{open}\, e\, \texttt{as}\ \langle \alpha, x \rangle \Rightarrow \langle \alpha, x \rangle =_\eta e : \exists\alpha.\tau \end{array}$$

6.4 Operational Semantics

For functions, foralls, and exists.

6.4.1 Call-by-Name

$$E \in \mathit{EvalCxt} ::= \Box \mid E\ e \mid E\ \tau \mid \texttt{open}\, E \,\texttt{as}\, (\alpha, x) \Rightarrow e$$

$$\begin{aligned}(\lambda x.e)\ e' &\mapsto e[e'/x]\\ (\Lambda\alpha.e)\ \tau &\mapsto e[\tau/\alpha]\\ \texttt{open}\ \langle\tau, e\rangle'\,\texttt{as}\,(\alpha, x) \Rightarrow e &\mapsto e[\tau/\alpha][e'/x]\end{aligned}$$

6.4.2 Call-by-Value

$$\begin{aligned}V \in \mathit{Value} &::= x \mid \lambda x.e \mid \Lambda\alpha.e \mid \langle\tau, V\rangle\\ E \in \mathit{EvalCxt} &::= \Box \mid E\ e \mid V\ E \mid E\ \tau \mid \texttt{open}\, E \,\texttt{as}\, (\alpha, x) \Rightarrow e \mid \langle\tau, E\rangle\end{aligned}$$

$$\begin{aligned}(\lambda x.e)\ V &\mapsto e[V/x]\\ (\Lambda\alpha.e)\ \tau &\mapsto e[\tau/\alpha]\\ \texttt{open}\ \langle\tau, V\rangle\,\texttt{as}\,(\alpha, x) \Rightarrow e &\mapsto e[\tau/\alpha][V/x]\end{aligned}$$

Chapter 7

Encodings

7.1 Untyped Encodings

7.1.1 Booleans

$$\mathit{IfThenElse} = \lambda x.\lambda t.\lambda f.x\ t\ f$$

$$\begin{aligned} \mathit{True} &= \lambda t.\lambda f.t \\ \mathit{False} &= \lambda t.\lambda f.f \end{aligned}$$

$$\begin{aligned} \mathit{And} &= \lambda x.\lambda y.\mathit{IfThenElse}\ x\ y\ \mathit{False} \\ \mathit{Or} &= \lambda x.\lambda y.\mathit{IfThenElse}\ x\ \mathit{True}\ y \\ \mathit{Not} &= \lambda x.\mathit{IfThenElse}\ x\ \mathit{False}\ \mathit{True} \end{aligned}$$

7.1.2 Sums

$$\mathit{Case} = \lambda i.\lambda l.\lambda r.i\ l\ r$$

$$\begin{aligned} \mathit{Inl} &= \lambda x.\lambda l.\lambda r.l\ x \\ \mathit{Inr} &= \lambda x.\lambda l.\lambda r.r\ x \end{aligned}$$

7.1.3 Products

$$\mathit{Pair} = \lambda x.\lambda y.\lambda p.p\ x\ y$$

$$\begin{aligned} \mathit{Fst} &= \lambda x.\lambda y.x \\ \mathit{Snd} &= \lambda x.\lambda y.y \end{aligned}$$

7.1.4 Numbers

$$\mathit{Iter} = \lambda n.\lambda z.\lambda s.n\ z\ s$$

$$\begin{aligned} \mathit{Zero} &= \lambda z.\lambda s.z \\ \mathit{Suc} &= \lambda n.\lambda z.\lambda s.s\ (n\ z\ s) \end{aligned}$$

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

7.1.5 Lists

$$\mathit{Fold} = \lambda l.\lambda n.\lambda c.l\ n\ c$$

$$\begin{aligned} \mathit{Nil} &= \lambda n.\lambda c.n \\ \mathit{Cons} &= \lambda x.\lambda l.\lambda n.\lambda c.c\ x\ (l\ n\ c) \end{aligned}$$

7.2 Typed Encodings

7.2.1 Booleans

$$\mathit{Bool} = \forall\delta.\delta \to \delta \to \delta$$

$$\begin{aligned} \mathit{IfThenElse} &: \forall\delta.\mathit{Bool} \to \delta \to \delta \to \delta \\ \mathit{IfThenElse} &= \Lambda\delta.\lambda x{:}\mathit{Bool}.\lambda t{:}\delta.\lambda f{:}\delta.x\ \delta\ t\ f \end{aligned}$$

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

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

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

7.2.2 Sums

$$\mathit{Sum}\ \tau_1\ \tau_2 = \forall\delta.(\tau_1 \to \delta) \to (\tau_2 \to \delta) \to \delta$$

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

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

7.2.3 Products

$$\mathit{Prod}\ \tau_1\ \tau_2 = \forall\delta.(\tau_1 \to \tau_2 \to \delta) \to \delta$$

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

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

7.2.4 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:

$$\begin{aligned} \tau \in \mathit{Type} &::= \dots \mid \lambda\alpha.\tau \mid \tau_1\ \tau_2 \\ k \in \mathit{Kind} &::= \star \mid k \to k' \end{aligned}$$

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 $\tau_1\ \tau_2 : \star$ (when $\tau_1 : \star$) 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:

$$\frac{\Gamma, \alpha : k \vdash \tau : k'}{\Gamma \vdash \lambda\alpha.\tau : k \to k'} \to I^2 \qquad\qquad \frac{\Gamma \vdash \tau_1 : k \to k' \quad \Gamma \vdash \tau_2 : k}{\Gamma \vdash \tau_1\ \tau_2 : k'} \to E^2$$

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 $\mu\alpha.\tau$, $\forall\alpha.\tau$ and $\exists\alpha.\tau$ then type equality incorporates α-equivalence which is easily decidable. When $\lambda\alpha.\tau$ and $\tau_1\ \tau_2$ 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.

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

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

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

7.2.5 Numbers

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

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

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

7.2.6 Lists

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

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

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

Chapter 8

Linear Logic

8.1 Connectives

$$\begin{aligned} \alpha, \beta \in \mathit{PropVariable} &::= \ldots \\ \tau \in \mathit{Proposition} &::= \tau_1 \otimes \tau_2 \mid \tau_1 \oplus \tau_2 \mid 1 \mid 0 \mid \tau_1 \,\&\, \tau_2 \mid \tau_1 \parr \tau_2 \mid \top \mid \bot \mid !\tau \mid ?\tau \end{aligned}$$

8.2 Classification

The non-exponential connectives for building propositions can be divided into two different ways.

- Additive vs. Multiplicative
  - Additive connectives ($\oplus$, 0, &, $\top$) "share" their environment among multiple premises.
  - Multiplicative connectives ($\otimes$, 1, $\parr$, $\bot$) "divide" their environment between multiple premises.
- Positive vs. Negative
  - Positive connectives ($\otimes$, $\oplus$, 0, 1) have reversible left rules.
  - Negative connectives (&, $\parr$, $\top$, $\bot$) have reversible right rules.

8.3 Inference Rules

The judgements are of the form $\tau_1, \tau_2, \ldots, \tau_n \vdash \tau'_1, \tau'_2 \ldots, \tau'_m$, which can be read as "if ALL of τ_1 AND τ_2 AND $\ldots \tau_n$ are true then ONE of τ'_1 OR τ'_2 OR $\ldots \tau'_m$ is true". Both lists to the left and right of $\vdash$ are unordered. I'll write Γ to the left of $\vdash$ and Δ to the right.

8.3.1 Core Identity Rules

$$\dfrac{}{\tau \vdash \tau}\ Id \qquad\qquad \dfrac{\Gamma \vdash \tau, \Delta \quad \Gamma', \tau \vdash \Delta'}{\Gamma', \Gamma \vdash \Delta', \Delta}\ Cut$$

8.3.2 Additive Conjunction

$$\dfrac{\Gamma \vdash \tau_1, \Delta \quad \Gamma \vdash \tau_2, \Delta}{\Gamma \vdash \tau_1 \mathbin{\&} \tau_2, \Delta}\ \&R \qquad\qquad \dfrac{\Gamma, \tau_i \vdash \Delta \quad (i \in \{1,2\})}{\Gamma, \tau_1 \mathbin{\&} \tau_2 \vdash \Delta}\ \&L$$

$$\dfrac{}{\Gamma \vdash \top, \Delta}\ \top R \qquad\qquad \text{no } \top L \text{ rule}$$

8.3.3 Multiplicative Conjunction

$$\dfrac{\Gamma_1 \vdash \tau_1, \Delta_1 \quad \Gamma_2 \vdash \tau_2, \Delta_2}{\Gamma_1, \Gamma_2 \vdash \tau_1 \otimes \tau_2, \Delta_1, \Delta_2}\ \otimes R \qquad\qquad \dfrac{\Gamma, \tau_1, \tau_2 \vdash \Delta}{\Gamma, \tau_1 \otimes \tau_2 \vdash \Delta}\ \otimes L$$

$$\dfrac{}{\bullet \vdash 1}\ 1R \qquad\qquad \dfrac{\Gamma \vdash \Delta}{\Gamma, 1 \vdash \Delta}\ 1L$$

8.3.4 Additive Disjunction

$$\dfrac{\Gamma \vdash \tau_i, \Delta \quad (i \in \{1,2\})}{\Gamma \vdash \tau_1 \oplus \tau_2, \Delta}\ \oplus R \qquad\qquad \dfrac{\Gamma \vdash \tau_1, \Delta \quad \Gamma \vdash \tau_2, \Delta}{\Gamma \vdash \tau_1 \mathbin{\&} \tau_2, \Delta}\ \oplus L$$

$$\text{no } 0R \text{ rule} \qquad\qquad \dfrac{}{\Gamma, 0 \vdash \Delta}\ 0L$$

8.3.5 Multiplicative Disjunction

$$\dfrac{\Gamma \vdash \tau_1, \tau_2, \Delta}{\Gamma \vdash \tau_1 \parr \tau_2, \Delta}\ \parr R \qquad\qquad \dfrac{\Gamma_1 \vdash \tau_1, \Delta_1 \quad \Gamma_2 \vdash \tau_2, \Delta_2}{\Gamma_1, \Gamma_2, \tau_1 \parr \tau_2 \vdash \Delta_1, \Delta_2}\ \parr L$$

$$\dfrac{\Gamma \vdash \Delta}{\Gamma \vdash \bot, \Delta}\ \bot R \qquad\qquad \dfrac{}{\bot \vdash \bullet}\ \bot L$$

8.3.6 Exponentials

$$\dfrac{!\Gamma \vdash \tau, ?\Delta}{!\Gamma \vdash !\tau, ?\Delta}\ !R \qquad\qquad \dfrac{\Gamma, \tau \vdash \Delta}{\Gamma, !\tau \vdash \Delta}\ !L\ (\textit{a.k.a. } !\textit{Dereliction})$$

$$\dfrac{\Gamma, !\tau \vdash \Delta}{\Gamma \vdash \Delta}\ !W \qquad\qquad \dfrac{\Gamma, !\tau, !\tau \vdash \Delta}{\Gamma, !\tau \vdash \Delta}\ !C$$

$$\frac{!\Gamma, \tau \vdash ?\Delta}{!\Gamma, ?\tau \vdash ?\Delta}\ ?L \qquad \frac{\Gamma \vdash \tau, \Delta}{\Gamma \vdash ?\tau, \Delta}\ ?R \text{ (\textit{a.k.a.} ?\textit{Dereliction})}$$

$$\frac{\Gamma \vdash ?\tau, \Delta}{\Gamma \vdash \Delta}\ ?W \qquad \frac{\Gamma \vdash ?\tau, ?\tau, \Delta}{\Gamma \vdash ?\tau, \Delta}\ ?C$$

8.4 Examples

All great linear logic introductions talk about food, so here are some culinary allegories for remembering the difference between linear connectives.

Suppose I'm a cook at a diner, and for lunch we're making burgers and fries. I know how to make a burger out of beef and a bun, written $beef, bun \vdash burger$. I know how to make fries out of potatoes and oil, written $potato, oil \vdash fries$. So, I can make a single *burger and fries* meal by cooking both and putting them together on the same plate. Clearly, to make a burger and fries plate, I need all the ingredients for both parts, so

$$\frac{beef, bun \vdash burger \quad potato, oil \vdash fries}{beef, bun, potato, oil \vdash burger \otimes fries}\ \otimes R$$

A customer who orders a $burger \otimes fries$ receives *both* a burger and a pile of fries, and I expect them to eat it all or I will be very cross at the waste.

For the side, I'll make a soup of the day, which is usually either tomato or chicken noodle soup. Tomato soup is made from tomatoes and cream, written $tomato, cream \vdash tomato\ soup$. Chicken noodle is made from chicken and noodles, written $chicken, noodle \vdash chicken\ noodle\ soup$. As the cook, I get to pick which soup to make, depending on the ingredients at hand. Today, I happen to have tomato and cream, so tomato soup it is

$$\frac{tomato, cream \vdash tomato\ soup}{tomato, cream \vdash tomato\ soup \oplus chicken\ noodle\ soup}\ \oplus R$$

The customer doesn't get any choice of the soup of the day, they just have to take what they get and be happy with whichever one it happens to be that day.

For dessert, I can make one of two fruit-based treats. I can make a fruit salad out of strawberries and bananas, written $strawberry, banana \vdash fruit\ salad$. I can also make a smoothie out of the same ingredients, written $strawberry, banana \vdash smoothie$. As a flexible cook, I'll give the customer the choice of which of the two desserts they want. But no matter which they choose, I use the same stock of ingredients, so

$$\frac{strawberry, banana \vdash fruit\ salad \quad strawberry, banana \vdash smoothie}{strawberry, banana \vdash fruit\ salad \mathbin{\&} smoothie}\ \&R$$

I am offering the *potential* of a fruit salad and a smoothing, and the customer has to choose only one of the two options. If they choose to get a fruit salad, then

I've used all my ingredients and they pass up the opportunity of a smoothie, and vice versa.

Cooking for a single customer, including a main dish, side, and desert, can then be written all together with one big $\otimes$,

$$\begin{aligned}
main &= burger \otimes fries \\
soup &= tomato\ soup \oplus chicken\ noodle\ soup \\
dessert &= fruit\ salad \mathbin{\&} smoothie \\
lunch &= main \otimes soup \otimes dessert \\
ingredients &= beef, bun, potato, oil, tomato, cream, strawberry, banana \\
ingredients &\vdash lunch
\end{aligned}$$

What about par ($⅋$)? That represents a cook serving multiple customers at once during the same lunch hour. In order to cook one lunch order, I need a single set of the ingredients. If I need to cook up two, then I'd need two sets of the same ingredients, and three for three, and so on. So, if I have two customers both wanting a lunch, I can serve them "in parallel" with two sets of the standard ingredients,

$$\dfrac{\dfrac{\dfrac{\vdots}{ingredients \vdash lunch} \quad \dfrac{\vdots}{ingredients \vdash lunch}}{ingredients, ingredients, ingredients \vdash lunch, lunch}\ Mix}{ingredients, ingredients \vdash lunch ⅋ lunch}\ ⅋R$$

That extra rule,

$$\dfrac{\Gamma \vdash \Delta \quad \Gamma' \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ Mix$$

is like a *Cut* without the proposition being cut out. It lets us separate out two independent derivations from each other, doing them both in parallel with one another, within a single umbrella derivation. Whereas a *Cut* corresponds to connecting a lemma with a proof that uses that lemma, or connecting an implementation of a library function with a program that uses that library function, a *Mix* just puts together two independent proofs into one common collection of results, or runs two non-interacting/non-communicating subroutines in the same program.

Some extra properties differentiating the two versions of conjunction and disjunction:

- $\tau \dashv\vdash \tau \mathbin{\&} \tau$ but $\tau \not\dashv\vdash \tau \otimes \tau$.
- $\tau, \tau \dashv\vdash \tau \otimes \tau$ (using *Mix* for $\tau \otimes \tau \vdash \tau, \tau$) but $\tau, \tau \not\dashv\vdash \tau \mathbin{\&} \tau$.
- $\tau \oplus \tau \dashv\vdash \tau$ but $\tau ⅋ \tau \not\dashv\vdash \tau$.
- $\tau ⅋ \tau \dashv\vdash \tau, \tau$ (using *Mix* for $\tau, \tau \vdash \tau ⅋ \tau$) but $\tau \oplus \tau \not\dashv\vdash \tau$.

8.5 Negation and Duality

The negation *operation* (*not* connective!) in linear logic is defined by induction on proposition, following familiar De Morgan-like rules:

$$\begin{array}{rclcrcl}
(\tau_1 \oplus \tau_2)^\perp &=& \tau_1^\perp \mathbin{\&} \tau_2^\perp && (\tau_1 \mathbin{\&} \tau_2)^\perp &=& \tau_1^\perp \oplus \tau_2^\perp \\
(\tau_1 \otimes \tau_2)^\perp &=& \tau_1^\perp \parr \tau_2^\perp && (\tau_1 \parr \tau_2)^\perp &=& \tau_1^\perp \otimes \tau_2^\perp \\
0^\perp &=& \top && \top^\perp &=& 0 \\
1^\perp &=& \perp && \perp^\perp &=& 1 \\
(!\tau)^\perp &=& ?\tau^\perp && (?\tau)^\perp &=& !\tau^\perp
\end{array}$$

Linear implication can be defined in terms of the negation operation and multiplicative disjunction:

$$\tau_1 \multimap \tau_2 = \tau_1^\perp \parr \tau_2$$

This negation operation is *involutive* by definition.

Theorem 8.1. *For all propositions τ, $\tau^{\perp\perp} = \tau$.*

Proof. By induction on the syntax of τ. □

Combined with nice symmetries, this gives linear logic a deep sense of duality.

Corollary 8.1 (Logical duality). $\Gamma \vdash \Delta$ *is provable if and only if* $\Delta^\perp \vdash \Gamma^\perp$ *is provable.*

Duality means that the following two inversion rules *should* be included in linear logic:

$$\frac{\Gamma, \tau \vdash \Delta}{\Gamma \vdash \tau^\perp, \Delta}\,InvR \qquad\qquad \frac{\Gamma \vdash \tau, \Delta}{\Gamma, \tau^\perp \vdash \Delta}\,InvL$$

You need these inversion rules in order for the encoding of linear implication $\multimap$ in terms of multiplicative disjunction $\parr$ to mean what you want it to mean. For example, it should be the case that the identity map $\bullet \vdash \tau \multimap \tau$ is derivable. This is because the linear implication $\tau \multimap \tau$ is defined as $\tau^\perp \parr \tau$, and the following derivation proves this proposition using an inversion rule along with the right rule for $\parr$:

$$\cfrac{\cfrac{\cfrac{}{\tau \vdash \tau}\,Id}{\bullet \vdash \tau^\perp, \tau}\,InvR}{\bullet \vdash \tau^\perp \parr \tau}\,\parr R$$

Because of this duality, and to cut down on some of the extra repetition caused by duplicating all the connectives, you sometimes see linear logic presented in a *one-sided* style. Instead of putting propositions on both sides of the turnstyle like $\Gamma \vdash \Delta$, they can all be lumped together on the right like $\vdash \Gamma^{\perp}, \Delta$ (or dually on the left like $\Gamma, \Delta^{\perp} \vdash$, but this is not common). Doing so means you only need to write down half as many logical rules for the connectives by only using the right-hand ones. The only thing that needs to change are the cut and identity rules whose presentation was inherently two-sided. The one-sided version of the cut and identity rules are:

$$\frac{}{\vdash \tau, \tau^{\perp}}\ Id \qquad\qquad \frac{\vdash \tau, \Gamma \quad \vdash \tau^{\perp}, \Delta}{\vdash \Gamma, \Delta}\ Cut$$

Chapter 9

A Linear Lambda Calculus

9.1 Asymmetrical, Intuitionistic, Linear Logic

$$\tau \in \mathit{Proposition} ::= \top \mid 0 \mid 1 \mid \top \mid \tau_1 \multimap \tau_2 \mid \tau_1 \,\&\, \tau_2 \mid \tau_1 \oplus \tau_2 \mid \tau_1 \otimes \tau_2 \mid\, !\tau$$

Judgements will have the asymmetric form $\tau_1, \tau_2, \ldots \tau_n; \tau_1', \tau_2', \ldots, \tau_m' \vdash \tau''$ where the outer $\tau_1 \ldots \tau_n$ are *non-linear* propositions and the inner $\tau_1' \ldots \tau_m'$ are *linear* resources. We will write this judgement as $\Gamma; \Delta \vdash \tau$.

$$\dfrac{}{\Gamma; \tau \vdash \tau}\ Id^1 \qquad \dfrac{}{\Gamma, \tau; \bullet \vdash \tau}\ Id^* \qquad \dfrac{\Gamma; \Delta \vdash \tau \quad \Gamma; \Delta', \tau \vdash \tau'}{\Gamma; \Delta, \Delta' \vdash \tau'}\ Cut$$

$$\dfrac{\Gamma; \Delta, \tau_1 \vdash \tau_2}{\Gamma; \Delta \vdash \tau_1 \multimap \tau_2}\ {\multimap} I \qquad \dfrac{\Gamma; \Delta_1 \vdash \tau_1 \multimap \tau_2 \quad \Gamma; \Delta_2 \vdash \tau_1}{\Gamma; \Delta_1, \Delta_2 \vdash \tau_2}\ {\multimap} E$$

$$\dfrac{\Gamma; \Delta \vdash \tau_1 \quad \Gamma; \Delta \vdash \tau_2}{\Gamma; \Delta \vdash \tau_1 \,\&\, \tau_2}\ \&I \qquad \dfrac{\Gamma; \Delta \vdash \tau_1 \,\&\, \tau_2}{\Gamma; \Delta \vdash \tau_i}\ \&E$$

$$\dfrac{\Gamma; \Delta \vdash \tau_i}{\Gamma; \Delta \vdash \tau_1 \oplus \tau_2}\ \oplus I \qquad \dfrac{\Gamma; \Delta \vdash \tau_1 \oplus \tau_2 \quad \Gamma; \Delta', \tau_1 \vdash \tau' \quad \Gamma; \Delta', \tau_2 \vdash \tau'}{\Gamma; \Delta, \Delta' \vdash \tau'}\ \oplus E$$

$$\dfrac{\Gamma; \Delta_1 \vdash \tau_1 \quad \Gamma; \Delta_1 \vdash \tau_2}{\Gamma; \Delta_1, \Delta_2 \vdash \tau_1 \otimes \tau_2}\ \otimes I \qquad \dfrac{\Gamma; \Delta \vdash \tau_1 \otimes \tau_2 \quad \Gamma; \Delta', \tau_1, \tau_2 \vdash \tau'}{\Gamma; \Delta, \Delta' \vdash \tau'}\ \otimes E$$

$$\dfrac{}{\Gamma; \Delta \vdash \top}\ \top I \qquad \dfrac{\Gamma; \Delta \vdash 0}{\Gamma; \Delta \vdash \tau}\ 0E \qquad \dfrac{}{\Gamma; \bullet \vdash 1}\ 1I \qquad \dfrac{\Gamma; \Delta \vdash 1 \quad \Gamma; \Delta' \vdash \tau'}{\Gamma; \Delta, \Delta' \vdash \tau'}\ 1E$$

$$\dfrac{\Gamma; \bullet \vdash \tau}{\Gamma; \bullet \vdash\, !\tau}\ !I \qquad \dfrac{\Gamma; \Delta \vdash\, !\tau \quad \Gamma, \tau; \Delta' \vdash \tau'}{\Gamma; \Delta, \Delta' \vdash \tau'}\ !E$$

Admissible rules:

$$\dfrac{\Gamma; \Delta \vdash \tau'}{\Gamma, \tau; \Delta \vdash \tau'}\ !W \qquad \dfrac{\Gamma, \tau, \tau; \Delta \vdash \tau'}{\Gamma, \tau; \Delta \vdash \tau'}\ !C$$

9.2 A Resource-Sensitive Lambda Calculus

$$
\begin{aligned}
e \in \mathit{Expr} ::= {} & x \mid \mathtt{let}\, x = e \,\mathtt{in}\, e' \\
& \mid \lambda x.e \mid e\ e' \\
& \mid \{\pi_1 \Rightarrow e_1 \mid \pi_2 \Rightarrow e_2\} \mid e{\cdot}\pi_i \\
& \mid \iota_i{\cdot}e \mid \mathtt{case}\, e \,\mathtt{of}\, \{\iota_1{\cdot}x \Rightarrow e_1' \mid \iota_2{\cdot}y \Rightarrow e_2'\} \\
& \mid \langle e_1, e_2\rangle \mid \mathtt{case}\, e \,\mathtt{of}\ \langle x, y\rangle \Rightarrow e' \\
& \mid \{\} \mid \mathtt{case}\, e\ \{\} \mid \langle\rangle \mid \mathtt{case}\, e \,\mathtt{of}\ \langle\rangle \Rightarrow e' \\
& \mid \mathtt{many}\,(e) \mid \mathtt{case}\, e \,\mathtt{of}\ \mathtt{many}\,(x) \Rightarrow e'
\end{aligned}
$$

$$
\frac{}{\Gamma; x : \tau \vdash x : \tau}\ \mathit{Id}^1 \qquad
\frac{}{\Gamma, x : \tau; \bullet \vdash x : \tau}\ \mathit{Id}^* \qquad
\frac{\Gamma; \Delta \vdash e : \tau \quad \Gamma; \Delta, x : \tau \vdash e' : \tau'}{\Gamma; \Delta, \Delta' \vdash \mathtt{let}\, x = e \,\mathtt{in}\, e' : \tau'}\ \mathit{Cut}
$$

$$
\frac{\Gamma; \Delta, x : \tau_1 \vdash e : \tau_2}{\Gamma; \Delta \vdash \lambda x.e : \tau_1 \multimap \tau_2}\ {\multimap} I \qquad
\frac{\Gamma; \Delta_1 \vdash e : \tau_1 \multimap \tau_2 \quad \Gamma; \Delta_2 \vdash e_1 : \tau_1}{\Gamma; \Delta_1, \Delta_2 \vdash e\ e_1 : \tau_2}\ {\multimap} E
$$

$$
\frac{\Gamma; \Delta \vdash e_1 : \tau_1 \quad \Gamma; \Delta \vdash e_2 : \tau_2}{\Gamma; \Delta \vdash \{\pi_1 \Rightarrow e_1 \mid \pi_2 \Rightarrow e_2\} : \tau_1 \,\&\, \tau_2}\ \&I \qquad
\frac{\Gamma; \Delta \vdash e : \tau_1 \,\&\, \tau_2}{\Gamma; \Delta \vdash e{\cdot}\pi_i : \tau_i}\ \&E
$$

$$
\frac{\Gamma; \Delta \vdash e : \tau_i}{\Gamma; \Delta \vdash \iota_i{\cdot}e : \tau_1 \oplus \tau_2}\ \oplus I \qquad
\frac{\Gamma; \Delta \vdash e : \tau_1 \oplus \tau_2 \quad \Gamma; \Delta', x : \tau_1 \vdash e_1' : \tau' \quad \Gamma; \Delta', y : \tau_2 \vdash e_2' : \tau'}{\Gamma; \Delta, \Delta' \vdash \mathtt{case}\, e \,\mathtt{of}\, \{\iota_1{\cdot}x \Rightarrow e_1' \mid \iota_2{\cdot}y \Rightarrow e_2'\} : \tau'}\ \oplus E
$$

$$
\frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \quad \Gamma; \Delta_2 \vdash e_2 : \tau_2}{\Gamma; \Delta_1, \Delta_2 \vdash \langle e_1, e_2\rangle : \tau_1 \otimes \tau_2}\ \otimes I \qquad
\frac{\Gamma; \Delta \vdash e : \tau_1 \otimes \tau_2 \quad \Gamma; \Delta', x : \tau_1, y : \tau_2 \vdash e' : \tau'}{\Gamma; \Delta, \Delta' \vdash \mathtt{case}\, e \,\mathtt{of}\ \langle x, y\rangle \Rightarrow e' : \tau'}\ \otimes E
$$

$$
\frac{}{\Delta \vdash \{\} : \top}\ \top I \qquad
\frac{\Gamma; \Delta \vdash e : 0}{\Gamma; \Delta \vdash \mathtt{case}\, e \,\mathtt{of}\, \{\} : \tau'}\ 0E
$$

$$
\frac{}{\Gamma; \bullet \vdash \langle\rangle : 1}\ 1I \qquad
\frac{\Gamma; \Delta \vdash e : 1 \quad \Gamma; \Delta' \vdash e' : \tau'}{\Gamma; \Delta, \Delta' \vdash \mathtt{case}\, e \,\mathtt{of}\ \langle\rangle \Rightarrow e' : \tau'}\ 1E
$$

$$
\frac{\Gamma; \bullet \vdash e : \tau}{\Gamma; \bullet \vdash \mathtt{many}\,(e) : !\tau}\ !I \qquad
\frac{\Gamma; \Delta \vdash e : !\tau \quad \Gamma, x : \tau; \Delta' \vdash e' : \tau'}{\Gamma; \Delta, \Delta' \vdash \mathtt{case}\, e \,\mathtt{of}\ \mathtt{many}\,(x) \to e' : \tau'}\ !E
$$

9.3 Operational Semantics

$$
\begin{aligned}
V \in \mathit{Value} &::= x \mid \lambda x.e \mid \{\pi_1 \Rightarrow e_1 \mid \pi_2 \Rightarrow e_2\} \mid \iota_i{\cdot}V \mid \langle V_1, V_2\rangle \mid \{\} \mid \langle\rangle \mid \mathtt{many}\,(V) \\
E \in \mathit{EvalCxt} &::= \Box \mid \mathtt{let}\, x = E \,\mathtt{in}\, e' \mid E\ e' \mid V\ E \mid E{\cdot}\pi_i \\
&\quad \mid \mathtt{case}\, E\ \{\ldots\} \mid \iota_i{\cdot}E \mid \langle E, e_2\rangle \mid \langle V_1, E\rangle \mid \mathtt{many}\,(E)
\end{aligned}
$$

Note that the instance of `case` stands for *any* of the possible `case` expressions for the different positive types.

$$\begin{aligned}
\texttt{let}\, x = V \,\texttt{in}\, e &\mapsto e[V/x] \\
(\lambda x.e)\ V &\mapsto e[V/x] \\
\{\pi_1 \Rightarrow e_1 \mid \pi_2 \Rightarrow e_2\}{\cdot}\pi_i &\mapsto e_i \\
\texttt{case}\, \iota_i{\cdot}V \,\texttt{of}\, \{\iota_1{\cdot}x \Rightarrow e_1 \mid \iota_2{\cdot}y \Rightarrow e_2\} &\mapsto e_i[V/x] \\
\texttt{case}\ \langle V_1, V_2\rangle \,\texttt{of}\ \langle x, y\rangle \Rightarrow e &\mapsto e[V_1/x, V_2/y] \\
\texttt{case}\ \langle\rangle \,\texttt{of}\ \langle\rangle \Rightarrow e &\mapsto e \\
\texttt{case}\, \texttt{many}\,(V) \,\texttt{of}\, \texttt{many}\,(x) \,\texttt{in}\, e &\mapsto e[V/x]
\end{aligned}$$

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

9.4 Type Safety: Resource Safety

Lemma 9.1 (Structural rules)**.** *The following typing rules are admissible:*

$$\frac{\Gamma; \Delta \vdash e' : \tau'}{\Gamma, x : \tau; \Delta \vdash e' : \tau'}\ !W \qquad \frac{\Gamma, x : \tau, y : \tau; \Delta \vdash e' : \tau'}{\Gamma, z : \tau; \Delta \vdash e'[z/x, z/y] : \tau'}\ !C$$

Proof. By induction on the premise of both rules. □

Lemma 9.2 (Unique Use)**.** *If $\Gamma; \Delta, x : \tau \vdash E[x] : \tau'$ is derivable, then x is not a free variable of E.*

Proof. By induction on the typing derivation of $\Gamma; \Delta, x : \tau \vdash E[x] : \tau'$. □

Lemma 9.3 (Linear Substitution)**.** *If $\Gamma; \Delta \vdash e : \tau$ and $\Gamma; \Delta', x : \tau \vdash e' : \tau'$ then $\Gamma; \Delta', \Delta \vdash e'[e/x] : \tau'$.*

Proof. By induction on the typing derivation of $\Gamma; \Delta, x : \tau \vdash e' : \tau'$. □

Lemma 9.4 (Non-linear substitution)**.** *If $\Gamma; \bullet \vdash e : \tau$ and $\Gamma, x : \tau; \Delta \vdash e' : \tau'$ then $\Gamma; \Delta \vdash e'[e/x] : \tau'$*

Proof. By induction on the typing derivation of $\Gamma, x : \tau; \Delta \vdash e' : \tau'$, using weakening and contraction as necessary. □

Lemma 9.5 (Progress)**.** *If $\bullet; \bullet \vdash e : \tau$ then either e is an answer A or there exists an expression e' such that $e \mapsto e'$.*

Proof. By induction on the typing derivation of $\bullet; \bullet \vdash e : \tau$. □

Lemma 9.6 (Preservation)**.** *If $\Gamma; \Delta \vdash e : \tau$ and $e \mapsto e'$ then $\Gamma; \Delta \vdash e' : \tau$.*

Proof. For e to take a step, it must be that $e = E[e_1]$, $e_1 \mapsto e_1'$ by one of the operational rules, and $E[e_1'] = e'$. Preservation follows by induction on the typing derivation of $\Gamma; \Delta \vdash E[e_1] : \tau$, using the above two substitution lemmas in the base case ($E = \square$) where an operational rule applies. □

Chapter 10

Reducibility

Definition 10.1 (Closed)**.** An expression e is *closed* if $FV(e) = \{\}$.

Definition 10.2 (Well-typed)**.** An expression e is *well-typed* if there exists an environment Γ and type τ such that $\Gamma \vdash e : \tau$ is derivable.

Definition 10.3 (Termination)**.** An expression e is *terminating* if there exists another expression e' such that $e \mapsto^* e'$ and $e' \not\mapsto$. The set of all terminating expressions is

$$\mathit{Termin} = \{e \in \mathit{Expr} \mid \exists e' \in \mathit{Expr}. e \mapsto^* e' \not\mapsto\}$$

For this chapter, let's take the simply typed λ-calculus with booleans as our language. That is, the language

$$\begin{aligned} \tau \in \mathit{Type} &::= \texttt{bool} \mid \tau_1 \to \tau_2 \\ b \in \mathit{Bool} &::= \texttt{true} \mid \texttt{false} \\ e \in \mathit{Expr} &::= x \mid \lambda x.e \mid e_1\ e_2 \mid b \mid \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \end{aligned}$$

along with the associated static and dynamic semantics described above. The goal is to show that all closed and well-typed expressions of this language are terminating.

10.1 A False Start on Termination

Theorem 10.1. *If* $\bullet \vdash e : \tau$ *then* e *is terminating.*

Proof. By induction on the derivation $\mathcal{D}$ of $\bullet \vdash e : \tau$.

- $\mathcal{D} = \overline{\bullet \vdash b : \texttt{bool}}$

  Then $b \mapsto^* b \not\mapsto$, so b is terminating.

- $$\mathcal{D} = \dfrac{\overset{\vdots\ \mathcal{D}'}{\bullet \vdash e : \texttt{bool}} \quad \overset{\vdots\ \mathcal{D}_1}{\bullet \vdash e_1 : \tau} \quad \overset{\vdots\ \mathcal{D}_2}{\bullet \vdash e_2 : \tau}}{\bullet \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau}$$

  Inductive Hypothesis. *e, e_1, and e_2 are all terminating.*

  Since e is terminating, we know there is an e' such that $e \mapsto^* e' \not\mapsto$. By type safety, we know that e' is not stuck, so it must be that e' is some value, and furthermore by preservation and canonical forms, it must be that e' is some boolean b. In either case ($b = \texttt{true}$ or $b = \texttt{false}$), we have

  $$\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \mapsto^* \texttt{if}\, b\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \mapsto e_i$$

  where e_i is one of e_1 or e_2. Since both e_1 and e_2 are terminating, then there must be some e'' such that $e_i \mapsto^* e'' \not\mapsto$. Therefore,

  $$\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \mapsto^* \texttt{if}\, b\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \mapsto e_i \mapsto^* e' \not\mapsto$$

- $$\mathcal{D} = \dfrac{\overset{\vdots\ \mathcal{D}'}{x : \tau_1 \vdash e : \tau_2}}{\bullet \vdash \lambda x.e : \tau_1 \to \tau_2}$$

  Note that there is no useful inductive hypothesis, because in the premise, the environment used to type check e is not empty.

  But that's fine, since λ-abstractions are already results,

  $$\lambda x.e \mapsto^* \lambda x.e \not\mapsto$$

  so $\lambda x.e$ is terminating already.

- $$\mathcal{D} = \dfrac{\bullet \vdash e_1 : \tau_2 \to \tau_1 \quad \bullet \vdash e_2 : \tau_2}{\bullet \vdash e_1\ e_2 : \tau_1}$$

  Inductive Hypothesis. *e_1 and e_2 are both terminating.*

  Since e_1 is terminating, there must be some e_1' such that $e_1 \mapsto^* e_1' \not\mapsto$. By type safety, we know that e_1' is not stuck so it must be some value, and furthermore by preservation and canonical forms, it must be that e_1' is some λ-abstraction $\lambda x.e'$. So we have

  $$e_1\ e_2 \mapsto^* (\lambda x.e')\ e_2 \mapsto e'[e_2/x]$$

  But neither inductive hypothesis says anything about $e'[e_2/x]$! So we are stuck. ⊠

We can't complete the proof because the inductive hypothesis is *too weak* to say anything useful about function application. So lets try again by *strengthening the inductive hypothesis* to make application the easy case.

10.2 Reducibility Model of Types

Trick: define an interpretation of type as a set of expressions (i.e. a unary relation on expressions) that they describe. Make sure that each such set only contains terminating expressions by definition. Then show that all well-typed expressions are in the appropriate set based on their type.

For describing booleans, it's not just enough to say that $\mathtt{bool} = \{\mathtt{true}, \mathtt{false}\}$. Why? Because you might have other expressions that are not simple boolean values *yet* like

$$\mathtt{if\ true\ then\ false\ else\ true} \mapsto \mathtt{false}$$

$\bullet \vdash \mathtt{if\ true\ then\ false\ else\ true} : \mathtt{bool}$, so it should be considered a `bool` too! So we'll need to consider the *expansion* of some set of values to include expressions which reduce to one of those values.

$$\mathbb{C}^* = \{e \in \mathit{Expr} \mid \exists e' \in \mathbb{C}, e \mapsto^* e'\}$$

In other words, for a set $\mathbb{C}$, $\mathbb{C}^*$ is the *closure of* $\mathbb{C}$ *under expansion.* Some properties about this closure:

- $\mathbb{C}^{**} = \mathbb{C}^*$
- $\mathbb{C} \subseteq \mathbb{C}^*$
- If $\mathbb{C} \subseteq \mathit{Termin}$ then $\mathbb{C}^* \subseteq \mathit{Termin}$

We can now interpret each type as a set of terms:

$$\begin{aligned} [\![\mathtt{bool}]\!] &= \{\mathtt{true}, \mathtt{false}\}^* = \{e \in \mathit{Expr} \mid \exists b. e \mapsto^* b\} \\ [\![\tau_1 \to \tau_2]\!] &= \{e \in \mathit{Termin} \mid \forall e_1 \in [\![\tau_1]\!]. e\ e_1 \in [\![\tau_2]\!]\} \end{aligned}$$

Lemma 10.1. *For all types τ, if $e \in [\![\tau]\!]$ then e is terminating. That is to say, $[\![\tau]\!] \subseteq \mathit{Termin}$ for any τ.*

Proof. By cases on τ. $[\![\mathtt{bool}]\!] \subseteq \mathit{Termin}$ because $\{\mathtt{true}, \mathtt{false}\} \subseteq \mathit{Termin}$. $[\![\tau_1 \to \tau_2]\!] \subseteq \mathit{Termin}$ by definition. □

10.3 Termination

Now, we must extend the interpretation of types to the interpretation of hypothetical judgements. That way, we won't get stuck in a rule that extends the environment.

$$\begin{aligned} [\![\Gamma]\!] &= \{\sigma \in \mathit{Subst} \mid \forall (x : \tau) \in \Gamma. x[\sigma] \in [\![\tau]\!]\} \\ [\![\Gamma \vdash e : \tau]\!] &= \forall \sigma \in [\![\Gamma]\!]. e[\sigma] \in [\![\tau]\!] \end{aligned}$$

Lemma 10.2 (Adequacy, *a.k.a.* the Fundamental Lemma)**.** *If $\Gamma \vdash e : \tau$ is derivable, then $[\![\Gamma \vdash e : \tau]\!]$ is true.*

Proof. By induction on the derivation $\mathcal{D}$ of $\Gamma \vdash e : \tau$.

- $\mathcal{D} = \overline{\Gamma, x : \tau \vdash x : \tau}$

  We must show $[\![\Gamma, x : \tau \vdash x : \tau]\!]$ is true, i.e. , for all $\sigma \in [\![\Gamma, x : \tau]\!]$, $x[\sigma] \in [\![\tau]\!]$. This follows by definition of $[\![\Gamma, x : \tau]\!]$ since $(x : \tau) \in (\Gamma, x : \tau)$.

- $\mathcal{D} = \overline{\Gamma \vdash b : \texttt{bool}}$

  We must show $[\![\Gamma \vdash b : \texttt{bool}]\!]$ is true, i.e. , for all $\sigma \in [\![\Gamma]\!]$, $b[\sigma] = b \in [\![\texttt{bool}]\!]$. Note that $b \in \{\texttt{true}, \texttt{false}\}$, so $b \in [\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}\}^*$ since $b \mapsto^* b$.

- $\mathcal{D} = \dfrac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau}$

  Inductive Hypothesis. *$[\![\Gamma \vdash e : \texttt{bool}]\!]$, $[\![\Gamma \vdash e_1 : \tau]\!]$, and $[\![\Gamma \vdash e_2 : \tau]\!]$ are both true.*

  We must show $[\![\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau]\!]$ is true, i.e. , for all $\sigma \in [\![\Gamma]\!]$,

  $$(\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2)[\sigma] = \texttt{if}\, e[\sigma]\, \texttt{then}\, e_1[\sigma]\, \texttt{else}\, e_2[\sigma] \in [\![\tau]\!]$$

  Suppose $\sigma \in [\![\Gamma]\!]$. From the inductive hypothesis, we learn that

  $$e[\sigma] \in [\![\texttt{bool}]\!] \qquad e_1[\sigma] \in [\![\tau]\!] \qquad e_2[\sigma] \in [\![\tau]\!]$$

  Because $e[\sigma] \in [\![\texttt{bool}]\!]$, then by definition of $[\![\texttt{bool}]\!]$, there must be some b such that $e[\sigma] \mapsto^* b$. Therefore,

  $$\texttt{if}\, e[\sigma]\, \texttt{then}\, e_1[\sigma]\, \texttt{else}\, e_2[\sigma] \mapsto^* \texttt{if}\, b\, \texttt{then}\, e_1[\sigma]\, \texttt{else}\, e_2[\sigma] \mapsto e_i[\sigma]$$

  where e_i is either e_1 (when $b = \texttt{true}$) or e_2 (when $b = \texttt{false}$). In either case, $e_i[\sigma] \in [\![\tau]\!]$. So, since $[\![\tau]\!]$ is closed under expansion (PENDING), it must be that

  $$\texttt{if}\, e[\sigma]\, \texttt{then}\, e_1[\sigma]\, \texttt{else}\, e_2[\sigma] \in [\![\tau]\!]$$

- $\mathcal{D} = \dfrac{\overset{\vdots\, \mathcal{D}'}{\Gamma \vdash e : \tau_1 \to \tau_2} \quad \overset{\vdots\, \mathcal{D}_1}{\Gamma \vdash e_1 : \tau_1}}{\Gamma \vdash e\ e_1 : \tau_2}$

  Inductive Hypothesis. *Both $[\![\Gamma \vdash e : \tau_1 \to \tau_2]\!]$ and $[\![\Gamma \vdash e_1 : \tau_1]\!]$ are true.*

  We must show that $[\![\Gamma \vdash e\ e_1 : \tau_2]\!]$ is true, i.e. , for all $\sigma \in [\![\Gamma]\!]$,

  $$(e\ e_1)[\sigma] = e[\sigma]\ e_1[\sigma] \in [\![\tau_2]\!]$$

  Suppose $\sigma \in [\![\Gamma]\!]$. From the inductive hypothesis, we learn that $e[\sigma] \in [\![\tau_1 \to \tau_2]\!]$ and $e_1[\sigma] \in [\![\tau_1]\!]$. So by definition of $[\![\tau_1 \to \tau_2]\!]$, $e[\sigma]\ e_1[\sigma] \in [\![\tau_2]\!]$.

- $$\mathcal{D} = \cfrac{\begin{array}{c}\vdots\ \mathcal{D}' \\ \Gamma, x : \tau_1 \vdash e : \tau_2\end{array}}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

  Inductive Hypothesis. $[\![\Gamma, x : \tau_1 \vdash e : \tau_2]\!]$ *is true.*

  We must show that $[\![\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2]\!]$ is true, i.e. , for all $\sigma \in [\![\Gamma]\!]$,[1]

  $$(\lambda x.e)[\sigma] = \lambda x.(e[\sigma]) \in [\![\tau_1 \to \tau_2]\!]$$

  By definition of $[\![\tau_1 \to \tau_2]\!]$, $\lambda x.e[\sigma] \in [\![\tau_1 \to \tau_2]\!]$ exactly when $(\lambda x.e[\sigma])\ e_1 \in [\![\tau_2]\!]$ for all $e_1 \in [\![\tau_1]\!]$, so suppose some arbitrary $e_1 \in [\![\tau_1]\!]$. Observe that

  $$(\lambda x.e[\sigma])\ e_1 \mapsto e[\sigma, e_1/x]$$

  Since $e_1 \in [\![\tau_1]\!]$ and $\sigma \in [\![\Gamma]\!]$, $\sigma, e_1/x \in [\![\Gamma, x : \tau_1]\!]$. And from the inductive hypothesis, we learn that $e[\sigma, e_1/x] \in [\![\tau_2]\!]$, which means that by closure under expansion (PENDING), $(\lambda x.e[\sigma])\ e_1 \in [\![\tau_2]\!]$ as well. Therefore, $\lambda x.e[\sigma] \in [\![\tau_1 \to \tau_2]\!]$ for any σ. □

Lemma 10.3 (Closure under expansion)**.** *For all types τ, if $e \mapsto^* e'$ and $e' \in [\![\tau]\!]$ then $e \in [\![\tau]\!]$ as well. That is to say, $[\![\tau]\!]^* \subseteq [\![\tau]\!]$ for any τ.*

Proof. By induction on the syntax of the type τ.

- $\tau = \texttt{bool}$: $[\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}\}^* = \{\texttt{true}, \texttt{false}\}^{**} = [\![\texttt{bool}]\!]^*$

- $\tau = \tau_1 \to \tau_2$: We must show that if $e \mapsto^* e'$ and $e' \in [\![\tau_1 \to \tau_2]\!]$, then $e \in [\![\tau_1 \to \tau_2]\!]$.

  Inductive Hypothesis. $[\![\tau_1]\!] = [\![\tau_1]\!]^*$ *and* $[\![\tau_2]\!] = [\![\tau_2]\!]^*$.

  Suppose e and e' are arbitrary expressions such that $e \mapsto^* e' \in [\![\tau_1 \to \tau_2]\!]$. Showing that $e \in [\![\tau_1 \to \tau_2]\!]$ is the same as showing that for all $e_1 \in [\![\tau_1 \to \tau_2]\!]$, $e\ e_1 \in [\![\tau_2]\!]$. So also let e_1 be an arbitrary expression in $[\![\tau_1]\!]$. Now observe that

  $$e\ e_1 \mapsto^* e'\ e_1 \in [\![\tau_2]\!]$$

  because $e' \in [\![\tau_1 \to \tau_2]\!]$. Therefore, we learn from the inductive hypothesis $[\![\tau_2]\!] = [\![\tau_2]\!]^*$ that $e\ e_1 \in [\![\tau_2]\!]$ as well, and so $e \in [\![\tau_1 \to \tau_2]\!]$ by definition. □

Corollary 10.1 (Closed Reducibility)**.** *If $\bullet \vdash e : \tau$ is derivable then $e \in [\![\tau]\!]$.*

[1]Up to α equivalence, we may assume that this substitution is defined, meaning that x does not appear in the domain of σ (there is no $x/e' \in \sigma$) and is not free in any of the expressions in the codomain of σ (for every $y/e' \in \sigma$, $x \notin FV(e')$). This is because we can always pick a fresh name that does not appear anywhere else to rename the bound variable of the λ-abstraction if either of these conditions is not met.

Corollary 10.2 (Termination)**.** *If $\bullet \vdash e : \tau$ is derivable then e is terminating.*

Proof. From the above, we know that $e \in [\![\tau]\!] \subseteq \mathit{Termin}$. $\square$

Corollary 10.3 (Boolean Evaluation)**.** *If $\bullet \vdash e : \texttt{bool}$ is derivable, then there is a boolean b such that $e \mapsto^* b$.*

Proof. From the above, we know that $e \in [\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}\}^*$. $\square$

Chapter 11

Parametricity

What happens if we want to generalize the reducibility relation to quantifiers (universal and existential polymorphism)? This

$$[\![\forall\alpha.\tau]\!] = \{e \in \mathit{Termin} \mid \forall\tau' \in \mathit{Type}.e\ \tau' \in [\![\tau[\tau'/\alpha]]\!]\}$$

doesn't make sense! The definition doesn't follow by induction on the type in $[\![\_]\!]$ because $\tau[\tau'/\alpha]$ is not a sub-term of the type $\forall\alpha.\tau$. Consider $[\![\forall\alpha.\alpha]\!]$. One of the types that τ' ranges over is again $\forall\alpha.\alpha$! That means to understand $[\![\forall\alpha.\alpha]\!]$, we must first understand $[\![\alpha[(\forall\alpha.\alpha)/\alpha]]\!] = [\![\forall\alpha.\alpha]\!]$, so we've gotten nowhere.

11.1 Reducibility Candidates

We'll need to say what to do for a type variable α. What's the meaning of a type variable? Substitution!

$$[\![\alpha]\!]_\theta = \theta(\alpha)$$

The "substitution environment" θ is a function from type variables to... what? Something that looks like the interpretation of a type. That is, a *candidate* for a sensible *reducibility* predicate.

Definition 11.1 (Reducibility Candidate)**.** A *reducibility candidate* $\mathbb{C}$ is a set of expressions (i.e. a unary predicate on expressions) satisfying the following two properties:

- *termination*: all expressions in $\mathbb{C}$ are terminating.
- *expansion*: if $e \mapsto^* e' \in \mathbb{C}$ then $e \in \mathbb{C}$.

In other words, a reducibility candidate is any set of expressions $\mathbb{C}$ such that

$$\mathbb{C}^* \subseteq \mathbb{C} \subseteq \mathit{Termin}$$

The set of all reducibility candidates is

$$\mathit{CR} = \{\mathbb{C} \in \wp(\mathit{Termin}) \mid \mathbb{C}^* \subseteq \mathbb{C}\}$$

Lemma 11.1. *For any $\mathbb{A} \subseteq \mathit{Termin}$, $\mathbb{A}^*$ is a reducibility candidate.*

Proof. First, note that from $\mathbb{A} \subseteq \mathit{Termin}$ we know that $\mathbb{A}^* \subseteq \mathit{Termin}$. Furthermore, $\mathbb{A}^{**} = \mathbb{A}^*$, so

$$\mathbb{A}^{**} \subseteq \mathbb{A}^* \subseteq \mathit{Termin} \qquad \square$$

So the substitution environment θ in $[\![\alpha]\!]_\theta$ is a function $\mathit{TypeVar} \to \mathit{CR}$.

11.2 Logical Relations

We can now define the interpretation of the quantifiers by deception! For universal quantifier, pick whatever reducibility candidate you want (the domain of reducibility candidates are now a fully-defined entity, independent of this interpretation), and then completely lie by telling the polymorphic expression you're using it at some other unrelated type. For existential quantifier, the packing operation can completely lie to its client by saying it used one abstract type, but actually used any arbitrary candidate of its liking.

$$\begin{aligned}
[\![\alpha]\!]_\theta &= \theta(\alpha) \\
[\![\tau_1 \to \tau_2]\!]_\theta &= \{e \in \mathit{Termin} \mid \forall e' \in [\![\tau_1]\!]_\theta.\ e\ e' \in [\![\tau_2]\!]_\theta\} \\
[\![\forall\alpha.\tau]\!]_\theta &= \{e \in \mathit{Termin} \mid \forall \tau \in \mathit{Type}, \mathbb{C} \in \mathit{CR}.\ e\ \tau \in [\![\tau]\!]_{\theta,\mathbb{C}/\alpha}\} \\
[\![\exists\alpha.\tau]\!]_\theta &= \{\langle \tau, e\rangle \mid \exists \mathbb{C} \in \mathit{CR}.\ e \in [\![\tau]\!]_{\theta,\mathbb{C}/\alpha}\}^*
\end{aligned}$$

This deception makes the definition well-formed by induction on the syntax of types. But also is a strong model of parametricity: the program *cannot* rely on the type annotations (in packing and specialization) to decide what to do, because those annotations are allowed to be completely bogus nonsense.

11.3 Termination

Now, we must extend the interpretation of types to the interpretation of hypothetical judgements. That way, we won't get stuck in a rule that extends the environment.

$$\begin{aligned}
[\![\Theta]\!] &= \{\Theta\} \to \mathit{CR} \\
[\![\Theta;\Gamma]\!]_\theta &= \{\sigma \in \mathit{Subst} \mid (\forall (x:\tau) \in \Gamma. x[\sigma] \in [\![\tau]\!]_\theta) \wedge (\forall \alpha \in \Delta. \alpha[\sigma] \in \mathit{Type})\} \\
[\![\Theta \vdash \tau : \star]\!] &= \forall \theta \in [\![\Theta]\!]. [\![\tau]\!]_\theta \in \mathit{CR} \\
[\![\Theta;\Gamma \vdash e : \tau]\!] &= \forall \theta \in [\![\Theta]\!]. \sigma \in [\![\Theta;\Gamma]\!]_\theta . e[\sigma] \in [\![\tau]\!]_\theta
\end{aligned}$$

Lemma 11.2 (Semantic substitution)**.** $[\![\tau[\tau'/\alpha]]\!]_\theta = [\![\tau]\!]_{\theta,[\![\tau']\!]_\theta/\alpha}$

Proof. By induction on the syntax of the type τ (assume renaming so that the bound type variables of τ are distinct from α and the free variables of τ').

- α: $[\![\alpha[\tau'/\alpha]]\!]_\theta = [\![\tau']\!]_\theta = (\theta, [\![\tau']\!]_\theta/\alpha)(\alpha) = [\![\alpha]\!]_{\theta,[\![\tau']\!]_\theta/\alpha}$
- β where $\beta \neq \alpha$: $[\![\beta[\tau'/\alpha]]\!]_\theta = [\![\beta]\!]_\theta = \theta(\beta) = (\theta, [\![\tau']\!]_\theta/\alpha)(\beta) = [\![\beta]\!]_{\theta,[\![\tau']\!]_\theta/\alpha}$
- $\tau_1 \to \tau_2$: $[\![(\tau_1 \to \tau_2)[\tau'/\alpha]]\!]_\theta = [\![\tau_1[\tau'/\alpha] \to \tau_2[\tau'/\alpha]]\!]_\theta =_{IH} [\![\tau_1 \to \tau_2]\!]_{\theta,[\![\tau']\!]_\theta/\alpha}$
- $\forall\beta.\tau$: $[\![(\forall\beta.\tau)[\tau'/\alpha]]\!]_\theta = [\![\forall\beta.(\tau[\tau'/\alpha])]\!]_\theta =_{IH} [\![\forall\beta.\tau]\!]_{\theta,[\![\tau']\!]_\theta/\alpha}$
- $\exists\beta.\tau$: $[\![(\exists\beta.\tau)[\tau'/\alpha]]\!]_\theta = [\![\exists\beta.(\tau[\tau'/\alpha])]\!]_\theta =_{IH} [\![\exists\beta.\tau]\!]_{\theta,[\![\tau']\!]_\theta/\alpha}$ □

Lemma 11.3 (Semantic weakening)**.** *If $\alpha \notin FV(\tau)$ then for any $\mathbb{C} \in CR$, $[\![\tau]\!]_{\theta,\mathbb{C}/\alpha} = [\![\tau]\!]_\theta$.*

Proof. By induction on the syntax of the type τ. □

Lemma 11.4 (Adequacy of Types)**.** *If $\Theta \vdash \tau : \star$ is derivable then $[\![\Theta \vdash \tau : \star]\!]$ is true.*

Proof. By induction on the derivation of $\Theta \vdash \tau : \star$:

- $\dfrac{}{\Theta, \alpha \vdash \alpha : \star}$

  Suppose $\theta \in [\![\Theta, \alpha]\!]$. Then $[\![\alpha]\!]_\theta = \theta(\alpha) \in CR$

- $\dfrac{\Theta \vdash \tau_1 : \star \quad \Theta \vdash \tau_2 : \star}{\Theta \vdash \tau_1 \to \tau_2 : \star}$

  Inductive Hypothesis. *Both $[\![\Theta \vdash \tau_1 : \star]\!]$ and $[\![\Theta \vdash \tau_2 : \star]\!]$ are true.*

  Suppose $\theta \in [\![\Theta]\!]$. We must then show that $[\![\tau_1 \to \tau_2]\!]_\theta \in CR$, i.e. the set of expressions has the termination and expansion properties. First, every expression in $[\![\tau_1 \to \tau_2]\!]_\theta$ is terminating by definition. Second, suppose that $e \mapsto^* e' \in [\![\tau_1 \to \tau_2]\!]_\theta$ and some arbitrary $e_1 \in [\![\tau_1]\!]_\theta$ so that $e'\ e_1 \in [\![\tau_2]\!]_\theta$ by definition. Observe that

  $$e\ e_1 \mapsto^* e'\ e_1 \in [\![\tau_2]\!]_\theta$$

  We learn from the inductive hypothesis that $[\![\tau_2]\!]_\theta \in CR$, so $e\ e_1 \in [\![\tau_2]\!]_\theta$ by expansion. Therefore, $e \in [\![\tau_1 \to \tau_2]\!]_\theta$.

- $\dfrac{\Theta, \alpha \vdash \tau : \star}{\Theta \vdash \forall\alpha.\tau : \star}$

  Inductive Hypothesis. *$[\![\Theta, \alpha \vdash \tau : \star]\!]$ is true.*

  Suppose $\theta \in [\![\Theta]\!]$. We must show that $[\![\forall\alpha.\tau]\!]_\theta \in CR$. First, every expression in $[\![\forall\alpha.\tau]\!]_\theta$ is terminating by definition. Second, suppose that $e \mapsto^* e' \in [\![\forall\alpha.\tau]\!]_\theta$, and suppose some arbitrary type τ and candidate $\mathbb{C}$ so that $e'\ \tau \in [\![\tau]\!]_{\theta,\mathbb{C}/\alpha}$ by definition. Observe that

  $$e\ \tau \mapsto^* e'\ \tau \in [\![\tau]\!]_{\theta,\mathbb{C}/\alpha}$$

  Since $\theta, \mathbb{C}/\alpha \in [\![\Theta]\!]$, we learn from the inductive hypothesis that $[\![\tau]\!]_{\theta,\mathbb{C}/\alpha} \in CR$, so $e\ \tau \in [\![\tau]\!]_{\theta,\mathbb{C}/\alpha}$ by expansion. Therefore, $e \in [\![\forall\alpha.\tau]\!]_\theta$.

- $\dfrac{\Theta, \alpha \vdash \tau : \star}{\Theta \vdash \exists \alpha.\tau : \star}$

  Inductive Hypothesis. $[\![\Theta, \alpha \vdash \tau : \star]\!]$ *is true.*

  Suppose $\theta \in [\![\Theta]\!]$. We must show that $[\![\exists \alpha.\tau]\!]_\theta \in CR$. Since $[\![\exists \alpha.\tau]\!]_\theta$ is defined as the $\cdot^*$-closure of a set of non-reducing expressions, it has the termination and expansion properties by definition. □

Lemma 11.5 (Adequacy of Programs)**.** *If* $\Theta; \Gamma \vdash e : \tau$ *is derivable, then* $[\![\Theta; \Gamma \vdash e : \tau]\!]$ *is true.*

Proof. By induction on the derivation of $\Theta; \Gamma \vdash e : \tau$. In each case, note that since $\Theta; \Gamma \vdash e : \tau$ is derivable, so too is $\Theta \vdash \tau : \star$, which means that $[\![\Theta \vdash \tau : \star]\!]$ by adequacy of types. The cases for variables (x), functions ($\lambda x.e$), and applications ($e_1\ e_2$) are analogous to the proof for the simply typed lambda calculus. The new cases for the quantifiers are:

- $\dfrac{\Theta; \Gamma \vdash e : \forall \alpha.\tau \quad \Theta \vdash \tau' : \star}{\Theta; \Gamma \vdash e\ \tau' : \tau[\tau'/\alpha]}\ \forall E$

  Inductive Hypothesis. *Both* $[\![\Theta; \Gamma \vdash e : \forall \alpha.\tau]\!]$ *and* $[\![\Theta \vdash \tau' : \star]\!]$ *are true.*

  Suppose that $\theta \in [\![\Theta]\!]$ and $\sigma \in [\![\Theta; \Gamma]\!]_\theta$. We must show that

  $$(e\ \tau')[\sigma] = e[\sigma]\ \tau[\sigma] \in [\![\tau[\tau'/\alpha]]\!]_\theta = [\![\tau]\!]_{\theta, [\![\tau']\!]_\theta/\alpha}$$

  But from the inductive hypothesis, we learn that $e[\sigma] \in [\![\forall \alpha.\tau]\!]_\theta$ and $[\![\tau']\!]_\theta \in CR$. So $e[\sigma]\ \tau' \in [\![\tau]\!]_{\theta, [\![\tau']\!]_\theta/\alpha}$ by definition of $e[\sigma] \in [\![\forall \alpha.\tau]\!]_\theta$.

- $\dfrac{\Theta, \alpha; \Gamma \vdash e : \tau}{\Theta; \Gamma \vdash \Lambda \alpha.e : \forall \alpha.\tau}\ \forall I$

  Inductive Hypothesis. $[\![\Theta, \alpha; \Gamma \vdash e : \tau]\!]$ *is true.*

  Suppose that $\theta \in [\![\Theta]\!]$ and $\sigma \in [\![\Theta; \Gamma]\!]_\theta$. We must show that

  $$(\Lambda \alpha.e)[\sigma] = \Lambda \alpha.(e[\sigma]) \in [\![\forall \alpha.\tau]\!]_\theta$$

  i.e. , that for an arbitrary type $\tau' \in \mathit{Type}$ and candidate $\mathbb{C} \in CR$,

  $$(\Lambda \alpha.e[\sigma])\ \tau' \in [\![\tau]\!]_{\theta, \mathbb{C}/\alpha}$$

  Since $\theta, \mathbb{C}/\alpha \in \{\Theta, \alpha\} \to CR$ and $\sigma, \tau'/\alpha \in [\![\Theta, \alpha; \Gamma]\!]_\theta$, we learn from the inductive hypothesis that

  $$(\Lambda \alpha.e[\sigma])\ \tau' \mapsto e[\sigma, \tau'/\alpha] \in [\![\tau]\!]_{\theta, \mathbb{C}/\alpha}$$

  Furthermore, since $[\![\tau]\!]_{\theta, \mathbb{C}/\alpha} \in CR$ (derived from adequacy of types and the fact that $\Theta, \alpha \vdash \tau : \star$), the expansion property ensures that $(\Lambda \alpha.e[\sigma])\ \tau' \in [\![\tau]\!]_{\theta, \mathbb{C}/\alpha}$ as well. Therefore, $\Lambda \alpha.e[\sigma] \in [\![\forall \alpha.\tau]\!]_\theta$.

- $$\frac{\Theta \vdash \tau' : \star \quad \Theta; \Gamma \vdash e : \tau[\tau'/\alpha]}{\Theta; \Gamma \vdash \langle \tau', e \rangle : \exists \alpha.\tau} \exists I$$

  Inductive Hypothesis. *Both* $[\![\Theta \vdash \tau' : \star]\!]$ *and* $[\![\Theta; \Gamma \vdash e : \tau[\tau'/\alpha]]\!]$ *are true.*

  Suppose that $\theta \in [\![\Theta]\!]$ and $\sigma \in [\![\Theta; \Gamma]\!]_\theta$. We must show that

  $$\langle \tau', e \rangle [\sigma] = \langle \tau'[\sigma], e[\sigma] \rangle \in [\![\exists \alpha.\tau]\!]_\theta$$

  We learn from the inductive hypothesis that $[\![\tau']\!]_\theta \in CR$ and

  $$e[\sigma] \in [\![\tau[\tau'/\alpha]]\!]_\theta = [\![\tau]\!]_{\theta, [\![\tau']\!]_\theta/\alpha}$$

  Therefore, $\langle \tau'[\sigma], e[\sigma] \rangle \in [\![\exists \alpha.\tau]\!]_\theta$ by definition.

- $$\frac{\Theta; \Gamma \vdash e : \exists \alpha.\tau \quad \Theta, \alpha; \Gamma, x : \tau \vdash e' : \tau' \quad \Theta \vdash \tau' : \star}{\Theta; \Gamma \vdash \texttt{open}\, e \,\texttt{as}\, \langle \alpha, x \rangle \Rightarrow e' : \tau'} \exists E$$

  Inductive Hypothesis. $[\![\Theta; \Gamma \vdash e : \exists \alpha.\tau]\!]$*,* $[\![\Theta, \alpha; \Gamma, x : \tau \vdash e' : \tau']\!]$*, and* $[\![\Theta \vdash \tau' : \star]\!]$ *are all true.*

  Suppose that $\theta \in [\![\Theta]\!]$ and $\sigma \in [\![\Theta; \Gamma]\!]_\theta$. We must show that

  $$(\texttt{open}\, e \,\texttt{as}\, \langle \alpha, x \rangle \Rightarrow e')[\sigma] = \texttt{open}\, e[\sigma] \,\texttt{as}\, \langle \alpha, x \rangle \Rightarrow e'[\sigma] \in [\![\tau']\!]_\theta$$

  From the inductive hypothesis, we learn that $e[\sigma] \in [\![\exists \alpha.\tau]\!]_\theta$, which means that

  $$e[\sigma] \mapsto^* \langle \tau_1, e_1 \rangle$$

  for some arbitrary type $\tau_1 \in \mathit{Type}$, candidate $\mathbb{C} \in CR$, and expression $e_1 \in [\![\tau]\!]_{\theta, \mathbb{C}/\alpha}$. And because the extended $\theta, \mathbb{C} \in \{\Theta, \alpha\}$ and $\sigma, \tau_1/\alpha, e_1/x \in [\![\Theta, \alpha; \Gamma, x : \tau]\!]_{\theta, \mathbb{C}/\alpha}$, we learn again from the inductive hypothesis that

  $$e'[\sigma, \tau_1/\alpha, e_1/x] \in [\![\tau']\!]_{\theta, \mathbb{C}/\alpha}$$

  Now observe that

  $$\begin{aligned} \texttt{open}\, e[\sigma] \,\texttt{as}\, \langle \alpha, x \rangle \Rightarrow e'[\sigma] &\mapsto^* \texttt{open}\ \langle \tau_1, e_1 \rangle \,\texttt{as}\, \langle \alpha, x \rangle \Rightarrow e'[\sigma] \\ &\mapsto e'[\sigma, \tau_1/\alpha, e_1/x] \\ &\in [\![\tau']\!]_{\theta, \mathbb{C}/\alpha} = [\![\tau']\!]_\theta \in CR \end{aligned}$$

  by semantic weakening because $\alpha \notin FV(\tau')$ which is a consequence of the premise $\Theta \vdash \tau' : \tau$. Therefore, by the expansion property of $[\![\tau']\!]_\theta$ (derived from $\Theta \vdash \tau' : \star$ and the adequacy of types), it must be that $\texttt{open}\, e[\sigma] \,\texttt{as}\, \langle \alpha, x \rangle \Rightarrow e'[\sigma] \in [\![\tau']\!]_\theta$. □

Corollary 11.1 (Closed Reducibility)**.** *If* $\bullet; \bullet \vdash e : \tau$ *is derivable, then* $e \in [\![\tau]\!]_\bullet$*.*

Corollary 11.2 (Termination)**.** *If* $\bullet; \bullet \vdash e : \tau$ *is derivable, then* e *is terminating.*

Proof. From the above, we know that $e \in [\![\tau]\!]_\bullet \subseteq \mathit{Termin}$. □

Chapter 12

Free Theorems

12.1 Logical Operators

As useful shorthand, define some "logical" operations on reducibility candidates.

$$\begin{aligned}
(\_ \Rightarrow \_) &: CR \times CR \to CR \\
\mathbb{A} \Rightarrow \mathbb{B} &= \{e \in \mathit{Termin} \mid \forall e' \in \mathbb{A}.\ e\ e' \in \mathbb{B}\} \\
\forall &: (CR \to CR) \to CR \\
\forall(\mathbb{F}) &= \{e \in \mathit{Termin} \mid \forall \tau \in \mathit{Type}, \mathbb{C} \in CR.\ e\ \tau \in \mathbb{F}(\mathbb{C})\} \\
\exists &: (CR \to CR) \to CR \\
\exists(\mathbb{F}) &= \{\langle \tau, e \rangle \mid \exists \mathbb{C} \in CR.\ e \in \mathbb{F}(\mathbb{C})\}^*
\end{aligned}$$

And notice that these operators are the essential meaning of the reducibility interpretation of types:

$$\begin{aligned}
[\![\alpha]\!]_\theta &= \theta(\alpha) \\
[\![\tau_1 \to \tau_2]\!]_\theta &= [\![\tau_1]\!]_\theta \Rightarrow [\![\tau_2]\!]_\theta \\
[\![\forall \alpha.\tau]\!]_\theta &= \forall(\mathbb{C}.\ [\![\tau]\!]_{\theta, \mathbb{C}/\alpha}) \\
[\![\exists \alpha.\tau]\!]_\theta &= \exists(\mathbb{C}.\ [\![\tau]\!]_{\theta, \mathbb{C}/\alpha})
\end{aligned}$$

12.2 Polymorphic Absurdity (Void Type)

Theorem 12.1. *There is no expression e such that $\bullet; \bullet \vdash e : \forall \alpha.\alpha$ is derivable.*

Proof. Suppose that we had some e and a derivation of $\bullet \vdash e : \forall \alpha.\alpha$. From Corollary 11.1, we would know that $e \in [\![\forall \alpha.\alpha]\!] = \forall(\mathbb{C}.\ \mathbb{C})$. In other words, for any type τ and reducibility candidate $\mathbb{C} \in CR$, it must be that $e\ \tau \in \mathbb{C}$. So let's choose the empty reducibility candidate

$$\mathbb{C} = \{\}$$

which vacuously has the termination and expansion properties. It follows that $e\ \tau \in \mathbb{C} = \{\}$ which is a contradiction. Therefore, there is no such $\bullet \vdash e : \forall\alpha.\alpha$. □

12.3 Polymorphic Identity (Unit Type)

Theorem 12.2. *If $\bullet;\bullet \vdash e : \forall\alpha.\alpha \to \alpha$ then $e =_{\beta\eta} \Lambda\alpha.\lambda x{:}\alpha.x$.*

Proof. From Corollary 11.1, we know that $e \in [\![\forall\alpha.\alpha \to \alpha]\!] = \forall(\mathbb{C}.\ \mathbb{C} \Rightarrow \mathbb{C})$ In other words, for any type τ, reducibility candidate $\mathbb{C} \in \mathit{CR}$, and expression $e_1 \in \mathbb{C}$, it must be that $e\ \tau\ e_1 \in \mathbb{C}$. So let's choose the reducibility candidate

$$\mathbb{C} = \{x\}^*$$

where the variable x is in $\mathbb{C}$ by definition. It follows that $e\ \alpha\ x \in \mathbb{C}$ as well, meaning $e\ \alpha\ x \mapsto^* x$ which implies $e\ \alpha\ x =_\beta x$. Therefore

$$e =_\eta \Lambda\alpha.e\ \alpha =_\eta \Lambda\alpha.\lambda x{:}\alpha.e\ \alpha\ x =_\beta \Lambda\alpha.\lambda x{:}\alpha.x$$ □

12.4 Encodings

12.4.1 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 12.3 (Canonicity)**.** *If $\bullet;\bullet \vdash e : \mathit{Bool}$ then $e =_{\beta\eta} \mathit{True}$ or $e =_{\beta\eta} \mathit{False}$.*

Proof. From Corollary 11.1, we know that

$$e \in [\![\mathit{Bool}]\!] = [\![\forall\delta.\delta \to \delta \to \delta]\!] = \forall(\mathbb{C}.\ \mathbb{C} \Rightarrow \mathbb{C} \Rightarrow \mathbb{C})$$

In other words, for any type τ, reducibility candidate $\mathbb{C} \in \mathit{CR}$, and terms $e_1, e_2 \in \mathbb{C}$, it must be that $e\ \tau\ e_1\ e_2 \in \mathbb{C}$. So let's choose the reducibility candidate

$$\mathbb{C} = \{x, y\}^*$$

where the variables x and y are in $\mathbb{C}$. It follows that $e\ \delta\ x\ y \in \mathbb{C}$ as well, meaning that either $e\ \delta\ x\ y \mapsto^* x$ or $e\ \delta\ x\ y \mapsto^* y$. Therefore, we have the two possible cases:

$$\begin{aligned} e &=_\eta \Lambda\delta.\lambda x{:}\delta.\lambda y{:}\delta.e\ \delta\ x\ y =_\beta \Lambda\delta.\lambda x{:}\delta.\lambda y{:}\delta.x = \mathit{True} \\ e &=_\eta \Lambda\delta.\lambda x{:}\delta.\lambda y{:}\delta.e\ \delta\ x\ y =_\beta \Lambda\delta.\lambda x{:}\delta.\lambda y{:}\delta.y = \mathit{False} \end{aligned}$$ □

12.4.2 Sums

Recall that

$$\begin{aligned}
\mathit{Sum}\ \tau_1\ \tau_2 &= \forall\delta.(\tau_1 \to \delta) \to (\tau_2 \to \delta) \to \delta \\
\mathit{Left} &: \forall\alpha.\forall\beta.\alpha \to \mathit{Sum}\ \alpha\ \beta \\
\mathit{Left} &= \Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\Lambda\delta.\lambda l{:}\alpha \to \delta.\lambda r{:}\beta \to \delta.l\ x \\
\mathit{Right} &: \forall\alpha.\forall\beta.\beta \to \mathit{Sum}\ \alpha\ \beta \\
\mathit{Right} &= \Lambda\alpha.\Lambda\beta.\lambda x{:}\beta.\Lambda\delta.\lambda l{:}\alpha \to \delta.\lambda r{:}\beta \to \delta.r\ x
\end{aligned}$$

Theorem 12.4 (Canonicity)**.** *If $\bullet;\bullet \vdash e : \mathit{Sum}\ \tau_1\ \tau_2$ then either $e =_{\beta\eta}$ $\mathit{Left}\ \tau_1\ \tau_2\ e_1$ for some $e_1 \in [\![\tau_1]\!]$ or $e =_{\beta\eta} \mathit{Right}\ \tau_1\ \tau_2\ e_2$ for some $e_2 \in [\![\tau_2]\!]$.*

Proof. From Corollary 11.1, we know that

$$e \in [\![\mathit{Sum}\ \tau_1\ \tau_2]\!][\![\forall\delta.(\tau_1 \to \delta) \to (\tau_2 \to \delta) \to \delta]\!] = \forall(\mathbb{C}.([\![\tau_1]\!]_\bullet \Rightarrow \mathbb{C}) \Rightarrow ([\![\tau_2]\!]_\bullet \Rightarrow \mathbb{C}) \Rightarrow \mathbb{C})$$

In other words, for any type τ, reducibility candidate $\mathbb{C} \in \mathit{CR}$, and expressions $e_1 \in [\![\tau_1]\!]_\bullet \Rightarrow \mathbb{C}$ and $e_2 \in [\![\tau_2]\!]_\bullet \Rightarrow \mathbb{C}$, it must be that $e\ \tau\ e_1\ e_2 \in \mathbb{C}$. So let's choose the reducibility candidate

$$\mathbb{C} = (\{l\ e_1 \mid e_1 \in [\![\tau_1]\!]_\bullet\} \cup \{r\ e_2 \mid e_2 \in [\![\tau_2]\!]_\bullet\})^*$$

where l and r are variables so that $l \in [\![\tau_1]\!]_\bullet \Rightarrow \mathbb{C}$ and $r \in [\![\tau_2]\!]_\bullet \Rightarrow \mathbb{C}$ by definition. It follows that $e\ \delta\ l\ r \in \mathbb{C}$ as well, meaning that either $e\ \delta\ l\ r \mapsto^* l\ e_1$ for some $e_1 \in [\![\tau_1]\!]$ or $e\ \delta\ l\ r \mapsto^* r\ e_2$ for some $e_2 \in [\![\tau_2]\!]$. Therefore, we have the two possible cases:

$$e =_\eta \Lambda\delta.\lambda l{:}\tau_1 \to \delta.\lambda r{:}\tau_2 \to \delta.e\ \delta\ l\ r =_\beta \Lambda\delta.\lambda l{:}\tau_1 \to \delta.\lambda r{:}\tau_2 \to \delta.l\ e_1 =_\beta \mathit{Left}\ \tau_1\ \tau_2\ e_1$$
$$e =_\eta \Lambda\delta.\lambda l{:}\tau_1 \to \delta.\lambda r{:}\tau_2 \to \delta.e\ \delta\ l\ r =_\beta \Lambda\delta.\lambda l{:}\tau_1 \to \delta.\lambda r{:}\tau_2 \to \delta.r\ e_2 =_\beta \mathit{Right}\ \tau_1\ \tau_2\ e_2$$

where $e_1 \in [\![\tau_1]\!]$ and $e_2 \in [\![\tau_2]\!]$. ◻

12.4.3 Products

Recall that

$$\begin{aligned}
\mathit{Prod}\ \tau_1\ \tau_2 &= \forall\delta.(\tau_1 \to \tau_2 \to \delta) \to \delta \\
\mathit{Pair} &: \forall\alpha.\forall\beta.\alpha \to \beta \to \mathit{Prod}\ \alpha\ \beta \\
\mathit{Pair} &= \Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.\Lambda\delta.\lambda p{:}\alpha \to \beta \to \delta.p\ x\ y
\end{aligned}$$

Theorem 12.5 (Canonicity)**.** *If $\bullet;\bullet \vdash e : \mathit{Prod}\ \tau_1\ \tau_2$ then $e =_{\beta\eta} \mathit{Pair}\ \tau_1\ \tau_2\ e_1\ e_2$ for some $e_1 \in [\![\tau_1]\!]$ and $e_2 \in [\![\tau_2]\!]$.*

Proof. From Corollary 11.1, we know that

$$e \in [\![\mathit{Prod}\ \tau_1\ \tau_2]\!] = [\![\forall\delta.(\tau_1 \to \tau_2 \to \delta) \to \delta]\!] = \forall(\mathbb{C}.([\![\tau_1]\!]_\bullet \Rightarrow [\![\tau_2]\!]_\bullet \Rightarrow \mathbb{C}) \Rightarrow \mathbb{C})$$

In other words, for any type τ, reducibility candidate $\mathbb{C} \in CR$, and expression $e' \in (\llbracket \tau_1 \rrbracket_\bullet \Rightarrow \llbracket \tau_2 \rrbracket_\bullet \Rightarrow \mathbb{C})$, it must be that $e\ \tau\ e' \in \mathbb{C}$. So let's choose the reducibility candidate

$$\mathbb{C} = \{p\ e_1\ e_2 \mid e_1 \in \llbracket \tau_1 \rrbracket_\bullet, e_2 \in \llbracket \tau_2 \rrbracket_\bullet\}^*$$

where p is a variable so that $p \in (\llbracket \tau_1 \rrbracket_\bullet \Rightarrow \llbracket \tau_2 \rrbracket_\bullet \Rightarrow \mathbb{C})$ by definition. It follows that $e\ \tau\ p \in \mathbb{C}$ as well, meaning that $e\ \tau\ p \mapsto^* p\ e_1\ e_2$ for some $e_1 \in \llbracket \tau_1 \rrbracket_\bullet$ and $e_2 \in \llbracket \tau_2 \rrbracket_\bullet$. Therefore

$$e =_\eta \Lambda\delta.\lambda p{:}\tau_1 \to \tau_2 \to \delta.e\ \delta\ p =_\beta \Lambda\delta.\lambda p{:}\tau_1 \to \tau_2 \to \delta.p\ e_1\ e_2 =_\beta \mathit{Pair}\ \tau_1\ \tau_2\ e_1\ e_2$$

where $e_1 \in \llbracket \tau_1 \rrbracket_\bullet$ and $e_2 \in \llbracket \tau_2 \rrbracket_\bullet$. □

12.4.4 Numbers

Recall that

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

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 12.6 (Numericity). *If $\bullet;\bullet \vdash e : \mathit{Nat}$ then $e\ \delta\ z\ s =_\beta s^n\ z$ for some $n \in \mathbb{N}$.*

Proof. From Corollary 11.1, we know that

$$e \in \llbracket \mathit{Nat} \rrbracket = \llbracket \forall\delta.\delta \to (\delta \to \delta) \to \delta \rrbracket = \forall(\mathbb{C}.\mathbb{C} \Rightarrow (\mathbb{C} \Rightarrow \mathbb{C}) \Rightarrow \mathbb{C})$$

In other words, for any type τ, reducibility candidate $\mathbb{C} \in CR$, and terms $e_0 \in \mathbb{C}$ and $e_1 \in \mathbb{C} \Rightarrow \mathbb{C}$, it must be that $e\ \tau\ e_0\ e_1 \in \mathbb{C}$. So let's choose the inductively defined reducibility candidate

$$\mathbb{C} = (\{z\} \cup \{s\ e' \mid e' \in \mathbb{C}\})^*$$

where z and s are variables so that $z \in \mathbb{C}$ and $s \in \mathbb{C} \Rightarrow \mathbb{C}$ by definition. Note that, for every $e \in \mathbb{C}$, $e =_\beta s^n\ z$ for some $n \in \mathbb{N}$ by induction on the definition of $\mathbb{C}$. It follows that $e\ \delta\ s\ z \in \mathbb{C}$, meaning that $e\ \delta\ s\ z =_\beta s^n\ z$ for some $n \in \mathbb{N}$. □

Corollary 12.1 (Canonicity). *If $\bullet;\bullet \vdash e : \mathit{Nat}$ then either $e =_{\beta\eta} \mathit{Zero}$ or $e =_{\beta\eta} \mathit{Succ}\ e'$ for some $\bullet;\bullet \vdash e' : \mathit{Nat}$.*

Proof. From numericity, we know that $e =_{\beta\eta} \Lambda\delta.\lambda z.\lambda s.s^n\ z$ for some $n \in \mathbb{N}$. It follows that either $e =_{\beta\eta} \mathit{Zero}$ or $e =_{\beta\eta} \mathit{Succ}\ e'$ for some e' for some $\bullet;\bullet \vdash e' : \mathit{Nat}$ by induction on n. □

12.5 Relational parametricity

Notation: $\mathit{Expr}^2 = \mathit{Expr} \times \mathit{Expr}$ is the cartesian product containing every pair of expressions, which is also the maximal relation in which all expressions are related.

We can generalize the closure under expansion to work on a binary relation $\mathbb{C} \subseteq \mathit{Expr}^2$ as well as follows:

$$\mathbb{C}^* = \{(e_1, e_1') \mid \exists (e_2, e_2') \in \mathbb{C}. e_1 \mapsto^* e_2 \wedge e_1' \mapsto^* e_2'\}$$

This has analogous properties as the similar closure operation on sets of expressions (i.e. unary relations).

Definition 12.1 (Relation candidates)**.** A binary relation on expressions $\mathbb{C}$ is a *relation candidate* when the following condition is met

- *expansion*: if $e_1 \mapsto^* e_2$ and $e_1' \mapsto^* e_2'$ and $(e_2, e_2') \in \mathbb{C}$ then $(e_1, e_1') \in \mathbb{C}$.

In other words, $\mathbb{C}$ is a relation candidate exactly when

$$\mathbb{C}^* \subseteq \mathbb{C} \subseteq \mathit{Expr}^2$$

We write the set of all relation candidates as

$$\mathit{CR}^2 = \{\mathbb{C} \in \wp(\mathit{Expr}^2) \mid \mathbb{C}^* \subseteq \mathbb{C}\}$$

Likewise, lets generalize the logical operators to work over binary relation candidates.

$$\begin{aligned}
(\_ \Rightarrow^2 \_) &: \mathit{CR}^2 \times \mathit{CR}^2 \to \mathit{CR}^2 \\
\mathbb{A} \Rightarrow^2 \mathbb{B} &= \{(e_1, e_1') \in \mathit{Expr}^2 \mid \forall (e_2, e_2') \in \mathbb{A}.\ (e_1\ e_2, e_1'\ e_2') \in \mathbb{B}\} \\
(\_ \divideontimes^2 \_) &: \mathit{CR}^2 \times \mathit{CR}^2 \to \mathit{CR}^2 \\
\mathbb{A} \divideontimes^2 \mathbb{B} &= \{(e, e') \in \mathit{Expr}^2 \mid (e{\cdot}\pi_1, e'{\cdot}\pi_1) \in \mathbb{A} \wedge (e{\cdot}\pi_2, e'{\cdot}\pi_2) \in \mathbb{B}\} \\
\mathbb{\forall}^2 &: (\mathit{CR}^2 \to \mathit{CR}^2) \to \mathit{CR}^2 \\
\mathbb{\forall}^2(\mathbb{F}) &= \{(e, e') \in \mathit{Expr}^2 \mid \forall \tau, \tau' \in \mathit{Type}, \mathbb{C} \in \mathit{CR}^2.\ (e\ \tau, e'\ \tau') \in \mathbb{F}(\mathbb{C})\} \\
\mathbb{\exists}^2 &: (\mathit{CR}^2 \to \mathit{CR}^2) \to \mathit{CR}^2 \\
\mathbb{\exists}^2(\mathbb{F}) &= \{(\langle \tau, e\rangle, \langle \tau', e'\rangle) \mid \exists \mathbb{C} \in \mathit{CR}^2.\ (e, e') \in \mathbb{F}(\mathbb{C})\}^*
\end{aligned}$$

Adding strings to the language.

$$\begin{aligned}
s \in \mathit{String} &::= \ldots \\
e \in \mathit{Expr} &::= \ldots \mid \text{“}\mathtt{s}\text{”} \\
\tau \in \mathit{Type} &::= \ldots \mid \mathtt{String}
\end{aligned}$$

$$\frac{}{\Gamma \vdash \text{“}\texttt{s}\text{”} : \texttt{String}}$$

Using these binary versions of the logical operations, it is easy to define the interpretation of types as binary relation candidates, where now $\theta : \mathit{TypeVar} \rightharpoonup \mathit{CR}^2$.

$$\begin{aligned}
[\![\alpha]\!]^2_\theta &= \theta(\alpha)\\
[\![\texttt{String}]\!]^2_\theta &= \{(s,s)\}^*\\
[\![\tau_1 \to \tau_2]\!]^2_\theta &= [\![\tau_1]\!]^2_\theta \Rrightarrow^2 [\![\tau_2]\!]^2_\theta\\
[\![\tau_1 \times \tau_2]\!]^2_\theta &= [\![\tau_1]\!]^2_\theta \divideontimes^2 [\![\tau_2]\!]^2_\theta\\
[\![\forall\alpha.\tau]\!]^2_\theta &= \forall^2(\mathbb{C}.\ [\![\tau]\!]^2_{\theta,\mathbb{C}/\alpha})\\
[\![\exists\alpha.\tau]\!]^2_\theta &= \exists^2(\mathbb{C}.\ [\![\tau]\!]^2_{\theta,\mathbb{C}/\alpha})
\end{aligned}$$

The interpretation of environments and judgements are generalized to binary relations accordingly.

$$\begin{aligned}
[\![\Theta]\!]^2 &= \{\Theta\} \to \mathit{CR}^2\\
[\![\Theta;\Gamma]\!]^2_\theta &= \{(\sigma,\sigma) \in \mathit{Subst} \times \mathit{Subst}\\
&\qquad \mid (\forall (x:\tau) \in \Gamma.(x[\sigma], x[\sigma']) \in [\![\tau]\!]^2_\theta)\\
&\qquad \wedge (\forall \alpha \in \Delta.\alpha[\sigma], \alpha[\sigma'] \in \mathit{Type})\}\\
[\![\Theta \vdash \tau : \star]\!]^2 &= \forall\theta \in [\![\Theta]\!]^2.[\![\tau]\!]^2_\theta \in \mathit{CR}^2\\
[\![\Theta;\Gamma \vdash e : \tau]\!]^2 &= \forall\theta \in [\![\Theta]\!]^2.(\sigma,\sigma') \in [\![\Theta;\Gamma]\!]^2_\theta.(e[\sigma], e[\sigma']) \in [\![\tau]\!]^2_\theta
\end{aligned}$$

Theorem 12.7 (Parametricity)**.** *If $\Theta;\Gamma \vdash e : \tau$ is derivable then $[\![\Theta;\Gamma \vdash e : \tau]\!]^2$ is true. In other words, for any*

- *expression e such that $\Gamma \vdash e : \tau$ is derivable,*
- *mapping $\theta : \{\Theta\} \to \mathit{CR}^2$, and*
- *related substitutions (σ, σ') such that $\alpha[\sigma], \alpha[\sigma'] \in \mathit{Type}$ for all $\alpha \in \Theta$ and $(x[\sigma], x[\sigma]) \in [\![\tau']\!]^2_\theta$ for all $(x : \tau') \in \Gamma$,*

it follows that $(e[\sigma], e[\sigma']) \in [\![\tau]\!]^2_\theta$.

Corollary 12.2. *If $\bullet;\bullet \vdash e : \tau$ is derivable then $(e, e) \in [\![\tau]\!]^2_\bullet$.*

12.5.1 Red vs Blue

$\mathit{red\_v\_blue}_{1,2} : \exists\alpha.(\alpha \times \alpha \times (\alpha \to \texttt{String}))$

$\quad \mathit{red\_v\_blue}_1 = \langle \texttt{String}, \langle \text{“}\texttt{red}\text{”}, \text{“}\texttt{blue}\text{”}, \lambda x.x\rangle\rangle$

$\quad \mathit{red\_v\_blue}_2 = \langle \texttt{String} + \texttt{String}, \langle \iota_1 \cdot \text{“”}, \iota_2 \cdot \text{“”}, \lambda x.\ \texttt{case}\ x\ \texttt{of}\ \iota_1 \cdot z \Rightarrow \text{“}\texttt{red}\text{”} \mid \iota_2 \cdot z \Rightarrow \text{“}\texttt{blue}\text{”}\rangle\rangle$

Want to show that $\mathit{red\_v\_blue}_1 \cong \mathit{red\_v\_blue}_2$ in some appropriate sense.

Theorem 12.8. $(\textit{red\_v\_blue}_1, \textit{red\_v\_blue}_2) \in [\![\exists\alpha.(\alpha \times \alpha \times (\alpha \to \texttt{String}))]\!]^2$

Proof. By Corollary 12.2. We must choose some relation $\mathbb{A}$ such that $\textit{red\_v\_blue}_1$ and $\textit{red\_v\_blue}_2$ are related when α is instantiated with $\mathbb{A}$. So consider the following relation $\mathbb{A}$:

$$\mathbb{A} = \{(\text{“}\texttt{red}\text{”}, \iota_1 \cdot \text{“”}), (\text{“}\texttt{blue}\text{”}, \iota_2 \cdot \text{“”})\}^*$$

To show

$$\begin{aligned}(\textit{red\_v\_blue}_1, \textit{red\_v\_blue}_2) &\in [\![\exists\alpha.(\alpha \times \alpha \times (\alpha \to \texttt{String}))]\!] \\ &= \exists^2(\mathbb{C}.\mathbb{C} \divideontimes^2 \mathbb{C} \divideontimes^2 (\mathbb{C} \Rightarrow^2 [\![\texttt{String}]\!]^2))\end{aligned}$$

it suffices to show that

$$\begin{aligned}&(\langle \text{“}\texttt{red}\text{”}, \text{“}\texttt{blue}\text{”}, e\rangle, \langle \iota_1 \cdot \text{“”}, \iota_2 \cdot \text{“”}, e'\rangle) \in \mathbb{A} \divideontimes^2 \mathbb{A} \divideontimes^2 (\mathbb{A} \Rightarrow^2 [\![\texttt{String}]\!]^2) \\ &\qquad \texttt{where}\; e = \lambda x.x \\ &\qquad\qquad e' = \lambda x.\, \texttt{case}\, x\, \texttt{of}\, \iota_1 \cdot z \Rightarrow \text{“}\texttt{red}\text{”} \mid \iota_2 \cdot z \Rightarrow \text{“}\texttt{blue}\text{”}\end{aligned}$$

In other words, we may show that $(\text{“}\texttt{red}\text{”}, \iota_1 \cdot \text{“”}) \in \mathbb{A}$, $(\text{“}\texttt{blue}\text{”}, \iota_2 \cdot \text{“”}) \in \mathbb{A}$, and $(e\ e_1, e'\ e_1') \in [\![\texttt{String}]\!]^2$ for all $(e_1, e_1') \in \mathbb{A}$. The first two facts follow immediately from the definition of $\mathbb{A}$. For the third fact, consider an arbitrary $(e_1, e_1') \in \mathbb{A}$, where it must be that either $e_1 \mapsto^* \text{“}\texttt{red}\text{”}$ and $e_1' \mapsto^* \iota_1 \cdot \text{“”}$ or $e_1 \mapsto^* \text{“}\texttt{blue}\text{”}$ and $e_1' \mapsto^* \iota_2 \cdot \text{“”}$. In the first case, we have the reduction

$$\begin{aligned}e\ e_1 &\mapsto e_1 \mapsto^* \text{“}\texttt{red}\text{”} \\ e'\ e_1' &\mapsto \texttt{case}\, e_1'\, \texttt{of}\, \iota_1 \cdot z \Rightarrow \text{“}\texttt{red}\text{”} \mid \iota_2 \cdot z \Rightarrow \text{“}\texttt{blue}\text{”} \\ &\mapsto^* \texttt{case}\, \iota_1 \cdot \text{“”}\, \texttt{of}\, \iota_1 \cdot z \Rightarrow \text{“}\texttt{red}\text{”} \mid \iota_2 \cdot z \Rightarrow \text{“}\texttt{blue}\text{”} \\ &\mapsto \text{“}\texttt{red}\text{”}\end{aligned}$$

and in the second case we have the reduction

$$\begin{aligned}e\ e_1 &\mapsto e_1 \mapsto^* \text{“}\texttt{blue}\text{”} \\ e'\ e_1' &\mapsto \texttt{case}\, e_1'\, \texttt{of}\, \iota_1 \cdot z \Rightarrow \text{“}\texttt{red}\text{”} \mid \iota_2 \cdot z \Rightarrow \text{“}\texttt{blue}\text{”} \\ &\mapsto^* \texttt{case}\, \iota_2 \cdot \text{“”}\, \texttt{of}\, \iota_1 \cdot z \Rightarrow \text{“}\texttt{red}\text{”} \mid \iota_2 \cdot z \Rightarrow \text{“}\texttt{blue}\text{”} \\ &\mapsto \text{“}\texttt{blue}\text{”}\end{aligned}$$

so in either case, $(e\ e_1, e'\ e_1') \in [\![\texttt{String}]\!]^2$. □