Programming Languages

Lectures by Yoni Zohar
Summary by Ari Feiglin (ari.feiglin@gmail.com)

Contents

1 Semantics of Expressions

In this section, we will define a simple programming language called **While**. The syntax of **While** has five categories: numerals **Num**, variables **Var**, arithmetic expressions **Aexp**, boolean expressions **Bexp**, and statements **Stm**. The structure for **Aexp**, **Bexp**, and **Stm** are given respectively as follows:

$$\begin{array}{lll} \textbf{(Aexp)} & a ::= & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\ \textbf{(Bexp)} & b ::= & \mathsf{true} \mid \mathsf{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\ \textbf{(Stm)} & S ::= & x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S \end{array}$$

Explicitly, arithmetic expressions are defined recursively as so:

(1) numerals and variables are arithmetic expressions,

(2) if $a_1, a_2 \in$ **Aexp** then $a_1 + a_2, a_1 \star a_2, a_1 - a_2 \in$ **Aexp**.

Similarly boolean expressions are defined recursively

(1) true and false are boolean expressions,

(2) if $a_1, a_2 \in$ **Aexp** then $a_1 = a_2, a_1 \leq a_2 \in$ **Bexp**,

(3) if $b_1, b_2 \in$ **Bexp** then $\neg b_1, b_1 \wedge b_2 \in$ **Bexp**.

And finally statements:

(1) if x is a variable and a is an arithmetic expression then $x := a$ is a statement,

(2) `skip` is a statement,

(3) if S_1, S_2 are statements, then $S_1; S_2$ is a statement,

(4) if $b \in$ **Bexp** and S_1, S_2 are statements then $\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$ and $\texttt{while}\ b\ \texttt{do}\ S_1$ are statements.

So for example, if x, y are variables then

$$x := 5;\ y := 10;\ \texttt{while}\ x \leq 10\ \texttt{do}\ \texttt{if}\ 0 \leq y\ \texttt{then}\ y := y - x\ \texttt{else}\ \texttt{skip};\ x := x + y$$

is a statement. What exactly it does is not important yet, but what is important is that it's a statement.

1.1 Definition

A **state** is a function **Var** $\longrightarrow \mathbb{Z}$, define **State** to be the set of all states (all functions **Var** $\longrightarrow \mathbb{Z}$).

1.2 Definition

We define the function $\mathcal{A}\colon$ **Aexp** $\longrightarrow$ (**State** $\longrightarrow \mathbb{Z}$), which assigns to every **Aexp** its numerical value when evaluated at a specific state. We define $\mathcal{A}$ recursively on the structure of **Aexp**:

(1) for a numeral n, $\mathcal{A}[\![n]\!]s = n$,

(2) for a variable x, $\mathcal{A}[\![x]\!]s = s\,x$,

(3) $\mathcal{A}[\![a_1 + a_2]\!]s = \mathcal{A}[a_1]s + \mathcal{A}[a_2]s$,

(4) $\mathcal{A}[\![a_1 \star a_2]\!]s = \mathcal{A}[a_1]s \cdot \mathcal{A}[a_2]s$,

(5) $\mathcal{A}[\![a_1 - a_2]\!]s = \mathcal{A}[a_1]s - \mathcal{A}[a_2]s$.

So for example, if s is a state which maps $x \to 1$ and $y \to 3$ then

$$\begin{aligned} \mathcal{A}[\![x + ((x \star y) + 1)]\!]s = \mathcal{A}[\![x]\!]s + \mathcal{A}[\![(x \star y) + 1]\!] = \mathcal{A}[\![x]\!]s + \mathcal{A}[\![x \star y]\!]s + \mathcal{A}[\![1]\!]s \\ = \mathcal{A}[\![x]\!]s + \mathcal{A}[\![x]\!]s \cdot \mathcal{A}[\![y]\!]s + \mathcal{A}[\![1]\!]s = 1 + 1 \cdot 3 + 1 = 5 \end{aligned}$$

1.3 Definition

We define $\mathcal{B}\colon \mathbf{Bexp} \longrightarrow (\mathbf{State} \longrightarrow \{tt, ff\})$ which assigns to every boolean expression a boolean value when evaluated at a specific state. Similar to $\mathcal{A}$, we define it recursively:

(1) $\mathcal{B}[\![\mathsf{true}]\!]s = tt$, $\mathcal{B}[\![\mathsf{false}]\!]s = ff$,

(2) $\mathcal{B}[\![a_1 = a_2]\!]s$ is tt if $\mathcal{A}[a_1]s = \mathcal{A}[a_2]s$ and ff otherwise,

(3) $\mathcal{B}[\![a_1 \leq a_2]\!]s$ is tt if $\mathcal{A}[\![a_1]\!]s \leq \mathcal{A}[\![a_2]\!]s$ and ff otherwise,

(4) $\mathcal{B}[\![\neg b]\!]s = \neg\mathcal{B}[\![b]\!]s$,

(5) $\mathcal{B}[\![b_1 \wedge b_2]\!]s = \mathcal{B}[\![b_1]\!]s \wedge \mathcal{B}[\![b_2]\!]s$.

Where $\neg$ and $\wedge$ are defined as one would expect on $\{tt, ff\}$.

1.4 Definition

Let s be a state, x a variable, and v a number. Define $s[x \mapsto v]$ to be the state defined by

$$s[x \mapsto v]y = \begin{cases} v & x = y \\ s\,y & \text{else} \end{cases}$$

So $s[x \mapsto v]$ is the state obtained by overwriting the value of x in s to be v.

We now define the semantics of **While**. A program in **While** is a statement and a state, then the statement is run and a new state is produced. Formally we define a transition relation $\langle \cdot, \cdot \rangle \to \cdot \subseteq (\mathbf{Stm} \times \mathbf{State} \times \mathbf{State})$, here we read $\langle S, s\rangle \to s'$ as "s' is derivable from S, s". We write

$$\frac{\langle S_1, s_1\rangle \to s_1', \ldots \langle S_n, s_n\rangle \to s_n'}{\langle S, s\rangle \to s'} \quad \text{if} \ldots$$

To mean that if $\langle S_i, s_i\rangle \to s_i'$ hold for $1 \leq i \leq n$ and the condition in $\ldots$ holds, then $\langle S, s\rangle \to s'$. If there are no conditions, then we will forgo the horizontal line and just write $\langle S, s\rangle \to s'$.

We now list the transitions:

$[\text{ass}_{\text{ns}}]$ $\quad \langle x := a, s\rangle \to s\big[x \mapsto \mathcal{A}[\![a]\!]s\big]$

$[\text{skip}_{\text{ns}}]$ $\quad \langle \mathtt{skip}, s\rangle \to s$

$[\text{comp}_{\text{ns}}]$ $\quad \dfrac{\langle S_1, s\rangle \to s' \qquad \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$

$[\text{if}^{\text{tt}}_{\text{ns}}]$ $\quad \dfrac{\langle S_1, s\rangle \to s'}{\langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2\rangle \to s'} \quad \text{if } \mathcal{B}[\![b]\!]s = tt$

$[\text{if}^{\text{tt}}_{\text{ns}}]$ $\quad \dfrac{\langle S_2, s\rangle \to s'}{\langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2\rangle \to s'} \quad \text{if } \mathcal{B}[\![b]\!]s = ff$

$[\text{while}^{\text{tt}}_{\text{ns}}]$ $\quad \dfrac{\langle S, s\rangle \to s' \qquad \langle \mathtt{while}\ b\ \mathtt{do}\ S, s'\rangle \to s''}{\langle \mathtt{while}\ b\ \mathtt{do}\ S\rangle \to s''} \quad \text{if } \mathcal{B}[\![b]\!]s = tt$

$[\text{while}^{\text{ff}}_{\text{ns}}]$ $\quad \langle \mathtt{while}\ b\ \mathtt{do}\ S, s\rangle \to s \text{ if } \mathcal{B}[\![b]\!]s = ff$

We can compute transitions by successive applications of axioms (transitions without assumptions) and transitions.

1.5 Definition

The **deductive tree** of $\langle S, s\rangle \to s'$ is a tree whose root is $\langle S, s\rangle \to s'$ and the leaves are axioms. Every inner node is a transition which is a consequence of its children. We define $\langle S, s\rangle \to s'$ if the sequent has a deductive tree.

The deductive tree will be written with the root on the bottom. For example, let s_0 be the state such that $x \mapsto 5$ and $y \mapsto 7$, define $s_1 = s_0[z \mapsto 5]$, $s_2 = s_1[x \mapsto 7]$, and $s_3 = s_2[y \mapsto 5]$. We claim that $\langle (z := x; x := y); y := z, s_0\rangle \to s_3$.

$$\cfrac{\cfrac{\langle z := x, s_0\rangle \to s_1 \ \ \text{ass} \qquad \langle x := y, s_1\rangle \to s_2 \ \ \text{ass}}{\langle z := x; x := y, s_0\rangle \to s_2}\,\text{comp} \qquad \langle y := z, s_2\rangle \to s_3 \ \ \text{ass}}{\langle (z := x; x := y); y := z, s_0\rangle \to s_3}\,\text{comp}$$

1.6 Definition

We say that two statements S_1, S_2 are **semantically equivalent** if for every two states s, s', $\langle S_1, s\rangle \to s'$ if and only if $\langle S_2, s\rangle \to s'$.

So for example, S is semantically equivalent to $S; \texttt{skip}$ for every $S \in \textbf{Stm}$. We will prove this: suppose $\langle S, s\rangle \to s'$ then it has a deductive tree T, and so

$$\cfrac{\cfrac{T}{\langle s, s\rangle \to s'} \qquad \langle \texttt{skip}, s'\rangle \to s' \ \ \text{skip}}{\langle s; \texttt{skip}, s\rangle \to s'}$$

So we have that $\langle S; \texttt{skip}, s\rangle \to s'$. Now suppose the converse, but its deductive tree must end with

$$\cfrac{\cfrac{T}{\langle s, s\rangle \to s'} \qquad \langle \texttt{skip}, s'\rangle \to s' \ \ \text{skip}}{\langle s; \texttt{skip}, s\rangle \to s'}$$

and so $\langle S, s\rangle \to s'$. ■

In general if we want to prove something about the transition relation, we can induct on the shape of derivation trees: first we prove it for all simple derivation trees (which have a single axioms); then for each rule, assume the property holds for its premises and then show it holds for the conclusion of the rule.

1.7 Theorem

If $\langle S, s\rangle \to s'$ and $\langle S, s\rangle \to s''$ then $s' = s''$.

Proof: first we prove it for simple derivation trees, which are formed from $[\text{ass}_{\text{ns}}]$ or $[\text{skip}_{\text{ns}}]$. Then we proceed to the other rules.

(1) $[\text{ass}_{\text{ns}}]$: suppose S is $x := a$ and then s' is $s[x \mapsto \mathcal{A}[\![a]\!]s]$, which is unique ($s''$ must also be this).

(2) $[\text{skip}_{\text{ns}}]$: S is $\texttt{skip}$ and so $s' = s$.

(3) $[\text{comp}_{\text{ns}}]$: assume $\langle S_1; S_2, s\rangle \to s'$ holds because $\langle S_1, s\rangle \to s_0$ and $\langle S_2, s_0\rangle \to s'$ for some s_0. The only rule which can be applied to get $\langle S_1; S_2, s\rangle \to s''$ is $[\text{comp}_{\text{ns}}]$, so there is a state s_1 such that $\langle S_1, s\rangle \to s_1$ and $\langle S_2, s_1\rangle \to s''$. But by induction, $s_1 = s_0$ and then applying induction again, $s' = s''$.

(4) $[\text{if}^{\text{tt}}_{\text{ns}}]$: assume that $\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to s'$ holds because $\mathcal{B}[\![b]\!]s = tt$ and $\langle S_1, s\rangle \to s'$. Since $\mathcal{B}[\![b]\!]s = tt$, the only rule which can be applied to get $\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to s''$ is $[\text{if}^{\text{tt}}_{\text{ns}}]$, so $\langle S_1, s\rangle \to s''$, and by induction $s' = s''$.

(5) $[\text{if}^{\text{ff}}_{\text{ns}}]$: similar.

(6) $[\text{while}^{\text{tt}}_{\text{ns}}]$: assume that $\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to s'$ because $\mathcal{B}[\![b]\!]s = tt$, $\langle S, s\rangle \to s_0$, and $\langle \texttt{while}\ b\ \texttt{do}\ S, s_0\rangle \to s'$ for some s_0. The only rule which could be applied to get $\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to s''$ is $[\text{while}^{\text{tt}}_{\text{ns}}]$ in lieu of

$\mathcal{B}[\![b]\!]s = tt$. So there exists a s_1 such that $\langle S, s\rangle \to s_1$ and $\langle \texttt{while}\ b\ \texttt{do}\ S, s_1\rangle \to s'$. But then by induction $s_0 = s_1$ and by induction again, $s' = s''$.

(7) [while$^{\text{ff}}_{\text{ns}}$]: straightforward. ■

Note that not every statement can derive a state: for example `while true do skip` has an infinite derivation tree and thus derives no state (for any initial state s). Thus we could define $\langle\cdot,\cdot\rangle$ to be a partial function

$$\langle\cdot,\cdot\rangle\colon \mathbf{Stm} \longrightarrow (\mathbf{State} \hookrightarrow \mathbf{State})$$

which accepts a statement and a state and returns the state which it derives, if it exists.

2 Untyped Lambda Calculus

Lambda calculus is a way of formalizing computations, it generalizes the concept of functions. A function in lambda calculus has the form $\lambda x.t$ and should be thought of a function $x \mapsto t(x)$, in a language like OCaml, this corresponds to a function definition of the form $\texttt{fun}\, x \to t$. It is built from syntax, and we then utilize semantics to give this syntax meaning.

2.1 Definition

Let V be an infinite set of variable symbols, then terms in lambda calculus are constructed recursively as follows:

(1) every variable is an term,

(2) if $x \in V$ is a variable and t is an term, then $\lambda x.t$ is an term,

(3) if t_1 and t_2 are terms, then so is $t_1 t_2$.

Notice that lambda calculus terms have the *unique reconstruction property*: every term t has one of the above forms, and such a form is *unique*. We can then construct functions on lambda terms via term recursion, as given by the following examples.

2.2 Definition

Given an term of the form $\lambda x.t$, every instance of x in the term t is called **bound**, and all other instances are **free**. Formally we can define the set of free variables in an term recursively as follows:

(1) for an term of the form x for a variable x, $\mathit{var}(x) = \{x\}$, $\mathit{free}(x) = \{x\}$, $\mathit{bnd}(x) = \varnothing$,

(2) for an term of the form $\lambda x.t$, $\mathit{var}(\lambda x.t) = \mathit{var}(t) \cup \{x\}$, $\mathit{free}(\lambda x.t) = \mathit{free}(t) \setminus \{x\}$, and $\mathit{bnd}(\lambda x.t) = \mathit{bnd}(t) \cup \{x\}$,

(3) for an term of the form $t_1 t_2$, $\mathit{var}(t_1 t_2) = \mathit{var}(t_1) \cup \mathit{var}(t_2)$, $\mathit{free}(t_1 t_2) = \mathit{free}(t_1) \cup \mathit{free}(t_2)$ and $\mathit{bnd}(t_1 t_2) = \mathit{bnd}(t_1) \cup \mathit{bnd}(t_2)$.

Alternatively, a **bound occurrence** of a variable x in t is an occurrence which occurs in t' where $\lambda x.t'$ is a subterm of t. A **free occurrence** is an occurrence which is not bound. Then $\mathit{free}(t)$ is the set of all variables which occur free in t, bndt is the set of all variables which occur bound in t.

So for example, let $t = (\lambda x.\lambda y.x)x\,z$, then $\mathit{var}(t) = \{x, y, z\}$, $\mathit{free}(t) = \{x, z\}$, $\mathit{bnd}(t) = \{x, y\}$. Here the x and y in $\lambda x.\lambda y.x$ are bound occurrences, and the x and z following it (in $x\,z$) are free. Notice that always $\mathit{var}(t) = \mathit{free}(t) \cup \mathit{bnd}(t)$, but as the above example shows, these two sets are not always disjoint. A proof of this union is done via term induction: prove it for $t = x$, then for $t = \lambda x.t'$, then finally for $t = t_1 t_2$.

(1) for $t = x$, $\mathit{var}(t) = \{x\}$, $\mathit{free}(t) = \{x\}$, and $\mathit{bnd}(t) = \varnothing$, so the union holds.

(2) for $t = \lambda x.t'$, $\mathit{var}(t) = \mathit{var}(t') \cup \{x\}$ which by induction is equal to $\mathit{free}(t') \cup \mathit{bnd}(t') \cup \{x\}$. Now $\mathit{free}(t) = \mathit{free}(t') \setminus \{x\}$, $\mathit{bnd}(t) = \mathit{bnd}(t') \cup \{x\}$ and so we see that $\mathit{free}(t) \cup \mathit{bnd}(t) = \mathit{var}(t)$ as required.

(3) for $t = t_1 t_2$, $\mathit{var}(t) = \mathit{var}(t_1) \cup \mathit{var}(t_2)$ which by induction is $\mathit{free}(t_1) \cup \mathit{free}(t_2) \cup \mathit{bnd}(t_1) \cup \mathit{bnd}(t_2) = \mathit{free}(t) \cup \mathit{bnd}(t)$.

2.3 Definition

An term without free variables is called a **combinator**. The **identity combinator** is the combinator $\mathsf{id} = \lambda x.x$.

Suppose we'd like to take a term t and substitute x with another term t'. For example, suppose t' is the variable z, then λ`y.x` should become λ`y.z`. But then what should λ`x.x` become? Surely not λ`x.z`, as that alters the entire interpretation of the function. So variables should be substituted only at free occurrences. But what about if t' were x and t was λ`x.y`, then substituting at y gives λ`x.x`, which once again changes the meaning of the function. So we should only substitute at free occurrences, if the λ-variable is not free in the term being substituted.

2.4 Definition

Let t, t' be terms and x a variable. Then $t[x \mapsto t']$ is the term obtained by substituting x with t' according to the following rules:

(1) $x[x \mapsto t'] = t'$,

(2) $y[x \mapsto t'] = y$ if y is a variable distinct from x,

(3) $(\lambda x.t)[x \mapsto t'] = \lambda x.t$,

(4) $(\lambda y.t)[x \mapsto t'] = \lambda y.(t[x \mapsto t'])$ if $y \neq x$ and $y \notin \mathit{free}(t')$,

(5) $(t_1\, t_2)[x \mapsto t'] = t_1[x \mapsto t']\, t_2[x \mapsto t']$.

But then what would the substitution $(\lambda y.x\, y)[x \mapsto y\, z]$ look like? Well y is free in the substituted term, so it doesn't match any of the above conditions. In such a case we take upon ourselves the following convention:

Convention (α-equivalence)

Terms that differ only in the named of bound variables are equivalent.

This means that we can view $\lambda y.x\, y$ as $\lambda w.x\, w$ and so the substitution becomes $\lambda w.y\, z\, w$. The rules for α-equivalence is that if $\lambda x.t$ is equivalent to $\lambda y.t[x \mapsto y]$ if $y \notin \mathit{var} t$.

2.5 Definition

A term of the form $(\lambda x.t)t'$ is called a **redex**. A term of the form $\lambda x.t$ is called a **abstraction**. We define the β **reduction** on terms which maps redexes to terms by $(\lambda x.t)t' \xrightarrow{\beta} t[x \mapsto t']$ where $t[x \mapsto t']$ is the term obtained by substituting t' at all the free occurrences of x. A **value** is an abstraction or variable.

For example, $(\lambda x.x)y \to y$, and

$$\big(\lambda x.(\lambda x.x)x\big)(u\, r) \to (\lambda x.x)(u\, r) = u\, r$$

When performing a β-reduction, we need to consider the order with which we perform the reduction. There are 4 ways:

(1) *Full β-reduction*, in which any redex can be reduced at any time. So at each step, we can arbitrarily choose a redex and reduce it. For example, take

$$\texttt{(}\lambda\texttt{x.x) ((}\lambda\texttt{x.x) (}\lambda\texttt{z.(}\lambda\texttt{x.x) z))}$$

which is just $\mathsf{id}(\mathsf{id}(\lambda z.\mathsf{id}z))$. This term contains three redexes:

$$\underline{\mathsf{id}(\mathsf{id}(\lambda z.\mathsf{id}\ z))},\quad \mathsf{id}(\underline{\mathsf{id}(\lambda z.\mathsf{id}\ z)}),\quad \mathsf{id}(\mathsf{id}(\lambda z.\underline{\mathsf{id}\ z}))$$

So we can choose for example to begin from the innermost redex and move outward:

$$\begin{aligned} & \mathsf{id}(\mathsf{id}(\lambda z.\underline{\mathsf{id}z})) \\ \to\ & \mathsf{id}(\underline{\mathsf{id}(\lambda z.z)}) \\ \to\ & \underline{\mathsf{id}(\lambda z.z)} \\ \to\ & \lambda z.z \end{aligned}$$

which cannot be reduced any more.

(2) *Normal order*, in which the leftmost outermost redex is reduced first. So using the same example as above:

$$\begin{aligned} & \underline{\mathsf{id}(\mathsf{id}(\lambda z.\mathsf{id}z))} \\ \to\ & \underline{\mathsf{id}(\lambda z.\mathsf{id}z)} \\ \to\ & \lambda z.\underline{\mathsf{id}z} \\ \to\ & \lambda z.z \end{aligned}$$

The rules for normal order reduction are as follows:

$$\frac{}{(\lambda x.t)s \to t[x \mapsto s]}\ ,\quad \frac{t \to t'}{ts \to t's}\ \text{if } t \text{ is not a value}\ ,\quad \frac{t \to t'}{\lambda x.t \to \lambda x.t'}\ \text{if } t \text{ is not a value}$$

(**3**) *Call-by-name*, which is similar to normal order but it performs no reductions inside abstractions. Using the same example:

$$\begin{aligned} & \underline{\mathtt{id(id(\lambda z.idz))}} \\ \to\ & \underline{\mathtt{id(\lambda z.idz)}} \\ \to\ & \mathtt{\lambda z.idz} \end{aligned}$$

The rules for call-by-name reduction are as follows:

$$\frac{}{\mathtt{(\lambda x.t)s} \;\to\; \mathtt{t[x\mapsto s]}}\ , \qquad \frac{\mathtt{t\to t'}}{\mathtt{ts} \;\to\; \mathtt{t's}} \text{ if } \mathtt{t} \text{ is not a value}$$

(**4**) *Call-by-value*, which is the most commonly used in programming languages, like call-by-name, but a redex is reduced only when its right-hand side has already been reduced to a *value* (a term which cannot be reduced further, in this lambda calculus these are only abstractions).

$$\begin{aligned} & \mathtt{id(\underline{id(\lambda z.idz)})} \\ \to\ & \underline{\mathtt{id(\lambda z.idz)}} \\ \to\ & \mathtt{\lambda z.idz} \end{aligned}$$

The rules for call-by-value reduction are

$$\frac{}{\mathtt{(\lambda x.t)v} \;\to\; \mathtt{t[x\mapsto v]}} \text{ if } \mathtt{v} \text{ is a value}\ , \qquad \frac{\mathtt{s\to s'}}{\mathtt{(\lambda x.t)s} \;\to\; \mathtt{(\lambda x.t)s'}} \text{ if } \mathtt{s} \text{ is not a value}\ ,$$

$$\frac{\mathtt{t\to t'}}{\mathtt{ts} \;\to\; \mathtt{t's}} \text{ if } \mathtt{t} \text{ is not a value}$$

In this course we use call-by-value, since it is the most commonly used evaluation strategy.

Notice that in lambda calculus, all functions accept a single parameter as input. As in OCaml, to write a function which accepts multiple functions, we write one which accepts a single input and returns a function which also accepts a single input. So for example $f = \lambda x.\lambda y.x$ can then be called like $f\,u\,r$ and will return u after two β-reductions.

We now define booleans in lambda calculus (called Church booleans):

$$\mathsf{tru} = \lambda t.\lambda f.t, \qquad \mathsf{fls} = \lambda t.\lambda f.f$$

So tru accepts two arguments and returns the first, fls accepts two and returns the second. We now define

$$\mathtt{test} = \lambda b.\lambda m.\lambda n.\, b\,m\,n$$

So `test` accepts three arguments, the first b is a boolean (either tru or fls), and it applies it to the other two arguments. So for example

$$\mathtt{test}\,\mathsf{tru}\,v\,w = (\lambda b.\lambda m.\lambda n.\, b\,m\,n)\mathsf{tru}\,v\,w \to (\lambda m.\lambda n.\mathsf{tru}\,m\,n)v\,w \to (\lambda n.\mathsf{tru}\,v\,n)w \to \mathsf{tru}\,v\,w \to v$$

This doesn't do much, it just returns the first argument (after the boolean) if the boolean is true, and the second if it is false.

We can define a more interesting combinator

$$\mathtt{and} = \lambda b.\lambda c.b\,c\,\mathsf{fls}$$

Here b, c are booleans. Then if b is tru, $\mathtt{and}\,b\,c \to c$ after a β-reduction, and otherwise it will reduce to c. So if c is false, then $\mathtt{and}\,b\,c \to c = \mathsf{fls}$ and if c is true then it reduces to $c = \mathsf{tru}$, and if b is false then $\mathtt{and}\,b\,c \to b\,c\,\mathsf{fls} \to \mathsf{fls}$. So `and` functions as one would expect it to.

Utilizing booleans, we can encode pairs of values as terms:

$$\begin{aligned} \mathtt{pair} &= \lambda \mathsf{f}.\lambda \mathsf{s}.\lambda \mathsf{b}.\mathsf{b}\,\mathsf{f}\,\mathsf{s} \\ \mathtt{fst} &= \lambda \mathsf{p}.\mathsf{p}\,\mathsf{tru} \\ \mathtt{snd} &= \lambda \mathsf{p}.\mathsf{p}\,\mathsf{fls} \end{aligned}$$

Notice then that

$$\begin{array}{rll}
& \texttt{fst(pair v w)} & \\
= & \texttt{fst(}\underline{\texttt{(}\lambda\texttt{f.}\lambda\texttt{s.}\lambda\texttt{b.b f s)}}\texttt{ v w)} & \text{by definition} \\
\to & \texttt{fst(}\underline{\texttt{(}\lambda\texttt{s.}\lambda\texttt{b.b v s) w}}\texttt{)} & \beta\text{-reduction on underlined redex} \\
\to & \texttt{fst(}\underline{\lambda\texttt{b.b v w}}\texttt{)} & \beta\text{-reduction on underlined redex} \\
= & \underline{\texttt{(}\lambda\texttt{p.p tru)(}\lambda\texttt{b.b v w)}} & \text{by definition} \\
\to & \underline{\texttt{(}\lambda\texttt{b.b v w)tru}} & \beta\text{-reduction on underlined redex} \\
\to & \texttt{tru v w} & \beta\text{-reduction on underlined redex} \\
\to & \texttt{v} & \text{by definition of } \texttt{tru}
\end{array}$$

In a similar manner we can show that `snd(pair v w)`$\to$`w`.

We now demonstrate how we can represent numbers in lambda calculus, via Church numerals:

$$\begin{array}{ll}
\texttt{c}_0 = & \lambda\texttt{s.}\lambda\texttt{z.z} \\
\texttt{c}_1 = & \lambda\texttt{s.}\lambda\texttt{z.s z} \\
\texttt{c}_2 = & \lambda\texttt{s.}\lambda\texttt{z.s(s z)} \\
\texttt{c}_3 = & \lambda\texttt{s.}\lambda\texttt{z.s(s(s z))} \\
\text{etc.} &
\end{array}$$

In general if we write $\texttt{s}^n$ z for $\texttt{s(s(}\cdots\texttt{s z}\cdots\texttt{))}$ (n times), then $\texttt{c}_n = \lambda\texttt{s.}\lambda\texttt{z.s}^n\ \texttt{z}$. So each number n is represented by the combinator $\texttt{c}_n$ which accepts `s,z` and applies `s` n times to `z`. Notice that $\texttt{c}_0 = \textsf{fls}$, which is reminiscent of the fact that false and zero mean the same thing in many compiled languages.

Let us define

$$\texttt{scc=}\ \lambda\texttt{n.}\lambda\texttt{s.}\lambda\texttt{z.s(n s z)}$$

We see then that

$$\texttt{scc}\ \texttt{c}_n\ \texttt{s z = }\lambda\texttt{s.}\lambda\texttt{z.s(c}_n\texttt{ s z) s z = s(s}^n\texttt{ z) = s}^{n+1}\texttt{ z = c}_{n+1}\texttt{ z s}$$

so `scc` $\texttt{c}_n$ and $\texttt{c}_{n+1}$ are equivalent in the sense that they operate the same on the same input. But bare in mind: `scc` $\texttt{c}_n$ = $\lambda\texttt{s.}\lambda\texttt{z.s(c}_n\texttt{ s z)}$ which is not *equal* to $\texttt{c}_{n+1}$.

Similarly we can define

$$\texttt{plus=}\ \lambda\texttt{n.}\lambda\texttt{m.}\lambda\texttt{s.}\lambda\texttt{z.m s (n s z)}$$

so that `plusn m s z` will apply `s n s z` m times, resulting in $\texttt{s}^m\texttt{s}^n\texttt{z} = \texttt{s}^{n+m}\texttt{z}$ as desired. Similarly we define

$$\texttt{times = }\lambda\texttt{n.}\lambda\texttt{m.m (plus n) c}_0$$

so that `times n m` will apply `plusn` m times to $\texttt{c}_0$, resulting in $n + n + \cdots + n + 0 = n \cdot m$. In a similar vein, we can define `pow` = $\lambda\texttt{n.}\lambda\texttt{m.m (times n) c}_1$, so that `pow` $\texttt{c}_n\ \texttt{c}_m$ is equal to $\texttt{c}_{n^m}$.

To test if a numeral is zero, we'd like to find a functions `ss` and `zz` such that applying `ss` one or more times to `zz` yields false, while not applying it at all yields true. That way when we do $\texttt{c}_n$ `ss zz`, it will result in `tru` only if `ss` was never applied, meaning $n = 0$. Necessarily then `zz` must be `tru`, and have `ss` be the function which maps every input to fls. So we define

$$\texttt{iszro=}\ \lambda\texttt{n.n (}\lambda\texttt{x.}\textsf{fls}\texttt{) tru}$$

To define the predecessor combinator, we must be a bit more clever than with the successor. One implementation is

$$\begin{array}{rl}
\texttt{zz} = & \texttt{pair c}_0\ \texttt{c}_0 \\
\texttt{ss} = & \lambda\texttt{p.pair(snd p)(plus }\texttt{c}_1\texttt{ (snd p))} \\
\texttt{prd} = & \lambda\texttt{m.fst(m ss zz)}
\end{array}$$

The idea here is that applying `ss` to a (n, m) will result in $(m, m + 1)$. So starting from $(0, 0)$, you get $(0, 1)$ then $(1, 2)$ then $(2, 3)$ and so on. In general $\texttt{ss}^n\texttt{z} = (n - 1, n)$ for $n \geq 1$ and so the predecessor is just the first value.

Using the predecessor combinator we can define a subtraction combinator similar to addition:

$$\texttt{sub=}\ \lambda\texttt{m.}\lambda\texttt{n.m prdn}$$

Notice though that `sub` cannot give negative numbers, after all we didn't define negative numbers, so if $n \leq m$ then $\texttt{c}_n\texttt{-c}_m$ is just $\texttt{c}_0$. Thus we can define

$$\begin{array}{rl}
\texttt{leq} = & \lambda\texttt{m.}\lambda\texttt{n.iszro(sub m n)} \\
\texttt{equal} = & \lambda\texttt{m.}\lambda\texttt{n.and(leq n m) (leq m n)}
\end{array}$$

2.6 Definition

A term without a redex is called a **normal form**. The normal form of a term t is the normal form obtained through β reduction. A term without a normal form is called **divergent**.

For example, the normal form of $\texttt{(}\lambda\texttt{x.}\lambda\texttt{y.x)y}$ can be reduced to $\lambda\texttt{y.y}$ which is its normal form. One example of a divergent combinator is

$$\texttt{omega= }(\lambda\texttt{x.x x})(\lambda\texttt{x.x x})$$

Since a single β reduction gives you back `omega`, which gives what is essentially an infinite loop. We can also define the following combinator

$$\texttt{fix= }\lambda\texttt{f.}(\lambda\texttt{x. f}(\lambda\texttt{y. x x y}))\ (\lambda\texttt{x. f}(\lambda\texttt{y. x x y}))$$

Suppose we'd like to write a function to compute factorials, which can be written as

```
if n=0 then 1
else n * factorial(n-1)
```

The idea is to unravel the function definition, to get something of the form

```
if n=0 then 1
else n * (if n-1=0 then 1
          else (n-1) * (if n-2=0 then 1
                        else (n-2) * ...))
```

Using Church numerals, we get

$$\begin{array}{l}
\texttt{test (equal n } c_0\texttt{)}\\
\quad c_1\\
\quad \texttt{times n (test (equal (prd n) } c_0\texttt{)}\\
\qquad\quad c_1\\
\qquad\quad \texttt{times (prd n) (test (equal (prd (prd n)) } c_0\texttt{)}\\
\qquad\qquad\qquad\qquad c_1\\
\qquad\qquad\qquad\qquad \texttt{times (prd (prd n)) (...)))}
\end{array}$$

Then we define

$$\begin{aligned}
\texttt{g} &= \lambda\texttt{fct.}\lambda\texttt{n. test (equal n } c_0\texttt{) } c_1 \texttt{ (times n (fct (prd n)))}\\
\texttt{factorial} &= \texttt{fix g}
\end{aligned}$$

Let us give an example run of `factorial` c_3:

$$\begin{array}{lll}
 & \texttt{factorial } c_3 & \\
= & \texttt{fix g } c_3 & \\
\to & \texttt{h h } c_3 & \text{where } \texttt{h=}\lambda\texttt{x.g(}\lambda\texttt{y.x x y)}\\
\to & \texttt{g fct } c_3 & \text{where } \texttt{fct=}\lambda\texttt{y. h h y}\\
\to & \big(\lambda\texttt{n. test(equal n } c_0\texttt{) } c_1 \texttt{ (times n (fct (prd n)))}\big)c_3 & \\
\to & \texttt{test(equal } c_3\ c_0\texttt{) } c_1 \texttt{ (times } c_3 \texttt{ (fct (prd } c_3\texttt{)))} & \\
\to & \texttt{times } c_3 \texttt{ (fct (prd } c_3\texttt{))} & \\
\to & \texttt{times } c_3 \texttt{ (fct } c_2\texttt{)} & \\
\to & \texttt{times } c_3 \texttt{ (h h } c_2\texttt{)} & \\
\to & \texttt{times } c_3 \texttt{ (g fct } c_2\texttt{)} & \text{similar to how } \texttt{h h } c_3 \text{ can be reduced to } \texttt{g fct } c_3\\
\to & \texttt{times } c_3 \texttt{ (times } c_2 \texttt{ (g fct } c_1\texttt{))} & \text{by the same process that we did for } c_3\\
\to & \texttt{times } c_3 \texttt{ (times } c_2 \texttt{ (times } c_1 \texttt{ (g fct } c_0\texttt{)))} & \\
\to & \texttt{times } c_3 \texttt{ (times } c_2 \texttt{ (times } c_1 \texttt{ (test (equal } c_0\ c_0\texttt{) } c_1 \texttt{ ...)))} & \\
\to & \texttt{times } c_3 \texttt{ (times } c_2 \texttt{ (times } c_1\ c_1\texttt{))} & \\
\to & c_6 &
\end{array}$$

Let us prove that this works. Suppose we have a recurrence $\texttt{r=}\lambda\texttt{x.}\langle\texttt{code with r}\rangle$, let us use the notation $\langle\texttt{r c}\rangle$ to mean that within the recurrence, `r` is called on the value `c`. Let us define $\texttt{g=}\lambda\texttt{r.}\lambda\texttt{x.}\langle\texttt{code with r}\rangle$, which is like `r` but it accepts the function it should run on. So if we were to define `r`, then `r` and `g r` would be functionally the same. We claim then that `r=fix g` is a term which is equivalent to `r` (does the same thing). Let us reduce it a bit on some term `c`

$$\begin{array}{lll}
 & \texttt{r c} & \\
= & \texttt{fix g c} & \\
\to & \texttt{h h c} & \text{where } \texttt{h=}\lambda\texttt{x.g(}\lambda\texttt{y.x x y)}\\
\to & \texttt{g r' c} & \text{where } \texttt{r'=}\lambda\texttt{y.h h y}
\end{array}$$

Now we claim that `g r' c` gives the same result as `r c`, which we will prove on the number of recursive calls that `r c` makes. If we were to reduce this one more time, we'd get $\langle\texttt{code with r'}\rangle$ `c`, but since `r` makes no recursive calls on the input `c`, this functions the same as $\langle\texttt{code with r}\rangle$ `c`, which is `r c`. Now, suppose that on the first recursive call, the program calls `r' c'`, meaning for `r` it would call `r c'`. Now `r' c' = h h c' = g r' c'`, and by our inductive hypothesis `g r' c' = r c'`, so the code performs the same.

We can also define the *Y-combinator*:

$$\mathtt{Y} = \lambda\mathtt{f.}(\lambda\mathtt{x.f(x\ x)})(\lambda\mathtt{x.f(x\ x)})$$

Which can similarly perform recursion. Like `fix`, it is a *fixed-point* combinator, which is a combinator fix such that $f(\mathsf{fix} f) = \mathsf{fix} f$. Indeed:

$$\begin{array}{lll}
 & \mathtt{Y\ g} & \\
= & (\lambda\mathtt{f.}(\lambda\mathtt{x.f(x\ x)})(\lambda\mathtt{x.f(x\ x)}))\ \mathtt{g} & \text{by definition} \\
\to & (\lambda\mathtt{x.g(x\ x)})(\lambda\mathtt{x.g(x\ x)}) & \text{by } \beta\text{-reduction} \\
\to & \mathtt{g}((\lambda\mathtt{x.g(x\ x)})\ (\lambda\mathtt{x.g(x\ x)})) & \text{by } \beta\text{-reduction} \\
= & \mathtt{g(Y\ g)} & \text{by the second equality}
\end{array}$$

Though the final equality is only true up to β-reduction, meaning that `Y g` and `g(Y g)` both reduce to a similar term, not to one another. This is the trait which allows for recursion.

3 Simply Typed Lambda Calculus

3.1 Definition

We define **types** in our simply typed lambda calculus recursively as follows:

(1) Bool is a type,

(2) if T_1, T_2 are types, so is $T_1 \to T_2$.

Here $\to$ is right-associative, meaning $T_1 \to T_2 \to T_3$ is taken to mean $T_1 \to (T_2 \to T_3)$.

3.2 Definition

We define terms once again recursively:

(1) every variable is a term,

(2) if x is a variable, t a term, and T a type, then $\lambda x\colon T.t$ is a term (here the type refers to the variable, we will explain later),

(3) if t_1, t_2 are terms then so is $t_1\, t_2$,

(4) $\mathsf{true}, \mathsf{false}$ are terms,

(5) if t_1, t_2, t_3 are terms, then so is $\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$.

Let us define $\mathsf{id} = \lambda x\colon \mathsf{Bool}.x$, then id is a term.

3.3 Definition

We define β-reduction on simply typed redexes as follows:

(1) $(\lambda x\colon T.t)t' \xrightarrow{\beta} t[x \mapsto t']$,

(2) $\texttt{if}\ \mathsf{true}\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \xrightarrow{\textit{hello}} \textit{there} \rightarrow t_1$,

(3) $\texttt{if}\ \mathsf{false}\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \xrightarrow{\beta} t_2$.

So for example, let $\texttt{f}=\lambda\texttt{x}\colon\mathsf{Bool}\to\mathsf{Bool}.\lambda\texttt{y}\colon\mathsf{Bool}.\texttt{x y}$, then

$$\begin{array}{lll}
 & \texttt{f}\ \mathsf{idtrue} & \\
= & \underline{(\lambda\texttt{x}\colon\mathsf{Bool}\to\mathsf{Bool}.\lambda\texttt{y}\colon\mathsf{Bool}.\texttt{x y})\mathsf{id}}\ \mathsf{true} & \text{definition} \\
\to & \underline{(\lambda\texttt{y}\colon\mathsf{Bool}.\mathsf{id}\ \texttt{y})\ \mathsf{true}} & \beta\text{-reduction on the underlined redex} \\
\to & \underline{\mathsf{id}\ \mathsf{true}} & \beta\text{-reduction on the underlined redex} \\
\to & \mathsf{true} & \beta\text{-reduction on the underlined redex}
\end{array}$$

And

$$\begin{array}{lll}
 & \texttt{f}\ \mathsf{true}\ \mathsf{id} & \\
= & \underline{(\lambda\texttt{x}\colon\mathsf{Bool}\to\mathsf{Bool}.\lambda\texttt{y}\colon\mathsf{Bool}.\texttt{x y})\mathsf{true}}\ \mathsf{id} & \text{definition} \\
\to & \underline{(\lambda\texttt{y}\colon\mathsf{Bool}.\mathsf{true}\ \texttt{y})\ \mathsf{id}} & \beta\text{-reduction on the underlined redex} \\
\to & \mathsf{true}\ \mathsf{id} & \beta\text{-reduction on the underlined redex}
\end{array}$$

We'd like to assign to terms a type. Suppose Γ is a set containing elements of the form $x\colon T'$ where x ranges over all the variables (and each variable occurs only once), then we write $\Gamma \vdash t\colon T$ to mean that if we assume Γ then t has the type T. If Γ is a such a set, we write $\Gamma, t'\colon T'$ to mean $\Gamma \cup \{t'\colon T'\}$, and instead of $\varnothing \vdash t\colon T$ we write $\vdash t\colon T$. We utilize Gentzen-style rules to form a deductive system for deducing the type of an abstraction. The first rule is for abstractions,

$$\frac{\Gamma, \texttt{x}\colon\texttt{T} \vdash \texttt{t}\colon\texttt{T'}}{\Gamma \vdash \lambda\texttt{x}\colon\texttt{T.t} : \texttt{T}\to\texttt{T'}} \tag{T-Abs}$$

This just means that if we assume `x` has type T then `t` has type T', then we can conclude that λ`x:T.t` has type T$\to$T'. Suppose for example we take the language C, and we set `t=x+x`, then if `x:float` we can conclude that `t:float` as well, so λ`x:float.x+x` has type `float`$\to$`float`. But if `x` is of type `int`, then `t` is of the same type and λ`x:int.x+x` has type `int`$\to$`int`. Importantly, these examples are given to give some intuition for the rule, they are not valid λ-terms!

Obviously if $\mathtt{x}:\mathsf{T}$ is already in Γ then Γ should deduce $\mathtt{x}:\mathsf{T}$:

$$\frac{\mathtt{x}:\mathsf{T}\in\Gamma}{\Gamma\vdash\mathtt{x}:\mathsf{T}} \tag{T-Var}$$

We also need a rule for applications:

$$\frac{\Gamma\vdash\mathtt{t}:\mathsf{T'}\to\mathsf{T}\mid\Gamma\vdash\mathtt{t'}:\mathsf{T'}}{\Gamma\vdash\mathtt{t\ t'}:\mathsf{T}} \tag{T-App}$$

Which means that if `t` is a function $\mathsf{T'}\to\mathsf{T}$ and `t'` has type `T'`, then the application `t t'` has type `T`. And for conditionals

$$\frac{\Gamma\vdash\mathtt{t}_1:\mathsf{Bool}\mid\Gamma\vdash\mathtt{t}_2:\mathsf{T}\mid\Gamma\vdash\mathtt{t}_3:\mathsf{T}}{\Gamma\vdash\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3\ :\ \mathsf{T}} \tag{T-If}$$

And of course true and false are Boolean types:

$$\frac{}{\Gamma\vdash\mathsf{true}:\mathsf{Bool}},\qquad\frac{}{\Gamma\vdash\mathsf{false}:\mathsf{Bool}} \tag{T-True),(T-False}$$

Let us now show that $\vdash\lambda\mathtt{x}:\mathsf{Bool}.\ \mathtt{if\ x\ then\ true\ else\ x}:\mathsf{Bool}\to\mathsf{Bool}$. We form a deductive tree:

$$\dfrac{\dfrac{\mathtt{x}:\mathsf{Bool}\vdash\mathtt{x}:\mathsf{Bool}\ \ \text{T-Var}\qquad\mathtt{x}:\mathsf{Bool}\vdash\mathsf{true}:\mathsf{Bool}\ \ \text{T-True}\qquad\mathtt{x}:\mathsf{Bool}\vdash\mathsf{false}:\mathsf{Bool}\ \ \text{T-False}}{\mathtt{x}:\mathsf{Bool}\vdash\mathtt{if\ x\ then\ true\ else\ x}}\ \text{T-If}}{\vdash\lambda\mathtt{x}:\mathsf{Bool}.\ \mathtt{if\ x\ then\ true\ else\ x}:\mathsf{Bool}\to\mathsf{Bool}}\ \text{T-Abs}$$

3.4 Definition

A term t is **well-typed** if its type can be deduced from the empty set, ie. $\vdash\mathtt{t}:\mathsf{T}$ for some T.

3.5 Definition

A term of the form true, false, or $\lambda\mathtt{x}:\mathsf{T}.\mathtt{t}$ (an abstraction) is called a **value**.

3.6 Lemma (Progress Lemma)

If t is a closed (meaning it has no free variables) well-typed term. Then t is either a value or there is some t' with $t\to t'$ through a step of β-reduction.

Proof: if t is a boolean or an abstraction, then it is a value. Otherwise $\mathtt{t}=\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$, then `t` is closed if and only if all $\mathtt{t}_i$ are and by the derivation rule $\mathtt{t}_1:\mathsf{Bool}$ which means that $\mathtt{t}_1$ must be a Boolean, and so `t` can be reduced. Finally if $\mathtt{t}=\mathtt{t}_1\ \mathtt{t}_2$ then `t` is closed and well-typed, then $\mathtt{t}_1:\mathtt{T'}\to\mathtt{T}$ and $\mathtt{t}_2:\mathtt{T'}$, which means that $\mathtt{t}_1$ is either a value or can be reduced, likewise for $\mathtt{t}_2$. If either can be reduced, then so too can `t` (since if $\mathtt{t}\to\mathtt{t}_0$ then $\mathtt{t\ t'}\to\mathtt{t}_0\ \mathtt{t'}$ and similar for `t'`). If both are a values, then $\mathtt{t}_1$ is of the form $\lambda\mathtt{x}.\mathtt{t}_{11}$ and so it can be applied to a value and reduced. ∎

3.7 Lemma (Substitution Lemma)

If $\Gamma,\mathtt{x}:\mathsf{T'}\vdash\mathtt{t}:\mathsf{T}$ and $\Gamma\vdash\mathtt{t'}:\mathsf{T'}$, then $\Gamma\vdash\mathtt{t[x\mapsto t']}:\mathsf{T}$.

Proof: by induction on the derivation of $\Gamma,\mathtt{x}:\mathsf{T'}\vdash\mathtt{t}:\mathsf{T}$.

(1) T-Var: so $\mathtt{t}=\mathtt{z}$ and $\mathtt{z}:\mathsf{T}\in\Gamma,\mathtt{x}:\mathsf{T}'$. If $\mathtt{z}=\mathtt{x}$ then $\mathtt{t}=\mathtt{z}=\mathtt{x}$, so $\mathsf{T}=\mathsf{T}'$ and $\mathtt{t}[\mathtt{x}\mapsto\mathtt{t}']=\mathtt{t}'$. We must prove that $\Gamma\vdash\mathtt{t}':\mathsf{T}$, but we know that $\mathtt{t}':\mathsf{T}'=\mathsf{T}$ so this holds. If $\mathtt{z}\neq\mathtt{x}$ then $\mathtt{t}[\mathtt{x}\mapsto\mathtt{t}']=\mathtt{z}$ and this is satisfied trivially.

(2) T-Abs: then $\mathtt{t}=\lambda\mathtt{y}:\mathsf{T}_2.\mathtt{t}_1$, $\mathsf{T}=\mathsf{T}_2\to\mathsf{T}_1$, and $\Gamma,\mathtt{x}:\mathsf{T}'\vdash\lambda\mathtt{y}:\mathsf{T}_2.\mathtt{t}_1:\mathsf{T}$ so that $\Gamma,\mathtt{x}:\mathsf{T}',\mathtt{y}:\mathsf{T}_2\vdash\mathtt{t}_1:\mathsf{T}_1$. We may assume by convention that $\mathtt{x}\neq\mathtt{y}$ and that `y` is not free in t'. Since $\Gamma\vdash\mathtt{t}':\mathsf{T}'$, we get $\Gamma,\mathtt{y}:\mathsf{T}_2\vdash\mathtt{t}':\mathsf{T}'$, and so by the induction hypothesis $\Gamma,\mathtt{y}:\mathsf{T}_2\vdash\mathtt{t}[\mathtt{x}\mapsto\mathtt{t}']:\mathsf{T}_1$. By T-Abs, we get $\Gamma\vdash\lambda\mathtt{y}.\mathtt{t}_1[\mathtt{x}\mapsto\mathtt{t}']:\mathsf{T}$, but $\lambda\mathtt{y}.\mathtt{t}_1[\mathtt{x}\mapsto\mathtt{t}']=(\lambda\mathtt{y}.\mathtt{t}_1)[\mathtt{x}\mapsto\mathtt{t}']=\mathtt{t}[\mathtt{x}\mapsto\mathtt{t}']$ as required.

(3) T-True and T-False are immediate since $\mathtt{t} = \mathsf{true}$ or false and $\mathsf{T} = \mathsf{Bool}$ and so $\mathtt{t}[\mathtt{x} \mapsto \mathtt{t}'] = \mathtt{t}$.

(4) T-If is straightforward. ∎

3.8 Theorem (Preservation Theorem)

If $\Gamma \vdash \mathtt{t} \colon \mathsf{T}$ and $\mathtt{t} \to \mathtt{t}'$ by β-reduction, then $\Gamma \vdash \mathtt{t}' \colon \mathsf{T}$.

Proof: suppose $\mathtt{t} = (\lambda \mathtt{x} \colon \mathsf{T}_1 . \mathtt{t}_1)\mathtt{t}_2 \colon \mathsf{T}_2$ then let us look at the derivation of t:

$$\dfrac{\dfrac{\Gamma, \mathtt{x}:\mathsf{T}_1 \vdash \mathtt{t}_1 : \mathsf{T}_2}{\Gamma \vdash \lambda \mathtt{x}:\mathsf{T}_1 . \mathtt{t}_1 :\ \mathsf{T}_1 \to \mathsf{T}_2}\ \text{T-Abs} \qquad \Gamma \vdash \mathtt{t}_2 : \mathsf{T}_2}{\Gamma \vdash (\lambda \mathtt{x}:\mathsf{T}_1 . \mathtt{t}_1)\ \mathtt{t}_2 :\ \mathsf{T}_2}\ \text{T-App}$$

Our goal is to show $\Gamma \vdash \mathtt{t}_1[\mathtt{x} \mapsto \mathtt{t}_2]$. But we have that $\Gamma, \mathtt{x} \colon \mathsf{T}_1 \vdash \mathtt{t}_1 \colon \mathsf{T}_2$ and $\Gamma \vdash \mathtt{t}_2 \colon \mathsf{T}_2$ which gives us by the substitution lemma precisely this. ∎

3.9 Definition

A term t **can be normalized** if there exists a value t' such that t can be reduced to t'.

Our goal is to prove that a closed well-typed term can be normalized. To do so we require some further mechanisms and proofs.

3.10 Definition

Let T be a type, then we define the predicate R_T on terms recursively as follows:

(1) R_{Bool} is the set of all terms of type Bool which can be normalized.

(2) $R_{\mathsf{T}_1 \to \mathsf{T}_2}$ is the set of all terms t of type $\mathsf{T}_1 \to \mathsf{T}_2$ that can be normalized and if $R_{\mathsf{T}_1}(s)$ then $R_{\mathsf{T}_2}(ts)$.

3.11 Lemma

Suppose $\vdash t \colon \mathsf{T}$ and t can be reduced to t' then $R_\mathsf{T}(t)$ if and only if $R_\mathsf{T}(t')$.

Proof: by induction on T. For $\mathsf{T} = \mathsf{Bool}$ then if $t \colon \mathsf{Bool}$ and t can be normalized, so can t' and $t' \colon \mathsf{Bool}$ by the preservation theorem. And if $t' \colon \mathsf{Bool}$ then $t \colon \mathsf{Bool}$ again by the preservation theorem.

Now suppose $\mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2$, if $R_\mathsf{T}(t)$ then it is obvious by the preservation theorem that $t' \colon \mathsf{T}$. Now let $R_{\mathsf{T}_1}(s)$ then we must show that $R_{\mathsf{T}_2}(t's)$, but since $ts \to t's$ both of their types must be T_2 as required. ∎

3.12 Corollary

Suppose $x_1 \colon \mathsf{T}_1, \ldots, x_n \colon \mathsf{T}_n \vdash t \colon \mathsf{T}$ and $v_1, \ldots, v_n$ are values of type T_i such that $R_{\mathsf{T}_i}(r_i)$. Then for $t' = t[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$, $R_\mathsf{T}(t')$.

Proof: by induction on the derivation $x_1 \colon \mathsf{T}_1, \ldots, x_n \colon \mathsf{T}_n \vdash t \colon \mathsf{T}$. For T-Var this is simply because $t = x_i$ and $\mathsf{T} = \mathsf{T}_i$ for some i, and the result is immediate. For T-Abs, $t = \lambda x \colon \mathsf{S}_1 . s_2$, $x_1 \colon \mathsf{T}_1, \ldots, x_n \colon \mathsf{T}_n, x \colon \mathsf{S}_1 \vdash s_2 \colon \mathsf{S}_2$, and $\mathsf{T} = \mathsf{S}_1 \to \mathsf{S}_2$.

3.13 Theorem

Every closed well-typed term t can be normalized.

Proof: in the book. ∎

4 λ-OCaml

We define a language λ-OCaml similar to untyped λ-calculus as follows:

4.1 Definition

Terms in λ-OCaml are defined recursively as follows:

(1) all variables are terms,

(2) if $\mathtt{x}$ is a variable and $\mathtt{t}$ a term, then $\mathtt{fun\ x{\to}t}$ is a term,

(3) if $\mathtt{t}_1$ is a term, then $\mathtt{t}_1\ \mathtt{t}_2$ is a term.

This is obviously equivalent to untyped λ-calculus where instead of $\lambda\mathtt{x.t}$ we write $\mathtt{fun\ x{\to}t}$. We also define types:

4.2 Definition

Suppose we have an infinite set of type variables, then a type is defined recursively as follows:

(1) all type variables are types,

(2) if T and S are types, so is $\mathsf{T}{\to}\mathsf{S}$.

Similar to typed λ-calculus we define the *type relation* $\Gamma \vdash \mathtt{t}:\mathsf{T}$ where $\mathtt{t}$ is a term, T is a type, and Γ is a variable type set of which contains elements of the form $\mathtt{x}:\mathsf{S}$ for variables $\mathtt{x}$ and types S, such that every variable is given a single type. It is a Gentzen calculus defined using the rules:

$$\frac{\mathtt{x}:\mathsf{T} \in \Gamma}{\Gamma \vdash \mathtt{x}:\mathsf{T}} \tag{O-Var}$$

$$\frac{\Gamma \vdash \mathtt{t}_{12}:\mathsf{T}_1{\to}\mathsf{T}_2 \mid \Gamma \vdash \mathtt{t}_1:\mathsf{T}_1}{\Gamma \vdash \mathtt{t}_{12}\mathtt{t}_1:\mathsf{T}_2} \tag{O-App}$$

$$\frac{\Gamma \vdash \mathtt{x}:\mathsf{T} \mid \Gamma \vdash \mathtt{t}:\mathsf{S}}{\Gamma \vdash (\mathtt{fun\ x{\to}t}):\mathsf{T}{\to}\mathsf{S}} \tag{O-Abs}$$

Notice that this is similar to simply typed λ-calculus except for O-Abs, where instead of viewing what type has $\mathtt{t}$ has under the assumption that $\mathtt{x}$ has type T, we give them both a type under the plain assumptions in Γ.

4.3 Definition

The problem of **type inference** is the problem of finding mapping between terms and types. Its input is a term t, and its output is a variable type set Γ and a map m between subterms of t (including t) such that $\Gamma \vdash t' : m(t')$ for all subterms t'.

We will solve this problem in three steps: (1) creating a system of equations between types, (2) solving the system, and (3) converting the solution to the appropriate Γ and m.

4.4 Definition

A term t is called **normalized** if for every two subterms $\mathtt{t}_1\ =\ \mathtt{fun\ x{\to}t}_{11}$ and $\mathtt{t}_2\ =\ \mathtt{fun\ y{\to}t}_{22}$, $\mathtt{x}$ and $\mathtt{y}$ are distinct variables.

By α-equivalence, every term has an equivalent normalized term.

4.5 Definition

Let t be a term, let us define the set of equations A_t as follows: for every subterm t' correspond a unique type variable α, then

(1) if α and β correspond to different occurrences of the same subterm, then $\alpha = \beta \in A_t$,

(2) suppose $t_1 t_2$ is a subterm such that α is the variable of t_1, β of t_2, and γ of $t_1 t_2$, then $\alpha = \beta \to \gamma \in A_t$,

(3) for every subterm `fun x→t'`, if α is the variable of x, β of t', and γ of `fun x→t'`, then $\gamma = \alpha \to \beta \in A_t$.

For example, let `t` be `(fun x→x)y`, then let us map the subterms to type variables as follows:

$$y \mapsto \alpha_y, \quad x \mapsto \alpha_x^1, \quad x \mapsto \alpha_x^2, \quad \texttt{fun x→x} \mapsto \alpha_f, \quad t \mapsto \alpha_t$$

Then

$$A_t = \{\alpha_x^1 = \alpha_x^2,\ \alpha_f = \alpha_x^1 \to \alpha_x^2,\ \alpha_f = \alpha_y \to \alpha_t\}$$

Now that we have finished step (1), we skip step (2) and progress to step (3). We will return to step (2) later.

4.6 Definition

A **substitution** is a function σ which maps between type terms such that $\sigma(\mathsf{T}_1 \to \mathsf{T}_2) = \sigma(\mathsf{T}_1) \to \sigma(\mathsf{T}_2)$. σ **preserves** an equality $\mathsf{T}_1 = \mathsf{T}_2$ if $\sigma(\mathsf{T}_1) = \sigma(\mathsf{T}_2)$, and it preserves a set of equations if it preserves every equality in the set.

So in the example above, one such substitution which preserves A_t is $\sigma(\alpha_x^1) = \sigma(\alpha_x^2) = \sigma(\alpha_y) = \sigma(\alpha_t) = \alpha$, and $\sigma(\alpha_f) = \alpha \to \alpha$.o

4.7 Definition

Let t be a term and β a function which maps subterms t' to their type variables. Suppose A_t is the resulting set of equations, and σ a substitution which preserves it. Then we define

$$\Gamma_\sigma^\beta = \{x{:}\,\sigma(\beta(x)) \mid x \in \mathit{vart}\}$$

So using the above example where β is the map

$$\beta\colon \qquad y \mapsto \alpha_y, \quad x \mapsto \alpha_x^1, \quad x \mapsto \alpha_x^2, \quad \texttt{fun x→x} \mapsto \alpha_f, \quad t \mapsto \alpha_t$$

Then using the above substitution σ, we have that

$$\Gamma_\sigma^\beta = \{x{:}\,\alpha,\ y{:}\,\alpha\}$$

4.8 Theorem

Let t be a term, β a correspondence between subterms and type variables, and σ a substitution which preserves A_t. Then for every subterm t' of t,

$$\Gamma_\sigma^\beta \vdash t'{:}\,\sigma(\beta(t'))$$

Thus if we define $m := \sigma \circ \beta$ and $\Gamma := \Gamma_\sigma^\beta$, we have a solution to the problem of type inference for t. And if there is no σ which preserves A_t then there is no solution to the problem of type inference for t.

Proof: by induction on t'.

(1) If t' is a variable then $t'{:}\,\sigma(\beta(t'))$ is in Γ_σ^β and thus this follows from O-Var.

(2) If t' is of the form `fun` $x \to t''$, then let $\alpha_1, \alpha_2, \alpha_3$ be the types of x, t'', t' respectively. Then $\alpha_3 = \alpha_1 \to \alpha_2$ is an equation in A_t so $\sigma(\alpha_3) = \alpha(\alpha_1) \to \sigma(\alpha_2)$. In other words, $\sigma(\beta(t')) = \sigma(\beta(x)) \to \sigma(\beta(t''))$. Now, by induction we have that

$$\Gamma_\sigma^\beta \vdash x\colon \sigma(\beta(x)), \quad \Gamma_\sigma^\beta \vdash t''\colon \sigma(\beta(t''))$$

So applying O-Abs yields

$$\Gamma_\sigma^\beta \vdash t'\colon \sigma(\beta(x)) \to \sigma(\beta(t'')) = \sigma(\beta(t'))$$

as required.

(3) if $t' = t_1 t_2$ then this follows similarly to the above case.

If m solves the problem of type inference, then define $\sigma = m \circ \beta^{-1}$ and we claim that this is a substitution which preserves A_t. We split into cases by the type of equations in A_t:

(1) Equations arising from different occurrences of the same subterm and so m will map this term to the same type, independent of the occurrence.

(2) Equations arising from $t' =$ `fun` $x \to t''$, then this follows from O-Abs.

(3) Equations arising from $t_1 t_2$ follows from O-App. ■

So to solve step (2) all we must do is find a suitable substitution. This is called the problem of *unification*: given a set of equations of type variables, we must find a substitution which preserves it.

The unification algorithm, due to Hindley-Milner, functions as follows (its code written in OCaml):

```
type id = string

type term =
  | Var of id
  | Term of id * (term list)

type substitution = (id * term) list
```

Let us take a quick second to understand the types here. Firstly `id` is simply an alias for `string`. `terms` (denoted τ) are type terms, whose formal definition is:

$$\tau ::= \alpha \mid C\tau \ldots \tau$$

α is the set of type variables, and C is a set of *type functions*. So for example C may contain the 0-ary `int` and `string` which represent types of integers and strings respectively. Another example is `Map` which is a 2-ary type function: `Map string int` is a hashmap from `strings` to `ints`. Yet another example is `Set`: `Set int` is a set of `ints`. In this C, we can chain type functions, so for example `Map (Set string) int` is a type term.

Hindley-Milner imposes only the restriction that C must include the type function $\to$ which represents a function. Instead of $\to \tau\tau'$ though, we write $\tau \to \tau'$ (use infix instead of polish notation).

A `substitution` is simply a map from `Vars` to terms.

```
let rec occurs (x : id) (t : term) : bool =
  match t with
  | Var y -> x = y
  | Term(_,s) -> List.exists (occurs x) s
```

`occurs` takes a variable x and a term t and checks if x occurs in t. It does so as follows: if t is a variable y, then return if $x = y$. Otherwise $t = Ct_1 \ldots t_n$, so return if x occurs in any t_i.

```
let rec subst (s : term) (x : id) (t : term) : term =
  match t with
  | Var y -> if x = y then s else t
  | Term(f,u) -> Term(f, List.map (subst s x) u)
```

`subst s x t`

corresponds to the substitution `t[x↦s]`. So recursively, if t is the variable y, if $x = y$ then $t[x \mapsto s] = s$ otherwise t remains the same. And

$$(Ct_1 \dots t_n)[x \mapsto s] = C(t_1[x \mapsto s]) \dots (t_n[x \mapsto s])$$

```
let apply (s : substitution) (t : term) : term =
  List.fold_right (fun (x,u) -> subst u x) s t
```

Here, if $s = [(x_1, t_1), \dots, (x_n, t_n)]$ we want to apply s to a term t. Then what we want in the end is to get $t[x_n \mapsto t_n] \cdots [x_1 \mapsto t_1]$ which is what this does.

Now we get to the meat of the algorithm: the code which actually unifies the a list of equations. First we have the function `unify_one` which unifies a single equation $s = t$. Then using this we define `unify`.

```
let rec unify_one (s : term) (t : term) : substitution =
  match (s,t) with
  | (Var x, Var y) -> if x = y then [] else [(x,t)]
  | (Term(f,sc), Term(g,tc)) ->
    if f = g && List.length sc = List.length tc
    then unify (List.combine sc tc)
    else failwith "not unifiable: head symbol conflict"
  | (Var x, Term(_,_) as t) | (Term(_,_) as t, Var x) ->
    if occurs x t
    then failwith "not unifiable: circularity"
    else: [(x,t)]

and unify (e : (term * term) list) : substitution =
  match e with
  | [] -> []
  | (x,y) :: t ->
    let s2 = unify t in
    let s1 = unify_one (apply s2 x) (apply s2 y) in
    s1 @ s2
```

So the algorithm is as follows: given a list of equivalences `e=(x,y)::t` first unify the equivalences in `t` recursively to get a substitution `s2`. Apply this substitution to `x` and `y` and unify them.

To unify a single equivalence $s = t$, we split into cases:

(1) if both $s = x$ and $t = y$ are variables, then if they are the same variable no unification is needed. Otherwise we simply have x be substituted with t.

(2) If $s = Cs_1 \dots s_n$ and $t = C't_1 \dots t_m$, we must have that $C = C'$ and $n = m$ (we cannot unify `Set` α with `Map int` β for example). If such is the case, we unify the list $[(s_1, t_1), \dots, (s_n, t_n)]$ recursively, since we now have the equivalences $s_i = t_i$.

(3) If $s = x$ is a variable and $t = Ct_1 \dots t_n$ is a compound term (or vice versa), then we cannot have that x occurs in t (we cannot unify `Set` α with α for example). If such is the case (that x does not occur in t), then we simply have that x is substituted with t.

Notice that the same equivalence can have multiple unifiers. For example the equivalence between $f\ x\ (g\ y)$ and $f\ (g\ z)\ w$ has the unifiers: we can take $S = [x \mapsto g\ z,\ w \mapsto g\ y]$. Applying this to both yields $f\ (g\ z)\ (g\ y)$. But we can also have the unifier $T = [x \mapsto g(f\ a\ b),\ y \mapsto f\ b\ a,\ z \mapsto f\ a\ b,\ w \mapsto g(f\ b\ a)]$. Both terms then substitute out to $f\ (g\ (f\ a\ b))\ (g\ (f\ b\ a))$.

4.9 Definition

The **most general unifier** (mgu) for a set of equations A_t is a unifier σ such that for every other unifier σ' there exists a substitution σ'' such that $\sigma' = \sigma\sigma''$. Meaning that all other unifiers can be obtained from σ using further substitutions.

In the above example, S is an mgu and $T = S[z \mapsto f\ a\ b,\ y \mapsto f\ b\ a]$. Hindley-Milner's algorithm returns an mgu for the set of equivalences.

5 Closure

5.1 Definition

An **environment** is a list $(x_1 : v_1) :: \cdots :: (x_n : v_n)$ where x_i is a variable and v_i is a value (in whatever language, so a number, boolean, function, etc). Given an environment E, we define $E(x)$ to be the value given to x by E (ie. E is of the form $\cdots :: (x : E(x)) :: \cdots$).

We define a ternary relation between environments, expressions, and values:

$$E \vdash e \Rightarrow v$$

which is to be read as "the expression e has value v in environment E". We define this using the following Gentzen-style rules:

$$\frac{}{E \vdash v \Rightarrow v}\ (\textsc{Val}) \qquad \frac{E(x) = v}{E \vdash x \Rightarrow v}\ (\textsc{Var})$$

$$\frac{E \vdash e_1 \Rightarrow v_1 \mid \cdots \mid E \vdash e_n \Rightarrow v_n}{E \vdash (e_1, \ldots, e_n) \Rightarrow (v_1, \ldots, v_n)}\ (\text{N-}\textsc{Tuple})$$

$$\frac{E \vdash e_1 \Rightarrow \mathsf{true} \mid E \vdash e_2 \Rightarrow v}{E \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Rightarrow v}\ (\textsc{Cond1}) \qquad \frac{E \vdash e_1 \Rightarrow \mathsf{false} \mid E \vdash e_3 \Rightarrow v}{E \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Rightarrow v}\ (\textsc{Cond2})$$

These should all be pretty self-explanatory rules.

Now, for a function we must somehow give it a value which captures all the values in the environment, thus we create new values of the form $\langle E, f\rangle$ where E is an environment and f is a function. This is called the *closure* of f in E. We continue developing rules for $\vdash$:

$$\frac{}{E \vdash (\texttt{fun } x \to e) \Rightarrow \langle E, (\texttt{fun } x \to e)\rangle}\ (\textsc{Fun1})$$

$$\frac{}{E \vdash (\texttt{fun } (x_1, \ldots, x_n) \to e) \Rightarrow \langle E, (\texttt{fun } (x_1, \ldots, x_n) \to e)\rangle}\ (\textsc{Fun2})$$

So we give to a function f the value of its closure $\langle E, f\rangle$ in the environment E.

$$\frac{E \vdash e_1 \Rightarrow \langle E', (\texttt{fun } x \to e)\rangle \mid E \vdash e_2 \Rightarrow v' \mid (x : v') :: E' \vdash e \Rightarrow v}{E \vdash (e_1\ e_2) \Rightarrow v}\ (\textsc{App1})$$

$$\frac{E \vdash e_1 \Rightarrow \langle E', (\texttt{fun } (x_1, \ldots, x_n) \to e)\rangle \mid E \vdash e_2 \Rightarrow (v_1, \ldots, v_n) \mid (x : v_1) :: \cdots :: (x : v_n) :: E' \vdash e \Rightarrow v}{E \vdash (e_1\ e_2) \Rightarrow v}\ (\textsc{App1})$$

Now recall that the syntax for setting a variable is $\texttt{let } x = e_1 \texttt{ in } e_2$ So the rule for `let` is that we just add $(x : e_1)$ to the environment:

$$\frac{E \vdash e_1 \Rightarrow v_1 \mid (x : v_1) :: E \vdash e_2 \Rightarrow v}{E \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \Rightarrow v}\ (\textsc{Let})$$

We also have `let rec` whose syntax is $\texttt{let rec } f\ x = e_1 \texttt{ in } e_2$. Now the idea is that $\texttt{let rec} f\ x = e$ will be given the closure $\langle E, (\texttt{fun } x \to e)\rangle$ where E contains an infinite pair $f : \langle E, (\texttt{fun } x \to e)\rangle$. Such an object can be represented in memory as an object with a pointer which points to itself. So given an environment E, a function name f, and an expression $(\texttt{fun } x \to e)$ (where importantly e may contain occurrences of f), we assumed we can construct an environment E' such that $E' = (f : \langle E', (\texttt{fun } x \to e)\rangle) :: E$, and then the rule is:

$$\frac{E' = (f : \langle E', (\texttt{fun } x \to e)\rangle) :: E \vdash e_2 \Rightarrow v}{E \vdash \texttt{let rec } f\ x = e_1 \texttt{ in } e_2 \Rightarrow v}\ (\textsc{Letrec})$$

Before giving an example, let us define some boolean and arithmetic rules:

$$\frac{E \vdash e_1 \Rightarrow v_1 \mid E \vdash e_2 \Rightarrow v_2 \mid v_1 \leq v_2}{E \vdash e_1 \leq e_2 \Rightarrow \texttt{true}} \text{ (LEQ1)} \qquad \frac{E \vdash e_1 \Rightarrow v_1 \mid E \vdash e_2 \Rightarrow v_2 \mid v_1 > v_2}{E \vdash e_1 \leq e_2 \Rightarrow \texttt{false}} \text{ (LEQ2)}$$

$$\frac{E \vdash e_1 \Rightarrow v_1 \mid E \vdash e_2 \Rightarrow v_2 \mid v_1 = v_2}{E \vdash e_1 = e_2 \Rightarrow \texttt{true}} \text{ (EQ1)} \qquad \frac{E \vdash e_1 \Rightarrow v_1 \mid E \vdash e_2 \Rightarrow v_2 \mid v_1 \neq v_2}{E \vdash e_1 = e_2 \Rightarrow \texttt{false}} \text{ (EQ2)}$$

$$\frac{E \vdash e_1 \Rightarrow v_1 \mid E \vdash e_2 \Rightarrow v_2}{E \vdash e_1 + e_2 \Rightarrow v_1 + v_2} \text{ (ADD)} \qquad \frac{E \vdash e_1 \Rightarrow v_1 \mid E \vdash e_2 \Rightarrow v_2}{E \vdash e_1 - e_2 \Rightarrow v_1 - v_2} \text{ (SUB)}$$

$$\frac{E \vdash e_1 \Rightarrow v_1 \mid E \vdash e_2 \Rightarrow v_2}{E \vdash e_1 \cdot e_2 \Rightarrow v_1 \cdot v_2} \text{ (MUL)} \qquad \frac{E \vdash e_1 \Rightarrow v_1 \mid E \vdash e_2 \Rightarrow v_2}{E \vdash e_1 / e_2 \Rightarrow v_1 / v_2} \text{ (DIV)}$$

So for example, suppose our initial environment is the empty list, and we'd like to compute the value of

$$\texttt{let rec } f\ x = (\texttt{if } x = 1 \texttt{ then } 1 \texttt{ else } x \cdot f(x-1)) \texttt{ in } (f\ 2)$$

Which should give $2! = 2$. Let us write `fact` in place of $\texttt{fun } x \to (\texttt{if } x = 0 \texttt{ then } 1 \texttt{ else } x \cdot f(x-1))$, so

$$
\dfrac{
  \dfrac{
    E \vdash f \Rightarrow \langle E, \texttt{fact} \rangle \;\; \textsc{Var}
    \qquad
    E \vdash 2 \Rightarrow 2 \;\; \textsc{Val}
    \qquad
    \dfrac{
      \dfrac{
        E' \vdash x \Rightarrow 2 \;\; \textsc{Var}
        \qquad
        E' \vdash 1 \Rightarrow 1 \;\; \textsc{Val}
      }{
        E' \vdash x = 1 \Rightarrow \texttt{false}
      }\;\textsc{Eq2}
      \qquad
      \dfrac{
        E' \vdash x \Rightarrow 2 \;\; \textsc{Var}
        \qquad
        \dfrac{
          E' \vdash f \Rightarrow \texttt{fact} \;\; \textsc{Var}
          \qquad
          \dfrac{
            E' \vdash x \Rightarrow 2 \;\; \textsc{Var}
            \qquad
            E' \vdash 1 \Rightarrow 1 \;\; \textsc{Val}
          }{
            E' \vdash x - 1 \Rightarrow 1
          }\;\textsc{Sub}
          \qquad
          \dfrac{
            \dfrac{
              E'' \vdash x \Rightarrow 1 \;\; \textsc{Var}
              \qquad
              E'' \vdash 1 \Rightarrow 1 \;\; \textsc{Val}
            }{
              E'' \vdash x = 1 \Rightarrow \texttt{true}
            }\;\textsc{Eq}
            \qquad
            E'' \vdash 1 \Rightarrow 1 \;\; \textsc{Val}
          }{
            E'' = (x:1) :: E' \vdash (\texttt{if}\ x = 1\ \texttt{then}\ 1\ \texttt{else}\ x \cdot f(x-1)) \Rightarrow 1
          }\;\textsc{Cond1}
        }{
          E' \vdash f(x-1) \Rightarrow 1
        }\;\textsc{App1}
      }{
        E' \vdash x \cdot f(x-1) \Rightarrow 2
      }\;\textsc{Mul}
    }{
      E' = (x:2) :: E \vdash (\texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x \cdot f(x-1))
    }\;\textsc{Cond2}
  }{
    E = (f : \langle E, \texttt{fact} \rangle) \vdash (f\ 2) \Rightarrow 2
  }\;\textsc{App1}
}{
  [] \vdash \texttt{let rec}\ f\ x = (\texttt{if}\ x = 1\ \texttt{then}\ 1\ \texttt{else}\ x \cdot f(x-1))\ \texttt{in}\ (f\ 2) \Rightarrow 2
}\;\textsc{Letrec}
$$