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:

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

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 $\mathcal{A}$: **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:

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

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

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

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

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

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

$[\text{while}^{\text{ff}}_{\text{ns}}]$ $\quad \langle \texttt{while } b \texttt{ do } S, s \rangle \to s$ 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 $\langle (z := x; x := y); y := z, s_0\rangle \to s_3$.

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

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 $S \in \mathbf{Stm}$. We will prove this: suppose $\langle S, s\rangle \to s'$ then it has a deductive tree T, and so

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

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

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

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) [$\text{while}^{\text{ff}}_{\text{ns}}$]: 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, $var(x) = \{x\}$, $free(x) = \{x\}$, $bnd(x) = \varnothing$,

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

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

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 $free(t)$ 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 $var(t) = \{x, y, z\}$, $free(t) = \{x, z\}$, $bnd(t) = \{x, y\}$. 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 $var(t) = free(t) \cup bnd(t)$, 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 $t = x$, $var(t) = \{x\}$, $free(t) = \{x\}$, and $bnd(t) = \varnothing$, so the union holds.

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

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

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 $\lambda\texttt{y.x}$ should become $\lambda\texttt{y.z}$. But then what should $\lambda\texttt{x.x}$ become? Surely not $\lambda\texttt{x.z}$, 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 $\lambda\texttt{x.y}$, then substituting at y gives $\lambda\texttt{x.x}$, 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

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 $\lambda w.y\, z\, w$.

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.

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:

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

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

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

(**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).

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

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 $\mathtt{test}$ 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

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

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 $\mathtt{and}$ 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

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

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

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

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

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

Let us define

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

We see then that

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

so `scc` $\mathtt{c}_n$ and $\mathtt{c}_{n+1}$ are the same.

Similarly we can define

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

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

$$\mathtt{times\ =}\ \lambda\mathtt{n}.\lambda\mathtt{m}.\lambda\mathtt{s}.\lambda\mathtt{z}.\mathtt{m\ (plus\ n)}\ \mathtt{c}_0$$

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

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 $\mathtt{c}_n$ `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

$$\mathtt{iszro=}\ \lambda\mathtt{n}.\mathtt{n}\ (\lambda\mathtt{x}.\mathsf{fls})\ \mathtt{tru}$$

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

$$\begin{aligned} \mathtt{zz} &= \mathtt{pair}\ \mathtt{c}_0\ \mathtt{c}_0 \\ \mathtt{ss} &= \lambda\mathtt{p}.\mathtt{pair(snd\ p)(plus}\ \mathtt{1}\ \mathtt{(snd\ p))} \\ \mathtt{prd} &= \lambda\mathtt{m}.\mathtt{fst(m\ ss\ zz)} \end{aligned}$$

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 $(3, 2)$ and so on. In general $\mathtt{ss}^n\mathtt{z} = (n, n - 1)$ for $n \geq 1$ and so the predecessor is just the second value.

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

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

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

$$\begin{aligned} \mathtt{leq} &= \lambda\mathtt{m}.\lambda\mathtt{n}.\mathtt{iszro(sub\ m\ n)} \\ \mathtt{equal} &= \lambda\mathtt{m}.\lambda\mathtt{n}.\mathtt{and(leq\ n\ m)\ (leq\ m\ n)} \end{aligned}$$

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

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

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

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

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

$$
\begin{array}{l}
\texttt{test (equal n } \mathtt{c}_0\texttt{)} \\
\quad \mathtt{c}_1 \\
\quad \texttt{times n (test (equal (prd n) } \mathtt{c}_0\texttt{)} \\
\qquad\quad \mathtt{c}_1 \\
\qquad\quad \texttt{times (prd n) (test (equal (prd (prd n)) } \mathtt{c}_0\texttt{)} \\
\qquad\qquad\qquad\qquad\quad \mathtt{c}_1 \\
\qquad\qquad\qquad\qquad\quad \texttt{times (prd (prd n)) (...)))}
\end{array}
$$

Then we define

$$
\begin{aligned}
\texttt{g} &= \lambda\texttt{fct.}\lambda\texttt{n. test (equal n } \mathtt{c}_0\texttt{) } \mathtt{c}_1 \texttt{ (times n (fct (prd n)))} \\
\texttt{factorial} &= \texttt{fix g}
\end{aligned}
$$

Let us give an example run of `factorial` $\mathtt{c}_3$:

$$
\begin{array}{lll}
 & \texttt{factorial } \mathtt{c}_3 & \\
= & \texttt{fix g } \mathtt{c}_3 & \\
\to & \texttt{h h } \mathtt{c}_3 & \text{where } \texttt{h=}\lambda\texttt{x.g(}\lambda\texttt{y.x x y)} \\
\to & \texttt{g fct } \mathtt{c}_3 & \text{where } \texttt{fct=}\lambda\texttt{y. h h y} \\
\to & \big(\lambda\texttt{n. test(equal n } \mathtt{c}_0\texttt{) } \mathtt{c}_1 \texttt{ (times n (fct (prd n)))}\big)\mathtt{c}_3 & \\
\to & \texttt{test(equal } \mathtt{c}_3 \ \mathtt{c}_0\texttt{) } \mathtt{c}_1 \texttt{ (times } \mathtt{c}_3 \texttt{ (fct (prd } \mathtt{c}_3\texttt{)))} & \\
\to & \texttt{times } \mathtt{c}_3 \texttt{ (fct (prd } \mathtt{c}_3\texttt{))} & \\
\to & \texttt{times } \mathtt{c}_3 \texttt{ (fct } \mathtt{c}_2\texttt{)} & \\
\to & \texttt{times } \mathtt{c}_3 \texttt{ (h h } \mathtt{c}_2\texttt{)} & \\
\to & \texttt{times } \mathtt{c}_3 \texttt{ (g fct } \mathtt{c}_2\texttt{)} & \text{similar to how } \texttt{h h } \mathtt{c}_3 \text{ can be reduced to } \texttt{g fct } \mathtt{c}_3 \\
\to & \texttt{times } \mathtt{c}_3 \texttt{ (times } \mathtt{c}_2 \texttt{ (g fct } \mathtt{c}_1\texttt{))} & \text{by the same process that we did for } \mathtt{c}_3 \\
\to & \texttt{times } \mathtt{c}_3 \texttt{ (times } \mathtt{c}_2 \texttt{ (times } \mathtt{c}_1 \texttt{ (g fct } \mathtt{c}_0\texttt{)))} & \\
\to & \texttt{times } \mathtt{c}_3 \texttt{ (times } \mathtt{c}_2 \texttt{ (times } \mathtt{c}_1 \texttt{ (test (equal } \mathtt{c}_0 \ \mathtt{c}_0\texttt{) } \mathtt{c}_1 \texttt{ ...)))} & \\
\to & \texttt{times } \mathtt{c}_3 \texttt{ (times } \mathtt{c}_2 \texttt{ (times } \mathtt{c}_1 \ \mathtt{c}_1\texttt{))} & \\
\to & \mathtt{c}_6 &
\end{array}
$$

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*:

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

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:

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

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.

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 $\lambda x{:}\,T.t$ 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 $\mathsf{id} = \lambda x{:}\,\mathsf{Bool}.x$, then id is a term.

3.3 Definition

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

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

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

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

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

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

And

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

We'd like to assign to terms a type. Suppose Γ is a set containing elements of the form $t'{:}\,T'$, then we write $\Gamma \vdash t{:}\,T$ to mean that if we assume Γ then t has the type T. If Γ is a such a set, we write $\Gamma, t'{:}\,T'$ to mean $\Gamma \cup \{t'{:}\,T'\}$, and instead of $\varnothing \vdash t{:}\,T$ we write $\vdash t{:}\,T$. We utilize Gentzen-style rules to form a deductive system for deducing the type of an abstraction. The first rule is for abstractions,

$$\frac{\Gamma, \texttt{x:T} \vdash \texttt{t:T'}}{\Gamma \vdash \lambda\texttt{x:T.t : T}\to\texttt{T'}} \qquad (\text{T-Abs})$$

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

$\texttt{t:float}$ as well, so $\lambda\texttt{x:float.x+x}$ has type $\texttt{float}\to\texttt{float}$. But if $\texttt{x}$ is of type $\texttt{int}$, then $\texttt{t}$ is of the same type and $\lambda\texttt{x:int.x+x}$ has type $\texttt{int}\to\texttt{int}$. Importantly, these examples are given to give some intuition for the rule, they are not valid λ-terms!

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

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

We also need a rule for applications:

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

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

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

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 $\vdash\lambda\texttt{x}:\mathsf{Bool}.\ \texttt{if x then true else x}:\mathsf{Bool}\to\mathsf{Bool}$. We form a deductive tree:

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

3.4 Definition

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

3.5 Definition

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

3.7 Lemma (Substitution Lemma)

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

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

(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 $\mathtt{y}$ 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 $\Gamma, \mathtt{y}\colon \mathsf{T}_2 \vdash \mathtt{t}[\mathtt{x} \mapsto \mathtt{t}']\colon \mathsf{T}_1$. By T-Abs, we get $\Gamma \vdash \lambda \mathtt{y}.\mathtt{t}_1[\mathtt{x} \mapsto \mathtt{t}']\colon \mathsf{T}$, 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:

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

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'.

3.10 Theorem

Every closed well-typed term t can be normalized.

Proof: in the book. ∎