

Programming Languages

Lectures by Yoni Zohar

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

Contents

1	Semantics of Expressions	1
2	Untyped Lambda Calculus	5
3	Simply Typed Lambda Calculus	11
4	λ -OCaml	14

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:

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

Explicitly, arithmetic expressions are defined recursively as so:

- (1) numerals and variables are arithmetic expressions,
- (2) if $a_1, a_2 \in \mathbf{Aexp}$ then $a_1 + a_2, a_1 \star a_2, a_1 - a_2 \in \mathbf{Aexp}$.

Similarly boolean expressions are defined recursively

- (1) true and false are boolean expressions,
- (2) if $a_1, a_2 \in \mathbf{Aexp}$ then $a_1 = a_2, a_1 \leq a_2 \in \mathbf{Bexp}$,
- (3) if $b_1, b_2 \in \mathbf{Bexp}$ then $\neg b_1, b_1 \wedge b_2 \in \mathbf{Bexp}$.

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

So for example, if x, y are variables then

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

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 $\mathbf{Var} \rightarrow \mathbb{Z}$, define **State** to be the set of all states (all functions $\mathbf{Var} \rightarrow \mathbb{Z}$).

1.2 Definition

We define the function $\mathcal{A}: \mathbf{Aexp} \rightarrow (\mathbf{State} \rightarrow \mathbb{Z})$, 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 $x \rightarrow 1$ and $y \rightarrow 3$ then

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

1.3 Definition

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

- (1) $\mathcal{B}[\mathbf{true}]s = tt$, $\mathcal{B}[\mathbf{false}]s = ff$,
- (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 $\langle \cdot, \cdot \rangle \rightarrow \cdot \subseteq (\mathbf{Stm} \times \mathbf{State} \times \mathbf{State})$, here we read $\langle S, s \rangle \rightarrow s'$ as “ s' is derivable from S, s ”. We write

$$\frac{\langle S_1, s_1 \rangle \rightarrow s'_1, \dots, \langle S_n, s_n \rangle \rightarrow s'_n}{\langle S, s \rangle \rightarrow s'} \quad \text{if } \dots$$

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

We now list the transitions:

$$\begin{array}{ll} [\mathbf{ass}_{\text{ns}}] & \langle x := a, s \rangle \rightarrow s[x \mapsto \mathcal{A}[a]s] \\ [\mathbf{skip}_{\text{ns}}] & \langle \mathbf{skip}, s \rangle \rightarrow s \\ [\mathbf{comp}_{\text{ns}}] & \frac{\langle S_1, s \rangle \rightarrow s' \quad \langle S_2, s' \rangle \rightarrow s''}{\langle S_1; S_2, s \rangle \rightarrow s''} \\ [\mathbf{if}_{\text{ns}}^{\text{tt}}] & \frac{\langle S_1, s \rangle \rightarrow s'}{\langle \mathbf{if } b \text{ then } S_1 \text{ else } S_2 \rangle \rightarrow s'} \quad \text{if } \mathcal{B}[b]s = tt \\ [\mathbf{if}_{\text{ns}}^{\text{ff}}] & \frac{\langle S_2, s \rangle \rightarrow s'}{\langle \mathbf{if } b \text{ then } S_1 \text{ else } S_2 \rangle \rightarrow s'} \quad \text{if } \mathcal{B}[b]s = ff \\ [\mathbf{while}_{\text{ns}}^{\text{tt}}] & \frac{\langle S, s \rangle \rightarrow s' \quad \langle \mathbf{while } b \text{ do } S, s' \rangle \rightarrow s''}{\langle \mathbf{while } b \text{ do } S \rangle \rightarrow s''} \quad \text{if } \mathcal{B}[b]s = tt \\ [\mathbf{while}_{\text{ns}}^{\text{ff}}] & \langle \mathbf{while } b \text{ do } S, s \rangle \rightarrow s \text{ if } \mathcal{B}[b]s = ff \end{array}$$

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

1.5 Definition

The **deductive tree** of $\langle S, s \rangle \rightarrow s'$ is a tree whose root is $\langle S, s \rangle \rightarrow s'$ and the leaves are axioms. Every inner node is a transition which is a consequence of its children. We define $\langle S, s \rangle \rightarrow s'$ 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 $\langle (z := x; x := y); y := z, s_0 \rangle \rightarrow s_3$.

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

1.6 Definition

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

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

$$\frac{\frac{T}{\langle s, s \rangle \rightarrow s'} \quad \langle \text{skip}, s' \rangle \rightarrow s' \quad \text{skip}}{\langle s; \text{skip}, s \rangle \rightarrow s'}$$

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

$$\frac{\frac{T}{\langle s, s \rangle \rightarrow s'} \quad \langle \text{skip}, s' \rangle \rightarrow s' \quad \text{skip}}{\langle s; \text{skip}, s \rangle \rightarrow s'}$$

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

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 $\langle S, s \rangle \rightarrow s'$ and $\langle S, s \rangle \rightarrow s''$ 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 skip and so $s' = s$.
- (3) $[\text{comp}_{\text{ns}}]$: assume $\langle S_1; S_2, s \rangle \rightarrow s'$ holds because $\langle S_1, s \rangle \rightarrow s_0$ and $\langle S_2, s_0 \rangle \rightarrow s'$ for some s_0 . The only rule which can be applied to get $\langle S_1; S_2, s \rangle \rightarrow s''$ is $[\text{comp}_{\text{ns}}]$, so there is a state s_1 such that $\langle S_1, s \rangle \rightarrow s_1$ and $\langle S_2, s_1 \rangle \rightarrow s''$. But by induction, $s_1 = s_0$ and then applying induction again, $s' = s''$.
- (4) $[\text{if}_{\text{ns}}^{\text{tt}}]$: assume that $\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'$ holds because $\mathcal{B}[b]s = tt$ and $\langle S_1, s \rangle \rightarrow s'$. Since $\mathcal{B}[b]s = tt$, the only rule which can be applied to get $\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s''$ is $[\text{if}_{\text{ns}}^{\text{tt}}]$, so $\langle S_1, s \rangle \rightarrow s''$, and by induction $s' = s''$.
- (5) $[\text{if}_{\text{ns}}^{\text{ff}}]$: similar.
- (6) $[\text{while}_{\text{ns}}^{\text{tt}}]$: assume that $\langle \text{while } b \text{ do } S, s \rangle \rightarrow s'$ because $\mathcal{B}[b]s = tt$, $\langle S, s \rangle \rightarrow s_0$, and $\langle \text{while } b \text{ do } S, s_0 \rangle \rightarrow s'$ for some s_0 . The only rule which could be applied to get $\langle \text{while } b \text{ do } S, s \rangle \rightarrow s''$ is $[\text{while}_{\text{ns}}^{\text{tt}}]$ in lieu of

4 *Semantics of Expressions*

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

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

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

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 `fun x → 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 , $\text{var}(x) = \{x\}$, $\text{free}(x) = \{x\}$, $\text{bnd}(x) = \emptyset$,
- (2) for an term of the form $\lambda x.t$, $\text{var}(\lambda x.t) = \text{var}(t) \cup \{x\}$, $\text{free}(\lambda x.t) = \text{free}(t) \setminus \{x\}$, and $\text{bnd}(\lambda x.t) = \text{bnd}(t) \cup \{x\}$,
- (3) for an term of the form $t_1 t_2$, $\text{var}(t_1 t_2) = \text{var}(t_1) \cup \text{var}(t_2)$, $\text{free}(t_1 t_2) = \text{free}(t_1) \cup \text{free}(t_2)$ and $\text{bnd}(t_1 t_2) = \text{bnd}(t_1) \cup \text{bnd}(t_2)$.

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 $\text{free}(t)$ is the set of all variables which occur free in t , $\text{bnd}t$ is the set of all variables which occur bound in t .

So for example, let $t = (\lambda x.\lambda y.x) x z$, then $\text{var}(t) = \{x, y, z\}$, $\text{free}(t) = \{x, z\}$, $\text{bnd}(t) = \{x, y\}$. 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 $\text{var}(t) = \text{free}(t) \cup \text{bnd}(t)$, 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 $t = x$, $\text{var}(t) = \{x\}$, $\text{free}(t) = \{x\}$, and $\text{bnd}(t) = \emptyset$, so the union holds.
- (2) for $t = \lambda x.t'$, $\text{var}(t) = \text{var}(t') \cup \{x\}$ which by induction is equal to $\text{free}(t') \cup \text{bnd}(t') \cup \{x\}$. Now $\text{free}(t) = \text{free}(t') \setminus \{x\}$, $\text{bnd}(t) = \text{bnd}(t') \cup \{x\}$ and so we see that $\text{free}(t) \cup \text{bnd}(t) = \text{var}(t)$ as required.
- (3) for $t = t_1 t_2$, $\text{var}(t) = \text{var}(t_1) \cup \text{var}(t_2)$ which by induction is $\text{free}(t_1) \cup \text{free}(t_2) \cup \text{bnd}(t_1) \cup \text{bnd}(t_2) = \text{free}(t) \cup \text{bnd}(t)$.

2.3 Definition

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

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 $\lambda y.x$ should become $\lambda y.z$. But then what should $\lambda x.x$ become? Surely not $\lambda x.z$, 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 $\lambda x.y$, then substituting at y gives $\lambda x.x$, 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 $y \notin \text{free}(t')$,
- (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 $\lambda w.y z w$.

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, $(\lambda x.x)y \rightarrow y$, and

$$(\lambda x.(\lambda x.x)x)(ur) \rightarrow (\lambda x.x)(ur) = ur$$

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

$$(\lambda x.x) ((\lambda x.x) (\lambda z.(\lambda x.x) z))$$

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

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

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

$$\begin{aligned} & \text{id}(\text{id}(\lambda z.\text{id}z)) \\ & \rightarrow \text{id}(\text{id}(\lambda z.z)) \\ & \rightarrow \text{id}(\lambda z.z) \\ & \rightarrow \lambda z.z \end{aligned}$$

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:

$$\begin{aligned} & \text{id}(\text{id}(\lambda z.\text{id}z)) \\ & \rightarrow \text{id}(\lambda z.\text{id}z) \\ & \rightarrow \lambda z.\text{id}z \\ & \rightarrow \lambda z.z \end{aligned}$$

The rules for normal order reduction are as follows:

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

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

$$\begin{aligned} & \text{id}(\text{id}(\lambda z. \text{id}z)) \\ \rightarrow & \text{id}(\lambda z. \text{id}z) \\ \rightarrow & \lambda z. \text{id}z \end{aligned}$$

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

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

- (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).

$$\begin{aligned} & \text{id}(\text{id}(\lambda z. \text{id}z)) \\ \rightarrow & \text{id}(\lambda z. \text{id}z) \\ \rightarrow & \lambda z. \text{id}z \end{aligned}$$

The rules for call-by-value reduction are

$$\begin{aligned} & \frac{}{(\lambda x. t)v \rightarrow t[x \mapsto v]} \text{ if } v \text{ is a value} \quad , \quad \frac{s \rightarrow s'}{(\lambda x. t)s \rightarrow (\lambda x. t)s'} \text{ if } s \text{ is not a value} \quad , \\ & \frac{t \rightarrow t'}{ts \rightarrow t's} \text{ if } t \text{ is not a value} \end{aligned}$$

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):

$$\text{tru} = \lambda t. \lambda f. t, \quad \text{fls} = \lambda t. \lambda f. f$$

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

$$\text{test} = \lambda b. \lambda m. \lambda n. b \ m \ n$$

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

$$\text{test tru } v \ w = (\lambda b. \lambda m. \lambda n. b \ m \ n) \text{tru } v \ w \rightarrow (\lambda m. \lambda n. \text{tru } m \ n) v \ w \rightarrow (\lambda n. \text{tru } v \ n) w \rightarrow \text{tru } v \ w \rightarrow v$$

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

$$\text{and} = \lambda b. \lambda c. b \ c \ \text{fls}$$

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

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

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

Notice then that

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

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

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

$$\begin{aligned}
c_0 &= \lambda s. \lambda z. z \\
c_1 &= \lambda s. \lambda z. s \ z \\
c_2 &= \lambda s. \lambda z. s (s \ z) \\
c_3 &= \lambda s. \lambda z. s (s (s \ z)) \\
&\text{etc.}
\end{aligned}$$

In general if we write $s^n z$ for $s(s(\dots s \ z \dots))$ (n times), then $c_n = \lambda s. \lambda z. s^n \ z$. So each number n is represented by the combinator c_n which accepts s, z and applies s n times to z . Notice that $c_0 = \text{fls}$, which is reminiscent of the fact that false and zero mean the same thing in many compiled languages.

Let us define

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

We see then that

$$\text{scc } c_n \ z \ s = \lambda s. \lambda z. s (c_n \ s \ z) \ s \ z = s (s^n \ z) = s^{n+1} \ z = c_{n+1} \ z \ s$$

so $\text{scc } c_n$ and c_{n+1} are the same.

Similarly we can define

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

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

$$\text{times} = \lambda n. \lambda m. \lambda s. \lambda z. m \ (\text{plus } n) \ c_0$$

so that $\text{times } n \ m \ s \ z$ will apply $\text{plus } n \ m$ times to c_0 , resulting in $n + n + \dots + n + 0 = n \cdot m$. In a similar vein, we can define $\text{pow} = \lambda n. \lambda m. \lambda s. \lambda z. m \ (\text{times } n) \ c_1$, so that $\text{pow } c_n \ c_m$ is equal to 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 $c_n \ \text{ss} \ \text{zz}$, 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

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

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

$$\begin{aligned}
\text{zz} &= \text{pair } c_0 \ c_0 \\
\text{ss} &= \lambda p. \text{pair}(\text{snd } p) (\text{plus } 1 \ (\text{snd } p)) \\
\text{prd} &= \lambda m. \text{fst}(m \ \text{ss} \ \text{zz})
\end{aligned}$$

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 $\text{ss}^n z = (n, n-1)$ 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:

$$\text{sub} = \lambda m. \lambda n. m \ \text{prd} \ n$$

Notice though that sub cannot give negative numbers, after all we didn't define negative numbers, so if $n \leq m$ then $c_n - c_m$ is just c_0 . Thus we can define

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

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

$$\omega = (\lambda_{x,x} \ x) (\lambda_{x,x} \ x)$$

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

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

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

```
test (equal n c0)
  c1
  times n (test (equal (prd n) c0)
    c1
    times (prd n) (test (equal (prd (prd n)) c0)
      c1
      times (prd (prd n)) (...)))
```

Then we define

```

      g = λfct.λn. test (equal n c0) c1 (times n (fct (prd n)))
factorial = fix g

```

Let us give an example run of `factorial c3`:

```

factorial c3
= fix g c3
→ h h c3                                where h=λx.g(λy.x x y)
→ g fct c3                             where fct=λy. h h y
→ (λn. test(equal n c0) c1 (times n (fct (prd n))))c3
→ test(equal c3 c0) c1 (times c3 (fct (prd c3)))
→ times c3 (fct (prd c3))
→ times c3 (fct c2)
→ times c3 (h h c2)
→ times c3 (g fct c2)                   similar to how h h c3 can be reduced to g fct c3
→ times c3 (times c2 (g fct c1))       by the same process that we did for c3
→ times c3 (times c2 (times c1 (g fct c0)))
→ times c3 (times c2 (times c1 (test (equal c0 c0) c1 ...)))
→ times c3 (times c2 (times c1 c1))
→ c6

```

Let us prove that this works. Suppose we have a recurrence $r = \lambda x. \langle \text{code with } r \rangle$, let us use the notation $\langle r \ c \rangle$ to mean that within the recurrence, r is called on the value c . Let us define $g = \lambda r. \lambda x. \langle \text{code with } r \rangle$, 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 $r = \text{fix } g$ is a term which is equivalent to r (does the same thing). Let us reduce it a bit on some term c

$$\begin{array}{ll} & \text{r c} \\ = & \text{fix g c} \\ \rightarrow & \text{h h c} \quad \text{where h} = \lambda x. g(\lambda y. x \ x \ y) \\ \rightarrow & \text{g r' c} \quad \text{where r'} = \lambda y. \text{h h y} \end{array}$$

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

We can also define the *Y-combinator*:

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

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

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

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 $T_1 \rightarrow T_2$.

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

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) `true`, `false` are terms,
- (5) if t_1, t_2, t_3 are terms, then so is `if t_1 then t_2 else t_3` .

Let us define `id` = $\lambda x:\text{Bool}.x$, 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) `if true then t_1 else t_2` $\xrightarrow{\text{hello there}}$ t_1 ,
- (3) `if false then t_1 else t_2` $\xrightarrow{\beta}$ t_2 .

So for example, let `f` = $\lambda x:\text{Bool} \rightarrow \text{Bool}.\lambda y:\text{Bool}.x y$, then

<code>f id true</code>	
$= (\lambda x:\text{Bool} \rightarrow \text{Bool}.\lambda y:\text{Bool}.x y)\text{id true}$	definition
$\rightarrow (\lambda y:\text{Bool}.\text{id } y) \text{ true}$	β -reduction on the underlined redex
$\rightarrow \text{id true}$	β -reduction on the underlined redex
$\rightarrow \text{true}$	β -reduction on the underlined redex

And

<code>f true id</code>	
$= (\lambda x:\text{Bool} \rightarrow \text{Bool}.\lambda y:\text{Bool}.x y)\text{true id}$	definition
$\rightarrow (\lambda y:\text{Bool}.\text{true } y) \text{ id}$	β -reduction on the underlined redex
$\rightarrow \text{true id}$	β -reduction on the underlined redex

We'd like to assign to terms a type. Suppose Γ is a set containing elements of the form $x:T'$ where x ranges over all the variables (and each variable occurs only once), 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 $\emptyset \vdash t:T$ 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,

$$\frac{\Gamma, x:T \vdash t:T'}{\Gamma \vdash \lambda x:T.t : T \rightarrow T'} \quad (\text{T-ABS})$$

This just means that if we assume x has type T then t has type T' , then we can conclude that $\lambda x:T.t$ has type $T \rightarrow T'$. 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 $\lambda x:\text{float}.x+x$ has type `float` \rightarrow `float`. But if x is of type `int`, then `t` is of the same type

and $\lambda x:\text{int}.x+x$ has type $\text{int} \rightarrow \text{int}$. Importantly, these examples are given to give some intuition for the rule, they are not valid λ -terms!

Obviously if $x:T$ is already in Γ then Γ should deduce $x:T$:

$$\frac{x:T \in \Gamma}{\Gamma \vdash x:T} \quad (\text{T-VAR})$$

We also need a rule for applications:

$$\frac{\Gamma \vdash t:T' \rightarrow T \mid \Gamma \vdash t':T'}{\Gamma \vdash t \ t':T} \quad (\text{T-APP})$$

Which means that if t is a function $T' \rightarrow T$ and t' has type T' , then the application $t \ t'$ has type T . And for conditionals

$$\frac{\Gamma \vdash t_1:\text{Bool} \mid \Gamma \vdash t_2:T \mid \Gamma \vdash t_3:T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad (\text{T-IF})$$

And of course **true** and **false** are Boolean types:

$$\frac{}{\Gamma \vdash \text{true} : \text{Bool}}, \quad \frac{}{\Gamma \vdash \text{false} : \text{Bool}} \quad (\text{T-TRUE}), (\text{T-FALSE})$$

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

$$\frac{\frac{\frac{x:\text{Bool} \vdash x:\text{Bool} \quad \text{T-VAR} \quad x:\text{Bool} \vdash \text{true} : \text{Bool} \quad \text{T-TRUE} \quad x:\text{Bool} \vdash \text{false} : \text{Bool} \quad \text{T-FALSE}}{x:\text{Bool} \vdash \text{if } x \text{ then true else } x \quad \text{T-IF}}}{\vdash \lambda x:\text{Bool}. \text{if } x \text{ then true else } x : \text{Bool} \rightarrow \text{Bool}} \quad \text{T-ABS}$$

3.4 Definition

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

3.5 Definition

A term of the form **true**, **false**, or $\lambda x:T. t$ (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 $t \rightarrow t'$ through a step of β -reduction.

Proof: if t is a boolean or an abstraction, then it is a value. Otherwise $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$, then t is closed if and only if all t_i are and by the derivation rule $t_1:\text{Bool}$ which means that t_1 must be a Boolean, and so t can be reduced. Finally if $t = t_1 \ t_2$ then t is closed and well-typed, then $t_1:T' \rightarrow T$ and $t_2:T'$, which means that t_1 is either a value or can be reduced, likewise for t_2 . If either can be reduced, then so too can t (since if $t \rightarrow t_0$ then $t \ t' \rightarrow t_0 \ t'$ and similar for t'). If both are values, then t_1 is of the form $\lambda x.t_{11}$ and so it can be applied to a value and reduced. ■

3.7 Lemma (Substitution Lemma)

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

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

- (1) T-VAR: so $t = z$ and $z:T \in \Gamma, x:T'$. If $z = x$ then $t = z = x$, so $T = T'$ and $t[x \mapsto t'] = t'$. We must prove that $\Gamma \vdash t':T$, but we know that $t':T' = T$ so this holds. If $z \neq x$ then $t[x \mapsto t'] = z$ and this is satisfied trivially.
- (2) T-ABS: then $t = \lambda y:T_2. t_1$, $T = T_2 \rightarrow T_1$, and $\Gamma, x:T' \vdash \lambda y:T_2. t_1:T$ so that $\Gamma, x:T', y:T_2 \vdash t_1:T_1$. We may assume by convention that $x \neq y$ and that y is not free in t' . Since $\Gamma \vdash t':T'$, we get $\Gamma, y:T_2 \vdash t':T'$, and so by the induction hypothesis $\Gamma, y:T_2 \vdash t[x \mapsto t']:T_1$. By T-ABS, we get $\Gamma \vdash \lambda y.T_2. t_1[x \mapsto t']:T$, but $\lambda y.T_2. t_1[x \mapsto t'] = (\lambda y.T_2. t_1)[x \mapsto t'] = t[x \mapsto t']$ as required.

- (3) T-TRUE and T-FALSE are immediate since $t = \text{true}$ or false and $T = \text{Bool}$ and so $t[x \mapsto t'] = t$.
- (4) T-IF is straightforward. ■

3.8 Theorem (Preservation Theorem)

If $\Gamma \vdash t : T$ and $t \rightarrow t'$ by β -reduction, then $\Gamma \vdash t' : T$.

Proof: suppose $t = (\lambda x : T_1. t_1) t_2 : T_2$ then let us look at the derivation of t :

$$\frac{\frac{\Gamma, x : T_1 \vdash t_1 : T_2}{\Gamma \vdash \lambda x : T_1. t_1 : T_1 \rightarrow T_2} \text{T-ABS} \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (\lambda x : T_1. t_1) t_2 : T_2} \text{T-APP}$$

Our goal is to show $\Gamma \vdash t_1[x \mapsto t_2]$. But we have that $\Gamma, x : T_1 \vdash t_1 : T_2$ and $\Gamma \vdash t_2 : T_2$ 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. ■

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 x is a variable and t a term, then `fun $x \rightarrow t$` is a term,
- (3) if t_1 is a term, then $t_1 \ t_2$ is a term.

This is obviously equivalent to untyped λ -calculus where instead of $\lambda x.t$ we write `fun $x \rightarrow t$` . 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 $T \rightarrow S$.

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{O-VAR})$$

$$\frac{\Gamma \vdash t_{12} : T_1 \rightarrow T_2 \mid \Gamma \vdash t_1 : T_1}{\Gamma \vdash t_{12} t_1 : T_2} \quad (\text{O-APP})$$

$$\frac{\Gamma \vdash x : T \mid \Gamma \vdash t : S}{\Gamma \vdash (\text{fun } x \rightarrow t) : T \rightarrow S} \quad (\text{O-ABS})$$

Notice that this is similar to simply typed λ -calculus except for O-ABS, where instead of viewing what type has t has under the assumption that 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 $t_1 = \text{fun } x \rightarrow t_{11}$ and $t_2 = \text{fun } y \rightarrow t_{22}$, x and 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 $\alpha = \beta \rightarrow \gamma \in A_t$,
- (3) for every subterm $\text{fun } x \rightarrow t'$, if α is the variable of x , β of t' , and γ of $\text{fun } x \rightarrow t'$, then $\gamma = \alpha \rightarrow \beta \in A_t$.

For example, let t be $(\text{fun } x \rightarrow x)y$, then let us map the subterms to type variables as follows:

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

Then

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

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 variables such that $\sigma(T_1 \rightarrow T_2) = \sigma(T_1) \rightarrow \sigma(T_2)$. σ **preserves** an equality $T_1 = T_2$ if $\sigma(T_1) = \sigma(T_2)$, 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 $\sigma(\alpha_f) = \alpha \rightarrow \alpha$.

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

$$\Gamma_\sigma^\beta = \{x: \sigma(\beta(x)) \mid x \in \text{var } t\}$$

So using the above example where β is the map

$$\beta: \quad y \mapsto \alpha_y, \quad x \mapsto \alpha_x^1, \quad x \mapsto \alpha_x^2, \quad \text{fun } x \rightarrow x \mapsto \alpha_f, \quad t \mapsto \alpha_t$$

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 $\text{fun } x \rightarrow t''$, then let $\alpha_1, \alpha_2, \alpha_3$ be the types of x, t'', t' respectively. Then $\alpha_3 = \alpha_1 \rightarrow \alpha_2$ is an equation in A_t so $\sigma(\alpha_3) = \sigma(\alpha_1) \rightarrow \sigma(\alpha_2)$. In other words, $\sigma(\beta(t')) = \sigma(\beta(x)) \rightarrow \sigma(\beta(t''))$. Now, by induction we have that

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

So applying O-ABS yields

$$\Gamma_\sigma^\beta \vdash t' : \sigma(\beta(x)) \rightarrow \sigma(\beta(t'')) = \sigma(\beta(t'))$$

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 $t' = \text{fun } x \rightarrow t''$, 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 terms (since β is a correspondence, we can view the equation of type variables as equations of terms) finding a substitution which preserves it.