14: The Lambda Calculus: Introduction

As Curry points out in his classic work on Combinatory Logic,

> *Curiously a systematic notation for functions is lacking in ordinary mathematics. The usual notation '$f(x)$' does not distinguish between the function itself and the value of this function for an undetermined value of the argument.*

Let us consider the nature of functions, higher-order functions (functionals) and the use of naming in mathematics, through some examples.

Example 14.1 *Let $y = x^2$ be the squaring function on the reals. Here it is commonly understood that x is the "independent" variable and y is the "dependent" variable when we look on it as plotting the function $f(x) = x^2$ on the $x - y$ axis.*

Example 14.2 *Often a function may be named and written as $f(x) = x^n$ to indicate that x is the independent variable and n is understood (somehow!) to be some constant. Here f, x and n are all names with different connotations. Similarly in the quadratic polynomial $ax^2 + bx + c$ it is somehow understood that a, b and c denote constants and that x is the independent variable. Implicitly by using the names like a, b and c we are endeavouring to convey the impression that we consider the class $\{ax^2 + bx + c \mid a, b, c \in \mathbb{R}\}$ of all quadratic polynomials of the given form.*

Example 14.3 *As another example, consider the uni-variate polynomial $p(x) = x^2 + 2x + 3$. Is this polynomial the same as $p(y) = y^2 + 2y + 3$? Clearly they cannot be the same since the product $p(x).p(y)$ is a polynomial in two variables whereas $p(x).p(x)$ yields a uni-variate polynomial of degree 4. However, in the case of the function f in example 14.1 it does not matter whether we define the squaring function as $f(x) = x^2$ or as $f(y) = y^2$.*

Example 14.4 *The function $f(x) = x^2$ is a continuous and differentiable real-valued function (in the variable x) and its derivative is $f'(x) = 2x$. Whether we regard f' as the name of a new function or we regard the $'$ as an operation on f which yields its derivative seems to make no difference.*

Example 14.5 *Referring again to the functions $f(x)$ and $f'(x)$ in example 14.4, it is commonly understood that $f'(0)$ refers to the value of the derivative of f at 0 which is also the value of the function f' takes at 0. Now let us consider $f'(x+1)$. Going by the commonly understood notion, since $f'(x) = 2x$, we would have $f'(x+1) = 2(x+1)$. Then for $x = 0$ we have $f'(x+1) = f'(0+1) = f'(1) = 2 \times 1 = 2$. We could also think of it as the function $f'(g(0))$ where g is the function defined by $g(x) = x + 1$, then $f'(g(0)) = 2g(0) = 2$ which yields the same result.*

The examples above give us some idea of why there is no systematic notation for functions which distinguishes between a function definition and the application of the same function to some argument. It simply did not matter!

However, this ambiguity in mathematical notation could lead to differing interpretationas and results in the context of mathematical theories involving higher-order functions (or "functionals" as they are often referred to). One common higher order function is the derivative (the differentiation operation) and another is the indefinite integral. Most mathematical texts emphasize the higher-order nature of a function by enclosing their arguments in (square) brackets. Hence if O is a functional which transforms a function $f(x)$ into a function $g(x)$, this fact is usually written $O[f(x)] = g(x)$.

Example 14.6 *Consider the functional E (on continuous real-valued functions of one real variable x) defined as follows.*

$$E[f(x)] = \begin{cases} f'(0) & \text{if } x = 0 \\ \dfrac{f(x) - f(0)}{x} & \text{if } x \neq 0 \end{cases}$$

The main question we ask now is "What does $E[f(x+1)]$ mean?"

It turns out that there are at least two ways of interpreting $E[f(x+1)]$ and unlike the case of example 14.5, the two interpretations actually yield different results!.

1. *We may interpret $E[f(x+1)]$ to mean that we first apply the transformation E to the function $f(x)$ and then*

substitute $x+1$ for x in the resulting expression. We then have the following.

$$\begin{aligned} & E[f(x)] \\ = & \begin{cases} f'(0) & \text{if } x = 0 \\ \dfrac{f(x)-f(0)}{x} & \text{if } x \neq 0 \end{cases} \\ = & \begin{cases} 0 & \text{if } x = 0 \\ x & \text{if } x \neq 0 \end{cases} \\ = & \ x \end{aligned}$$

Since $E[f(x)] = x$, $E[f(x+1)] = x+1$.

2. *Since $f(x+1) = f(g(x))$ where $g(x) = x+1$, we may interpret $E[f(x+1)]$ as applying the operator E to the function $h(x) = f(g(x))$. Hence $E[f(x+1)] = E[h(x)]$ where $h(x) = f(g(x)) = (x+1)^2 = x^2+2x+1$. Noting that $h'(x) = 2x+2$, $h(0) = 1$ and $h'(0) = 2$, we get*

$$\begin{aligned} & E[h(x)] \\ = & \begin{cases} h'(0) & \text{if } x = 0 \\ \dfrac{h(x)-h(0)}{x} & \text{if } x \neq 0 \end{cases} \\ = & \begin{cases} 2 & \text{if } x = 0 \\ x+2 & \text{if } x \neq 0 \end{cases} \\ = & \ x+2 \end{aligned}$$

The last example should clearly convince the reader that there is a need to disambiguate between a function definition and its application.

In function definitions the independent variables are "bound" by a λ which acts as a pre-declaration of the name that is going to be used in the expression that defines a function.

The notation $f(x)$, which is interpreted to refer to "the value of function f at x", will be replaced by $(f\ x)$ to denote an application of a function f to the (known or unknown) value x.

In our notation of the untyped applied λ-calculus the functions and their applications in the examples in subsection 14.1 would be rewritten as follows.

Squaring . $\lambda\ x[x^2]$ is the squaring function.

Example 14.2 . $q \stackrel{df}{=} \lambda\ a\ b\ c\ x[ax^2 + bx + c]$ refers to any quadratic polynomial with coefficients unknown or symbolic. To obtain a particular member of this family such as $1x^2 + 2x + 3$, one would have to evaluate $(((q\ 1)\ 2)\ 3)$ which would yield $\lambda\ x[1x^2 + 2x + 3]$.

Example 14.3 . $p \stackrel{df}{=} \lambda\ x[x^2 + 2x + 3]$. Then $p(x)$ would be written as $(p\ x)$ i.e. as the function p applied to the argument x to yield the expression $x^2 + 2x + 3$. Likewise $p(y)$ would be $(p\ y)$ which would yield $y^2 + 2y + 3$. The products $(p\ x).(p\ x)$ and $(p\ x).(p\ y)$ are indeed different and distinct.

Example 14.5 Let us denote the operation of obtaining the derivative of a real-valued function f of one independent variable x by the simple symbol D (instead of the more confusing $\frac{d}{dx}$). Then for any function f, $(D\ f)$ would yield the derivative. In particular $(D\ \lambda\ x[x^2]) = \lambda\ x[2x]$ and the value of the derivative at 0 would be obtained by the application $(\lambda\ x[2x]\ 0)$ which would yield 0. Likewise the value of the derivative at $x + 1$ would be expressed as the application $(\lambda\ x[2x]\ (x + 1))$. Thus for any function f the value of its derivative at $x + 1$ is simply the application $((D\ f)\ (x + 1))$.

The function $g(x) = x + 1$ would be defined as $g \stackrel{df}{=} \lambda\ x[x + 1]$ and $(g\ x) = x + 1$. Thus the alternative

definition of the derivative of f at $x+1$ is simply the application $((D\ f)\ (g\ x))$.

Example 14.6 The two interpretations of the expression $E[f(x+1)]$ are respectively the following.

1. $((E\ f)\ (x+1))$ and
2. $((E\ h)\ x)$ where $h \stackrel{df}{=} \lambda\ x[(f\ (g\ x))]$

15: The Pure Untyped Lambda Calculus: Basics

Pure Untyped λ-Calculus: Syntax

The language Λ of pure untyped λ-terms is the smallest set built up from an infinite set V of *variables*

$$\begin{array}{rcll} L, M, N & ::= & x & \text{Variable} \\ & | & \lambda x[L] & \text{Abstraction} \\ & | & (L\ M) & \text{Application} \end{array}$$

where $x \in V$.

- A *Variable* denotes a possible binding in the external environment.
- An *Abstraction* denotes a function which takes a formal parameter.
- An *Application* denotes the application of a function to an actual parameter.

Free and Bound Variables

Definition 15.1 *For any term N the set of* **free variables** *and the set of all variables are defined by induction on the structure of terms.*

N	$FV(N)$	$Var(N)$
x	$\{x\}$	$\{x\}$
$\lambda x[L]$	$FV(L) - \{x\}$	$Var(L) \cup \{x\}$
$(L\ M)$	$FV(L) \cup FV(M)$	$Var(L) \cup Var(M)$

- *The set of* **bound variables** $BV(N) = Var(N) - FV(N)$.
- *The same variable name may be used with different bindings in a single term (e.g. $(\lambda x[x]\ \lambda x[(x\ y)])$)*
- *The brackets "[" and "]" delimit the scope of the bound variable x in the term $\lambda x[L]$.*
- **Combinators**: *$\Lambda_0 \subseteq \Lambda$ is the set of λ-terms with no free variables.*

Notational Conventions

To minimize use of brackets unambiguously

1. $\lambda x_1 x_2 \ldots x_m[L]$ denotes $\lambda x_1[\lambda x_2[\ldots \lambda x_m[L] \cdots]]$ i.e. L is the scope of each of the variables $x_1, x_2, \ldots x_m$.
2. $(L_1\ L_2\ \cdots L_m)$ denotes $(\cdots(L_1\ L_2)\ \cdots L_m)$ i.e. application is *left-associative*.

Substitution

Definition 15.2 *For any terms L, M and N and any variable x, the* **substitution** *of the term N for a variable x is defined as follows:*

$$
\begin{array}{lcll}
\{N/x\}x & \equiv & N & \\
\{N/x\}y & \equiv & y & \text{if } y \not\equiv x \\
\{N/x\}\lambda x[L] & \equiv & \lambda x[L] & \\
\{N/x\}\lambda y[L] & \equiv & \lambda y[\{N/x\}L] & \text{if } y \not\equiv x \text{ and } y \notin FV(N) \\
\{N/x\}\lambda y[L] & \equiv & \lambda z[\{N/x\}\{z/y\}L] & \text{if } y \not\equiv x \text{ and } y \in FV(N) \text{ and } z \text{ is ``fresh''} \\
\{N/x\}(L\ M) & \equiv & (\{N/x\}L\ \{N/x\}M) &
\end{array}
$$

- In the above definition it is necessary to ensure that the free variables of N continue to remain free after substitution.
- The phrase "z is fresh" may be taken to mean $z \notin FV(N) \cup Var(L)$.
- z could be "fresh" even if $z \in BV(N)$.

α-equivalence

Definition 15.3 (α-equivalence) $\lambda x[L] \equiv_\alpha \lambda y[\{y/x\}L]$ *provided* $y \notin Var(L)$.

- Here again if $y \in FV(L)$ it must not be captured by a change of bound variables.
- On the other hand if $y \in BV(L)$ then the substitution will replace all free occurrences of x in L and bind some of them to an inner binding of y.

In the sequel we will often omit the subscript α and consider two alpha equivalent terms to be syntactically equivalent.

Untyped λ-Calculus: Basic β-Reduction

Definition 15.4

- *Any (sub-)term of the form* $(\lambda x[L]\ M)$ *is called a* β*-redex*
- *Basic* β*-reduction is the relation*

$$\boxed{(\lambda x[L]\ M) \to_\beta \{M/x\}L'} \tag{6}$$

where $L' \equiv_\alpha L$.

Untyped λ-Calculus: 1-step β-Reduction

Definition 15.5 *A 1-step β-reduction $\rightarrow_\beta^1$ is the smallest (under the $\subseteq$ ordering) relation such that*

$$\beta_1 \quad \frac{L \rightarrow_\beta M}{L \rightarrow_\beta^1 M}$$

$$\beta_1\mathbf{Abs} \quad \frac{L \rightarrow_\beta^1 M}{\lambda x[L] \rightarrow_\beta^1 \lambda x[M]}$$

$$\beta_1\mathbf{AppL} \quad \frac{L \rightarrow_\beta^1 M}{(L\ N) \rightarrow_\beta^1 (M\ N)}$$

$$\beta_1\mathbf{AppR} \quad \frac{L \rightarrow_\beta^1 M}{(N\ L) \rightarrow_\beta^1 (N\ M)}$$

- $\rightarrow_\beta^1$ is the compatible closure of basic β-reduction to all contexts.
- We will often omit the superscript ^{1} as understood.

Untyped λ-Calculus: β-Reduction

Definition 15.6

- *For all integers $n \geq 0$, n-step β-reduction $\rightarrow_\beta^n$ is defined by induction on 1-step β-reduction*

$$\beta_{\mathbf{n}}\mathbf{Basis}\ \frac{}{L \rightarrow_\beta^0 L} \qquad \beta_{\mathbf{n}}\mathbf{Induction}\ \frac{L \rightarrow_\beta^m M \quad M \rightarrow_\beta^1 N}{L \rightarrow_\beta^{m+1} N}\ (m \geq 0)$$

- *β-reduction $\rightarrow_\beta^*$ is the reflexive-transitive closure of 1-step β-reduction. That is,*

$$\beta_*\ \frac{L \rightarrow_\beta^n M}{L \rightarrow_\beta^* M}\ (n \geq 0)$$

Untyped λ-Calculus: Normalization

Definition 15.7

- *A term is called a β-normal form (β-nf) if it has no β-redexes.*
- *A term is weakly normalising (β-WN) if it reduces to a β-normal form.*
- *A term L is strong normalising (β-SN) if it has no infinite reduction sequence $L \to^1_\beta L_1 \to^1_\beta L_2 \to^1_\beta \cdots$*

Untyped λ-Calculus: Examples

Example 15.8

1. $\mathsf{K} \stackrel{df}{=} \lambda x\ y[x]$, $\mathsf{I} \stackrel{df}{=} \lambda x[x]$, $\mathsf{S} \stackrel{df}{=} \lambda x\ y\ z[((x\ z)\ (y\ z))]$, $\omega \stackrel{df}{=} \lambda x[(x\ x)]$ *are all* β*-nfs.*

2. $\Omega \stackrel{df}{=} (\omega\ \omega)$ *has no* β*-nf. Hence it is neither weakly nor strongly normalising.*

3. $(\mathsf{K}\ (\omega\ \omega))$ *cannot reduce to any normal form because it has no finite reduction sequences. All its reductions are of the form*

$$(\mathsf{K}\ (\omega\ \omega)) \rightarrow^1_\beta (\mathsf{K}\ (\omega\ \omega)) \rightarrow^1_\beta (\mathsf{K}\ (\omega\ \omega)) \rightarrow^1_\beta \cdots$$

or at some point it could transform to

$$(\mathsf{K}\ (\omega\ \omega)) \rightarrow^1_\beta \lambda y[(\omega\ \omega)] \rightarrow^1_\beta \lambda y[(\omega\ \omega)] \rightarrow^1_\beta \cdots$$

4. $((\mathsf{K}\ \omega)\ \Omega)$ *is* $weakly$ *normalising because it can reduce to the normal form* ω *but it is* not $strongly$ *normalising because it also has an infinite reduction sequence*

$$((\mathsf{K}\ \omega)\ \Omega) \rightarrow^1_\beta ((\mathsf{K}\ \omega)\ \Omega) \rightarrow^1_\beta \cdots$$

Examples of Strong Normalization

Example 15.9

1. ((K ω) ω) *is strongly normalising because it reduces to the normal form* ω *in a single step.*
2. *Consider the term* ((S K) K)*. Its reduction sequences go as follows:*

$$((\mathsf{S}\ \mathsf{K})\ \mathsf{K}) \to^1_\beta \lambda z[((\mathsf{K}\ z)\ (\mathsf{K}\ z))] \to^1_\beta \lambda z[z] \equiv \mathsf{I}$$

16: Notions of Reduction

Reduction

For any function such as $p = \lambda x[3.x.x + 4.x + 1]$,

$$(p\ 2) = 3.2.2 + 4.2 + 1 = 21$$

However there is something *asymmetric* about the identity, in the sense that while $(p\ 2)$ deterministically produces $3.2.2 + 4.2 + 1$ which in turn simplifies deterministically to 21, it is not possible to deterministically infer that 21 came from $(p\ 2)$. It would be more accurate to refer to this sequence as a *reduction sequence* and capture the asymmetry as follows:

$$(p\ 2) \rightsquigarrow 3.2.2 + 4.2 + 1 \rightsquigarrow 21$$

And yet they are *behaviourally* equivalent and mutually substitutable in all contexts (*referentially transparent*).

1. Reduction (specifically β-reduction) captures this asymmetry.
2. Since reduction produces behaviourally *equal* terms we have the following notion of equality.

Untyped λ-Calculus: β-Equality

Definition 16.1 *β-equality or β-conversion (denoted $=_\beta$) is the smallest equivalence relation containing β-reduction ($\rightarrow_\beta^*$).*

The following are equivalent definitions.

1. $=_\beta$ is the reflexive-symmetric-transitive closure of 1-step β-reduction.
2. $=_\beta$ is the smallest relation defined by the following rules.

$$=_\beta \textbf{ Basis } \frac{L \rightarrow_\beta^* M}{L =_\beta M} \qquad =_\beta \textbf{ Reflexivity } \frac{}{L =_\beta L}$$

$$=_\beta \textbf{ Symmetry } \frac{L =_\beta M}{M =_\beta L} \qquad =_\beta \textbf{ Transitivity } \frac{L =_\beta M,\ M =_\beta N}{L =_\beta N}$$

The Paradoxical Combinator

Example 16.5 *Consider Curry's paradoxical combinator*

$$\mathsf{Y}_\mathsf{C} \stackrel{df}{=} \lambda f[(\mathsf{C}\ \mathsf{C})]$$

where

$$\mathsf{C} \stackrel{df}{=} \lambda x[(f\ (x\ x))]$$

asymmetry and
recursion is the motivation for it!

For any term L *we have*

$$\begin{aligned}
(\mathsf{Y}_\mathsf{C}\ L) &\to^1_\beta (\lambda x[(L\ (x\ x))]\ \lambda x[(L\ (x\ x))]) \\
&\equiv_\alpha (\lambda y[(L\ (y\ y))]\ \lambda x[(L\ (x\ x))]) \\
&\to^1_\beta (L\ \underbrace{(\lambda x[(L\ (x\ x))]\ \lambda x[(L\ (x\ x))])}) \\
&=_\beta (L\ \overbrace{(\mathsf{Y}_\mathsf{C}\ \mathsf{L})})
\end{aligned}$$

Hence $\boxed{(\mathsf{Y}_\mathsf{C}\ L) =_\beta (L\ (\mathsf{Y}_\mathsf{C}\ L))}$. *However* $(L\ (\mathsf{Y}_\mathsf{C}\ L))$ *will never* β*-reduce to* $(\mathsf{Y}_\mathsf{C}\ L)$.

Recursion and the Y combinator.

Since the lambda calculus only has variables and expressions and there is no place for names themselves (we use names such as K and S for our convenience in discourse, but the language itself allows only (untyped) variables and is meant to define functions anonymously as expressions in the language). In such a situation, recursion poses a problem in the language.

Recursion in most programming languages requires the use of an identifier which names an expression that contains a call to the very name of the function that it is supposed to define. This is at variance with the aim of the lambda calculus wherein the only names belong to variables and even functions may be defined anonymously as mere expressions.

This notion of recursive definitions may be generalised to a system of mutually recursive definitions.

The name of a recursive function, acts as a place holder in the body of the definition (which in turn has the name acting as a place holder for a copy of the body of the definition and so on ad infinitum). However no language can have sentences of infinite length.

The combinator Y_C helps in providing copies of any lambda term L whenever demanded in a more disciplined fashion. This helps in the modelling of recursive definitions anonymously. What the Y_C combinator provides is mechanism for recursion "unfolding" which is precisely our understanding of how recursion should work. Hence it is easy to see from $(\mathsf{Y}_\mathsf{C}\ L) =_\beta (L\ (\mathsf{Y}_\mathsf{C}\ L))$ that

$$(\mathsf{Y}_\mathsf{C}\ L) =_\beta (L\ (\mathsf{Y}_\mathsf{C}\ L)) =_\beta (L\ (L\ (\mathsf{Y}_\mathsf{C}\ L))) =_\beta (L\ (L\ (L\ (\mathsf{Y}_\mathsf{C}\ L)))) =_\beta \cdots$$

Many other researchers have defined other combinators which mimic the behaviour of the combinator

$\mathsf{Y_C}$. Of particular interest is Turing's combinator $\mathsf{Y_T} \stackrel{df}{=} (\mathsf{T}\ \mathsf{T})$ where $\mathsf{T} \stackrel{df}{=} \lambda x\ y[(y\ ((x\ x)\ y))]$. Notice that

$$\begin{array}{rl} & (\mathsf{T}\ \mathsf{T}) \\ \equiv & (\lambda x\ y[(y\ ((x\ x)\ y))]\ \mathsf{T}) \\ \rightarrow_\beta^1 & \lambda y[(y\ ((\mathsf{T}\ \mathsf{T})\ y))] \\ \equiv & \lambda y[(y\ (\mathsf{Y_T}\ y))] \end{array}$$

from which, by compatible closure, for any term L we get

$$\begin{array}{rl} & (\mathsf{Y_T}\ L) \\ \equiv & ((\mathsf{T}\ \mathsf{T})\ L) \\ \rightarrow_\beta^* & (\lambda y[(y\ (\mathsf{Y_T}\ y))]\ L) \\ \rightarrow_\beta^1 & (L\ (\mathsf{Y_T}\ L)) \end{array}$$

Thus $\mathsf{Y_T}$ is also a recursion unfolding combinator yielding

$$(\mathsf{Y_T}\ L) =_\beta (L\ (\mathsf{Y_T}\ L)) =_\beta (L\ (L\ (\mathsf{Y_T}\ L))) =_\beta (L\ (L\ (L\ (\mathsf{Y_T}\ L)))) =_\beta \cdots$$

Notice however that unlike the case of $(\mathsf{Y_C}\ L)$ which never *directly* reduces to $(L\ (\mathsf{Y_C}\ L))$, $(\mathsf{Y_T}\ L)$ does directly reduce to its unfolded version. That is,

$$(\mathsf{Y_T}\ L) \longrightarrow_\beta^* (L\ (\mathsf{Y_T}\ L)) \longrightarrow_\beta^* (L\ (L\ (\mathsf{Y_T}\ L))) \longrightarrow_\beta^* (L\ (L\ (L\ (\mathsf{Y_T}\ L)))) \longrightarrow_\beta^* \cdots$$

The Boolean Constants

$$\text{True} \stackrel{df}{=} \lambda x[\lambda y[x]] \qquad \text{(True)}$$

$$\text{False} \stackrel{df}{=} \lambda x[\lambda y[y]] \qquad \text{(False)}$$

Negation

$$\text{Not} \stackrel{df}{=} \lambda x[((x\ \text{False})\ \text{True})] \qquad \text{(not)}$$

The Conditional

$$\text{Ite} \stackrel{df}{=} \lambda x\ y\ z[(x\ y\ z)] \qquad \text{(ite)}$$

Exercise 17.1

1. *Prove that*

$$(\text{Not True}) =_{\beta\eta} \text{False} \tag{8}$$
$$(\text{Not False}) =_{\beta\eta} \text{True} \tag{9}$$

2. *Prove that*

$$(\text{Ite True } L\ M) =_{\beta\eta} L \tag{10}$$
$$(\text{Ite False } L\ M) =_{\beta\eta} M \tag{11}$$
$$\tag{12}$$

3. *We know from Theorem 7.7 that the boolean constants and the conditional form a functionally complete (adequate) set for propositional logic. Use the conditional combinator* Ite *and the constant combinators* True *and* False *to express the following boolean operators upto βη-equivalence.*
   - Not. *Verify that it is α-equivalent to (not).*
   - And*: conjunction*
   - Or*: disjunction*
   - Xor*: exclusive OR*
4. *Prove the de Morgan laws for the boolean combinators, using only βη-reductions.*
5. *Does* ((And K) I) *have a βη-normal form?*

The Church Numerals There are many ways to represent the natural numbers as lambda expressions. Here we present Church's original encoding of the naturals in the λ-calculus. We represent a natural n as a combinator n.

$$\mathsf{0} \stackrel{df}{=} \lambda f\ x[x] \qquad \text{(numeral-0)}$$

$$\mathsf{1} \stackrel{df}{=} \lambda f\ x[(f\ x)] \qquad \text{(numeral-1)}$$

$$\ldots$$

$$\mathsf{n+1} \stackrel{df}{=} \lambda f\ x[(f\ (f^n\ x))] \qquad \text{(numeral-n+1)}$$

$$\ldots$$

where $(f^n\ x)$ denotes the n-fold application of f to x. That is, $(f^n\ x) = \underbrace{(f\ (f\ \ldots\ (f\ x)\ldots))}_{f \text{ applied } n \text{ times}}$.

"Arithmagic"

We follow the operators of Peano arithmetic and the postulates of first order arithmetic (as treated in any course in first order logic) and obtain "magically"[1] the following combinators for the basic operations of arithmetic and checking for 0.

[1] There are geniuses out there somewhere who manage to come up with these things. Don't ask me how they thought of them!

$$\mathsf{Succ} \stackrel{df}{=} \lambda n\ f\ x[((n\ f)\ (f\ x))] \tag{Succ}$$

$$\mathsf{Add} \stackrel{df}{=} \lambda m\ n\ f\ x[((m\ f)\ (n\ f\ x))] \tag{Add}$$

$$\mathsf{IsZero} \stackrel{df}{=} \lambda n[(n\ \lambda x[\mathsf{False}]\ \mathsf{True})] \tag{13}$$

The only way to convince oneself that the above are correct, is to verify that they do produce the expected results.

Exercise 17.2

1. *Prove the following.*
   - *(a)* (Succ 0) $=_{\beta\eta}$ 1
   - *(b)* (Succ n) $=_{\beta\eta}$ n + 1
   - *(c)* (IsZero 0) $=_{\beta\eta}$ True
   - *(d)* (IsZero (Succ n)) $=_{\beta\eta}$ False
   - *(e)* (Add 0 n) $=_{\beta\eta}$ n
   - *(f)* (Add m 0) $=_{\beta\eta}$ m
   - *(g)* (Add m n) $=_{\beta\eta}$ p *where* p *denotes the combinator for* $p = m + n$
2. *Try to reduce* (Add K S) *to its* β*-normal form. Can you interpret the resulting lambda term as representing some meaningful function?*

Ordered Pairs and Tuples

$$\mathsf{Pair} \stackrel{df}{=} \lambda x\ y\ p[(p\ x\ y)] \tag{14}$$

$$\mathsf{Fst} \stackrel{df}{=} \lambda p[(p\ \mathsf{True})] \tag{15}$$

$$\mathsf{Snd} \stackrel{df}{=} \lambda p[(p\ \mathsf{False})] \tag{16}$$

(17)

We may define an n-tuple inductively as a pair consisting of the first element of the n-tuple and an $n-1$ tuple of the other $n-1$ elements. Let $\langle L, M\rangle$ represent a pair. We then have for any $n > 2$

$$\langle L_1, \ldots, L_n\rangle = (\mathsf{Pair}\ L_1\ \langle L_2, \ldots, L_n\rangle)$$

Note the isomorphism between lists of length n and n-tuples for each $n \geq 2$ (ordered pairs are 2-tuples).

Exercise 17.3

1. *Let* $P \stackrel{df}{=} (\mathsf{Pair}\ L\ M)$. *Verify that* $(\mathsf{Pair}\ (\mathsf{Fst}\ P)\ (\mathsf{Snd}\ P)) =_{\beta\eta} P$.

2. *Let* $\mathsf{Sfst} \stackrel{df}{=} (\mathsf{Fst}\ \mathsf{S})$ *and* $\mathsf{Ssnd} \stackrel{df}{=} (\mathsf{Snd}\ \mathsf{S})$.

   (a) *Compute the* $\beta\eta$ *normal form of* $(\mathsf{Pair}\ \mathsf{Sfst}\ \mathsf{Ssnd})$*? Is it* $\beta\eta$*-equal to* S*?*

   (b) *Now compute the* $\beta\eta$ *normal forms of* $(\mathsf{Fst}\ (\mathsf{Pair}\ \mathsf{Sfst}\ \mathsf{Ssnd}))$ *and* $(\mathsf{Snd}\ (\mathsf{Pair}\ \mathsf{Sfst}\ \mathsf{Ssnd}))$*. What are their* $\beta\eta$ *normal forms?*

   (c) *What can you conclude from the above?*

3. *For any* k, $0 \leq k < n$, *define combinators which extract the* k*-th component of an* n*-tuple.*

4. (a) *Define a combinator* $\mathsf{Bintree}$ *that constructs binary trees from* λ*-terms with node labels drawn from the Church numerals.*

   (b) *Define combinators* Root, Lst *and* Rst *which yield respectively the root, the left subtree and the right subtree of a binary tree.*

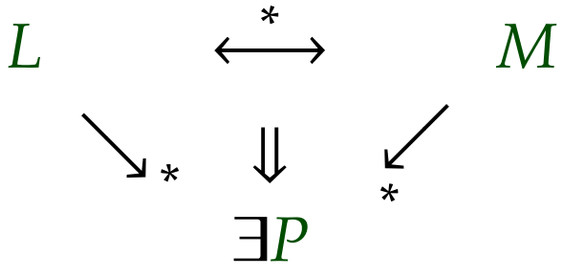
   (c) *Prove that for any such binary tree* B *expressed as a* λ*-term,* $(\mathsf{Bintree}\ (\mathsf{Root}\ B)\ (\mathsf{Lst}\ B)\ (\mathsf{Rst}\ B)) =_{\beta\eta} B$.

Reduction: Church-Rosser

Definition 18.9 $\longrightarrow$ *is* **Church-Rosser** *if for all* L, M,

$$L \overset{*}{\longleftrightarrow} M \Rightarrow \exists P : L \longrightarrow^{*} P \overset{*}{\longleftarrow} M$$

which we denote by



An Applied Lambda-Calculus

A Simple Language of Terms: FL0

Let X be an infinite collection of variables (names). Consider the language (actually a collection of abstract syntax trees) of terms $T_\Omega(X)$ defined by the following constructors (along with their intended meanings).

Construct	Arity	Informal Meaning
`Z`	0	The number **0**
`T`	0	The truth value **true**
`F`	0	The truth value **false**
`P`	1	The **predecessor** function on numbers
`S`	1	The **successor** function on numbers
`ITE`	3	The **if-then-else** construct (on numbers and truth values)
`IZ`	1	The **is-zero** predicate on numbers
`GTZ`	1	The **greater-than-zero** predicate on numbers

FL(X): Language, Datatype or Instruction Set?

The set of terms $T_\Omega(X)$ may be alternatively defined by the BNF:

$$t ::= x \in X \mid \texttt{Z} \mid (\texttt{P}\ t) \mid (\texttt{S}\ t) \mid \texttt{T} \mid \texttt{F} \mid (\texttt{ITE}\ \langle t, t_1, t_0 \rangle) \mid (\texttt{IZ}\ t) \mid (\texttt{GTZ}\ t) \tag{26}$$

- It could be thought of as a user-defined data-type
- It could be thought of as the instruction-set of a particularly simple hardware machine.
- It could be thought of as a simple functional programming language without recursion.
- It is a language with two simple types of data: integers and booleans
- Notice that the constructor $(\texttt{ITE}\ \langle t, t_1, t_0 \rangle)$ is overloaded.

Extending the language

To make this simple language safe we require

Type-checking : to ensure that arbitrary expressions are not mixed in ways they are not "intended" to be used. For example

- t cannot be a boolean expression in $\texttt{S}(t)$, $\texttt{P}(t)$, $\texttt{IZ}(t)$ and $\texttt{GTZ}(t)$
- $\texttt{ITE}(t, t_1, t_0)$ may be used as a conditional expression for both integers and booleans, but t needs to be a boolean and either both t_1 and t_0 are integer expressions or both are boolean expressions.

Functions : To be a useful programming language we need to be able to define functions.

Recursion : to be able to define complex functions in a well-typed fashion. Recursion should also be well-typed

Typing FL Expressions

We have only two types of objects – integers and booleans which we represent by `int` and `bool` respectively. We then have the following elementary typing annotations for the expressions, which may be obtained by pattern matching.

1. `Z` : `int`
2. `T` : `bool`
3. `F` : `bool`
4. `S` : `int` → `int`
5. `P` : `int` → `int`
6. `IZ` : `int` → `bool`
7. `GTZ` : `int` → `bool`
8. `ITEI` : `bool` * `int` * `int` → `int`
9. `ITEB` : `bool` * `bool` * `bool` → `bool`

Λ+FL(X): The Power of Functions

To make the language powerful we require the ability to define functions, both non-recursive and recursive. We define an applied lambda-calculus of lambda terms $\Lambda_\Omega(X)$ over this set of terms as follows:

$$L, M, N ::= t \in T_\Omega(X) \mid \lambda x[L] \mid (L\ M) \tag{27}$$

This is two-level grammar combining the term grammar (26) with λ-abstraction and λ-application.

While this makes it possible to use the operators of $T_\Omega(X)$ as part of functions (λ-expressions), it does not allow us to use the operators of $T_\Omega(X)$ outside of λ-abstractions and λ-applications.

Λ+FL(X): Lack of Higher-order Power?

Example 21.1 *The grammar (27) does not allow us to define expressions such as the following:*

1. *the successor of the result of an application* $(\mathtt{S}\ (L\ M))$
2. *higher order conditionals e.g.* $\lambda x[(\mathtt{ITE}\ \langle (L\ x), (M\ x), (N\ x)\rangle)]$ *where* $(L\ x)$ *yields a boolean value for an argument of the appropriate type.*
3. *In general, it does not allow the constructors to be applied to* λ*-expressions.*

So we extend the language by allowing a free intermixing of λ-terms and terms of the sub-language $T_\Omega(X)$.

$\Lambda_{FL}(X)$: Higher order functions

We need to *flatten* the grammar of (27) to allow λ-terms also to be used as arguments of the constructors of the term-grammar (26). The language of applied λ-terms (viz. $\Lambda_{\Omega}(X)$) now is defined by the grammar.

$$
\begin{aligned}
L, M, N ::= {} & x \in X \mid \mathtt{Z} \\
& \mid (\mathtt{P}\ L) \mid (\mathtt{S}\ L) \\
& \mid \mathtt{T} \mid \mathtt{F} \\
& \mid (\mathtt{IZ}\ L) \mid (\mathtt{GTZ}\ L) \\
& \mid (\mathtt{ITE}\ \langle L, M, N \rangle) \\
& \mid \lambda x[L] \mid (L\ M)
\end{aligned}
\tag{28}
$$

The Normal forms for Integers

We need reduction rules to simplify (non-recursive) expressions.

Zero . $\mathtt{Z}$ is the unique representation of the number 0 and every integer expression that is equal to 0 must be reducible to $\mathtt{Z}$.

Positive integers . Each positive integer k is uniquely represented by the expression $\mathtt{S}^k(\mathtt{Z})$ where the super-script k denotes a k-fold application of $\mathtt{S}$.

Negative integers . Each negative integer $-k$ is uniquely represented by the expression $\mathtt{P}^k(\mathtt{Z})$ where the super-script k denotes a k-fold application of $\mathtt{P}$.

δ-rules

$$(\mathtt{P}\ (\mathtt{S}\ x)) \longrightarrow_\delta x \tag{29}$$

$$(\mathtt{S}\ (\mathtt{P}\ x)) \longrightarrow_\delta x \tag{30}$$

Reduction Rules for Boolean Expressions

Pure Boolean Reductions . The constructs T and F are the normal forms for boolean values.

$$(\mathtt{ITE}\ \langle b, x, x\rangle) \longrightarrow_\delta x \tag{31}$$
$$(\mathtt{ITE}\ \langle \mathtt{T}, x, y\rangle) \longrightarrow_\delta x \tag{32}$$
$$(\mathtt{ITE}\ \langle \mathtt{F}, x, y\rangle) \longrightarrow_\delta y \tag{33}$$

Testing for zero .

$$(\mathtt{IZ}\ \mathtt{Z}) \longrightarrow_\delta \mathtt{T} \tag{34}$$
$$(\mathtt{IZ}\ (\mathtt{S}\ n)) \longrightarrow_\delta \mathtt{F}, \text{ where } (\mathtt{S}\ n) \text{ is a } \delta\text{-nf} \tag{35}$$
$$(\mathtt{IZ}\ (\mathtt{P}\ n)) \longrightarrow_\delta \mathtt{F}, \text{ where } (\mathtt{P}\ n) \text{ is a } \delta\text{-nf} \tag{36}$$

Testing for Positivity

$$(\texttt{GTZ Z}) \longrightarrow_\delta \texttt{F} \tag{37}$$

$$(\texttt{GTZ (S } n\texttt{)}) \longrightarrow_\delta \texttt{T}\text{, where } (\texttt{S } n) \text{ is a } \delta\text{-nf} \tag{38}$$

$$(\texttt{GTZ (P } n\texttt{)}) \longrightarrow_\delta \texttt{F}\text{, where } (\texttt{P } n) \text{ is a } \delta\text{-nf} \tag{39}$$

Other Non-recursive Operators

We may "program" the other boolean operations as follows:

$$\text{NOT} \stackrel{df}{=} \lambda x[\texttt{ITE}\ \langle x, \texttt{F}, \texttt{T}\rangle]$$

$$\text{AND} \stackrel{df}{=} \lambda\langle x, y\rangle[\texttt{ITE}\ \langle x, y, \texttt{F}\rangle]$$

$$\text{OR} \stackrel{df}{=} \lambda\langle x, y\rangle[\texttt{ITE}\ \langle x, \texttt{T}, y\rangle]$$

We may also "program" the other integer comparison operations as follows:

$$\text{GEZ} \stackrel{df}{=} \lambda x[\texttt{OR}\ \langle(\texttt{IZ}\ x), (\texttt{GTZ}\ x)\rangle]$$

$$\text{LTZ} \stackrel{df}{=} \lambda x[\texttt{NOT}\ (\texttt{GEZ}\ x)]$$

$$\text{LEZ} \stackrel{df}{=} \lambda x[\texttt{OR}\ \langle(\texttt{IZ}x), (\texttt{LTZ}\ x)\rangle]$$

Recursion in the Applied Lambda-calculus

The full power of a programming language will not be realised without a recursion mechanism. The untyped lambda-calculus has "paradoxical combinators" which behave like recursion operators upto $=_\beta$.

Definition 21.2 *A combinator* Y *is called a* **fixed-point combinator** *if for every lambda term* L, $\boxed{(\mathsf{Y}\ L) =_\beta (L\ (\mathsf{Y}\ L))}$

Curry's Υ combinator ($\mathsf{Y_C}$)

$$\mathsf{Y_C} \stackrel{df}{=} \lambda f[(\mathsf{C}\ \mathsf{C})] \text{ where } \mathsf{C} \stackrel{df}{=} \lambda x[(f\ (x\ x))]$$

Turing's Υ combinator ($\mathsf{Y_T}$)

$$\mathsf{Y_T} \stackrel{df}{=} (\mathsf{T}\ \mathsf{T}) \text{ where } \mathsf{T} \stackrel{df}{=} \lambda y\ x[(x\ (y\ y\ x\ ))]$$

But the various Y combinators unfortunately will not satisfy any typing rules that we may define for the language, because they are all "self-applicative" in nature.

$\Lambda_{RecFL}(X)$: Adding Recursion

Instead it is more convenient to use the fixed-point property and define a new constructor with a δ-rule which satisfies the fixed-point property (definition 21.2).

We extend the language FL with a new constructor

$$L ::= \ldots \quad | \quad (\text{REC } L)$$

and add the fixed point property as a δ-rule

$$(\text{REC } L) \longrightarrow_{\delta} (L\ (\text{REC } L)) \tag{40}$$

Recursion Example: Addition

Consider addition on integers as a binary operation to be defined in this language. We use the following properties of addition on the integers to define it by induction on the first argument.

$$x + y = \begin{cases} y & \text{if } x = 0 \\ (x-1) + (y+1) & \text{if } x > 0 \\ (x+1) + (y-1) & \text{if } x < 0 \end{cases} \tag{41}$$

Using the constructors of $\Lambda_{RecFL}(X)$ we require that any (curried) definition of addition on numbers should be a solution to the following equation in $\Lambda_{RecFL}(X)$ for all (integer) expression values of x and y.

$$(plusc\ x\ y) =_{\beta\delta} \texttt{ITE}\ \langle(\texttt{IZ}\ x), y, \texttt{ITE}\ \langle(\texttt{GTZ}\ x), (plusc\ (\texttt{P}\ x)\ (\texttt{S}\ y)), (plusc\ (\texttt{S}\ x)\ (\texttt{P}\ y))\rangle\rangle \tag{42}$$

Equation (42) may be rewritten using abstraction as follows:

$$plusc =_{\beta\delta} \lambda x[\lambda y[\texttt{ITE}\ \langle(\texttt{IZ}\ x), y, \texttt{ITE}\ \langle(\texttt{GTZ}\ x), (plusc\ (\texttt{P}\ x)\ (\texttt{S}\ y)), (plusc\ (\texttt{S}\ x)\ (\texttt{P}\ y))\rangle\rangle]] \tag{43}$$

We may think of equation (43) as an equation to be solved in the unknown variable *plusc*.

Consider the (applied) λ-term obtained from the right-hand-side of equation (43) by simply abstracting the unknown *plusc*.

$$\mathsf{addc} \stackrel{df}{=} \lambda f[\lambda x\ y[\texttt{ITE}\ \langle(\texttt{IZ}\ x), y, \texttt{ITE}\ \langle(\texttt{GTZ}\ x), (f\ (\texttt{P}\ x)\ (\texttt{S}\ y)), (f\ (\texttt{S}\ x)\ (\texttt{P}\ y))\rangle\rangle]] \tag{44}$$

Claim 21.3

$$(\texttt{REC}\ \mathsf{addc}) \longrightarrow_{\delta} (\mathsf{addc}\ (\texttt{REC}\ \mathsf{addc})) \tag{45}$$

and hence

$$(\texttt{REC}\ \mathsf{addc}) =_{\beta\delta} (\mathsf{addc}\ (\texttt{REC}\ \mathsf{addc})) \tag{46}$$

Claim 21.4 $(\texttt{REC}\ \mathsf{addc})$ *satisfies exactly the equation (43). That is*

$$((\texttt{REC}\ \mathsf{addc})\ x\ y) =_{\beta\delta} \texttt{ITE}\ \langle(\texttt{IZ}\ x), y, \texttt{ITE}\ \langle(\texttt{GTZ}\ x), ((\texttt{REC}\ \mathsf{addc})\ (\texttt{P}\ x)\ (\texttt{S}\ y)), ((\texttt{REC}\ \mathsf{addc})\ (\texttt{S}\ x)\ (\texttt{P}\ y))\rangle\rangle \tag{47}$$

Hence we may regard $(\texttt{REC}\ \mathsf{addc})$ where addc is defined by right-hand-side of definition (44) as the required solution to the equation (42) in which *plusc* is an unknown.

The abstraction shown in (44) and the claims (21.3) and (21.4) simply go to show that $M \equiv_\alpha \lambda f[\{f/z\}L]$ is a solution to the equation $z =_{\beta\delta} L$, whenever such a solution does exist. Further, the claims also show that we may "unfold" the recursion (on demand) by simply performing the substitution $\{L/z\}L$ for each free occurrence of z within L.

Exercise 21.1

1. *Prove that the relation* $\longrightarrow_\delta$ *is confluent.*
2. *The language FL does not have any operators that take boolean arguments and yields integer values. Define a standard conversion function* B2I *which maps the value* F *to* Z *and* T *to* (S Z).
3. *Prove that* $\mathsf{Y_C}$ *and* $\mathsf{Y_T}$ *are both fixed-point combinators.*
4. *Using the combinator* add *and the other constructs of* $\Lambda_\Sigma(X)$ *to*
   (a) *define the equation for products of numbers in the language.*
   (b) *define the multiplication operation* mult *on integers and prove that it satisfies the equation(s) for products.*
5. *The equation (41) is defined conditionally. However the following is equally valid for all integer values* x *and* y*.*

$$x + y = (x - 1) + (y + 1) \tag{48}$$

   (a) *Follow the steps used in the construction of* addc *to define a new applied* addc′ *that instead uses equation (48).*
   (b) *Is* (REC addc′) $=_{\beta\delta}$ (addc′ (REC addc′))*?*
   (c) *Is* addc $=_{\beta\delta}$ addc′*?*
   (d) *Is* (REC addc) $=_{\beta\delta}$ (REC addc′)*?*
6. *The function* addc *was defined in curried form. Use the pairing function in the untyped* λ*-calculus, to define*
   (a) *addition and multiplication as binary functions independently of the existing functions.*
   (b) *the binary 'curry' function which takes a binary function and its arguments and creates a curried version of the binary function.*

Typing FL expressions

We have already seen that the simple language FL has

- two kinds of expressions: integer expressions and boolean expressions,
- there are also constructors which take integer expressions as arguments and yield boolean values
- there are also function types which allow various kinds of functions to be defined on boolean expressions and integer expressions.

The Need for typing in FL

- A type is an important *attribute* of any variable, constant or expression, since every such object can only be used in certain kinds of expressions.
- Besides the need for type-checking rules on $T_\Omega(X)$ to prevent illegal constructor operations,
  - rules are necessary to ensure that λ-applications occur only between terms of appropriate types in order to remain meaningful.
  - rules are necessary to ensure that all terms have clearly defined types at compile-time so that there are no run-time type violations.

TL: A Simple Language of Types

Consider the following language of types (in fully parenthesized form) defined over an infinite collection $'a \in TV$ of type variables. We also have two type constants `int` and `bool`.

$$\sigma, \tau ::= \texttt{int} \mid \texttt{bool} \mid 'a \in TV \mid (\sigma * \tau) \mid (\sigma \rightarrow \tau)$$

Notes.

- `int` and `bool` are *type constants*.
- $*$ is the product operation on types and
- $\rightarrow$ is the function operator on types.
- We require $*$ because of the possibility of defining functions of various kinds of arities in $\Lambda_\Omega(X)$.
- **Precedence.** We assume $*$ has a higher precedence than $\rightarrow$.
- **Associativity.** $\rightarrow$ is *right* associative whereas $*$ is *left* associative.
- In any type expression τ, $TVar(\tau)$ is the set of type variables

Type-inference Rules: Infrastructure

The question of assigning types to complicated expressions which may have variables in them still remains to be addressed.

Type inferencing. Can be done using type assignment rules, by a recursive travel of the abstract syntax tree.

Free variables (names) are already present in the *environment* (symbol table).

Constants and Constructors. May have their types either pre-defined or there may be axioms assigning them types.

Bound variables. May be necessary to introduce "fresh" type variables in the environment.

Type Inferencing: Infrastructure

The elementary typing previously defined for the elementary expressions of FL does not suffice

1. in the presence of λ abstraction and application, which allow for higher-order functions to be defined
2. in the presence of polymorphism, especially when we do not want to unnecessarily decorate expressions with their types.

Type Assignment: Infrastructure

- Assume Γ is the environment[a] (an association list) which may be looked up to determine the types of individual names. For each variable $x \in X$, $\Gamma(x)$ yields the type of x i.e. $\Gamma(x) = \sigma$ if $x : \sigma \in \Gamma$.
- For each (sub-)expression in FL we define a set C of *type constraints* of the form $\sigma = \tau$, where T is the set of type variables used in C.
- The type constraints are defined by induction on the structure of the expressions in the language FL.
- The expressions of FL could have free variables. The type of the expression would then depend on the types assigned to the free variables. This is a simple kind of polymorphism.
- It may be necessary to generate new type variables as and when required during the process of inferencing and assignment.

[a] usually a part of the symbol table

Constraint Typing Relation

Definition 22.1 *For each term $L \in \Lambda_\Sigma(X)$ the* **constraint typing relation** *is of the form*

$$\Gamma \vdash L : \tau \ \rhd_T \ C$$

where

- Γ *is called the* context[a] *and defines the stack of assumptions*[b] *that may be needed to assign a type (expression) to the (sub-)expression* L.
- τ *is the type(-expression) assigned to* L
- C *is the set of constraints*
- T *is the set of "fresh" type variables used in the (sub-)derivations*

[a] usually in the symbol table
[b] including new type variables

Typing axioms: Basic

The following axioms (c.f Typing FL Expressions) may be applied during the scanning and parsing phases of the compiler to assign types to the individual tokens.

Z $\dfrac{}{\Gamma \vdash \mathtt{Z} : \underline{\mathtt{int}} \ \triangleright_{\emptyset} \ \emptyset}$

T $\dfrac{}{\Gamma \vdash \mathtt{T} : \underline{\mathtt{bool}} \ \triangleright_{\emptyset} \ \emptyset}$

F $\dfrac{}{\Gamma \vdash \mathtt{F} : \underline{\mathtt{bool}} \ \triangleright_{\emptyset} \ \emptyset}$

S $\dfrac{}{\Gamma \vdash \mathtt{S} : \underline{\mathtt{int}} \rightarrow \underline{\mathtt{int}} \ \triangleright_{\emptyset} \ \emptyset}$

P $\dfrac{}{\Gamma \vdash \mathtt{P} : \underline{\mathtt{int}} \rightarrow \underline{\mathtt{int}} \ \triangleright_{\emptyset} \ \emptyset}$

IZ $\dfrac{}{\Gamma \vdash \mathtt{IZ} : \underline{\mathtt{int}} \rightarrow \underline{\mathtt{bool}} \ \triangleright_{\emptyset} \ \emptyset}$

GTZ $\dfrac{}{\Gamma \vdash \mathtt{GTZ} : \underline{\mathtt{int}} \rightarrow \underline{\mathtt{bool}} \ \triangleright_{\emptyset} \ \emptyset}$

ITEI $\dfrac{}{\Gamma \vdash \mathtt{ITE} : \underline{\mathtt{bool}} * \underline{\mathtt{int}} * \underline{\mathtt{int}} \rightarrow \underline{\mathtt{int}} \ \triangleright_{\emptyset} \ \emptyset}$

ITEB $\dfrac{}{\Gamma \vdash \mathtt{ITE} : \underline{\mathtt{bool}} * \underline{\mathtt{bool}} * \underline{\mathtt{bool}} \rightarrow \underline{\mathtt{bool}} \ \triangleright_{\emptyset} \ \emptyset}$

Notice that the constructor ITE is *overloaded* and actually is two constructors ITEI and ITEB. Which constructor is actually used will depend on the context and the type-inferencing mechanism.

Type Rules for FL: 2

$$\textbf{Var}\ \frac{}{\Gamma \vdash x : \Gamma(x) \ \rhd_{\emptyset}\ \emptyset}$$

$$\textbf{Abs}\ \frac{\Gamma, x : \sigma \vdash L : \tau \ \rhd_T \ C}{\Gamma \vdash \lambda x[L] : \sigma \rightarrow \tau \ \rhd_T \ C}$$

$$\textbf{App}\ \frac{\begin{array}{c}\Gamma \vdash L : \sigma \ \rhd_{T_1} \ C_1 \\ \Gamma \vdash M : \tau \ \rhd_{T_2} \ C_2\end{array}}{\Gamma \vdash (L\ M) : {'a} \ \rhd_{T'} \ C'}\ \textbf{(Conditions 1. and 2.)}$$

where

- **Condition 1.** $T_1 \cap T_2 = T_1 \cap TVar(\tau) = T_2 \cap TVar(\sigma) = \emptyset$
  Condition 2. ${'a} \notin T_1 \cup T_2 \cup TVar(\sigma) \cup TVar(\tau) \cup TVar(C_1) \cup TVar(C_2)$.
- $T' = T_1 \cup T_2 \cup \{{'a}\}$
- $C' = C_1 \cup C_2 \cup \{\sigma = \tau \rightarrow {'a}\}$

Example 22.2 *Consider the following simple combinator* $\lambda x[\lambda y[\lambda z[(x\ (y\ z))]]]$ *which defines the function composition operator. Since there are three bound variables* x, y *and* z *we begin with an initial assumption* $\Gamma = x : 'a, y : 'b, z : 'c$ *which assign arbitrary types to the bound variables, represented by the type variables* $'a$, $'b$ *and* $'c$ *respectively. Note however, that since it has no free variables, its type does not depend on the types of any variables. We expect that at the end of the proof there would be no assumptions.Our inference for the type of the combinator then proceeds as follows.*

1. $x : 'a, y : 'b, z : 'c \vdash x : 'a \ \triangleright_{\emptyset}\ \emptyset$ **(Var)**
2. $x : 'a, y : 'b, z : 'c \vdash y : 'b \ \triangleright_{\emptyset}\ \emptyset$ **(Var)**
3. $x : 'a, y : 'b, z : 'c \vdash z : 'c \ \triangleright_{\emptyset}\ \emptyset$ **(Var)**
4. $x : 'a, y : 'b, z : 'c \vdash (y\ z) : 'd \ \triangleright_{\{'d\}}\ \{'b = 'c \rightarrow 'd\}$ **(App)**
5. $x : 'a, y : 'b, z : 'c \vdash (x\ (y\ z)) : 'e \ \triangleright_{\{'d,'e\}}\ \{'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e\}$ **(App)**
6. $x : 'a, y : 'b \vdash \lambda z[(x\ (y\ z))] : 'c \rightarrow 'e \ \triangleright_{\{'d,'e\}}\ \{'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e\}$ **(Abs)**
7. $x : 'a \vdash \lambda x[\lambda y[\lambda z[(x\ (y\ z))]]] : 'b \rightarrow 'c \rightarrow 'e \ \triangleright_{\{'d,'e\}}\ \{'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e\}$ **(Abs)**
8. $\vdash \lambda x[\lambda y[\lambda z[(x\ (y\ z))]]] : 'a \rightarrow 'b \rightarrow 'c \rightarrow 'e \ \triangleright_{\{'d,'e\}}\ \{'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e\}$ **(Abs)**

Hence $\lambda x[\lambda y[\lambda z[(x\ (y\ z))]]] : 'a \rightarrow 'b \rightarrow 'c \rightarrow 'e$ *subject to the constraints given by* $\{'b = 'c \rightarrow 'd, 'a = 'd \rightarrow 'e\}$ *which yields* $\lambda x[\lambda y[\lambda z[(x\ (y\ z))]]] : ('d \rightarrow 'e) \rightarrow ('c \rightarrow 'd) \rightarrow 'c \rightarrow 'e$

Principal Type Schemes

Definition 22.3 *A solution for* $\Gamma \vdash L : \tau \ \rhd_T \ C$ *is a pair* $\langle S, \sigma \rangle$ *where* S *is a substitution of type variables in* τ *such that* $S(\tau) = \sigma$.

- The rules yield a *principal type scheme* for each well-typed applied λ-term.
- The term is *ill-typed* if there is no solution that satisfies the constraints.
- Any substitution of the type variables which satisfies the constraints C is an instance of the most general polymorphic type that may be assigned to the term.

Exercise 22.1

1. *The language has several constructors which behave like functions. Derive the following rules for terms in* $T_\Omega(X)$ *from the basic typing axioms and the rule* **App**.

$$\textbf{Sx}\ \frac{\Gamma \vdash t : \tau \ \triangleright_T \ C}{\Gamma \vdash (\mathtt{S}\ t) : \underline{\mathtt{int}} \ \triangleright_T \ C \cup \{\tau = \underline{\mathtt{int}}\}}$$

$$\textbf{Px}\ \frac{\Gamma \vdash t : \tau \ \triangleright_T \ C}{\Gamma \vdash (\mathtt{P}\ t) : \underline{\mathtt{int}} \ \triangleright_T \ C \cup \{\tau = \underline{\mathtt{int}}\}}$$

$$\textbf{IZx}\ \frac{\Gamma \vdash t : \tau \ \triangleright_T \ C}{\Gamma \vdash (\mathtt{IZ}\ t) : \underline{\mathtt{bool}} \ \triangleright_T \ C \cup \{\tau = \underline{\mathtt{int}}\}}$$

$$\textbf{GTZx}\ \frac{\Gamma \vdash t : \tau \ \triangleright_T \ C}{\Gamma \vdash (\mathtt{GTZ}\ t) : \underline{\mathtt{bool}} \ \triangleright_T \ C \cup \{\tau = \underline{\mathtt{int}}\}}$$

$$\textbf{ITEx}\ \frac{\begin{array}{c}\Gamma \vdash t : \sigma \ \triangleright_T \ C \\ \Gamma \vdash t_1 : \tau \ \triangleright_{T_1} \ C_1 \\ \Gamma \vdash t_0 : \upsilon \ \triangleright_{T_0} \ C_0\end{array}}{\Gamma \vdash (\mathtt{ITE}\ \langle t, t_1, t_0\rangle) : \tau \ \triangleright_{T'} \ C'} \quad (T \cap T_1 = T_1 \cap T_0 = T_0 \cap T = \emptyset)$$

where $T' = T \cup T_1 \cup T_0$ *and* $C' = C \cup C_1 \cup C_0 \cup \{\sigma = \underline{\mathtt{bool}}, \tau = \upsilon\}$

2. *Use the rules to define the type of the combinators* K *and* S*?*

3. *How would you define a type assignment for the recursive function* addc *defined by equation (44).*

4. *Prove that the terms,* $\omega = \lambda x[(x\ x)]$ *and* $\Omega = (\omega\ \omega)$ *are ill-typed.*

5. *Are the following well-typed or ill-typed? Prove your answer.*

   (a) (K S)

   (b) ((K S) ω)

   (c) (((S K) K) ω)

   (d) (ITE $\langle$(IZ x), T, (K x)$\rangle$)