is no corresponding complexity measure in the proof of the strong normalization theorem.

Please refer to Chapters 4 and 6 of "Proofs and Types" by Girard, Lafont, and Taylor [2] for the proofs of Theorems 7.1 and 7.2, respectively.

# 8 Polymorphism

The polymorphic lambda calculus, also known as "System F", is obtained extending the Curry-Howard isomorphism to the quantifier $\forall$. For example, consider the identity function $\lambda x^A.x$. This function has type $A \to A$. Another identity function is $\lambda x^B.x$ of type $B \to B$, and so forth for every type. We can thus think of the identity function as a family of functions, one for each type. In the polymorphic lambda calculus, there is a dedicated syntax for such families, and we write $\Lambda\alpha.\lambda x^\alpha.x$ of type $\forall\alpha.\alpha \to \alpha$.

System F was independently discovered by Jean-Yves Girard and John Reynolds in the early 1970's.

## 8.1 Syntax of System F

The primary difference between System F and simply-typed lambda calculus is that System F has a new kind of function that takes a *type*, rather than a *term*, as its argument. We can also think of such a function as a family of terms that is indexed by a type.

Let $\alpha, \beta, \gamma$ range over a countable set of *type variables*. The types of System F are given by the grammar

$$\text{Types:}\quad A, B ::= \alpha \;\big|\; A \to B \;\big|\; \forall\alpha.A$$

A type of the form $A \to B$ is called a *function type*, and a type of the form $\forall\alpha.A$ is called a *universal type*. The type variable $\alpha$ is bound in $\forall\alpha.A$, and we identify types up to renaming of bound variables; thus, $\forall\alpha.\alpha \to \alpha$ and $\forall\beta.\beta \to \beta$ are the same type. We write $FTV(A)$ for the set of free type variables of a type $A$, defined inductively by:

- $FTV(\alpha) = \{\alpha\}$,
- $FTV(A \to B) = FTV(A) \cup FTV(B)$,

$$
\begin{array}{cc}
(\textit{var}) & \dfrac{}{\Gamma, x{:}A \vdash x : A} \\[2ex]
(\textit{app}) & \dfrac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \\[2ex]
(\textit{abs}) & \dfrac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x^A.M : A \to B} \\[2ex]
(\textit{typeapp}) & \dfrac{\Gamma \vdash M : \forall\alpha.A}{\Gamma \vdash MB : A[B/\alpha]} \\[2ex]
(\textit{typeabs}) & \dfrac{\Gamma \vdash M : A \qquad \alpha \notin FTV(\Gamma)}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.A}
\end{array}
$$

Table 6: Typing rules for System F

- $FTV(\forall\alpha.A) = FTV(A) \setminus \{\alpha\}$.

We also write $A[B/\alpha]$ for the result of replacing all free occurrences of $\alpha$ by $B$ in $A$. Just like the substitution of terms (see Section 2.3), this type substitution must be *capture-free*, i.e., special care must be taken to rename any bound variables of $A$ so that their names are different from the free variables of $B$.

The terms of System F are:

$$\text{Terms:}\quad M, N ::= x \mid MN \mid \lambda x^A.M \mid MA \mid \Lambda\alpha.M$$

Of these, variables $x$, applications $MN$, and lambda abstractions $\lambda x^A.M$ are exactly as for the simply-typed lambda calculus. The new terms are *type application* $MA$, which is the application of a type function $M$ to a type $A$, and *type abstraction* $\Lambda\alpha.M$, which denotes the type function that maps a type $\alpha$ to a term $M$. The typing rules for System F are shown in Table 6.

We also write $FTV(M)$ for the set of free type variables in the term $M$. We need a final notion of substitution: if $M$ is a term, $B$ a type, and $\alpha$ a type variable, we write $M[B/\alpha]$ for the capture-free substitution of $B$ for $\alpha$ in $M$.

## 8.2 Reduction rules

In System F, there are two rules for $\beta$-reduction. The first one is the familiar rule for the application of a function to a term. The second one is an analogous rule for the application of a type function to a type.

$$\begin{array}{lllll} (\beta_{\to}) & (\lambda x^A.M)N & \to & M[N/x], \\ (\beta_{\forall}) & (\Lambda\alpha.M)A & \to & M[A/\alpha], \end{array}$$

Similarly, there are two rules for $\eta$-reduction.

$$\begin{array}{lllll} (\eta_{\to}) & \lambda x^A.Mx & \to & M, & \text{if } x \not\in FV(M), \\ (\eta_{\forall}) & \Lambda\alpha.M\alpha & \to & M, & \text{if } \alpha \not\in FTV(M). \end{array}$$

The congruence and $\xi$-rules are as expected:

$$\frac{M \to M'}{MN \to M'N} \qquad \frac{N \to N'}{MN \to MN'} \qquad \frac{M \to M'}{\lambda x^A M \to \lambda x^A M'}$$

$$\frac{M \to M'}{MA \to M'A} \qquad \frac{M \to M'}{\Lambda\alpha M \to \Lambda\alpha M'}$$

## 8.3 Examples

Just as in the untyped lambda calculus, many interesting data types and operations can be encoded in System F.

### 8.3.1 Booleans

Define the System F type **bool**, and terms $\mathbf{T}, \mathbf{F} : \textbf{bool}$, as follows:

$$\begin{array}{rcl} \textbf{bool} & = & \forall\alpha.\alpha \to \alpha \to \alpha, \\ \mathbf{T} & = & \Lambda\alpha.\lambda x^\alpha.\lambda y^\alpha.x, \\ \mathbf{F} & = & \Lambda\alpha.\lambda x^\alpha.\lambda y^\alpha.y. \end{array}$$

It is easy to see from the typing rules that $\vdash \mathbf{T} : \textbf{bool}$ and $\vdash \mathbf{F} : \textbf{bool}$ are valid typing judgments. We can define an if-then-else operation

$$\begin{array}{rl} \textbf{if\_then\_else} & : \forall\beta.\,\textbf{bool} \to \beta \to \beta \to \beta, \\ \textbf{if\_then\_else} & = \Lambda\beta.\lambda z^{\textbf{bool}}.z\beta. \end{array}$$

It is then easy to see that, for any type $B$ and any pair of terms $M, N : B$, we have

$$\begin{array}{rcl} \textbf{if\_then\_else}\ B\,\mathbf{T}\,MN & \twoheadrightarrow_\beta & M, \\ \textbf{if\_then\_else}\ B\,\mathbf{F}\,MN & \twoheadrightarrow_\beta & N. \end{array}$$

Once we have if-then-else, it is easy to define other boolean operations, for example

$$\begin{array}{rcl}\textbf{and} & = & \lambda a^{\textbf{bool}}.\lambda b^{\textbf{bool}}.\,\textbf{if\_then\_else bool}\;a\,b\,\mathbf{F},\\ \textbf{or} & = & \lambda a^{\textbf{bool}}.\lambda b^{\textbf{bool}}.\,\textbf{if\_then\_else bool}\;a\,\mathbf{T}\,b,\\ \textbf{not} & = & \lambda a^{\textbf{bool}}.\,\textbf{if\_then\_else bool}\;a\,\mathbf{F}\,\mathbf{T}.\end{array}$$

Later, in Proposition 8.8, we will show that up to $\beta\eta$ equality, $\mathbf{T}$ and and $\mathbf{F}$ are the *only* closed terms of type **bool**. This, together with the if-then-else operation, justifies calling this the type of booleans.

Note that the above encodings of the booleans and their if-then-else operation in System F is exactly the same as the corresponding encodings in the untyped lambda calculus from Section 3.1, provided that one erases all the types and type abstractions. However, there is an important difference: in the untyped lambda calculus, the booleans were just two terms among many, and there was no guarantee that the argument of a boolean function (such as **and** and **or**) was actually a boolean. In System F, the typing guarantees that all closed boolean terms eventually reduce to either $\mathbf{T}$ or $\mathbf{F}$.

### 8.3.2 Natural numbers

We can also define a type of Church numerals in System F. We define:

$$\begin{array}{rcl}\textbf{nat} & = & \forall\alpha.(\alpha\to\alpha)\to\alpha\to\alpha,\\ \overline{0} & = & \Lambda\alpha.\lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.x,\\ \overline{1} & = & \Lambda\alpha.\lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.fx,\\ \overline{2} & = & \Lambda\alpha.\lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.f(fx),\\ \overline{3} & = & \Lambda\alpha.\lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.f(f(fx)),\\ \ldots & & \end{array}$$

It is then easy to define simple functions, such as successor, addition, and multiplication:

$$\begin{array}{rcl}\textbf{succ} & = & \lambda n^{\textbf{nat}}.\Lambda\alpha.\lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.f(n\alpha fx),\\ \textbf{add} & = & \lambda n^{\textbf{nat}}.\lambda m^{\textbf{nat}}.\Lambda\alpha.\lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.n\alpha f(m\alpha fx),\\ \textbf{mult} & = & \lambda n^{\textbf{nat}}.\lambda m^{\textbf{nat}}.\Lambda\alpha.\lambda f^{\alpha\to\alpha}.n\alpha(m\alpha f).\end{array}$$

Just as for the booleans, these encodings of the Church numerals and functions are exactly the same as those of the untyped lambda calculus from Section 3.2, if one erases all the types and type abstractions. We will show in Proposition 8.9 below that the Church numerals are, up to $\beta\eta$-equivalence, the only closed terms of type **nat**.

### 8.3.3 Pairs

You will have noticed that we didn't include a cartesian product type $A \times B$ in the definition of System F. This is because such a type is definable. Specifically, let

$$\begin{array}{rcl} A \times B & = & \forall\alpha.(A \to B \to \alpha) \to \alpha, \\ \langle M, N\rangle & = & \Lambda\alpha.\lambda f^{A\to B\to\alpha}.fMN. \end{array}$$

Note that when $M : A$ and $N : B$, then $\langle M, N\rangle : A \times B$. Moreover, for any pair of types $A, B$, we have projection functions $\pi_1 AB : A \times B \to A$ and $\pi_2 AB : A \times B \to B$, defined by

$$\begin{array}{rcl} \pi_1 & = & \Lambda\alpha.\Lambda\beta.\lambda p^{\alpha\times\beta}.p\alpha(\lambda x^\alpha.\lambda y^\beta.x), \\ \pi_2 & = & \Lambda\alpha.\Lambda\beta.\lambda p^{\alpha\times\beta}.p\beta(\lambda x^\alpha.\lambda y^\beta.y). \end{array}$$

This satisfies the usual laws

$$\begin{array}{rcl} \pi_1 AB\langle M, N\rangle & \twoheadrightarrow_\beta & M, \\ \pi_2 AB\langle M, N\rangle & \twoheadrightarrow_\beta & N. \end{array}$$

Once again, these encodings of pairs and projections are exactly the same as those we used in the untyped lambda calculus, when one erases all the type-related parts of the terms. You will show in Exercise 36 that every closed term of type $A \times B$ is $\beta\eta$-equivalent to a term of the form $\langle M, N\rangle$.

*Remark* 8.1. It is also worth noting that the corresponding $\eta$-laws, such as

$$\langle \pi_1 AB\, M, \pi_2 AB\, M\rangle = M,$$

are *not* derivable in System F. These laws hold whenever $M$ is a closed term, but not necessarily when $M$ contains free variables.

**Exercise 33.** Find suitable encodings in System F of the types 1, $A + B$, and 0, along with the corresponding terms $*$, $\mathrm{in}_1$, $\mathrm{in}_2$, case $M$ of $x^A \Rightarrow N \mid y^B \Rightarrow P$, and $\Box_A M$.

## 8.4 Church-Rosser property and strong normalization

**Theorem 8.2** (Church-Rosser)**.** *System F satisfies the Church-Rosser property, both for $\beta$-reduction and for $\beta\eta$-reduction.*

**Theorem 8.3** (Strong normalization)**.** *In System F, all terms are strongly normalizing.*

The proof of the Church-Rosser property is similar to that of the simply-typed lambda calculus, and is left as an exercise. The proof of strong normalization is much more complex; it can be found in Chapter 14 of "Proofs and Types" [2].

## 8.5 The Curry-Howard isomorphism

From the point of view of the Curry-Howard isomorphism, $\forall\alpha.A$ is the universally quantified logical statement "for all $\alpha$, $A$ is true". Here $\alpha$ ranges over atomic propositions. For example, the formula $\forall\alpha.\forall\beta.\alpha \to (\beta \to \alpha)$ expresses the valid fact that the implication $\alpha \to (\beta \to \alpha)$ is true for all propositions $\alpha$ and $\beta$. Since this quantifier ranges over *propositions*, it is called a *second-order quantifier*, and the corresponding logic is *second-order propositional logic*.

Under the Curry-Howard isomorphism, the typing rules for System F become the following logical rules:

- (Axiom)

$$(ax_x)\ \frac{}{\Gamma, x{:}A \vdash A}$$

- ($\to$-introduction)

$$(\to\text{-}I_x)\ \frac{\Gamma, x{:}A \vdash B}{\Gamma \vdash A \to B}$$

- ($\to$-elimination)

$$(\to\text{-}E)\ \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

- ($\forall$-introduction)

$$(\forall\text{-}I)\ \frac{\Gamma \vdash A \qquad \alpha \notin FTV(\Gamma)}{\Gamma \vdash \forall\alpha.A}$$

- ($\forall$-elimination)

$$(\forall\text{-}E)\ \frac{\Gamma \vdash \forall\alpha.A}{\Gamma \vdash A[B/\alpha]}$$

The first three of these rules are familiar from propositional logic.

The $\forall$-introduction rule is also known as *universal generalization*. It corresponds to a well-known logical reasoning principle: If a statement $A$ has been proven for some *arbitrary* $\alpha$, then it follows that it holds for *all* $\alpha$. The requirement that $\alpha$ is "arbitrary" has been formalized in the logic by requiring that $\alpha$ does not appear in any of the hypotheses that were used to derive $A$, or in other words, that $\alpha$ is not among the free type variables of $\Gamma$.

The $\forall$-elimination rule is also known as *universal specialization*. It is the simple principle that if some statement is true for all propositions $\alpha$, then the same statement is true for any particular proposition $B$. Note that, unlike the $\forall$-introduction rule, this rule does not require a side condition.

Finally, we note that the side condition in the $\forall$-introduction rule is of course the same as that of the typing rule (*typeabs*) of Table 6. From the point of view of logic, the side condition is justified because it asserts that $\alpha$ is "arbitrary", i.e., no assumptions have been made about it. From a lambda calculus view, the side condition also makes sense: otherwise, the term $\lambda x^{\alpha}.\Lambda\alpha.x$ would be well-typed of type $\alpha \to \forall\alpha.\alpha$, which clearly does not make any sense: there is no way that an element $x$ of some fixed type $\alpha$ could suddenly become an element of an arbitrary type.

## 8.6 Supplying the missing logical connectives

It turns out that a logic with only implication $\to$ and a second-order universal quantifier $\forall$ is sufficient for expressing all the other usual logical connectives, for example:

$$\begin{aligned}
A \wedge B &\iff \forall\alpha.(A \to B \to \alpha) \to \alpha, & (1)\\
A \vee B &\iff \forall\alpha.(A \to \alpha) \to (B \to \alpha) \to \alpha, & (2)\\
\neg A &\iff \forall\alpha.A \to \alpha, & (3)\\
\top &\iff \forall\alpha.\alpha \to \alpha, & (4)\\
\bot &\iff \forall\alpha.\alpha, & (5)\\
\exists\beta.A &\iff \forall\alpha.(\forall\beta.(A \to \alpha)) \to \alpha. & (6)
\end{aligned}$$

**Exercise 34.** Using informal intuitionistic reasoning, prove that the left-hand side is logically equivalent to the right-hand side for each of (1)–(6).

*Remark* 8.4. The definitions (1)–(6) are somewhat reminiscent of De Morgan's laws and double negations. Indeed, if we replace the type variable $\alpha$ by the constant $\mathbf{F}$ in (1), the right-hand side becomes $(A \to B \to \mathbf{F}) \to \mathbf{F}$, which is intuitionistically equivalent to $\neg\neg(A \wedge B)$. Similarly, the right-hand side of (2) becomes $(A \to \mathbf{F}) \to (B \to \mathbf{F}) \to \mathbf{F}$, which is intuitionistically equivalent to

$\neg(\neg A \wedge \neg B)$, and similarly for the remaining connectives. However, the versions of (1), (2), and (6) using **F** are only *classically*, but not *intuitionistically* equivalent to their respective left-hand sides. On the other hand, it is remarkable that by the use of $\forall\alpha$, each right-hand side is *intuitionistically* equivalent to the left-hand sides.

*Remark* 8.5. Note the resemblance between (1) and the definition of $A \times B$ given in Section 8.3.3. Naturally, this is not a coincidence, as logical conjunction $A \wedge B$ should correspond to cartesian product $A \times B$ under the Curry-Howard correspondence. Indeed, by applying the same principle to the other logical connectives, one arrives at a good hint for Exercise 33.

**Exercise 35.** Extend System F with an existential quantifier $\exists\beta.A$, not by using (6), but by adding a new type with explicit introduction and elimination rules to the language. Justify the resulting rules by comparing them with the usual rules of mathematical reasoning for "there exists". Can you explain the meaning of the type $\exists\beta.A$ from a programming language or lambda calculus point of view?

## 8.7 Normal forms and long normal forms

Recall that a $\beta$-normal form of System F is, by definition, a term that contains no $\beta$-redex, i.e., no subterm of the form $(\lambda x^A.M)N$ or $(\Lambda\alpha.M)A$. The following proposition gives another useful way to characterize the $\beta$-normal forms.

**Proposition 8.6** (Normal forms). *A term of System F is a $\beta$-normal form if and only if it is of the form*

$$\Lambda a_1.\Lambda a_2 \ldots \Lambda a_n.zQ_1Q_2 \ldots Q_k, \qquad (7)$$

*where:*

- $n \geqslant 0$ and $k \geqslant 0$;
- Each $\Lambda a_i$ is either a lambda abstraction $\lambda x_i^{A_i}$ or a type abstraction $\Lambda\alpha_i$;
- Each $Q_j$ is either a term $M_j$ or a type $B_j$; and
- Each $Q_j$, when it is a term, is recursively in normal form.

*Proof.* First, it is clear that every term of the form (7) is in normal form: the term cannot itself be a redex, and the only place where a redex could occur is inside one of the $Q_j$, but these are assumed to be normal.

For the converse, consider a term $M$ in $\beta$-normal form. We show that $M$ is of the form (7) by induction on $M$.

- If $M = z$ is a variable, then it is of the form (7) with $n = 0$ and $k = 0$.
- If $M = NP$ is normal, then $N$ is normal, so by induction hypothesis, $N$ is of the form (7). But since $NP$ is normal, $N$ cannot be a lambda abstraction, so we must have $n = 0$. It follows that $NP = zQ_1Q_2\ldots Q_kP$ is itself of the form (7).
- If $M = \lambda x^A.N$ is normal, then $N$ is normal, so by induction hypothesis, $N$ is of the form (7). It follows immediately that $\lambda x^A.N$ is also of the form (7).
- The case for $M = NA$ is like the case for $M = NP$.
- The case for $M = \Lambda\alpha.N$ is like the case for $M = \lambda x^A.N$. □

**Definition.** In a term of the form (7), the variable $z$ is called the *head variable* of the term.

Of course, by the Church-Rosser property together with strong normalization, it follows that every term of System F is $\beta$-equivalent to a unique $\beta$-normal form, which must then be of the form (7). On the other hand, the normal forms (7) are not unique up to $\eta$-conversion; for example, $\lambda x^{A\to B}.x$ and $\lambda x^{A\to B}.\lambda y^A.xy$ are $\eta$-equivalent terms and are both of the form (7). In order to achieve uniqueness up to $\beta\eta$-conversion, we introduce the notion of a *long normal form*.

**Definition.** A term of System F is a *long normal form* if

- it is of the form (7);
- the body $zQ_1\ldots Q_k$ is of atomic type (i.e., its type is a type variable); and
- each $Q_j$, when it is a term, is recursively in long normal form.

**Proposition 8.7.** *Every term of System F is $\beta\eta$-equivalent to a unique long normal form.*

*Proof.* By strong normalization and the Church-Rosser property of $\beta$-reduction, we already know that every term is $\beta$-equivalent to a unique $\beta$-normal form. It therefore suffices to show that every $\beta$-normal form is $\eta$-equivalent to a unique long normal form.

We first show that every $\beta$-normal form is $\eta$-equivalent to some long normal form. We prove this by induction. Indeed, consider a $\beta$-normal form of the form (7). By induction hypothesis, each of $Q_1, \ldots, Q_k$ can be $\eta$-converted to long normal form. Now we proceed by induction on the type $A$ of $zQ_1 \ldots Q_k$. If $A = \alpha$ is atomic, then the normal form is already long, and there is nothing to show. If $A = B \to C$, then we can $\eta$-expand (7) to

$$\Lambda a_1.\Lambda a_2 \ldots \Lambda a_n.\lambda w^B.zQ_1Q_2 \ldots Q_k w$$

and proceed by the inner induction hypothesis. If $A = \forall\alpha.B$, then we can $\eta$-expand (7) to

$$\Lambda a_1.\Lambda a_2 \ldots \Lambda a_n.\Lambda\alpha.zQ_1Q_2 \ldots Q_k\alpha$$

and proceed by the inner induction hypothesis.

For uniqueness, we must show that no two different long normal forms can be $\beta\eta$-equivalent to each other. We leave this as an exercise. $\square$

## 8.8 The structure of closed normal forms

It is a remarkable fact that if $M$ is in long normal form, then a lot of the structure of $M$ is completely determined by its type. Specifically: if the type of $M$ is atomic, then $M$ must start with a head variable. If the type of $M$ is of the form $B \to C$, then $M$ must be, up to $\alpha$-equivalence, of the form $\lambda x^B.N$, where $N$ is a long normal form of type $C$. And if the type of $M$ is of the form $\forall\alpha.C$, then $M$ must be, up to $\alpha$-equivalence, of the form $\Lambda\alpha.N$, where $N$ is a long normal form of type $C$.

So for example, consider the type

$$A = B_1 \to B_2 \to \forall\alpha_3.B_4 \to \forall\alpha_5.\beta.$$

We say that this type have five *prefixes*, where each prefix is of the form "$B_i \to$" or "$\forall\alpha_i.$". Therefore, every long normal form of type $A$ must also start with five prefixes; specifically, it must start with

$$\lambda x_1^{B_1}.\lambda x_2^{B_2}.\Lambda\alpha_3.\lambda x_4^{B_4}.\Lambda\alpha_5. \ldots$$

The next part of the long normal form is a choice of head variable. If the term is closed, the head variable must be one of the $x_1$, $x_2$, or $x_4$. Once the head variable has been chosen, then *its* type determines how many arguments $Q_1, \ldots, Q_k$ the head variable must be applied to, and the types of these arguments. The structure of each of $Q_1, \ldots, Q_k$ is then recursively determined by its type, with its own choice of head variable, which then recursively determines its subterms, and so on.

In other words, the degree of freedom in a long normal form is a choice of head variable at each level. This choice of head variables completely determines the long normal form.

Perhaps the preceding discussion can be made more comprehensible by means of some concrete examples. The examples take the form of the following propositions and their proofs.

**Proposition 8.8.** *Every closed term of type* **bool** *is $\beta\eta$-equivalent to either* $\mathbf{T}$ *or* $\mathbf{F}$.

*Proof.* Let $M$ be a closed term of type **bool**. By Proposition 8.7, we may assume that $M$ is a long normal form. Since **bool** $= \forall\alpha.\alpha \to \alpha \to \alpha$, every long normal form of this type must start, up to $\alpha$-equivalence, with

$$\Lambda\alpha.\lambda x^{\alpha}.\lambda y^{\alpha}.\ldots.$$

This must be followed by a head variable, which, since $M$ is closed, can only be $x$ or $y$. Since both $x$ and $y$ have atomic type, neither of them can be applied to further arguments, and therefore, the only two possible long normal forms are:

$$\begin{array}{l}\Lambda\alpha.\lambda x^{\alpha}.\lambda y^{\alpha}.x \\ \Lambda\alpha.\lambda x^{\alpha}.\lambda y^{\alpha}.y,\end{array}$$

which are $\mathbf{T}$ and $\mathbf{F}$, respectively. □

**Proposition 8.9.** *Every closed term of type* **nat** *is $\beta\eta$-equivalent to a Church numeral $\overline{n}$, for some $n \in \mathbb{N}$.*

*Proof.* Let $M$ be a closed term of type **nat**. By Proposition 8.7, we may assume that $M$ is a long normal form. Since **nat** $= \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$, every long normal form of this type must start, up to $\alpha$-equivalence, with

$$\Lambda\alpha.\lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.\ldots.$$

This must be followed by a head variable, which, since $M$ is closed, can only be $x$ or $f$. If the head variable is $x$, then it takes no argument, and we have

$$M = \Lambda\alpha.\lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.x$$

If the head variable is $f$, then it takes exactly one argument, so $M$ is of the form

$$M = \Lambda\alpha.\lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.fQ_1.$$

Because $Q_1$ has type $\alpha$, its own long normal form has no prefix; therefore $Q_1$ must start with a head variable, which must again be $x$ or $f$. If $Q_1 = x$, we have

$$M = \Lambda\alpha.\lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.fx.$$

If $Q_1$ has head variable $f$, then we have $Q_1 = fQ_2$, and proceeding in this manner, we find that $M$ has to be of the form

$$M = \Lambda\alpha.\lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.f(f(\ldots(fx)\ldots)),$$

i.e., a Church numeral. □

**Exercise 36.** Prove that every closed term of type $A \times B$ is $\beta\eta$-equivalent to a term of the form $\langle M, N\rangle$, where $M : A$ and $N : B$.

## 8.9 Application: representation of arbitrary data in System F

Let us consider the definition of a long normal form one more time. By definition, every long normal form is of the form

$$\Lambda a_1.\Lambda a_2\ldots\Lambda a_n.zQ_1Q_2\ldots Q_k, \tag{8}$$

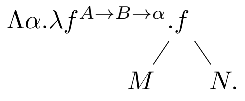
where $zQ_1Q_2\ldots Q_k$ has atomic type and $Q_1,\ldots,Q_k$ are, recursively, long normal forms. Instead of writing the long normal form on a single line as in (8), let us write it in tree form instead:

$$\begin{tikzcd}[column sep=small] & \Lambda a_1.\Lambda a_2\ldots\Lambda a_n.z \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] \arrow[drr, dash] & & \\ Q_1 & Q_2 & \cdots & Q_k, \end{tikzcd}$$

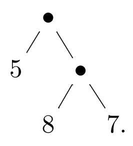
where the long normal forms $Q_1,\ldots,Q_k$ are recursively also written as trees. For example, with this notation, the Church numeral $\overline{2}$ becomes

$$\begin{tikzcd}[row sep=small] \Lambda\alpha.\lambda f^{\alpha\to\alpha}.\lambda x^{\alpha}.f \arrow[d, dash] \\ f \arrow[d, dash] \\ x, \end{tikzcd} \tag{9}$$

and the pair $\langle M, N\rangle$ becomes



We can use this very idea to encode (almost) arbitrary data structures. For example, suppose that the data structure we wish to encode is a binary tree whose leaves are labelled by natural numbers. Let's call such a thing a *leaf-labelled binary tree*. Here is an example:



(10)

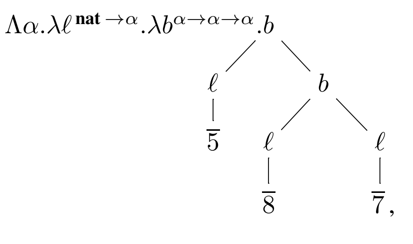
In general, every leaf-labelled binary tree is either a *leaf*, which is labelled by a natural number, or else a *branch* that has exactly two *children* (a left one and a right one), each of which is a leaf-labelled binary tree. Written as a BNF, we have the following grammar for leaf-labelled binary trees:

$$\text{Tree:}\quad T, S ::= \textbf{leaf}\,(n) \;\big|\; \textbf{branch}\,(T, S).$$

When translating this as a System F type, we think along the lines of long normal forms. We need a type variable $\alpha$ to represent leaf-labelled binary trees. We need two head variables whose type ends in $\alpha$: The first head variable, let's call it $\ell$, represents a leaf, and takes a single argument that is a natural number. Thus $\ell : \textbf{nat} \to \alpha$. The second head variable, let's call it $b$, represents a branch, and takes two arguments that are leaf-labelled binary trees. Thus $b : \alpha \to \alpha \to \alpha$. We end up with the following System F type:

$$\textbf{tree} = \forall\alpha.(\textbf{nat} \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha.$$

A typical long normal form of this type is:



$\Lambda\alpha.\lambda\ell^{\textbf{nat}\to\alpha}.\lambda b^{\alpha\to\alpha\to\alpha}.b$
$\ell$
$b$
$\overline{5}$
$\ell$
$\ell$
$\overline{8}$
$\overline{7},$

where $\overline{5}$, $\overline{7}$, and $\overline{8}$ denote Church numerals as in (9), here not expanded into long normal form for brevity. Notice how closely this long normal form follows (10). Here is the same term written on a single line:

$$\Lambda\alpha.\lambda\ell^{\mathbf{nat}\to\alpha}.\lambda b^{\alpha\to\alpha\to\alpha}.b(\ell\,\overline{5})(b(\ell\,\overline{8})(\ell\,\overline{7}))$$

**Exercise 37.** Prove that the closed long normal forms of type **tree** are in one-to-one correspondence with leaf-labelled binary trees.

## 9 Type inference

In Section 6, we introduced the simply-typed lambda calculus, and we discussed what it means for a term to be well-typed. We have also asked the question, for a given term, whether it is typable or not.

In this section, we will discuss an algorithm that decides, given a term, whether it is typable or not, and if the answer is yes, it also outputs a type for the term. Such an algorithm is known as a *type inference algorithm.*

A weaker kind of algorithm is a *type checking algorithm.* A type checking algorithm takes as its input a term with full type annotations, as well as the types of any free variables, and it decides whether the term is well-typed or not. Thus, a type checking algorithm does not infer any types; the type must be given to it as an input and the algorithm merely checks whether the type is legal.

Many compilers of programming languages include a type checker, and programs that are not well-typed are typically refused. The compilers of some programming languages, such as ML or Haskell, go one step further and include a type inference algorithm. This allows programmers to write programs with no or very few type annotations, and the compiler will figure out the types automatically. This makes the programmer's life much easier, especially in the case of higher-order languages, where types such as $((A \to B) \to C) \to D$ are not uncommon and would be very cumbersome to write down. However, in the event that type inference *fails*, it is not always easy for the compiler to issue a meaningful error message that can help the human programmer fix the problem. Often, at least a basic understanding of how the type inference algorithm works is necessary for programmers to understand these error messages.