Programming in Higher-Order Typed Lambda-Calculi

Benjamin Pierce Scott Dietzen Spiro Michaylov

CMU-CS-89-111 2

March 10, 1989

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

(Also appears as ERGO-89-075)

Abstract

Typed λ-calculi have been objects of theoretical study for many years. Recently, it has been shown that all the inductively defined types (including numbers, booleans, lists, and trees, as well as more complex structures like typed terms and proofs) can be represented in higher-order λ-calculi with no built-in types or type constructors. This raises the possibility of designing practical programming languages based on pure typed λ-calculi.

This tutorial presents a hierarchy of increasingly powerful languages, beginning with Church's simply typed λ-calculus (F_1) and the second-order polymorphic λ-calculus of Girard and Reynolds, and culminating in a fragment of Girard's ω-order polymorphic λ-calculus (F_ω). Our focus throughout is on the unusual style of programming that arises in these languages, where all functions are total and the primary control construct is iteration rather than general recursion.

This work was supported in part by the Office of Naval Research and the Defense Advanced Research Projects Agency (DOD) under contract number N00014-84-K-0415. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. Government.

Contents

1 Introduction **3**

2 The Simply-typed λ-Calculus **6**

2.1 Definitions and Properties 6

2.2 Inductive Type Definitions 11

2.3 Programming with Iterators 15

3 The Second-order Polymorphic λ-Calculus **19**

3.1 Definitions 21

3.2 Representing F_1^i Inductive Type Definitions in F_2 23

3.3 The Power of F_2 31

4 The Third-order Polymorphic λ-Calculus **33**

4.1 Definition and Properties of F_3 33

4.2 Polymorphic Inductive Datatypes 35

4.3 Programming in F_3 36

5 F_3 as a Metalanguage **41**

5.1 A Simple Representation of F_1 Terms 41

5.2 A Complete Representation of F_1 Terms 43

5.3 Representation of F_2 45

5.4 Representing F_3 in F_4 46

5.5 Alternative Formulations of `Term` 47

6 The ω-order Polymorphic λ-Calculus **50**

6.1 Basic Definitions 50

6.2 Properties of F_ω 53

6.3 Types and Type Inference 54
6.4 F_ω as the Basis for a Programming Language 56

A Representing the Untyped λ-Calculus 59

B Symbols and Terminology 63

Chapter 1

Introduction

Typed λ-calculi have been objects of theoretical study for many years [10,20,18,53,52,51, *etc.*]. One of the earliest results in this area was the demonstration that a wide class of number-theoretic functions could be defined in the simply typed λ-calculus. The basic trick behind these results was a representation of natural numbers as typed terms (the so-called "Church numerals"):

```
0  ≡  λf:Nat→Nat. λx:Nat. x
1  ≡  λf:Nat→Nat. λx:Nat. f(x)
2  ≡  λf:Nat→Nat. λx:Nat. f(f(x))
⋮
```

More recently, Böhm and Berarducci [4], and independently Leivant [34], have shown that any set of inductively defined types[1] can be translated into a set of types in the polymorphic λ-calculus of Reynolds [53] and Girard [18,20]. For example, the standard inductive definition of the natural numbers

```
indtype Nat:* with
    zero: Nat
    succ: Nat→Nat
```

can be translated mechanically into the representation above. This technique makes it possible to define a host of commonly used data types—booleans, pairs, lists, trees, and so on—and to express functions over them, even though the pure polymorphic λ-calculus provides no built-in types whatsoever. (Steensgaard-Madsen [55] presents a similar idea.) Generalizing the technique to higher orders, Pfenning [45] has shown that inductively defined types with polymorphic constructors in the n^{th}-order λ-calculus can be translated into the pure $(n+1)^{th}$-order λ-calculus. This further expands the class of definable data structures to include, for example, representations of typed λ-terms and proofs in higher-order logic.

Variants of the second-order polymorphic λ-calculus have been used as the foundations of

[1]Or more technically, any heterogeneous term algebra.

a number of practical programming languages [7,24], but they are normally embellished with a number of built-in types and type constructors. In view of the recent work on representing data structures, it now makes sense to ask whether the *pure* higher-order typed λ-calculi might form a suitable basis of a practical language for program and proof manipulation [49].

The pure polymorphic λ-calculi all share the property that every reduction sequence terminates after a finite number of steps. This implies that only total functions are definable, and that the familiar control construct of general recursive definition is not available. Instead, functions must be expressed in terms of *primitive* recursion (or iteration) over inductively defined data structures. Our central purpose in this tutorial is to explore the unusual programming style that arises from these constraints.

Chapter 2 of the tutorial introduces the simply typed λ-calculus (called F_1 here), reviews some of its properties, and establishes basic notational conventions. This chapter also introduces the notion of an "inductively defined type" and shows how values of such types can be manipulated using a basic iteration construct. Chapter 3 introduces the polymorphic λ-calculus (F_2), and shows how inductive type definitions over F_1 can be translated into pure F_2. Chapter 4 introduces the third-order polymorphic λ-calculus (F_3) and shows how the techniques of the previous chapter can be generalized to allow inductive type definitions over F_2 to be translated into pure F_3. Chapter 5 uses the principles developed so far to experiment with metaprogramming in F_3—building data structures that can be used to represent and manipulate terms in F_1 and F_2. Chapter 6 completes the hierarchy of languages by discussing the definition and properties of the ω-order polymorphic λ-calculus (Girard's $F_\omega$without existential quantifiers). Appendix A presents an extended metaprogramming example—a representation of untyped λ-terms in F_2. Appendix B summarizes our notational conventions.

Most sections are supplemented with exercises. We strongly recommend that readers try working most of these, since it has been our experience that the only way to understand programs written in this style is to generate a fair number of them. The tutorial is intended to be self-contained, but the early sections will be easier for readers who are familiar with the basic concepts of the untyped λ-calculus [2,28], or have done some programming in a λ-calculus-based language like Scheme [1,50] or ML [23,26]. Some acquaintance with polymorphic type systems [8,27,52] will also be helpful. Technical details that may not be accessible to all of our readers are placed in footnotes.

This document grew out of discussions in the LEAP Working Group at CMU. It reflects the authors' state of understanding after only a few months of experience in the area, and hence falls lamentably short of a full treatment of any of the subjects it introduces. Furthermore, many the technical results that it presents are subjects of intense current research, which raises the possibility that our present perspective may turn out to be incorrect or misguided in any number of ways. Still, the document represents a significant expansion of our own knowledge, and we hope it will be a useful guide for other newcomers. We welcome corrections and suggestions for clarification.

The other members of the LEAP group—Ken Cline, Peter Lee, and Frank Pfenning—share credit for most of the constructions we describe. We are also indebted to Luca Cardelli, Bob Harper, John Mitchell, and John Reynolds for helpful discussions and technical guidance, and to Tim Freeman, Bob Harper, Nevin Heintze, Peter Lee, David Long, and Frank Pfenning for suggesting improvements to the text.

Chapter 2

The Simply-typed λ-Calculus

We begin by defining a simple programming language and studying some of its properties. We then extend this language with a powerful facility for defining data types inductively.

2.1 Definitions and Properties

The purest form of Church's simply-typed λ-calculus [10], which we call F_1, may be defined as follows:

Definition 2.1.1: The syntax of F_1 is given by the following inductively defined classes:

$$\begin{array}{lcl} \mathtt{T} & ::= & \mathtt{T}\rightarrow\mathtt{T}' \\ \mathtt{e} & ::= & \mathtt{x} \mid \lambda\mathtt{x}:\mathtt{T}.\,\mathtt{e} \mid \mathtt{e}\,\mathtt{e}' \end{array}$$

where T ranges over *types*, e ranges over *expressions* (also called *terms*), and x ranges over *variables*. An expression of the form λx:T. e is called a *λ-abstraction*; e e′ is an *application*.

The language of Definition 2.1.1 is theoretically important because it forms the base of an infinite sequerce of more and more powerful languages culminating in F_ω. But from a practical standpoint, there is a major problem. The equation defining T is an inductive definition with no base case (no constant types), so the set of types is empty. Furthermore, since λ-abstractions are typed, the set of λ-abstractions is empty. In order to do anything useful with F_1, we need to add some primitive types and terms. We do this first in an *ad hoc* way, adding a single constant type and some constant terms to F_1 to form a language we call F_1^+:

Definition 2.1.2: The syntax of F_1^+ is given by the following inductively defined classes:

$$\begin{array}{lcl} \mathtt{T} & ::= & \mathtt{Nat} \mid \mathtt{T}\rightarrow\mathtt{T}' \\ \mathtt{e} & ::= & \mathtt{zero} \mid \mathtt{succ} \mid \mathtt{iter} \mid \mathtt{x} \mid \lambda\mathtt{x}:\mathtt{T}.\,\mathtt{e} \mid \mathtt{e}\,\mathtt{e}' \end{array}$$

where T ranges over types, e ranges over expressions, and x ranges over variables.

A typical term of F_1^+ is

```
λn:Nat. succ (succ n)
```

which denotes the function that, given a number, increases it by two. The whole term is a λ-abstraction with *bound variable* n whose *body* (or *scope*) consists of the two nested applications `(succ (succ n))`.

The rest of this section uses F_1^+ to review some notations and conventions of the λ-calculus. (Hindley and Seldin [28] provide a more thorough introduction to the basics of both typed and untyped λ-calculi. Barendregt [2] is an excellent reference on the untyped λ-calculus.) Definitions 2.1.1 and 2.1.2 specify the *abstract syntax* of F_1 and F_1^+; we deal with questions of concrete syntax and parsing (e.g., precedence and associativity) informally. Following standard practice, the → symbol associates to the right and application associates to the left. Parentheses are used when necessary to override these conventions. The body of a λ-abstraction extends as far to the right as possible—to the end of the whole expression, or up to an unmatched right parenthesis.

An occurrence of a variable x is *bound* if it appears in the scope of a λ-abstraction with bound variable x, and *free* otherwise. A *closed* λ-term is one with no free variables.

The abstract syntax given in Definitions 2.1.1 and 2.1.2 allows us to write meaningless programs like `(succ succ)`. We focus our attention only on the *well-typed* terms of each of the languages we define. We write the *type judgment*

$$\Pi \vdash e \in T$$

to indicate that an expression e has type T in the context of the type environment Π (which maps variables to types). To denote an explicit environment Π we write a list of ordered pairs enclosed in brackets '⟨' and '⟩', separated by commas. An empty environment is written as ⟨⟩ or is simply omitted. Although Π is potentially multiple-valued, we will think of it as single-valued, and we agree that it is searched from right to left to find the appropriate pair. We can think of Π as being extended to terms if we agree that

$$\Pi(e) = \alpha \quad \text{where} \quad \Pi \vdash e \in \alpha.$$

$\Pi[x/T]$ denotes the *extension* of the environment Π to x such that $\Pi(x) = T$. By convention pairs are always added to the right end of Π. In cases where e is a closed term or Π is obvious from context we write

$$e \in T.$$

The symbols ':' and '∈' have intuitively similar meanings, since both declare something to have a particular type. The difference between them is that ':' is part of the object language—it is used *within* terms to declare the types of bound variables—whereas '∈' is a notation of the metalanguage used to make statements (e.g., type judgments) *about* terms.

We formally define the typing of F_1^+ via *type inference* rules. Each of the following rules has the property that, if the premises are all valid type judgments, then the conclusion is a valid judgment. Within rule names the *I* stands for 'introduction' and the *E* for 'elimination.'

(Var)	$\dfrac{}{\Pi \vdash \mathtt{x} \in \mathtt{T}}$	when $\Pi(\mathtt{x}) = \mathtt{T}$
($\rightarrow$I)	$\dfrac{\Pi[\mathtt{x}/\mathtt{T}] \vdash \mathtt{e} \in \mathtt{T}'}{\Pi \vdash \lambda \mathtt{x}{:}\mathtt{T}.\mathtt{e} \in \mathtt{T} \rightarrow \mathtt{T}'}$	
($\rightarrow$E)	$\dfrac{\Pi \vdash \mathtt{e} \in \mathtt{T} \rightarrow \mathtt{T}' \quad \Pi \vdash \mathtt{e}' \in \mathtt{T}}{\Pi \vdash \mathtt{e}\ \mathtt{e}' \in \mathtt{T}'}$	
(Zero)	$\dfrac{}{\Pi \vdash \mathtt{zero} \in \mathtt{Nat}}$	
(Succ)	$\dfrac{}{\Pi \vdash \mathtt{succ} \in \mathtt{Nat} \rightarrow \mathtt{Nat}}$	
(Iter)	$\dfrac{\Pi \vdash \mathtt{n} \in \mathtt{Nat} \quad \Pi \vdash \mathtt{z} \in \mathtt{Nat} \quad \Pi \vdash \mathtt{s} \in \mathtt{Nat} \rightarrow \mathtt{Nat}}{\Pi \vdash \mathtt{iter\ n\ z\ s} \in \mathtt{Nat}}$	

To show how these deduction rules are used, we use them to prove that

$\lambda\mathtt{n{:}Nat.\ succ\ (succ\ n)} \in \mathtt{Nat} \rightarrow \mathtt{Nat}.$

(1)	$\langle(\mathtt{n,Nat})\rangle \vdash \mathtt{succ} \in \mathtt{Nat} \rightarrow \mathtt{Nat}$	by (Succ)
(2)	$\langle(\mathtt{n,Nat})\rangle \vdash \mathtt{n} \in \mathtt{Nat}$	by (Var)
(3)	$\langle(\mathtt{n,Nat})\rangle \vdash \mathtt{succ\ n} \in \mathtt{Nat}$	by ($\rightarrow$E) from 1,2
(4)	$\langle(\mathtt{n,Nat})\rangle \vdash \mathtt{succ\ (succ\ n)} \in \mathtt{Nat}$	by ($\rightarrow$E) from 1,3
(5)	$\vdash \lambda\mathtt{n{:}Nat.\ succ\ (succ\ n)} \in \mathtt{Nat} \rightarrow \mathtt{Nat}$	by ($\rightarrow$I) from 4

Exercises 2.1.3:

1. Prove that:

   $\lambda\mathtt{f{:}Nat{\rightarrow}Nat.\ \lambda a{:}Nat.\ f\ a} \in \mathtt{(Nat{\rightarrow}Nat){\rightarrow}Nat{\rightarrow}Nat}$

2. Prove that:

   $\lambda\mathtt{x{:}Nat.\ iter\ x\ (succ\ zero)\ (\lambda n{:}Nat.\ succ\ (succ\ n))} \in \mathtt{Nat{\rightarrow}Nat}$

We define the operational meaning of programs via *reduction rules.* A *β-redex* is a term t of the form:

$(\lambda\mathtt{x{:}T.e})\ \mathtt{a}$

It is *β-reduced* (or just *reduced*) according to the rule

$(\lambda\mathtt{x{:}T.e})\ \mathtt{a} \quad \Rightarrow_\beta \quad \mathtt{e}[\mathtt{x}/\mathtt{a}]$

where e[x/a] is the term obtained by replacing each free occurrence of x in e by a, and renaming any bound variables in e as necessary to prevent capture of free variables in a. An *η-redex* is a term of the form

$\lambda\mathtt{x{:}T.e\ x}$

(where x is not free in e). It is *η-reduced* (or *reduced*) according to the rule

$\lambda\mathtt{x{:}T.e\ x} \quad \Rightarrow_\eta \quad \mathtt{e}$

(when x is not free in e).

A term e is *one-step β-reducible to* e′ if e′ can be obtained from e by a single application of the rule for β-reduction to a subterm of e. A term e_0 is *β-reducible* to e_n (written $e_0 \Rightarrow_\beta e_n$) if there is a *reduction sequence* $e_0 \Rightarrow_\beta e_1 \Rightarrow_\beta e_2 \Rightarrow_\beta \ldots \Rightarrow_\beta e_n$ (with $n \geq 0$) where each element one-step β-reduces to the next. *One-step β-conversion* and *β-conversion* are defined similarly, but allow β-reduction to be applied in either direction. Two terms are *β-equivalent* (written $=_\beta$) if one can be β-converted to the other. The definitions of η-reduction, η-conversion, and η-equivalence are similar; the definitions of βη-reduction, βη-conversion, and βη-equivalence allow the two rules to be intermixed. We often write just = instead of $=_{\beta\eta}$. A term is in *βη-normal form* if it contains no β- or η-redexes.

Throughout this document we frequently blur the distinction between terms and their denotations. For example, we will speak of an expression like

```
        λx:Nat. succ (succ x)
```

as *being* the function that adds two to its argument, when, more properly, we should say that the expression *denotes* this function in some mathematical model we have in mind, or else that when applied to a term representing a number n, it *reduces* to a term representing $n + 2$.

Terms differing only in the names of bound variables are said to be *α-equivalent* . (The renaming of bound variables within a term is often called *α-conversion.*) Following standard practice [2,28], we consider α-equivalent terms to be identical.

F_1 and F_1^+ share a number of interesting theoretical properties with the other languages we consider in this tutorial. Perhaps the most important is the fact that only terminating computations can be expressed:

Definition 2.1.4: A term is *strongly normalizable* (under a given set of reduction rules) if every sequence of reductions beginning with that term reaches a normal form after a finite number of steps. A set of reduction rules is *strongly normalizing* if there is no infinite reduction sequence on any term.

Theorem 2.1.5: The rules given above are strongly normalizing (i.e., there is no infinite βη-reduction on any term).

The proof of this theorem for F_1 and F_1^+ is fairly straightforward (see [28], for example). Intuitively, η-reduction always decreases the size of a term, while β-reduction may increase its size but always decreases the nesting of arrows in the types of bound variables. The general proof for F_n is much more delicate [17,18,19,31].

The next theorem assures us that it does not matter which redexes in a term are reduced first:

Theorem 2.1.6: (Church-Rosser) For any well-typed term e, if e $\Rightarrow_{\beta\eta}$ e_1 and e $\Rightarrow_{\beta\eta}$ e_2 then there exists a term e′ such that $e_1 \Rightarrow_{\beta\eta}$ e′ and $e_2 \Rightarrow_{\beta\eta}$ e′.

Together, Theorems 2.1.5 and 2.1.6 guarantee that every well-typed term reduces in a finite number of steps to a unique normal form. This stands in sharp contrast to the untyped λ-calculus, where non-normalizable terms like

```
Ω  ≡  (λx. x x) (λx. x x)
```

are easy to construct [2].

The constant `iter` provides iteration over natural numbers, which can be used to implement all the ordinary primitive recursive functions on numbers. It takes three arguments: a number x to "iterate over," a number z to be returned in case x is zero, and a function f to be "iterated" in case x is nonzero. For example:

```
iter x (succ zero) (λr:Nat. succ (succ r))
```

The reduction rules for `iter` are as follows.

```
iter zero z e      ⇒ z
iter (succ x) z e ⇒ (e (iter x z e))
```

Strictly speaking, these should have been included in our discussion above of reduction, conversion, strong normalization, and so on. But since our main task in Chapter 3 is to show how inductive types (like `Nat`) and iteration over them can be eliminated from the core language, we prefer to discuss them separately. The reduction of an `iter` expression is guaranteed to terminate (i.e. strong normalization is maintained), since when the second case applies, the first argument to `iter` is reduced by one. Hence the first case must eventually apply. It is easy to see that the result will be x applications of e to z. More graphically, if x is

```
succ (succ (succ (... (succ zero) ...)))
```

then `iter x z e` has exactly the same structure as x, with the `zero` replaced by z and each `succ` replaced by e:

```
e (e (e (... (e z) ...)))
```

Returning to the example,

```
λx:Nat. iter x (succ zero) (λr:Nat. succ (succ r))
```

denotes a function that returns 2n+1 when applied to a number n.

Exercises 2.1.7:

1. Define a function that returns 3n when applied to n.
2. Define a function that returns one when its argument is zero, and zero otherwise.

Similarly, a function that adds two numbers can be defined using `iter`:

```
λx:Nat. λy:Nat.
      iter x y (λr:Nat. succ r)
```

or more simply (by η-conversion):

```
λx:Nat. λy:Nat. iter x y succ
```

Given x and y, the iter expression has the effect of taking x successors of y.

This example also illustrates the trick of *currying* multi-argument functions into single-argument functions. For example, instead of taking both of its arguments at once, the curried plus function accepts x and returns a function from y to x+y. In general, a function of n arguments of types $T_1, \cdots, T_n$ returning an answer of type T_0 has type:

$$T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_0$$

Exercises 2.1.8:

1. Define a function that sums three numbers.
2. Define a function that multiplies two numbers.

To make our programs more manageable, we allow both terms and types to be abbreviated by individual symbols. For example:

```
BinaryFun  ≡  Nat→Nat→Nat

plus       ∈  BinaryFun
plus       ≡  λx:Nat. λy:Nat. iter x y succ

times      ∈  BinaryFun
times      ≡  λx:Nat. λy:Nat. iter x 0 (λr:Nat. plus r y)
```

These abbreviations should be thought of as global macro definitions that can be completely expanded away without affecting the meaning of any term that mentions them. (They should *not* be thought of as global definitions in the sense of ML or Scheme. In particular, they may not be recursive.)

2.2 Inductive Type Definitions

We have seen that F_1^+ can express some useful programs. But it leaves something to be desired in the way of available data types. We chose the primitive types and terms somewhat arbitrarily and then enshrined this choice in the very definition of the language, with no provision for extending the available types short of defining an entirely new language. In this section we take a more general approach, adding to F_1 a general type definition facility instead of a particular set of predefined types. We use the keyword indtype to introduce an "inductively defined" type. The rest of the language remains as before.

Definition 2.2.1: The syntax of F_1^i is given by the following inductively defined classes:

$$\begin{array}{lcl} \mathtt{P} & ::= & \mathtt{I\ e} \mid \mathtt{I\ P} \\ \mathtt{I} & ::= & \mathtt{indtype}\ \alpha : * \mid \mathtt{indtype}\ \alpha : *\ \mathtt{with\ C} \\ \mathtt{C} & ::= & \mathtt{x} : \mathtt{T} \mid \mathtt{x} : \mathtt{T}\ \mathbf{and}\ \mathtt{C} \\ \mathtt{T} & ::= & \alpha \mid \mathtt{T} {\rightarrow} \mathtt{T}' \\ \mathtt{e} & ::= & \mathtt{x} \mid \lambda\mathtt{x} : \mathtt{T}.\, \mathtt{e} \mid \mathtt{e\ e}' \end{array}$$

where P ranges over programs, I ranges over (lists of) inductive type definitions, C ranges over (lists of) constructors, α ranges over type variables, T ranges over types, e ranges over expressions, and x ranges over variables.
An F_1^i program consists of a sequence of global indtype definitions, followed by an expression.

The type of natural numbers can be defined in F_1^i as follows:[1]

```
indtype Nat:* with zero:Nat and succ:Nat→ Nat
```

This is precisely what we had in F_1^+, except that the iter of F_1^+ becomes iterNat[Nat]:

```
plus ≡ λx:Nat. λy:Nat. iterNat[Nat] x y succ
```

Instead of a single iter function, F_1^i has, for each inductively defined type T, an infinite number of iteration functions—one called iterT[V] for every type V. (For now, the square brackets and the type V should be thought of as part of the name of the iterator.) Each iterT[V] performs structural induction on elements of type T, returning a value of type V as the result of the induction.

To illustrate how a different instance of the iteration scheme for Nat might be used, here is an alternative definition of addition:

```
plus  ≡  λx:Nat.
             iterNat[Nat→ Nat]
                x
                (λy:Nat. y)
                (λr:Nat→Nat. λy:Nat. succ (r y))
```

This version uses iterNat[Nat→Nat] to construct a function that applies succ x times to its argument (y).

An example of a complete program in F_1^i is:

```
indtype Nat:* with zero:Nat and succ:Nat→ Nat
plus  ≡  λx:Nat. λy:Nat. iterNat[Nat] x y succ
2  ≡  succ (succ zero)
plus 2 2
```

[1]The * can be ignored for now. It just indicates that we are defining a type. Later on we use indtype to define more complicated things as well.

The definitions of plus and 2 are simply abbreviations, as before. We have allowed ourselves to write these macro definitions at any convenient point in the program text. However, the indtype definitions at the beginning of a program actually define a *global environment* in which the body of the program is evaluated. This points out a major difference between symbols defined as global abbreviations and the types and constructors introduced by indtype definitions: the latter cannot be expanded away; indeed, they must appear in fully-normalized programs since there are no primitive datatypes.

Having looked at Nat, we are ready to define the general form of inductive type definitions and iterators.[2]

It is important to distinguish between *inductive* types and the more general class of *reflexive* types. It is not the case that every instance of the indtype syntax actually defines an inductive type. For example,

```
indtype T:* with
    c: (T→T)→T
```

is reflexive, but not inductive.

Formally, an inductive type definition is one where the type being defined appears only *positively* in the types of the arguments to the constructors. The notions of positive and negative occurrences may be formulated within a pair of mutually-recursive functions:[3]

Definition 2.2.2: The set $Pos(U)$ of positively occurring variables in an F_1^i type expression U is defined by:

$$\begin{array}{lcll} Pos(\alpha) & = & \{\alpha\} & \text{(where } \alpha \text{ is a type variable)} \\ Pos(V \to W) & = & Neg(V) \cup Pos(W) & \end{array}$$

The set $Neg(U)$ of negatively occurring variables in a type expression U is defined by:

$$\begin{array}{lcll} Neg(\alpha) & = & \{\} & \text{(where } \alpha \text{ is a type variable)} \\ Neg(V \to W) & = & Pos(V) \cup Neg(W) & \end{array}$$

A type variable α is said to ***appear positively*** in U if $\alpha \in Pos(U)$ and to ***appear negatively*** in U if $\alpha \in Neg(U)$.

The words "positive" and "negative" come from logic. According to the well-known "Curry-Howard isomorphism" [29,14, Section 9E] between propositions and types, the type $A \to B$ corresponds to the logical proposition $A \supset B$, which, by the definition of logical implication, is equivalent to $\neg A \vee B$. The subproposition A here is obviously in

[2]Any many-sorted first-order algebraic signature without laws (or "heterogeneous free algebra") can be considered as an inductively defined type [4].

[3]The technical intuition behind the definition is roughly as follows. A data type definition of the form we have described can be translated into a function on the lattice of types. If the definition has the form of an *inductive* type definition, then this function will be covariant, and hence (by Tarski's fixed point theorem) will be guaranteed to have both a least and a greatest fixed point.

a "negative" position—that is, inside of an odd number of negations—if and only if the whole implication appears inside an even number of negations.

With the notion of inductivity in hand, we can now complete the definition of F_1^i.

Definition 2.2.1, continued: The general form of an **indtype** definition is:

$$
\begin{array}{ll}
\textbf{indtype}\ \mathtt{T:*}\ \textbf{with} & \\
\quad \mathtt{c_1 : U_{11} \to U_{12} \to \cdots \to U_{1n_1} \to T} & \\
\textbf{and}\ \mathtt{c_2 : U_{21} \to U_{22} \to \cdots \to U_{2n_2} \to T} & \\
\quad \vdots & \\
\textbf{and}\ \mathtt{c_m : U_{m1} \to U_{m2} \to \cdots \to U_{mn_m} \to T} &
\end{array}
$$

Note that the type being defined must appear as the rightmost component of the type of each constructor, and may appear positively (but not negatively) in the $\mathtt{U}_{ij}$'s.

Each such definition introduces the following globally-bound identifiers:

1. The type T.
2. Zero or more constructors $\mathtt{c}_i$. Each $\mathtt{c}_i$ takes zero or more arguments (of types $\mathtt{U}_{i1} \ldots \mathtt{U}_{in_i}$, respectively), and produces a result of type T.
3. An *iteration scheme* iterT[α] with infinitely many instances iterT[V], called *iteration operators*—one for each type V. An iterT[V] takes one argument for each constructor of T (in order), and returns a result of type V. More formally, the type of iterT[V] is

$$
\begin{array}{lll}
\mathtt{iterT[V]} \in \mathtt{T} & \to & (\hat{\mathtt{U}}_{11} \to \hat{\mathtt{U}}_{12} \to \cdots \to \hat{\mathtt{U}}_{1n_1} \to \mathtt{V}) \\
 & \to & (\hat{\mathtt{U}}_{21} \to \hat{\mathtt{U}}_{22} \to \cdots \to \hat{\mathtt{U}}_{2n_2} \to \mathtt{V}) \\
 & & \quad \vdots \\
 & \to & (\hat{\mathtt{U}}_{m1} \to \hat{\mathtt{U}}_{m2} \to \cdots \to \hat{\mathtt{U}}_{mn_m} \to \mathtt{V}) \\
 & \to & \mathtt{V}
\end{array}
$$

where $\hat{\mathtt{U}}$ denotes the result of substituting V for all occurrences of T in U.

An iterator iterT[V] associates a function building up values of type V with each constructor for type T. It pulls apart a term of type T, constructor by constructor, and applies the function associated with that constructor to the arguments, but only after recursively applying iterT[V] within them on all subterms of type T. We can express this as one reduction rule:

$$
\begin{array}{l}
\mathtt{iterT[V]}\ (\mathtt{c}_i\ \mathtt{a}_1\ \cdots\ \mathtt{a}_{n_i})\ \mathtt{e}_1\ \cdots\ \mathtt{e}_m \\
\qquad\qquad \Rightarrow \quad \mathtt{e}_i\ \hat{\mathtt{a}}_1\ \cdots\ \hat{\mathtt{a}}_{n_i}
\end{array}
$$

where $\mathtt{c}_i$ is the i^{th} constructor for type T, $\mathtt{a}_1$ to $\mathtt{a}_{n_i}$ are its arguments, and $\mathtt{e}_i \in \hat{\mathtt{U}}_{i1} \to \cdots \to \hat{\mathtt{U}}_{in_i} \to \mathtt{V}$ is the function corresponding to this constructor, and where $\hat{\mathtt{a}}$ is the result of replacing, in a, each subterm t of type T with iterT[V] t $\mathtt{e}_1 \cdots \mathtt{e}_m$.

2.3 Programming with Iterators

Of course, primitive data types other than Nat can also be defined inductively. The type Bool has a particularly simple form, where all of the constructors are constants:

```
        indtype Bool:* with true:Bool and false:Bool
```

Using iterBool[Nat], we can define an "if...then...else" construct for choosing between two numbers on the basis of some test:

```
        ife_Nat  ≡  λb:Bool. λt:Nat. λe:Nat.
                        iterBool[Nat] b t e
```

(Of course, it is equally easy to define if...then...else constructs for selecting between values of other types. However, a separate definition is required for each one because at this stage we don't have any way to parameterize functions with respect to types. Consequently, we use this ***underscore notation*** to include the type being returned is part of the *name* of the iterator.)

Exercise 2.3.1: Use iterBool[Bool] to define the binary and function.

An even simpler data type is Unit, which has just one constructor:

```
        indtype Unit:* with unit:Unit
```

Unit is often used in statically-typed languages with imperative constructs (e.g., Standard ML [23]) as the result type of functions that are executed purely for their side effects.

Continuing in the same vein, there is one even simpler inductive type, called Void, which has no constructors at all:

```
        indtype Void:*
```

(Obviously, no term in F_1^i can ever have type Void, but that does not prevent us from defining it.)

Exercise 2.3.2: Define a type Day with constant constructors sunday, monday, ..., saturday. Write a function weekday ∈ Day→Bool that returns true if its argument is in the range monday...friday.

A variety of other useful types can be defined if we expand our horizons to include more complicated inductive definitions. Additionally, it is often important to define *destructors* or *projection functions*—functions that take a term built up using constructors, and pull it apart.

For example, here is the type of pairs of numbers:

```
        indtype Pair_Nat:* with pair_Nat:Nat→Nat→Pair_Nat
```

The projection functions for Pair_Nat are easy to define by iteration:

```
        fst_Nat  ≡  λp:Pair_Nat. iterPair_Nat[Nat] p (λf:Nat. λs:Nat. f)
        snd_Nat  ≡  λp:Pair_Nat. iterPair_Nat[Nat] p (λf:Nat. λs:Nat. s)
```

Similarly, the type of finite lists of numbers is defined by:

```
        indtype List_Nat:* with
                    nil_Nat: List_Nat
                    cons_Nat: Nat→List_Nat→List_Nat
```

All the usual list manipulation functions can be defined on this representation. For example:

```
        car_Nat  ≡  λl:List_Nat. λd:Nat.
                      iterList_Nat[Nat] l d (λc:Nat. λr:Nat. c)
```

The "default" parameter d is needed so that car will have something to return if it happens to be passed an empty list.

Exercises 2.3.3:

1. Check carefully that the types of the arguments to the iterations in fst_Nat and car_Nat correspond to Definition 2.2.1.
2. What is the purpose of the parameter r in the inner λ-abstraction above? Why does it have type Nat?
3. Define an append_Nat function that takes two lists of numbers and returns their concatenation.
4. Define a function map_Nat ∈ List_Nat→(Nat→Nat)→List_Nat that takes a list l of numbers and a numeric function f, and returns the list formed by applying f to each element of l.

Somewhat unexpectedly, the definitior of cdr_Nat turns out to be quite a bit more complicated than car_Nat.

Exercise 2.3.4: Readers are encouraged to pause here before reading further and try to see why this is so. What goes wrong with a simple definition of cdr_Nat in terms of iterList_Nat[List_Nat]? Is there a way to fix it?

The solution introduces a very important trick (important enough to be promoted to a "technique"), which we will use again and again in the rest of the tutorial. It was first used by Kleene [9] to define a predecessor function on the Church Numerals, which are essentially the same as our inductive type Nat. In general terms, it allows us to express every primitive recursive function on an inductively defined type in terms of iteration and pairing.

First, we define a type of pairs of lists of numbers:

```
        indtype Pair_List_Nat:* with
                    pair_List_Nat: List_Nat→List_Nat→Pair_List_Nat
```

(with projection functions `fst_List_Nat` and `snd_List_Nat` as above).

The method of iterative definition forces us to start at the end of the list and build our result backwards. At each successive `cons`, it is not enough to know the `cdr` of the second argument to the `cons`: we need the second argument *itself*. But when we finish and return the final result, it is not enough to have built up a new copy of the list itself: this time we want the `cdr`. The trick is to maintain both pieces of information in parallel. We shall iterate over a `l` ∈ `List_Nat` producing successive pairs of `List_Nat`s. The first element of each pair is the `cdr_Nat` of `l`, while the second is `l` itself. A new pair `p'` is computed from the old `p` by pairing `(snd_List_Nat p)` and `(cons_Nat c (snd_List_Nat p))`. We will see in Section 3.2 that this corresponds in a fairly natural way to defining a function by primitive recursion.

```
        cdr_Nat  ≡  λl:List_Nat.
                     fst_List_Nat
                        (iterList_Nat[Pair_List_Nat] l
                            (pair_List_Nat nil_Nat nil_Nat)
                            (λc:Nat. λr:Pair_List_Nat.
                                  pair_List_Nat
                                     (snd_List_Nat r)
                                     (cons_Nat c (snd_List_Nat r))))
```

Exercises 2.3.5:

1. What is `(cdr_Nat nil_Nat)`?
2. Define a predecessor function on natural numbers.
3. Define an inductive type of binary trees with natural numbers as leaves. Write a function that sums the leaves of a tree. Write a function that extracts the right subtree of a tree.
4. Define a function that, given `n` ∈ `Nat`, computes the `n`th Fibonacci number.

By considering constructors that take functions as arguments, we can expand the space of definable data types still further. For example, here is the type of arbitrarily branching finite trees (that is, trees in which each node may have any finite number of children):

```
        indtype Tree:* with
            empty: Tree
            node: Nat → (Nat→Tree) → Tree
```

A tree may be totally empty, or may consist of a node and n subtrees. In the latter case, the tree is constructed by specifying n along with a function mapping each natural number $0 \leq i \leq n$ to the i^{th} subtree.

To provide an easier way of constructing trees, we can define a function **build** that takes a list of trees and constructs a new tree with these trees as children:

```
build  ≡  λl:List_Tree.
              node
                (length l)
                (λn:Nat. nth l n empty)
```

where nth is the function that takes a List_Tree l, a number n, and a default Tree to be returned in case l has less than n elements.

Exercises 2.3.6:

1. Define nth.
2. Write a function that counts the number of nodes in a Tree.
3. Extend the definition of Tree to include a numerical value at each leaf. Write a function that flattens a tree into a list of the values encountered during a depth-first left-to-right traversal.

The reader may wonder why have we used the rather complicated notion of iterators rather than just adding a **case** construct (*a la* ML or Pascal) to the language. The reason is that **case** is only usable for expressing computations over inductive data types when the language *also* has a construct for defining recursive functions (sometimes called letrec or labels). But such a construct would destroy the important property that all F_1 programs are strongly normalizing. Iteration, on the other hand, preserves this property. Primitive recursion, which also preserves strong normalization, could have been built into the language instead of iteration. But since primitive recursion can be defined in terms of iteration, we prefer to avoid the extra complication.

Chapter 3

The Second-order Polymorphic λ-Calculus

Consider the following simple F_1^+ functions:

`id_Nat`	$\equiv$	λ`x:Nat. x`
`double_Nat`	$\equiv$	λ`f:Nat`$\rightarrow$`Nat.` λ`x:Nat. f(f x)`

The first of these denotes the identity function on numbers, while the second takes a function on numbers and applies it twice. These functions would make perfect sense with `Bool`, `Pair_Nat`, or indeed any type whatsoever in place of `Nat`. Unfortunately, to express the same operations on `Bool`s, another pair of ***essentially identical*** functions must be written. Suppose instead we wanted to define the identity and doubling functions once for all types. We could start by replacing `Nat` by a variable, say α,

`id_`α	$\equiv$	λ`x:`α`. x`
`double_`α	$\equiv$	λ`f:`$\alpha\rightarrow\alpha$`.` λ`x:`α`. f(f x)`

giving two term schemas, each with an infinite number of instances. Now, in each situation where `id_`α or `double_`α is used, α is replaced by some actual type `T`. Instead we may extend the language so that α is explicitly abstracted; for example, we think of `id` as a ***function*** from types `T` to terms `id_T`. To remind ourselves that the argument to this function is a type rather than a term, we write the abstraction operator with a capital Λ instead of the usual λ:

`id`	$\equiv$	$\Lambda\alpha$`:*.` λ`x:`α`. x`
`double`	$\equiv$	$\Lambda\alpha$`:*.` λ`f:`$\alpha\rightarrow\alpha$`.` λ`x:`α`. f(f x)`

When we want to apply the identity function to an actual (term) argument, we must first ***instantiate*** it to one of its instances by supplying a type argument. Again, to remind ourselves that this is a different sort of application than before, we enclose type arguments in square brackets:

`id [Nat]`	$\Rightarrow$	`id_Nat`
`id [Nat] 5`	$\Rightarrow$	`5`
`double [Bool]`	$\Rightarrow$	`double_Bool`

```
double [Bool] not true ⇒ true
```

We say that functions taking type arguments in this manner are *polymorphic* in that they may be applied to terms of differing type.

We have extended the language of terms to include polymorphic functions, but have not considered the corresponding extension of the type language so that such terms may be given types. What, then, is the *type* of a polymorphic function? It is something like an $\rightarrow$ type, since it is a kind of function. But again, since it takes a type and returns a term, we want a different notation from $\rightarrow$. Also, the type of the result returned by such a function can vary based upon the argument given to it; thus we need an explicit way of indicating this dependence. We introduce a new symbol Δ that, like λ and Λ, binds a variable in the scope of another expression (but is used for describing types instead of terms). Now id, which takes a type α and returns a function from α to α, is said to have type $\Delta\alpha.\alpha\rightarrow\alpha$:

id	$\in$	$\Delta\alpha.\alpha\rightarrow\alpha$
id [Nat]	$\in$	Nat→Nat
double	$\in$	$\Delta\alpha.(\alpha\rightarrow\alpha)\rightarrow\alpha\rightarrow\alpha$
double [Bool]	$\in$	(Bool→Bool)→Bool→Bool

The language we are introducing, generally called the second-order polymorphic λ-calculus (or often just the polymorphic λ-calculus, since many authors do not consider related languages of order higher than two), is *explicitly* rather than *implicitly* polymorphic. In languages with explicit polymorphism, type quantifiers like Δ actually appear in type expressions and correspond to actual type abstractions with Λ. A polymorphic function must be applied explicitly to a type argument to give a monomorphic instance, which can then be applied to term arguments. On the other hand, implicitly polymorphic languages (notably ML [23]) generally omit types from the concrete syntax. Implicitly polymorphic functions may be applied directly to terms of different types; the task of determining the intended monomorphic instance is left to the interpreter/compiler. This topic is explored more fully in Section 6.3.

The polymorphic λ-calculus was invented by Girard in 1971 [20,18] and independently reinvented by Reynolds in 1974 [53]. Girard, a logician, was trying to extend the well-known Curry-Howard isomorphism between propositions and types [29,14, Section 9E], by regarding the binding operator $\Delta\alpha$ as a universal quantifier ranging over propositions. Reynolds, a computer scientist, developed essentially the same system by formalizing the idea of "passing types as parameters" in a programming language. Second-order type systems have been the object of much recent research. Reynolds [52,51] and Cardelli and Wegner [8] have written excellent introductions to the area. (Further readings are cited in the bibliographies of these papers.)

3.1 Definitions

We now proceed to a formal definition of F_2.

Definition 3.1.1: The syntax of F_2 (the second-order polymorphic λ-calculus) is given by the following inductively defined classes:

$$\begin{array}{lcl} \mathtt{T} & ::= & \alpha \mid \mathtt{T} \rightarrow \mathtt{T}' \mid \Delta\alpha : *.\mathtt{T} \\ \mathtt{e} & ::= & \mathtt{x} \mid \lambda \mathtt{x}:\mathtt{T}.\mathtt{e} \mid \mathtt{e}\,\mathtt{e}' \mid \Lambda\alpha:*.\mathtt{e} \mid \mathtt{e}\,[\mathtt{T}] \end{array}$$

where T ranges over types, α ranges over type variables, e ranges over expressions, and x ranges over variables.

As before, λ is used to construct functions that can be applied to a term, yielding a term—i.e., ***term abstractions***—whereas Λ constructs functions that can be applied to a type, yielding a term—i.e., ***type abstractions***. The $\rightarrow$ is used to represent the type of a term abstraction (λ), while Δ forms the type of a type abstraction (Λ). In addition to applying terms to other terms (***term application***), F_2 allows the application of terms to types with the syntax $\mathtt{e}\,[\alpha]$ (***type application***). As before $\gamma{:}*$ simply indicates that γ is a type.

The concrete syntax of F_2 follows the same conventions as F_1. The $\rightarrow$ symbol associates to the right, and application (of both terms and types) associates to the left. The Δ and Λ operators behave like λ in that their bodies extend as far as possible to the right—to the end of the whole expression, or up to an unmatched right parenthesis.

Type variables are ***free*** or ***bound*** in the same sense as the term variables of F_1. A ***closed*** F_2 term contains no free term- or type variables. A closed type expression contains no free type variables.

As in the original definition of F_1, there are no constant types or terms in F_2. However, unlike pure F_1, the sets of types and terms of pure F_2 are inhabited (i.e., non-empty)

The F_1 notions of α-conversion (renaming of bound variables) and $\beta\eta$-reduction and conversion are extended to include type abstraction over terms and term application:

$$\begin{array}{lcll} (\Lambda\alpha:*.\ \mathtt{e})\ [\mathtt{T}] & \Rightarrow_\beta & \mathtt{e}[\alpha/\mathtt{T}] & \\ (\Lambda\alpha:*.\ \mathtt{e}\ [\alpha]) & \Rightarrow_\eta & \mathtt{e} & \text{(provided } \alpha \text{ is not free in } \mathtt{e}) \end{array}$$

The appropriate analogues of Theorems 2.1.5 to 2.1.6 also hold for F_2 (though some of the proofs are significantly more difficult). In particular, every F_2 term is strongly normalizing.

We may define the typing of F_2 formally by extending the type inference rules of F_1. The first three rules (ENV-$\langle\rangle$), (ENV-term) and (ENV-type) deal with the well-formedness of environments, and *wf*(Π) is used to say that the environment Π is well-formed. The base case (Tvar) deals with type variables. The two rules (WF-$\rightarrow$) and (WF-Δ) also deal with the well-formedness of types, and this is how they get their names. The rule (Var) looks up the type of a variable in the current environment and checks that it is well formed.

The remaining four rules ($\rightarrow$I), ($\rightarrow$E), (ΔI) and (ΔE) also deal with the correct typing of terms, and are named according to whether the symbols $\rightarrow$ and Δ are introduced or eliminated at the type level by that rule. The notation $\mathtt{T'}[\alpha/\mathtt{T}]$ is the result of replacing all occurrences of the type variable α in $\mathtt{T'}$ with $\mathtt{T}$, while renaming bound variables in $\mathtt{T'}$ to avoid capture. We take "α is not free in Π" to mean that α is not free in any type expression assigned by Π.

(ENV-$\langle\rangle$)	$\dfrac{}{wf(\langle\rangle)}$	
(ENV-term)	$\dfrac{\Pi \vdash \mathtt{T} \in *}{wf(\Pi[\mathtt{x}/\mathtt{T}])}$	
(ENV-type)	$\dfrac{wf(\Pi)}{wf(\Pi[\alpha/*])}$	when α is not free in Π
(Tvar)	$\dfrac{wf(\Pi)}{\Pi \vdash \alpha \in *}$	when $\Pi(\alpha) = *$
(WF-$\rightarrow$)	$\dfrac{\Pi \vdash \mathtt{T} \in * \qquad \Pi \vdash \mathtt{T'} \in *}{\Pi \vdash \mathtt{T}\rightarrow\mathtt{T'} \in *}$	
(WF-Δ)	$\dfrac{\Pi[\alpha/*] \vdash \mathtt{T} \in *}{\Pi \vdash \Delta\alpha{:}*.\mathtt{T} \in *}$	when α is not free in Π
(Var)	$\dfrac{\Pi \vdash \mathtt{T} \in *}{\Pi \vdash \mathtt{x} \in \mathtt{T}}$	when $\Pi(\mathtt{x}) = \mathtt{T}$
($\rightarrow$I)	$\dfrac{\Pi \vdash \mathtt{T} \in * \qquad \Pi[\mathtt{x}/\mathtt{T}] \vdash \mathtt{e} \in \mathtt{T'}}{\Pi \vdash \lambda\mathtt{x}{:}\mathtt{T}.\mathtt{e} \in \mathtt{T}\rightarrow\mathtt{T'}}$	
($\rightarrow$E)	$\dfrac{\Pi \vdash \mathtt{e} \in \mathtt{T}\rightarrow\mathtt{T'} \qquad \Pi \vdash \mathtt{e'} \in \mathtt{T}}{\Pi \vdash \mathtt{e}\,\mathtt{e'} \in \mathtt{T'}}$	
(ΔI)	$\dfrac{\Pi[\alpha/*] \vdash \mathtt{e} \in \mathtt{T}}{\Pi \vdash \Lambda\alpha{:}*.\mathtt{e} \in \Delta\alpha{:}*.\mathtt{T}}$	when α is not free in Π
(ΔE)	$\dfrac{\Pi \vdash \mathtt{e} \in \Delta\alpha{:}*.\mathtt{T'} \qquad \Pi \vdash \mathtt{T} \in *}{\Pi \vdash \mathtt{e}[\mathtt{T}] \in \mathtt{T'}[\alpha/\mathtt{T}]}$	

We give an example of how to use these rules by proving that

$$\mathtt{double\ [Nat]\ succ\ zero} \quad \in \quad \mathtt{Nat}$$

where

$$\mathtt{double} \quad \equiv \quad \Lambda\alpha{:}*.\ \lambda\mathtt{f}{:}\alpha\rightarrow\alpha.\ \lambda\mathtt{x}{:}\alpha.\ \mathtt{f(f\ x)}$$

Note that in this example and the following exercise, we need to make some assumptions about the types Nat and Bool, since these are not included in F_2. In particular, we assume that in all environments we have $\mathtt{Nat} \in *$, $\mathtt{zero} \in \mathtt{Nat}$, $\mathtt{succ} \in \mathtt{Nat}\rightarrow\mathtt{Nat}$, $\mathtt{Bool} \in *$, $\mathtt{true} \in \mathtt{Bool}$ and $\mathtt{not} \in \mathtt{Bool}\rightarrow\mathtt{Bool}$. Later we will see how they can be defined, but for now we will treat them as if they were built in.

(1)	$wf(\ \langle\rangle\ )$	by (ENV-$\langle\rangle$)
(2)	$wf(\ \langle(\alpha,*)\rangle\ )$	by (ENV-type) from 1
(3)	$\langle(\alpha,*)\rangle \vdash \alpha \in *$	by (Tvar) from 2
(4)	$\langle(\alpha,*)\rangle \vdash \alpha\rightarrow\alpha \in *$	by (WF-$\rightarrow$) from 3,3
(5)	$wf(\ \langle(\alpha,*),(\mathtt{f},\alpha\rightarrow\alpha)\rangle\ )$	by (ENV-term) from 4
(6)	$\langle(\alpha,*),(\mathtt{f},\alpha\rightarrow\alpha)\rangle \vdash \alpha \in *$	by (Tvar) from 5
(7)	$wf(\ \langle(\alpha,*),(\mathtt{f},\alpha\rightarrow\alpha),(\mathtt{X},\alpha)\rangle\ )$	by (ENV-term) from 6
(8)	$\langle(\alpha,*),(\mathtt{f},\alpha\rightarrow\alpha),(\mathtt{x},\alpha)\rangle \vdash \alpha \in *$	by (Tvar) from 7

(9)	$\langle(\alpha,*),(\mathtt{f},\alpha\to\alpha),(\mathtt{x},\alpha)\rangle \vdash \mathtt{x} \in \alpha$	by (Var) from 8
(10)	$\langle(\alpha,*),(\mathtt{f},\alpha\to\alpha),(\mathtt{x},\alpha)\rangle \vdash \alpha\to\alpha \in *$	by (WF-$\to$) from 8,8
(11)	$\langle(\alpha,*),(\mathtt{f},\alpha\to\alpha),(\mathtt{x},\alpha)\rangle \vdash \mathtt{f} \in \alpha\to\alpha$	by (Var) from 10
(12)	$\langle(\alpha,*),(\mathtt{f},\alpha\to\alpha),(\mathtt{x},\alpha)\rangle \vdash \mathtt{f\ x} \in \alpha$	by ($\to$E) from 11,9
(13)	$\langle(\alpha,*),(\mathtt{f},\alpha\to\alpha),(\mathtt{x},\alpha)\rangle \vdash \mathtt{f\ (f\ x)} \in \alpha$	by ($\to$E) from 11,12
(14)	$\langle(\alpha,*),(\mathtt{f},\alpha\to\alpha)\rangle \vdash \lambda\mathtt{x}{:}\alpha.\mathtt{f\ (f\ x)} \in \alpha\to\alpha$	by ($\to$I) from 6,13
(15)	$\langle(\alpha,*)\rangle \vdash \lambda\mathtt{f}{:}\alpha\to\alpha.\lambda\mathtt{x}{:}\alpha.\mathtt{f\ (f\ x)} \in (\alpha\to\alpha)\to\alpha\to\alpha$	by ($\to$I) from 4,14
(16)	$\vdash \Lambda\alpha{:}*.\lambda\mathtt{f}{:}\alpha\to\alpha.\lambda\mathtt{x}{:}\alpha.\mathtt{f\ (f\ x)} \in \Delta\alpha.(\alpha\to\alpha)\to\alpha\to\alpha$	by (ΔI) from 15
(17)	$\vdash \mathtt{double} \in \Delta\alpha.(\alpha\to\alpha)\to\alpha\to\alpha$	by definition from 16
(18)	$\vdash \mathtt{Nat} \in *$	by assumption
(19)	$\vdash \mathtt{succ} \in \mathtt{Nat}\to\mathtt{Nat}$	by assumption
(20)	$\vdash \mathtt{double\ [Nat]} \in (\mathtt{Nat}\to\mathtt{Nat})\to\mathtt{Nat}\to\mathtt{Nat}$	by (ΔE) from 17,18
(21)	$\vdash \mathtt{double\ [Nat]\ succ} \in \mathtt{Nat}\to\mathtt{Nat}$	by ($\to$E) from 20,19
(22)	$\vdash \mathtt{zero} \in \mathtt{Nat}$	by assumption
(23)	$\vdash \mathtt{double\ [Nat]\ succ\ zero} \in \mathtt{Nat}$	by ($\to$E) from 21,22

Exercise 3.1.2: Use the type deduction rules for F_2 to prove that:
`double [Bool] not true` $\in$ `Bool`

3.2 Representing F_1^{i} Inductive Type Definitions in F_2

At this point, the reader may be expecting us to reintroduce the `indtype` mechanism for defining primitive types and constructors, producing F_2^{i}. But we need not take this step yet. It is a surprising fact that we can translate any inductive type definition in F_1^{i} into a representation in F_2. More specifically, each type `T` introduced by an `indtype` definition in F_1^{i} can be expressed as a closed type expression of F_2, and each of the constructors of `T` can be expressed as a closed term in F_2.

The formal details of the translation are quite technical (see [3,45,49]), but it is relatively easy to understand operationally.

Let us begin with the simplest example from F_1^{i}. The type `Void`, with no constructors, was defined by:

```
indtype Void:*
```

Without explaining, for the moment, how we arrive at it, let the F_2 type corresponding to this definition be:

$$\mathtt{Void} \equiv \Delta\gamma{:}*.\ \gamma$$

To justify the assertion that this type "represents" the `indtype Void`, we show that there are no closed F_2 terms of type `Void`. If there is any term of type `Void`, then there is one in normal-form. Since `Void` begins with a Δ, any closed normal-form term of this type must be a Λ-abstraction (this follows from the typing rules):

$$\Lambda\gamma{:}*.\ \boxed{?}$$

where the omitted subterm $\boxed{?}$ has type γ. Again, the normal-form theorem tells us that if there is any term that can take the place of $\boxed{?}$, then there is one in normal-form. Hence, there are three possibilities to consider:

1. $\boxed{?}$ is a λ-abstraction. But then the type of $\boxed{?}$ would be an arrow type ($\alpha \rightarrow \beta$, for some α and β), whereas γ is a type variable.
2. $\boxed{?}$ is a Λ-abstraction. But the type of $\boxed{?}$ would then begin with a Δ, while$\gamma$is a type variable.
3. $\boxed{?}$ is a variable. This variable would clearly have to be of type γ. But the whole term must be closed, so we cannot use a free variable of type γ, nor does the $\boxed{?}$ appear in the scope of any bound variable of type γ.

Thus there are no normal-form terms—and hence no terms at all—of type `Void`.

Again on faith, we render the F_1^{i} definition

```
indtype Unit:* with unit:Unit
```

as the F_2 type

$$\texttt{Unit} \equiv \Delta\gamma\texttt{:*}.\ \gamma \rightarrow \gamma$$

which has exactly one normal-form instance (corresponding to the single constant constructor `unit` $\in$ `Unit`). As before, any closed normal-form term of type `Unit` must begin with a Λ-abstraction. Then, since the Δ in the type is followed by an $\rightarrow$ (and there are no variables with arrow type available), the Λ-abstraction in the term must be followed by a λ-abstraction:

$$\Lambda\gamma\texttt{:*}.\ \lambda\texttt{u:}\gamma.\ \boxed{?}$$

Again, $\boxed{?}$ must be of type γ. Reasoning as before, we find that this time there *is* a normal-form term of type γ available to fill the place of $\boxed{?}$:

$$\texttt{unit} \equiv \Lambda\gamma\texttt{:*}.\ \lambda\texttt{u:}\gamma.\ \texttt{u}$$

Turning our attention to br oleans, we render

```
indtype Bool:* with true:Bool and false:Bool
```

as

$$\texttt{Bool} \equiv \Delta\gamma\texttt{:*}.\ \gamma\rightarrow\gamma\rightarrow\gamma$$

with the two possible normal-form terms of this type being these:

$$\texttt{true} \equiv \Lambda\gamma\texttt{:*}.\ \lambda\texttt{t:}\gamma.\ \lambda\texttt{f:}\gamma.\ \texttt{t}$$
$$\texttt{false} \equiv \Lambda\gamma\texttt{:*}.\ \lambda\texttt{t:}\gamma.\ \lambda\texttt{f:}\gamma.\ \texttt{f}$$

Let us consider the definition of the boolean negation function directly in F_2:

$$\begin{aligned} \texttt{not} &\in \texttt{Bool}\rightarrow\texttt{Bool} \\ \texttt{not} &\equiv \lambda\texttt{b:Bool}.\ \Lambda\gamma'{:}*.\ \lambda\texttt{t}'{:}\gamma'.\ \lambda\texttt{f}'{:}\gamma'.\ \texttt{b}[\gamma']\texttt{f}'\texttt{t}' \end{aligned}$$

The reader may feel somewhat overwhelmed by the syntax of the above expression, so let us attempt to piece together an understanding in stages. The function not takes a boolean argument and produces a boolean result, so it must clearly have the form

$$\lambda\texttt{b:Bool}.\ (\Lambda\gamma'{:}*.\ \lambda\texttt{t}'{:}\gamma'.\ \lambda\texttt{f}'{:}\gamma'.\ \boxed{?})$$

where $\boxed{?}$ has type γ'. The bound variables $\texttt{t}'$ and $\texttt{f}'$ are both of type γ', but we need to select between them on the basis of b's truth or falsity. The boolean b can be used (as an iterator) to do precisely this; that is, the boolean term b is represented by a function, and can therefore be applied to other terms. Since b is either the function `true` or the function **false**, it will select either its first or second argument, respectively. The above definition swaps the arguments to b, producing the negation of b.

It is often less than obvious that such a function behaves as expected. For example, not has a large number of arguments and seems to return an expression of type γ', rather than type Bool. The key is to remember that all functions are curried, and that when not is applied to an argument, only the first λ will be β-reduced. The arguments γ', $\texttt{t}'$ and $\texttt{f}'$ will not be instantiated. Rather, it is the t and f arguments of b that will be replaced by the outer $\texttt{t}'$ and $\texttt{f}'$. Let us go through and check that not is in fact correctly typed:

$$\begin{aligned} \texttt{not} &\in (\Delta\gamma{:}*.\ \gamma\rightarrow\gamma\rightarrow\gamma)\rightarrow\Delta\gamma'.\ \gamma'\rightarrow\gamma'\rightarrow\boxed{?} \\ &\in (\Delta\gamma.\ \gamma\rightarrow\gamma\rightarrow\gamma)\rightarrow\Delta\gamma'.\ \gamma'\rightarrow\gamma'\rightarrow\gamma' \\ &\qquad\qquad \text{since } \texttt{b}[\gamma']\texttt{ft} \in \gamma' \\ &\in (\Delta\gamma.\ \gamma\rightarrow\gamma\rightarrow\gamma)\rightarrow(\Delta\gamma'.\ \gamma'\rightarrow\gamma'\rightarrow\gamma') \\ &\in \texttt{Bool}\rightarrow\texttt{Bool} \end{aligned}$$

Exercise 3.2.1: Check that the above is valid by performing the β-reductions in (not true).

We have shown the representation of some F_1^i inductive types as F_2 types and terms, but we have not said anything about the associated iter operators. In fact, the iterators for F_1^i indtypes may also be specified within F_2. In the case of booleans, iterBool takes a type γ (specifying the result type of the iteration), and a Bool, followed by one argument (of type γ) for each of the constructors true and **false**:

$$\texttt{iterBool} \ \in\ \Delta\gamma.\ \texttt{Bool}\rightarrow\gamma\rightarrow\gamma\rightarrow\gamma$$

Depending on which constructor was used to produce the Bool, iterBool returns either the first argument (when b = true) or the second (when b = **false**):

$$\texttt{iterBool} \ \equiv\ \Lambda\gamma{:}*.\ \lambda\texttt{b:Bool}.\ \lambda\texttt{t}'{:}\gamma.\ \lambda\texttt{f}'{:}\gamma.\ \texttt{b}\ [\gamma]\ \texttt{t}'\ \texttt{f}'$$

(Again you might think of iterBool as an "if... then... else" expression: if b then $\texttt{t}'$ **else** $\texttt{f}'$.)

Consider that true and **false** are each type-parameterized, two-argument functions, similar to iterBool (except that the latter has the boolean argument b). We may use boolean terms directly as iterators by taking advantage of this similar structure; rather than explicitly abstracting `t'` and `f'` as above, we make use of the corresponding `t` and `f` parameters of `b`. So for `b` $\in$ `Bool`, we have the following equivalence

$$\texttt{iterBool}\ [\gamma]\ \texttt{b} \quad = \quad \texttt{b}\ [\gamma]$$

because for `t'`, `f'` $\in \gamma$:

$$\texttt{iterBool}\ [\gamma]\ \texttt{b}\ \texttt{t}'\ \texttt{f}' \quad = \quad \texttt{b}\ [\gamma]\ \texttt{t}'\ \texttt{f}'$$

In fact, we shall see that this "self-iterating" property of `Bool`s holds in general for the representation of all inductively defined types. Explicit iterators are, then, no longer necessary!

As before, we can use iteration to define various functions on booleans. For example, we may alternatively define `not` as

$$\texttt{not} \quad \equiv \quad \lambda\texttt{b:Bool.}\ \texttt{iterBool [Bool] b false true}$$

or now directly as

$$\texttt{not} \quad \equiv \quad \lambda\texttt{b:Bool.}\ \texttt{b [Bool] false true}$$

Although the above is similar to the original definition of `not`, the two are not $\beta\eta$-equivalent, as the latter does not explicitly set up the γ, `t`, and `f` arguments. Rather the booleans `true` and `false` are returned directly.

We may also define binary boolean functions:

$$\begin{array}{lll}
\texttt{or} & \in & \texttt{Bool}\rightarrow\texttt{Bool}\rightarrow\texttt{Bool} \\
\texttt{or} & \equiv & \lambda\texttt{b}_1\texttt{:Bool.}\ \lambda\texttt{b}_2\texttt{:Bool.}\ \texttt{b}_1\ \texttt{[Bool] true b}_2 \\
\texttt{and} & \in & \texttt{Bool}\rightarrow\texttt{Bool}\rightarrow\texttt{Bool} \\
\texttt{and} & \equiv & \lambda\texttt{b}_1\texttt{:Bool.}\ \lambda\texttt{b}_2\texttt{:Bool.}\ \texttt{b}_1\ \texttt{[Bool] b}_2\ \texttt{false}
\end{array}$$

The operation `or` works by returning `true` if $\texttt{b}_1$ is `true` and $\texttt{b}_2$ otherwise, while `and` yields $\texttt{b}_2$ if $\texttt{b}_1$ is `true` and `false` otherwise.

> **Exercise 3.2.2:** Apply `and` or `or` to combinations of `true` and `false` to convince yourself of their validity. Consider the definition of terms performing other boolean operations (e.g., `xor`, `implies`).

We have asked the reader to accept the translation of inductive definitions on faith, but now consider it more carefully. In rendering F_1^{+} types into F_2, the primary goal is to achieve the self-iteration property we have observed in booleans. To accomplish this, *term instances of the defined type must capture the structure of the induction that defined them.* Returning to the natural numbers, we have

$$\texttt{indtype Nat:* with zero:Nat and succ:Nat}\rightarrow\texttt{Nat}$$

rendered as:

$$\texttt{Nat} \quad \equiv \quad \Delta\gamma\texttt{:*.}\ \gamma\rightarrow(\gamma\rightarrow\gamma)\rightarrow\gamma$$

For self-iteration, we must be able to specify a type for the result. Hence instances of type Nat begin with a type abstraction so the terms may be specialized to produce the result type of the iteration. The first γ argument corresponds to the **zero** constructor (of type Nat), while the $\gamma\to\gamma$ parameter corresponds to succ $\in$ Nat$\to$Nat.

Instances of Nat take the following form:

$$\begin{array}{lcl} 0 & \equiv & \Lambda\gamma{:}*.\ \lambda z{:}\gamma.\ \lambda s{:}\gamma\to\gamma.\ z \\ 1 & \equiv & \Lambda\gamma{:}*.\ \lambda z{:}\gamma.\ \lambda s{:}\gamma\to\gamma.\ s\ z \\ 2 & \equiv & \Lambda\gamma{:}*.\ \lambda z{:}\gamma.\ \lambda s{:}\gamma\to\gamma.\ s\ (s\ z) \\ 3 & \equiv & \Lambda\gamma{:}*.\ \lambda z{:}\gamma.\ \lambda s{:}\gamma\to\gamma.\ s\ (s\ (s\ z)) \\ \vdots & & \end{array}$$

Exercise 3.2.3: Verify that all closed, normal-form terms of type Nat have this shape.

The zero function, **zero** $\in$ Nat, should just produce 0:

$$\textbf{zero} \equiv \Lambda\gamma{:}*.\ \lambda z{:}\gamma.\ \lambda s{:}\gamma\to\gamma.\ z$$

The successor function, succ $\in$ Nat$\to$Nat, may then be represented by

$$\text{succ} \equiv \lambda n{:}\text{Nat}.\ \Lambda\gamma{:}*.\ \lambda z{:}\gamma.\ \lambda s{:}\gamma\to\gamma.\ s(n\ [\gamma]\ z\ s)$$

or alternatively as

$$\text{succ}' \equiv \lambda n{:}\text{Nat}.\ \Lambda\gamma{:}*.\ \lambda z{:}\gamma.\ \lambda s{:}\gamma\to\gamma.\ n\ [\gamma]\ (s\ z)\ s$$

Let us make sure again that these definitions are well typed.

$$\begin{array}{ll} \text{succ},\text{succ}' & \in \text{Nat}\to\ \Delta\gamma{:}*.\gamma\to(\gamma\to\gamma)\to\boxed{?} \\ & \in \text{Nat}\to\ \Delta\gamma{:}*.\gamma\to(\gamma\to\gamma)\to\gamma \\ & \qquad\quad \text{since } n[\gamma](sz)s,\ s(n[\gamma]zs) \in \gamma \\ & \in \text{Nat}\to\text{Nat} \end{array}$$

As with the above definition of **not**, the arguments z and s are not actually instantiated when succ is applied to a number.

Exercise 3.2.4: Check that (succ 0) = 1.

The above definitions of succ and succ′ demonstrate how an operation may have two definitions that denote the same function but are not β-equivalent. However, in this case we should not be surprised that the two definitions are not convertible: they are clearly achieving the same result in two different ways. In the first definition, we replace the z part of a number by s(z). Since the arguments to a number are its "zero" and "successor" elements respectively, we have merely replaced the "zero" element with what is really "one." In the second definition, we apply an extra s to the "outside" of the number.

We have defined our representation so that the resulting terms serve as their own iterators, but let us define the explicit iteration schemes for Nat to reinforce this idea. The iterator over natural numbers is defined polymorphically:

```
iterNat  ∈   Δα:*. Nat → α → (α→α) → α
iterNat  ≡   Λα:*. λn:Nat. λz:α. λs:α→α. n[α]zs
```

As is the case with Bools, we have

```
iterNat [α] n  =  n [α]
```

For the remainder of this text, we will generally avoid explicit iterators since the self-iterating approach is both simpler and more elegant.

We previously defined the addition of m and n by taking the mth successor of n. Using the iteration implicit within our representation of Nat, we have

```
plus ∈ Nat → Nat → Nat
```

which may be defined as

```
plus  ≡  λm:Nat. λn:Nat. m [Nat] n succ
```

(The final arguments n and succ are substituted for the z and s arguments of m, respectively.)

Exercise 3.2.5: Try adding two small numbers.

Similarly, multiplication may be defined by iterating plus:

```
mult  ≡  λm:Nat. λn:Nat. n [Nat] 0 (plus m)
```

It might help to think of (plus m) as (λn:Nat. plus m n).

The following predicate tests for zero:

```
zero?  ≡  λm:Nat. m [Bool] true (λb:Bool. false)
```

Now let us return to the formulation of lists we gave in F_1^{i}:

```
indtype List_Nat:* with
                 nil_Nat: List_Nat
             and cons_Nat: Nat → List_Nat → List_Nat
```

The above is rendered in F_2 as:

```
List_Nat  ≡  Δγ:*. γ → (Nat→γ→γ) → γ
nil_Nat   ≡  Λγ:*. λn:γ. λc:Nat→γ→γ. n
cons_Nat  ≡  λx:Nat. λl:List_Nat. Λγ:*. λn:γ. λc:Nat→γ→γ.
                       c x (l [γ] n c)
```

We may use the representation to iterate over a List_Nat producing its sum:

```
sumlist  ∈  List_Nat → Nat
sumlist  ≡  λl:List_Nat. l [Nat] 0 plus
```

Exercise 3.2.6: Render the Tree indtype of Section 3 in F_2.

General or unrestricted recursion is not available in F_2, since recursion can be used to express nonterminating computations and we know that F_2 is strongly normalizing. However, the more restricted mechanism of primitive recursion may be formulated in our calculi.

Like iteration, primitive recursion performs structural induction—that is, computation is guided by the traversal of existing data structure. As these structures are finite, the computation must terminate. We only consider primitive recursion over numbers, but other domains may be formulated analogously.

The primitive recursion scheme on Nats is

$$\begin{aligned} \texttt{f 0} &= \texttt{z} \\ \texttt{f (succ n)} &= \texttt{s n (f n)} \end{aligned}$$

for $\texttt{f} \in \texttt{Nat}\to\alpha$, $\texttt{z} \in \alpha$, $\texttt{s} \in \texttt{Nat}\to\alpha\to\alpha$. Primitive recursion is somewhat more complex than iteration in that successive values may depend on the value of the predecessor n as well as the function result (f n) computed from n. It should be obvious that iteration is a special case of primitive recursion. However, it turns out that primitive recursion may similarly be derived from iteration.

To implement the primitive recursive functions within F_2, we require a representation of pairs. This comes from the need to iterate over s—a function of n and the value of s for n. The pair data structure makes both these values available to the incremental computation at each step of the iteration (as in the cdr of Section 2.3).

As with iterators and lists, we may form the family of cartesian pairs. (As is the case with iter, the need for a type family may be replaced with an abstraction, but the reader must wait for F_3.)

$$\begin{aligned} &\texttt{indtype Pair\_}\sigma\texttt{\_}\tau\texttt{ with} \\ &\qquad\qquad \texttt{pair\_}\sigma\texttt{\_}\tau : \sigma\to\tau\to\texttt{Pair\_}\sigma\texttt{\_}\tau \end{aligned}$$

This may be translated into F_2:

$$\begin{aligned} \texttt{Pair\_}\sigma\texttt{\_}\tau &\equiv \Delta\gamma{:}*.\ (\sigma\to\tau\to\gamma)\to\gamma \\ \texttt{pair\_}\sigma\texttt{\_}\tau &\equiv \lambda x{:}\sigma.\ \lambda y{:}\tau.\ \Lambda\gamma{:}*.\ \lambda p{:}\sigma\to\tau\to\gamma.\ p\ x\ y \end{aligned}$$

So, for example,

$$\texttt{pair\_Bool\_Nat true 1} \;=\; \Lambda\gamma{:}*.\ \lambda p{:}\texttt{Bool}\to\texttt{Nat}\to\gamma.\ p\ \texttt{true}\ 1$$

In order for our pairs to be of any use, we also need families of destructuring operators:

$$\begin{aligned} \texttt{fst\_}\sigma\texttt{\_}\tau &\in \texttt{Pair\_}\sigma\texttt{\_}\tau \to \sigma \\ \texttt{fst\_}\sigma\texttt{\_}\tau &\equiv \lambda w{:}\texttt{Pair\_}\sigma\texttt{\_}\tau.\ w\ [\sigma]\ (\lambda x{:}\sigma.\ \lambda y{:}\tau.\ x) \\ \texttt{snd\_}\sigma\texttt{\_}\tau &\in \texttt{Pair\_}\sigma\texttt{\_}\tau \to \tau \\ \texttt{snd\_}\sigma\texttt{\_}\tau &\equiv \lambda w{:}\texttt{Pair\_}\sigma\texttt{\_}\tau.\ w\ [\tau]\ (\lambda x{:}\sigma.\ \lambda y{:}\tau.\ y) \end{aligned}$$

For example:

$$\begin{aligned} &\texttt{fst\_Bool\_Nat (pair\_Bool\_Nat true 1)} \\ &\quad = (\Lambda\gamma{:}*.\ \lambda p{:}\texttt{Bool}\to\texttt{Nat}\to\gamma.\ p\ \texttt{true}\ 1)\ [\texttt{Bool}]\ (\lambda x{:}\texttt{Bool}.\ \lambda y{:}\texttt{Nat}.\ x) \\ &\quad = \texttt{true} \end{aligned}$$

We may now implement the primitive recursive function f from above as:

```
f ≡ λn:Nat. snd_Nat_α (n [Pair_Nat_α]
                (pair_Nat_α 0 z)
                (λw:Pair_Nat_α. pair_Nat_α
                                   (succ (fst_Nat_α w))
                                   (s (fst_Nat_α w) (snd_Nat_α w))))
```

The function f iterates over pairs of the form pair_Nat_α n (f n):

```
pair_Nat_α  0  z
pair_Nat_α  1  s 0 z
pair_Nat_α  2  s 1 (s 0 z)
pair_Nat_α  3  s 2 (s 1 (s 0 z))
⋮
```

Abstracting over z and s, we arrive at a general formulation of primitive recursion over numbers (Although we would like to abstract over α as well, this is not possible with the current formulation of pairs.):

```
primrec_α  ∈  α→(Nat→α→α)→Nat→α
primrec_α  ≡  λz:α. λs:Nat→α→α.
                λn:Nat. snd_Nat_α
                     (n [Pair_Nat_α]
                          (pair_Nat_α 0 z)
                          (λw:Pair_Nat_α. pair_Nat_α
                                              (succ (fst_Nat_α w))
                                              (s (fst_Nat_α w) (snd_Nat_α w)))
```

The predecessor function on Nat may now be defined as

```
pred  ≡  primrec_Nat  0 (λm:Nat. λn:Nat. m)
```

The pairs within the resulting iteration look like

```
pair_Nat_Nat  0  0
pair_Nat_Nat  1  ((λm:Nat. λn:Nat. m) 0 0)
pair_Nat_Nat  2  ((λm:Nat. λn:Nat. m) 1 0)
pair_Nat_Nat  3  ((λm:Nat. λn:Nat. m) 2 1)
⋮
```

Notice that our predecessor function is surprisingly inefficient, requiring order n time to compute pred n. Unfortunately, there is some theoretical evidence that this inefficiency is ***inherent***.

Exercises 3.2.7:

1. Define the factorial function using primrec.
2. Define a function in F_2 that returns true if its two numeric arguments are equal and false if they are unequal.

After all these examples, the general form of the translation from F_1^i indtype definitions to F_2 types should come as no surprise. The indtype definition

$$\begin{array}{ll} \texttt{indtype T:* with} & \\ & \texttt{c}_1 : \texttt{U}_{11} \rightarrow \texttt{U}_{12} \rightarrow \cdots \rightarrow \texttt{U}_{1n_1} \rightarrow \texttt{T} \\ \quad \textbf{and} & \texttt{c}_2 : \texttt{U}_{21} \rightarrow \texttt{U}_{22} \rightarrow \cdots \rightarrow \texttt{U}_{2n_2} \rightarrow \texttt{T} \\ & \vdots \\ \quad \textbf{and} & \texttt{c}_m : \texttt{U}_{m1} \rightarrow \texttt{U}_{m2} \rightarrow \cdots \rightarrow \texttt{U}_{mn_m} \rightarrow \texttt{T} \end{array}$$

becomes the F_2 type

$$\begin{array}{lll} \texttt{T} & \equiv & \Delta\gamma{:}{*}. \\ & & \quad (\texttt{c}_1 : \hat{\texttt{U}}_{11} \rightarrow \hat{\texttt{U}}_{12} \rightarrow \cdots \rightarrow \hat{\texttt{U}}_{1n_1} \rightarrow \gamma) \\ & & \quad \vdots \\ & & \rightarrow (\texttt{c}_m : \hat{\texttt{U}}_{m1} \rightarrow \hat{\texttt{U}}_{m2} \rightarrow \cdots \rightarrow \hat{\texttt{U}}_{mn_m} \rightarrow \gamma) \\ & & \rightarrow \gamma \end{array}$$

where $\hat{\texttt{U}}_{ij}$ denotes the result of substituting γ for T in $\texttt{U}_{ij}$.

3.3 The Power of F_2

We have seen that F_2 is sufficient to define the primitive recursive functions, but we have not discussed how powerful the language is. In fact, the class of functions definable in F_2 is much larger than the primitive recursive functions. In particular, consider Ackermann's function:

```
ack 0 n                   =  succ n
ack (succ m) 0            =  ack m 1
ack (succ m) (succ n)     =  ack m (ack (succ m) n)
```

Ackermann's function exhibits surprisingly explosive growth and is not primitive recursive (A proof is beyond the scope of this work, but may be found in [44].) It is not even immediately obvious that the function is indeed total—that is, that it always terminates. However, we may argue that it is total simply by the fact that it may be encoded within F_2 [52]:

```
ack  ≡  λm:Nat. m [Nat→Nat]
                  succ
                  λf:Nat→Nat. λn:Nat.
                      (succ n) [Nat] 1 f
```

(In the last line, the unnecessary parentheses indicate that it is the successor of n that is iterated.) As this is a complex term, we shall attempt to give the reader some insight into its behavior. For m, n = 0:

```
ack 0 0  ⇒  succ 0  ⇒  1
```

Similarly:

```
ack 1 1  ⇒  (λn:Nat. (succ n) [Nat] 1 succ) 1
         ⇒  2 [Nat] 1 succ
         ⇒  3
```

The situation gets more complex for $\mathtt{m}, \mathtt{n} = 2$:

```
ack 2 2  ⇒  (λn:Nat. (succ n) [Nat] 1
                    λn':Nat. (succ n') [Nat] 1 succ)   2
         ⇒  3 [Nat] 1 (λn':Nat. (succ n') [Nat] 1 succ)
         ⇒  ((λn₃:Nat. (succ n₃) [Nat] 1 succ)
              ((λn₂:Nat. (succ n₂) [Nat] 1 succ)
                ((λn₁:Nat. (succ n₁) [Nat] 1 succ)
                  1)))
         ⇒  ((λn₃:Nat. (succ n₃) [Nat] 1 succ)
              ((λn₂:Nat. (succ n₂) [Nat] 1 succ)
                3))
         ⇒  ((λn₃:Nat. (succ n₃) [Nat] 1 succ)
               5)
         ⇒  7
```

Ackermann's function serves to illustrate more of F_2's power than our previous examples.

Chapter 4

The Third-order Polymorphic λ-Calculus

We have used the polymorphism of F_2 several times now to translate `indtype` definitions of data types into their representations as closed polymorphic types. But so far, all of the data types we have translated have themselves been monomorphic. In order to go beyond these to "parametric" types like `List` and `Pair`, we have to extend our language again, bringing us to F_3.

4.1 Definition and Properties of F_3

In a sense, lists and cartesian pairs *can* be defined in F_2: for each type α, there is a type `List_`α. Similarly, for each pair of types σ and τ, the type of cartesian pairs of elements of σ with elements of τ can be expressed by the inductive type definition

$$\texttt{indtype Pair\_}\sigma\texttt{\_}\tau\ \texttt{with pair\_}\sigma\texttt{\_}\tau\texttt{:}\ \sigma \to \tau \to \texttt{Pair\_}\sigma\texttt{\_}\tau$$

which translates, by the usual method, to the following representation in F_2:

$$\texttt{Pair\_}\sigma\texttt{\_}\tau \equiv \Delta\gamma{:}*.\,(\sigma \to \tau \to \gamma) \to \gamma$$

But this formulation of pairs is awkward: each `Pair_`σ`_`τ instance is different from all others. Whenever we want to use a new kind of pair, we have to write down a new type definition that is structurally identical to the others. We need to abstract away from the types of the elements of the pair and write something like:

```
indtype Pair ...
```

Of course, `Pair` by itself is not a type. (There are no terms whose type is just `Pair`: each pair of terms has type `Pair_`σ`_`τ for some particular σ and τ.) `Pair` itself is better thought of as a "type constructor," a function from types to types. This sort of `indtype` definition cannot be expressed in F_1^i, nor to our knowledge can it be represented as a closed type in F_2. This motivates our next language extension, which takes us to F_3. (We will not bother defining a separate language F_2^i this time. It would be very similar to F_3, so it is simpler to begin with F_3 as a whole and then informally characterize the inductive type declarations that can be represented by closed types in F_3.)

The main new feature of F_3 is the ability to express functions from types to types. We will use the same notation as at the term level: the abstraction operator on types is written λ, and the juxtaposition of two type expressions denotes the application of the first to the second. (As with ordinary terms, application at the type level is left-associative.)

Finally, just as we needed types to make sure that terms involving abstraction and application were well formed, we now need some notion of the "types" of type expressions to keep *them* under control. We call the types of types *kinds*. There is a constant kind named $*$, which is the kind of types of terms; that is, each-well-typed term e has a type α and the kind of α is $*$. (Actually, we introduced $*$ in F_2, where it was the only kind and therefore could be treated as pure syntax.) If K is a kind, then so is $*\rightarrow K$ (the kind of a type function from $*$ to K).

Definition 4.1.1: The syntax of F_3 is given by the following inductively defined classes:

$$\begin{array}{lcl} K & ::= & * \mid *\rightarrow K \\ T & ::= & \alpha \mid T\rightarrow T' \mid \Delta\alpha : K.T \mid \lambda\alpha : K.T \mid T\,T' \\ e & ::= & x \mid \lambda x : T.e \mid e\,e' \mid \Lambda\alpha : K.e \mid e\,[T] \end{array}$$

where K ranges over kinds, T ranges over types, α ranges over type variables, e ranges over expressions, and x ranges over variables.

There can be no confusion between λ at the term level and λ at the type level. The former takes a term argument and returns a term; the latter takes a type argument and returns a type. However, the difference between λ (on types) and Δ can be confusing. Type expressions beginning with Δ are the types of terms beginning with Λ (and, of course, of variables of this type)—that is, the type of functions that take a type argument and return a term. Type expressions beginning with λ, on the other hand, do not correspond to any terms at all: before they may be the type of a term, they must be applied to enough arguments to produce a type whose outermost operator is either Δ or $\rightarrow$, that is, something of kind $*$. (It does not make sense for λ to be nested within Δ or $\rightarrow$ as the latter are of base kind $*$, that is, the types of terms.)

As an example of F_3, here is an application of an "even more polymorphic" identity function to some arguments:

```
(ΛΘ:*→*. Λσ:*. λx:Θσ. x)
    [λσ. List σ]
    [Nat]
    (cons [Nat] 5 (nil [Nat]))
```

(More useful examples appear below!)

An important notion we need to introduce here, corresponding to that of type judgment, is the notion of kind judgment. We write

$$\Pi \vdash \mathtt{T} \in \mathtt{K}$$

to mean that in the context Π the type expression $\mathtt{T}$ has kind $\mathtt{K}$. We may abbreviate this as

$$\mathtt{T} \in \mathtt{K}$$

when the context Π is clear. By analogy with terms, we also write

$$\Pi(\alpha) = \mathtt{K}$$

or:

$$\Pi(\mathtt{T}) = \mathtt{K}$$

and extend Π with the kind of a type variable by writing $\Pi[\alpha/\mathtt{K}]$.

4.2 Polymorphic Inductive Datatypes

As before, we think of `indtype` definitions as adding new constants for types, constructors, and iterators to a global environment. But now that we know we will be translating them into simple "macro definitions" of names for closed types and terms in F_3, we need not bother to be so formal.

One specific point of informality appears when we need to talk about the "result type" of the constructors, which is hard because of an asymmetry in the language. In F_1^i it was easy because the type of any constructor was just a sequence of zero or more arrows, one for each argument, with the argument types on the left of the arrows and the result type on the right of the last arrow. In F_2^i, the type of a constructor may be built up from any combination of $\rightarrow$s and Δs. Intuitively, this presents no problem: we can think of $\rightarrow$ and Δ as being the same sort of constructs (both are type operators) and explain their different appearances by observing that Δ binds a variable while $\rightarrow$ does not. (In the calculus of constructions [12], for example, this difference disappears and the two operators correspond exactly in form.) So we can still think of an innermost or "rightmost" component of a type. But the notational difficulties involved in giving a completely formal definition of F_2^i are formidable.

We need to know how the notion of positivity extends to types containing Δ:

Definition 4.2.1: The set $Pos(\mathtt{U})$ of positively occurring variables in an F_2 type expression $\mathtt{U}$ is defined by:

$$\begin{array}{lcll} Pos(\alpha) & = & \{\alpha\} & \text{(where } \alpha \text{ is a type variable)} \\ Pos(\mathtt{V}\rightarrow\mathtt{W}) & = & Neg(\mathtt{V}) \cup Pos(\mathtt{W}) & \\ Pos(\Delta\alpha{:}\mathtt{K}.\mathtt{V}) & = & Pos(\mathtt{V}) & \end{array}$$

The set $Neg(\mathtt{U})$ of negatively occurring variables in a type expression $\mathtt{U}$ is defined by:

$$\begin{array}{lcll} Neg(\alpha) & = & \{\} & \text{(where } \alpha \text{ is a type variable)} \\ Neg(\mathtt{V}\rightarrow\mathtt{W}) & = & Pos(\mathtt{V}) \cup Neg(\mathtt{W}) & \\ Neg(\Delta\alpha{:}\mathtt{K}.\mathtt{V}) & = & Neg(\mathtt{V}) & \end{array}$$

Another small point that we need to deal with in this definition is that the types given in an **indtype** definition may now involve λ at the type level, so it only makes sense to talk about the rightmost component of the *normal form* of a type expression.

Definition 4.2.2: The general form of an **indtype** definition with representation in F_3 is:

```
indtype T:*→···→* with
        x₁: U₁
        ⋮
    and xₘ: Uₘ
```

where each U_i is an F_3 type expression, the rightmost component (of the normal form) of which is T applied to $n-1$ types, $n \geq 1$ being the number of occurrences of $*$ on the first line. The type variable T may not occur negatively in any of the arguments to the constructors.

Each such definition introduces the following global constants:

1. The type constructor T.
2. Zero or more constructors x_i. Each x_i takes zero or more type and/or term arguments (as specified by U_i) and returns a term whose type is T applied to $n-1$ arguments.
3. An iteration operator **iterT**, whose type is

$$\begin{aligned} \texttt{iterT} \in\ & \Delta\alpha_1{:}*.\ \ldots\ \Delta\alpha_{n-1}{:}*.\ \Delta\Gamma{:}*\to\cdots\to*. \\ & \mathtt{T}\ \alpha_1\ \ldots\ \alpha_n \\ & \to \hat{\mathtt{U}}_1 \\ & \vdots \\ & \to \hat{\mathtt{U}}_m \\ & \to \Gamma\ \alpha_1\ \ldots\ \alpha_{n-1} \end{aligned}$$

where $\hat{\mathtt{U}}$ denotes the result of substituting Γ for all occurrences of T in U.

4.3 Programming in F_3

Now we can complete our definition of **Pair** as an **indtype**. We know so far that it takes two types as arguments and returns a type, and therefore that its kind is $*\to*\to*$:

$$\texttt{indtype Pair:}*\to*\to*\ \texttt{with}\ \boxed{?}$$

Next, we need to define the constructor **pair**. In general, it takes two terms and returns a pair whose first component has the type of the first term and whose second component has the type of the second term. But we cannot just write

$$\texttt{indtype Pair:}*\to*\to*\ \texttt{with pair:}\ \alpha\to\beta\to(\texttt{Pair}\ \alpha\ \beta)$$

because the type variables α and β are unbound: we do not know in advance what the types of the arguments to pair will be. So we take α and β as extra parameters:

$$\texttt{indtype Pair:}*\to*\to*\ \texttt{with pair:}\ \Delta\alpha{:}*.\ \Delta\beta{:}*.\ \alpha\to\beta\to(\texttt{Pair}\ \alpha\ \beta)$$

Whenever we use pair we will need to provide four arguments:

```
pair [Nat] [Bool] 5 true
```

A simple function manipulating polymorphic pairs is the destructor fst:

$$
\begin{aligned}
\texttt{fst} \;\equiv\; & \Lambda\alpha{:}*.\ \Lambda\beta{:}*.\ \lambda\texttt{w}{:}\texttt{Pair}\ \alpha\ \beta. \\
& \qquad \texttt{iterPair}\ [\alpha]\ [\beta]\ [\lambda\phi{:}*.\lambda\psi{:}*.\phi] \\
& \qquad\qquad \texttt{w} \\
& \qquad\qquad (\Lambda\phi{:}*.\ \Lambda\psi{:}*.\ \lambda\mathrm{x}{:}\phi.\ \lambda\mathrm{y}{:}\psi.\ \mathrm{x})
\end{aligned}
$$

Note that the iterator takes three type parameters instead of just one. The first two specify the type of the argument to the iterator, which is not just Pair, but Pair α β for some particular α and β. Moreover, the third argument is not a type, but a type function with two arguments (i.e., something of the same kind as Pair). It specifies how the type of the iteration's result depends on the types of the first and second projections of w.

Let us translate our "higher-order" indtype definition of Pair into a F_3 type definition. The structure of the translation is a bit different this time, but the intuition is the same. Pair itself is a type constructor, not a type; it should take two type parameters and return some type based on them. So our first approximation to the translation is

$$\texttt{Pair} \equiv \lambda\alpha{:}*.\ \lambda\beta{:}*.\ \boxed{?}$$

where $\boxed{?} \in *$.

Now we can proceed as before. The next argument is a type representing Pair—now of kind $*{\rightarrow}*{\rightarrow}*$ rather than just $*$, but playing the same role as before:

$$\texttt{Pair} \equiv \lambda\alpha{:}*.\ \lambda\beta{:}*.\ \Delta\Gamma{:}*{\rightarrow}*{\rightarrow}*.\ \boxed{?}$$

The indtype definition of Pair has one constructor, pair, which we represent here by an argument of the appropriate type (with Pair replaced by the bound variable Γ):

$$
\begin{aligned}
\texttt{Pair} \equiv\ & \lambda\alpha{:}*.\ \lambda\beta{:}*.\ \Delta\Gamma{:}*{\rightarrow}*{\rightarrow}*. \\
& \qquad (\Delta\sigma{:}*.\Delta\tau{:}*.\ \sigma{\rightarrow}\tau{\rightarrow}(\Gamma\ \sigma\ \tau)) \\
& \qquad \rightarrow \boxed{?}
\end{aligned}
$$

As usual, we finish by adding "$\rightarrow \boxed{?}$" at the end, where $\boxed{?}$ represents the type we are defining. In this case, we must apply the bound variable Γ to the arguments α and β to get something of kind $*$ (just Γ wouldn't make syntactic sense). Our complete definition is:

$$
\begin{aligned}
\texttt{Pair} \equiv\ & \lambda\alpha{:}*.\ \lambda\beta{:}*.\ \Delta\Gamma{:}*{\rightarrow}*{\rightarrow}*. \\
& \qquad (\Delta\sigma{:}*.\Delta\tau{:}*.\ \sigma{\rightarrow}\tau{\rightarrow}\Gamma\ \sigma\ \tau) \\
& \qquad \rightarrow\ (\Gamma\ \alpha\ \beta)
\end{aligned}
$$

Our next job is to define the constructor pair in F_3. Its arguments are two types, σ and τ, and two terms of types σ and τ, respectively.

$$\texttt{pair} \equiv \Lambda\sigma{:}*.\ \Lambda\tau{:}*.\ \lambda\mathrm{x}{:}\sigma.\ \lambda\mathrm{y}{:}\tau.\ \boxed{?}$$

Because $\boxed{?}$ must be something of type Pair σ τ, we can proceed by examining the definition of Pair. We see that $\boxed{?}$ must begin with a type abstraction for the bound variable Γ, followed by an abstraction for the bound constructor p:

$$\texttt{pair} \equiv \Lambda\sigma{:}*.\ \Lambda\tau{:}*.\ \lambda x{:}\sigma.\ \lambda y{:}\tau.\\ \Lambda\Gamma{:}*{\rightarrow}*{\rightarrow}*.\\ \lambda p{:}\ (\Delta\sigma'{:}*.\Delta\tau'{:}*.\sigma'{\rightarrow}\tau'{\rightarrow}(\Gamma\ \sigma'\ \tau')).\ \boxed{?}$$

There is only one choice for $\boxed{?}$ since it must be of type $\Gamma\sigma\tau$:

$$\texttt{pair} \equiv \Lambda\sigma{:}*.\ \Lambda\tau{:}*.\ \lambda x{:}\sigma.\ \lambda y{:}\tau.\\ \Lambda\Gamma{:}*{\rightarrow}*{\rightarrow}*.\\ \lambda p{:}\ \Delta\sigma'{:}*.\Delta\tau'{:}*.\sigma'{\rightarrow}\tau'{\rightarrow}(\Gamma\ \sigma'\ \tau').\\ p[\sigma][\tau]\ x,y$$

It should be starting to become clear that this sort of definition is not as circular as it looks, but to emphasize the point once again let us define the destructor **fst** without using the **iterPair** operator.

From its type

$$\texttt{fst} : \Delta\alpha{:}*.\ \Delta\beta{:}*.\ (\texttt{Pair}\ \alpha\ \beta) \rightarrow \alpha$$

we can see that the outermost part of the definition of **fst** must be

$$\texttt{fst} \equiv \Lambda\alpha{:}*.\ \Lambda\beta{:}*.\ \lambda w{:}\texttt{Pair}\ \alpha\ \beta.\ \boxed{?}$$

where $\boxed{?} \in \alpha$. Clearly, to get something of type α we need to do something with w. Expanding the definition of Pair,

$$\texttt{fst} \equiv \Lambda\alpha{:}*.\ \Lambda\beta{:}*.\\ \lambda w{:}(\Delta\Gamma{:}*{\rightarrow}*{\rightarrow}*.\ (\Delta\sigma{:}*.\Delta\tau{:}*.\sigma{\rightarrow}\tau{\rightarrow}\Gamma\sigma\tau) \rightarrow \Gamma\alpha\beta).\\ \boxed{?}$$

we see that $\boxed{?}$ will take the form w[F]f for some type constructor F whose kind is $*{\rightarrow}*{\rightarrow}*$, and some term f whose type is $\Delta\sigma{:}*.\Delta\tau{:}*.\sigma{\rightarrow}\tau{\rightarrow}F\sigma\tau$. The F and f that will do the job are

$$F \equiv \lambda\phi{:}*.\lambda\psi{:}*.\phi\\ f \equiv \Lambda\phi{:}*.\Lambda\psi{:}*.\lambda x{:}\phi.\lambda y{:}\psi.x$$

so the final definition is:

$$\texttt{fst} \equiv \Lambda\alpha{:}*.\ \Lambda\beta{:}*.\\ \lambda w{:}(\Delta\Gamma{:}*{\rightarrow}*{\rightarrow}*.\ (\Delta\sigma{:}*.\Delta\tau{:}*.\sigma{\rightarrow}\tau{\rightarrow}(\Gamma\ \sigma\ \tau)) \rightarrow (\Gamma\ \alpha\ \beta)).\\ w\ [\lambda\phi{:}*.\lambda\psi{:}*.\phi]\ (\Lambda\phi{:}*.\Lambda\psi{:}*.\lambda x{:}\phi.\lambda y{:}\psi.x)$$

Exercises 4.3.1:

1. Let x:T and y:U. Show that fst[T][U](pair[T][U] x y) = x.
2. By analogy with **fst**, define a destructor snd that returns the second component of a pair.
3. What happens to the first two arguments of iterPair when the F_2^i version of **fst** is translated into pure F_3?

Here is the **indtype** definition of the type of polymorphic lists:

```
indtype List:*→* with
                 nil: Δα:*. List α
            and cons: Δα:*. α → List α → List α
```

Reasoning as before, we begin its representation as a closed F_3 type with a type abstraction that supplies the type γ of list elements, followed by a higher-order type abstraction on a variable Γ representing `List` itself, followed by $\Gamma\gamma$:

```
List ≡ λγ:*. ΔΓ:*→*. ⍰ → Γγ.
```

Now we fill in the constructors `nil` and `cons`, replacing `List` in their types by Γ:

```
List ≡ λγ:*. ΔΓ:*→*.
                 (Δα:*. Γα)
               → (Δα:*. α → Γα → Γα)
               → Γγ
```

The constructors `nil` and `cons` are given by:

```
nil ≡ Λγ:*. ΛΓ:*→*.
               λn: (Δα:*. Γα).
               λc: (Δα:*. α → Γα → Γα).
                 n[γ]
cons ≡ Λγ:*. λx:γ. λl:List γ.
          ΛΓ:*→*.
               λn: (Δα:*. Γα).
               λc: (Δα:*. α → Γα → Γα).
                 c[γ] x (l[γ] n c)
```

By this stage, the reader should be able to check fairly easily that these definitions make sense.

The definition of `car` in Section 2.3 involved a straightforward use of induction on lists. Unfortunately, the polymorphic case is more difficult. The induction scheme for `car` is

```
car nil[γ] d        = d
car (cons[γ] x l) d = x
```

where γ is the type of the list elements and d is a default element representing the car of an empty list. This extra parameter is what causes the trouble. By analogy with Section 2.3, we would like to write:

```
car ≡ Λγ:*. λl:List γ. λd:γ.
          l [⍰]
            (Λα:*. d)
            (Λα:*. λx:α. λtl':α. x)
```

But there is no type expression to put in place of $\boxed{?}$ that makes both arms of the iterator well-typed. Writing $\lambda\theta{:}{*}.\gamma$ fails for the `cons` arm; writing $\lambda\theta{:}{*}.\theta$ fails for the `nil` arm.

Parameterized induction in F_3 always exhibits this difficulty with the typing of extra parameters. The solution is to reorder the applications so that types are instantiated differently:

```
car ≡ Λγ:*. λl:List γ. λd:γ.
        (l [λθ:*. θ→θ]
           (Λα:*. λd:α. d)
           (Λα:*. λx:α. λtl':α→α. λd:α. x))
        d
```

Exercises 4.3.2:

Hand evaluate

```
car [Nat] (nil [Nat]) 0
```

and

```
car [Nat] (cons [Nat] 3 (nil [Nat])) 0
```

1. Define the cdr operation on polymorphic lists. (It is a simple generalization of the F_2 version, since it requires no extra parameters.)

Chapter 5

F_3 as a Metalanguage

Higher-order inductive types appear whenever typed data structures (programs, proofs, *etc.*) are to be represented and manipulated by other programs. In this chapter we show how F_1 and F_2 terms can be represented as F_2^i data structures and discuss how this translation can be generalized to higher orders (representing F_3 in F_3^i, *etc.*)

5.1 A Simple Representation of F_1 Terms

To get some feeling for the technique, we begin with a very simple representation of F_1 terms as an inductive type in F_2^i, which, as usual, can be translated into a closed type in F_3. (We prefer to use inductive types in this section rather than programming directly in F_3 because it makes the definitions easier to read.) `Term` α is the type of data structures representing F_1 terms of type α. In this representation, only applications appear as explicit constructors in the data structure representing an F_1 term. Variables and λ-abstractions appear only inside of "rep" nodes at the leaves of a tree of applications:

```
indtype Term:*→* with
            rep: Δα:*. α → Term α
            app: Δα:*. Δβ:*. Term(α→β) → Term α → Term β
```

For example, the F_1 term[1]

```
(λx:Nat. λy:Nat. plus x y) 2 3
```

[1]Strictly speaking, this is an F_1^+ term, not a term of pure F_1. For illustrative purposes in the early parts of this chapter, we blur the distinction between the two languages and assume that types like `Nat` are available in F_1. The inconsistency will disappear when we extend our representation in Section 5.3 to include all of F_2.

is represented by the F_2^i data structure

```
app [Nat] [Nat]
    (app [Nat] [Nat→Nat]
        (rep [Nat→Nat→Nat] (λx:Nat. λy:Nat. plus x y))
        (rep [Nat] 2))
    (rep [Nat] 3)
```

Exercise 5.1.1: Translate the indtype definition of Term into pure F_3.

It is already apparent why we seem to need to get out of F_2 in order to represent F_1 terms. All of the inductively defined types that we can translate into F_2 by the techniques described in Chapter 3 share the property of being *homogeneous*. For example, every sublist of a List_Nat is also a List_Nat. With terms, this is no longer the case. A F_1 term of type α may have subterms of types $\beta \rightarrow \alpha$ and β, for arbitrary β. To represent such *nonhomogeneous* types with inductive definitions, constructors with Δ-types are required.

Another way of looking at this phenomenon is as follows: in the F_3 representation of the F_2^i type constructor Term, the type operator Γ is applied to different arguments. In the representation of List, it was possible to define an infinite set of F_2 types List_Nat, List_Bool, *etc.*, each corresponding to an instance of the type constructor List where the variable Γ (representing List itself) was applied throughout to exactly one argument α, making it possible to replace $(\Gamma\alpha)$ everywhere by a single type variable γ. We cannot think of Term in this way.

There may be many representations of a given F_1 term as a Term data structure. For example, the term above is also represented by:

```
rep [Nat] ((λx:Nat. λy:Nat. plus x y) 2 3)
```

This is a common characteristic of all the term representations discussed in this chapter. However, we can distinguish a "canonical" representation of each F_1 term as an instance of Term by stipulating that any application in the original term that is not in the scope of any λ must be represented by an app node. (So the first representation above is canonical, and the second is not.)

Formally, we can define the *canonical representation* $[\![t]\!]_\Pi$ of a term e in an environment Π where the types of e's free variables are given by Π. Π is a function taking each free variable of e to an F_1 type. We write $\Pi[x/T]$ for the environment Π' that agrees with Π everywhere except x, and that assigns x the type T. When Π is an environment and e is a well-typed F_1 term of type α (assuming its free variables have the types assigned to them by Π), we write $\Pi(e)=\alpha$.

$$
\begin{array}{lcll}
[\![x]\!]_\Pi & = & \texttt{rep } [\alpha] \texttt{ x} & \text{where } \Pi(x)=\alpha \\
[\![\lambda x{:}\alpha.e]\!]_\Pi & = & \texttt{rep } [\alpha\rightarrow\beta] \ (\lambda x{:}\alpha.e) & \text{where } \Pi(\lambda x{:}\alpha.e)=\alpha\rightarrow\beta \\
[\![e\ e']\!]_\Pi & = & \texttt{app } [\alpha] \ [\beta] \ [\![e]\!]_\Pi \ [\![e']\!]_\Pi & \text{where } \Pi(e)=\alpha\rightarrow\beta \text{ and } \Pi(e')=\alpha
\end{array}
$$

This translation allows us to use the double square brackets as syntactic sugar in F_3. We can write F_3 programs like

```
λx:Nat. cons [Term(Nat)] ⟦x⟧
                   (cons [Term(Nat)] ⟦plus x x⟧
                         (nil [Term(Nat)]))
```

(letting the typechecker fill in the appropriate environments), instead of:

```
λx:Nat. cons [Term(Nat)]
          (rep [Nat] x)
          (cons [Term(Nat)]
                (app [Nat] [Nat]
                     (app [Nat] [Nat→Nat]
                          (rep [Nat→Nat→Nat] plus)
                          (rep [Nat] x)
                          (rep [Nat] x)))
                (nil [Term(Nat)]))
```

Having defined a representation of F_1 terms in F_3, it is natural to ask what sorts of manipulations can be performed on this representation. This turns out to be a difficult question, and we defer a complete discussion until we have fully developed the representation itself (see Section 5.5). But it is worth noting here that it is easy to define an evaluation function on this representation:

```
eval  ∈  Δα:*. Term α → α
eval  ≡  Λα:*. λt:Term α.
             iterTerm [α] [λσ:*.σ] t
                 (Λσ:*. λu:σ. u)
                 (Λσ:*. Λτ:*. λu:σ→τ. λv:σ. u v)
```

Exercises 5.1.2:

1. Try applying eval to the two instances of Term given at the beginning of the section.
2. Translate eval into pure F_3.

5.2 A Complete Representation of F_1 Terms

Since app is used to represent applications and rep can be used to represent variables, the only construct left to consider is λ. There are several possible formulations for the lam constructor (see Section 5.5). This one is due to Pfenning and Lee [49]:

```
indtype Term:*→* with
            rep: Δα:*. α→Term α
            lam: Δα:*. Δβ:*. (α → Term β) → Term(α → β)
            app: Δα:*. Δβ:*. Term(α → β) → Term α → Term β
```

For example, the F_1 term

```
        (λf:Nat→Nat. f) (λx:Nat. x)
```

is represented by:

```
        app [Nat→Nat] [Nat→Nat]
            (lam [Nat→Nat] [Nat→Nat] (λf:Nat→Nat. rep [Nat→Nat] f))
            (lam [Nat] [Nat] (λx:Nat. rep [Nat] x))
```

This representation may be a little puzzling at first. Rather than introducing a data type for F_1 variables and having the `lam` constructor take the representation of a variable (the bound variable) and the representation of a term (the body of the λ-abstraction) as arguments, we represent an F_1 λ-abstraction by a ***higher-order*** `Term`—a function from `Term`s to `Term`s. Intuitively, we have incorporated the notion of substitution (or β-reduction) as an integral part of the definition of terms: F_1 variables have been replaced by F_3 variables. This method was introduced by Church [10] and appears in the work of Martin-Löf [42], the "Second-Order Patterns" of Huet and Lang [32], the encodings of logics in Harper, Honsell, and Plotkin's "Logical Framework" [25], and the idea of "Higher-Order Abstract Syntax" of Pfenning and Elliott [48].

The revised canonical representation of F_1 terms is:

$$\begin{array}{lcll}
[\![\mathtt{x}]\!]_\Pi & = & \mathtt{rep}\ [\alpha]\ \mathtt{x} & \text{where } \Pi(x)=\alpha \\
[\![\lambda\mathtt{x}{:}\alpha.\mathtt{e}]\!]_\Pi & = & \mathtt{lam}\ [\alpha]\ [\beta]\ \lambda\mathtt{x}{:}\alpha.[\![\mathtt{e}]\!]_{\Pi[\mathtt{x}/\alpha]} & \text{where } \Pi[\mathtt{x}/\alpha](\mathtt{e})=\beta \\
[\![\mathtt{e}\ \mathtt{e}']\!]_\Pi & = & \mathtt{app}\ [\alpha]\ [\beta]\ [\![\mathtt{e}]\!]_\Pi\ [\![\mathtt{e}']\!]_\Pi & \text{where } \Pi(\mathtt{e})=\alpha\to\beta \text{ and } \Pi(\mathtt{e}')=\alpha
\end{array}$$

The definition of `eval` is extended to:

```
        eval  ≡  Λα:*. λt:Term α.
                     iterTerm [α] [λσ:*.σ] t
                         (Λσ:*. λu:σ. u)
                         (Λσ:*. Λτ:*. λu:σ→τ. u)
                         (Λσ:*. Λτ:*. λu:σ→τ. λv:σ. u v)
```

Exercise 5.2.1: Translate the F_1 term

```
    (λx:Nat.x) 5
```

into its representation as a `Term`. Apply `eval` to the result.

Our original definition of `Term` in Section 5.1 allowed certain F_1 terms—those consisting only of applications and variables—to be represented canonically, in the sense that there was a bijection between F_1 terms not containing λ and instances of `Term`. We would hope that our new definition of `Term` would create a bijection between *all* the F_1 terms and those F_3 terms of type `Term` where `rep` is applied only to variables. This is not quite the case, since two F_1 terms that differ only in the names of bound variables will have identical canonical representations. But if we consider such pairs of F_1 terms to be identical, we can make the following conjecture:

Conjecture 5.2.2: There is a bijection between (α-equivalence classes of) well-typed terms in F_1 and normal-form terms of type `Term` in F_3.

The importance of this conjecture is that it does *not* appear to hold for the versions of `Term` without `lam`. (We can make a similar conjecture for the extensions of `Term` in Sections 5.3 and 5.4.)

5.3 Representation of F_2

Using the technique that we just applied to λ, it is also possible to represent type abstraction and application, giving a representation of F_2 in F_3.

```
indtype Term:*→* with
        rep: Δα:*. α→Term α
        lam: Δα:*. Δβ:*. (α → Term β) → Term(α → β)
        app: Δα:*. Δβ:*. Term(α → β) → Term α → Term β
        typlam: ΔΘ:*→*. (Δα:*. Term(Θα)) → Term(Δα:*. Θα)
        typapp: ΔΘ:*→*. Term(Δα:*. Θα) → Δα:*. Term(Θα)
```

The intuition behind the constructors `typlam` and `typapp` is exactly the same as for `lam`. It is an important characteristic of this representation of F_2 terms that, like variables, types are represented *implicitly*: there is no explicit encoding of F_2 types as an F_3 data structure. Instead, a `Term` representing an F_2 term of type α has type `Term` α, where α is an actual F_3 type. Among the benefits of this approach is the fact that it is impossible to construct a poorly typed `Term`. Only well-typed F_2 terms can be represented as instances of `Term`. (It is this fact that makes the definition of `eval` so simple, since it means that there is no need to deal explicitly with types, typechecking, or the possibility of failure at runtime.)

Like `lam`, the `typlam` constructor takes a *function* as its main argument—this time a function from types to representations of terms rather than from values to representations of terms. The type of the `Term` returned by this argument is some function ($\Theta\alpha$) of the type (α) it is passed. The first parameter to `typlam` specifies this dependency.

Thus, Δ can be thought of as almost exactly like $\rightarrow$, except that $\rightarrow$ does not require this extra Θ to specify how its right hand side depends on its left hand side. Otherwise the correspondence between Δ and $\rightarrow$ would be exact.)

Again, `typapp` is similar to application, except that it needs an extra parameter Θ to specify the relation between the argument and result types.

The extended representation algorithm is:

$$\begin{array}{lcll}
[\![x]\!]_\Pi & = & \texttt{rep}\ [\alpha]\ x & \text{where } \Pi(x)=\alpha \\
[\![\lambda x{:}\alpha.e]\!]_\Pi & = & \texttt{lam}\ [\alpha]\ [\beta]\ \lambda x{:}\alpha.[\![e]\!]_{\Pi[x/\alpha]} & \text{where } \Pi[x/\alpha](e)=\beta \\
[\![e\ e']\!]_\Pi & = & \texttt{app}\ [\alpha]\ [\beta]\ [\![e]\!]_\Pi\ [\![e']\!]_\Pi & \text{where } \Pi(e)=\alpha\rightarrow\beta \text{ and } \Pi(e')=\alpha \\
[\![\Lambda\alpha{:}*.e]\!]_\Pi & = & \texttt{typlam}\ [\Theta]\ \Lambda\alpha{:}*.[\![e]\!]_{\Pi[\alpha/*]} & \text{where } \Pi(\Lambda\alpha{:}*.e)=\Delta\alpha{:}*.\Theta\alpha \\
[\![e[T]]\!]_\Pi & = & \texttt{typapp}\ [\Theta]\ [\![e]\!]_\Pi\ [T] & \text{where } \Pi(e)=\Delta\alpha{:}*.\Theta\alpha
\end{array}$$

The extended `eval` function is:

```
eval  ≡  Λα:*. λt:Term α.
              iterTerm [α] [λσ:*.σ] t
                   (Λσ:*. λu:σ. u)
                   (Λσ:*. Λτ:*. λu:σ→τ. u)
                   (Λσ:*. Λτ:*. λu:σ→τ. λv:σ. u v)
                   (ΛΘ:*→*. λu:(Δα:*. Θα). u)
                   (ΛΘ:*→*. λu:(Δα:*. Θα). Λα:*. u[α])
```

Exercise 5.3.1: Translate the F_2 term

`(Λα:*.λx:α.x) [Nat] 5`

into its representation as a `Term`. Apply `eval` to the result.

5.4 Representing F_3 in F_4

The only difference between F_3 and F_4 is in the kinds that may appear in terms. Whereas F_3 uses only the kinds $*$ and $*\rightarrow*$, F_4 also allows the kind $(*\rightarrow*)\rightarrow*$, the kind of functions (at the type level) that take type functions as arguments. When we allow type expressions involving this new kind, it becomes possible to represent F_3 programs using an inductive type definition in F_3^i:

```
indtype Term:*→* with
        rep: Δα:*. α→Term α
        lam: Δα:*. Δβ:*. (α → Term β) → Term(α → β)
        app: Δα:*. Δβ:*. Term(α → β) → Term α → Term β
        typlam1: ΔΘ:*→*.
                      (Δα:*. Term(Θα)) → Term(Δα:*. Θα)
        typlam2: ΔΘ:(*→*)→*.
                      (Δα:*→*. Term(Θα)) → Term(Δα:*→*. Θα)
        typapp1: ΔΘ:*→*.
                      Term(Δα:*. Θα) → (Δα:*. Term(Θα))
        typapp2: ΔΘ:(*→*)→*.
                      Term(Δα:*→*. Θα) → (Δα:*→*. Term(Θα))
```

Note that since the kind of Θ appears explicitly in the definition of `Term`, we need to have two separate cases for `typlam` and `typapp`.

Exercises 5.4.1:

1. Generalize the definitions of $[\![]\!]$ and eval to correspond to this version of Term.
2. In F_5, functions at the type level may take arguments of any F_4 kind. Write down a Term representation for F_4 in F_4^i. How many cases are needed for **typlam** and **typapp**?

5.5 Alternative Formulations of Term

The higher-order-abstract-syntax-style formulation of lam in Section 5.2 may have seemed somewhat arbitrary and curious, but it has properties that are not shared by the other obvious formulations.

Starting from scratch, our first attempt at a constructor for lam might be:

$$\texttt{lam}:\ \Delta\alpha{:}*.\ \Delta\beta{:}*.\ (\texttt{Term}\ \alpha \rightarrow \texttt{Term}\ \beta) \rightarrow \texttt{Term}(\alpha \rightarrow \beta)$$

But this clearly won't work for us because it cannot be part of an inductive type definition (the variable Term appears negatively). We might then consider two possible patches.

First, we can erase the negative occurence of Term, leaving

$$\texttt{lam}:\ \Delta\alpha{:}*.\ \Delta\beta{:}*.\ (\alpha \rightarrow \texttt{Term}\ \beta) \rightarrow \texttt{Term}(\alpha \rightarrow \beta)$$

and rely on our ability to use rep to convert values of type α to values of type Term α when we're building functions to pass to the constructor. This is the approach we adopted in Section 5.2.

Alternatively, we can erase the parentheses so that Term remains but is no longer in a negative position:

$$\texttt{lam}:\ \Delta\alpha{:}*.\ \Delta\beta{:}*.\ \texttt{Term}\ \alpha \rightarrow \texttt{Term}\ \beta \rightarrow \texttt{Term}(\alpha \rightarrow \beta)$$

This almost corresponds to the syntactic notion that "a λ-abstraction of type $\alpha\rightarrow\beta$ is built up by taking a term of type β and abstracting it over a variable of type α." Following this intuition, it seems that what we want as the type of the first argument is Var α rather than Term α:[2]

$$\texttt{lam}:\ \Delta\alpha{:}*.\ \Delta\beta{:}*.\ \texttt{Var}\ \alpha \rightarrow \texttt{Term}\ \beta \rightarrow \texttt{Term}(\alpha \rightarrow \beta)$$

Of course, now we need to know what a Var α is. One sensible choice might be that a variable is specified by its type and a numeric index (since we need a countable number of variables of each type):

```
indtype Var:*→* with
             v: Δα:*. Nat → Var α
```

[2]We could also try to work with this formulation as-is, following the path taken, for example, by LCF [22].

The complete definition of `Term` would then be:

```
indtype Term:*→* with
          var: Δα:*. Var α → Term α
          app: Δα:*. Δβ:*. Term(α → β) → Term α → Term β
          lam: Δα:*. Δβ:*. Var α → Term β → Term(α → β)
```

This solution corresponds to a "first-order" view of variables: data structures representing variables appear explicitly in the data structure representing a term. For example, the F_1 term

```
λx:Nat. x
```

might be represented in this formulation by the `Term`:

```
lam [Nat] [Nat] (v [Nat] 5) (var [Nat] (v [Nat] 5))
```

Exercise 5.5.1: How would `λf:Nat→Nat. λa:Nat. f a` be represented in this formulation of `Term`?

Appendix A carries out the details of this kind of representation of *untyped* λ-calculus terms. Unfortunately, the approach used there apparently cannot be extended to the representation of typed terms. The most important operation on this representation is the substitution of one `Term` for a variable of the same type in another `Term`. But in order to decide which variables to replace, it is necessary to decide whether an *arbitrary* var node is equal to the variable being substituted, which requires an equality test on elements of `Var`. This test seems not to be implementable in F_ω. It is easy to implement a test for equality between two elements of most inductive types. But a *polymorphic* equality test, or equivalently a test for "run time" equality between types, cannot be added to F_ω without losing the strong normalization property.[3]

In both of these representations of F_1 terms, the F_1 types are subsumed as F_3 types—there is no explicit encoding of types as a data structure in their own right. Another formulation of `Term` would depend not only on a `Var` inductive type, but also on an inductive type called `Type`, perhaps with constructors `tvar` and `arrow`. This idea is pursued further at the end of Appendix A.

There are a number of properties that might be desirable in a representation of terms. The three representations above have different combinations of these properties:

Definability of `eval`. One of the interesting things to do with a data structure representing a term is to evaluate it to produce a value in the metalanguage. The higher-order representation of terms has this property—indeed, so little work is involved in defining `eval` that one almost feels a little cheated. We do not know how to define this kind of evaluation function for either of the first-order representations.

[3] We are grateful to Thierry Coquand and Christine Paulin-Mohring for pointing this out. The result is due to Girard [20].

Definability of substitution (and hence β-reduction). As far as we know, there is no way to define substitution for the higher-order representation of terms. The typed first-order representation requires an equality test for elements of `Var`, which we do not know how to provide. The first-order representation with explicitly encoded types clearly admits a substitution function.

Inability to represent ill-typed terms. One of the most important properties of both typed representations of terms is that it is impossible to construct a `Term` data structure corresponding to a poorly typed term. This eliminates the need to write typechecking functions for represented terms.

Ability to represent the whole metalanguage. Conventional wisdom holds that it is not possible to write a metacircular interpreter in a typed language. For some more exotic type systems—systems with reflexive types [36] or where "Type" is a type [6,35,37]—no complete story has been told.

Uniqueness of representation. Because of the `rep` constructor in the higher-order term representation, there are in general many `Terms` representing a given term. For the typed first-order representation, it is easy to see that there is an exact correspondence between terms and `Terms`. For the first-order representation with the encoding of types in the data structure, the situation in general is turned around: when this representation scheme is extended to F_3, there may again be many `Terms` corresponding to a given term (since it is possible to represent non-normalized type expressions in the `Term` data structure.)

Chapter 6

The ω-order Polymorphic λ-Calculus

We are finally ready to close off our hierarchy of languages by defining F_ω, the ω-order polymorphic λ-calculus, and discussing some of its properties.

6.1 Basic Definitions

The language F_ω differs from F_3, F_4, *etc.* only in that the set of legal kinds is larger:

Definition 6.1.1: The syntax of F_ω is given by the following inductively defined classes:

$$\begin{array}{lll} \text{(kinds)} & \mathtt{K} & ::= \; * \mid \mathtt{K}\rightarrow\mathtt{K}' \\ \text{(types)} & \mathtt{T} & ::= \; \alpha \mid \mathtt{T}\rightarrow\mathtt{T}' \mid \Delta\alpha:\mathtt{K}.\,\mathtt{T} \mid \lambda\alpha:\mathtt{K}.\,\mathtt{T} \mid \mathtt{T}\,\mathtt{T}' \\ \text{(terms)} & \mathtt{e} & ::= \; \mathtt{x} \mid \lambda\mathtt{x}:\mathtt{T}.\mathtt{e} \mid \mathtt{e}\,\mathtt{e}' \mid \Lambda\alpha:\mathtt{K}.\,\mathtt{e} \mid \mathtt{e}[\mathtt{T}] \end{array}$$

where K ranges over kinds, α ranges over type variables, T ranges over types, e ranges over expressions, and x ranges over variables.

Now we can finally list a full set of type and kind inference rules for F_ω, in much the same way as we have in previous chapters. The first three rules deal with the well-formedness of environments, and are named in the same way as before. The next five rules deal with the well-kindedness (well-formedness) of types. The base case (Tvar) deals with the kind of a type variable, while the other four, whose names begin with "WF," deal with the four ways that types may be built up. The next five rules deal with the correct typing of terms. The base case (Var) ensures that the type of a variable is of kind $*$. The other four deal with cases where $\rightarrow$ or Δ is introduced or eliminated at the type level, and are named accordingly. Finally, the ($\approx$) rule allows two types that are $\beta\eta$-convertible to be considered equal. There are two new notions in these rules: $\mathtt{T} \approx \mathtt{T}'$ indicates that the type expressions T and T$'$ are $\beta\eta$-convertible, and $\mathtt{T}'[\alpha/\mathtt{T}]$ is the result of replacing all occurrences of the type variable α in T' with T, renaming bound variables in T' to avoid capture of free variables in T. In both cases it is necessary to respect the binding constructs λ and Δ in a type expression. Note that the rules are designed so that for any term e, where $e \in T$ with T some type expression, it is assured that T is actually of kind $*$. This is usually left to the (Var) rule, but in the case of the ($\approx$) rule it must be written explicitly because a new type expression T is being introduced, which might not be of the same kind as T' although the two are $\beta\eta$-convertible. Similarly, in the case of ($\to$I) a new type expression is being introduced, so we must check explicitly that it is of kind $*$. We adopt the convention that any two α-convertible terms or type expressions are considered identical. Furthermore, we take "α is not free in Π" to mean that α is not free in any type expression assigned by Π. (The problem is that α may be free in $\Pi(\beta)$ in which case rebinding α might introduce an inconsistency. The same is not true of term variables since x can not appear in the range of Π.)

Here are the typing rules:

(ENV-$\langle\rangle$)	$\overline{wf(\langle\rangle)}$	
(ENV-term)	$\dfrac{\Pi \vdash T \in *}{wf(\Pi[x/T])}$	
(ENV-type)	$\dfrac{wf(\Pi)}{wf(\Pi[\alpha/K])}$	when α is not free in Π
(Tvar)	$\dfrac{wf(\Pi)}{\Pi \vdash \alpha \in K}$	when $\Pi(\alpha) = K$
(WF-$\to$)	$\dfrac{\Pi \vdash T \in * \qquad \Pi \vdash T' \in *}{\Pi \vdash T \to T' \in *}$	
(WF-Δ)	$\dfrac{\Pi[\alpha/K] \vdash T \in *}{\Pi \vdash \Delta\alpha{:}K.T \in *}$	when α is not free in Π
(WF-λ)	$\dfrac{\Pi[\alpha/K] \vdash T \in K'}{\Pi \vdash \lambda\alpha{:}K.T \in K \to K'}$	when α is not free in Π
(WF-app)	$\dfrac{\Pi \vdash T \in K \to K' \qquad \Pi \vdash T' \in K}{\Pi \vdash T\,T' \in K'}$	
(Var)	$\dfrac{\Pi \vdash T \in *}{\Pi \vdash x \in T}$	when $\Pi(x) = T$
($\to$I)	$\dfrac{\Pi \vdash T \in * \qquad \Pi[x/T] \vdash e \in T'}{\Pi \vdash \lambda x{:}T.e \in T \to T'}$	
($\to$E)	$\dfrac{\Pi \vdash e \in T \to T' \qquad \Pi \vdash e' \in T}{\Pi \vdash e\,e' \in T'}$	
(ΔI)	$\dfrac{\Pi[\alpha/K] \vdash e \in T}{\Pi \vdash \Lambda\alpha{:}K.e \in \Delta\alpha{:}K.T}$	when α is not free in Π
(ΔE)	$\dfrac{\Pi \vdash e \in \Delta\alpha{:}K.T' \qquad \Pi \vdash T \in K}{\Pi \vdash e[T] \in T'[\alpha/T]}$	
($\approx$)	$\dfrac{\Pi \vdash e \in T' \qquad T \approx T' \qquad \Pi \vdash T \in *}{\Pi \vdash e \in T}$	

We now show how the above inference rules are applied by checking the type judgment $t \in \texttt{Nat}$, where:

$$t \;\equiv\; (\Lambda\Theta{:}* \to *.\ \Lambda\alpha{:}*.\ \lambda x{:}\Theta\alpha.\ x)\ [\lambda\sigma{:}*.\sigma]\ [\texttt{Nat}]\ 5$$

For notational convenience we use the following abbreviations:

$$\begin{aligned}
\mathtt{t}_5 &\equiv \lambda \mathtt{x}{:}\Theta\alpha.\ \mathtt{x}\\
\mathtt{t}_4 &\equiv \Lambda\alpha{:}*.\ \mathtt{t}_5\\
\mathtt{t}_3 &\equiv \Lambda\Theta{:}*\to*.\ \mathtt{t}_4\\
\mathtt{t}_2 &\equiv \mathtt{t}_3\ [\lambda\sigma{:}*.\sigma]\\
\mathtt{t}_1 &\equiv \mathtt{t}_2\ [\mathtt{Nat}]
\end{aligned}$$

and hence:

$$\mathtt{t} \equiv \mathtt{t}_1\ \mathtt{5}$$

At each step of the proof we give the name of the inference rule used, together with the numbers of the lines of the proof used as premises (in order). When one line follows from another by application of the above abbreviations we say that it follows "by definition." Additionally, for brevity we allow ourselves to assume that $\mathtt{Nat} \in *$ and that $\mathtt{5} \in \mathtt{Nat}$, and we omit the sections of the proof dealing with the well-formedness of environments.

(1)	$\langle(\Theta,*\to*),(\alpha,*)\rangle \vdash \alpha \in *$	by (Tvar)
(2)	$\langle(\Theta,*\to*),(\alpha,*)\rangle \vdash \Theta \in *\to*$	by (Tvar)
(3)	$\langle(\Theta,*\to*),(\alpha,*)\rangle \vdash \Theta\alpha \in *$	by (WF-app) from 2,1
(4)	$\langle(\Theta,*\to*),(\alpha,*),(\mathtt{x},\Theta\alpha)\rangle \vdash \alpha \in *$	by (Tvar)
(5)	$\langle(\Theta,*\to*),(\alpha,*),(\mathtt{x},\Theta\alpha)\rangle \vdash \Theta \in *\to*$	by (Tvar)
(6)	$\langle(\Theta,*\to*),(\alpha,*),(\mathtt{x},\Theta\alpha)\rangle \vdash \Theta\alpha \in *$	by (WF-app) from 5,4
(7)	$\langle(\Theta,*\to*),(\alpha,*),(\mathtt{x},\Theta\alpha)\rangle \vdash \mathtt{x} \in \Theta\alpha$	by (Var) from 6
(8)	$\langle(\Theta,*\to*),(\alpha,*)\rangle \vdash \lambda\mathtt{x}{:}\Theta\alpha.\mathtt{x} \in \Theta\alpha\to\Theta\alpha$	by ($\to$I) from 3,7
(9)	$\langle(\Theta,*\to*),(\alpha,*)\rangle \vdash \mathtt{t}_5 \in \Theta\alpha\to\Theta\alpha$	by definition from 8
(10)	$\langle(\Theta,*\to*)\rangle \vdash \Lambda\alpha{:}*.\mathtt{t}_5 \in \Delta\alpha{:}*.\Theta\alpha\to\Theta\alpha$	by (ΔI) from 9
(11)	$\langle(\Theta,*\to*)\rangle \vdash \mathtt{t}_4 \in \Delta\alpha{:}*.\Theta\alpha\to\Theta\alpha$	by definition from 10
(12)	$\vdash \Lambda\Theta{:}*\to*.\mathtt{t}_4 \in \Delta\Theta{:}*\to*.\Delta\alpha{:}*.\Theta\alpha\to\Theta\alpha$	by (ΔI) from 11
(13)	$\vdash \mathtt{t}_3 \in \Delta\Theta{:}*\to*.\Delta\alpha{:}*.\Theta\alpha\to\Theta\alpha$	by definition from 12
(14)	$\langle(\sigma,*)\rangle \vdash \sigma \in *$	by (Tvar)
(15)	$\vdash \lambda\sigma{:}*.\sigma \in *\to*$	by (WF-λ) from 14
(16)	$\vdash \mathtt{t}_3\ [\lambda\sigma{:}*.\sigma] \in \Delta\alpha{:}*.(\lambda\sigma{:}*.\sigma)\alpha\to(\lambda\sigma{:}*.\sigma)\alpha$	by (ΔE) from 13,15
(17)	$\langle(\alpha,*)\rangle \vdash \alpha \in *$	by (Tvar)
(18)	$\langle(\alpha,*)\rangle \vdash \alpha\to\alpha \in *$	by (WF-$\to$) from 17,17
(19)	$\vdash \Delta\alpha{:}*.\alpha\to\alpha \approx \Delta\alpha{:}*.(\lambda\sigma{:}*.\sigma)\alpha\to(\lambda\sigma{:}*.\sigma)\alpha$	by $\beta\eta$-conversion of types
(20)	$\vdash \Delta\alpha{:}*.\alpha\to\alpha \in *$	by (WF-Δ) from 18
(21)	$\vdash \mathtt{t}_3\ [\lambda\sigma{:}*.\sigma] \in \Delta\alpha{:}*.\alpha\to\alpha$	by ($\approx$) from 16,19,20
(22)	$\vdash \mathtt{t}_2 \in \Delta\alpha{:}*.\alpha\to\alpha$	by definition from 21
(23)	$\vdash \mathtt{Nat} \in *$	by assumption
(24)	$\vdash \mathtt{t}_2\ [\mathtt{Nat}] \in \mathtt{Nat}\to\mathtt{Nat}$	by (ΔE) from 22,23
(25)	$\vdash \mathtt{t}_1 \in \mathtt{Nat}\to\mathtt{Nat}$	by definition from 24
(26)	$\vdash \mathtt{5} \in \mathtt{Nat}$	by assumption
(27)	$\vdash \mathtt{t}_1\ \mathtt{5} \in \mathtt{Nat}$	by (ΔE) from 26,25
(28)	$\vdash \mathtt{t} \in \mathtt{Nat}$	by definition from 27

Exercises 6.1.2:

Use the deduction rules for F_ω to prove that

`cons [Nat] 1 (nil [Nat] 0)` $\in$ `List Nat`

(Warning: This will take quite a lot of work.)

The languages F_1, F_2, *etc.* can now be defined as restrictions of F_ω.

Definition 6.1.3: The kind $*$ has order 1. If the greater of the orders of K and K' is i, then $K \rightarrow K'$ has order $i+1$.
The n^{th}-order polymorphic λ-calculus (F_n) consists of those terms of F_ω for which types can be derived using the above rules without mentioning any kinds of order greater than or equal to n.

6.2 Properties of F_ω

We have discussed various properties of the languages in our hierarchy of typed λ-calculi. To summarize, we can state the following theorems for F_ω:[1]

Theorem 6.2.1:

1. If $\Pi \vdash \mathbf{e} \in \alpha$ then $\Pi \vdash \alpha \in *$.
2. If $\Pi \vdash T \in K$ then T has a unique $\beta\eta$-normal form.
3. If $\Pi \vdash \mathbf{e} \in T$ then $\mathbf{e}$ has a unique $\beta\eta$-normal form.
4. $\Pi \vdash \mathbf{e} \in T$ is decidable.

Some other interesting properties of F_ω are:

- $F_\omega^i = F_\omega$. (Inductive data type definitions whose constructors have types in F_ω can be translated into closed type expressions in F_ω [45].)
- F_2 can express every function which is provable total under second-order peano arithmetic [18,16]. In general, F_n can express all functions whose totality is provable in *nth*-order arithmetic [19].
- Typed λ-terms can be extracted from proofs in higher-order logic [11,47,43]. A program extracted from a proof in n^{th}-order logic will be typable in F_n.

[1]Strictly speaking, these are conjectures: they have not been proved for the formulation of F_ω that we are using. Our system is intended not as an object of study in itself but as a concrete way of talking about other systems. For Girard's system [20,18], the appropriate analogues of these statements *are* theorems.

6.3 Types and Type Inference

In order to fit F_ω into the "big picture," we consider its relation to other programming languages along several dimensions.

Implicit vs. Explicit Types. We have remarked that F_ω is ***explicitly*** rather than ***implicitly*** typed: polymorphism is represented by explicit type abstraction (Λ), and polymorphic functions must be applied explicitly to a type argument (`[]`) yielding a monomorphic instance, which can in turn be applied to term arguments (or further type arguments). In contrast, types do not appear in the concrete syntax of a purely implicitly typed language (although they may eventually be derived). We use the terms ***explicit polymorphism*** (or explicit typing) and ***implicit polymorphism*** to denote polymorphism in an explicitly or implicitly typed language, respectively.

Explicit typing offers several advantages over implicit in that types may be given a greater role than just ensuring the well-formedness of terms. Although an implicitly typed program may be typable (i.e., well-formed), it may not be the one the author intended. Explicit typing adds an additional level of insurance that the writer's intentions are realized. Explicit types also serve as formal documentation and can increase the readability of programs.

In reality, explicitly and implicitly typed languages represent a continuum: at one end is a language with no types in its concrete syntax (purely implicit); at the other, a language in which every term has an associated syntactic type (purely explicit). Although we have classified F_ω as explicitly typed, not all terms have explicit types in the concrete syntax. In fact, it is only the variables bound by λ-abstractions that are given explicit types. Nevertheless, F_ω remains explicitly typed because the types of the remaining terms may easily be deduced from the type information provided (see Section 6.1). In general, ***type inference*** is the process of determining missing type information. Although this involves only minimal work in the case of F_ω, type inference is also applicable to implicitly typed languages in which more or all of the type information is missing. Terms before type inference may be ***partially-typed*** (under explicit polymorphism) or ***untyped*** (under implicit polymorphism), while the results of successful type inference are ***fully typed***.

A downside of explicit typing is that programs may become so verbose as to be unintelligible. In our development of `ack` (Section 3.3) one of the steps appeared as follows:

```
((λn₃:Nat. (succ n₃) [Nat] 1 succ)
    ((λn₂:Nat. (succ n₂) [Nat] 1 succ)
        3))
```

The problem with writing the above expression is that each instance of `Nat` may be determined by context: the iterations produce `Nat`s because their result is a combination of 1 and `succ`, while the iterations are over `Nat`s because `succ` is applied first. A remedy to the redundancy of explicit types is to omit type information when it may be inferred. The equivalent term without the redundant type information is

```
((λn₃. (succ n₃) [] 1 succ)
   ((λn₂. (succ n₂) [] 1 succ)
      3))
```

The task of expanding a partially-typed, or "type-elided", term into F_ω is called *partial* type inference. Enough type information must be included, however, so that "proper" typings are recreated in the process. (This is further developed below.)

Curry vs. Church Types. This leads us to a more fundamental difference between typed languages: What is the role of types? The first or "Curry" view of types is that terms are initially untyped and types serve merely to group (structurally or behaviorally) related terms [13]. Central to the Curry view is that untyped terms are meaningful and may be given a semantics without regard to any associated type. In fact, syntactic and semantic properties of the language (e.g., reduction) are formulated without types. Generally under such an approach, types semantically characterize terms, perhaps by denoting components of a mathematical model (e.g., a set). Types which *do* appear in the syntax of the language are viewed as meta-syntactic; the ':' is interpreted semantically (e.g., a predicate establishing membership within a set). Type inference is viewed as the process of determining where a term lives, that is, what is the class (type) of its semantic denotation.

In contrast, the "Church" view treats untyped terms as meaningless [10]. Types, in the Church view, are an integral part of the language. Indeed, the reduction rules are formulated with types. Similarly, type inference is defined syntactically. Each of the typed λ-calculi we have described fall into this school. Of course like Curry, the Church approach admits denotational semantics, but whereas a typical "Curry semantics" consists of one large domain containing the denotations of all terms, in a "Church semantics" there will be a family of domains indexed by types.

In the literature, the reader may find the Curry view associated with implicit types and the Church view with explicit. This is because of the importance of untyped terms within the Curry approach and of typed terms under Church. However, we wish to emphasize that they are independent notions: we might imagine a concrete syntax containing types which are all viewed meta-syntactically (Curry), or a concrete syntax initially free of types, but for which type inference generates a syntactically complete, fully typed term (Church). Curry vs. Church is primarily a philosophical issue, while explicit vs. implicit is a question of language syntax and implementation. (For further insight on these issues, see Mitchell and Harper [40].)

Principle Types and Milner vs. Girard/Reynolds Polymorphism. We have suggested that a type inference algorithm assigns at most one fully-typed term for a given partially or untyped one. However, this is usually not possible. Existing implicitly polymorphic languages (notably ML) admit complete type inference algorithms [5,38] that yield "principle" rather than unique types. A *principle* type is a type (or type scheme) from which all other valid typings may be derived by substitution.

The "Milner style" polymorphism of ML limits occurrences of universal quantification (our Δ) to the top-level or outermost scope of the type. In contrast, the more expressive "Girard/Reynolds" polymorphism of our calculi does not so constrain types. Milner's restriction disallows the passing of polymorphic functions as parameters, since nested Δ's are required to represent the type of the polymorphic argument. In fact, it is this restriction which makes the way for principle types. Consider the function `double` of Section 3. From an untyped version

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

type inference might derive either the original

$$\Lambda\alpha{:}*.\ \lambda\texttt{f}{:}\alpha\rightarrow\alpha.\ \lambda\texttt{x}{:}\alpha.\ \texttt{f(f x)}$$

of type

$$\Delta\alpha.(\alpha\rightarrow\alpha)\rightarrow\alpha\rightarrow\alpha$$

or instead

$$\lambda\texttt{f}{:}(\Delta\alpha{:}*.\alpha\rightarrow\alpha).\ \Lambda\beta{:}*.\ \lambda\texttt{x}{:}\beta.\ \texttt{f}\ [\beta]\ (\texttt{f}\ [\beta]\ \texttt{x})$$

which has type

$$(\Delta\alpha{:}*.(\alpha\rightarrow\alpha))\rightarrow\Delta\beta{:}*.\beta\rightarrow\beta$$

The second version takes a polymorphic function as an argument, and so is not permitted in a Milner style. That the above types are incomparable means that the richness of Girard/Reynolds polymorphism does not admit principle types. Furthermore, the decidability of type inference mapping untyped λ-calculus into F_2—that is, full type inference for F_2—remains open [33,39].

The above ambiguity for the untyped `double` may be removed by including empty type applications in the appropriate positions, thereby steering partial type inference toward the desired typing. However, this still does not yield principle types. The expressiveness inherent in the higher-order nature of Girard/Reynolds polymorphism complicates type inference: that types may be functions requires higher-order unification to determine the intended types. In fact, partial type inference [3,41] may be more expensive than full type inference. Pfenning has shown that partial type inference for the n^{th}-order λ-calculus (in which type applications are omitted but placeholders are left to show where they must appear) is equivalent to n^{th}-order unification, which is undecidable for $n \geq 2$ [21]. However, Huet's [30] algorithm has proven tractable for realistic problems, and using this, Pfenning describes a partial type inference semi-algorithm for F_ω [46].

6.4 F_ω as the Basis for a Programming Language

We have shown in this tutorial that F_ω can be used to express a surprisingly broad range of computations. Still, it falls short in many ways as a practical programming language. Pfenning and Lee [49] have studied a number of extensions to F_ω that either make it more convenient without adding to its power or disrupting its desirable theoretical properties, or strictly increase its power:

- As discussed in the previous section, a good type inference system can enormously reduce the redundancy and verbosity of programs in polymorphic λ-calculi.
- Most functional languages include some form of general recursive function definitions. Although we have left it out of our formulation of F_ω—partly to highlight the interesting programming style that arises without it and partly to preserve the theoretical properties of Girard's system—there is no reason why it cannot be added.
- Some "functional-style" languages also include imperative features like updateable references and powerful control constructs like exceptions or a "call with current continuation" operator. It appears possible to add these to F_ω as well.

Pfenning and Lee's work is currently focused on a family of languages called LEAP (a "Language with Eval and Polymorphism"). In addition to some of the ideas mentioned above, key features include:

- An extension of F_ω's polymorphism that makes it possible to write an `eval` function for an inductive representation of all of F_ω (something that does not appear to be possible in F_ω itself). Global definitions in LEAP are not considered to be simple abbreviations (like the ones in this tutorial), but rather global `let` bindings. In particular, kind variables are allowed to appear free in the right hand side of a global definition, and these variables are considered generic in exactly the sense of ML's generic type variables introduced by `let` statements [15]—they can be instantiated to different kinds at each point where the global definition is used. This introduces enough "additional polymorphism" to allow the definition of an `eval` function for a representation of terms essentially the same as that given in Section 5.3:

  ```
  eval  ≡  Λα:*. λt:Term α.
               iterTerm[λσ:*.σ] t
                   (Λσ:*. λu:σ. u)
                   (Λσ:*. Λτ:*. λu:σ→τ. λv:σ. uv)
                   (Λσ:*. Λτ:*. λu:σ→τ. u)
                   (ΛΘ:K→*. λu:(Δα:K. Θα).u)
                   (ΛΘ:K'→*. λu:(Δα:K'. Θα).Λα. u[α])
  ```

  Every time `eval` is used in a program, the appropriate values of `K` and `K'` are computed by type/kind inference. Looking at the earlier definition of `eval` for F_3 in terms of F_4, it is clear that the above is a schema for a definition that would otherwise have to be infinite.

- Even with a good partial type inference algorithm, the amount of syntactic "noise" introduced by placeholders for type applications can be substantial. Pfenning and Lee have introduced a very useful shorthand, which they call "star syntax." When an identifier is declared on the left hand side of a `let`, it may be annotated with some number of $*$'s to indicate how many type parameters it expects. Now when the defined variable is used (unstarred) elsewhere in the program, the correct number of empty type applications are inserted after it. These empty applications are then filled by the partial type inference algorithm, as before.

Appendix A

Representing the Untyped λ-Calculus

This appendix presents a representation of the untyped λ-calculus in F_2. Because it is homogeneous (in the sense discussed in Section 5.1) the untyped λ-calculus should be easier to represent than the typed λ-calculi. This makes it a good starting point for developing techniques, tools, and intuitions that may perhaps be applied to the more difficult situations. It also provides a good extended programming example.

We consider the simplest formulation of the untyped λ-calculus, with just variables, λ-abstractions, and applications:

```
indtype Term:* with
    var: Var → Term
    app: Term → Term → Term
    lam: Var → Term → Term
```

where natural numbers are used to represent variables:

```
Var  ≡  Nat
```

Our first task is to implement substitution. This turns out to require primitive recursion rather than simple iteration, because we need to be able to inhibit substitution in the scope of a λ whose bound variable is the same as the one being substituted.

The iterator `iterTerm[Pair_Term]` takes a `Term` to a pair of `Terms`. The first projection of the resulting pair is the substituted term; the second projection is the original term. (In the following, we omit some type tags when it is clear from context how they should be filled in. For example, write `ife` instead of `ife_Term`.)

```
subst₁  ≡  λt:Term. λv:Var. λs:Term.
        fst(iterTerm[Pair_Term] t
               (λx:Var.
                    pair
                       (ife (= x v) s (var x))
                       (var x))
               (λr:Pair_Term. λs:Pair_Term.
                    pair
                       (app (fst r) (fst s))
                       (app (snd r) (snd s)))
               (λx:Var. λr:Pair_Term.
                    pair
                       (ife (= x v)
                             (lam x (snd r))
                             (lam x (fst r)))
                       (lam x (snd r)))
           )
```

This version is not quite correct, however: it allows capture of free variables in **s** by bound variables in **t**. To use it safely, we need to first make sure that no such capture can occur, by α-converting **t** so that its bound variables all have greater indices than any variable used in **s**.

Exercise A.1: Define a function `maxvar` $\in$ `Term→Var` that calculates the maximum index of the variables used in its argument.

Taking `maxvar` as given, we can write:

```
subst  ≡  λt:Term. λv: Var. λs: Term.
            subst₁ (alphaconv t (maxvar s)) v s
```

To define **alphaconv**, we need to be able to keep track of the set of variables bound by enclosing λs so that we can decide whether or no' to change the index of a variable when we come to it. But the iteration construct makes this hard: since it starts at the leaves of the term and builds up the result, we see the variable at the leaf "before" we get to any enclosing λs. The trick is to have the iteration return not a term, but rather a function from variables to terms. ("You show me what the bound variables are and I'll show you what the α-converted term is.")

```
alphaconv  ≡  λt:Term. λd:Nat.
               (iterTerm[(Var→Bool)→Term] t
                   (λx:Var. λb:Var→Bool.
                       ife (b x)
                           (var (plus x d))
                           (var x))
                   (λr:(Var→Bool)→Term. λs:(Var→Bool)→Term.
                       λb:Var→Bool.
                           app (r b) (s b))
                   (λx:Var. λr:(Var→Bool)→Term. λb:Var→Bool.
                       lam (plus x d)
                           (r (λz:Var.
                                  ife (= z x)
                                      true
                                      (b z))))
              ) (λx:Var. false)
```

With substitution under control, it is now fairly easy to define β-reduction on terms. Of course, we cannot hope to completely normalize an arbitrary term of the untyped λ-calculus since any attempt to do so may diverge, which would contradict the strong normalization property of the metalanguage. The following pair of functions implements one-step β-reduction of all redices in a term, from bottom to top. (It should also be possible to reduce just one redex at a time, as defined in Section 2.1, but this seems harder than reducing several at once.)

The iteration in the first function does not alter the `Term` except at `app` nodes, which it hands off to the second function. This, in turn, does nothing unless the top-level constructor of the term on the left hand side of the application is a λ abstraction, in which case it uses `subst` to reduce the application to a substitution instance of the body of the λ.

```
beta  ≡  λt:Term.
            iterTerm[Term] t
                (λx:Var. var x)
                (λr:Term. λs:Term.
                     betalam r s)
                (λx:Var. λr:Term.
                     lam x r)
```

```
betalam  ≡  λt:Term. λu:Term.
                fst (iterTerm[Pair_Term] t
                        (λx:Var.
                            pair (app t u)
                                 (var x))
                        (λr:Pair_Term. λs:Pair_Term.
                            pair (app t u)
                                 (app (snd r) (snd s)))
                        (λx:Var. λr:Pair_Term.
                            pair (subst (snd r) x u)
                                 (lam x (snd r)))
                    )
```

This can easily be extended to perform any finite number of "simultaneous" β-reductions:

```
multibeta  ≡  λn:Nat. λt:Term.
                  iterNat[Term] n t beta
```

Exercises A.2:

1. Define an encoding of type expressions in F_1 as an inductive type.
2. Extend the definition of `Term` above to a representation of F_1, where types appear explicitly in the data structure (rather than being represented as types in the metalanguage as in Chapter 5).
3. Define substitution and β-reduction for this representation of F_1.
4. (Difficult.) Define a typechecker for this representation.

Appendix B

Symbols and Terminology

Notational Conventions

SYMBOL	USAGE	STYLE
x, y, map, car, ...	term variables and constants	l.c.
List, Bool, Nat, ...	globally-defined types	u.c.
α, β, θ, σ, ...	bound type variables of kind *	l.c. greek
Φ, Ψ, Θ	bound type variables of higher kind	u.c. greek
γ	result type of an iteration	
Γ	result type of a higher-kind iteration	
pair, cons, succ	globally-defined constructors	l.c. spelled-out
p, c, n, s	λ-bound placeholders for constructors	l.c. first letter
K	kind variable	

Symbols

SYMBOL	MEANING	SEE PAGE(S)
$\equiv$	global definitions	6
λx:T.e	term-to-term abstraction	6,6,12,21,34,50
ee'	term-to-term application	6,6,21,34,50
T$\rightarrow$T'	type of a term-to-term abstraction	6,6,12,21,34
F_1	the simply-typed λ-calculus	6
F_1^+	F_1 with primitive types and iteration	6
$\Pi[x/\alpha]$	extension of Π with the type of a variable	7
$\Pi \vdash e \in T$	type judgment	7
$=_\beta$	β-equivalence	8

$=_{\beta\eta}$	$\beta\eta$-equivalence	8
$\Rightarrow_{\beta}$	β-reduction	8
$\Rightarrow_{\eta}$	η-reduction	8
$\Rightarrow_{\beta\eta}$	$\beta\eta$-reduction	8
$\mathbf{e}[x/\mathbf{a}]$	term substitution	8
F_1^i	F_1 with inductive type definitions	12
`iterT[V]`	iteration	12,12
$\Lambda\alpha$:K.T	type-to-term abstraction	19,21,34,50
$*$	the kind of types	21,34,50
$\Delta\alpha$:K.T	universal type	20,21,34,50
$\mathbf{e}[\alpha]$	term-to-type application	21,34,50
F_2	second-order typed λ-calculus	21
F_2^i	F_2 with inductive type definitions	33,41
F_3	third-order typed λ-calculus	34
$\Pi[\alpha/K]$	extension of Π with the kind of a type variable	35
$\Pi \vdash T \in K$	kind judgment	35
$\lambda\alpha$:K.T	type-to-type abstraction	34,50
T T$'$	type-to-type application	34,50
$K \rightarrow K'$	kind of a type-to-type abstraction	34
$[\![\mathbf{e}]\!]_{\Pi}$	syntactic sugar for meta-representation	42
F_4	fourth-order typed λ-calculus	46
F_{ω}	ω-order typed λ-calculus	50
F_{ω}^i	$F_{\omega}$with inductive type definitions	50
$\approx$	$\beta\eta$-equivalence of type expressions	50
$T[\alpha/T']$	type substitution	50
LEAP	the LEAP programming language	56

Type and Kind Deduction Rules

RULE NAME	PURPOSE	SEE PAGE(S)
(Succ)	check type of use of `succ` in F_1^+	8
(Zero)	check type of use of `zero` in F_1^+	8
(Iter)	check type of use of `iter` in F_1^+	8
(Var)	checks that the type of a variable is of kind $*$	8,22,51
($\rightarrow$I)	$\rightarrow$ introduction in type of a term	8,22,51
($\rightarrow$E)	$\rightarrow$ elimination in type of a term	8.22,51
(Tvar)	checks the kind of a type variable	22,51
(ENV-$\langle\rangle$)	asserts than an emty environment is well-formed	22,51
(ENV-term)	checks well-formedness of term/type pair in an environment	22,51
(ENV-type)	checks well-formedness of type/kind pair in an environment	22,51
(WF-$\rightarrow$)	checks that $\rightarrow$ types are well-formed	22,51
(WF-Δ)	checks that Δ types are well-formed	22,51
(ΔI)	Δ introduction in type of a term	22,51
(ΔE)	Δ elimination in type of a term	22,51
(WF-λ)	checks that λ types are well-formed	51
(WF-app)	checks that a type-to-type application is well-formed	51
($\approx$)	$\beta\eta$-conversion for type expressions	51

Example Types

NAME	PURPOSE	SEE PAGE(S)
`Nat`	natural numbers	6,12,26
`Bool`	booleans	15,24
`Void`	type without any elements	15,23
`Unit`	type with exactly on element modulo α-equivalence	15,24
`Pair_Nat`	pair of natural numbers	16
`List_Nat`	list of natural numbers	16
`Pair_List_Nat`	pair of lists of natural numbers	17
`Tree`	polymorphic tree	17
`Pair`	polymorphic pairs	36
`List`	polymorphic lists	38
`Term`	λ-calculus terms	41

Example Functions

NAME	PURPOSE	SEE PAGE(S)
iter	F_1^+ iteration on naturals	6,10
zero	zero element of naturals	6,12,26
succ	successor function on naturals	6,12,26
plus	addition of natural numbers	11,12,12,28
times	multiplication of natural numbers	11,28
unit	unique object of type Unit	15,24
true, false	boolean constants	15,24
pair_Nat	pair of natural numbers (in F_1^i)	16
fst_Nat	first of a pair of natural numbers	16
snd_Nat	second of a pair of natural numbers	16
nil_Nat	empty list of natural numbers (in F_1^i)	16
cons_Nat	cons of a natural number and a list of naturals	16
car_Nat	car of a list of naturals	16
cdr_Nat	cdr of a list of naturals	16
pair_List_Nat	pair of lists of naturals	16
empty	polymorphic leaf node of a tree	17
node	polymorphic internal node of a tree	17
build	polymorphic function to construct trees	18
id	polymorphic identity	19
double	polymorphic double application	19
iterBool	iteration on booleans	25
not	boolean negation	24,26
zero?	test for zero on natural numbers	28
and, or	boolean conjunction/disjunction	26
iterNat	iteration on anturals	27
primrec	primitive recursion	30
pair	polymorphic pairing	36
fst	polymorphic first of a pair	36
snd	polymorphic second of a pair	36
nil	polymorphic empty list	38
cons	polymorphic list constructor	38
car	polymorphic list car	38
cdr	polymorphic list cdr	38
rep	representation fucntion for meta-programming	41
app	meta-level term application	41
eval	evaluation function for metaprogramming	43
lam	meta-level λ-abstraction	44
typapp	meta-level type application	45
typlam	meta-level type abstraction	45
typapp1, typapp2	F_2 and F_3 type application in F_4	46
typlam1, typlam2	F_2 and F_3 type abstraction in F_4	46

Concepts

CONCEPT	SEE PAGE(S)
simply-typed lambda calculus	6
λ-abstraction	6
application	6
scope	7
body	7
bound	7,21
free	7,21
type judgment	7
extension	7
inference rules	8
β-redex,β-reducible,β-reduction,β-conversion	8
η-redex,η-reducible,η-reduction,η-conversion	8
$\beta\eta$-reducible,$\beta\eta$-reduction,$\beta\eta$-conversion	8
$\beta\eta$-normal form	8
equivalence	8
α-equivalent	9
α-convertible	9
Strong Normalization Theorem	9
Church-Rosser Theorem	9
currying	11
inductive type definitions	11
global environment	13
positive occurrence	13,35
negative occurrence	35,35
underscore notation	15
polymorphism	20
polymorphic λ-calculus	20
term abstraction	21
type abstraction	21
term application	21
type application	21
closed	21
primitive recursion	29
type constructor	33
meta-programming	41
homogeneous, nonhomogeneous	42

type checking	50
explicit types	54
implicit types	54
explicit polymorphism	54
implicit polymorphism	54
type inference	54
partially typed	54
untyped	54
fully typed	54
Curry view	55
Church view	55
principle types	55
Milner style	55
Girard/Reynolds style	56
partial type inference	55

Bibliography

[1] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, New York, 1985.

[2] Hendrik P. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, revised edition, 1984.

[3] Hans-J. Boehm. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*, pages 339–345, IEEE, October 1985.

[4] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[5] Luca Cardelli. Basic polymorphic typechecking. *Polymorphism Newsletter*, 1986.

[6] Luca Cardelli. *A Polymorphic λ-calculus with Type:Type.* Technical Report DECS-10, Digital Systems Research Center, May 1986.

[7] Luca Cardelli. Typeful programming. 1989. Unpublished.

[8] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[9] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, New Jersey, 1941.

[10] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[11] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[12] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[13] H. B. Curry and R. Feys. *Combinatory Logic.* North-Holland, Amsterdam, 1958.

[14] H. B. Curry and R. Feys. *Combinatory Logic, Volume I.* North-Holland, Amsterdam, second edition, 1968.

[15] L. Damas and R. Milner. Principle type schemes for functional programs. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, ACM, 1982.

[16] Steven Fortune, Daniel Leivant, and Michael O'Donnell. The expressiveness of simple and second-order type structures. *Journal of the ACM*, 30:151–185, 1983.

[17] Jean H. Gallier. *On Girard's "Candidats de Reductibilité"*. 1988. Unpublished notes.

[18] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordere supérieur*. PhD thesis, Université Paris VII, 1972.

[19] Jean-Yves Girard. *Typed Lambda-Calculus*. April 1988. Draft book, translated by Paul Taylor and Yves Lafont.

[20] Jean-Yves Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, North-Holland Publishing Co., Amsterdam, London, 1971.

[21] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[22] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

[23] Robert Harper. *Introduction to Standard ML*. Technical Report ECS-LFCS-86-14, Laboratory for the Foundations of Computer Science, Edinburgh University, November 1986.

[24] Robert Harper. *Standard ML*. Technical Report ECS-LFCS-86-2, Laboratory for the Foundations of Computer Science, Edinburgh University, March 1986.

[25] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204, IEEE, June 1987.

[26] Robert Harper, David MacQueen, and Robin Milner. *Standard ML*. Technical Report ECS-LFCS-86-2, Laboratory for the Foundations of Computer Science, Edinburgh University, March 1986.

[27] Robert Harper, Robin Milner, and Mads Tofte. *The Semantics of Standard ML: Version 1*. Technical Report ECS-LFCS-87-36, Computer Science Department, University of Edinburgh, 1987.

[28] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-calculus*. Cambridge University Press, 1986. London Mathematical Society Student Texts: 1.

[29] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, Acadmeic Press, London, 1980.

[30] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[31] Gérard Huet. A uniform approach to type theory. 1988. Unpublished notes.

[32] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[33] Daniel Leivant. Polymorphic type inference. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1983.

[34] Daniel Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *Proccedings of the Twenty Fourth Annual Symposium on the Foundations of Computer Science*, pages 160–169, IEEE, 1983.

[35] Per Martin-Löf. *Hauptsatz* for the intuitionistic theory of iterated inductive definitions. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216, North Holland, Amsterdam, 1971.

[36] Paul Mendler. *Recursive Definition in Type Theory.* PhD thesis, Cornell University, 1987.

[37] Albert Meyer and Mark Reinhold. 'Type' is not a type: preliminary report. In *Proceedings of the 13th ACM Symposium on the Principles of Programming Languages*, 1986.

[38] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[39] John Mitchell. Type inference and type containment. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 257–277, Springer-Verlag LNCS 173, 1984.

[40] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):211–249, February/March 1988.

[41] John C. Mitchell. Second-order unification and types. June 1984. Unpublished notes.

[42] Bengt Nordström, Kent Petersson, and Jan Smith. Programming in Martin-Löf's type theory. 1988. Draft book.

[43] Christine Paulin-Mohring. Extracting $F\omega$'s programs from proofs in the calculus of constructions. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 89–104, ACM, January 1989.

[44] Rozsa Péter. *Recursive Functions.* Academic Press, New York, 1967.

[45] Frank Pfenning. *Inductively Defined Types in the Calculus of Constructions.* Ergo Report 88–069, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1988.

[46] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, ACM Press, July 1988.

[47] Frank Pfenning. *Program Development through Proof Transformation.* Ergo Report 87–047, Carnegie Mellon University, Pittsburgh, December 1987. Talk given at the *Workshop on Logic and Computation*, June 1987, Pittsburgh.

[48] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, ACM Press, June 1988. Available as Ergo Report 88–036.

[49] Frank Pfenning and Peter Lee. LEAP: a language with eval and polymorphism. In *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, Springer-Verlag LNCS, March 1989. To appear. Also available as Ergo Report 88–065.

[50] Jonathan Rees and William Clinger, editors. *Revised^3 report on the algorithmic language Scheme.* ACM, November 1986. In: SIGPLAN Notices 21(11).

[51] John Reynolds. An introduction to the polymorphic lambda calculus. In Gérard Huet, editor, *Logical Foundations of Functional Programming, Procedings of the Year of Programming Institute*, Addison-Wesley, 1988.

[52] John Reynolds. Three approaches to type structure. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Mathematical Foundations of Software Development*, pages 97–138, Springer-Verlag LNCS 185, March 1985.

[53] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, Springer-Verlag LNCS 19, New York, 1974.

[54] Richard Statman. Number theoretic functions computable by polymorphic programs. In *22nd Annual Symposium on Foundations of Computer Science*, pages 279–282, IEEE, October 1988.

[55] J. Steensgaard-Madsen. Typed representation of objects by functions. *ACM Transactions on Programming Languages and Systems*, 11(1):67–89, January 1989.