UNIVERSITY OF SUSSEX

COMPUTER SCIENCE

Notes on the Second-Order Lambda Calculus

Edmund Robinson

Abstract. An introduction to the theory of the second-order lambda calculus as it stands at present, concentrating on its use as a model for the type theories of programming languages and their related logical systems.

Notes for a Logic for IT weekend course
January 25-26, 1992
Leeds

School of Cognitive and Computing Sciences
University of Sussex
Brighton BN1 9QH

Email: edmundr@cogs.susx.ac.uk

Contents

Preface

The first version of these notes was written at the request of Harold Simmons. It was given out to students attending a weekend course on the polymorphic lambda calculus which he asked me to give under the auspices of the SERC "Logic for IT" initiative. This accounts for much of their character. The idea of the course was to provide an introduction to the area of type theory as seen through the spectacles of one particular formalism. There is by now a great deal of material in the area of polymorphic type theories. Some of it is extremely well known, and some, sometimes equally deserving, is not. The aim of these notes is to furnish an introduction to some of this material, not to provide a reference. Accordingly much of the material reflects my own personal bias about which problems are interesting or important, and much is quite sketchy. There are many places where the reader is referred elsewhere for a clearer more detailed account.

One area in which, by contrast, I have been more than usually specific is in my account of the syntax of the theory. This is partly because the syntax is often treated badly (sometimes to the point of error). But it is also because this calculus provides almost a minimal example of the use of multiple levels of contexts. And more particularly it is because there is a strong connection between the information conveyed in a well set up syntax, and the semantics of the theory. I hope the connection is made apparent here.

The notes are divided into six chapters. The first contains an introduction to the notion of polymorphism, and the syntax of the pure form of the second-order lambda calculus. The second chapter contains quite a detailed account of the extension of the theory to include pairing, and two different ("strict" and "lazy") translations back into the pure form of the theory. Chapter three contains an account of the implementation of certain common datatypes in the pure form of the calculus. This is one place where the material has been dealt with particularly extensively elsewhere, but I am more than usually specific about the limitations of the techniques. Chapter four contains a whirlwind account of the semantic theories available, concentrating on displaying the PER model in two different formalisms (this of course is an example of my personal bias coming through). The fifth chapter contains sketches of proofs of strong normalisation and the Church-Rosser theorems. Again, good clear accounts of these exist elsewhere. The final chapter does not contain any new formal material, but is by way of being a discussion of what has gone before, and something of a personal manifesto.

At this point, I would like to thank Francis Borceux and Pino Rosolini for inviting me to give an early version of the lecture course on which these notes are based at the Rixensart summer school on "Category Theory, Logic and Computer Science" in April 1991. I would also like to thank the ERASMUS program of the EC for providing funding.

These notes would not, however, have been written had I not been invited by Harold Simmons to repeat that course under the auspices of the "Logic for IT" initiative. This SERC-sponsored project was intended to raise the profile of logic in the UK research community. I would like to thank Harold and the others whose hard work over several years did a great deal to improve the lot of a generation of research students in this area. I am sorry that the program does not seem to have survived the loss of its SERC funding. Ironically, it fell a victim to its own success. As it is, I am proud to have been able to contribute to a unique series.

Finally, it will be apparent to any reader that a large part of these notes, particularly the second and third chapters, owes a great deal to the work of Gavin Wraith and his student, David Bosley. The part contained in Bosley's thesis is not yet publicly available. I hope it will be soon.

1 Introduction to Polymorphism

Type theories, Janus-like, have both a logical and a programming face. They are used to describe logical proof structures, and programming type structures. The type theory we are going to consider in this course was developed independently in both contexts. The historically prior invention is due to Jean-Yves Girard [Gir72, Gir71] who developed the calculus in connection with the proof theory of second-order intuitionistic logic, and called it "*Système F*". The same calculus (or rather a fragment of Girard's) was subsequently rediscovered by John Reynolds as part of an analysis of Strachey's notion of parametric polymorphism [Rey83].

Most of us are probably used to the idea of types in programming languages, and even to the notion of polymorphism there, as exemplified in languages such as ML, Miranda[1] and Haskell. However, it is worth pausing for a moment to refresh our thoughts. Types have more than one function in programming languages. From the user's point of view they are there to help organise data. From the implementation's point of view they are there to help organise storage. There are many languages (lisp, prolog) which seem untyped from the user's point of view (at least from the naive user's point of view), but there are very few that are untyped from the implementation's. The only example I can think of is BCPL.

Obviously a programming language must be designed so that the compiler knows how much storage to assign for a piece of data it is required to manipulate. In an untyped language (or rather a pseudo-untyped), what happens is that a package is formed. As well as the data (just a sequence of bits to the computer) the package contains information about the type of the data, and perhaps the size of the package. This information has to be checked at run-time each time the package is used in order to ensure that operations are only applied to the kinds of data for which they are intended, or to interpret, for example, whether an integer or a floating point arithmetic operation is required. This checking and disambiguation is a significant run-time overhead for a pseudo-untyped language, and many systems which aim at production quality code have facilities for minimising this run-time type-checking after the debugging phase. Despite this optimization, there is an obvious incentive to design a type system so that all the type-checking, disambiguation, and calculation of the size of pieces

[1] "Miranda", when used as the name of a programming language, is a trademark of Research Software Limited

of data, can be performed at compile-time, and so that the only thing needing to be stored is the sequence of bits representing the data itself.

Languages such as Pascal manage compile-time storage allocation in a very simple-minded way, which is why code in Pascal is strictly and monomorphically typed. An obvious example to illustrate the problems this can cause the programmer is the code to swap the contents of two variables. This procedure would be used in most sorting algorithms, as well as in many other places.

```
procedure SwapReal (var A,B:real);
 var Temp:real;
 begin
   Temp := A;
   A := B;
   B := Temp
 end;
```

This procedure cannot be used with integers, or booleans, even though the only way in which the code differs is in the types listed in the declarations.

C adopts a similar allocation strategy, but escapes monomorphic typing by abandoning all sense of hygiene when it comes to pointers.

Christopher Strachey observed that many common algorithms could be applied uniformly across types, and he proposed to distinguish between these, which he called "parametrically polymorphic", and operations which, although they could be applied to many, or even to all types, were not uniform. Examples here are operations such as equality testing and addition. He described this second form as "ad hoc polymorphism", though we would now tend to describe it as overloading (perhaps Haskell's type classes come closest to Stachey's intent).

Strachey envisaged polymorphic code more along the lines of

```
procedure Swap (type X; var A,B:X);
 var Temp:X;
 begin
   Temp := A;
   A := B;
   B := Temp
 end;
```

in which the type `X` is a variable instantiated at the time of call.

We are now familiar with languages such as ML, Miranda and Haskell, which allow us to adopt this viewpoint, although the types are deduced by the compiler, and need not appear explicitly in the code. So that the type of `swap` given by the SML declaration

```
fun swap (x,y) = (y,x)
```

is `'a * 'b -> 'b * 'a`, where `'a` and `'b` are type variables.

The viewpoint adopted by these languages is that a type variable is a place-holder which may be filled in by a monomorphic type. Moreover, the type-checking algorithm is advertised as finding the most general possible type for any piece of code. The ML-style type structure is convenient and flexible. It is natural to ask why we should look at anything else.

However, the ML-style type structures do have certain problem areas. One of these is in the area of record specification, particularly when an attempt is made to combine the instantiation coercions derived from core ML with field-dropping coercions, see for example Ohori and Buneman [OB88], or Wand [Wan89]. (This in turn leads on to problems in treating the structure of object-oriented languages with inheritance, and in treating the programming in the large structures of SML itself). Another problematic area is the introduction of continuations as first-class objects (cf. [DHM91]).

We shall attempt to give a couple of more basic examples intended to show that all is not quite as straightforward as it seems. It is not proposed that the second-order lambda calculus on its own provides a satisfactory solution to the problems posed by either of these, but I do believe that it can help by providing a perspective on what those problems actually are.

The first example is the typing of the `S` combinator. In SML, the declaration

```
fun s x y z = (x z) (y z);
```

gives

```
val s = fn : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

Now consider

```
fun x (f::fs) (v::vs) = (f v)::(x fs vs)
  | x [] (v::vs) = []
  | x (f::fs) [] = []
  | x [] [] = [];
val x = fn : ('a -> 'b) list -> 'a list -> 'b list

fun y vs = [] @ vs;
val y = fn : 'a list -> 'a list
```

```
val z = nil;
val z = [] : 'a list
```

Here, `x` is a function that takes a list of functions and a list of possible inputs, and returns the list of values obtained by applying functions to corresponding inputs until either the list of functions or the list of inputs runs out. There is nothing special about `y` beyond the fact that it is a function of type `'a list -> 'a list`.

When we try to run `s x y z` we get a type error:

```
std_in:8.1-8.7
  Error: operator and operand don't agree (circularity)
  operator domain: ('S -> 'T) list -> 'S list
  operand:         ('S -> 'T) list -> ('S -> 'T) list
  in expression:
    s x y
```

However:

```
- (x z) (y z);
val it = [] : 'a list
```

Thus, if we evaluate the code for `s` in an environment in which the parameters of `s` are bound to certain values, we have no problems, but if we try to call `s` on the same values we have a type error!

The problem is that the parameter `z` is used twice, and in our example the uses have different types (`'a list` and `('a -> 'b) list`). The ML type-deduction algorithm cannot cope with this. Perhaps we should be more precise and say the Hindley-Milner type-deduction algorithm, on which the ML algorithm is based, cannot cope. In general, ML and similarly typed languages cannot deal with code that has a polymorphic argument, and requires the argument to be instantiated at more than one type. (In ML, if you can pass a polymorphic argument to a function, then it is always possible to pass a monomorphic one instead).

The next example has to do with ref types. This has been "solved" in current implementations, but not in a very elegant fashion. A folklore example here is

```
let val x = ref(fn x => x)
 in x := (fn x => x+1)
    !x true
 end
```

Naively, `x` has type `('a -> 'a) ref`. This matches not only the dynamic assignment of a function of type `int -> int`, but also dereferencing, followed by application to a term of type `bool`. However, the combination yields nonsense. It is very easy to make up similar examples. The basic problem is that ref types should not be left open in this way.

There seem to be similarities between this problem with ref types and some of the more general difficulties experienced when attempting to treat continuations.

One approach to these problems is to say that the original Hindley-Milner style type structure, with only open types, is concealing parts of the structure we might wish to examine. In particular, it obscures the binding of type variables to types. We should develop a calculus that brings this more out into the open. Once we have made that decision, there is an obvious abstraction mechanism that allows us to say that an expression is polymorphic in a given type variable. This is to use the same technology for describing the binding times of type variables as we are already using to describe binding times at the individual level. Following this path leads us to the second-order lambda calculus. This is of course a historically very inaccurate picture of its development. Moreover, the recognition that there is a problem with the binding times of type variables does not lead inevitably to the second-order lambda calculus. It also covers the current SML solution, which is to give type variables an integer rank conveying the order in which they have to be bound to relatively concrete types. This, however, is a use of different techniques at the type and the individual levels.

We shall begin by looking at the simplest version of the second-order lambda calculus, and later will look at the possibility of encoding more sophisticated versions in this minimalist theory.

The only types in the minimal theory are variables, function types, and polymorphic types. As is quite common in logic and type theory generally, we can either do a certain amount of hand-waving about free and bound variables, or we can take a bit more trouble and set up a more precise notion of context. We shall attempt the latter. Since the second order lambda calculus has variables ranging over both types and values, our contexts must reflect this distinction. Accordingly a context will intuitively look like a set of type variables, together with a set of individual variables bound to types. It will be a property of contexts that the type variables occurring free in the types to which value variables are bound are always listed in the context.

More formally, assume we are given an infinite set $\{X, Y, \ldots\}$ of type variables, and an infinite but disjoint set $\{x, y, \ldots\}$ of value variables. We shall use Γ to range over type contexts, and Δ to range over value contexts. Our judgements will, initially, tell us when something is a context, when something is a type, and when something is a value of an appropriate type. Later on we shall add reduction rules.

We shall take a type context to be a list of (distinct) type variables. Really we should be more rigorous, and give a proper syntax.

The type formation judgements are:

$$(W^t) \qquad \frac{\Gamma \vdash \tau \ \mathsf{type}}{\Gamma, X \vdash \tau \ \mathsf{type}} \qquad (X \text{ not in } \Gamma)$$

$$(E^t) \qquad \frac{\Gamma, X, X', \Gamma' \vdash \tau \ \mathsf{type}}{\Gamma, X', X, \Gamma' \vdash \tau \ \mathsf{type}}$$

$$(v^t) \qquad \frac{}{X \vdash X \ \mathsf{type}}$$

$$(\to^t) \qquad \frac{\Gamma \vdash \sigma \ \mathsf{type} \qquad \Gamma \vdash \tau \ \mathsf{type}}{\Gamma \vdash \sigma \to \tau \ \mathsf{type}}$$

$$(\forall^t) \qquad \frac{\Gamma, X \vdash \tau \ \mathsf{type}}{\Gamma \vdash \forall X.\tau \ \mathsf{type}}$$

These rules tell us how the types of the calculus are formed. The first two are structural rules, whereas the others do some real work.

The next set of rules tells us how to form a context, and is similarly divided into structural and non-structural rules, but the only non-structural rule is trivial, and the work is done by the second weakening rule (W^{cv}).

$$(W^{ct}) \qquad \frac{\Gamma; \Delta \ \mathsf{context}}{\Gamma, X; \Delta \ \mathsf{context}} \qquad (X \text{ not in } \Gamma)$$

$$(W^{cv}) \qquad \frac{\Gamma; \Delta \ \mathsf{context} \qquad \Gamma \vdash \tau \ \mathsf{type}}{\Gamma; \Delta, x : \tau \ \mathsf{context}} \qquad (x \text{ not in } \Delta)$$

$$(E^{ct}) \qquad \frac{\Gamma, X, X', \Gamma'; \Delta \ \mathsf{context}}{\Gamma, X', X, \Gamma'; \Delta \ \mathsf{context}}$$

$$(E^{cv}) \qquad \frac{\Gamma; \Delta, x : \tau, x' : \tau', \Delta' \ \mathsf{context}}{\Gamma; \Delta, x' : \tau', x : \tau, \Delta' \ \mathsf{context}}$$

$$(I) \qquad \frac{}{\emptyset; \emptyset \ \mathsf{context}}$$

Finally we have the rules which tell us how to form terms in the calculus. Once again the first few rules are structural, and are usually omitted. Normally, only the last four or five rules are listed.

$$(W^{vt}) \qquad \frac{\Gamma; \Delta \vdash e\colon \tau}{\Gamma, X; \Delta \vdash e\colon \tau}$$

$$(W^{vv}) \qquad \frac{\Gamma; \Delta \vdash e\colon \tau \qquad \Gamma \vdash \sigma \ \mathsf{type}}{\Gamma; \Delta, x : \sigma \vdash e\colon \tau} \qquad (x \text{ not in } \Delta)$$

$$(E^{vt}) \qquad \frac{\Gamma, X, X', \Gamma'; \Delta \vdash e\colon \tau}{\Gamma, X', X, \Gamma'; \Delta \vdash e\colon \tau}$$

$$(E^{vv}) \qquad \frac{\Gamma; \Delta, x : \tau, x' : \tau', \Delta' \vdash e\colon \tau}{\Gamma; \Delta, x' : \tau', x : \tau, \Delta' \vdash e\colon \tau}$$

$$(V^{v}) \qquad \frac{\Gamma; \Delta, x : \sigma \ \mathsf{context}}{\Gamma; \Delta, x : \sigma \vdash x\colon \sigma}$$

$$(\to^{I}) \qquad \frac{\Gamma; \Delta, x : \sigma \vdash e\colon \tau}{\Gamma; \Delta \vdash \lambda x : \sigma . e\colon \sigma \to \tau}$$

$$(\to^{E}) \qquad \frac{\Gamma; \Delta \vdash u\colon \sigma \to \tau \qquad \Gamma; \Delta \vdash v\colon \sigma}{\Gamma; \Delta \vdash uv\colon \tau}$$

$$(\forall^{I}) \qquad \frac{\Gamma, X; \Delta \vdash e\colon \tau}{\Gamma; \Delta \vdash \Lambda X . e\colon \forall X . \tau} \qquad (X \text{ not free in the type of any variable in } \Delta)$$

$$(\forall^{E}) \qquad \frac{\Gamma; \Delta \vdash e\colon \forall X . \tau \qquad \Gamma \vdash \sigma \ \mathsf{type}}{\Gamma; \Delta \vdash e[\sigma]\colon \tau[X := \sigma]}$$

The most obvious thing to notice about these rules is the side condition in the rule $(\forall^I)$. This ensures that whenever we can deduce $\Gamma; \Delta \vdash e\colon \tau$, then any type variable occurring free in Δ or τ is listed in Γ, which is what we would expect. Indeed, this side-condition is needed even in those formulations of the calculus that do not use explicit contexts, and the major reason we have gone to the trouble of introducing contexts explicitly is to make its function apparent.

We should also note the rule $(\forall^E)$. This is technically a rule schema (in the same sense that β-reduction is a rule schema). Note that as yet we have no formal notion of equality between expression denoting values or between expressions denoting types. We shall shortly introduce one on values, but not on types (though see below). The fact that the type in the conclusion of this rule is $\tau[X := \sigma]$, and not $(\forall X.\tau)(\sigma)$, means that we do not have to introduce a notion of beta-reduction for expressions denoting types, and are able instead to say that two expressions denote the same type if and only if they are syntactically identical (though, once again, see below). Note that if we introduce types of the form $(\forall X.\tau)(\sigma)$, then the requirement that evaluation of expressions preserves their type will force the equivalence of this type with $\tau[X := \sigma]$. Furthermore, we must also introduce rules to ensure that a term of one type may also be deemed to have any equivalent type. This complicates the process of type-checking considerably, and makes type inference extremely difficult, except, possibly, when we have a normal form for type expressions.

At this point I must come clean, and admit that we still have to do something about alpha conversion. Accordingly, we shall regard type or value expressions which differ only up to alpha conversion (of either type or value variables) as syntactically identical. In the absence of formal definitions, this statement is of course nothing more than hand-waving. The best way to make it more rigorous is to use a formal meta-language to define the syntax of the calculus. We can then use alpha-conversion in the meta-language to define what we mean by alpha-conversion here. Indeed, the possibility of doing this kind of thing is one of the more obvious advantages of using such a formal metalanguage.

At last we come to the equations between value expressions. We have two forms of beta reduction, corresponding to the two connectives $\rightarrow$ and $\forall$.

$$(\beta^{\rightarrow}) \qquad \frac{\Gamma; \Delta \vdash (\lambda x : \sigma.e)v : \tau}{\Gamma; \Delta \vdash (\lambda x : \sigma.e)v = e[x := v] : \tau}$$

$$(\beta^{\forall}) \qquad \frac{\Gamma; \Delta \vdash (\Lambda X.e)[\sigma] : \tau}{\Gamma; \Delta \vdash (\Lambda X.e)[\sigma] = e[X := \sigma] : \tau}$$

The first of these is sufficiently familiar to require no explanation. The second could perhaps do with an example. Consider the polymorphic identity function. In our formulation of the calculus it is given by $\texttt{poly\_id} = \Lambda X.\lambda x : X.x$ (we have $\emptyset; \emptyset \vdash \Lambda X.\lambda x : X.x : \forall X.X \rightarrow X$). T Then the rule $(\beta^{\forall})$ tells us that $\texttt{poly\_id [int]} = (\lambda x : X.x)[X := \texttt{int}] = \lambda x : \texttt{int}.x$, which is the identity function on the integers.

We may also require η rules. In the untyped lambda calculus, the status of the η rule is rather equivocal. It is something of an optional extra. The same is true here. "Real" computation is embodied in the β rules. We shall discuss this further in the context of pairing, where things are clearer, but for the moment it suffices to note that we can impose η rules, and that there is one for each connective.

$$(\eta^{\to}) \qquad \frac{\Gamma;\Delta \vdash f{:}\sigma \to \tau}{\Gamma;\Delta \vdash \lambda x:\sigma.fx = f{:}\sigma \to \tau} \quad \text{(provided } x \text{ does not occur in } \Delta)$$

$$(\eta^{\forall}) \qquad \frac{\Gamma;\Delta \vdash e{:}\forall X.\tau}{\Gamma;\Delta \vdash \Lambda X.e[X] = e{:}\forall X.\tau} \quad \text{(provided } X \notin \Gamma)$$

Finally, there is a whole host of rules to ensure that equality is a congruence (i.e. an equivalence that respects all the term formation operations). The most famous of these rules is called ξ in the lambda calculus, and says that lambda abstraction respects equality.

The rules have been given for a theory of equality. They could of course equally well have been given for a reduction system. This would have involved ordering the β and η rules as reductions left to right, as usual in the untyped lambda calculus.

At this point it seems appropriate to give a couple of simple examples of types and terms. We shall be seeing these again later.

EXAMPLES. (i) *(Polymorphic identity)* We can verify that we can form the valid judgement:

$$\emptyset;\emptyset \vdash \Lambda X.\lambda x:X.x{:}\forall X.X \to X$$

This gives us the identity function on any type by type application. If we have

$$\Gamma \vdash \sigma \ \mathsf{type},$$

then

$$\Gamma;\emptyset \vdash (\Lambda X.\lambda x:X.x)[\sigma] = \lambda x:\sigma.x{:}\sigma \to \sigma$$

by $(\beta^{\forall})$.
(ii) *(Polymorphic Church Numerals)* A Church numeral is a term which takes an initial value and an endo-function and returns the result of applying the function to the initial value a certain number of times. We can take the value from, and the function to act on, a variable type X, and then abstract with respect to X to get the polymorphic Church numerals:

$$\emptyset;\emptyset \vdash \Lambda X.\lambda x:X.\lambda f:X \to X.f^n(x){:}\forall X.X \to (X \to X) \to X$$

We end this section with a brief discussion of type erasure. The idea behind this is that if we erase all mention of types from a term of the second-order lambda calculus, then we end up with a term of the untyped lambda calculus. Indeed, this idea is at the heart of some common implementation strategies for functional languages with compile-time typing. The types of the program are checked at compile-time, and are then erased leaving an untyped lambda-term, which is handed over to a general reduction engine for evaluation. Informally, the definition is obvious. Formally, we proceed by induction on the structure of the term:

$$\begin{aligned} x^\epsilon &= x \\ (\lambda x : \sigma.e)^\epsilon &= \lambda x.e^\epsilon \\ (uv)^\epsilon &= u^\epsilon v^\epsilon \\ (\Lambda X.e)^\epsilon &= e^\epsilon \\ e[\sigma]^\epsilon &= ere \end{aligned}$$

We can refine this to include contexts. The type variable part of the context is simply thrown away, as is the type information from the individual variable part of the context. Only the list of individual variables remains.

Notice that the instantiation of a polymorphic value at a type does not affect its type erasure. In this sense the second order lambda calculus provides a faithful implementation of Strachey's notion of parametric polymorphism.

EXERCISES. 1. Show that if $\Gamma; \Delta \vdash e\colon \tau$, and X is a type variable free in τ, or free in σ where $x : \sigma \in \Delta$, then $X \in \Gamma$.
2. Show that if $\Gamma; \Delta \vdash e\colon \tau$, then both $\Gamma; \Delta$ $\mathsf{context}$ and $\Gamma \vdash \tau$ type are valid.
3. Show that if $\Gamma; \Delta \vdash e\colon \tau$, and if x is free in e, then show that for some σ, $x : \sigma$ is in Δ. Show also that for any $x : \sigma$ in Δ, $\Gamma \vdash \sigma$ type.
4. Write down a complete set of equality rules for the calculus.
5. *(Substitution)* Show that the two following substitution rules for values and types are derived rules of the calculus.

$$(Sub^v)\frac{\Gamma; \Delta, x : \sigma \vdash e\colon \tau \qquad \Gamma; \Delta \vdash e'\colon \sigma}{\Gamma; \Delta \vdash e[x := e']\colon \tau}$$

$$(Sub^v)\frac{\Gamma, X; \Delta \vdash e\colon \tau \qquad \Gamma \vdash \sigma \ \mathsf{type}}{\Gamma; \Delta \vdash e[X := \sigma]\colon \tau[X := \sigma]}$$

6. *(Subject Reduction)* Show that if $\Gamma; \Delta \vdash e\colon \tau$, and e can be reduced to e', then $\Gamma; \Delta \vdash e'\colon \tau$ is valid.

2 Pairing in the second-order lambda calculus

Even without basic types and operations the second-order lambda calculus is surprisingly rich. We shall see that there are closed typed expressions which can be made to play the roles of datatypes such as the booleans and the natural numbers (and hence integers). This goes somewhat beyond the mere typing of some well-known lambda terms, as in the examples at the end of the last chapter. For example, the type

$$\mathtt{Bool} = \forall X.X \to X \to X$$

can be made to play the role of the booleans, where

$$\mathtt{true} = \Lambda X.\lambda x : X.\lambda y : X.x$$

$$\mathtt{false} = \Lambda X.\lambda x : X.\lambda y : X.y$$

and, as we would hope from the Church numeral example, the type

$$\mathtt{Nat} = \forall X.X \to (X \to X) \to X$$

plays the role of the natural numbers, where

$$\ulcorner n \urcorner = \Lambda X.\lambda x : X.\lambda f : X \to X.f^n(x).$$

The study of these types is interesting enough, and we shall be looking at them in some depth in the next chapter. However, it turns out to be simpler to make some of our initial points in what seems at first sight to be a more complex setting, that of general type constructors. In this chapter we shall look at the concrete case of pairing (the "product" type constructor). Specifically, in this chapter we shall look at the encoding of pairing in the second-order lambda calculus. The material here is based quite heavily on the work of David Bosley [Bos92], which is in turn founded on ideas of Wraith [Wra89]. Bosley's results are more general than any presented here, and will be contained in his DPhil thesis.

In order to be more precise about what is going on, we shall actually add cartesian products to the types available in the language, and then explore the interpretation of the extended language in the language of the previous chapter.

We define two new type forms:

$$(\times^t) \qquad \frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma \vdash \tau \text{ type}}{\Gamma \vdash \sigma \times \tau \text{ type}}$$

$$(\texttt{unit}^t) \qquad \frac{}{\emptyset \vdash \texttt{unit type}}$$

with associated term formation operations:

$$(\times^I) \qquad \frac{\Gamma; \Delta \vdash a{:}\,\sigma \qquad \Gamma; \Delta \vdash b{:}\,\tau}{\Gamma; \Delta \vdash \langle a, b\rangle{:}\,\sigma \times \tau}$$

$$(\times^E) \qquad \frac{\Gamma; \Delta, x : \sigma, y : \tau \vdash e{:}\,\nu \qquad \Gamma; \Delta \vdash e'{:}\,\sigma \times \tau}{\Gamma; \Delta \vdash \texttt{let } \langle x, y\rangle \Leftarrow e' \texttt{ in } e{:}\,\nu}$$

$$(\texttt{unit}^I) \qquad \frac{}{\emptyset; \emptyset \vdash (\ ){:}\,\texttt{unit}}$$

$$(\texttt{unit}^E) \qquad \frac{\Gamma; \Delta \vdash e{:}\,\nu \qquad \Gamma; \Delta \vdash e'{:}\,\texttt{unit}}{\Gamma; \Delta \vdash \texttt{let } (\ ) \Leftarrow e' \texttt{ in } e{:}\,\nu}$$

and beta and eta reduction laws:

$$(\beta^\times) \qquad \frac{\Gamma; \Delta \vdash a{:}\,\sigma \qquad \Gamma; \Delta \vdash b{:}\,\tau \qquad \Gamma; \Delta, x : \sigma, y : \tau \vdash e{:}\,\nu}{\Gamma; \Delta \vdash \begin{array}{l}\texttt{let } \langle x, y\rangle \Leftarrow \langle a, b\rangle \texttt{ in } e \\ = e[x := a, y := b] \ : \nu\end{array}}$$

$$(\beta^{\texttt{unit}}) \qquad \frac{\Gamma; \Delta \vdash e{:}\,\nu}{\Gamma; \Delta \vdash (\texttt{let } (\ ) \Leftarrow (\ ) \texttt{ in } e) = e : \nu}$$

$$(\eta^\times) \qquad \frac{\Gamma; \Delta \vdash p{:}\,\sigma \times \tau}{\Gamma; \Delta \vdash \begin{array}{l}\langle \texttt{let } \langle x, y\rangle \Leftarrow p \texttt{ in } x, \texttt{let } \langle x, y\rangle \Leftarrow p \texttt{ in } y\rangle \\ = \ p : \sigma \times \tau\end{array}}$$

$$(\eta^{\texttt{unit}}) \qquad \frac{\Gamma; \Delta \vdash p{:}\,\texttt{unit}}{\Gamma; \Delta \vdash (\ ) = p : \texttt{unit}}$$

In the Curry-Howard correspondence between propositions and types, the connective that corresponds to the product type constructor is conjunction (and the proposition that corresponds to the type `unit` is `true`). Now, it is well-known to logicians that in higher-order logic these are both definable in terms of implication. It is perhaps easier to see what is going on if we look at the sequent calculus laws rather than natural deduction.

Here we have:

$$(\top - L)\ \frac{\vdash X}{\top \vdash X} \qquad (\top - R)\ \frac{}{X \vdash \top}$$

$$(\wedge - L)\ \frac{A, B \vdash X}{A \wedge B \vdash X} \qquad (\wedge - R)\ \frac{X \vdash A \qquad X \vdash B}{X \vdash A \wedge B}$$

The left-side rules give us the types $\forall X.X \to X$ and $\forall X.(A \to B \to X) \to X$. The right-side rules give us $\forall Y.(\forall X.X \to Y) \to Y$ and $\forall Y.(\forall X.(X \to A) \to (X \to B) \to X \to Y) \to Y$. We shall not attempt an algorithm for converting rules into types, but hopefully some pattern is apparent.

An interesting way to show that these types do indeed represent the connectives intended is to exhibit terms in both directions between these polymorphic types and the genuine unit and product types in the extended calculus. The terms can then be decoded into proofs of the equivalence in higher-order logic. We have

1. $\emptyset; y : \texttt{unit} \vdash \Lambda X.\lambda x : X.x : \forall X.X \to X$
2. $\emptyset; x : \forall X.X \to X \vdash (\ ) : \texttt{unit}$
3. $\emptyset; y : \texttt{unit} \vdash \Lambda Y.\lambda\theta : (\forall X.X \to Y).\theta[\texttt{unit}](\ ) : \forall Y.(\forall X.X \to Y) \to Y$
4. $\emptyset; x : \forall Y.(\forall X.X \to Y) \to Y \vdash (\ ) : \texttt{unit}$
5. $\emptyset; p : A \times B \vdash \Lambda X.\lambda\theta : (A \to B \to X).$
$\theta\langle \texttt{let } \langle x,y\rangle \Leftarrow p \texttt{ in } x, \texttt{let } \langle x,y\rangle \Leftarrow p \texttt{ in } y\rangle$
$: \forall X.(A \to B \to X) \to X$
6. $\emptyset; r : \forall X.(A \to B \to X) \to X \vdash$
$\langle r[A](\lambda a : A.\lambda b : B.a), r[B](\lambda a : A.\lambda b : B.b)\rangle$
$: A \times B$
7. $\emptyset; p : A \times B \vdash \Lambda Y.\lambda\theta : (\forall X.(X \to A) \to (X \to B) \to X \to Y)$
$\theta[\texttt{unit}](\lambda u : \texttt{unit}.\texttt{let } \langle x,y\rangle \Leftarrow p \texttt{ in } x)$
$(\lambda u : \texttt{unit}.\texttt{let } \langle x,y\rangle \Leftarrow p \texttt{ in } y)$
$(\ )$
$: \forall Y.(\forall X.(X \to A) \to (X \to B) \to X \to Y) \to Y$

EXERCISES. 1. Produce a term taking a value of the second higher-order "pair type", $\forall Y.(\forall X.(X \to A) \to (X \to B) \to X \to Y) \to Y$, to a value of type $A \times B$.
2. Write down the proofs in higher-order logic corresponding to these terms.
3. Show that composition gives the identity on the true unit and product, but not on the higher-order representations.

The next stage is to translate the calculus with product and unit types into the primitive calculus. There are several ways of doing this (one for each possible choice of representation). We shall only look in detail at the two possible coherent choices.

We shall abbreviate the first second-order version of the pair type, $\forall X.(A \to B \to X) \to X$, as $A \times_s B$ and the second, $\forall Y.(\forall X.(X \to A) \to (X \to B) \to X \to Y) \to Y$, as $A \times_l B$. Similarly we shall abbreviate $\forall X.X \to X$ as $\mathtt{unit}_s$ and $\forall Y.(\forall X.X \to Y) \to Y$ as $\mathtt{unit}_l$. (s is for "strict" and l is for "lazy", a choice which has its limitations as well as its reasons, as we shall see later). These give us two different interpretations for the product and unit type constructors. As a corollary, we can define two different interpretations of the calculus with product and unit types into the primitive calculus without either. Formally these are by induction on a proof of the well-formedness of the type expressions. The first translation is given by:

$$
\begin{array}{ll}
(var) & [\![X]\!]_s = X \\
(\to) & [\![A \to B]\!]_s = [\![A]\!]_s \to [\![B]\!]_s \\
(\forall) & [\![\forall X.A]\!]_s = \forall X.[\![A]\!]_s \\
(\times) & [\![A \times B]\!]_s = [\![A]\!]_s \times_s [\![B]\!]_s = \forall X.([\![A]\!]_s \to [\![B]\!]_s \to X) \to X \\
(\mathtt{unit}) & [\![\mathtt{unit}]\!]_s = \mathtt{unit}_s = \forall X.X \to X
\end{array}
$$

whereas the second is given by

$$
\begin{array}{ll}
(var) & [\![X]\!]_l = X \\
(\to) & [\![A \to B]\!]_l = [\![A]\!]_l \to [\![B]\!]_l \\
(\forall) & [\![\forall X.A]\!]_l = \forall X.[\![A]\!]_l \\
(\times) & [\![A \times B]\!]_l = [\![A]\!]_l \times_l [\![B]\!]_l \\
 & \qquad = \forall Y.(\forall X.(X \to [\![A]\!]_l) \to (X \to [\![B]\!]_l) \to X \to Y) \to Y \\
(\mathtt{unit}) & [\![\mathtt{unit}]\!]_l = \mathtt{unit}_l = \forall Y.(\forall X.X \to Y) \to Y
\end{array}
$$

Note that this translation of syntax is the recursive application of the translation of constructors given above. This makes it slightly less general. We are not formally told what to do about ground types or any type constructors apart from the ones listed above. Moreover, there is a possibility of coherence problems if we have a notion of equivalence between types.

The next stage is to translate the terms of the calculus. For this we use:

$$
\begin{array}{l}
\langle a, b\rangle_s = \Lambda X.\lambda\theta : (A \to B \to X).\ \theta ab \\
(\ )_s = \Lambda X.\lambda x : X.\ x
\end{array}
$$

(where a, b have types A and B respectively), with corresponding eliminations:

$$\texttt{let } \langle x, y\rangle_s \Leftarrow e' \texttt{ in } e = e'[\nu](\lambda x : A\lambda y : B.e)$$
$$\texttt{let } (\ )_s \Leftarrow e' \texttt{ in } e = e'[\nu]e$$

(again, given the appropriate typing).

Exercises. 1. List appropriate types for the variables in the translation above.
2. Give a formal definition of a translation of the terms of the extended second-order lambda calculus into terms of the primitive second-order lambda calculus, using induction on the structure of a proof of well-formedness.

However, it is not sufficient to simply give a translation of terms, we must also verify that the translation preserves equivalence between terms. We must show, for example, that the beta rule for products can be derived in the primitive calculus:

$$\begin{aligned}
&\texttt{let } \langle x, y\rangle_s \Leftarrow \langle a, b\rangle_s \texttt{ in } e\\
&= (\Lambda X.\lambda\theta : A \to B \to X.\theta ab)[\nu](\lambda x : A.\lambda y : B.e)\\
&= (\lambda\theta : A \to B \to \nu.\theta ab)(\lambda x : A.\lambda y : B.e)\\
&= (\lambda x : A.\lambda y : B.e)ab\\
&= (\lambda y : B.e[x := a])b\\
&= e[x := a, y := b]
\end{aligned}$$

as required.

When we come to the eta rule we get:

$$\begin{aligned}
&\langle \texttt{let } \langle x, y\rangle_s \Leftarrow p \texttt{ in } x, \texttt{let } \langle x, y\rangle_s \Leftarrow p \texttt{ in } y\rangle_s\\
&= \Lambda X.\lambda\theta : A \to B \to X.\theta(p[X](\lambda x : A.\lambda y : B.x))(p[X](\lambda x : A.\lambda y : B.y))
\end{aligned}$$

This cannot be reduced further and so (given the strong normalisation results we shall see later) the eta rule fails for this translation.

Exercise. Verify that beta holds and eta fails even for the unit type.

These results are not a complete disaster. The eta rule has always been something of an optional extra, and now we should explain why. In the classical lambda calculus the eta-rule is often explained as an encoding of extensionality. This idea breaks down, even in the untyped setting, if instead of looking at the lambda calculus, you look at combinatory algebra. From this perspective you can see that the rule really says that every lambda term is a functional abstraction. This is even clearer in the typed lambda calculus. In the extended calculus with product and unit types the eta rules for product and unit play similar roles. The first says that anything of product type is constructed as a pair (the pairing of its first and second components), and the second says that anything of unit type is constructed as the canonical element of unit type, i.e. that the only element of unit type is ( ). We are now in a position to see why the eta rules are something of a luxury from a computational point of view. Functional programming languages work by evaluating expressions to canonical form. Now the only canonical form for expressions of a product type is a pair. This is true for both lazy and call-by-value languages (the difference is that in strict pairing the components of the pair must themselves be in canonical form, whereas in lazy pairing they need not). However, if we know that the expression p is a pair $\langle a, b\rangle$, then the eta rule follows from the beta rule:

$$
\begin{aligned}
&\langle \texttt{let } \langle x,y\rangle_s \Leftarrow p \texttt{ in } x, \texttt{let } \langle x,y\rangle_s \Leftarrow p \texttt{ in } y\rangle_s\\
&= \langle \texttt{let } \langle x,y\rangle_s \Leftarrow \langle a,b\rangle_s \texttt{ in } x, \texttt{let } \langle x,y\rangle_s \Leftarrow \langle a,b\rangle_s \texttt{ in } y\rangle_s\\
&= \langle x[x := a, y := b], y[x := a, y := b]\rangle_s\\
&= \langle a, b\rangle_s\\
&= p
\end{aligned}
$$

Now, in a call-by-value language we can assume that p is in canonical form, and hence a pair, when we commence the evaluation. It follows that we never need the full force of eta. In a lazy language, p may not yet have been evaluated to canonical form, but we know that it will be as soon as its value is required. So we can use the Church-Rosser theorem to show that we never really need eta here either.

It is perhaps worth remarking that in both call-by-value and lazy languages, any lambda form is canonical (evaluation never proceeds under a lambda). So a similar (but slightly more complex) argument shows that we don't need the full force of eta for the arrow type.

Finally, suppose we use the $[\![\ ]\!]_s$ translation to implement a language with pairing on top of a call-by-value language based on the simple second-order lambda calculus. Then in the course of the evaluation of the beta rule, both a and b will get evaluated. So, as we might expect, we have a strict notion of pairing.

We now repeat the discussion above for the other interpretation of our product and unit types. Once again we can translate the construction and elimination expressions.

$$\begin{aligned}
&\langle a,b\rangle_l = \Lambda Y.\lambda\psi : (\forall X.(X \to A) \to (X \to B) \to X \to Y).\\
&\qquad\qquad \psi[\mathtt{unit}_l](\lambda x : \mathtt{unit}_l.a)(\lambda x : \mathtt{unit}_l.b)(\ )_l\\
&(\ )_l = \Lambda Y.\lambda\theta : (\forall X.X \to Y).\theta[Y \to Y](\lambda y : Y.y)\\
&\mathtt{let}\ \langle x,y\rangle_l \Leftarrow e'\ \mathtt{in}\ e =\\
&\qquad e'[\nu](\Lambda X.\lambda f : X \to A.\lambda g : X \to B.\lambda x' : X.\\
&\qquad\qquad (\lambda x : A.\lambda y : B.e)(fx')(gx'))\\
&\mathtt{let}\ (\ )_l \Leftarrow e'\ \mathtt{in}\ e = e'[\nu](\Lambda X.\lambda x : X.e)
\end{aligned}$$

As before, we leave it as an exercise to set up the details of type assignment, and to give a formal translation of the calculus. Notice, however, that in the pairing operation, a and b have been wrapped up in closures.

We proceed with the verification of beta for pairing.

$$\begin{aligned}
&\mathtt{let}\ \langle x,y\rangle_l \Leftarrow \langle a,b\rangle_l\ \mathtt{in}\ e\\
&= (\Lambda Y.\lambda\psi : (\forall X.(X \to A) \to (X \to B) \to X \to Y).\\
&\qquad\qquad \psi[\mathtt{unit}_l](\lambda x : \mathtt{unit}_l.a)(\lambda x : \mathtt{unit}_l.b)(\ )_l)\\
&\qquad [\nu]\\
&\qquad (\Lambda X.\lambda f : (X \to A).\lambda g : (X \to B).\lambda x' : X.\\
&\qquad\qquad (\lambda x : A.\lambda y : B.e)(fx')(gx'))\\
&= (\lambda\psi : (\forall X.(X \to A) \to (X \to B) \to X \to \nu).\\
&\qquad\qquad \psi[\mathtt{unit}_l](\lambda x : \mathtt{unit}_l.a)(\lambda x : \mathtt{unit}_l.b)(\ )_l)\\
&\qquad (\Lambda X.\lambda f : (X \to A).\lambda g : (X \to B).\lambda x' : X.\\
&\qquad\qquad (\lambda x : A.\lambda y : B.e)(fx')(gx'))\\
&= (\Lambda X.\lambda f : (X \to A).\lambda g : (X \to B).\lambda x' : X.\\
&\qquad\qquad (\lambda x : A.\lambda y : B.e)(fx')(gx'))\\
&\qquad [\mathtt{unit}_l]\\
&\qquad (\lambda x : \mathtt{unit}_l.a)(\lambda x : \mathtt{unit}_l.b)(\ )_l)\\
&= (\lambda f : (\mathtt{unit}_l \to A).\lambda g : (\mathtt{unit}_l \to B).\lambda x' : \mathtt{unit}_l.\\
&\qquad\qquad (\lambda x : A.\lambda y : B.e)(fx')(gx'))\\
&\qquad (\lambda x : \mathtt{unit}_l.a)(\lambda x : \mathtt{unit}_l.b)(\ )_l)\\
&= (\lambda x : A.\lambda y : B.e)((\lambda x : \mathtt{unit}_l.a)(\ )_l)((\lambda x : \mathtt{unit}_l.b)(\ )_l)\\
&= (\lambda x : A.\lambda y : B.e)ab\\
&= e[x := a, y := b]
\end{aligned}$$

Note, as before, that if we imagine that evaluation proceeds under a call-by-value regime, then a and b will be evaluated during the course of this. Hence this is also a strict product.

Eta once again fails, though this is left as an exercise for the brave (or perhaps the foolhardy). Similarly, we shall leave the verification of the corresponding facts for the unit type.

We observed earlier that the components of a pair were wrapped up in closures, and also that even under a call-by-value regime the contents of a closure are not evaluated until an argument is supplied to the closure. This however has not prevented the evaluation of the components of the pair, since an argument has been prematurely supplied. I do not see how to avoid this while using a purely structural form of translation. However, we can give a slightly modified translation, which is defined only for terms of the calculus, by using substitution instead of lambda abstraction followed by application. We replace

$$\begin{aligned}&\texttt{let } \langle x,y\rangle_l \Leftarrow e' \texttt{ in } e =\\ &\quad e'[\nu](\Lambda X.\lambda f : X \to A.\lambda g : X \to B.\lambda x' : X.\\ &\qquad\qquad (\lambda x : A.\lambda y : B.e)(fx')(gx'))\end{aligned}$$

by

$$\begin{aligned}&\texttt{let } \langle x,y\rangle_l \Leftarrow e' \texttt{ in } e =\\ &\quad e'[\nu](\Lambda X.\lambda f : X \to A.\lambda g : X \to B.\lambda x' : X.\\ &\qquad\qquad (e[x := fx', y := gx']))\end{aligned}$$

Notice that we cannot formulate this generically as before, since if e were a variable, this translation would give us:

$$\begin{aligned}&\texttt{let } \langle x,y\rangle_l \Leftarrow e' \texttt{ in } e =\\ &\quad e'[\nu](\Lambda X.\lambda f : X \to A.\lambda g : X \to B.\lambda x' : X.\\ &\qquad\qquad (e))\end{aligned}$$

which is not what we want at all.

If we try the beta rule with this new definition, we get:

$$\begin{aligned}&\texttt{let } \langle x,y\rangle_l \Leftarrow \langle a,b\rangle_l \texttt{ in } e\\ &= (\Lambda Y.\lambda\psi : (\forall X.(X \to A) \to (X \to B) \to X \to Y).\\ &\qquad\qquad \psi[\texttt{unit}_l](\lambda x : \texttt{unit}_l.a)(\lambda x : \texttt{unit}_l.b)(\ )_l)\\ &\quad [\nu]\\ &\quad (\Lambda X.\lambda f : X \to A.\lambda g : X \to B.\lambda x' : X.\\ &\qquad (e[x := fx', y := gx']))\end{aligned}$$

$$
\begin{aligned}
&= (\Lambda X.\lambda f : X \to A.\lambda g : X \to B.\lambda x' : X.\\
&\qquad\qquad (e[x := fx', y := gx']))\\
&\qquad [\mathtt{unit}_l]\\
&\qquad (\lambda x : \mathtt{unit}_l.a)(\lambda x : \mathtt{unit}_l.b)(\ )_l)\\
&= e[x := ((\lambda x : \mathtt{unit}_l.a)(\ )_l), y := ((\lambda x : \mathtt{unit}_l.b)(\ )_l)]
\end{aligned}
$$

in which the a and b expressions are still wrapped in their closures, and hence still unevaluated. It follows that with this revised definition, we now have lazy pairing, even under a call-by-value evaluation strategy.

To make this properly rigorous (indeed to make our entire discussion properly rigorous) we should set up a formal operational semantics for lazy and strict pairing in the context of a general call-by-value regime, and verify that the translations given preserve the validity of the reduction rules.

Readers may be wondering how this relates to the classical work of Plotkin on call-by-name and call-by-value variants of the untyped lambda calculus [Plo75]. Plotkin's translations use a continuation passing style in order to enforce a particular order of evaluation. It is not clear how this relates to the type translations given here.

Finally, we should remind the reader that although this chapter has been based quite heavily on work of Bosley and Wraith [Bos92, Wra89], their results are significantly more general, and encompass even the types discussed at the end of the next chapter.

EXERCISE. Given that the expression that takes the first component of a pair $\langle a, b\rangle$ is

$$\mathtt{let}\ \langle x, y\rangle \Leftarrow p\ \mathtt{in}\ x,$$

verify that with this lazy definition of pairing the second component of a pair remains unevaluated when we take the first component.

3 Representing some basic types

At the start of the last section we made a claim that the types $\forall X.X \to X \to X$ and $\forall X.X \to (X \to X) \to X$ could be used to represent booleans and natural numbers respectively. Later we shall see that many other types can be represented in a similar fashion, but for the moment we shall concentrate on making good our claims about these two.

We preview two important results about the polymorphic lambda calculus from the fifth chapter of these notes. We recall that a reduction system is:
– *strongly normalizing* (SN) if there are no infinite sequences of reductions
– *Church-Rosser* (CR) if whenever there are reduction sequences reducing a term in more than one way, say $A \longrightarrow^* B$ and $A \longrightarrow^* B'$, then there is a fourth term C such that both B and B' have reduction sequences leading to C ($B \longrightarrow^* C$ and $B' \longrightarrow^* C$).

The second-order lambda calculus is strongly normalizing and Church-Rosser, and this holds for both beta and beta-eta reduction. An immediate consequence of this is that any expression reduces to a unique normal form. Moreover, if f is a closed term of type $\sigma \to \tau$, then f induces a mapping taking the closed normal forms of type σ to closed normal forms of type τ (and of course analogously for open terms).

What is the significance of this for us?

Proposition 3.1.

(i) *There are no closed terms of type* $\forall X.X$.
(ii) *The only closed normal form of type* $\forall X.X \to X$ *is the polymorphic identity:* $\Lambda X.\lambda x : X.x$.
(iii) *The closed normal forms of type* $\texttt{Bool} = \forall X.X \to X \to X$ *are*
$\texttt{true} = \Lambda X.\lambda x : X.\lambda y : X.x$
and
$\texttt{false} = \Lambda X.\lambda x : X.\lambda y : X.y$.
(iv) *The closed normal forms of type* $\texttt{Nat} = \forall X.X \to (X \to X) \to X$ *are*
$\ulcorner n \urcorner = \Lambda X.\lambda x : X.\lambda f : X \to X.f^n(x)$.

These are all instances of a more general result due to Böhm and Berarducci [BB85]. The first is in a sense a distinctly non-trivial result. It is equivalent (if one interprets the calculus as the formalization of a logic) to the consistency of the logic formalized – second-order arithmetic. Nevertheless, there is an easy model-theoretic proof.

Proof. (of (i), sketch) Interpret types as either $\emptyset$ or $\{*\}$. Calculate the function space as in set theory (so $\emptyset \to \{*\} = \emptyset \to \emptyset = \{*\} \to \{*\} = \{*\}$) and interpret quantification by

$$
\begin{array}{ll}
[\![\forall X.T(X)]\!] = \{*\} & \text{if both } [\![T(\emptyset)]\!] = \{*\} \text{ and } [\![T(\{*\})]\!] = \{*\}, \\
[\![\forall X.T(X)]\!] = \emptyset & \text{otherwise.}
\end{array}
$$

(It is worth observing, with reference to the next chapter, that this is the cartesian product of $[\![T(\emptyset)]\!]$ and $[\![T(\{*\})]\!]$, and hence the product of $[\![T(X)]\!]$ over all types.)

The interpretation of terms is uniquely constrained by this. Note that $[\![\forall X.X]\!]$ is empty. This would not be possible if there were any closed terms of this type. $\square$

LEMMA 3.2. *The only closed normal forms of type $\sigma \to \tau$ or $\forall X.T(X)$ are of the forms $\lambda x : \sigma.e$ and $\Lambda X.e$ respectively.*

Proof. We prove these results simultaneously by induction on the structure of a proof that the expression is well-typed. There are four possible cases for the structure of an arbitrary expression e: e is a lambda abstraction ($\lambda x :?.e_1$), e is an application ($e = e_1e_2$), e is a type abstraction ($e = \Lambda X.e_1$) or e is a type application ($e = e_1[?]$). If e has type $\sigma \to \tau$ then, in the first case "?" must be σ (by matching the type expressions), and this is what is required. We can rule out the third case, the type abstraction, immediately, since the type of the resulting expression is wrong, but to deal with the other cases we must appeal to the inductive hypothesis. In the first case, the application, e_1 must have type $? \to (\sigma \to \tau)$, which is an arrow type. So, by induction, e_1 must be a lambda-expression. This means that e_1e_2 is a beta-redex, and hence not in normal form. The argument for the type application $e_1[?]$ is similar. e_1 must have some polymorphic type, and so by induction has form $\Lambda X.e_2$, and hence $e_1[?]$ is a type redex. The argument for terms of polymorphic type is similar, and omitted. $\square$

Proof. (of (iv), sketch) The lemma above tells us that any normal form of type $\forall X.X \to (X \to X) \to X$ must be of the form $\Lambda X.\lambda x : X.e$ where $X; x : X \vdash e\colon (X \to X) \to X$. An extension to the argument given above allows us to say that e must have form $\lambda f : X \to X.e'$, where $X; x : X, f : X \to X \vdash e'\colon X$. e' must either be a variable, in which case it is x or an application e_1e_2. e_2 must be an expression of function type, but cannot be a lambda form, and so must be the variable f. It follows that e_2 is again an expression of type X, so is either x or f applied to an expression of type X. It follows that $e' = f^n x$ for some n. So the expression is $\ulcorner n \urcorner$, as required.

We leave the remaining details (as well as the other simpler proofs) as exercises. $\square$

Since the normal forms of type $\mathtt{Nat} = \forall X.X \to (X \to X) \to X$ correspond to natural numbers, any term of type $\mathtt{Nat} \to \mathtt{Nat} = \forall X.X \to (X \to X) \to X \to \forall X.X \to (X \to X) \to X$ induces a (total) function on the natural numbers. If we are to use this type to represent the natural numbers in any meaningful way, then we want the collection of functions that arise in this way to be as extensive as possible. We shall show that it contains the primitive recursive functions, and thus all the functions that can be computed in a reasonable amount of time (recall that any recursive function that can be computed in primitive recursive time is itself primitive recursive). Note, however, that the class of functions on the natural numbers definable in the second-order lambda calculus is in fact much larger than the primitive recursive functions; to be precise it is the class of recursive functions which have an algorithm which is provably terminating in second-order arithmetic (see later in this chapter).

To interpret the primitive recursive functions we shall make use of the fact that they are exactly the functions which can be computed by programs written in the following simple imperative language:

```
variables V
commands C ::= V := 0   |
               inc V    |
               V := V'  |
               C ; C'   |
               loop V C end
```

For a program written in this language, a state is an n-tuple of natural numbers, corresponding to the variables in the program. Since variables only take natural numbers as values, we can implement this using `Nat` and the pairing types of the last chapter. We shall write `State` for the type that represents the states of a particular program. A program's semantics will be given as a function `State` $\to$ `State`. We obtain the primitive recursive functions by designating some variables as parameters, setting all but these to zero initially, and taking the value of the function to be the value left in a designated result variable at the end of the run.

It is clear that we can find semantics for `V:= 0` and `V := V'`. `inc V` can be implemented using the successor function

$$\lambda n : \mathtt{Nat}.\Lambda X.\lambda x : X.\lambda f : X \to X.\ f(n[X]xf)$$

(N.B. we could also use

$$\lambda n : \mathtt{Nat}.\Lambda X.\lambda x : X.\lambda f : X \to X.\ n[X](fx)f$$

which also represents the successor function).

Sequencing is implemented, as one would expect, by function composition:

$$[\![\mathtt{C};\mathtt{C}']\!] = \lambda s : \mathtt{State}.\ [\![\mathtt{C}']\!]([\![\mathtt{C}]\!]s).$$

Finally, the loop construct, `loop V C end`, is implemented using the fact that natural numbers are themselves implemented as iterators. Let `val_V` be the projection which extracts the value of `V` from a state, then

$$[\![\mathtt{loop\ V\ C\ end}]\!] = \lambda s : \mathtt{State}.(\mathtt{val\_V}\ s)[\mathtt{State}][\![\mathtt{C}]\!]s.$$

It follows immediately that any primitive recursive function is representable in the second-order lambda calculus.

Exercise. Given that the program:

```
R := 0;
Flag := 0;
loop X
  loop Flag
    inc R
  end;
  Flag := 0;
  inc Flag
end
```

calculates the predecessor of X, find a second-order lambda term representing the predecessor function.

Similarly, the booleans $\forall X.X \to X \to X$ are implemented as a kind of natural binary branch type. Let

$$\mathtt{cond} = \Lambda Y.\lambda b : \mathtt{Bool}.\Lambda y_1 : Y.\Lambda y_2 : Y.b[Y]y_1y_2$$

so

$$\mathtt{cond} : \forall Y.\mathtt{Bool} \to Y \to Y \to Y.$$

Then

$$\mathtt{cond\ true}\ y_1\ y_2 = y_1$$
$$\mathtt{cond\ false}\ y_1\ y_2 = y_2$$

However, not everything in the garden is rosy. The types do not have enough closed terms for the action of a function on normal forms to determine the function completely. We have already seen an example of that in the existence of two quite separate implementations of the successor function. However more revealing examples are given by looking at non-standard implementations of the identity function. Given two elements x_0 x_1 of a type X we have a map $\texttt{Bool} \to X$ given by v

$$\lambda b : \texttt{Bool}.\ b[X]x_0 x_1.$$

If we take X to be `Bool`, and x_0 and x_1 to be `true` and `false`, then this is equal to the identity on closed terms. We might therefore hope that it would be equal to the identity in the calculus (which would be a rather strong form of extensionality). In fact it gives us

$$\lambda b : \texttt{Bool}.\ b[X]\texttt{true false},$$

which is a normal form not equal to $\lambda b : \texttt{Bool}.b$. Similarly we can define a function $\texttt{Nat} \to \texttt{Nat}$ by iterating the successor function starting at $\ulcorner 0 \urcorner$. This again should give us the identity (if we think of `Nat` as really being the set of natural numbers), but does not. It gives us $\lambda n : \texttt{Nat}.n[\texttt{Nat}]\ulcorner 0 \urcorner\texttt{succ}$.

The situation described above is very similar to that of the last chapter, where beta laws held, but eta rules failed. Here, if we had a "true" natural numbers type, or a "true" boolean type, then they would be retracts of their representations in the same way that the product types of the last chapter were. Similarly, from a computational point of view we are more interested in the beta reductions than the eta, so this does not always matter too much.

The style of representation we have given above generalises quite a long way, at least to algebraic datatypes, which form a large part of the ML recursive datatypes. Recall that a signature Σ for a (single-sorted) algebraic theory is given by giving a carrier sort X for the algebra, together with a collection of function symbols $(f_i)_{i \in I}$, each equipped with an "arity", which we can perhaps best think of as a type $A_1 \times \ldots A_n \to X$, where each A_i is some pre-existing parameter type (e.g. `int` or `bool`) or is X itself. As usual, the case $n = 0$ is deemed to cover constants. The canonical example is the signature for lists over some base type A, $\Sigma_{\texttt{list}(A)}$. Since we shall want to consider general algebras for this signature, and not just the algebra of lists itself, we shall continue to use X for the carrier sort. $\Sigma_{\texttt{list}(A)}$ has two functions, `nil` and `cons`. The first of these, `nil`, takes no arguments, and so represents a constant list. The second, `cons` has type $A \times X \to X$. Booleans, natural numbers, and even products can also be thought of in this context. The signature for booleans, $\Sigma_{\texttt{Bool}}$, has two constants, `true` and `false`. The signature for natural numbers $\Sigma_{\texttt{Nat}}$, has one constant $\ulcorner 0 \urcorner$, and one function `succ` of arity $X \to X$. The signature for the product of types A and B, $\Sigma_{A \times B}$, can be thought of as having one function of type $A \times B \to X$.

The corresponding datatypes are generally thought of as being the term algebras for terms of sort X in the algebraic theory.

We can represent these types in type theory by adding to the syntax of types:

$$\frac{\Gamma \vdash A \ \mathsf{type}}{\Gamma \vdash \mathtt{list}(A) \ \mathsf{type}}$$

giving introduction rules corresponding to the function symbols:

$$\frac{\Gamma \vdash \mathtt{list}(A) \ \mathsf{type}}{\Gamma; \emptyset \vdash \mathtt{nil}: \mathtt{list}(A)}$$

$$\frac{\Gamma; \Delta \vdash h: A \qquad \Gamma; \Delta \vdash t: \mathtt{list}(A)}{\Gamma; \Delta \vdash \mathtt{cons}(h, t): \mathtt{list}(A)}$$

and a single elimination rule, which we can write as:

$$\frac{\Gamma; \Delta \vdash c_1: C \qquad \Gamma; \Delta, h : A, t : \mathtt{list}(A) \vdash c_2: C}{\Gamma; \Delta, l : \mathtt{list}(A) \vdash \left(\begin{array}{l}\mathtt{case}\ l\ \mathtt{of} \\ \quad \mathtt{nil}\ \mathtt{=>}\ c_1 \\ \mid \mathtt{cons}(h,t)\ \mathtt{=>}\ c_2)\end{array}\right) : C}$$

Finally we need beta and perhaps eta rules. There is one beta rule for each introduction rule:

$$\frac{\Gamma; \Delta \vdash c_1: C \qquad \Gamma; \Delta \vdash c_2: C}{\Gamma; \Delta \vdash \left(\begin{array}{l}\mathtt{case\ nil\ of} \\ \quad \mathtt{nil}\ \mathtt{=>}\ c_1 \\ \mid \mathtt{cons}(h,t)\ \mathtt{=>}\ c_2)\end{array}\right) = c_1: C}$$

$$\frac{\begin{array}{cc}\Gamma; \Delta \vdash a: A & \Gamma; \Delta, h : A, t : \mathtt{list}(A) \vdash c_1: C \\ \Gamma; \Delta \vdash as: \mathtt{list}(A) & \Gamma; \Delta, h : A, t : \mathtt{list}(A) \vdash c_2: C\end{array}}{\Gamma; \Delta \vdash \left(\begin{array}{l}\mathtt{case\ cons}(a,as)\ \mathtt{of} \\ \quad \mathtt{nil}\ \mathtt{=>}\ c_1 \\ \mid \mathtt{cons}(h,t)\ \mathtt{=>}\ c_2)\end{array}\right) = c_2[h := a, t := as]: C}$$

and a single eta rule:

$$\frac{\Gamma; \Delta \vdash l: \mathtt{list}(A)}{\Gamma; \Delta \vdash \left(\begin{array}{l}\mathtt{case}\ l\ \mathtt{of} \\ \quad \mathtt{nil}\ \mathtt{=>}\ \mathtt{nil} \\ \mid \mathtt{cons}(h,t)\ \mathtt{=>}\ \mathtt{cons}(h,t))\end{array}\right) = l: \mathtt{list}(A)}$$

Readers are invited to compare these with the rules of the previous section.

To get the representing type in the second-order lambda calculus, first curry the types of all the functions in the signature. For $\Sigma_{\mathtt{list}(A)}$ this gives us X and $A \to X \to X$. Then take the type that represents a function from the product of all these types to X, and quantify. (So you can think of an element of this type as being a polymorphic function, that given a type, and a Σ-algebra structure on that type, returns an element of the type; the idea being that the only way to get such a function uniformly is to say "take the interpretation of some fixed term in the free Σ-algebra".) For $\Sigma_{\mathtt{list}(A)}$ this gives us

$$\forall X.X \to (A \to X \to X) \to X.$$

Note that following this recipe for $\Sigma_{\mathtt{Bool}}$ gives us $\forall X.X \to X \to X$, following it for $\Sigma_{\mathtt{Nat}}$ gives us $\forall X.X \to (X \to X) \to X$, and following it for $\Sigma_{A\times B}$ gives us $\forall X.(A \to B \to X) \to X$.

A slightly different way of looking at this is to consider a general theory of definition by iteration (see Böhm and Berarducci [BB85] and Leivant [Lei83]). For the theory of lists, a definition of $\mathtt{F} : \mathtt{list}(A) \to C$ by iteration takes the form:

```
F(nil) = c1
F(cons(h,t)) = c2(h,F(t))
```

where `c1` and `c2` are previously defined functions. (If we compare this with the definitional mechanism embodied in the list elimination rule, then the latter has `F(cons(h,t))= c2(h,t)`. The two are equivalent in the presence of pairing.) For natural numbers we have

```
F(0) = c1
F(n+1) = c2(F(n))
```

which is equivalent in the presence of pairing to primitive recursion.

We want to find types which somehow embody these definition mechanisms. Consider the definition by iteration over lists. The data we need consist of an element $c_1 : C$ and a function $c_2 : A \times C \to C$ (or $A \to C \to C$). We note in passing that this is exactly an algebra structure on C for the signature of lists over the type A. Given this we can obtain $F : \mathtt{list}(A) \to C$. We would like an operation to do this generically, which amounts to a suitable term of type

$$\begin{aligned}
&[C \times (A \to C \to C)] \to [\mathtt{list}(A) \to C] \\
&\simeq [C \times (A \to C \to C) \times \mathtt{list}(A)] \to C \\
&\simeq [\mathtt{list}(A) \times C \times (A \to C \to C)] \to C \\
&\simeq \mathtt{list}(A) \to [C \to (A \to C \to C) \to C]
\end{aligned}$$

It would be nice to have this polymorphically, so this amounts to a term of type

$$\begin{aligned} &\forall C.(\texttt{list}(A) \to [C \to (A \to C \to C) \to C]) \\ &\simeq \texttt{list}(A) \to \forall C.[C \to (A \to C \to C) \to C] \end{aligned}$$

We wish this to be a good canonical term, and it is up to us to choose the representation for $\texttt{list}(A)$ to make it as canonical as possible. We take $\texttt{list}(A)$ to be $\forall X.X \to (A \to X \to X) \to X$, and the term to be the identity.

If we choose this representation, F is a function $\texttt{list}(A) \to C$ defined by iteration from c_1 and c_2, then if l is a list, $F(l)$ is obtained by applying l to c_1 and c_2 (specifically, $F(l) = l[C]c_1c_2$).

EXERCISE. Verify that the map $\texttt{list}(A) \to C$ defined by iteration from c_1 and c_2 is always an algebra homomorphism. Deduce that this interpretation of $\texttt{list}(A)$ is a weakly initial algebra.

Wraith [Wra89] observes that this is not the only representation possible for lists. There is also a weakly terminal algebra (or rather several, one rather ghastly possibility for lists on A is $\forall Y.(\forall X.(X \to (\forall Z.Z \to (A \to X \to Z) \to Z)) \to X \to Y) \to Y)$. Readers should consult Wraith's paper for details of the construction.

There is a connection between weak initiality and strictness, and weak terminality and laziness which has similarities to the connections we have already seen for pairing in the last chapter. The theory of course generalises to algebraic structures other than lists. It even generalises beyond algebraic types, and is presented by Wraith in that generality.

This leads onto an area of current research interest. There is a tendency for increasing amounts of the structure of a language to be incorporated in its type system. The first clear instance of this shift in perspective was probably the change in status of the conditional from being a basic programming construct, to its current status of being derived from a general form of the case statement used to eliminate sum types (`if a then b else c` is taken to be an abbreviation for `case a of true => b | false => c`). However, people are now looking at types as embodying more operational information. Thus Abramsky in [Abr90] proposes that the form of the introduction laws for linear types suggests a natural evaluation strategy for values of that type, either lazy or strict. Others are working on type structures that incorporate information about strictness analysis (e.g. Bierman in his doctoral thesis, [Bie92]). Still others have proposed defining "strict" types by means of constructors, and "lazy" types by means of destructors (e.g. [Wra89], [CS92]).

We should also mention the area of continuations. There has been some investigation of the translation of continuation types into the second-order lambda calculus, but none of it, as far as I know, involving a serious study of the translation of a type theory with continuations into the primitive calculus. There are now some such type theories (see e.g. [DHM91]), and it should be possible to look at suitable translations, and investigate the relationship between strict and lazy datatypes as presented in say the work of Wraith, and (the translations of) their continuation-passing implementations.

So far we have seen that a large number of datatypes can be represented as closed types of the second-order lambda calculus, provided we do not demand the representations preserve the validity of eta rules. We have shown that the functions definable on the types include (for the natural numbers) the primitive recursive functions, and for recursive datatypes we have sketched out a notion of definition by iteration in such a way that it is (hopefully) obvious that any function that can be defined be iteration may be represented in the calculus. This already gives us as much as we are likely to want, provided we are prepared to stick to terminating computations. However, the collection of functions representable in the calculus is vastly larger. We shall conclude this chapter with a discussion of precisely what functions are representable, and an example of something that isn't.

Logicians traditionally use the number of functions $\mathbf{N} \to \mathbf{N}$ that can be represented in a formal system as a measure of the strength of that system. Thus we have the following result.

THEOREM 3.3. *The functions $\mathbf{N} \to \mathbf{N}$ that are representable in the second-order lambda calculus are precisely those that have an algorithm whose termination can be proved in second-order arithmetic ($\mathbf{PA}^2$).*

For a detailed proof of this see Girard [Gir89], but note that the proof splits into two parts. One part shows that the functions representable in the calculus are those with an algorithm provably total in second-order intuitionistic arithmetic ($\mathbf{HA}^2$). The other consists of a proof that the functions provable total in second-order intuitionistic arithmetic include all those provably total in second-order classical arithmetic ($\mathbf{PA}^2$). We shall concentrate explaining the connection between the second-order lambda calculus and a formal second-order (intuitionistic) logic, in order to make the result at least credible.

In second-order logic we consider a universe V of values. Propositions, or types, correspond to subsets of V. Thus individual variables (x) range over the elements of V, and second-order variables (X) range over the subsets of V. We shall also assume that we are given some collection of basic constants and function symbols as in the usual definition of a language for first-order logic, i.e. a one-sorted algebraic signature in the classical sense. For second-order arithmetic, we will be thinking of V as the natural numbers, and taking a constant 0 and function S. The basic propositions of the logic either take the form "$a = b$", where a and b are terms built from applications of the given function symbols to basic constants or variables, or "$a \in X$", where a is a term and X is a type variable. We have first-order connectives as normal, e.g. $\forall \xi.(\xi = a) \to (\xi \in X)$, and second-order universal quantification, e.g. $\forall X$ as in $\forall X.\forall \xi.\xi = a \to \xi \in X$, interpreted as "for all subsets X of V, ..."

The rules of the calculus are as normal for first-order terms, but we have in addition:

$$(\forall^{2I}) \qquad \frac{\Delta \vdash A}{\Delta \vdash \forall X.A} \quad (X \text{ not free in } \Delta)$$

$$(\forall^{2E}) \qquad \frac{\Delta \vdash \forall X.A}{\Delta \vdash A[\{\xi.C\}/X]}$$

where $A[\{\xi.C\}/X]$ means, wherever we have an atomic proposition $a \in X$ in A (which we are to think of as $X(a)$), then replace it by $C[\xi := a]$, i.e. replace $a \in X$ by $C(a)$.

We can translate propositions in second-order logic into types in the second-order lambda calculus. At the bottom level

$$\begin{aligned} [\![a \in X]\!] &= X \\ [\![a = b]\!] &= [\![\top]\!] = \forall X.X \to X \end{aligned}$$

Note that all first-order terms completely disappear at this stage in the translation. The definition of $[\![a = b]\!]$ is slightly confusing. It should not be read as a claim that all values are equal, but rather that equality gives us no new algorithmic information (this is something at the heart of many proofs involving realizability).

There are no real surprises amongst the propositional connectives, implication translates into the function type, and the other connectives are translated using second-order equivalents:

$$\begin{aligned} [\![P \to Q]\!] &= [\![P]\!] \to [\![Q]\!] \\ [\![P \wedge Q]\!] &= \prod [\![P]\!][\![Q]\!] \\ [\![P \vee Q]\!] &= \forall X.([\![P]\!] \to X) \to ([\![Q]\!] \to X) \to X \\ [\![\bot]\!] &= \forall X.X \\ [\![\top]\!] &= \forall X.X \to X \end{aligned}$$

The first-order quantifiers are perhaps a surprise, but they continue the theme of disappearance of first-order terms:

$$[\![\forall x.P]\!] = [\![P]\!]$$
$$[\![\exists x.P]\!] = [\![P]\!]$$

Finally, the second-order quantification is again as expected:

$$[\![\forall X.P]\!] = \forall X.[\![P]\!]$$

It is routine from a comparison of the rules to check that a proof of a proposition in second-order logic gives rise to a term of corresponding type in the second order lambda calculus, and vice versa. Hence the second-order lambda calculus can be regarded as a notation for proofs in second-order logic.

To represent the natural numbers in second-order logic, we take a language with a constant "0", and a function symbol "S", and define

$$\mathtt{Nat}(\xi) = \forall X.(0 \in X \to (\forall x.x \in X \to Sx \in X) \to \xi \in X).$$

Thus ξ is a natural number iff it is contained in all subsets of V which contain 0 and are closed under S.

Note that if we remove all the first-order information from this type we get $\forall X.X \to (X \to X) \to X$, as might be expected.

A function F is said to be provably terminating if its graph is defined by some predicate A such that

$$\vdash \forall n.\mathtt{Nat}(n) \to \exists m.\mathtt{Nat}(m) \wedge A(n,m).$$

The proof of this gives us a term in the second-order lambda calculus (with pairing) of type

$$\mathtt{Nat} \to (\mathtt{Nat} \times A).$$

Take the second projection of this and we get a term of type

$$\mathtt{Nat} \to \mathtt{Nat}$$

which represents the function F.

Thus, the calculus encodes a huge collection of functions, but not all the recursive functions. Moreover, since all functions are total, it cannot, for standard reasons encode its own evaluation function. Furthermore, although it can be argued that all the functions we might ever want are encoded, some of the encodings are very inefficient. This is inevitable, since the algorithms that can be encoded are those that can be proved total in a fixed fragment of arithmetic. There are inevitably (for standard recursion-theoretic reasons) some algorithms which, though fast running, are too hard to prove total. There is a good illustration to show that this is not simply a theoretical problem, but can actually occur in practice. It consists of an algorithm for the function to compute the minimum of two natural numbers. If we use Church numerals to represent natural numbers, then there is an algorithm in the untyped lambda calculus to compute the minimum of m and n which runs under normal order evaluation in time $\min(m, n)$. Algorithms in the second-order lambda calculus take at least time $m + n$. The algorithm is due to Maurey, and is discussed in Krivine [Kri87].

Maurey's algorithm itself is fairly simple, and it is worth taking the time to present it. Let F be the combinator $\lambda f.\lambda g.gf$, then Maurey observes that if μ and ν are Church numerals,

$$(\mu(\lambda x.a)F)(\nu(\lambda x.b)F)$$

evaluates to a if $\mu \leq nu$ and to v otherwise.

Proof. Suppose $\mu = \lambda x.\lambda f.f^m x$ and $\nu = \lambda x.\lambda f.f^n x$.

Then $\mu(\lambda x.a)F = F^m(\lambda x.a)$ and $\nu(\lambda x.a)F = F^n(\lambda x.a)$.

If $m = 0$ then $\mu(\lambda x.a)F = (\lambda x.a)$, so

$$\begin{aligned}&(\mu(\lambda x.a)F)(\nu(\lambda x.b)F)\\&= (\lambda x.a)(\nu(\lambda x.b)F)\\&= a,\end{aligned}$$

as required.

Otherwise, $F^m(\lambda x.a) = F(F^{m-1}(\lambda x.a))$, so

$$\begin{aligned}&(\mu(\lambda x.a)F)(\nu(\lambda x.b)F)\\&= F(F^{m-1}(\lambda x.a))(F^n(\lambda x.b))\\&= (F^n(\lambda x.b))(F^{m-1}(\lambda x.a))\end{aligned}$$

and the result follows by induction. □

We refer the reader to Krivine [Kri87] for the proof that this algorithm is not representable in the second-order lambda calculus.

4 Semantics of the second-order lambda calculus

In this chapter we shall discuss the semantics of the second-order lambda calculus. We shall discuss two different notions of what it means to give a model for the calculus, and present one actual model as it appears in the two formalisms. The model (or more accurately family of models) that we shall present is the PER model. It is possible to find domain-theoretic models, but these are more complicated to present. It is also possible to find models as small subcategories of toposes, and we shall discuss these briefly at the end of the chapter.

The major difference between the two flavours of model is the interpretation put on contexts. In the first kind of model [BM84], a context is regarded as a list of identifiers which must themselves be given meaning. Thus the semantics of an expression (or type) is defined relative to an environment which gives meaning to the identifiers listed in the context in which the expression (or type) is defined. In the second kind of model, the notion of context is interpreted abstractly [See87, Pit87]. The effect is that this second form of model is a kind of Lindenbaum algebra for the type theory, while the environment models suffer from a form of extensionality.

In order to give semantics to any kind of type theory one has to explain the meanings of the judgement forms in such a way that the rules become valid. Thus, we have to explain what it means to be a type in the context of a set of type variables, what it means to be a context (including individual variables), and what it means to be a term of a given type in the context of a set of type variables, and individual variables bound to types. In a Bruce-Meyer style environment model, all comes down to identifying closed types and closed terms. We shall see this in detail later, but for now will begin with our account of PER models.

PER models are constructed using partial equivalence relations, that is to say relations which are symmetric and transitive, but not necessarily reflexive. In more elementary terms, if we write aAb to mean that a and b are related by the binary relation A, then A is a partial equivalence relation if whenever aAb then bAa, and whenever aAb and bAc then aAc. It is not necessarily always the case that aAa, though this will hold for any element related to something else.

The classical PER model is constructed using partial equivalence relations on the natural numbers. If A and B are PER's then a natural number n gives a map of PER's $A \to B$ if whenever aAa then $\{n\}(a)$ is defined, and whenever aAa' then $\{n\}(a)B\{n\}(a')$. Here $\{n\}(a)$ denotes "Kleene application", the result of evaluating the n'th partial recursive function on the argument a.

You should think of a PER A as being a way of describing its set of equivalence classes (for which I shall write $\bar{A}$). So if A is a PER and aAa', then a and a' are different notations for the same element of $\bar{A}$. Given this perspective, n gives a map $A \to B$ of PER's if and only if $\{n\}$ generates a map $\bar{A} \to \bar{B}$, i.e. it "tracks" a map of equivalence classes. This viewpoint also reveals that there is a natural "extensional" notion of equality on maps, in which n and n' denote the same map $A \to B$ if and only if they track the same map $\bar{A} \to \bar{B}$. We can put this in elementary terms, n and n' denote the same map iff:

$$aAa' \Rightarrow \{n\}aB\{n'\}a'.$$

This itself a partial equivalence relation, and so the maps $A \to B$ themselves form a PER $[A \to B]$ where

$$\begin{aligned} n[A \to B]n' \;\text{ iff }\; & \forall a.aAa \Rightarrow \{n\}a\downarrow \& \{n'\}a\downarrow \\ & \text{and } \forall a,a'.aAa' \Rightarrow \{n\}aB\{n'\}a'. \end{aligned}$$

This will give us our interpretation of the function types.

We have followed the traditional route in using the natural numbers and Kleene application to give our model. We should note, however, that the construction is not restricted to this particular structure. Technically we can use PER's on any partial combinatory algebra. Formally, a *partial combinatory algebra* is a set A together with a partial binary operation intended to represent function application, which has two elements $\mathbf{S}$ and $\mathbf{K}$ such that $\mathbf{S}XY$ and $\mathbf{K}X$ are always defined, and $\mathbf{S}XYZ = (XZ)(YZ)$ and $\mathbf{K}XY = X$, where both equalities are to be interpreted as meaning that if either side is defined, then so is the other, and they are equal. So a combinatory algebra is a partial combinatory algebra in which the application operation is total.

Important partial combinatory algebras include combinatory algebras, such as β- or $\beta\eta$-equivalence classes of lambda terms. There are however genuinely partial examples, such as the natural numbers with Kleene application, and other examples even more closely related to the compilation of functional programming languages. These latter provide much of the justification for looking at this style of model.

Let us return to the construction of our model. Type variables range over PER's. In the environment form of the model, to interpret a judgement

$$X_1, \ldots, X_n \vdash \sigma \ \mathsf{type}$$

we must parametrize the interpretation with respect to environments χ binding the X_i to PER's $\chi(X_i)$. Thus the interpretation of a judgement of this form is a function from environments to PER's.

The interpretation of a context

$$X_1, \ldots, X_n; x_1 : \tau_1, \ldots x_m : \tau_m$$

is a type environment, χ, binding each of the X_i to PER's together with a value environment ρ binding each of the x_j. Of course we know that as a precursor to forming the context, we must first have

$$X_1, \ldots, X_n \vdash \tau_j \ \mathsf{type}$$

for each j, and so impose the consistency requirement that $\rho(x_j)$ be an equivalence class of the PER

$$[\![X_1, \ldots, X_n \vdash \tau_j \ \mathsf{type}]\!]\chi.$$

Finally, to interpret

$$X_1, \ldots, X_n; x_1 : \tau_1, \ldots x_m : \tau_m \vdash e : \tau$$

we need both a type environment χ and a value environment ρ where

$$[\![X_1, \ldots, X_n; x_1 : \tau_1, \ldots x_m : \tau_m \vdash e : \sigma]\!]\chi\rho$$

is an element (equivalence class) of the PER

$$[\![X_1, \ldots, X_n \vdash \sigma \ \mathsf{type}]\!]\chi.$$

Type and value constructions are interpreted in a denotational fashion, that is to say we interpret the rules defining the constructions, rather than giving an inductive definition of an interpretation of the syntax of the calculus. The rule $(\rightarrow^t)$ is

$$\frac{\Gamma \vdash \sigma \ \mathsf{type} \qquad \Gamma \vdash \tau \ \mathsf{type}}{\Gamma \vdash \sigma \rightarrow \tau \ \mathsf{type}}.$$

To interpret $[\![\Gamma \vdash \sigma \rightarrow \tau \ \mathsf{type}]\!]\chi$ we first suppose that $[\![\Gamma \vdash \sigma \ \mathsf{type}]\!]\chi = A$ and $[\![\Gamma \vdash \tau \ \mathsf{type}]\!]\chi = B$. Then we define

$$[\![\Gamma \vdash \sigma \rightarrow \tau \ \mathsf{type}]\!]\chi = [A \rightarrow B],$$

where $[A \rightarrow B]$ is the PER of functions from A to B, defined above.

Defining the intepretation of the rule $(\forall^t)$ is a little more complex.

$$\frac{\Gamma, X \vdash \tau \ \mathsf{type}}{\Gamma \vdash \forall X.\tau \ \mathsf{type}}$$

To define the meaning of $[\![\Gamma \vdash \forall X.\tau \ \mathsf{type}]\!]$ we must first know the meaning of $[\![\Gamma, X \vdash \tau \ \mathsf{type}]\!]$. However, χ is not a suitable environment for this; we also need to find a binding for X. This could be any PER, so, playing fast and loose, we can regard

$$[\![\Gamma, X \vdash \tau \ \mathsf{type}]\!]\chi$$

as a function PER $\to$ PER, whose value at the PER A is

$$[\![\Gamma, X \vdash \tau \ \mathsf{type}]\!](\chi\hat{\ }[X \mapsto A]).$$

We now try to follow Strachey's idea that an element of a polymorphic type should be an algorithm which is uniform across all types. This suggests that n should be in

$$[\![\Gamma \vdash \forall X.\tau \ \mathsf{type}]\!]\chi$$

if and only if n is in

$$[\![\Gamma, X \vdash \tau \ \mathsf{type}]\!](\chi\hat{\ }[X \mapsto A])$$

for all A. Moreover, n should denote the same element as n' if and only if they denote the same element of $[\![\Gamma, X \vdash \tau \ \mathsf{type}]\!](\chi\hat{\ }[X \mapsto A])$ for all A. Rewriting this in elementary terms gives us

$$[\![\Gamma \vdash \forall X.\tau \ \mathsf{type}]\!]\chi = \bigcap_{A \in \mathrm{PER}} [\![\Gamma, X \vdash \tau \ \mathsf{type}]\!](\chi\hat{\ }[X \mapsto A]).$$

We must also interpret the term-formation rules and then the final stage is to verify that the equality rules (particularly the beta and eta rules) are satisfied.

The significant term-formation rules are:

$$(V^v) \qquad \frac{\Gamma; \Delta, x : \sigma \ \mathsf{context}}{\Gamma; \Delta, x : \sigma \vdash x : \sigma}$$

$$(\to^I) \qquad \frac{\Gamma; \Delta, x : \sigma \vdash e : \tau}{\Gamma; \Delta \vdash \lambda x : \sigma.e : \sigma \to \tau}$$

$$(\to^E) \qquad \frac{\Gamma; \Delta \vdash u : \sigma \to \tau \qquad \Gamma; \Delta \vdash v : \sigma}{\Gamma; \Delta \vdash uv : \tau}$$

$$(\forall^I) \qquad \frac{\Gamma, X; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \Lambda X.e : \forall X.\tau} \qquad \text{(X not free in the type of any variable in Δ)}$$

$$(\forall^E) \qquad \frac{\Gamma; \Delta \vdash e : \forall X.\tau \qquad \Gamma \vdash \sigma \ \mathsf{type}}{\Gamma; \Delta \vdash e[\sigma] : \tau[X := \sigma]}$$

The rule (V^v) is interpreted as follows:
if $\Gamma; \Delta, x : \sigma$ context, then

$$[\![\Gamma; \Delta, x : \sigma \vdash x : \sigma]\!]\chi\rho = \rho(x).$$

The interpretation of the rule $(\to^E)$ is straightforward: if $[\![\Gamma;\Delta \vdash u{:}\,\sigma \to \tau]\!]\chi\rho = n$ and $[\![\Gamma;\Delta \vdash v{:}\,\sigma]\!]\chi\rho = m$, then

$$[\![\Gamma;\Delta \vdash uv{:}\,\tau]\!]\chi\rho = \{n\}(m).$$

The interpretation of $(\to^I)$ shows similarities to the $(\forall^t)$ rule: $[\![\Gamma;\Delta \vdash \lambda x : \sigma.e{:}\,\sigma \to \tau]\!]\chi\rho$ is an index for the partial recursive function

$$\lambda m.[\![\Gamma;\Delta, x : \sigma \vdash e{:}\,\tau]\!]\chi(\rho\bar{}[x \mapsto m]).$$

Note that this function is total on the required domain.

The rules for the polymorphic types make explicit that polymorphic abstraction and instantiation do not affect the "algorithm". If

$$[\![\Gamma, X;\Delta \vdash e{:}\,\tau]\!](\chi\hat{}[X \mapsto A])\rho = n,$$

then

$$[\![\Gamma;\Delta \vdash \Lambda X.e{:}\,\forall X.\tau]\!]\chi\rho = n,$$

and if

$$[\![\Gamma;\Delta \vdash e{:}\,\forall X.\tau]\!]\chi\rho = n,$$

then

$$[\![\Gamma;\Delta \vdash e[\sigma]{:}\,\tau[X := \sigma]]\!]\chi\rho = n.$$

In all these rules we have abused integer notation to mean "equivalence class in the relevant PER". We really should check that the definitions are all stable under this equivalence.

EXERCISES. 1. Verify that the definitions given are stable under equivalence in the relevant PER's.
2. Give interpretations for the remaining structural rules.
3. Verify that the beta and eta rules are validated.
4. Verify that the collection of partial equivalence relations on any (partial) combinatory algebra A can be used in a similar fashion to provide a model.

We will not attempt here to give a formal definition of the Bruce-Meyer form of an environment model, the reader is referred instead to their paper [BM84]. We shall instead discuss another formalization of what it means to give a model for a calculus such as this. The major difference between the two forms, as we mentioned above, lies in the interpretation of contexts. In an environment model a context is interpreted in a very concrete fashion (as an environment). However, it is also possible to interpret a context abstractly as an indexing structure. Seen from this point of view a type context is a collection of type variables. If we fix the context, then we can define what it means to be a type relative to this context, what it means to be a value context, and what it means to be an element of a type relative to some value context. There is more, however. If we have an n-tuple of types defined relative to some other context, then we can substitute these types for the type variables in our original context. Suppose, for example, we consider the type $X_1, X_2 \vdash \forall Z.X_1 \to Z \to X_2$ type, then we can substitute $Y \vdash Y$ type for both X_1 and X_2 to get $Y \vdash \forall Z.Y \to Z \to Y$ type. This gives us a mapping taking types defined relative to one context to types defined relative to another, and also individuals defined relative to one context to individuals defined relative to another.

The best way to formalize this is to use the technology of indexed category theory. We have a category `TypeContext`, whose objects are contexts, and whose morphisms are substitutions of types for type variables. So for the second order lambda calculus, the objects of `TypeContext` are n-tuples of type variables, identified up to alpha-equivalence. (This means that the variables are viewed just as place-holders, and hence that the category has as objects the natural numbers, where we think of n as representing the context $(X_1, \ldots, X_n)$.) The morphisms $(Y_1, \ldots, Y_m) \to (X_1, \ldots, X_n)$ are substitutions of types defined over the Y_i for each of the X_j, and thus n-tuples of types $(\sigma_1, \ldots, \sigma_n)$ such that $(Y_1, \ldots, Y_m \vdash \sigma_j$ type for all j. Composition, and the identities are obvious. There is a contravariant functor from this category to another whose objects are the kind of structure we want to use to model types, and whose morphisms are the homomorphisms appropriate to this sort of structure. The question is what sort of structure is this?

Note that if we confine ourselves to a particular type context, then the types and terms we can define form a model for the simply typed lambda calculus. This means that the structure we are going to use must be some sort of model for the simply typed lambda calculus. Now, in the simply typed lambda calculus, we have to say what it means to be a type, and what it means to be a term defined with respect to a given context of variables bound to types. This is the same problem at the level of terms as we have just faced at the level of types. The ideologically sound solution is to mimic the treatment we have given above, and to define a structure indexed over the category of contexts and (simultaneous) substitutions (of terms for individual variables). For each value context we should have another structure modeling types and terms with that value context. This can be done using another category. Let's fix a type variable context, Γ, and a value context Δ. Then the objects of this category are types σ such that

$$\Gamma \vdash \sigma \ \mathsf{type}.$$

The morphisms $\sigma \to \tau$ are beta-eta equivalence classes of terms such that

$$\Gamma; \Delta, x : \sigma \vdash e{:}\,\tau.$$

(Note that we do not need function types to make this definition.)

It is perfectly possible to do this for the form of the lambda calculus we have given. However, in most of the published work which adopts this approach (e.g. [Pit87], [See87]), the definition of model is given for the extended calculus with genuine products (i.e. both beta and eta laws hold for the products). The reason for this is that it enables a considerable formal simplification. Models of the simply typed lambda calculus with products are cartesian closed categories. This is dealt with in considerable detail in [LS86], but not quite in the terms we have been using.

The category of contexts and substitutions always has finite products (though linear logicians would disagree). If the type theory has function types, then the category of contexts is cartesian closed (even if the type theory does not have products). However, if the type theory has genuine products, then there is an equivalence between the category of contexts and the category of types as defined above. In this equivalence, a context

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n$$

is associated with the type

$$\sigma_1 \times \ldots \times \sigma_n.$$

This observation allows us to conflate the category of contexts with the category of types which is supposed to be indexed over it, at least in the case of a type theory with products.

Thus to model the second order lambda calculus with products, we give a category (supposed to be the category of type contexts and type substitutions), and a contravariant functor from this category to the category of cartesian closed categories. Can we say anything more? Well, we need to be careful about alpha-congruence in type contexts, so the simplest thing to do is to think of a type context as an ordered set of place-holders. This means that the objects of our category of contexts correspond to the natural numbers. The maps correspond to substitutions, so $hom(m, n)$ is an n-tuple of types with at most m free variables. This means that the object n is the product of n copies of 1. Moreover, $hom(m, 1)$ corresponds to the types with at most m free variables. Thus `TypeContext` is the Lawvere algebraic theory of types. As usual with Lawvere algebraic theories, there is an unfortunate clash of notation. The terminal object of the category is 0, not 1.

Finally, we have still not said anything about the interpretation of polymorphic types. The definition of a polymorphic type involves two type contexts, Γ and its extension by an additional type variable X. Suppose $\Gamma \vdash \sigma$ type, and $\Gamma, X \vdash \tau$ type, then we also have $\Gamma, X \vdash \sigma$ type, and a morphism from σ to τ in the category of types defined over (Γ, X) is given by

$$\Gamma, X; x : \sigma \vdash e{:}\,\tau.$$

So we have

$$\Gamma; x : \sigma \vdash \Lambda X.e{:}\,\forall X.\tau.$$

Moreover a typical morphism σ to $\forall X.\tau$ in the category of types defined over Γ is given by

$$\Gamma; x : \sigma \vdash e'{:}\,\forall X.\tau.$$

In this case we have

$$\Gamma, X; x : \sigma \vdash e'[X]{:}\,\tau.$$

Now, $(\Lambda X.e)[X] = e$, by $(\forall^{\beta})$, and $\Lambda X.e'[X] = e'$ by $(\forall^{\eta})$. Furthermore, we can check that these correspondences are natural. (Fortunately the beta and eta rules we have given do a bit more than imply the correspondences above!) It follows that polymorphic quantification is right adjoint to the natural inclusion given by weakening. Thus, we demand that our model should have these right adjoints. (Note: this is the same argument that at the value level tells us that function space is right adjoint to product.)

This is still not quite the whole of the story. Polymorphic abstraction is preserved by substitution (of types for type variables), and we have still not imposed sufficient requirements to force this. We must also do so explicitly. The way this is generally handled is through the imposition of the so-called "Beck-Chevalley" conditions. For the full details of this form of model see [Pit87], but notice that once we have given the category of contexts, and the functor from there to the category of (cartesian closed) categories, then the rest of the structure is defined uniquely up to isomorphism if it exists.

Thus to describe the PER model in this formalism, we must first give the category of type contexts. As always, this has the natural numbers as objects, and what changes is the morphisms. A morphism m to 1 is a function $\mathrm{PER}^m \to \mathrm{PER}$. A morphism m to n is an n-tuple of such functions (or a function $\mathrm{PER}^m \to \mathrm{PER}^n$). Think of a function $F : \mathrm{PER}^m \to \mathrm{PER}$ as a type $F(X_1, \ldots, X_m)$, with $F(A_1, \ldots, A_m)$ giving the value of the type $F(X_1, \ldots, X_m)[X_1 := A_1, \ldots, X_m := A_m]$.

The cartesian closed category over m is supposed to represent the category of types with m free variables. Each of those free variables ranges over PER's. The category is therefore the product of $\mathsf{ob}(\mathrm{PER})^m$ copies of the category PER. This is cartesian closed, with the products and exponentials being calculated componentwise. Polymorphic quantification is supposed to take us from the category over $m+1$ to the category over m. Suppose $F : \mathsf{ob}(\mathrm{PER})^{m+1} \to \mathrm{PER}$ is an object of the category over $m+1$, then $\forall X_{m+1}.F$ is interpreted as

$$(\forall X_{m+1}.F)(X_1, \ldots, X_m) = \bigcap_{A \in \mathsf{ob}(\mathrm{PER})} F(X_1, \ldots, X_m, A),$$

with quantification over the other variables being interpreted similarly.

Exercises. 1. Verify that this is a cartesian closed category.
2. Work out the action of the functor on these categories, and verify that the cartesian closed structure is preserved.
3. Verify that intersection of PER's indeed gives the appropriate right adjoint.

Some time ago, Dana Scott [Sco80] observed that some meat could be put on the rather slender thought that types were rather like sets and function types like function spaces in set theory, by observing that any model for the simply typed lambda calculus (i.e. any cartesian closed category) could be regarded as a full subcategory of a topos. Now a topos is a model for a form of intuitionistic set theory, and the content of Scott's observation is not simply that this can be done, but that it can be done in such a way that the arrow types in the type theory are interpreted as function types in the topos.

This picture was extended to the second order lambda calculus by Pitts in [Pit87]. Pitts showed that any model for the second order lambda calculus could be regarded as a full subcategory of the topos in such a way that the arrow types were interpreted as function spaces in the topos, and moreover, there was an object in the topos which played the role of the set of all types, and the polymorphic types were interpreted as products over the set of all types. This is not something you can do in classical set theory (except in very trivial cases, such as the one in chapter two). So in Pitts' formalism, an object of the polymorphic type $\forall X.T(X)$ is an element of

$$\prod_{A \in \mathsf{Type}} T(A).$$

In other words it is simply a function that takes a type A and returns a value of type $T(A)$.

However, for the particular case of the PER model, it turns out to be more interesting not to regard the model as given by a subcategory of the topos constructed by Pitts (which is a Grothendieck topos), but to regard it as a subcategory of the Effective topos, which is not. This approach was originally suggested by Moggi (predating the work of Pitts), and has since been followed up by many people. The category has strong completeness properties, the definitive treatment of which is in the work of Hyland, Rosolini and myself, [Hyl87, HRR90, Rob89], and which enable us to model many other stronger type theories using PER and related categories (see for example Phoa [Pho91]).

What this work does is to prove the soundness of certain forms of set-theoretic reasoning about datatypes in the second-order lambda calculus. Pitts' results form a completeness theorem, in fact a completeness theorem valid for type theories at least up to the strength of F_ω, the full higher-order variant of this calculus. The strong completeness properties of PER allow us to use this set-theoretic style to obtain rapid naive interpretations for many extensions of the calculus, and hence soundness results for these extensions. So it is no surprise to see it wheeled out time and time again.

Finally, we should note that there are also domain-based models of the calculus. Some of these arise as internal categories of domains in the effective topos (as in work of Phoa and Rosolini), and lead to yet more PER models. Others can be given a more elementary treatment. Not all of these are denotational models in the sense that constructions are defined for all types, some are simply structures which support the interpretation of the syntax of the pure second-order lambda calculus, and validate its true equations. Suitable references here are McCracken [McC79], MacQueen, Plotkin and Sethi [MPS86], and Girard [Gir89].

5 The Strong Normalization and Church-Rosser Theorems

In this chapter we shall look at two powerful results about the pure second order lambda calculus: the Church-Rosser and the strong normalization theorems. The strong normalization theorem says that there are no infinite reduction sequences, and hence that whatever strategy we follow for reducing a term, we will eventually end up with something that cannot be reduced further. The Church-Rosser theorem implies that a term can have only one normal form, which is thus independent of the evaluation strategy.

If we have a reduction system (we shall use $\rightarrow$ as the symbol for a single reduction, and $\rightarrow^*$ as the symbol for a sequence of zero or more reductions), then the Church-Rosser (or strong confluence) property for that reduction relation says:
whenever $A \rightarrow^* B$ and $A \rightarrow^* B'$ (i.e. possibly by many reduction steps), then there is a fourth term C such that both B and B' can be reduced to C ($B \rightarrow^* C$ and $B' \rightarrow^* C$).

This is a stronger property in general than the similar weak Church-Rosser (or weak confluence) property:
whenever $A \rightarrow B$ and $A \rightarrow B'$ (i.e. by a single reduction step), then there is a fourth term C such that both B and B' can be reduced to C ($B \rightarrow^* C$ and $B' \rightarrow^* C$).

An example can be given by the rewrite system in which A rewrites either to R or to LB, and B rewrites either to L or to RA. This system is weakly confluent, but not confluent, since A rewrites to both R and LL which are normal forms and therefore cannot be merged. (This is an adaption of an example due to Huet.)

However, in the presence of strong normalization, the weak and strong properties are equivalent (observe that in the counterexample A rewrites to LRA, $LRLRA$, $LRLRLRA, \ldots$). This observation will make the proof of the confluence property substantially easier for us.

However, we shall begin by sketching a proof of the strong normalization property, or rather we shall begin by sketching a non-proof.

Recall the PER model from the last chapter. We remarked that this model need not be based on the natural numbers, but that similar models could be constructed on any partial combinatory algebra. In particular we can use the algebra of untyped lambda terms under beta-eta equivalence.

If one follows through the definitions given in the last chapter, then the interpretation of any polymorphic term is easily seen to be given by its type erasure. This is of no use to us as it stands, but let's suppose that instead of the algebra of all lambda terms, we were to use only the strongly normalizing ones. Then this model would tell us that the type erasure of any second order lambda term must be strongly normalizing. However, $(\beta^{\rightarrow})$ and $(\eta^{\rightarrow})$ reductions of a second-order term correspond precisely to beta and eta reductions of its type erasure. Moreover, $(\beta^{\forall})$ and $(\eta^{\forall})$ reductions of the second order term leave the type erasure unchanged, and reduce the number of Λ's in the term. It follows that we can lift the strong normalization property of the type erasure to the second order term.

The flaw in this argument is that the strongly normalizing untyped terms do not form a partial combinatory algebra, and so we can't form such a model. Nevertheless, there are obvious connections between the fallacious argument just given, and Girard's proof of strong normalization which uses an interpretation of the syntax of the second order lambda calculus (rather than an formal model) in which the types are sets of strongly normalizing terms which he calls "candidats de réductibilité". This proof is discussed in Girard's book [Gir89], but there is also an excellent, careful and extremely readable account in Gallier [Gal90].

We sketch Gallier's rewriting of Girard's proof. This is a model-theoretic argument, at least to the extent that it involves a structure which interprets types and terms. It does not, however, give any support to the interpretation of equality.

Suppose $\mathcal{C}$ is a family of sets of (open untyped) lambda terms, then $\mathcal{C}$ is said to be $\mathcal{T}$-closed if it satisfies the following conditions:

(i) every $C \in \mathcal{C}$ is non-empty

(ii) if $S, T \in \mathcal{C}$, then so is the "function space" $[S \to T]$, which is defined as

$$[S \to T] = \{M \in \Lambda | \forall N \in S. MN \in T\}.$$

(iii) $\mathcal{C}$ is closed under $\mathcal{C}$-indexed intersection: if $(A_C)_{C\in\mathcal{C}}$ is a $\mathcal{C}$-indexed family of sets in $\mathcal{C}$, then $\bigcap_{C\in\mathcal{C}} A_C \in \mathcal{C}$.

If we are given a $\mathcal{T}$-closed family $\mathcal{C}$, then we can define an interpretation of the types of the second-order lambda calculus, in which the function types are interpreted using the function space defined above, and the universally quantified types are interpreted by intersection, just as for the PER model. There is an obvious similarity between this construction and the PER model of the last chapter. However, whereas the PER model was constructed with an idea of interpreting the extensional equality of terms (notably in the definition of its function space), that is ignored here.

In order to extend the interpretation to terms, we need a little more. A family $\mathcal{C}$ of sets of untyped lambda terms is said to be a family of "reducibility candidates" if it is $\mathcal{T}$-closed, and if for every $C \in \mathcal{C}$:

(CR1) C contains all individual variables x.

(CR2) Whenever M and N are terms appearing in some element of $\mathcal{C}$, then whenever $M[x := N]$ is in C, then so is the term $(\lambda x.M)N$.

The aim of the proof is to construct a particular family of reducibility candidates all of whose elements contain only strongly normalizing terms. For a family of reducibility candidates, the interpretation of a second order term is its type erasure. This gives us a proof that the type erasure of any second order term is strongly normalizing in the untyped calculus. Now the problem with the fallacious proof given above had nothing to do with the part that showed that the strong normalization property could be lifted from the type erasure of a second order term to the term itself, and so that can justifiably be used here. Thus we obtain a genuine proof that the second order lambda calculus is strongly normalizing.

The hard part of the argument is to show the existence of a suitable family. To do this, we strengthen the closure conditions on the sets within the family by making them right-stable under application.

A non-empty set S of untyped lambda terms is said to be "closed" if whenever $Mx \in S$ then M is in S. A subset C of S is said to be (S)-saturated if whenever x is a variable, then $xN_1 \dots N_n \in C$ for any $N_1, \dots, N_n \in S$ (cf. (CR1)), and whenever $(M[x := N])N_1 \dots N_n \in C$, then $(\lambda x.M)NN_1 \dots N_n \in C$ $(M, N, N_1 \dots N_n \in S)$ (cf. (CR2)).

We now have the following proposition, which Gallier credits to Girard, Tait and Mitchell.

PROPOSITION 5.1. *If S is a non-empty set of lambda terms which is itself S-saturated, then the set of saturated subsets of S forms a family of reducibility candidates.*

The proof of this is not difficult. Nor is it now too hard to show that the sets of untyped lambda terms which are strongly normalizing under beta or beta-eta reduction are closed and self-saturated, and so to complete the proof. It does however call for a certain amount of attention to detail, in which we do not attempt to compete with Gallier, and accordingly we refer the reader to him for the detailed exposition.

Gallier gives the proof in slightly greater generality than we have allowed. He allows constants in the theory, but not equations between terms involving the constants (for obvious reasons). As far as the proof is concerned, constants are treated much as if they were variables, and so we need to add closure conditions referring to them both to (CR1) and to the first saturation condition.

Gallier also uses the same techniques to give a proof of the Church-Rosser property (the set of terms all of whose reductions satisfy the Church-Rosser property is closed and self-saturated). We do not in fact need to use anything quite so advanced. Once we know that strong normalization holds, we can use the fact that the strong confluence property follows from the weak. The weak confluence property can be proved via a relatively simple argument, since the two redexes involved cannot overlap.

The results we have given in this chapter show that if we write any program in the second order lambda calculus, then its evaluation will always terminate, whatever order of evaluation we use. Moreover, the result is deterministic. This has been used in the past to make large claims about the utility of the calculus (see for example Reynolds [Rey85]). In the next and final chapter we shall examine these.

6 Summary and further directions

We began these notes by observing that type theories had two faces, one looking towards logic and the other towards programming languages. Although when type theories were born the dominant face was certainly that looking towards logic, nowadays it is the programming face by which their beauty is judged. We have to ask how the second order lambda calculus stands up.

We have seen how the pure calculus has some very strong properties. Any reduction sequence for any term must terminate, and terminate deterministically, in that normal forms are unique. This means that if we translate programs into the calculus for evaluation, then the evaluation process must always terminate, even if we mix different evaluation strategies. However, it is clear that we want to write programs not intended to terminate. For example, I do not want my operating system suddenly to pack up and inform me that the answer is 42. This seems at first sight to be a serious criticism of the utility of any strongly normalizing calculus, but it is not as bad as it seems. There is some work, more from the software engineering side, that has looked at the topology of non-terminating systems. The viewpoint adopted is that a non-terminating system often has the top-level structure of a finite automaton, in which, although the automaton's computation may never terminate, we want to be sure that each state transition does. Programs such as operating systems and editors can certainly be viewed in this light. Theories such as the second order lambda calculus can certainly be used to analyse the components of such a system, even if they inevitably have problems with the system as a whole.

We have spent a great deal of time talking about the representation of datatypes in the pure second order lambda calculus. We have seen that it is possible to represent a large class of datatypes in a reasonably satisfactory manner, but not totally so. We have tended to lose the (computationally largely unnecessary) eta rules for the types. Should we care about this? I think the argument is largely operational. We certainly want to write programs in a language with an extended class of datatypes. On the other hand we would like to be able to use a small, and so hopefully theoretically tractable, language to prove their properties in. In order to do this we must have a language into which we can translate the types, terms and reduction mechanisms of the extended calculus. We have discussed this at some length for the case of products, and somewhat more briefly for algebraic types. We have seen that if the extended language has a call-by-value regime, then we can translate the reduction rules for all these types, and we have sketched out how to cope with a lazy form of product (and there is nothing special about products here, we can equally well define lazy versions of the other types). Perhaps, though, the crucial case to consider is that of the reals. There is no good natural representation of the reals. We could perhaps use a type of sequences of natural numbers implementing a representation by continued fractions, but, though practicable, this is hardly elegant. From a certain perspective that does not matter. The existence of the translation may be enough to prove properties such as termination or independence of evaluation order. To prove these properties we do not have to use the details of the translation, but merely that it takes a reduction in the extended language to some possible sequence of reductions in the target language. If we want more refined properties, we may have to use the details of the representation, and this may well become painful.

Thus, the Church-Rosser and strong normalization theorems are surprisingly robust. What happens though if we extend the calculus by adjoining additional type constructors, and insisting on eta reductions? There is a fair amount of material on pure calculi, involving new constructors. This is generally buried at the back of papers on the simpler versions of calculi (e.g. in [Gal90]). There is very little on extended calculi with new base types, operations defined on those new base types, and, crucially, with equations on the new operations. This can destroy both Church-Rosser and strong normalization properties very rapidly. However, there is some hope if the new operations given form an confluent terminating algebraic rewrite system, and this is the approach adopted in the only work I know in this area, Breazu-Tannen [BT88].

So, it seems that we can get some mileage out of the fact that a program, or a type, could be implemented in the second-order lambda calculus. Some authors have used this together with the fact that the calculus contains all feasible functions to suggest that programs *should* be implemented in the calculus. This seems to me to be unwise. The price we pay for a strongly normalising calculus is inefficient algorithms (we have seen one example). I do not think it is realistic to expect a compiler to optimise all of this out.

Model-theoretically things do not seem to be in bad shape. There are now a fair number of models for this form of type theory. I think the PER model is particularly simple to understand, and it has the attractive feature that it extends well to stronger type theories. Moreover there are arguments which, while they fall short of formal completeness theorems, suggest that PER-like models of fairly general. Briefly, if we consider (denotations of) the terms in a higher-order language, then these form a set with a partial applicative structure (some things are functions which can be applied to others). This falls short of being a partial combinatory algebra, largely because although we may well be able to argue for the existence of **K**, **S** is more problematic. However, if our base types are represented as subsets of this algebra, then first-order function types are most likely PER's (since different terms give extensionally equal functions). Moreover, they are sub-PER's of the function space defined as in the PER model. Now, let's go back to Strachey's original analysis of the notion of polymorphism. Strachey's idea seems to have been that a function was parametrically polymorphic, if it had the same algorithm at each type. In order to make sense of this, we must assume that whatever we are using to denote values is at least potentially untyped, and that polymorphic instantiation is capable of acting as the identity. This gives us that polymorphic types are given by some subset of an intersection. In other words, if we are to model faithfully Strachey's notion of polymorphism, then we expect to find something very like a PER model.

Finally, again showing my personal bias, I think some of the most interesting recent work on the semantics of polymorphic lambda calculi has been a series of attempts to characterise a suitable notion of parametricity. By and large these have not been attempts to formalise Strachey's notion, but have been based on other ideas. For example, Reynolds has linked the concept firmly to the notion of data abstraction. Out of a long list of work here I will cite the work of Reynolds and his student Ma, [MR91]. [FRR92] also contains some suggestive preliminary results obtained in collaboration with Peter Freyd and Pino Rosolini. I expect further developments soon.

References

[Abr90] S. Abramsky. Computational interpretations of linear logic. Technical Report DOC 90/20, Imperial College, London, 1990.

[BB85] C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theor. Comp. Sci.*, 39:135–154, 1985.

[Bie92] G. Bierman. *Type Systems, Linearity and Functional Programming Languages.* PhD thesis, paper presented to the CLICS workshop, Aarhus, 1992.

[BM84] K.B. Bruce and A.R. Meyer. The semantics of second order polymorphic lambda calculus. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, number 173 in Lecture Notes in Computer Science, pages 131–144, Berlin, 1984. Springer.

[Bos92] D.C. Bosley. *The Categorical Fragment of the Polymorphic Lambda Calculus.* PhD thesis, University of Sussex, in preparation, 1992.

[BT88] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, pages 82–90. Computer Society Press of the IEEE, 1988.

[CS92] J.R.B. Cockett and D. Spencer. Strong categorical datatypes, I. In *Category Theory 1991.* Canadian Mathematical Society Proceedings, 1992.

[DHM91] B.F. Duba, R. Harper, and D. MacQueen. Typing first-class continuations in ML. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM Press, 1991.

[FRR92] P.J. Freyd, E.P. Robinson, and G. Rosolini. Functorial parametricity. In A. Scedrov, editor, *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science, June 22-25, 1992, Santa Cruz, California*, pages 444–452, Los Alamitos, California, 1992. IEEE Computer Society Press.

[Gal90] J. Gallier. On Girard's "Candidats de réductibilité". In P. Odifreddi, editor, *Logic and Computer Science*, pages 123–203. Academic Press, 1990.

[Gir71] J-Y. Girard. Une extension de l'interprétation de Gödel. In J.E. Fenstad, editor, *Proc. of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Company, Amsterdam, 1971.

[Gir72] J-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique de l'ordre supérieur.* Thèse de doctorat d'état, Paris, 1972.

[Gir89] J-Y. Girard. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, 1989. translated and with appendices by P. Taylor and Y. Lafont.

[HRR90] J.M.E. Hyland, E.P. Robinson, and G. Rosolini. The discrete objects in the effective topos. *Proceedings of the London Mathematical Society*, 60(3):1–36, 1990.

[Hyl87] J.M.E. Hyland. A small complete category. In *Proc. of the Conference on Church's Thesis: Fifty Years Later*, 1987.

[Kri87] J-L. Krivine. Un algorithme non-typable dans le système F. *C. R. Acad. Sc. Paris, t. 304, Série I., n^o 5*, pages 123–126, 1987.

[Lei83] D. Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *24th Annual IEEE Symposium on the Foundations of Computer Science, Tucson, Arizona*, pages 460–496, 1983.

[LS86] J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, 1986.

[McC79] N.J. McCracken. *An investigation of a programming language with a polymorphic type structure.* PhD thesis, Computer and Information Science, Syracuse University, 1979.

[MPS86] D. MacQueen, G.D. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[MR91] Qing-Ming Ma and J.C. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In *Proceedings of the 1991 Mathematical Foundations of Programming Semantics Conference.* Springer-Verlag, 1991.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *1988 ACM Conference on Lisp and Functional Programming*, pages 174–183. ACM, ACM Press, 1988.

[Pho91] W.K.S. Phoa. *Domain Theory in Realizability Toposes.* PhD thesis, Cambridge University, 1991. published as Edinburgh University Computer Science technical report CST-82-91.

[Pit87] A.M. Pitts. Polymorphism is set-theoretic, constructively. In D.H. Pitt, et al., editor, *Category Theory and Computer Science*, number 283 in Lecture Notes in Computer Science, pages 12–39. Springer-Verlag, Berlin, 1987.

[Plo75] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Rey83] J.C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523. North Holland, 1983.

[Rey85] J.C. Reynolds. Three approaches to type structure. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Mathematical Foundations of Software Development,* Lecture Notes in Computer Science 185, pages 97–138, Berlin, 1985. Springer-Verlag.

[Rob89] E.P. Robinson. How complete is PER? In A. Meyer, editor, *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 106–111. Computer Society Press of the IEEE, 1989.

[Sco80] D.S. Scott. Relating theories of the lambda calculus. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: essays on combinatory logic, lambda calculus and formalism*, pages 403–450. Academic Press, 1980.

[See87] R.A.G. Seely. Categorical semantics for higher order polymorphic lambda calculus. *Journal of Symbolic Logic*, 52:969–989, 1987.

[Wan89] M. Wand. Type inference for record concatenation and multiple inheritance. In *Fourth Annual Symposium on Logic in Computer Science*, pages 92–97. IEEE, IEEE Computer Society Press, 1989.

[Wra89] G.C. Wraith. A note on categorical datatypes. In D.H. Pitt, D.E. Rydeheard, P. Dybjer, A.M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science,* Lecture Notes in Computer Science 389, pages 118–127, Berlin, 1989. Springer.