Chapter 4

Types

Types are a means of distinguishing different sorts of data, like booleans, natural numbers and functions, and making sure that these distinctions are respected, by, for example, ensuring that functions cannot be applied to arguments of the wrong type. Why should we want to add types to lambda calculus, and to programming languages derived from it? We can distinguish reasons from logic and reasons from programming.

From the logical point of view, we have seen that the Russell paradox makes it difficult to extend lambda calculus with set theory without contradiction. The Russell paradox may well arise because of the peculiar self-referential nature of the trick involved: we apply a function to itself. Even if it weren't necessary to avoid paradox, we might feel that intuitively we really have a poor grip of what's going on in a system where we can do such things. Certainly applying some functions to themselves, e.g. the identity function $\lambda x.\, x$ or a constant function $\lambda x.\, y$, seems quite innocuous. But surely we would have a clearer picture of what sort of functions lambda terms denote if we knew exactly what their domains and codomains were, and only applied them to arguments in their domains. These were the sorts of reasons why Russell originally introduced types in *Principia Mathematica*.

Types also arose, perhaps at first largely independently of the above considerations, in programming languages, and since we view lambda calculus as a programming language, this motivates us too. Even FORTRAN distinguished integers and floating point values. One reason was clearly efficiency: the machine can generate more efficient code, and use storage more effectively, by knowing more about a variable. For example the implementation of C's pointer arithmetic varies according to the size of the referenced objects. If p is a pointer to objects occupying 4 bytes, then what C programmers write as $p+1$ becomes $p+4$ inside a byte-addressed machine. C's ancestor BCPL is untyped, and is therefore unable to distinguish pointers from integers; to support the same style of pointer arithmetic it is necessary to apply a scaling at every indirection, a significant penalty.

Apart from efficiency, as time went by, types also began to be appreciated more and more for their value in providing limited static checks on programs. Many programming errors, either trivial typos or more substantial conceptual errors, are suggested by type clashes, and these can be detected before running the program, during the compilation process. Moreover, types often serve as useful documentation for people reading code. Finally, types can be used to achieve better modularization and data hiding by 'artificially' distinguishing some data structure from its internal representation.

At the same time, some programmers dislike types, because they find the restrictions placed on their programming style irksome. There are untyped programming languages, imperative ones like BCPL and functional ones like ISWIM, SASL and Erlang. Others, like PL/I, have only *weak typing*, with the compiler allowing some mixtures of different types and casting them appropriately. There are also languages like LISP that perform typechecking *dynamically*, during execution, rather than statically during compilation. This can (at least in principle) degrade runtime performance, since it means that the computer is spending some time on making sure types are respected during execution — compare checking of array bounds during execution, another common runtime overhead. By contrast, we have suggested that static typing, if anything, can improve performance.

Just how restrictive a type system is found depends a lot on the nature of the programmer's applications and programming style. Finding a type system that allows useful static typechecking, while at the same time allowing programmers plenty of flexibility, is still an active research area. The ML type system is an important step along this road, since it features *polymorphism*, allowing the same function to be used with various different types. This retains strong static typechecking while giving some of the benefits of weak or dynamic typing.[1] Moreover, programmers never need to specify *any* types in ML — the computer can infer a most general type for every expression, and reject expressions that are not typable. We will consider polymorphism in the setting of lambda calculus below. Certainly, the ML type system provides a congenial environment for many kinds of programming. Nevertheless, we do not want to give the impression that it is necessarily the answer to all programming problems.

4.1 Typed lambda calculus

It is a fairly straightforward matter to modify lambda calculus with a notion of type, but the resulting changes, as we shall see, are far-reaching. The basic idea is that every lambda term has a *type*, and a term s can only be applied to a term t in a combination $s\ t$ when the types match up appropriately, i.e. s has the type of a function $\sigma \to \tau$ and t has type σ. The result, $s\ t$, then has type τ. This is, in programming language parlance, *strong typing*. The term t must have *exactly*

[1] As well, perhaps, as some of the overheads.

the type σ; there is no notion of subtyping or coercion. This contrasts with some programming languages like C where a function expecting an argument of type `float` or `double` will accept one of type `int` and appropriately cast it. Similar notions of subtyping and coercion can also be added to lambda calculus, but to consider these here would lead us too far astray.

We will use '$t : \sigma$' to mean 't has type σ'. This is already the standard notation in mathematics where function spaces are concerned, because $f : \sigma \to \tau$ means that f is a function from the set σ to the set τ. We will think of types as sets in which the corresponding objects live, and imagine $t : \sigma$ to mean $t \in \sigma$. Though the reader may like to do likewise, as a heuristic device, we will view typed lambda calculus purely as a formal system and make the rules independent of any such interpretation.

4.1.1 The stock of types

The first stage in our formalization is to specify exactly what the types are. We suppose first of all that we have some set of *primitive types*, which might, for example, contain `bool` and `int`. We can construct composite types from these using the function space *type constructor*. Formally, we make the following inductive definition of the set Ty_C of types based on a set of primitive types C:

$$\frac{\sigma \in C}{\sigma \in Ty_C}$$

$$\frac{\sigma \in Ty_C \quad \tau \in Ty_C}{\sigma \to \tau \in Ty_C}$$

For example, possible types might be int, $bool \to bool$ and $(int \to bool) \to int \to bool$. We assume that the function arrow associates to the right, i.e. $\sigma \to \tau \to \upsilon$ means $\sigma \to (\tau \to \upsilon)$. This fits naturally with the other syntactic conventions connected with currying.

Before long, we shall extend the type system in two ways. First, we will allow so-called *type variables* as well as primitive type constants — these are the vehicle for polymorphism. Secondly, we will allow the introduction of other type constructors besides the function arrow. For example, we will later introduce a constructor $\times$ for the product type. In that case a new clause:

$$\frac{\sigma \in Ty_C \quad \tau \in Ty_C}{\sigma \times \tau \in Ty_C}$$

needs to be added to the inductive definition. Once we move to the more concrete case of ML, there is the possibility of new, user-defined types and type constructors, so we will get used to arbitrary sets of constructors of any arity. We will write $(\alpha_1, \ldots, \alpha_n)con$ for the application of an n-ary type constructor con to the arguments α_i. (Only in a few special cases like $\to$ and $\times$ do we use infix syntax, for the sake of familiarity.) For example $(\sigma)list$ is the type of lists whose elements all have type σ.

An important property of the types, provable because of *free* inductive generation, is that $\sigma \to \tau \neq \sigma$. (In fact, more generally, a type cannot be the same as any proper syntactic subexpression of itself.) This rules out the possibility of applying a term to itself, unless the two instances in question have distinct types.

4.1.2 Church and Curry typing

There are two major approaches to defining typed lambda calculus. One approach, due to Church, is *explicit*. A (single) type is attached to each term. That is, in the construction of terms, the untyped terms that we have presented are modified with an additional field for the type. In the case of constants, this type is pre-assigned, but variables may be of any type. The rules for valid term formation are then:

$$\frac{}{v : \sigma}$$

$$\frac{\text{Constant } c \text{ has type } \sigma}{c : \sigma}$$

$$\frac{s : \sigma \to \tau \quad t : \sigma}{s\ t : \tau}$$

$$\frac{v : \sigma \quad t : \tau}{\lambda v.\ t : \sigma \to \tau}$$

For our purposes, however, we prefer Curry's approach to typing, which is purely *implicit*. The terms are exactly as in the untyped case, and a term may or may not have a type, and if it does, may have many different types.[2] For example, the identity function $\lambda x.\ x$ may be given any type of the form $\sigma \to \sigma$, reasonably enough. There are two related reasons why we prefer this approach. First, it provides a smoother treatment of ML's polymorphism, and secondly, it corresponds well to the actual use of ML, where it is never necessary to specify types explicitly.

At the same time, some of the formal details of Curry-style type assignment are a bit more complex. We do not merely define a relation of typability in isolation, but with respect to a *context*, i.e. a finite set of typing assumptions about variables. We write:

$$\Gamma \vdash t : \sigma$$

[2]Some purists would argue that this isn't properly speaking 'typed lambda calculus' but rather 'untyped lambda calculus with a separate notion of type assignment'.

to mean 'in context Γ, the term t can be given type σ'. (We will, however, write $\vdash t : \sigma$ or just $t : \sigma$ when the typing judgement holds in the empty context.) The elements of Γ are of the form $v : \sigma$, i.e. they are themselves typing assumptions about variables, typically those that are components of the term. We assume that Γ never gives contradictory assignments to the same variable; if preferred we can think of it as a partial function from the indexing set of the variables into the set of types. The use of the symbol $\vdash$ corresponds to its general use for judgements in logic of the form $\Gamma \vdash \phi$, read as 'ϕ follows from assumptions Γ'.

4.1.3 Formal typability rules

The rules for typing expressions are fairly natural. Remember the interpretation of $t : \sigma$ as 't *could* be given type σ'.

$$\frac{v : \sigma \in \Gamma}{\Gamma \vdash v : \sigma}$$

$$\frac{\text{Constant } c \text{ has type } \sigma}{c : \sigma}$$

$$\frac{\Gamma \vdash s : \sigma \to \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash s\ t : \tau}$$

$$\frac{\Gamma \cup \{v : \sigma\} \vdash t : \tau}{\Gamma \vdash \lambda v.\, t : \sigma \to \tau}$$

Once again, this is to be regarded as an inductive definition of the typability relation, so a term only has a type if it can be derived by the above rules. For example, let us see how to type the identity function. By the rule for variables we have:

$$\{x : \sigma\} \vdash x : \sigma$$

and therefore by the last rule we get:

$$\emptyset \vdash \lambda x.\, x : \sigma \to \sigma$$

By our established convention for empty contexts, we write simply $\lambda x.\, x : \sigma \to \sigma$. This example also illustrates the need for the context in Curry typing, and why it can be avoided in the Church version. Without the context we could deduce $x : \tau$ for any τ, and then by the last rule get $\lambda x.\, x : \sigma \to \tau$, clearly at variance with the intuitive interpretation of the identity function. This problem doesn't arise in Church typing, since in that case either both variables have type σ, in which case the rules imply that $\lambda x.x : \sigma \to \sigma$, or else the two x's are actually *different variables*, since their types differ and the types are a component of the term. Now, indeed, $\lambda x : \sigma.\, (x : \tau) : \sigma \to \tau$, but this is reasonable because the term is alpha-equivalent to $\lambda x : \sigma. (y : \tau) : \sigma \to \tau$. In the Curry version of typing, the types are not attached to the terms, and so we need some way of connecting instances of the same variable.

4.1.4 Type preservation

Since the underlying terms of our typed lambda calculus are the same as in the untyped case, we can take over the calculus of conversions unchanged. However, for this to make sense, we need to check that all the conversions preserve typability, a property called *type preservation*. This is not difficult; we will sketch a formal proof for η-conversion and leave the other, similar cases to the reader. First we prove a couple of lemmas that are fairly obvious but which need to be established formally. First, adding new entries to the context cannot inhibit typability:

Lemma 4.1 *If $\Gamma \vdash t : \sigma$ and $\Gamma \subseteq \Delta$ then $\Delta \vdash t : \sigma$.*
Proof: *By induction on the structure of t. Formally we are fixing t and proving the above for all Γ and Δ, since in the inductive step for abstractions, the sets concerned change. If t is a variable we must have that $t : \sigma \in \Gamma$. Therefore $t : \sigma \in \Delta$ and the result follows. If t is a constant the result is immediate since the stock of constants and their possible types is independent of the context. If t is a combination $s\ u$ then we have, for some type τ, that $\Gamma \vdash s : \tau \to \sigma$ and $\Gamma \vdash u : \tau$. By the inductive hypothesis, $\Delta \vdash s : \tau \to \sigma$ and $\Delta \vdash u : \tau$ and the result follows. Finally, if t is an abstraction $\lambda x.\, s$ then we must have, by the last rule, that σ is of the form $\tau \to \tau'$ and that $\Gamma \cup \{x : \tau\} \vdash s : \tau'$. Since $\Gamma \subseteq \Delta$ we also have $\Gamma \cup \{x : \tau\} \subseteq \Delta \cup \{x : \tau\}$, and so by the inductive hypothesis $\Delta \cup \{x : \tau\} \vdash s : \tau'$. By applying the rule for abstractions, the result follows. Q.E.D.*

Secondly, entries in the context for variables not free in the relevant term may be discarded.

Lemma 4.2 *If $\Gamma \vdash t : \sigma$, then $\Gamma_t \vdash t : \sigma$ where Γ_t contains only entries for variables free in t, or more formally $\Gamma_t = \{x : \alpha \mid x : \alpha \in \Gamma \text{ and } x \in FV(t)\}$.*
Proof: *Again, we prove the above for all Γ and the corresponding Γ_t by structural induction over t. If t is a variable, then since $\Gamma \vdash t : \sigma$ we must have an entry $x : \sigma$ in the context. But by the first rule we know $\{x : \sigma\} \vdash x : \sigma$ as required. In the case of a constant, then the typing is independent of the context, so the result holds trivial ly. If t is a combination $s\ u$ then we have for some τ that $\Gamma \vdash s : \tau \to \sigma$ and $\Gamma \vdash u : \tau$. By the inductive hypothesis, we have $\Gamma_s \vdash s : \tau \to \sigma$ and $\Gamma_u \vdash u : \tau$. By the monotonicity lemma, we therefore have $\Gamma_{su} \vdash s : \tau \to \sigma$ and $\Gamma_{su} \vdash u : \tau$, since $FV(s\ u) = FV(s) \cup FV(u)$. Hence by the rule for combinations we get $\Gamma_{su} \vdash t : \sigma$. Finally, if t has the form $\lambda x.\, s$, we must have*

$\Gamma \cup \{x : \tau\} \vdash s : \tau'$ with σ of the form $\tau \to \tau'$. By the inductive hypothesis we have $(\Gamma \cup \{x : \tau\})_s \vdash s : \tau'$, and so $(\Gamma \cup \{x : \tau\})_s - \{x : \tau\} \vdash (\lambda x.\ s) : \sigma$. Now we need simply observe that $(\Gamma \cup \{x : \tau\})_s - \{x : \tau\} \subseteq \Gamma_t$ and appeal once more to monotonicity. Q.E.D.

Now we come to the main result.

Theorem 4.3 *If $\Gamma \vdash t : \sigma$ and $t \underset{\eta}{\longrightarrow} t'$, then also $\Gamma \vdash t' : \sigma$*

Proof: *Since by hypothesis t is an η-redex, it must be of the form $(\lambda x.t\ x)$ with $x \notin FV(t)$. Now, its type can only have arisen by the last typing rule, and therefore σ must be of the form $\tau \to \tau'$, and we must have $\{x : \tau\} \vdash (t\ x) : \tau'$. Now, this typing can only have arisen by the rule for combinations. Since the context can only give one typing to each variable, we must therefore have $\{x : \tau\} \vdash t : \tau \to \tau'$. But since by hypothesis, $x \notin FV(t)$, the lemma yields $\vdash t : \tau \to \tau'$ as required. Q.E.D.*

Putting together all these results for other conversions, we see that if $\Gamma \vdash t : \sigma$ and $t \longrightarrow t'$, then we have $\Gamma \vdash t' : \sigma$. This is reassuring, since if the evaluation rules that are applied during execution could change the types of expressions, it would cast doubt on the whole enterprise of static typing.

4.2 Polymorphism

The Curry typing system already gives us a form of *polymorphism*, in that a given term may have different types. Let us start by distinguish between the similar concepts of polymorphism and *overloading*. Both of them mean that an expression may have multiple types. However in the case of polymorphism, all the types bear a systematic relationship to each other, and all types following the pattern are allowed. For example, the identity function may have type $\sigma \to \sigma$, or $\tau \to \tau$, or $(\sigma \to \tau) \to (\sigma \to \tau)$, but all the instances have the same structure. By contrast, in overloading, a given function may have different types that bear no structural relation to each other, or only certain instances may be selected. For example, a function $+$ may be allowed to have type $int \to int \to int$ or type $float \to float \to float$, but not $bool \to bool \to bool$.[3] Yet a third related notion is subtyping, which is a more principled version of overloading, regarding certain types as embedded in others. This, however, turns out to be a lot more complex than it seems.[4]

[3]Strachey, who coined the term 'polymorphism', referred to what we call polymorphism and overloading as *parametric* and *ad hoc* polymorphism respectively.

[4]For example, if α is a subtype of α', should $\alpha \to \beta$ be a subtype or a supertype of $\alpha' \to \beta$? One can defend either interpretation ('covariant' or 'contravariant') in certain situations.

4.2.1 Let polymorphism

Unfortunately the type system we have so far places some unpleasant restrictions on polymorphism. For example, the following expression is perfectly acceptable:

$$\text{if } (\lambda x.\ x) \text{ true then } (\lambda x.\ x)\ 1 \text{ else } 0$$

Here is a proof according to the above rules. We assume that the constants can be typed in the empty environment, and we fold together two successive applications of the rule for `if`, remembering that 'if b then t_1 else t_2' is shorthand for 'COND b t_1 t_2'.

$$\dfrac{\dfrac{\dfrac{\{x : bool\} \vdash x : bool}{\vdash (\lambda x.\ x) : bool \to bool} \quad \dfrac{}{\vdash \text{true} : bool}}{\vdash (\lambda x.\ x) \text{ true} : bool} \quad \dfrac{\dfrac{\{x : int\} \vdash x : int}{\vdash (\lambda x.\ x) : int \to int} \quad \dfrac{}{\vdash 1 : int}}{\vdash (\lambda x.\ x)\ 1 : int} \quad \dfrac{}{\vdash 0 : int}}{\vdash \text{if } (\lambda x.\ x) \text{ true then } (\lambda x.\ x)\ 1 \text{ else } 0 : int}$$

The two instances of the identity function get types $bool \to bool$ and $int \to int$. But consider the following:

$$\text{let } I = \lambda x.\ x \text{ in if } I \text{ true then } I\ 1 \text{ else } 0$$

According to our definitions, this is just syntactic sugar for:

$$(\lambda I.\ \text{if } I \text{ true then } I\ 1 \text{ else } 0)\ (\lambda x.\ x)$$

As the reader may readily confirm, this cannot be typed according to our rules. There is just *one* instance of the identity function and it must be given a single type. This feature is unbearable, since in practical functional programming we rely heavily on using `let`. Unless we change the typing rules, many of the benefits of polymorphism are lost. Our solution is to make `let` primitive, rather than interpret it as syntactic sugar, and add the new typing rule:

$$\frac{\Gamma \vdash s : \sigma \quad \Gamma \vdash t[s/x] : \tau}{\Gamma \vdash \text{let } x = s \text{ in } t : \tau}$$

This rule, which sets up *let polymorphism*, expresses the fact that, with respect to typing at least, we treat `let` bindings just as if separate instances of the named expression were written out. The additional hypothesis $\Gamma \vdash s : \sigma$ merely ensures that s is typable, the exact type being irrelevant. This is to avoid admitting as well-typed terms such as:

$$\text{let } x = \lambda f.\ f\ f \text{ in } 0$$

Now, for example, we can type the troublesome example:

$$\dfrac{\dfrac{\overline{\{x:\sigma\} \vdash x:\sigma}}{\vdash \lambda x.\ x : \sigma \to \sigma} \qquad \dfrac{\textbf{As given above}}{\vdash \text{if } (\lambda x.\ x) \text{ true then } (\lambda x.\ x)\ 1 \text{ else } 0 : \mathit{int}}}{\vdash \text{let } I = \lambda x.\ x \text{ in if } I \text{ true then } I\ 1 \text{ else } 0 : \mathit{int}}$$

4.2.2 Most general types

As we have said, some expressions have no type, e.g. $\lambda f.\ f\ f$ or $\lambda f.\ (f \text{ true}, f\ 1)$. Typable expressions normally have many types, though some, e.g. *true*, have exactly one. We have already commented that the form of polymorphism in ML is *parametric*, i.e. all the different types an expression can have bear a structural similarity to each other. In fact more is true: there exists for each typable expression a *most general type* or *principal type*, and all possible types for the expression are instances of this most general type. Before stating this precisely we need to establish some terminology.

First, we extend our notion of types with *type variables*. That is, types can be built up by applying type constructors either to type constants or variables. Normally we use the letters α and β to represent type variables, and σ and τ arbitrary types. Given this extension, we can define what it means to substitute one type for a type variable in another type. This is precisely analogous to substitution at the term level, and we even use the same notation. For example $(\sigma \to \mathit{bool})[(\sigma \to \tau)/\sigma] = (\sigma \to \tau) \to \mathit{bool}$. The formal definition is much simpler than at the term level, because we do not have to worry about bindings. At the same time, it is convenient to extend it to multiple parallel substitutions:

$$\begin{array}{rcl}
\alpha_i[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_k] & = & \tau_i \text{ if } \alpha_i \neq \beta \text{ for } 1 \leq i \leq k \\
\beta[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_k] & = & \beta \text{ if } \alpha_i \neq \beta \text{ for } 1 \leq i \leq k \\
(\sigma_1, \ldots, \sigma_n)\mathit{con}[\theta] & = & (\sigma_1[\theta], \ldots, \sigma_n[\theta])\mathit{con}
\end{array}$$

For simplicity we treat type constants as nullary constructors, e.g. consider $()\mathit{int}$ rather than int, but if preferred it is easy to include them as a special case. Given this definition of substitution, we can define the notion of a type σ being *more general* than a type σ', written $\sigma \preceq \sigma'$.[5] This is defined to be true precisely if there is a set of substitutions θ such that $\sigma' = \sigma\theta$. For example we have:

$$\begin{array}{rcl}
\alpha & \preceq & \sigma \\
\alpha \to \alpha & \preceq & \beta \to \beta \\
\alpha \to \mathit{bool} & \preceq & (\beta \to \beta) \to \mathit{bool}
\end{array}$$

[5]This relation is reflexive, so we should really say ‘at least as general as’ rather than ‘more general than’.

$$\begin{array}{rcl} \beta \to \alpha & \preceq & \alpha \to \beta \\ \alpha \to \alpha & \not\preceq & (\beta \to \beta) \to \beta \end{array}$$

Now we come to the main theorem:

Theorem 4.4 *Every typable term has a principal type, i.e. if $t : \tau$, then there is some σ such that $t : \sigma$ and for any σ', if $t : \sigma'$ then $\sigma \preceq \sigma'$.*

It is easy to see that the relation $\preceq$ is a preorder relation, i.e. is reflexive and transitive. The principal type is not unique, but it is unique up to renaming of type variables. More precisely, if σ and τ are both principal types for an expression, then $\sigma \sim \tau$, meaning that $\sigma \preceq \tau$ and $\tau \preceq \sigma$.

We will not even sketch a proof of the principal type theorem; it is not particularly difficult, but is rather long. What is important to understand is that the proof gives a concrete procedure for finding a principal type. This procedure is known as *Milner's algorithm*, or often the Hindley-Milner algorithm.[6] All implementations of ML, and several other functional languages, incorporate a version of this algorithm, so expressions can automatically be allocated a principal type, or rejected as ill-typed.

4.3 Strong normalization

Recall our examples of terms with no normal form, such as:

$$\begin{array}{rl} & ((\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)) \\ \longrightarrow & ((\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)) \\ \longrightarrow & (\cdots) \end{array}$$

In typed lambda calculus, this cannot happen by virtue of the following *strong normalization* theorem, whose proof is too long for us to give here.

Theorem 4.5 *Every typable term has a normal form, and every possible reduction sequence starting from a typable term terminates.*[7]

At first sight this looks good — a functional program respecting our type discipline can be evaluated in any order, and it will always terminate in the unique normal form. (The uniqueness comes from the Church-Rosser theorem, which is

[6]The above principal type theorem in the form we have presented it is due to Milner (1978), but the theorem and an associated algorithm had earlier been discovered by Hindley in a similar system of combinatory logic.

[7]Weak normalization means the first part only, i.e. every term can be reduced to normal form, but there may still be reduction sequences that fail to terminate.

still true in the typed system.) However, the ability to write nonterminating functions is essential to Turing completeness,[8] so we are no longer able to define all computable functions, not even all total ones.

This might not matter if we could define all 'practically interesting' functions. However this is not so; the class of functions definable as typed lambda terms is fairly limited. Schwichtenberg (1976) has proved that if Church numerals are used, then the definable functions are those that are polynomials or result from polynomials using definition by cases. Note that this is strictly intensional: if another numeral representation is used, a different class of functions is obtained. In any case, it is not enough for a general programming language.

Since all definable functions are total, we are clearly unable to make arbitrary recursive definitions. Indeed, the usual fixpoint combinators must be untypable; $Y = \lambda f.\ (\lambda x.\ f(x\ x))(\lambda x.\ f(x\ x))$ clearly isn't well-typed because it applies x to itself and x is bound by a lambda. In order to regain Turing-completeness we simply add a way of defining arbitrary recursive functions that *is* well-typed. We introduce a polymorphic recursion operator with all types of the form:

$$Rec : ((\sigma \to \tau) \to (\sigma \to \tau)) \to \sigma \to \tau$$

and the extra reduction rule that for any $F : (\sigma \to \tau) \to (\sigma \to \tau)$ we have:

$$Rec\ F \longrightarrow F\ (Rec\ F)$$

From now on, we will suppose that recursive definitions, signalled by 'let rec', map down to the use of these recursion operators.

Further reading

Barendregt (1984) and Hindley and Seldin (1986) also consider typed lambda calculus. The original paper by Milner (1978) is still a good source of information about polymorphic typing and the algorithm for finding principal types. A nice elementary treatment of typed lambda calculus, which gives a proof of Strong Normalization and discusses some interesting connections with logic, is Girard, Lafont, and Taylor (1989). This also discusses a more advanced version of typed lambda calculus called 'System F' where even though strong normalization holds, the majority of 'interesting' functions are definable.

Exercises

1. Is it true that if $\Gamma \vdash t : \sigma$ then for any substitution θ we have $\Gamma \vdash t : (\sigma\theta)$?

[8] Given any recursive enumeration of total computable functions, we can always create one not in the list by diagonalization. This will be illustrated in the Computability Theory course.

2. Prove formally the type preservation theorems for α and β conversion.

3. Show that the converse to type preservation does not hold, i.e. it is possible that $t \longrightarrow t'$ and $\Gamma \vdash t' : \sigma$ but not $\Gamma \vdash t : \sigma$.

4. (*) Prove that every term of typed lambda calculus with *principal* type $(\alpha \to \alpha) \to (\alpha \to \alpha)$ reduces to a Church numeral.

5. (*) To what extent can the typechecking process be reversed, i.e. the term inferred from the type? For instance, is it true that in pure typed lambda calculus with type variables but no constants and no recursion operator, that any $t : \alpha \to \beta \to \alpha$ is in fact equal to $K = \lambda x\ y.\ x$ in the usual sense of lambda equality? If so, how far can this be generalized?[9]

6. (*) We will say that an arbitrary reduction relation $\longrightarrow$ is *weakly Church-Rosser* if whenever $t \longrightarrow t_1$ and $t \longrightarrow t_2$, then there is a u with $t_1 \longrightarrow^* u$ and $t_2 \longrightarrow^* u$, where $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$. Prove *Newman's Lemma*, which states that if a relation is weakly Church-Rosser and obeys strong normalization, then $\longrightarrow^*$ is Church-Rosser. (Hint: use a form of wellfounded induction.)[10]

[9]See Mairson (1991) for more on this question.

[10]If you get stuck, there is a nice proof in Huet (1980).