In common with several other modern programming languages, Miranda has the property that a programmer need not specify the types of the objects defined in his program. The compiler can work out those types, if the program can be consistently typed at all. The part of the compiler that does this is usually called the 'type-checker'. It attempts to infer the types of expressions in the program from their contexts. This kind of type-checking was first implemented for the language ML, around 1976. The type discipline was first expounded by Milner [1978].

Whether or not a type-checker requires information from the programmer to check that a program is well typed, type-checking is of great value in drawing the programmer's attention to a variety of errors, from trivial slips in program entry, to gross logical blunders. It helps us to write robust programs.

Another advantage of type-checking is that it helps to build faster implementations of programming languages. If a program is passed by the type-checker, then no type error should occur at run-time, such as the use of an integer as if it were a function, a boolean as if it were an integer, or a function as if it were a tuple. In Milner's words, well-typed expressions do not 'go wrong': at run-time we will never misinterpret the representation of an expression. By omitting run-time checks for such errors, the implementation of a language can be made simpler and faster. Of course, any implementation should still provide for diagnosis of its own internal errors.

The purpose of this chapter is to explain in some detail how a type-checker works. Then, in Chapter 9, we put the ideas into practice by constructing a type-checker for a simple functional language. The type-checker is constructed in Miranda, in the hope that the development of such a functional program may itself be of some additional interest.
Given the informal spirit of this book, and its concentration on setting up intuitions rather than on attaining impregnable conceptual rigor, it is not appropriate to proceed 'from the ground up'. Instead, we shall assume that the reader already has some understanding of the notion of a type, and wishes to see how that notion can be applied in practice. Nevertheless, some cautionary remarks may be in order, and they are made at the end of the chapter.

This chapter is organized as follows. Section 8.1 reviews some basic concepts, and notations for types. Section 8.2 illustrates the concept of polymorphism, using several examples. Section 8.3 shows in an informal way how types may be inferred from the structure of a definition. Section 8.4 sets out the language for which we will build a type-checker. Section 8.5 considers the detailed type structure of expressions in the language, and attempts to clarify the rules of type inference, which are summarized in Section 8.6. Section 8.7 contains the cautionary remarks referred to before.

*Important note:* The type-checker described here is actually somewhat more liberal than that of the Miranda compiler itself, in that it will succeed in type-checking some programs which the Miranda compiler would reject. This difference is explained in Section 8.5.5. The Miranda type-checker is also considerably more sophisticated than the one we describe here, because it supports features, such as abstract data types and a module structure, which are beyond the scope of this book.

### 8.1 Informal Notation for Types

The types with which we are concerned in functional programming include ground types such as characters, numbers and booleans, types of tuples, lists and, of course, functions. To talk about these types, we will use the following notation. Capital letters will be used for type variables. A type variable $A$ stands for a type in much the same way that a numerical variable $n$ stands for a number in mathematics. Lower-case letters will be used for the elements of types. The notation

$$a :: A$$

means that $a$ has type $A$. For example, $42 :: \text{num}$, 'f' :: \text{char}, where \text{num} is the type of numbers, and \text{char} is the type of characters. (Note: the notation used for types in this chapter differs from that of Miranda – in Miranda an upper case letter cannot stand for a type.)

#### 8.1.1 Tuples

Given types $A$ and $B$, $(A,B)$ is the type of ordered pairs $(a,b)$ where $a :: A$, and $b :: B$. Using Descartes' terminology, $a$ is the first *coordinate* of $(a,b)$, and $b$ is the second. More generally, if $n \geq 2$ and $A_1, \ldots, A_n$ are types, then
is the type whose values are of the form of tuples

\[(a_1, \ldots, a_n)\]

where \(a_1::A_1, \ldots, a_n::A_n\). The important points about tuples, so far as typing is concerned, are:

(i) the coordinates of a tuple need not be of the same type;
(ii) the type of a tuple determines the number of its coordinates (that is, its dimension), and their types.

### 8.1.2 Lists

Given a type \(B\), \([B]\) is the type of lists whose entries are of type \(B\). More specifically, an object of type \([B]\) must be

(i) either the empty list, which is denoted by \([]\);
(ii) or a non-empty list, formed by prefixing an object \(b::B\) to a list \(bs::[B]\), which is denoted by \(b::bs\).

If all the successive entries \(b_1, \ldots, b_k\) of a finite list are known, we may write it using the notation

\([b_1, \ldots, b_k]\)

The important points about lists, so far as typing is concerned, are:

(i) In contrast with the coordinates of a tuple, all entries of a list must be of the same type. For example, it would make no sense to form a list in which the entries were alternately characters and truth values. (We could in fact define a type of such entities, but they would not be lists.)
(ii) In contrast with the dimension of a tuple, the length of a list is not determined by its type. Indeed, when programming in a lazy language, we may operate with infinite lists such as the list of positive integers. There is no requirement that a list must be built up from the empty list by a finite number of applications of the prefixing operation \((b::bs)\), or that a principle of well-founded induction on the structure of lists should be valid.

### 8.1.3 Structured Types

Tuple types and list types are both examples of structured types, which were introduced in Chapter 4. As explained there, in Miranda the general form of a declaration of an operator for forming structured types is:

\[
\text{name } v_1 \ldots v_k ::= \quad c_1 \cdot t_{1,1} \ldots t_{1,r_1} \\
\quad | \quad \ldots \\
\quad | \quad c_m \cdot t_{m,1} \ldots t_{m,r_m}
\]

where \(m \geq 1, r_i \geq 0\) for \(1 \leq i \leq m\), and \(k \geq 0\). Here \(v_1, \ldots, v_k\) stand for schematic
type variables, which in Miranda have the special form *, **, *** etc. Also, \( t_1, \ldots, t_{m,r_m} \) are type expressions, built up using variables from the list \( v_1, \ldots, v_k \) and names for type-forming operations which are either built-in or declared elsewhere in the script.

For example, in the type declaration

\[
\text{tree } * ::= \text{LEAF } * \mid \text{BRANCH (tree *) (tree *)}
\]

\( v_1 \) is *, \( c_1 \) is LEAF, and \( t_{1,1} \) is *; \( c_2 \) is BRANCH, and \( t_{2,1}, t_{2,2} \) are both (tree *).

'tree' is a type-forming operator since, given a type as 'argument', it produces a type as its 'result'; for example, (tree char), (tree num), (tree (tree num)). In this sense, the built-in basic types (such as char, num, bool) are simply type-forming operators which take no arguments.

A declaration with the form above means that an object of a type

\[
\text{name } t'_1 \ldots t'_k
\]

must have one of the constructed forms

\[
c_1 x_1 \ldots x_{r_1}
\]

where \( x_i::t'_{i,j} \) for \( 1 \leq j \leq r_i \), and \( t'_{i,j} \) denotes the result of simultaneously substituting the type expressions \( t'_1, \ldots, t'_k \) for the type variables \( v_1, \ldots, v_k \) in the type expression \( t_{i,j} \).

For example, here is an object of type (tree char):

\[
\text{BRANCH (LEAF 'a') (LEAF 'b')}
\]

In this case, \( t'_1 \) is char; the form of the object is a BRANCH, and \( x_1 \) is (LEAF 'a'); \( x_2 \) is (LEAF 'b'); \( x_3 \) is tree char.

### 8.1.4 Functions

Given types \( A \) and \( B \), we use the notation:

\[
A \rightarrow B
\]

to denote the type of functions \( f \) applicable to objects \( a::A \) whose values \( f(a) \) are of type \( B \).

For example, \( \text{(char} \rightarrow \text{num}) \) is the type of integer-valued functions of characters. The function 'code' which maps a character to its ASCII code is of this type.

\( \text{(char} \rightarrow \text{bool}) \) is the type of boolean-valued functions of characters. For example, the function

\[
is\text{digit ch} = (\text{code '0' <= x}) \& (x <= \text{code '9'})
\]

where \( x = \text{code ch} \)

is a function of this type.

\( \text{([char] \rightarrow [num])} \) is the type of functions whose arguments are lists of characters, and whose values are lists of integers. The function which returns the list of ASCII codes corresponding to a character list is of this type.
Section 8.2 Polymorphism

(Note: in functional programming, we consider a function to belong to a type \((A \rightarrow B)\) even though it is not totally defined on the domain type \(A\). For example, the partial function which assigns to every even number its successor has type \((\text{num} \rightarrow \text{num})\).

The arrow in the function type notation \((A \rightarrow B)\) is considered to be a right-associative binary operator. So

\[ A \rightarrow B \rightarrow C \]

means the same as

\[ A \rightarrow (B \rightarrow C) \]

and

\[
(A \rightarrow B \rightarrow C) \\
\rightarrow (A \rightarrow B) \\
\rightarrow A \\
\rightarrow C
\]

means the same as

\[(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))\]

(We shall often lay out a large type expression over several lines, as above.)

The reason we choose \(\rightarrow\) to be right associiative can be seen by considering a (curried) function \(f\) of two arguments \(a::A\) and \(b::B\). Then we have:

\[
f :: A \rightarrow B \rightarrow C \\
(f \ a) :: B \rightarrow C \\
(f \ a \ b) :: C
\]

If \(\rightarrow\) were left associative, we would have to write

\[
f :: A \rightarrow (B \rightarrow C)
\]

which is less convenient, since it uses more brackets.

8.2 Polymorphism

Many of the functions we define in a functional program are to a greater or lesser degree indifferent to the types of their arguments. This can be illustrated with a few examples.

8.2.1 The Identity Function

The identity function \(id\), defined by

\[
id \ x = x
\]

works equally well on arguments of any type. For example, in

\[
id \ 3 = 3 \\
id \ 'a' = 'a'
\]
id (3,'a') = (3,'a')

the function id is used with the types

\[ \text{num} \rightarrow \text{num} \]
\[ \text{char} \rightarrow \text{char} \]
\[ (\text{num},\text{char}) \rightarrow (\text{num},\text{char}) \]

In this sense, id is indifferent to the type of its arguments. However, id always returns a result of the same type as its argument. We express this by saying that id is of type \( A \rightarrow A \), for all types A.

Sometimes we omit the 'for all types A' (the jargon for which is schematic generality; A is said to be a schematic (or generic) variable). When the schematic variables are not given explicitly, every type variable is here to be understood as a schematic variable.

To say that id is of type \( A \rightarrow A \) for all types A means that the name id can occur in a larger expression in any context suitable for a function whose type is of that form. When we indicate a form by means of a type expression, we should say which parts of the expression may vary, by indicating the schematic variables. To say that a type T is of the form

\[ \ldots A \ldots B \ldots A \ldots C \ldots \]

where A and B are the schematic variables, is to say that T may be obtained by substituting certain types TA and TB for the schematic variables. In other words, T is a substitution instance of the indicated type. The types

\[ \text{num} \rightarrow \text{num} \]
\[ \text{char} \rightarrow \text{char} \]
\[ (\text{num},\text{char}) \rightarrow (\text{num},\text{char}) \]

are all substitution instances of the form

\[ A \rightarrow A \]

where it is understood that A is the schematic variable.

For a final example, consider the expression:

\[ \text{id (code (id 'a'))} \]

The first occurrence of id must have type \( \text{num} \rightarrow \text{num} \), and the second must have type \( \text{char} \rightarrow \text{char} \). Since these are both substitution instances of the type of id, \( A \rightarrow A \), the expression is correctly typed.

Note: What we here call schematic type variables are called in Miranda generic type variables and written using the special symbols \*, **, etc. to distinguish them from ordinary (non-generic) names for types.

### 8.2.2 The length Function

The function which returns the length of a list may be defined by the equations

\[
\begin{align*}
\text{length} \ [\ ] & = 0 \\
\text{length} \ (x:x\hspace{0.17em}\text{ss}) & = (\text{length} \ x\hspace{0.17em}\text{ss}) + 1
\end{align*}
\]
The function length works equally well on any list, regardless of the type of its entries. For example, in the equations:

\[
\begin{align*}
\text{length } [7,1,4] &= 3 \\
\text{length } ['7','1','4','z'] &= 4 \\
\text{length } [(3,'a'),(26,'z')] &= 2 \\
\text{length } [\text{id},\text{id}] &= 2
\end{align*}
\]

the function is used with the types:

\[
\begin{align*}
[\text{num}] & \rightarrow \text{num} \\
[\text{char}] & \rightarrow \text{num} \\
[(\text{num,}\text{char})] & \rightarrow \text{num} \\
[(\text{A } \rightarrow \text{A})] & \rightarrow \text{num}
\end{align*}
\]

respectively. We express the type of length by

\[
\text{length : : } [\text{A}] \rightarrow \text{num}, \text{ for all types A}
\]

which conveys that

(i) length is a function;
(ii) its arguments are lists;
(iii) its values are numbers;
(iv) the type of the entries in the argument list does not matter.

### 8.2.3 The Composition Function

Let us represent the composition of two functions \( f \) and \( g \) with a right-associative infix dot, and define

\[
(f \cdot g) x = f \ (g \ x)
\]

(We shall write the composition function 'compose' when we do not want to indicate its arguments.) Composition is well defined so long as both its left- and right-hand arguments are functions, and the type of arguments of its left-hand argument is the same as the type of values of its right-hand argument. For example, the following make perfect sense:

(i) \( \text{decode} \cdot \text{succ} \cdot \text{code} \)

where succ denotes the successor of an integer. The expression denotes a function which returns 'b' from 'a', 'c' from 'b', and so on. The composition function is used here with the type:

\[
(\text{num } \rightarrow \text{char}) \rightarrow (\text{char } \rightarrow \text{num}) \rightarrow \text{char } \rightarrow \text{char}
\]

at its first occurrence, and with the type:

\[
(\text{num } \rightarrow \text{num}) \rightarrow (\text{char } \rightarrow \text{num}) \rightarrow \text{char } \rightarrow \text{num}
\]

at its second.
(ii) \texttt{code . id, and id . code}

where \texttt{id} is the identity function discussed above. In these expressions, the composition function is used with the types:

\[
\begin{align*}
(\text{char} \rightarrow \text{num}) & \rightarrow (\text{char} \rightarrow \text{char}) \rightarrow \text{char} \rightarrow \text{num} \\
(\text{num} \rightarrow \text{num}) & \rightarrow (\text{char} \rightarrow \text{num}) \rightarrow \text{char} \rightarrow \text{num}
\end{align*}
\]

respectively.

(iii) \texttt{isdigit . decode}

which is the predicate of an integer which is itself the ASCII code of a decimal digit. Here the composition function is used with type:

\[
(\text{char} \rightarrow \text{bool}) \rightarrow (\text{num} \rightarrow \text{char}) \rightarrow \text{num} \rightarrow \text{bool}
\]

We can express the constraint on the types of the arguments of \texttt{compose} by saying:

\[
\texttt{compose :: (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C}
\]

where \(A, B\) and \(C\) are the schematic variables.

### 8.2.4 The Function \texttt{foldr}

The function \texttt{foldr} may be defined by the equation

\[
\begin{align*}
\texttt{foldr f b \ [\ ]} & = b \\
\texttt{foldr f b \ (a:as)} & = f \ a \ (\texttt{foldr f b as})
\end{align*}
\]

Again, \texttt{foldr} is to a certain extent indifferent to the types of its arguments. For example, the following make perfect sense:

(i) \texttt{foldr plus 0 \ [7,1,4]}

where \texttt{plus} means binary addition. The function \texttt{foldr} is used here with the type:

\[
\begin{align*}
(\text{num} \rightarrow \text{num} \rightarrow \text{num}) & \rightarrow \text{num} \\
& \rightarrow [\text{num}] \\
& \rightarrow \text{num}
\end{align*}
\]

(ii) \texttt{foldr append \ [\ ] \ ["str1","str2","str3"]}

Here \texttt{append} is the function which concatenates two lists. The function \texttt{foldr} is being used here with type:

\[
\begin{align*}
(\text{string} \rightarrow \text{string} \rightarrow \text{string}) & \rightarrow \text{string} \\
& \rightarrow [\text{string}] \\
& \rightarrow \text{string}
\end{align*}
\]
(iii) \texttt{foldr \ cons \ [] \ [5,4,1,4,1]}

Here \texttt{cons \ x \ y = x:y}. In this expression, \texttt{foldr} is used with the type:
\[
\begin{align*}
\text{(num} & \rightarrow \text{[num]} \rightarrow \text{[num]} \\
 & \rightarrow \text{[num]} \rightarrow \text{[num]} \rightarrow \text{[num]} \\
\end{align*}
\]

In general, \texttt{foldr} may be used in any context which requires a type of the form:
\[
\begin{align*}
(A & \rightarrow B \rightarrow B) \\
& \rightarrow B \\
& \rightarrow [A] \\
& \rightarrow B \\
\end{align*}
\]

where A and B are the schematic variables.

8.2.5 What Polymorphism Means

Polymorphism is a style of type discipline which seems to have been first identified by Christopher Strachey [1967]. A programming language has a polymorphic type discipline if it permits us to define functions which work uniformly for arguments of different types. For example, in a polymorphic language, we can define a single function \texttt{length} of type:
\[
[A] \rightarrow \text{num}
\]

In contrast, a language with a monomorphic type discipline forces the programmer to define different functions to return the length of a list of integers, a list of floating point numbers, a list of binary numerical functions, and so on. Languages such as Pascal and Algol 68 are monomorphic.

Strachey distinguished between \textit{ad hoc polymorphism}, and \textit{parametric polymorphism}. A type discipline exhibits \textit{ad hoc} polymorphism if it permits the use of the same expression to denote distinct operations at distinct types, such as the use of the addition symbol to denote addition of integers, rationals, real numbers, ordinals, complex numbers, and so on. This characteristic of a language is often now described as the ability to \textit{overload} expressions. On the other hand, parametric polymorphism is just polymorphism as explained above.

The words \textit{polymorphic} and \textit{monomorphic} are also sometimes used to distinguish between objects whose types are described by expressions with schematic type variables, and those whose type expressions have none. For example, the empty list is polymorphic, the functions \texttt{id}, \texttt{compose}, \texttt{length} and \texttt{foldr} are polymorphic, while the function \texttt{decode} which returns from an integer the character with that ASCII code is monomorphic.

A polymorphic object may take on different types at different occurrences, where these different types are substitution instances of the schematic type of the function. For example, we do not need to have different versions of \texttt{foldr} for each pair of types that instantiate A and B in the type expression
\[
(A \rightarrow B \rightarrow B) \rightarrow B \rightarrow [A] \rightarrow B
\]
or to parameterize foldr with the type variables A and B. Precisely the same code is executed whatever the types A and B (at least in a naïve implementation of the compiler), and it would be artificial to duplicate that code, or name it differently for each pair of types.

The terminology is also sometimes (perhaps unfortunately) applied to types themselves. For example, it is said that foldr possesses a ‘polymorphic’ type, meaning that its type is expressed with schematic variables. (Going by etymology, ‘polymorphic’ should mean ‘of many forms’, and it is precisely in order to identify a single form that we use an expression with schematic variables.)

A polymorphic type discipline was first worked out for the language ML around 1976, and since then has been incorporated in a number of functional and imperative languages. In pragmatic terms at least, polymorphism represents a significant advance over the type disciplines of languages such as Pascal or Algol 68.

### 8.3 Type Inference

This type discipline is not only polymorphic; it has the property that the only places in a program where we have to mention types at all are in the type definitions themselves. The type-checker is able, as part of a single process,

(i) to determine whether the program is well typed; and
(ii) if the program is well typed, to determine the type of any expression in the program.

(Of course, to make a program easier to understand we should almost always accompany a definition with a specification of the type of the defined entity.)

Before delving into the details of type-checking, we should ask ourselves how we can informally deduce the types of functions given only their defining equations.

Consider the definition:

```plaintext
isdigit ch = (code '0' <= x) & (x <= code '9')
where x = code ch
```

From the right-hand side of the definition we can see that, if the function is well defined at all, its value must be a truth-value, since the outermost operator & (conjunction) produces truth-values. Moreover, the infix operator <= which supplies its values as arguments to & also produces truth-values. (So we can see that & is used consistently with its type.) The arguments to <= must both have the type num, and this is clearly the case for the actual arguments, namely (code '0') and (code '9'). It follows that x must be a number, and for this to hold, ch must have type char. So the right-hand side of the definition is
well typed, with type bool, provided that the argument ch has type char. Since the left-hand side of an equation must have the same type as the right-hand side, we deduce that:

\[
\text{isdigit} :: \text{char} \rightarrow \text{bool}
\]

Consider now the definition of length, repeated here:

\[
\text{length} \; [] = 0 \\
\text{length} \; (x:xs) = (\text{length} \; xs) + 1
\]

From the first equation, it is clear that the type of length is of the form

\[
[A] \rightarrow \text{num}
\]

We must also look at the second equation to see whether it constrains the type A any further. For example, if the second equation were something like

\[
\text{length} \; (x:xs) = (\text{length} \; xs) + 1, \quad x = \text{'a'}
\]

(using a conditional expression), we would have to conclude that the type A is not in fact completely general, but completely specific: it is the type char. But in the case of the function length, the second clause imposes no further constraint, so we can say that

\[
\text{length} :: [A] \rightarrow \text{num}, \text{for all types A}
\]

Consider now the function foldr, with definition

\[
\text{foldr} \; f \; x = g \; \text{where} \; g \; [] = x \\
g \; (a:as) = f \; a \; (g \; as)
\]

The local function g is evidently a function on lists, since it is defined by cases on the two constructors of list form. So suppose g has type \([A] \rightarrow \text{B}\). Both x and \((f \; a \; (g \; as))\) must be of type B. Since \((g \; as)\) has type B, \(f\) must have type \((A \rightarrow \text{B} \rightarrow \text{B})\). So, all in all,

\[
\text{foldr} :: (A \rightarrow \text{B} \rightarrow \text{B}) \rightarrow \text{B} \rightarrow [A] \rightarrow \text{B}
\]

In general, by examining the context of an expression, we may be able to deduce an expression for the form of the type of an object which can fit into that context. By examining the expression itself, we may be able to deduce the form of the types which that expression can take on. So we have two type expressions that will usually contain variables, the first giving the form of the type required by the context (deduced from the 'outside'), and the second giving the form of type which the object can take (deduced from the 'inside'). For the whole expression to be well typed, these two type expressions must match, in the sense that by substituting for the schematic variables of the type expressions, they can be brought to the same form.
8.4 The Intermediate Language

The language for which we will construct a type-checker is the language of the lambda calculus. We will use the form of that language in which recursion is expressed using the letrec construct rather than by using the Y combinator. Briefly, the forms of expression are these:

(i) Variables: x, y, etc.
(ii) Lambda abstractions: \( \lambda x. E \)
(iii) Application: \( E_1 E_2 \)
(iv) Simultaneous definitions (let-expressions):

\[
\begin{align*}
&\text{let } x_1 = E_1 \\
&\ldots \\
&x_k = E_k \\
&\text{in } E
\end{align*}
\]

(v) Mutual recursion (letrec-expressions):

\[
\begin{align*}
&\text{letrec } x_1 = E_1 \\
&\ldots \\
&x_k = E_k \\
&\text{in } E
\end{align*}
\]

The type-checker should be invoked when the source program has been brought into this form, and before lambda-lifting, or transformation to a supercombinator program (see Chapter 13). It is, however, important that the program is subjected to the dependency analysis referred to in Section 6.2.8 before type-checking. This is for the following reason. If we include in a letrec-expression a definition whose right-hand side does not 'really' depend on the other names defined in the letrec, we may not be able to type-check the program at all. (For an explanation of this, see Mycroft [1984].)

The most conspicuous absentee from this list of constructs is anything corresponding to function definitions by pattern-matching. But as is shown in Chapters 4–6, we can replace such definitions by using instead built-in case functions associated with the type-forming operations defined by the programmer or supplied by the system. The names of these case functions, and indeed of the associated discriminators and selectors, can be regarded as the names of variables with predeclared types. Hence they are of no special interest in the type-checker.

(In the same vein, we might have taken the easy way out in our treatment of recursion, and used the Y combinator, regarding this as having a priori the predeclared type

\[ Y :: (A \rightarrow A) \rightarrow A, \text{ for all types } A \]

However, the issues involved in the problem of how a type discipline should treat recursion are rather subtle. Although the solution we have adopted is in fact precisely equivalent to adoption of the Y combinator for the expression of
recursion, we take the point of view that to do this would be to sweep the problem under the carpet.)

The type-checking algorithm can still be developed when pattern-matching is present in the language. Indeed for practical reasons it is better to type-check while the program is still close to the form in which it was entered, in order that error messages can refer to program text that the programmer can recognize.

8.5 How to Find Types

Presumably, when we construct an expression \( E \) in a program, we reason to ourselves that it is well typed. As a product of this reasoning, we are in a position to say what the type is of any subexpression \( E' \) of \( E \). We can, as it were, label each subexpression with the type which we think it has. When we enter that expression into the text of our program, that 'labelling' has been lost. It is the job of the type-checker to reason out the type structure of the expression once again, and to recover the labelling.

If we accept that type-checking is a species of inference, this raises the question as to what forms of inference we may validly employ in checking the type of an expression. We shall not go so far as to try to state those forms of inference explicitly (akin to an exercise in formal logic), but rather by considering a sufficient variety of examples (as it were, particular syllogisms), try to work up some confidence that we can tell the difference between right and wrong inference.

8.5.1 Simple Cases, and Lambda Abstractions

In order to make enough space to expose the type structure of an expression, let us lay it out as a tree, where at the top we have the variables and constants, and as we proceed down towards the root, we pass through nodes labelled with the constructors applied in the formation of the expression. For an example containing both application and abstraction nodes, take the expression

\[(\lambda x. \lambda y. \lambda z. \ x \ z \ (y \ z))\]

Laid out as a tree this becomes

```
      @
    @   @
  x   y   z
 @
\lambda x. \lambda y. \lambda z.
```

Each node in this tree corresponds to a subexpression of the original expression, and should therefore possess a type. Assign arbitrary type labels
\( T_0, T_1, \ldots, T_7 \) to the nodes of the tree. Drawing the tree in a slightly different way to use less space, we get:

\[
\begin{array}{c}
x:: T_0 & y:: T_2 \\
\frac{z:: T_1}{T_4} & \frac{z:: T_3}{T_5} \\
\frac{\lambda x. \lambda y. \lambda z.}{T_6}
\end{array}
\]

In order to be sure than an expression \((E_1 E_2)\) of application form is well typed, the function \(E_1\) must have a functional type \((A \rightarrow B)\), where \(E_2\) is of type \(A\), and \((E_1 E_2)\) is of type \(B\). So whatever else is clear, the types of the sub-expressions must be related by the following equations:

\[
\begin{align*}
T_0 &= T_1 \rightarrow T_4 \\
T_2 &= T_3 \rightarrow T_5 \\
T_4 &= T_5 \rightarrow T_6
\end{align*}
\]

Substituting back in the tree, we get

\[
\begin{array}{c}
x:: T_1 \rightarrow T_5 \rightarrow T_6 & y:: T_3 \rightarrow T_5 \\
\frac{z:: T_1}{T_5 \rightarrow T_6} & \frac{z:: T_3}{T_5} \\
\frac{\lambda x. \lambda y. \lambda z.}{T_6}
\end{array}
\]

Now what should we say about the abstraction? Certainly \(T_7\) will have the form

\[(T_1 \rightarrow T_5 \rightarrow T_6) \rightarrow (T_3 \rightarrow T_5) \rightarrow \ldots\]

but it is not immediately clear what to do about the two type labels \(T_1\) and \(T_3\) for the two occurrences of the variable \(z\). It would be simple if we could see some reason to say that the labels \(T_1\) and \(T_3\) must stand for the same type. For then we could add two more equations to the set above, namely

\[
\begin{align*}
T_1 &= T_3 \\
T_7 &= (T_0 \rightarrow T_2 \rightarrow T_1 \rightarrow T_6)
\end{align*}
\]

and then on substituting back in the tree we would get

\[
\begin{array}{c}
x:: T_1 \rightarrow T_5 \rightarrow T_6 & y:: T_1 \rightarrow T_5 \\
\frac{z:: T_1}{T_5 \rightarrow T_6} & \frac{z:: T_1}{T_5} \\
\frac{\lambda x. \lambda y. \lambda z.}{T_6}
\end{array}
\]

\[(T_1 \rightarrow T_5 \rightarrow T_6) \rightarrow (T_1 \rightarrow T_5) \rightarrow T_1 \rightarrow T_6\]

On the other hand, we have already seen in Section 8.2.3 expressions such as

\[1 . \text{code} . 1\]
which make perfect sense, but in which the two occurrences of the composition function receive different types (to be sure, types sharing a common form, but nonetheless different).

So it is not obvious that we should require all occurrences of a variable bound by a lambda abstraction to have the same type. However, let us take this requirement as an assumption, and explore its consequences using the following example

\[ F = \lambda f . \lambda a . \lambda b . \lambda c . \ (f \ a) \ (f \ b) \]

and laid out as a tree, the expression is

\[
\begin{array}{c}
\frac{f :: T0 \ a :: T1}{c :: T2 \ T3}@\frac{f :: T4 \ b :: T5}{T6}@\frac{T7}{T8}@\frac{\lambda f . \lambda a . \lambda b . \lambda c .}{T9}
\end{array}
\]

from which we derive the equations

\[
\begin{align*}
T0 &= T1 \rightarrow T3 \\
T2 &= T3 \rightarrow T6 \\
T4 &= T5 \rightarrow T7 \\
T6 &= T7 \rightarrow T8
\end{align*}
\]

If we now require that the different occurrences of \( f \) have the same type, we can add the equation \( T0 = T4 \) to the list above. But then we must also have that \( T1 = T5 \) and \( T3 = T7 \), which gives the tree

\[
\begin{array}{c}
\frac{f :: T1 \rightarrow T3 \ a :: T1}{c :: T3 \rightarrow T3 \rightarrow T8}@\frac{f :: T1 \rightarrow T3 \ b :: T1}{T3}@\frac{T3 \rightarrow T8}{@}
\end{array}
\]

\[
\begin{array}{c}
\frac{T8}{@}
\end{array}
\]

\[
(T1 \rightarrow T3) \rightarrow T1 \rightarrow T1 \rightarrow (T3 \rightarrow T3 \rightarrow T8) \rightarrow T8
\]

By demanding that both occurrences of \( f \) should have the same type, we have forced \( a \) and \( b \) to be of the same type. Renaming variables, the function \( F \) has type

\[
(A \rightarrow B) \rightarrow A \rightarrow A \rightarrow (B \rightarrow B \rightarrow C) \rightarrow C
\]

according to our assumption.

It is not hard to think of contexts \( (F \ f \ a \ b) \) which would make sense when \( a \) and \( b \) are of different types. For example

\[ F \mid 0 \ 'a' \]
seems to be the function which when applied to a function \( c \) of type 
\( \text{num} \rightarrow \text{char} \rightarrow A \) returns the value \( (c \, 0 \, 'a') \). On the other hand,

\[
F \, \text{code} \, 0 \, 'a' \, K
\]

would certainly be an error, since it would result in the evaluation of \( (\text{code} \, 0) \), whereas the function \( \text{code} \) is applicable only to characters. At last we can see the point of the assumption. In order for an expression to be well typed, it is not enough that it cannot ‘go wrong’ when evaluated on its own, or in a particularly favorable context. We have to make sure that it cannot ‘go wrong’ when plugged into any well-typed context.

So we shall require that variables bound in a lambda abstraction receive the same type at all their occurrences. Without ‘outside knowledge’ of the arguments to which an abstraction will be applied, we must assume the worst: all occurrences of a variable bound by the same lambda abstraction must share the same type.

To sum up, so far we have adopted the following rules:

(i) The function part \( f \) of an application \( (f \, a) \) has a function type \( (A \rightarrow V) \), where \( A \) is the type of the argument part \( a \) and \( V \) is the type of the application \( (f \, a) \).

(ii) All occurrences of a \( \lambda \)-bound variable must have the same types.

Moreover, when solving a system of equations, we have used the following rule:

If \( (T_1 \rightarrow T_2) = (T_1' \rightarrow T_2') \), then \( T_1 = T_1' \) and \( T_2 = T_2' \).

(This follows from a more general law which states that if two compound type expressions are equal, then they must be formed with the same construction, and their corresponding parts must be equal.)

### 8.5.2 A Mistyping

Consider the expression

\[
\lambda n. \lambda a. \lambda b. \; b \; n \; (n \; a \; b)
\]

(This is sometimes used to define the successor function on natural numbers in the type-free lambda calculus.) Written as a tree, the expression is:

```
T8
  @
  @
T7
  @
  @
T6
  T4
  @
  @
T3
  T5
  @
  @
T1
  T0
  @
  @
T8
tl .la .lb.
```
From which we get the equations:

\[
\begin{align*}
T_0 &= T_1 \rightarrow T_4 \\
T_2 &= T_3 \rightarrow T_6 \\
T_4 &= T_5 \rightarrow T_7 \\
T_6 &= T_7 \rightarrow T_8 \\
T_9 &= T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_8 \\
T_3 &= T_0 \\
T_5 &= T_2
\end{align*}
\]

Eliminating \( T_4 \) and \( T_6 \), these become

\[
\begin{align*}
T_0 &= T_1 \rightarrow T_5 \rightarrow T_7 \\
T_2 &= T_3 \rightarrow T_7 \rightarrow T_8 \\
T_9 &= T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_8 \\
T_3 &= T_0 \\
T_5 &= T_2
\end{align*}
\]

Now note that these equations contain a circularity. If we try to use the last two equations to eliminate \( T_3 \) and \( T_5 \), we get

\[
\begin{align*}
T_0 &= T_1 \rightarrow T_2 \rightarrow T_7 \\
     &= T_1 \rightarrow (T_3 \rightarrow T_7 \rightarrow T_8) \rightarrow T_7 \\
     &= T_1 \rightarrow (T_0 \rightarrow T_7 \rightarrow T_8) \rightarrow T_7
\end{align*}
\]

(since \( T_5 = T_2 \))

(since \( T_3 = T_0 \))

So it is clear that the type \( T_0 \) is not finite, and so neither is the type \( T_9 \).

Nevertheless, \( T_9 \) possesses an infinite type, which may be expressed informally:

\[
T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_8
\]

where

\[
T_0 = T_1 \rightarrow (T_0 \rightarrow T_7 \rightarrow T_8) \rightarrow T_7
\]

There are many difficulties in dealing with infinite types. We shall simply avoid them by imposing the rule:

If \( T_1 = \ldots T_1 \ldots \), where the type variable \( T_1 \) occurs properly within the right-hand side of the equation, then the system of equations cannot be solved, and the expression from which the system was derived is ill-typed.

As a consequence of this, the definition in Section 2.4.2 of the fixed-point combinator \( Y \) is ill-typed.

### 8.5.3 Top-level lets

Consider the expression

\[
\begin{align*}
\text{let } S &= \lambda x. \lambda y. \lambda z. x \ z \ (y \ z) \\
K &= \lambda x. \lambda y. \ x \\
\text{in } S \ K \ K
\end{align*}
\]
It seems intuitively reasonable that we allow $K$ to take on different types at its different occurrences in the body of the \texttt{let}-expression. Indeed, it is hard to see what polymorphism would mean if we insisted that variables introduced by a \texttt{let} definition should have the same type, as with variables bound by $\lambda$.

To examine the type structure of this expression, we need to extend the tree notation to represent it:

\[
\begin{array}{c}
S : T_8 \rightarrow T_7 \rightarrow T_8 \quad K : T_6 \\
\text{Tree-S} \quad \text{Tree-K} \quad \text{T7 \rightarrow T8} \quad K : T_7 \\
S : T_S \\
K : T_K \\
T_8 \rightarrow \text{let } S.K.
\end{array}
\]

Since we already know how to type-check the right-hand sides of the definitions of $S$ and $K$, we have merely indicated their type trees, to save space. Moreover, we have skipped a few steps in representing the type structure of $(S \ K \ K)$. The equations for the type structure of the right-hand sides of the definitions of $S$ and $K$ can be solved to yield:

\[
T_S = (T_0 \rightarrow T_1 \rightarrow T_2) \rightarrow (T_0 \rightarrow T_1) \rightarrow T_0 \rightarrow T_2 \\
T_K = T_3 \rightarrow T_4 \rightarrow T_3
\]

The new constraints we have to consider are those relating $T_8$ to $T_9$, and the types $T_S$ and $T_K$ to the types of their occurrences in the body of the \texttt{let}-expression.

For the first constraint, plainly we should require that $T_8 = T_9$. As for the second, the constraint is that the type of the occurrence of $S$ should be an instance of the type $T_S$, and the types of the two occurrences of $K$ should each be an instance of the type $T_K$. But how should we represent such a requirement by means of an equation?

When working out the equations by hand, it is quite natural to proceed as follows: \textit{refrain} from making any such representation at the outset. Instead, obtain first a fully evaluated expression for the type of $T_S$ and $T_K$ (as we have done). Then introduce new type labels for the instantiated variables at each occurrence of $S$ and $K$ in the body of the \texttt{let}-expression. (In this case, there are three such variables in the type for $S$, namely $T_0$, $T_1$ and $T_2$; and two in the type of $K$, namely $T_3$ and $T_4$.) If we use a fresh set of variables for each occurrence, then we can still work with equations, and leave the values of those fresh variables to be worked out while we are exploring the type structure of the body. So in this case we should add new variables $T_{10}$, $T_{11}$, $T_{12}$ to instantiate $T_S$ at its first occurrence, $T_{13}$ and $T_{14}$ to instantiate $T_K$ at the first occurrence of $K$, and $T_{15}$ and $T_{16}$ to instantiate $T_K$ at the second occurrence of $K$. We then add the equations

\[
\begin{align*}
T_6 \rightarrow T_7 \rightarrow T_8 &= (T_{10} \rightarrow T_{11} \rightarrow T_{12}) \\
&\quad \rightarrow (T_{10} \rightarrow T_{11}) \rightarrow T_{10} \rightarrow T_{12} \\
T_6 &= T_{13} \rightarrow T_{14} \rightarrow T_{13} \\
T_7 &= T_{15} \rightarrow T_{16} \rightarrow T_{15}
\end{align*}
\]
From the first of these we derive:

\[ T_6 = T_{10} \rightarrow T_{11} \rightarrow T_{12} \]
\[ T_7 = T_{10} \rightarrow T_{11} \]
\[ T_8 = T_{10} \rightarrow T_{12} \]

reasoning that if \((T_1 \rightarrow T_2) = (T'_1 \rightarrow T'_2)\), then \(T_1 = T'_1\) and \(T_2 = T'_2\).

By the same reasoning, we have

\[ T_{10} = T_{13} = T_{12} \]
\[ T_{11} = T_{14} \]
\[ T_{10} = T_{15} \]
\[ T_{11} = T_{16} \rightarrow T_{15} \]

which allows us to express the types of the two occurrences of \(K\) as

\[ T_6 = T_{10} \rightarrow (T_{16} \rightarrow T_{10}) \rightarrow T_{10} \]
\[ T_7 = T_{10} \rightarrow T_{16} \rightarrow T_{10} \]

and the type of the whole expression as

\[ T_9 = T_8 = T_{10} \rightarrow T_{10} \]

So the rule we adopt as the type-constraint for let-expressions is that the types of the occurrences of the defined names in the body must be instances of the types of the corresponding right-hand sides. The procedure we adopt to compute those instances is to instantiate the variables in the types of those right-hand sides with new variables, making a fresh instance for each occurrence of the defined name in the body of the let. In fact, we shall not in general be able to instantiate all the type variables, as we shall see shortly.

### 8.5.4 Top-level letrecs

Turning now to letrecs, it seems clear that a variable introduced by a letrec definition should be capable of taking on different types in the body of the program governed by the letrec, just as in the case of let-definitions. So in

```
letrec f = (...) in (...f...f...f...)...
```

we expect \(f\) to be capable of taking on different types throughout the expression body. However, there is a new question we must answer. The variable introduced by a recursive definition can also have many occurrences in the right-hand side of its definition, as it were ‘while’ it is being defined, as well as ‘after’. In general, when there are several mutually recursive definitions, as in

```
letrec x_1 = (...x_1...x_i...x_k...) 
    ...
    x_k = (...x_1...x_i...x_k...) 
    in (...x_1...x_i...x_j...x_k...) 
```

...
any one of the defined names \(x_i\) can occur many times in many right-hand sides, as well as in the body. Should we insist that all these occurrences have the same type, in the sense of requiring equality to hold between the type labels for the variable occurrences in the definitions? Or should we treat them as we treat them in the body, and require only that at each such occurrence, the type be an instance of the type of the corresponding right-hand side? Unfortunately, in the nature of things, there is no obvious answer. Nevertheless, to see what the question means, consider the example

\[
\text{letrec } Y = (\lambda f. f (Y f)) \text{ in } ...
\]

Written out as a tree, the first definition is

\[
\begin{array}{c}
Y :: T_0 \\
I :: T_1 \\
\hline
f :: T_2 \quad T_3 \\
\hline
T_4 \\
\hline
\lambda f.
\end{array}
\]

The constraints we can write down straight away are these:

\[
\begin{align*}
T_1 &= T_2 \\
T_0 &= T_1 \rightarrow T_3 \\
T_2 &= T_3 \rightarrow T_4 \\
T_5 &= T_1 \rightarrow T_4
\end{align*}
\]

from which it follows that

\[
T_0 = (T_3 \rightarrow T_4) \rightarrow T_3
\]

and

\[
T_5 = (T_3 \rightarrow T_4) \rightarrow T_4
\]

The question is, should we ask that \(T_0 = T_5\), or only that \(T_0\) be an instance of \(T_5\)? In the former case, the only solution is \(T_5 = ((T_4 \rightarrow T_4) \rightarrow T_4)\), as we would expect of a fixed-point function. On the other hand, the alternative requires only that \(T_3\) be an instance of \(T_4\), so again \(T_5 = ((T_4 \rightarrow T_4) \rightarrow T_4)\) is a solution.

We shall adopt the (usual) approach according to which ‘during’ such definitions all occurrences of the defined variables must share the same type as the right-hand side of their definitions. On the other hand, ‘after’ the definitions, the defined variables are polymorphic, and the type of such a variable can be instantiated differently to satisfy the local constraints on different occurrences of the variables in the body of the definition. If nothing else, this approach has at least the merit of simplicity.

Some different approaches to the type-checking of recursive definitions have been explored by Mycroft [1984]. In some (but not all) of these approaches the problem of whether an expression is well typed becomes only semi-decidable.
8.5.5 Local Definitions

We have presented type-checking as the search for the solution of a system of constraints, represented by equations $T' = T$ between type expressions. So far, we know that when type-checking an expression of let or letrec form, we should impose the constraint that the types of the occurrences of the defined variables in the body should equal new instances of the types derived for their right-hand sides. But just which type variables may be instantiated?

To understand this issue, we have to probe a little into the reason for our conviction that a defined name can take on different types in the body of its definition. The reason seems to be this:

An expression $(\text{let } x = E \text{ in } E')$ is well typed just in case the expression $E'[E/x]$ is well typed, which is the expression obtained by substituting $E$ for the free occurrences of $x$ in $E'$.

For each occurrence of $x$ in $E'$, we should be able to instantiate the type variables in the type tree for $E$ in such a way that it forms a subtree of the type tree for $E'[E/x]$. This instantiation is only possible if we do not thereby violate the law that occurrences of a $\lambda$-bound variable must have the same type, or the corresponding law for $\text{letrecs}$.

Consider the expression $(\lambda x. \text{let } y = x \text{ in } y y)$. By the principle above, this is well typed just in case $(\lambda x. x x)$ is well typed, which it blatantly is not. The problem is that the type expression for $y$ contains $(\text{is!})$ a variable occurring in the type of a more global $\lambda$-bound variable. We cannot instantiate that variable differently at the different occurrences of $y$ in $(y y)$.

Consider the partial expression

$\lambda x.$

let $I = \lambda z. z$

prxl = $\lambda c. (c x I)$

p1 = $\lambda x. \lambda y. x$

p2 = $\lambda x. \lambda y. y$

in ... 

Informally, the types of the defined names are

$\begin{align*}
I & :: A \rightarrow A \\
prxl & :: (X \rightarrow (A \rightarrow A) \rightarrow B) \rightarrow B \\
p1 & :: A \rightarrow B \rightarrow A \\
p2 & :: A \rightarrow B \rightarrow B
\end{align*}$

where $A$ and $B$ are schematic variables, and $X$ is the type of $x$. If we take the body of the let-expression to be the expression

$\text{prxl } p1 (\text{prxl } p1)$

then it cannot be typed. For to satisfy the type constraints of this body, we would have to instantiate $X$ differently at the different occurrences of prxl. On the other hand, if the body were

$\text{prxl } p2 (\text{prxl } p2)$
then the expression is well typed. For the structure of that expression does not constrain \( X \) to be instantiated differently at the different occurrences of \( \text{prx} \).

When we are type-checking the body \( B \) of a \( \text{let} \) or \( \text{letrec} \) definition, we must therefore distinguish the type variables in the type derived for a defined name according to whether they may or may not be differently instantiated at the various occurrences of the name. Variables of the former kind are those that do not occur in the type of any \( \text{constrained} \) variable in the definition of the name. A constrained variable is one which is a bound variable of a lambda abstraction enclosing \( B \), or one defined in a \( \text{letrec} \)-expression enclosing \( B \) in one of its right-hand sides.

This is one of the points at which the type regime of Miranda differs from that of the type checker described here. The Miranda compiler requires that all occurrences of a variable bound in a local definition share a single type. This has the effect that local definitions cannot introduce new polymorphism into a program. We will not explore the implications of this difference here—the type checking rules given in this and the following chapter are for a standard implementation of the Milner type discipline.

We have used the notion of type trees to help elucidate the type structure of expressions, and guide us towards a sharper view of the rules we use when constructing and checking the types of expressions. In the next section we summarize those rules. With luck, the device will have served its purpose, and we can then consider how to turn our intuitions into algorithms.

### 8.6 Summary of Rules for Correct Typing

The following rules are intended to describe the local ‘look’ of the type structure of a well-typed expression. To lighten the notational burden, we shall sometimes simplify the expression whose type tree is depicted in the figures. The simplifications are indicated in the commentary.

#### 8.6.1 Rule for Applications

\[
\frac{A \rightarrow B \quad A}{B} 
\]

#### 8.6.2 Rule for Lambda Abstractions

\[
\frac{\ldots \text{x :: A} \quad \ldots \text{x :: A} \quad \ldots \quad B}{\lambda \text{x}. \quad A \rightarrow B}
\]

Note that all occurrences of the variable \( x \) bound by the abstraction must have the same type.
8.6.3 Rule for let-expressions

\[
\begin{align*}
\{ \ldots y :: C \ldots \} \\
\ldots \ldots \ldots \ldots x :: A' \ldots x :: A'' \ldots \\
\ldots \ldots \ldots \ldots B \\
x :: A \\
\hline
\text{let } x. \\
\hline
\{ \}
\end{align*}
\]

Here we have shown only the case where just one definition is made in the let-expression: \( \text{let } x = E \) in \( E' \).

\textit{Restriction:} \( A' \) and \( A'' \) are instances of \( A \). No variable may be instantiated which occurs in the type of a variable bound in a more global lambda abstraction or letrec-expression (i.e. one further down the tree). The portions of the figure in curly brackets indicate such a situation. Any type variables in \( A \) shared with \( C \) may not be instantiated in forming \( A' \) and \( A'' \).

8.6.4 Rule for letrec-expressions

\[
\begin{align*}
\ldots \ldots x :: A \ldots x :: A' \ldots x :: A'' \ldots \\
\ldots \ldots x :: A \ldots x :: A' \ldots x :: A'' \ldots \\
\ldots \ldots \ldots \ldots B \\
x :: A \\
\hline
\text{letrec } x. \\
\hline
\end{align*}
\]

Here we have shown only the case where just one definition is made in the letrec-expression: \( \text{letrec } x = E \) in \( E' \). Note that the occurrences of \( x \) within the right-hand side of the definition must have the same type.

\textit{Restriction:} just as in the let rule.

8.7 Some Cautionary Remarks

There is a beguiling similarity between the notion of type which we use in mathematics, and the notion which we use in functional programming. It is all too easy to transfer intuitions concerning the mathematical notion of type to the notion used in programming. There are at least two important differences.

First, the types in a functional language are types of partial objects, whose evaluation may not terminate. In contrast, the mathematical notion of type, whose study began with Frege [Gaeck and Black, 1970] and Whitehead and Russell [1910–1913], concerns total objects, whose definitions are well
founded. The purpose of the mathematical notion of type is to elucidate the foundations of mathematics. The purpose of the notion in functional programming is to assure us at compile-time that a program will not 'go wrong', where we do not count a program to have gone wrong if it does not terminate, or a function is applied to arguments for which it has not been defined.

Second, in functional programming 'recursion' is interpreted in a very liberal sense, going far beyond recursion on well-founded structures, or positive inductive definitions. As a direct result of this, the notion of a type in functional programming cannot be the same notion that we use in mathematics. For example, in a functional program we can define an integer omega, where

\[ \text{omega} = \text{omega} + 1 \]

and this cannot belong to the (mathematical) type of integers. Another symptom of this liberal attitude to recursion is exhibited by the definition of the algebraic type

\[ D ::= \text{LAMBDA} (D \rightarrow D) \]

in which the defined type occurs negatively (to the left of the arrow) on the right-hand side of the definition. This is not to say that there is no mathematical sense in the functional programming notions. On the contrary, there is a rich and sophisticated theory (domain theory) which aims to give a mathematical interpretation to just such constructs. But while constructing that theory, and reasoning about the mathematical structures it involves, we are using on the metalevel the ordinary mathematical notion of type.

We hope that this chapter has shown that a naive understanding of the notion of type certainly gives us plenty to go on. We also hope to have achieved another goal: that of showing that there are limits to the questions we can settle on a naive basis alone.

References


In this chapter we will construct a type-checker in Miranda, taking the rules developed in the previous chapter as the basis for the type discipline.

Sections 9.1 and 9.2 show how the expressions of the intermediate language and its type expressions can be represented as Miranda data types. Sections 9.3 to 9.6 are concerned with the basic mechanisms of the type-checker, which is itself defined in Section 9.7.

9.1 Representation of Programs

Since we propose to write a type-checker in Miranda, we will have to represent the program to be type-checked as a Miranda data structure, which is passed as an argument to the type-checking function.

The program to be checked will be represented by an object of the structured type vexp, defined below. Each line of the type definition is derived directly from the corresponding construct in the concrete syntax.

\[
\begin{align*}
\text{vname} & \equiv [\text{char}] \\
\text{vexp} & \equiv \text{VAR vname} \\
& \quad \text{LAMBDA vname vexp} \\
& \quad \text{AP vexp vexp} \\
& \quad \text{LET [vname] [vexp] vexp} \\
& \quad \text{LETREC [vname] [vexp] vexp}
\end{align*}
\]

In a sense, this type encompasses slightly too much. We shall suppose that the program is not 'trivially' malformed: in a LET or LETREC construct, the list of variables must have the same length as the list of right-hand sides; the variable list in a LET or LETREC construct must not be empty, and should
contain no repetitions. Moreover, the free variables in an expression must be among those associated with predeclared types, either because they are supplied by the system, or because their types can be deduced from type definitions in the program. We can assure ourselves that these restrictions are met in a simple recursive scan through the program.

To understand the representation, let us take for an example the following trivial program:

```plaintext
let S = λx.λy.λz. x z (y z)
K = λx.λy. x
in S K K
```

Considered as an object in the type `vexp`, the program becomes:

```plaintext
LET ["S","K"] [rhs_S, rhs_K] main
where
  var_S = VAR "S"
  var_K = VAR "K"
  var_x = VAR "x"
  var_y = VAR "y"
  var_z = VAR "z"
main = AP (AP var_S var_K) var_K
rhs_S = plambda ["x","y","z"] body_S
rhs_K = plambda ["x","y"] body_K
body_S = AP (AP var_x var_z) (AP var_y var_z)
body_K = var_x
plambda vs e = foldr LAMBDA e vs
```

which the reader may write out without using 'where' if so inclined.

### 9.2 Representation of Type Expressions

To construct the type-checker, we will need to represent type expressions by Miranda data structures. We need a type for the names of type variables and, for the moment, we will take this to be the type of lists of characters. (For technical convenience, we will revise this definition in Section 9.6.)

```plaintext
tcname = [char]
type_exp ::= TVAR tcname
            | TCONS [char] [type_exp]
```

This definition says that a type expression must be either a type variable or a compound type (such as (A -> B), [A] or (A,B)). We represent such compound types by the name of the operator (e.g. "arrow" for (A -> B), "cross" for (A,B)), and a list of the operands.

Whatever other type-forming operators we have, we will certainly need the function type operator. So let us define:

```plaintext
arrow :: type_exp -> type_exp -> type_exp
arrow t1 t2 = TCONS "arrow" [t1,t2]
```
If \( t_1 \) and \( t_2 \) are of type \( \text{type\_exp} \), and we know what types they represent, then (arrow \( t_1 \ t_2 \)) will represent the type of functions from \( t_1 \) to \( t_2 \). Using Miranda's dollar notation for infixes, we may write this in the form (\( t_1 \ \$\text{arrow} \ t_2 \)), which adheres more closely to the informal notation.

The other type-forming operations we have mentioned could be represented in a similar way:

\[
\begin{align*}
\text{int} & :: \text{type\_exp} \\
\text{int} & \quad = \text{TCONS} \quad \text{"int"} \quad [] \\
\text{cross} & :: \text{type\_exp} \rightarrow \text{type\_exp} \rightarrow \text{type\_exp} \\
\text{cross} \ t_1 \ t_2 & \quad = \text{TCONS} \quad \text{"cross"} \quad [t_1,t_2] \\
\text{list} & :: \text{type\_exp} \rightarrow \text{type\_exp} \\
\text{list} \ t & \quad = \text{TCONS} \quad \text{"list"} \quad [t]
\end{align*}
\]

The function \text{tvars\_in} returns a list of the variable names that occur in a type expression. (The list may contain repetitions.)

\[
\begin{align*}
\text{tvars\_in} & :: \text{type\_exp} \rightarrow \text{[tname]} \\
\text{tvars\_in} \ t & \quad = \text{tvars\_in'} \ t \quad [] \\
\text{where} \\
\quad \text{tvars\_in'} \ (\text{TVAR} \ x) \ t & \quad = x:1 \\
\quad \text{tvars\_in'} \ (\text{TCONS} \ y \ ts) \ t & \quad = \text{foldr} \ \text{tvars\_in'} \ 1 \ ts
\end{align*}
\]

### 9.3 Success and Failure

Since type-checking is something that can succeed or fail, we have to choose a mechanism for representing success and failure within Miranda.

We shall use the type (reply *) for the type of the values of a function which may succeed (returning an object of type *) or fail (returning no indication as to why).

\[
\text{reply} * \quad ::= \text{OK} * \mid \text{FAILURE}
\]

It would not be acceptable for a practical type-checker to return no indication as to why a check has failed. One might then use a slightly more complicated operator, such as

\[
\text{reply'} * \quad ::= \text{OK'} * \mid \text{FAILURE'} *
\]

which is capable of returning error information. It is notoriously difficult to write error-handling code without obscuring the code to handle correct cases, so we will use instead the simpler, less informative operator. Any error detected while type-checking will be propagated up to the top level without further examination of the program. Here, too, there may be grounds for complaint, which we counter with the same excuse.

(There is more than one way to represent success and failure. An alternative approach to the one taken here is described by Wadler [1985].)
9.4 Solving Equations

Consider type-checking an application (AP e1 e2), where we have worked out the type t1 for e1 and the type t2 for e2. To do this, we try to 'solve the equation'

\[ t1 = t2 \rightarrow (\text{TVAR } n) \]

where \( n \) is a type variable name that has not been used before. As we have seen, the structure of an expression gives rise to a system of such equations.

How should we represent solutions of systems of type equations? In mathematics, the solution of simultaneous equations

\[
\begin{align*}
    a_{1,1} \times x_1 + a_{1,2} \times x_2 &= b_1 \\
    a_{2,1} \times x_1 + a_{2,2} \times x_2 &= b_2
\end{align*}
\]

is expressed by giving values for each of the unknowns \( x_1 \) and \( x_2 \) which satisfy the equations. Analogously, an alleged solution of a system of type equations can be expressed as a function from type variables (the unknowns) to type expressions (their values). The allegation is that the equations are satisfied when we replace (i.e. substitute) the unknowns by their values under the function. We therefore take

\[
\text{subst} \equiv \text{tvname} \rightarrow \text{type_exp}
\]

to be the type of substitutions. We shall see how to determine whether a set of equations between type expressions has a solution, and if so how to construct a substitution that satisfies them. We shall use identifiers such as phi, phi', psi, as variables over substitutions.

9.4.1 Substitutions

Given a substitution function phi and a type expression te, we define (sub_type phi te) to be the type expression obtained by performing the phi substitution on all the type variables in te:

\[
\begin{align*}
    \text{sub_type} &:: \text{subst} \rightarrow \text{type_exp} \rightarrow \text{type_exp} \\
    \text{sub_type phi (TVAR tvn)} &= \text{phi tvn} \\
    \text{sub_type phi (TCONS tcn ts)} &= \text{TCONS tcn (map (sub_type phi) ts)}
\end{align*}
\]

Here map is the function that applies a function to each entry in a list:

\[
\begin{align*}
    \text{map} &:: (\ast \rightarrow \ast\ast) \rightarrow [\ast] \rightarrow [\ast\ast] \\
    \text{map } f \ [] &= [] \\
    \text{map } f \ (x\!:xs) &= f \ x : \text{map } f \ xs
\end{align*}
\]

Two substitutions can be composed to give a further substitution:

\[
\begin{align*}
    \text{scomp} &:: \text{subst} \rightarrow \text{subst} \rightarrow \text{subst} \\
    \text{scomp sub2 sub1 tvn} &= \text{sub_type sub2 (sub1 tvn)}
\end{align*}
\]
The crucial property of scomp is that

\[
\text{sub\_type } (\text{scomp } \phi \ psi) = (\text{sub\_type } \phi) \cdot (\text{sub\_type } \psi)
\]

(Remember that function composition is represented by an infix dot.)

The identity substitution \text{id\_subst} has the property that

\[
\text{sub\_type } \text{id\_subst } t = t
\]

for all \(t::\text{type\_exp}\). It can be defined by:

\[
\text{id\_subst } :: \text{subst} \\
\text{id\_subst } \text{tn} = \text{TVAR } \text{tn}
\]

A \textit{delta substitution} is one that affects one variable only. We define:

\[
\text{delta } :: \text{tvname } -> \text{type\_exp } -> \text{subst} \\
\text{delta } \text{tn} \ t \ \text{tn}' \ = \ t, \quad \text{tn} = \text{tn}' \\
\quad \quad \quad \quad \quad = \text{TVAR } \text{tn}'
\]

Hence, \((\text{sub\_type } (\text{delta } \text{tn} \ t))\) is the function that maps a type expression to one that contains \(t\) where before it had \((\text{TVAR } \text{tn})\).

In fact, all the substitutions we need will be built up from the identity substitution \text{id\_subst} by composition on the left with substitutions of delta form.

In general, a substitution may associate a variable with a value which itself contains variables. If those variables in turn are given values different from themselves, then the substitution is not ‘fully worked out’. When we work out a set of equations

\[x_1 = t_1; \ldots; x_k = t_k\]

by substituting \(t_i\) for \(x_i\) at all of its occurrences in \(t_1, \ldots, t_k\), we may have to iterate the substitution many times before the equations stabilize to their final forms. (Of course, this iterative process does not terminate if there is a circularity in the equations.) In general, we are interested in obtaining ‘fully worked out’ substitutions, which do not have to be re-applied. The next definition is intended to capture what we mean by such a substitution.

A substitution \(\phi\) is \textit{idempotent} if

\[
(\text{sub\_type } \phi) \cdot (\text{sub\_type } \phi) = \text{sub\_type } \phi
\]

or equivalently, \((\phi \ \text{scomp } \phi) = \phi\). In other words, if you apply the substitution twice, you get nothing different the second time. A type expression \(t\) is a \textit{fixed point} of a substitution \(\phi\) if

\[
\text{sub\_type } \phi \ t = t
\]

In particular, if \((\text{TVAR } x)\) is a fixed point of \(\phi\), then we say that \(x\) is \textit{unmoved} by \(\phi\).
Note that if \( \phi \) is idempotent, and \( \phi \) moves \( t_0 \), then

\[
\text{sub_type} \ phi (\text{VAR} \ t_0)
\]

is a fixed point of \( \phi \), and hence cannot contain \( t_0 \).

### 9.4.2 Unification

In this section we will show how to construct a substitution which solves a given set of type equations, using a process called *unification*.

A system of type equations can be represented by a list of pairs of type expressions, where each pair \((t_1, t_2)\) represents the equation

\[
t_1 = t_2
\]

To solve the equations, we have to find a substitution \( \phi \) which *unifies* the left- and right-hand sides of all equations in the system, where \( \phi \) unifies the pair \((t_1, t_2)\) if

\[
\text{sub_type} \ \phi \ t_1 = \text{sub_type} \ \phi \ t_2
\]

If this equation holds, \( \phi \) is said to be a *unifier* of \( t_1 \) and \( t_2 \). If \( \phi \) is a unifier of each pair in the list representing a set of equations, we may think of \( \phi \) as a simultaneous solution of the equations.

If the substitution \( \phi \) solves a system of equations, then clearly any substitution \( \psi \) of the form \( \psi = \text{scomp} \ \phi \) is also a solution, but \( \phi \) will usually be a more general solution than \( \psi \). A substitution \( \phi \) is *no less general* than a substitution \( \psi \) if there is a substitution \( \rho \) such that

\[
\psi = \rho \ \text{scomp} \ \phi
\]

If such an equation holds, then \( \psi \) is said to be an *extension* of \( \phi \).

If we have constructed a solution \( \phi \) of a system of type equations, and we have done no more than is necessary to satisfy the equations, we will have a solution which is *maximally general*, in the sense that it is no less general than any other solution.

For an example (in informal terms), consider the type expressions

\[
T_1 = (A \rightarrow B) \rightarrow C \\
T_2 = (B \rightarrow A) \rightarrow (A \rightarrow B)
\]

The substitutions \( \phi_1 \) and \( \phi_2 \), where

\[
\phi_1 \ A = B, \ \phi_1 \ C = (B \rightarrow B) \\
\phi_2 \ B = A, \ \phi_2 \ C = (A \rightarrow A)
\]

are both unifiers of \( T_1 \) and \( T_2 \). In fact, they are examples of maximally general unifiers: they each do (one version of) the minimum necessary to make \( T_1 \) and \( T_2 \) equal, so that any other unifier of \( T_1 \) and \( T_2 \) is an extension of each of them.
The problem of unification is to find a maximally general idempotent unifier of a set of pairs of expressions. The method we use is Robinson's [1965] unification algorithm. It is convenient when coding the algorithm to concentrate on the problem of extending a given substitution, which solves a set of equations
\[ t_1 = t'_1; \ldots; t_k = t'_k \]
to one that solves an extended set
\[ t_1 = t'_1; \ldots; t_k = t'_k; t_{k+1} = t_{k+1}' \]
So we shall pose the problem in the following way. Given a pair \((t_1, t_2)\) of type expressions, and an idempotent substitution \(\phi\), our algorithm should return \text{FAILURE} if there is no extension of \(\phi\) which unifies \((t_1, t_2)\), and it should return \text{OK} \(\psi\), where \(\psi\) is an idempotent unifier of \((t_1, t_2)\) which extends \(\phi\). (In fact, the one we construct will be maximally general among such extensions of \(\phi\).)

The simplest equation we can consider is one of the form

\[ \text{TVAR } tvn = t \]

To handle such cases in the unification algorithm, we will make use of the following function:

\[
\text{extend :: subst} \rightarrow \text{tvname} \rightarrow \text{type_exp} \rightarrow \text{reply subst}
\]

\[
\text{extend } \phi \text{ } tvn \text{ } t \quad = \text{OK } \phi, \quad t = \text{TVAR } tvn \\
\text{extend } \phi \text{ } tvn \text{ } t \quad = \text{FAILURE}, \quad tvn \text{ } $in \text{tvars_in } t \\
\text{extend } \phi \text{ } tvn \text{ } t \quad = \text{OK } (((\delta \text{ } tvn \text{ } t) \text{ } $scomp \phi))
\]

An expression \((\text{extend } \phi \text{ } tvn \text{ } t)\) will be evaluated only when:

(i) \(\phi\) is an idempotent substitution (the solution we are trying to extend);
(ii) \(t\) is a fixed point of \(\phi\);
(iii) \(tvn\) is unmoved by \(\phi\) (\(tvn\) does not already have a value under \(\phi\)).

The value of the expression is either \text{FAILURE}, or of the form \((\text{OK } \phi')\), where \(\phi'\) is an idempotent substitution extending \(\phi\), such that

\[
\text{sub_type } \phi' \text{ } t' = t \quad \text{ if } t' = \text{TVAR } tvn \\
\text{sub_type } \phi' \text{ } t' = \text{sub_type } \phi \text{ } t' \quad \text{otherwise}
\]

In fact, \(\phi'\) is maximally general among extensions of \(\phi\) which solve the equation:

\[ \text{TVAR } tvn = t \]

Note that if \(\phi\) is idempotent, \(t\) is a fixed point of \(\phi\) and \(tvn\) is moved by \(\phi\), then \(tvn\) can occur in neither \((\phi \text{ } tvn)\) nor \(t\).
We can code the unification algorithm as follows:

\[
\text{unify :: subst} \to (\text{type}_\text{exp}, \text{type}_\text{exp}) \to \text{reply subst}
\]

\[
\text{unify \ phi } ((\text{TVAR } \text{tvn}), t) \\
= \text{extend \ phi \ tvn \ phi}, \quad \text{phi} = \text{TVAR \ tvn} \\
= \text{unify \ phi \ (phi}tvn,\phi) \\
\quad \text{where}
\quad \phi tvn = \phi \ tvn \\
\quad \phi t = \text{sub_type \ phi \ t}
\]

\[
\text{unify \ phi } ((\text{TCONS } tcn \ ts), (\text{TVAR } \text{tvn})) \\
= \text{unify \ phi } ((\text{TVAR } \text{tvn}), (\text{TCONS } tcn \ ts))
\]

\[
\text{unify \ phi } ((\text{TCONS } tcn \ ts), (\text{TCONS } tcn' \ ts')) \\
= \text{unifyl \ phi } (ts \ \$\ \text{zip} \ ts'), \quad tcn = tcn'
\]

\[= \text{FAILURE}\]

The function \(\text{zip}\), which is generally useful, turns a pair of lists into a list of pairs, whose length is the same as that of the shorter of the lists:

\[
\text{zip :: [\*] \to [\*\*] \to [(\*\*,\*)]} \\
\text{zip []} \ xs \quad = \quad [\\] \\
\text{zip \ (x:xs)} \ [] \quad = \quad [] \\
\text{zip \ (x:xs)} \ (y:ys) \quad = \quad (x,y):\text{zip} \ xs \ ys
\]

The function \(\text{unifyl}\) is defined such that \((\text{unifyl \ phi \ pts})\) constructs a substitution extending \phi which unifies corresponding entries in the list of pairs \pts. This function is also generally useful, so it is defined globally too.

\[
\text{unifyl :: subst} \to ([\text{type}_\text{exp},\text{type}_\text{exp}]) \to \text{reply subst}
\]

\[
\text{unifyl \ phi \ eqns} = \text{foldr \ unify'} (\text{OK \ phi}) \ eqns \\
\quad \text{where}
\quad \text{unify'} \ eqn (\text{OK \ phi}) = \text{unify \ phi \ eqn} \\
\quad \text{unify'} \ eqn \ FAILURE = \ FAILURE
\]

This completes the definition of the unification algorithm.

It is important to see why the unification algorithm terminates. After all, in the definition above we have defined the value of \((\text{unify \ (TVAR \ tvn) \ t})\) in terms of \((\text{unify \ phi}tvn \ \phi)\) where \phi tvn = (\phi \ tvn) and \phi t = (\text{sub_type \ phi \ t}), which may be very much larger expressions than \((\text{TVAR \ tvn})\) and \(t\). However, we only use that clause of the definition in circumstances when \text{tvn} cannot occur in \phi tvn or \phi t. Define the solution set of \phi to be the set of variables which occur in an expression \((\phi \ tvn')\), where \text{tvn'} is moved by \phi. We can prove that \((\text{unify \ phi \ (t1,t2)})\) terminates, by a nested induction: the outer induction is on the number of variables in \(t1\) and \(t2\) which are not in the solution set of \phi, and the inner induction is on the combined length of \(t1\) and \(t2\).

The unification algorithm has many applications other than type-checking. In particular it is a key algorithm in the implementation of programming languages such as Prolog.
9.5 Keeping Track of Types

When type-checking an expression with free variables, there are two ways to proceed.

9.5.1 Method 1: Look to the Occurrences

We can find the constraints imposed on the types of the free variables by the manner in which they occur in the expression. In a complete program, the free variables must stand for the system’s built-in functions or functions associated with type definitions. We would then look to see whether the types deduced for each occurrence of a free variable can be instances of the type supplied a priori for that variable. When type-checking a lambda abstraction ($\lambda x . E$), we would check that the types deduced for the various occurrences of $x$ within $E$ can be unified to the same type expression, and we would handle occurrences of defined variables in the right-hand sides of a letrec-expression in the same way.

It is quite possible to develop a type-checker along these lines: one is presented in Damas [1985].

9.5.2 Method 2: Look to the Variables

It is technically rather a nuisance that distinct occurrences of the same variable in an expression are associated with different type expressions. Is there something which we can associate with each variable instead?

Suppose we wish to type-check a let-expression. First of all we type-check the definitions of the let, thus deducing a type for each variable defined by the let. Then it seems that we could associate each variable with its type, and proceed to type-check the body of the let-expression. At each occurrence of one of these defined variables in the body, we should construct an instance of its associated type, substituting fresh type variables for the schematic variables in the type (see Section 8.5.3). However, as we discovered in Section 8.5.5, some of the variables in the type are constrained and should not be substituted for, and the instantiation mechanism must take account of this.

What is needed, therefore, is to associate with each variable a kind of type template, in which the schematic variables are distinguished from the non-schematic variables. Then the template can be instantiated by copying it, substituting a fresh type variable for each occurrence of a schematic variable (but copying non-schematic variables unchanged). This type template is called a type scheme. To summarize:

(i) The schematic type variables in a type scheme associated with a variable are those that may be freely instantiated to conform with the type constraints on the various occurrences of that variable.

(ii) All the other (non-schematic) variables in a type scheme are constrained,
and must not be instantiated when instantiating the type scheme. As we
remarked in Section 9.4, they behave in a similar way to the unknowns of
a mathematical equation. For example, consider the simultaneous
equations
\[
\begin{align*}
a_{1,1} \times x_1 + a_{1,2} \times x_2 &= b_1 \\
a_{2,1} \times x_1 + a_{2,2} \times x_2 &= b_2
\end{align*}
\]
We seek values for the unknowns \( x_1 \) and \( x_2 \), by solving the equations, but
they must be consistently instantiated, so that \( x_1 \) stands for the same value
wherever it occurs (and likewise \( x_2 \)).

By analogy, we will refer to the non-schematic variables of a type
scheme as \textit{unknowns}. They are the type variables whose values we seek
by solving the system of type constraints implied by the structure of an
expression.

(In papers about type-checking, schematic variables are often called \textit{generic}
variables, and unknowns are called \textit{non-generic}. We mention this only to
make it easier to link up with the literature, and will not use that terminology
here.)

There is a partial analogy between type schemes and lambda abstractions.
The schematic variables of a type scheme correspond to the formal parameter
of a lambda abstraction, and the unknowns of a type scheme correspond to
the free variables of a lambda abstraction. Applying a lambda abstraction to
an argument involves constructing an instance of its body, substituting the
argument for occurrences of the formal parameter (but copying free variables
unchanged). This is very similar to the process of instantiating a type scheme,
which involves constructing an instance of the type scheme template, substituting
fresh type variables for occurrences of the schematic variables
(but copying unknowns unchanged).

We will represent type schemes by objects of the type

\[
type\_scheme ::= \text{SCHEME } [\text{tvname}] \text{ type\_exp}
\]

A type variable occurring in a type scheme (\text{SCHEME scvs e}) is schematic if
its name occurs in the list \text{scvs}, otherwise it is an unknown.

\[
\begin{align*}
\text{unknowns\_scheme} ::& \quad \text{type\_scheme } \rightarrow [\text{tvname}] \\
\text{unknowns\_scheme} (\text{SCHEME scvs t}) &= \text{tvars\_in t $bar scvs}
\end{align*}
\]

where

\[
\begin{align*}
\text{bar} :: & \quad [*] \rightarrow [*] \rightarrow [*] \\
\text{bar} \ xs \ ys &= [\ x \leftarrow xs \mid (x \ $\in ys) ] \\
\text{in} :: & \quad [*] \rightarrow [*] \rightarrow \text{bool} \\
\text{in} \ x' [ ] &= \text{False} \\
\text{in} \ x' (x:\xs) &= \text{True,} \quad x = x' \\
& \quad = x' \ $\in \xs
\end{align*}
\]
During the course of type-checking we will have occasion to apply a substitution to a type scheme, to reflect additional information we have on its unknowns. When doing this, we should take care that only the unknowns are affected (remember that the schematic variables function like the formal parameter of a lambda abstraction, and have only local significance):

\[
\text{sub\_scheme} :: \text{subst} \to \text{type\_scheme} \to \text{type\_scheme} \\
\text{sub\_scheme} \text{ phi} (\text{SCHEME} \text{ scvs} \text{ t}) \\
= \text{SCHEME} \text{ scvs} (\text{sub\_type} (\text{exclude phi scvs}) \text{ t})
\]

where

\[
\text{exclude phi scvs tvn} = \text{TVAR} \text{ tvn, tvn}$ in scvs \\
= \text{phi tvn}
\]

In Section 2.2.6 we demonstrated the irritating problem of 'name-capture', whereby a free variable of a lambda abstraction could become bound by being substituted inside another lambda abstraction. There is a similar problem here with substitution into type schemes. We must take care that the expression

\[
\text{sub\_scheme} \text{ phi} (\text{SCHEME} \text{ scvs} \text{ t})
\]

is only evaluated when the schematic variables scvs are distinct from any variables occurring in the result of applying the substitution phi to any of the unknowns of t. Otherwise a type variable in the range of the substitution (which is always an unknown) might surreptitiously be changed into a schematic variable. The way in which we ensure this is to guarantee that the names of the schematic type variables in the type scheme are always distinct from those which can occur in the range of the substitution (which are always unknowns).

### 9.5.3 Association Lists

Having decided to associate a type scheme with each free variable in an expression, rather than a type expression with each occurrence of a free variable, we now have to decide how this information should be provided to the type-checker. There are two requirements on the data structure we use:

(i) It should provide a mapping from the free variables of the expression to type schemes.

(ii) We should be able to determine the range of that mapping.

To understand the second point, consider type-checking (let \(x = E\) in \(E')\). We start by deriving a type \(t\) for \(E\), in a type environment

\[
x_1 :: ts_1, \ldots, x_k :: ts_k
\]

which associates a type scheme \(ts_i\) with each variable \(x_i\) free in \(E\) (the \(ts_i\) thus constitute the range of the type environment). In other words, we attempt to
build a solution \( \phi \) to the type equations implied by the structure of \( E \), such that

\[
E :: t \quad \text{provided that} \quad x_1 :: t_{s_1}, \ldots, x_k :: t_{s_k}
\]

where \( t_{s_i} \) is the image of \( t_s \) under the substitution \( \phi \). We then form the type scheme \( ts \) to be associated with \( x \) when type-checking \( E' \), in the extended environment

\[
x_1 :: t_{s_1}, \ldots, x_k :: t_{s_k}, x :: ts
\]

The schematic variables of \( ts \) are all of the type variables of \( t \) except those that are unknown (non-schematic) in any of the schemes \( t_{s_1}, \ldots, t_{s_k} \). So whatever data structure we choose to represent the environment of the type-checker, it should give us ready access to the set of unknowns in its range (the \( t_{s_i} \)).

An association list provides us with a suitable data structure.

\[
\text{assoc\_list} * * * \equiv [(*,**)]
\]

Here * stands for the type of keys, and ** for the type of associated values. A key \( k \) is associated with a value \( v \) by means of the pair \( (k,v) \). The partial function itself is represented by a list of such associations. We shall use \( al, al' \), etc. as variables over association lists.

\[
\text{dom} :: \text{assoc\_list} * * * \to [*] \\
\text{dom} al = [ k \mid (k,v) <- al]
\]

\((\text{dom } al)\) returns a list (possibly with duplications) of the keys associated with values in the list, which is how we shall represent the domain of a partial function.

\[
\text{val} :: \text{assoc\_list} * * * \to * \to * * \\
\text{val} al k = \text{hd} [ v \mid (k',v) <- al ; k = k' ]
\]

If \( k \) is a key in \((\text{dom } al)\), then \((\text{val } al k)\) returns the first value in the list which is associated with \( k \). When using this function, we should be careful to ensure that the second argument belongs to the domain of the association list.

\[
\text{install } al k v = (k,v):al
\]

\((\text{install } al k v)\) returns an association list which implements the same partial function as \( al \), except that the key \( k \) is now mapped to the value \( v \).

\[
\text{rng} :: \text{assoc\_list} * * * \to [**] \\
\text{rng} al = \text{map} (\text{val } al) (\text{dom } al)
\]

The property which \( \text{rng} \) is intended to satisfy is that every entry in \((\text{rng } al)\) is a value of \((\text{val } al)\).

We shall represent the information passed to the type-checker about the
types of the free variables of an expression by means of an object of the following type:

\[
\text{type_env} \equiv \text{assoc_list vname type_scheme}
\]

We shall use \( \gamma, \gamma', \) etc. as variables standing for type environments. The functions \text{unknowns_scheme} and \text{sub_scheme} can be extended to act on type environments, in the obvious way:

\[
\begin{align*}
\text{unknowns_te} & :: \text{type_env} \rightarrow [\text{vname}] \\
\text{unknowns_te} \, \gamma & = \text{appendlist} \, (\text{map} \, \text{unknowns_scheme} \, (\text{rng} \, \gamma)) \\
\text{appendlist} & :: [\, [\ast] \, ] \rightarrow [\ast] \\
\text{appendlist} \, lls & = \text{foldr} \, (++) \, [[]] \, lls \\
\text{sub_te} & :: \text{subst} \rightarrow \text{type_env} \rightarrow \text{type_env} \\
\text{sub_te} \, \phi \, \gamma & = [\, (x, \text{sub_scheme} \, \phi \, st) \mid (x, st) \leftarrow \gamma \, ]
\end{align*}
\]

### 9.6 New Variables

When type-checking a closed expression, we first assigned a distinct type variable to each subexpression, and then wrote down equations expressing the constraints on those variables imposed by the structure of the expression. When type-checking an expression containing variables defined in a let- or letrec-expression, we chose first to work out the schematic types of those variables (i.e. we checked the definitions first). We then assigned to each occurrence of such a variable a type expression obtained by substituting new unknown variables for the schematic variables, using a distinct set of unknowns for each distinct occurrence.

So we will need a mechanism that enables us to 'make up' new type variables, and guarantees that they are distinct from type variables we may introduce in the future. There are many ways to provide such a mechanism. The one we adopt here is to postulate that there is a type \text{name_supply}, and functions

\[
\begin{align*}
\text{next_name} & :: \text{name_supply} \rightarrow \text{tname} \\
\text{deplete} & :: \text{name_supply} \rightarrow \text{name_supply} \\
\text{split} & :: \text{name_supply} \rightarrow (\text{name_supply}, \text{name_supply})
\end{align*}
\]

such that if \( ns \) is a name supply, then \((\text{next_name} \, ns)\) is distinct from any name supplied by \((\text{deplete} \, ns)\), and if \((ns0, ns1) = \text{split} \, ns\), then any name supplied from \( ns0 \) is distinct from any name supplied by \( ns1 \). One way to implement such a type is to (re)define \( t\text{name} \), thus:

\[
\begin{align*}
t\text{name} & \equiv [\text{num}] \\
\text{name_supply} & \equiv t\text{name} \\
\text{next_name} \, ns & = ns \\
\text{deplete} \, (n:ns) & = (n+2:ns) \\
\text{split} \, ns & = (0:ns, 1:ns)
\end{align*}
\]
For example, if we start with the name supply [0], then the names it will supply are [0], [2], [4], ..., while the names supplied by splitting the supply into [0,0] and [1,0] will be [0,0], [2,0], [4,0], ..., and [1,0], [3,0], [5,0], ..., respectively. (The +2 in the definition of deplete is only an artifice to ensure that the two halves of a split name supply are forever distinct.)

The function name_sequence returns from a name supply an infinite sequence of distinct names derived from that supply:

\[
\text{name_sequence :: name_supply -> [tname]} \\
\text{name_sequence ns = next_name ns : name_sequence (deplete ns)}
\]

In practice, it is probably better to adopt an approach other than the supply of new variables, according to which variables are named by integers, and the name supply represented by the name of the next variable to be allocated. The type-checker would then take the name supply as an argument, and return the depleted supply as part of its value. We have adopted an approach which wastes large portions of the variable name space, in order not to encumber the type-checker code with a further avoidable detail.

## 9.7 The Type-checker

Finally, we are in a position to define the type-checker. This will take the form of a function \( \text{tc gamma ns e} \) where

(i) \( \text{gamma} \) is a type environment, associating type schemes with each of the free variables of \( e \). When the type-checker is invoked upon a complete program, this type environment should be initialized to contain declarations of the types of the built-in system-supplied identifiers.

(ii) \( \text{ns} \) is a supply of type variable names.

(iii) \( \text{e} \) is the expression to be checked.

The value returned will be a reply, which in the case of success will return a pair of the form \( (\phi, t) \) where

(i) \( \phi \) is a substitution defined on the unknown type variables in \( \text{gamma} \).

(ii) \( t \) is a type derived for the expression \( e \), in the type environment \( \text{sub_te phi gamma} \). It will in fact be a fixed point of the substitution \( \phi \).

In other words, if

\[
\text{tc gamma ns e = OK (phi,t)}
\]

then \( e :: t \) can be derived from \( \text{gamma} \), provided that each unknown \( tvn \) in \( \text{gamma} \) has the value given it by \( \phi \).

We shall define the function \( \text{tc} \) by induction on the structure of the expression, with a different clause for each form which an expression can take:
Section 9.7 The type-checker

\[ tc :: \text{type_env} \rightarrow \text{name_supply} \rightarrow \text{vexp} \rightarrow \text{reply} (\text{subst}, \text{type_exp}) \]
\[ tc \text{ gamma ns (VAR x)} = \text{tcvar gamma ns x} \]
\[ tc \text{ gamma ns (AP e1 e2)} = \text{tcap gamma ns e1 e2} \]
\[ tc \text{ gamma ns (LAMBDA x e)} = \text{tclambda gamma ns x e} \]
\[ tc \text{ gamma ns (LET xs es e)} = \text{tclet gamma ns xs es e} \]
\[ tc \text{ gamma ns (LETRC xs es e)} = \text{tcletrec gamma ns xs es e} \]

We will describe each of these cases in a separate section, beginning at Section 9.7.2. First, however, we define a useful auxiliary function \( tcl \).

9.7.1 Type-checking Lists of Expressions

It is convenient to define a function \( tcl \text{ es gamma n} \) which applies to a list of expressions \( es \), and will return in the case of success a similar result \( \text{OK (phi,ts)} \), where \( ts \) is a list of types derived for corresponding components of the list \( es \) in the type environment \( (\text{sub_te phi gamma}) \). \( phi \) embodies all the constraints on \( gamma \) necessary to derive those types simultaneously. The function is defined from \( tc \) by the equations:

\[ tcl :: \text{type_env} \rightarrow \text{name_supply} \rightarrow [\text{vexp}] \rightarrow \text{reply} (\text{subst}, [\text{type_exp}]) \]
\[ tcl \text{ gamma ns [ ] } = \text{OK (id_subst,[])} \]
\[ tcl \text{ gamma ns (e:es)} = \text{tcl1 gamma ns0 es (tc gamma ns1 e)} \]
\[ \text{where } (\text{ns0,ns1}) = \text{split ns} \]

\[ \text{tcl1 gamma ns es FAILURE } = \text{FAILURE} \]
\[ \text{tcl1 gamma ns es (OK (phi,t)) } = \text{tcl2 phi t (tcl gamma' ns es)} \]
\[ \text{where gamma'} = \text{sub_te phi gamma} \]

\[ \text{tcl2 phi t FAILURE } = \text{FAILURE} \]
\[ \text{tcl2 phi t (OK (psi,ts)) } = \text{OK (psi $scomp phi, (sub_type psi t) : ts)} \]

The substitution can be thought of as built up in two stages. In the first stage, we type-check each entry in the list, in the type environment ‘seen’ through the substitutions derived for previous entries. Then in the second stage, we form the substitution by cumulative composition, and ensure that each type returned for an expression is a fixed point of the composite substitution.

9.7.2 Type-checking Variables

When type-checking a variable \( x \) in a given type environment \( gamma \), with name supply \( ns \), we look up the type scheme associated with that variable by \( gamma \). Recall that in a type scheme, a type variable is \textit{either schematic}, in which case we substitute a fresh type variable for it, or \textit{unknown}, in which case we leave it as it is.

So we return a new instance of the schematic type associated with the variable, in which the schematic variables have been replaced by fresh type variables. In this way, the type constraints on different occurrences of a
variable $x$ can be resolved independently, as indicated by the schematic variables in the type scheme associated with $x$.

\[
\text{tcvar} :: \text{type_env} \to \text{name_supply} \to \text{vname} \\
\qquad \quad \to \text{reply (subst,type_exp)} \\
\text{tcvar gamma ns x} \\
\quad = \text{OK (id\_subst, newinstance ns scheme)} \\
\quad \quad \text{where scheme} = \text{val gamma} x
\]

where

\[
\text{newinstance} :: \text{name_supply} \to \text{type_scheme} \to \text{type_exp} \\
\text{newinstance ns (SCHEME scvs t)} \\
\quad = \text{sub\_type phi t} \\
\quad \quad \text{where al} = \text{scvs $zip (name\_sequence ns)} \\
\quad \quad \phi = \text{al\_to\_subst al}
\]

Here we have built an association list between the schematic variables and an initial segment of the name sequence built on the given name supply. Such an association list can be made into a substitution, by means of the function:

\[
\text{al\_to\_subst} :: \text{assoc\_list tvname tvname} \to \text{subst} \\
\text{al\_to\_subst al tvn} \\
\quad = \text{TVAR (val al tvn), tvn $in (dom al)} \\
\quad = \text{TVAR tvn}
\]

### 9.7.3 Type-checking Application

When type-checking an expression ($\text{AP e1 e2}$) with respect to a type environment gamma, we first of all try to construct a substitution $\phi$ which solves the type constraints on $e1$ and $e2$ together. Suppose that the types $t1$ and $t2$ are derived for $e1$ and $e2$. We then try to construct an extension of $\phi$ which satisfies the additional constraint

\[
t1 = t2 \to t'
\]

where $t'$ is a new type variable. We obtain this extension, as usual, by unifying $t1$ with $t2 \to t'$.

\[
\text{tcap} :: \text{type_env} \to \text{name_supply} \to \text{vexp} \to \text{vexp} \\
\qquad \quad \to \text{reply (subst,type_exp)} \\
\text{tcap gamma ns e1 e2} \\
\quad = \text{tcap1 tvn (tcl gamma ns' [e1,e2])} \\
\quad \quad \text{where tvn} = \text{next\_name ns} \\
\quad \quad \text{ns' = deplete ns}
\]

\[
\text{tcap1 tvn FAILURE} \\
\quad = \text{FAILURE} \\
\text{tcap1 tvn (OK (phi,[t1,t2]))} \\
\quad = \text{tcap2 tvn (unify phi (t1,t2 $arrow (TVAR tvn)))}
\]

\[
\text{tcap2 tvn FAILURE = FAILURE} \\
\text{tcap2 tvn (OK phi) = OK (phi, phi tvn)}
\]
9.7.4 Type-checking Lambda Abstractions

When type-checking (LAMBDA x e), we know nothing at the outset about the type of x. So we associate x with a scheme of the form (SCHEME [] (TVAR tvn)), where tvn is a new type variable. Because this scheme has no schematic type variables, the various occurrences of the variable will be assigned the value of the same type variable. This is the formal counterpart of our decision to insist that all occurrences of the same LAMBDA-bound variable should have the same type.

\[
\begin{align*}
t\text{clambda} &:: \text{type_env} \rightarrow \text{name_supply} \rightarrow \text{vname} \rightarrow \text{vexp} \\
& \quad \rightarrow \text{reply (subst,type_exp)} \\
\text{tclambda gamma ns x e} \\
& = \text{tclambda1 tvn (tc gamma' ns' e)} \\
& \quad \text{where ns' = delete ns} \\
& \quad \quad \gamma = \text{new_bvar (x,tvn)} : \gamma \\
& \quad \quad \text{tvn} = \text{next_name ns} \\
\text{tclambda1 tvn FAILURE} \\
& = \text{FAILURE} \\
\text{tclambda1 tvn (OK (phi,t))} \\
& = \text{OK (phi, (phi tvn) $arrow t)} \\
\text{new_bvar (x,tvn)} & = (x,\text{SCHEME [] (TVAR tvn)})
\end{align*}
\]

9.7.5 Type-checking let-expressions

When type-checking an expression (LET xs es e), we first of all type-check the right-hand sides in the list es. We then have to update the environment so that it associates the appropriate schematic types with the names in the list xs, and type-check the body e. The details of constructing the 'appropriate' schematic types are slightly involved, so we shall hide them in the definition of a function add_decs.

\[
\begin{align*}
t\text{clet} &:: \text{type_env} \rightarrow \text{name_supply} \\
& \quad \rightarrow [\text{vname}] \rightarrow [\text{vexp}] \rightarrow \text{vexp} \\
& \quad \rightarrow \text{reply (subst, type_exp)} \\
\text{tclet gamma ns xs es e} \\
& = \text{tclet1 gamma ns0 xs e (tcl gamma ns1 es)} \\
& \quad \text{where (ns0,ns1) = split ns} \\
\text{tclet1 gamma ns xs e FAILURE} \\
& = \text{FAILURE} \\
\text{tclet1 gamma ns xs e (OK (phi,ts))} \\
& = \text{tclet2 phi (tc gamma'' ns1 e)} \\
& \quad \text{where gamma'' = add_decls gamma' ns0 xs ts} \\
& \quad \gamma = \text{sub_te phi gamma} \\
& \quad \text{(ns0,ns1) = split ns}
\end{align*}
\]
tclet2 phi FAILURE
    = FAILURE
    tclet2 phi (OK (phi',t))
    = OK (phi' $scomp phi, t)

The purpose of add_decls is to update a type environment gamma so that it
associates schematic types formed from the types ts with the variables xs. The
variables which become schematic variables are those that are not unknowns
in gamma. The definition is slightly complicated by our obligation to ensure
that the names of the schematic variables are distinct from the names of any
unknown variables which can occur in the range of a substitution. We use the
name sequence ns to supply new names for the schematic variables.

add_decls :: type_env -> name_supply
    -> [vname] -> [type_exp] -> type_env
add_decls gamma ns xs ts
    = (xs $zip schemes) ++ gamma
    where schemes = map (genbar unknowns ns) ts
          unknowns = unknowns_le gamma

genbar unknowns ns t
    = SCHEME (map snd al) t'
    where al = scvs $zip (name_sequence ns)
            scvs = (nodups (tvars_in t)) $bar unknowns
            t' = sub_type (al_to_subst al) t

Here snd is a function which projects a pair to its second coordinate. The
projection functions for pairs are defined by

fst :: (*,**) -> *
fst (x,y) = x

snd :: (*,**) -> **
snd (x,y) = y

The function nodups returns a list with the same set of entries as its
argument list, but without duplicates:

nodups :: [*] -> [*]
nodups xs = f [] xs
    where
        f acc [] = acc
        f acc (x:xs) = f acc xs, x $in acc
                        = f (x:acc) xs

9.7.6 Type-checking letrec-expressions

The definition of the function invoked to type-check expressions
(LETREC xs es s) is rather intricate, as there are many things to do. In
outline, they are these:
(i) Associate new type schemes with the variables xs. These schemes will have no schematic variables, in accordance with our decision to insist that all occurrences of a defined name in the right-hand sides of a recursive definition should have the same type.

(ii) Type-check the right-hand sides. If successful, this will yield a substitution and a list of types which may be derived for the right-hand sides if the type environment is constrained by the substitution.

(iii) Unify the types derived for the right-hand sides with the types associated with the corresponding variables, in the context of that substitution. This is in accordance with our decision that the right-hand sides of recursive definitions must receive the same types as occurrences of the corresponding variables. Should the unification succeed, that constraint can be met.

(iv) We are now in much the same situation as we were in with expressions of LET form, when the definitions had been processed, and it remained to type-check the body e, after updating the type environment with appropriate schematic types.

tcletrec :: type_env -> name_supply
  
  -> [vname] -> [vexp] -> vexp
  
  -> reply (subst, type_exp)

  tcletrec gamma ns xs es e
    = tcletrec1 gamma ns0 nbvs e (tcl (nbvs ++ gamma) ns1 es)
    where (ns0,ns') = split ns
          (ns1,ns2) = split ns'
          nbvs = new_bvars xs ns2

  new_bvars xs ns = map new_bvar (xs $zip (name_sequence ns))

  tcletrec1 gamma ns nbvs e FAILURE
    = FAILURE

  tcletrec1 gamma ns nbvs e (OK (phi,ts))
    = tcletrec2 gamma' ns nbvs' e (unifyl phi (ts $zip ts'))
    where ts' = map old_bvar nbvs'
          nbvs' = sub_te phi nbvs
          gamma' = sub_te phi gamma

  old_bvar (x,SCHEME [] t) = t

  tcletrec2 gamma ns nbvs e FAILURE
    = FAILURE

  tcletrec2 gamma ns nbvs e (OK phi)
    = tclet2 phi (tc gamma'' ns1 e)
    where ts = map old_bvar nbvs'
          nbvs' = sub_te phi nbvs
          gamma' = sub_te phi gamma
          gamma'' = add_decls gamma' ns0 (map fst nbvs) ts
          (ns0,ns1) = split ns

The definition of the type-checker is now complete.
References

Department of Computer Science, University of Edinburgh. April.


Jouannaud (editor). LNCS 201. Springer Verlag.