Part 3
Analysis of functional programs

Chapter 6 Type assignment systems

Chapter 7 Strictness analysis

This part treats two important kinds of static analysis of functional programs: *type assignment systems* and *strictness analysis*.

The purpose of static analysis is to deduce certain properties of a particular program at compile-time. The properties that are of interest can vary from properties that are important for the *correctness* of the program, such as type checking, to properties that are important for an *efficient execution,* such as complexity analysis, termination analysis, reference count analysis and strictness analysis.

Properties are generally studied on the level of the model of computation. Most interesting properties turn out to be undecidable. Therefore, with a static analysis usually only a decidable approximation of the desired property can be derived. This results in a methodology for the analysis that can be applied on the level of the model and also in practice directly on some internal data structure of a compiler. Every static analysis tries to deduce at compile-time as much information as possible from the program.

Type systems are traditionally studied in the λ-calculus via formal schemes to infer types of λ-terms. We also study a type system for TRSs that can easily be transferred to FGRSs.

Strictness analysis is usually also studied in the λ-calculus via a technique called *abstract interpretation*. We also describe a methodology for the analysis that is performed in rewriting systems via a technique called *abstract reduction*.

Chapter 6
Type assignment systems

6.1 Type assignment for the λ-calculus
6.2 Polymorphism and recursion
6.3 Type assignment for term rewriting systems

Adding type information to a program is important for several reasons. Type information makes a program more *readable* because it gives a human being additional information about the structure of a program. Furthermore, type information plays an essential role in the implementation: the information is needed to obtain an *efficient implementation* and it also makes *separate compilation* of program modules possible (see Part 4). However, one of the most important aspects of type systems is that they warn the programmer at an early stage (at compile-time) if a program contains *severe errors.* If a program is type-error free, it is assumed to be safe to run: *'Typed programs cannot go wrong'* (Milner, 1978). So a compile-time analysis of a program rejects programs in which errors may occur at run-time. However, some errors depend on the values computed at run-time (e.g. divide by zero or stack overflow). They cannot be detected at compile-time. So only some *special classes of run-time errors* can be found at compile-time.

Typing concerns the analysis of the domain and the range on which functions are defined and a check that these functions are indeed applied consistently. The exact domain and range of a function can depend on its run-time behaviour and is therefore undecidable at compile-time. A decidable compile-time approximation can be found by looking at the syntactic structure of the program only. This chapter deals with such type assignment systems for functional languages.

Although, as shown in Part 2, graph rewriting systems are best suited as a basis to reason about functional languages and their implementation, a type system that exploits the special properties of GRSs (e.g. by using cyclic types) has not yet been developed. Therefore, type systems for functional languages are still based upon type systems for the λ-calculus. In this chapter, type systems are treated in their historical context: the λ-calculus, followed by a context that is more close to functional languages: term rewriting systems. These type systems for the λ-calculus and TRSs can be transferred easily to FGRSs.

The basis of every type system for the λ-calculus (Section 6.1) has been defined by H.B. Curry (Curry and Feys, 1958). Type assignment in this system is decidable and most type assignment algorithms used in functional programming languages are based upon it. Unfortunately, the type system can only type terms that are **strongly normalizing**, i.e. terms that reduce to normal form in all possible reduction orders. Furthermore, the type system can only type a subclass of these strongly normalizing terms. This subclass does not contain terms in which self-application occurs. Examples of terms with self-application are fixed point combinators which play an essential role in dealing with recursion.

The λ-calculus, and a type assignment system for it, can be extended in a natural way to incorporate recursion. In this chapter it will be done by presenting an intermediate language Λ^+ (Section 6.2) in which λ-terms can be named such that recursion can be expressed. Both Milner's as well as Mycroft's type assignment systems are presented. Their properties are analysed and compared.

Finally, also type assignment on TRSs is discussed (Section 6.3) as a generalization of the Mycroft type assignment system for Λ^+. It turns out that some general properties of Mycroft's system (like subject reduction) are only valid for TRSs under certain restrictions.

6.1 Type assignment for the λ-calculus

H.B. Curry was the first to study typing for the pure λ-calculus. He designed a straightforward system that is completely syntax-directed: like the definition of the set of λ-terms, the Curry type system also expresses *abstraction* (function definition) and (function) *application*.

First, the basic concepts of this system are explained using the concept of *typed trees*. A tree representation of a λ-term can be labelled with types according to *type assignment rules*. The resulting typed tree corresponds with a formal logic deduction in standard type theory. Such a deduction is constructed with inference rules that correspond directly to the type assignment rules that label the tree.

Then, it is shown how intuitively a type can be derived in this system. A formal specification of several related type assignment algorithms is given in a functional style.

Finally, some general properties of the Curry system are discussed.

6.1.1 Basic concepts of the Curry type system

Intuitively, the Curry type system restricts all occurrences of the *same bound variable* to have the *same type* and furthermore in every application the *argument* is of the desired *domain type* and the *result* of the function applied to that argument has to be of the desired *range type*. This means that a function *definition* is of type $\sigma \rightarrow \tau$ if the function body is of type τ (the range) and in the body the argument is used consistently with type σ (the domain). For a function *application* this means that the application is of type τ when a function of type $\sigma \rightarrow \tau$ is applied to an argument of type σ.

Curry types

The set of Curry types contains all types that can be built from type variables and the type constructor $\rightarrow$ (arrow). Its inductive definition is:

(1) all type variables φ_0, φ_1, … are Curry types.

(2) if σ and τ are Curry types, then $(\sigma \rightarrow \tau)$ is a Curry type.

In this chapter, σ, τ, ρ_0, ρ_1, … are used to name arbitrary Curry types. M and N are used to name arbitrary λ-terms. Not all brackets are used in the notation of a type. The arrow constructor is *right associative* and *outermost brackets* are omitted.

Typed trees

Assigning a type to a λ-term consists of first constructing a tree representation of that term (see Chapter 3), extended with an edge to the root node. Then, the nodes and edges of this tree are labelled by adding labels of the form '::', followed by a type. This results in a *typed tree* (Figure 6.1).

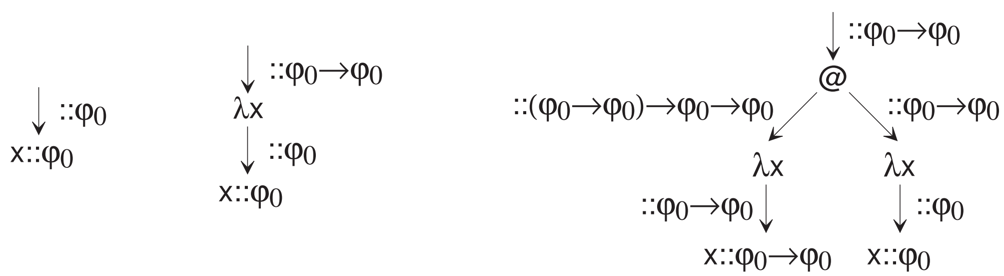

Figure 6.1 A typed tree for respectively, x, λx.x and (λx.x)(λx.x). Note that in the rightmost tree x::$\varphi_0 \rightarrow \varphi_0$ was assumed in the left subtree while x::φ_0 was assumed in the right subtree.

A **typed tree** is a tree representation of a λ-expression labelled with type information in such a way that the types of variables, application nodes and abstraction nodes are *consistent* with respect to the Curry type restrictions mentioned above. Then, the type at the top edge of a typed tree is a correct Curry type for the λ-term involved. The leaves of the tree (containing term variables) are called **assumptions**, and the set of all assumptions for a term is called the **basis**. The combination of type σ at the top edge and basis is called a **pair** for the term N. The notation N::σ is called a **statement** and expresses the fact that σ is a suitable type for N; the combination of this type and λ-term is called the **conclusion** of the typed tree.

Type assignment constraints

Type assignment constraints are a translation of the Curry type restrictions to consistency requirements for typed trees: assigning types to a λ-term is **consistent** if it satisfies the type assignment constraints defined below. There are only two type assignment constraints in Curry's system (Figure 6.2), the **application constraint** (@), and **abstraction constraint** (λ). In the literature these constraints are defined by giving type assignment rules respectively **arrow elimination** (→E) and **arrow introduction** (→I). Here, starting with typed subtrees a constraint states how these subtrees can be combined into other typed trees. Furthermore, the type assignment constraints play an important role in the type assignment algorithm that constructs a typed tree.

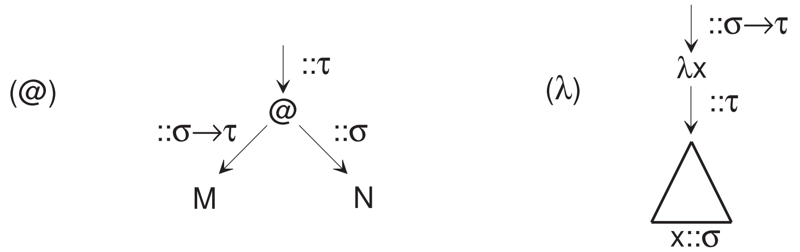

Figure 6.2 The two constraints of the Curry type system.

The type assignment constraint (@) assumes a type σ→τ was found for M and σ for N. Put together this gives type τ for the application M N.

In both subtrees the assumption x::φ_0 is made giving rise to a conflict in the root of the tree. No conclusion is possible since the types do not obey the application-constraint (@).

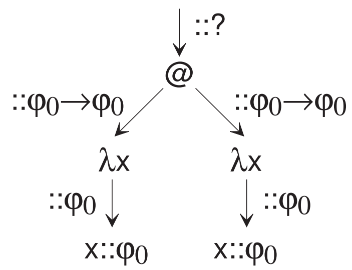

Figure 6.3 *Impossible* typed tree for (λx.x)(λx.x): inconsistent types.

The type assignment constraint (λ) is used to construct a type to a λ-abstraction term λx.M. It assumes that constructing the typed tree for M was successful and the type found for M was τ. If the assumption x::σ is the only assumption for x needed to reach the conclusion $M::\tau$ (so no other assumption like $x::\rho$ is used where ρ is different from σ) the (λ)-step can be performed. The result will be that finding a type for the λ-term $\lambda x.M$ was successful and the type found is $\sigma \to \tau$.

There is no separate constraint for term variables. The constraints (λ) and $(@)$ only limit the possible combinations of typed (sub)trees (see Figure 6.3). These constraints do not express *how* a type is found for a specific λ-term: they only formalize *what* a consistent typed tree should look like.

6.1.2 Finding a type for a term

Type assignment is decidable in Curry's system. There is an algorithm that, given a λ-term M, will say 'NO' if there is no possible typed tree for M and will yield a type for M if there is a possible typed tree for M. In this algorithm, *unification* (described below) plays a crucial role.

The basic idea of the type deduction algorithm is the following. It starts at the variables in the leaves of the tree. It assumes that a variable is of the most general type and assigns a *fresh* (i.e. new, not yet used) type variable to it. Then the tree is traversed upwards. For each visited node the consistency of the subtree with that node as root is checked against the type assignment constraints. If there is an inconsistency an attempt is made to make the tree consistent by deriving more accurate types. This is done by unification. If the unification succeeds the algorithm continues, otherwise no typed tree can be created at all.

Unification

Types consist of arrows and type variables. Types can be obtained from other types by replacing, in a consistent way, type variables by types. An operation that uniformly replaces type variables by types (i.e. each occurrence of the same variable is replaced by the same type) is called a **substitution**. Substitution is a **sound** operation on typed trees, i.e. if in a typed tree all types are replaced by a substitution with other types, then the resulting typed tree is consistent. A type that is produced by substitution is called an **instance** of the original type.

Two types that are not equal can have a common instance:

$((\varphi_6 \to \varphi_6) \to \varphi_5) \to \varphi_6 \to \varphi_6$ can be obtained from the type $\varphi_2 \to \varphi_3 \to \varphi_3$ by uniformly replacing φ_2 by $(\varphi_6 \to \varphi_6) \to \varphi_5$ and φ_3 by φ_6.
$((\varphi_6 \to \varphi_6) \to \varphi_5) \to \varphi_6 \to \varphi_6$ is also an instance of the type $(\varphi_0 \to \varphi_1) \to \varphi_0$.

Two types are **equivalent** if there exists both a substitution of type variables to type variables such that the substitution applied to the first type gives the second type as well as a similar substitution from the second type to the first type. In the following, types are considered modulo this equivalency. So equivalent types are considered to be **equal**.

A type that is an instance of two different types is called a **unifier** of these types. If two types have a unifier, they also have a **most general unifier**, i.e. all unifiers of the two types are instances of their most general unifier. Trying to find the most general unifier for two types is called **unification**. Two types are **unifiable** if a (most general) unifier exists. For Curry types, unification is decidable.

Robinson's unification algorithm

In Robinson (1965) an algorithm unify is given that takes two types σ and τ and either fails, or succeeds yielding a substitution S such that:

- $S\ \sigma = S\ \tau$, so S unifies σ and τ;
- $S\ \sigma$ is the most general unifier of σ and τ;
- S is only defined on variables occurring in σ and τ.

For example, let

$$S \;=\; \text{unify}\ \varphi_2 \to \varphi_3 \to \varphi_3\ \ (\varphi_0 \to \varphi_1) \to \varphi_0$$

then S unifies $\varphi_2 \to \varphi_3 \to \varphi_3$ and $(\varphi_0 \to \varphi_1) \to \varphi_0$ and

$$\begin{aligned} S\ \varphi_0 &= \varphi_3 \to \varphi_3 \\ S\ \varphi_1 &= \varphi_1 \\ S\ \varphi_2 &= (\varphi_3 \to \varphi_3) \to \varphi_1 \\ S\ \varphi_3 &= \varphi_3 \end{aligned}$$

and therefore

$$\begin{aligned} S\ \varphi_2 \to \varphi_3 \to \varphi_3 &= ((\varphi_3 \to \varphi_3) \to \varphi_1) \to \varphi_3 \to \varphi_3 \\ S\ (\varphi_0 \to \varphi_1) \to \varphi_0 &= ((\varphi_3 \to \varphi_3) \to \varphi_1) \to \varphi_3 \to \varphi_3 \end{aligned}$$

which gives that $((\varphi_3 \to \varphi_3) \to \varphi_1) \to \varphi_3 \to \varphi_3$ is the most general unifier of $\varphi_2 \to \varphi_3 \to \varphi_3$ and $(\varphi_0 \to \varphi_1) \to \varphi_0$. Another unifier of these two types is $((\varphi_4 \to \varphi_4) \to \varphi_4 \to \varphi_4) \to \varphi_4 \to \varphi_4$. It is indeed an instance of the most general unifier by substituting φ_1 by $\varphi_4 \to \varphi_4$ and φ_3 by φ_4.

Robinson's algorithm, written in a functional style:

```
type            :: =   Typevar typevar |   Arrow type type
substitution    = =    type -> type
typevar         = =    num
```

```
unify:: type -> type -> substitution
unify (Typevar φ0)  (Typevar φ1)  =  sub φ0 (Typevar φ1)
unify (Typevar φ)   τ             =  sub φ τ,              if not_in τ φ
unify σ             (Typevar φ)   =  unify (Typevar φ) σ
unify (Arrow σ0 σ1) (Arrow τ0 τ1) =  res1 . res0
                                     where  res0  = unify σ0 τ0
                                            res1  = unify (res0 σ1) (res0 τ1)

sub:: typevar -> type -> substitution
sub φ0 τ   =  s
              where  s  (Typevar φ1)  =  τ,               if φ0 = φ1
                                      =  Typevar φ1,      otherwise
                     s  (Arrow σ ρ)   =  Arrow ((sub φ0 τ) σ) ((sub φ0 τ) ρ)

not_in:: type -> typevar -> bool
not_in (Typevar φ0) φ1    =  φ1 ~= φ0
not_in (Arrow σ τ) φ1     =  not_in σ φ1   &  not_in τ φ1
```

In this algorithm sub φ σ returns the substitution that replaces the type variable φ by the type σ and does not affect other type variables. $R\mathbf{.}T$ (the function composition of R and T) is the substitution that first performs T and afterwards performs R.

The result of the function unify is built by function composition of the substitutions found while unifying subtypes. The substitution found while unifying the left-hand sides of two arrow types is first performed on the right-hand sides before unifying them.

Assigning types to trees

The general idea of the method to find the most general type for every λ-term in Curry's system is as follows. Start by building a tree in which all labels are empty. Then, traverse the tree to label the nodes containing variables and the edges with types, making use of the type assignment constraints. Sometimes, while trying to assign a type to a node, a unification is performed. In that case the resulting substitution is performed on the whole tree. Because substitution is a sound operation, such replacing of type variables by types in a correctly typed tree will always give a correctly typed tree. If unification fails, the whole term cannot be typed.

More precisely, the following algorithm is used: traverse the tree depth-first from left to right performing at each position one of the following three steps:

- If a leaf is encountered, the variable and the edge going into this node are labelled with the same fresh type variable.

- If an abstraction node is visited, the subtree below it is a tree that is already typed, say M. Now an abstraction step (λ) to a variable, say x, is performed, to type λx.M.

  The algorithm therefore checks the assumptions made on the corresponding x (all labelled) to see whether they are all typed with the same type. If they are the same, everything is fine. If no assumptions have been made, because the variable x did not occur in the term, a fresh type variable will be assumed. If the assumptions on x are not the same, the algorithm tries to unify the assumed types. If this succeeds, the substitution generated by this unification will be performed on the typed subtree, giving a typed subtree in which all assumptions on x are labelled with the found unifier.

  In either case the result will be a typed subtree with conclusion, say, M::τ in which all assumptions on x are labelled with the same type, say ::σ. The edge going into the abstraction node λx.M is then labelled with ::$\sigma \to \tau$.
- If an application node is visited, there are two typed subtrees with conclusions, say M::σ and N::τ. Now an application step (@) is performed. To perform this step, the types on both subtrees should match the requirements of (@).

  The algorithm employs a fresh type variable φ_0 and unifies the types σ and $\tau \to \varphi_0$. If this unification is successful, a substitution S is yielded such that $S(\sigma) = S(\tau \to \varphi_0) = \rho_0 \to \rho_1$, so $S(\sigma) = \rho_0 \to \rho_1$ and $S(\tau) = \rho_0$. Applying this substitution to the two typed subtrees yields a left subtree with conclusion M::$\rho_0 \to \rho_1$, and a right subtree with conclusion N::ρ_0. Now the (@)-constraint can be taken into account labelling the edge into the application node with ::ρ_1.

The type assignment algorithm returns the unused fresh variables, the type found for the term, and the set of typed term variables that are free in the term. The algorithm is illustrated below with two examples.

> Example 1: (λa.λb.a b)(λx.x). First, an untyped tree is constructed (Figure 6.4(a)). Then the algorithm visits the left subtree, assuming a::φ_1 and b::φ_2. An (@)-step should be performed, so a new type variable φ_3 is picked and the algorithm unifies φ_1 and $\varphi_2 \to \varphi_3$. This yields that φ_1 will be replaced by $\varphi_2 \to \varphi_3$. The edge going into the application node will be labelled by φ_3. The two abstraction nodes give no difficulties (this results in Figure 6.4(b)). Then, the algorithm visits the right subtree. It assumes a type variable for the assumption x::φ_4. Again the abstraction node gives no difficulties. This yields Figure 6.4(c). Lastly, the algorithm visits the top node. It is an application node so the subtree should satisfy the @-constraint. The types involved are $(\varphi_2 \to \varphi_3) \to \varphi_2 \to \varphi_3$ and $\varphi_4 \to \varphi_4$; a fresh type variable φ_5 is taken and $(\varphi_2 \to \varphi_3) \to \varphi_2 \to \varphi_3$ and $(\varphi_4 \to \varphi_4) \to \varphi_5$ are unified. This yields the substitution that replaces both φ_2 and φ_3 by φ_4, and φ_5 by $\varphi_4 \to \varphi_4$ (Figure 6.4(d)).

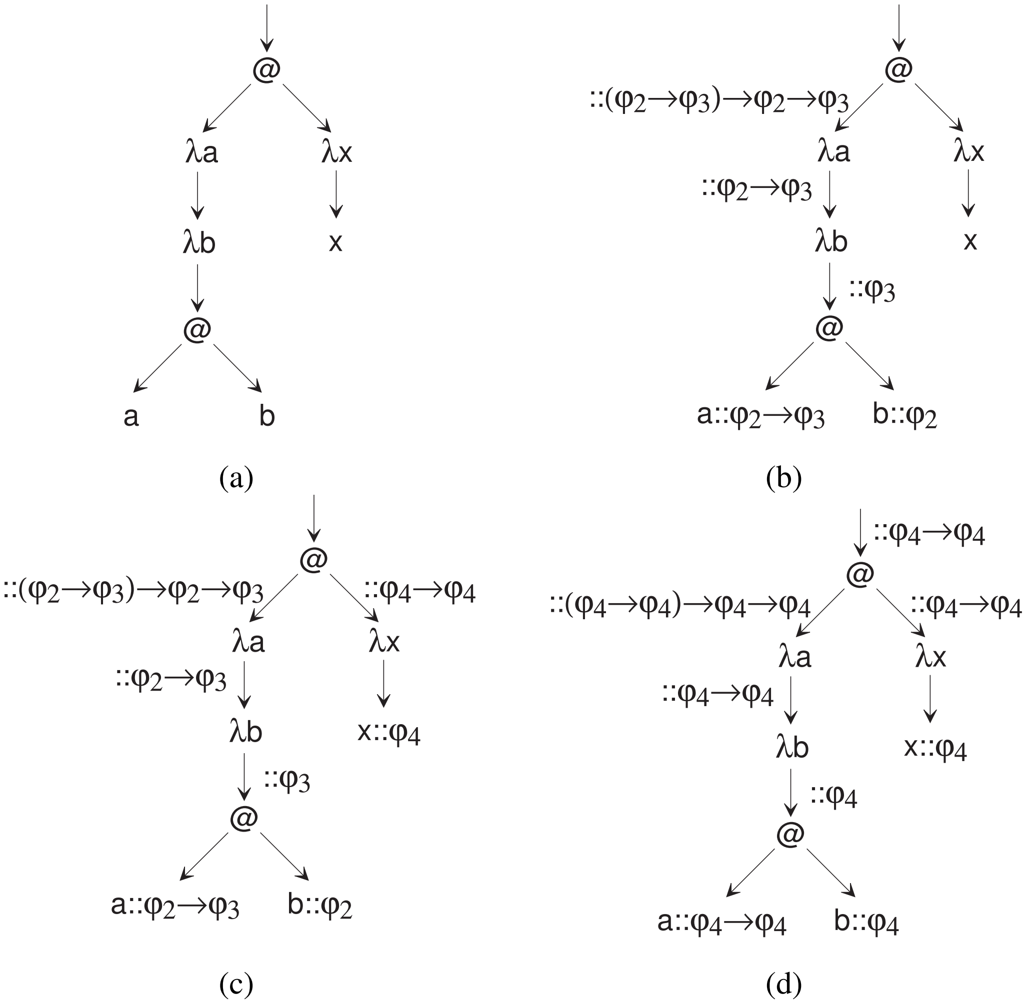

Figure 6.4 (a) Initial tree, (b) typed left subtree, (c) right subtree, (d) conclusion.

Example 2: λa.λb.a. In the initial tree (Figure 6.5(a)) it is assumed that a has type φ_0. For the first abstraction node, there is no assumption made on b yet. This means that any type can be taken for b, because the typed tree puts no limits on the type for b. The result is shown in Figure 6.5(b).

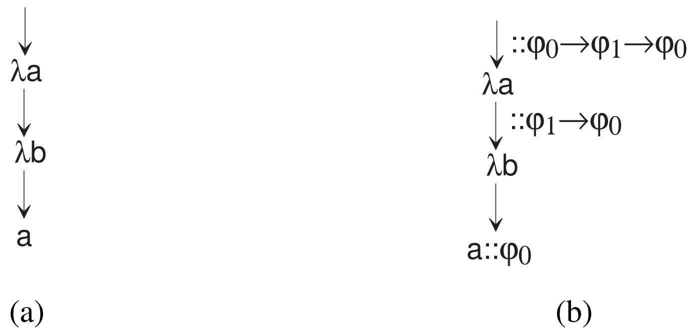

Figure 6.5 (a) Initial tree, (b) final typed tree.

A Curry-type assignment algorithm

The function curry takes a list of fresh type variables and a λ-term and produces as its result a triple consisting of the unused fresh type variables, the required basis and the assigned type.

```
λ-term          ::=  Var var  |  Abstr var λ-term  |  Appl λ-term λ-term
var             ==   char
assumption      ==   (var, type)
basis           ==   [assumption]
freshtypevars   ==   [typevar]
typeresults     ==   (freshtypevars, basis, type)

curry:: freshtypevars -> λ-term -> typeresults
curry (φ : ftvs) (Var x)
   =  (ftvs, [(x, Typevar φ)], Typevar φ)
curry (φ : ftvs) (Appl f g)
   =  (ftvs'', basis, s (Typevar φ))
      where  basis =  substitute s (basleft ++ basright)
             s     =  unify ρ (Arrow σ (Typevar φ))
             (ftvs', basleft, ρ)     =  curry ftvs f
             (ftvs'', basright, σ)   =  curry ftvs' g
curry (φ : ftvs) (Abstr x g)
   =  (ftvs', basis, s (Arrow (Typevar φ) τ))
      where  (basis, s)        =  check_assmptn (x, Typevar φ) bas
             (ftvs', bas, τ)   =  curry ftvs g

substitute:: substitution -> basis -> basis
substitute s [ ]                =  [ ]
substitute s ((x, σ) : rest )   =  (x, s σ) : substitute s rest

check_assmptn:: assumption -> basis -> (basis, substitution)
check_assmptn ass    [ ] =  ([ ], identsub)
check_assmptn (x, σ)  ((y, τ) : rest)
                   =  (substitute (s2 . s1) bas1, s2 . s1),   if y = x
                   =  (substitute s ((y, τ) : bas), s),       otherwise
                      where   s1              =  unify σ τ
                              (bas1, s2)      =  check_assmptn (x, s1 σ) rest
                              (bas, s)        =  check_assmptn (x, σ) rest

identsub:: substitution
identsub    σ     =  σ
```

The function check_assmptn goes through the basis and unifies the type it has already found (initially φ) with the types for the term variable. It returns a basis that equals the original one from which all statements for the term variable are removed, together with the substitution that unifies all types in the statements for the term variable.

6.1.3 Properties of the Curry system

Curry's type assignment system satisfies the **subject reduction property**: if σ is a suitable type for M, and M can be reduced to N, then σ is also a suitable type for N. So a type found for a λ-term will also be a type for any of its reducts (such as the normal form). In a way, this property formalizes the notion that typed programs cannot go wrong.

Curry's system also has the **principal type property** (Hindley, 1969): for every closed λ-term M that is typeable, a type σ_M can be found such that all other types that can be found for M in Curry's system are instances of σ_M and can therefore be obtained by substitution. This type σ_M is called the **principal type** of M.

For open terms there exists a **principal pair**, consisting of the *principal basis* and the *principal type*. All other pairs for this open term can be obtained from this principal pair by substitution. The algorithm presented above returns the principal pair for each typeable λ-term: the last part of its result is the principal type.

> The principal type of λx.x is $\varphi_0 \rightarrow \varphi_0$. All other possible types are instances of this principal type. For example: $(\varphi_2 \rightarrow \varphi_1) \rightarrow \varphi_2 \rightarrow \varphi_1$ is such a type.

The principal type of a term M can be used as a scheme for the construction of the actual type of M when it occurs as a subterm in a specific context: the actual type must be an instance of the principal type. This property forms the basis for polymorphic type systems, as described in the next section.

Principal types are not preserved under reduction: if σ is the principal type for M, and M can be reduced to N, σ does not need to be the principal type for N; it can be that N has a principal type τ that is different from σ. Of course, σ and τ are in such cases related: because of subject reduction, σ is a correct type for the reduct N, and because τ is its principal type, σ is then an instance of τ. The intuition behind this is the following: an application generally contains more information than its reduct. So the type of an application will in general be more specific than the type of its reduct.

> The principal type of (λx.λy.λz.x z (y z)) (λa.λb.a) is $(\varphi_0 \rightarrow \varphi_1) \rightarrow \varphi_0 \rightarrow \varphi_0$, that of λy.λz.z is $\varphi_2 \rightarrow \varphi_3 \rightarrow \varphi_3$. ($\lambda$x.$\lambda$y.$\lambda$z.x z (y z)) ($\lambda$a.$\lambda$b.a) reduces in three steps to λy.λz.z, and $(\varphi_0 \rightarrow \varphi_1) \rightarrow \varphi_0 \rightarrow \varphi_0$ is an instance of the type $\varphi_2 \rightarrow \varphi_3 \rightarrow \varphi_3$. The function in the original application requires its second argument y to be applicable on the third argument z. The reduct does not require this any more.

Drawbacks

Curry's type assignment system is decidable but an important class of terms is not typeable within this system.

The typed subtree $x{::}\varphi_0$ uses two different assumptions for x that cannot be unified. This is easily checked by computing the result of unify $\varphi_1 \rightarrow \varphi_0\ \varphi_1$.

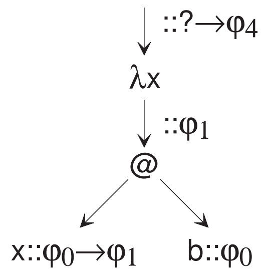

Figure 6.6 λx.x x is not typeable in Curry's type system.

Curry's type assignment system can only type a subclass of the strongly normalizing terms. This subclass does not contain terms in which self-application occurs (Figure 6.6). Hence, recursion cannot be modelled since all fixed combinators contain self-application.

6.2 Polymorphism and recursion

The language Λ^+ presented in this section is an extension of the λ-calculus that enables us to focus on polymorphism and recursion by introducing names for closed λ-terms. Several solutions for finding a type for recursive definitions are discussed.

6.2.1 Syntax of Λ^+

```
Program      = {Definition} Term ;
Definition   = Name '=' Term ;
Term         = Var | Name | Abstraction | Application ;
Abstraction  = '(' 'λ' Var '.' Term ')' ;
Application  = '(' Term Term ')' ;
```

Variables are characters, names are strings starting with a capital. This syntax allows recursive definitions. Redundant brackets will be omitted.

Example of a program written in Λ^+:

```
S = λx.λy.λz.x z (y z)
K = λx.λy.x
I = λx.x

S K I
```

Programs written in Λ^+ can easily be translated into λ-terms. For non-recursive programs the translation consists of replacing, starting with the final term, all names by their bodies. In the case of a recursive definition, the translation to the λ-calculus has to use a fixed point combinator.

6.2.2 Finding a type in Λ^+

Type assignment in Λ^+ is an extension of Curry's type assignment for the λ-calculus. Basically, principal Curry types are assigned to named λ-terms. Furthermore, there is a special way to deal with recursion, as described below.

When trying to find a type for the final term in a program, each definition is typed separately. Its right-hand side is treated as a closed λ-term. So in principle the typed tree representing the principal type of the term is derived as usual.

For every definition, a pair, consisting of the principal type (found for the right-hand side) and the name of the defined function, is put in a list, the **environment**. The environment is represented as a function from names to types. This representation of an environment is possible assuming that all names are (made) different. The environment is used to find types for terms that contain names: each occurrence of a name Name in the program can be regarded as an abbreviation of the right-hand side in its definition, and, therefore, the type associated with the occurrence should be an instance of the principal type of Name in the environment. The different occurrences of Name can have different types, the only relation between their types is that they are all instances of the principal type of Name. The type used for an occurrence of Name does not affect the type of Name in the environment: the type of Name is called **generic**, and the function defined by Name is called a **polymorphic function**.

Every time Name is encountered in a right-hand side, the algorithm looks in the environment to see what its type is. It takes a **fresh instance** of this type, i.e. an instance which is produced by substituting all type variables consistently with fresh type variables. This fresh instance is used to find a type for the right-hand side. In this way, the types actually used for Name in the final typed tree will always be instances of the principal type that is associated with Name in the environment.

For reasons of simplicity, it is assumed that names are defined before they are used. A problem arises when Name does not occur in the environment. This can be the case when the program is recursive. Recursion is dealt with later on.

Consider I = λy.y and the application I I

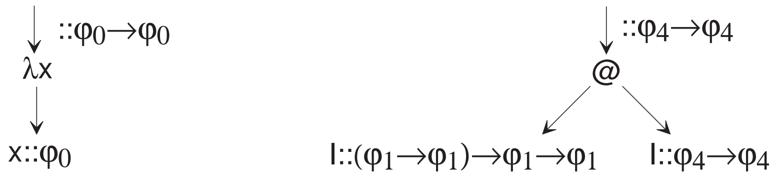

Figure 6.7 Typed trees for I and I I.

The term I I in the program above is typeable by $\varphi_1 \rightarrow \varphi_1$. In finding this type the polymorphism of I is used. The first I in I I is typed by $(\varphi_1 \rightarrow \varphi_1) \rightarrow \varphi_1 \rightarrow \varphi_1$, whereas the second one is typed by $\varphi_1 \rightarrow \varphi_1$. Both types are substitution instances of the principal type found for I, $\varphi_0 \rightarrow \varphi_0$ (see also Figure 6.7).

Solving recursion with a fixed point construction

Also for occurrences of recursively defined names in a right-hand side the types actually used for the occurrences of a name G have to be at least instances of the type found for that name. However, a problem arises when dealing with the occurrences of G in its own body, because when finding the type for G, no type is associated with G in the environment. So in the case of recursive definitions, this condition is difficult to meet. In general, a fixed point construction is necessary to solve the repeated unifications that are involved.

For *mutually recursive* definitions one needs a special kind of fixed point construction that treats all mutually dependent definitions together. For reasons of clarity that kind of recursive definition is *not* considered here. In the rest of this chapter only directly recursive definitions are considered.

A fixed point construction is a general method to find solutions of recursive equations (see also Section 7.3). Starting from an initial approximation of the solution the equations are solved and the solution is taken as a new approximation. If in this approximating process a solution is found that equals the previous approximation, the *fixed point* is reached: the solution of the original equation.

Take for example the following program with a recursive definition:

```
I = λx.x
K = λa.λb.a
G = λv.K (G I) (G K)
```

To find the type for G, a fixed point construction is needed: the first approximation assumes a type variable φ_0 for G. When building the type for the body of G, this type variable will be instantiated to, respectively, $(\varphi_1 \rightarrow \varphi_1) \rightarrow \varphi_2$ and $(\varphi_3 \rightarrow \varphi_4 \rightarrow \varphi_3) \rightarrow \varphi_5$. The type found for the body of G is $\varphi_6 \rightarrow \varphi_2$. The original assumption for G, φ_0, is not an instance of the type found. Apparently this assumed type was too general. It should be replaced by a type that is at least an instance of $\varphi_6 \rightarrow \varphi_2$. So in the fixed point construction a next approximation is made to find the type for G, using as a new assumption the type $\varphi_6 \rightarrow \varphi_2$ found for G in the first try. In this case, this second attempt will give exactly the same typed tree for the body of G (Figure 6.8). So the fixed point is reached.

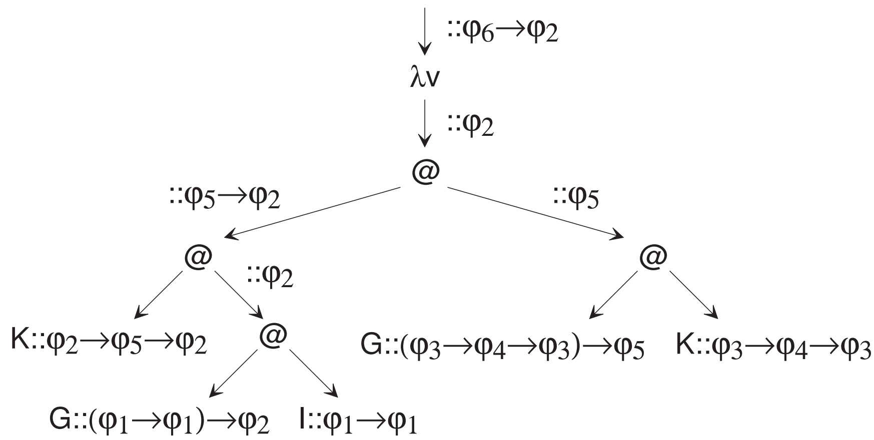

Figure 6.8 Typed tree for G in the example above.

This fixed point construction *always* finds a correct type *if one exists*. Unfortunately, type assignment with the fixed point construction is undecidable due to the possibility of non-termination, as is shown in the next example.

Example of non-termination of the fixed point construction:

F = λx.F

The fixed point construction for F starts with assuming that F is of type φ_0. Then it performs the (λ)-step and finds $\varphi_1 \rightarrow \varphi_0$ for λx.F. Obviously, the type φ_0 is not an instance of $\varphi_1 \rightarrow \varphi_0$. So a second approximation has to be made assuming $\varphi_1 \rightarrow \varphi_0$ for F. The type found is then $\varphi_2 \rightarrow \varphi_1 \rightarrow \varphi_0$ and another approximation has to be made. This will go on and on.

Two different solutions for the problem of recursive definitions are commonly used, both avoiding the fixed point calculation such that both systems become decidable. In fact, they each perform only the first step of the fixed point calculation, each in a different way.

Milner's solution

The first solution is part of Milner's type assignment algorithm defined for the functional programming language ML (Milner, 1978) in which naming of terms is possible via special constructs: the let-construct for non-recursive definitions and the fix-construct for recursive ones.

Instead of calculating the fixed point, Milner's solution simply requires that all types used for a recursively defined name in its definition are one and the same. This seems to be a severe restriction, but with Milner's solution surprisingly many programs can be adequately typed. In Miranda the Milner approach is used when no types are specified by the programmer. In most cases (see Chapter 2) the recursive definitions written down by the programmer can be typed, so the restriction is not a big problem in practice.

It is important to note that Milner's solution is not sufficiently powerful to find a type for G in the example given above. $(\varphi_1 \rightarrow \varphi_1) \rightarrow \varphi_2$ and $(\varphi_3 \rightarrow \varphi_4 \rightarrow \varphi_3) \rightarrow \varphi_5$ are the types actually used for G in its definition. These types cannot be unified.

The intuition behind Milner's solution becomes apparent in the translation of functional programs in Λ^+ to λ-terms. In this translation the fix-construct is translated using a fixed point combinator Y, which is assumed to be typeable by the type $(\varphi_0 \rightarrow \varphi_0) \rightarrow \varphi_0$. Note that in the translation the λ-term λf.(λx.f (x x))(λx.f (x x)) is not used, because it cannot be typed. The algorithm below will assume that occurrences of a recursively defined name in its own definition are represented as Defrecname name.

The Milner algorithm

```
program      = =  ([definition], term)
definition   = =  (name, term)
name         = =  [char]
term         :: = Var var  | Defrecname name     | Name name
                           | Abstr var term      | Appl term term
environment  = =  name -> type

find_type:: freshtypevars -> environment -> program -> type
find_type ftvs env ([ ], t)
=  σ
   where   ((ftvs', bas, σ), s)     =  milner ftvs env t
find_type (φ:ftvs) env (((n, body) : defs), t)
=  find_type ftvs' (addto env n (s1 σ1)) (defs, t),        if occurs_in n body
=  find_type ftvs'' (addto env n σ2) (defs, t),            otherwise
   where   ((ftvs', bas1, σ1), s2)     =  milner ftvs
                                                   (addto env n (Typevar φ)) body
           s1                          =  unify (s2 (Typevar φ)) σ1
           ((ftvs'', bas2, σ2), s3)    =  milner (φ:ftvs) env body

milner:: freshtypevars -> environment -> term -> (typeresults, substitution)
milner (φ:ftvs) env (Var x)
```

```
= ((ftvs, [(x, Typevar φ)], Typevar φ), identsub)
milner ftvs env (Defrecname n)
= ((ftvs, [ ], env n), identsub)
milner ftvs env (Name n)
= ((ftvs', [ ], typeinstance), identsub)
   where   (ftvs', typeinstance)     =  fresh_instance ftvs (env n)
milner (φ:ftvs) env (Appl f g)
= ((ftvs'', bas, s (Typevar φ)), s2 . s1)
   where   bas  =  substitute s (basl ++ basr)
           s    =  unify ρ (Arrow τ (Typevar φ))
           ((ftvs', basl, ρ), s1)  =  milner ftvs env f
           ((ftvs'', basr, τ), s2)  =  milner ftvs' (s1 . env) g
milner (φ:ftvs) env (Abstr x g)
= ((ftvs', resultbas, s2 (Arrow (Typevar φ) τ)), s2 . s1)
   where   (resultbas, s2)        =  check_assmptn (x, Typevar φ) bas
           ((ftvs', bas, τ), s1)   =  milner ftvs env g

addto:: environment -> name -> type -> environment
addto env x σ     =  newenv
                     where   newenv z  =  σ,        if x = z
                                       =  env z,    otherwise
```

The function find_type takes three operands, a list of fresh variables, the environment and the program. An infinite list of fresh variables is assumed to be passed initially as an argument.

As its result find_type produces the type for the final term. When find_type encounters a definition, it checks whether it is a recursive definition (with the function occurs_in, which is left undefined here). If so, it assumes the name is typed by a type variable and it tries to find a type for its body, by calling the function milner. The type variable assumed will very probably be changed by the resulting substitution s_2. The final type should be the same as the type used for the name and therefore the resulting type σ_1 is unified with $s_2\ \varphi$, yielding s_1. If this unification is successful, $s_1\ \sigma_1$ is the type found for the name.

The algorithm milner takes three operands: the fresh variables list, the environment and the term to be typed, and returns the type results found for the term and a substitution. The algorithm milner is almost the same as the algorithm curry. It differs in the way it is capable of dealing with recursive definitions. For recursive definitions, all occurrences of the defined name in the recursive definition (Defrecnames) must have the same type. If a non-recursive name is encountered in a term, the type that is associated with that name in the environment can be instantiated: the function fresh_instance (not defined here) returns a fresh instance of its argument and the remaining part of the list of fresh variables.

If an identifier occurs in its own body, the type in the environment is taken itself instead of being instantiated. This type can be changed in the process of finding a type for the recursive definition: the substitution that is found can also affect the environment. Therefore this substitution is passed as a result of the function milner, and the substitution that is returned by the first recursive call when dealing with an application is also performed on the environment when performing the second recursive call. So all types used for a name in its body will be the same. The substitution is passed as an argument for this purpose only.

Mycroft's solution

The second solution is based on a type assignment system defined by A. Mycroft (Mycroft, 1984). This solution deals with the problem in a way frequently used for many hard problems in computer science: the programmer has to specify the solution; it insists on the type for a recursively defined name being given by the programmer. Instead of calculating the fixed point, this solution simply requires that all types used for a recursively defined name in its definition are instances of the given type. This always terminates, since in looking for a type for the body, the given type is instantiated and no new assumptions are made. Of course, the given type also has to correspond with the type derived for the body. It is even sufficient to require that the given type is an instance of the type derived. This makes it possible that the programmer specifies a type that is more restrictive than the type derived.

Mycroft's solution solves the example:

```
I:: φ1→φ1
I  =  λx.x

K:: φ1→φ2→φ1
K  =  λa.λb.a

G:: φ1→φ2
G  =  λv.K (G I) (G K)
```

Again the types actually used for G are $(\varphi_1 \to \varphi_1) \to \varphi_2$ and $(\varphi_3 \to \varphi_4 \to \varphi_3) \to \varphi_5$. These are both instances of the type given, which is the same as the type found. Therefore Mycroft's solution succeeds.

For any σ and τ, Mycroft's solution rejects the following typed definition:

```
F:: σ→τ
F  =  λx.F
```

The Mycroft algorithm

```
mycroft:: freshtypevars -> environment -> term -> typeresults
mycroft (φ:ftvs) env (Var x)
             = (ftvs, [(x, Typevar φ)], Typevar φ)
mycroft ftvs env (Name n)
             = (ftvs', [ ], typeinstance)
               where  (ftvs', typeinstance)    = fresh_instance ftvs (env n)
mycroft (φ:ftvs) env (Appl f g)
             = (ftvs'', bas, s (Typevar φ))
               where  bas  = substitute s (basl ++ basr)
                      s    = unify ρ (Arrow σ (Typevar φ))
                      (ftvs', basl, ρ)    = mycroft ftvs env f
                      (ftvs'', basr, σ)   = mycroft ftvs' env g
mycroft (φ:ftvs) env (Abstr x g)
             = (ftvs', resultbas, s (Arrow (Typevar φ) τ))
               where  (resultbas, s)  = check_assmptn (x, Typevar φ) bas
                      (ftvs', bas, τ)  = mycroft ftvs env g

check_type:: freshtypevars -> environment -> program -> type
check_type ftvs env ([ ], g)
             = σ
               where  (ftvs', bas, σ)     = mycroft ftvs env g
check_type ftvs env ((n, body):defs, g)
             = check_type ftvs' env (defs, g), if equal_types σ τ
               where  σ                   = env n
                      (ftvs', bas, τ)     = mycroft ftvs env body
```

The algorithm mycroft differs from the algorithm milner in some points. mycroft does not distinguish between different kinds of occurrences of names. All names are dealt with in the same manner (so Defrecname is assumed never to be used). Furthermore, the algorithm assumes that in the environment a type is returned for every name that appears in the program. So if mycroft encounters a name, it takes an instance of the type that is already in the environment. Hence, this algorithm does not affect types in the environment, and the resulting substitution need not be performed on the environment as in milner. Consequently, this substitution is not returned as a part of the result.

The function equal_types (not defined here) checks whether two types are equal.

A mixed approach solution

The functional programming language Miranda uses a combination of Milner's and Mycroft's solutions. If the programmer has defined a type for a recursive definition, the algorithm tries to find it using Mycroft's solution, otherwise it uses Milner's. In Miranda therefore there is a major difference between **type checking**, taking types supplied for definitions by the programmer and checking whether these types are correct, and **type inferencing**, analysing the definitions in a program and trying to find the principal types for them (see also Chapter 2).

The mixed approach algorithm

```
mixed_type:: freshtypevars -> environment -> program -> type
mixed_type ftvs env ([ ], t)
        = σ      where   ((ftvs', bas, σ), s)   =  milner ftvs env t
mixed_type ftvs env (((n, body):b), t)
        = find_type ftvs env (((n, body):b), t),      if env n = Undefined
        = check_type ftvs env (((n, body):b), t),     otherwise
```

In order to be able to test whether in the environment a type is already defined for a name, the definition of type is assumed to be extended with Undefined. As a consequence, functions that are applied on objects of type type have to be extended accordingly.

6.2.3 Properties of type assignment in Λ^+

Both type assignment systems for Λ^+ cover important features of functional programming languages: polymorphism and recursion. They also have the principal type property and satisfy subject reduction.

However, some terms still cannot be typed in the proposed systems. For example, self-applications like λx.x x cannot be typed. Moreover, the type systems cannot deal with definitions that have **internal polymorphism**: i.e. different occurrences of a defined object have different types *inside the definition*.

Take for example the following program:

```
I  =  λy.y
E  =  λc.λd.c d
K  =  λa.λb.a
F  =  λf.λg.λc.λi.f (g c)(g i)

F K I K E
```

While reducing the term F K I K E, I will be applied to two different terms K and E, whose principal types are not unifiable. One might think that, because I is polymorphic and can be used several times with different, maybe even non-unifiable types, the final term will be typeable. This is not true.

The problem is the λ-term that is at the basis of the Milner untypeable function given in Chapter 2. In the tree for the principal type of λf.λg.λc.λi.f (g c) (g i) (and therefore in all possible types for that term) the types for c and i should be the same. This is best illustrated by giving the principal typed tree for the λ-term λc.λi.f (g c) (g i) (Figure 6.9) and performing the (λ)-step to g. Until this step two different assumptions for g, g::$\varphi_0 \rightarrow \varphi_1$ and g::$\varphi_5 \rightarrow \varphi_3$ are used. The (λ)-step to g unifies these types, giving a substitution that replaces φ_0 by φ_5 and φ_1 by φ_3. Therefore it is concluded that the types used for c and i are to be the same (see Figure 6.10). This conclusion is too restrictive. Since the types for c and i are the same, when trying to find a type for the final term F K I K E instances of the types for K and E are unified. This will fail.

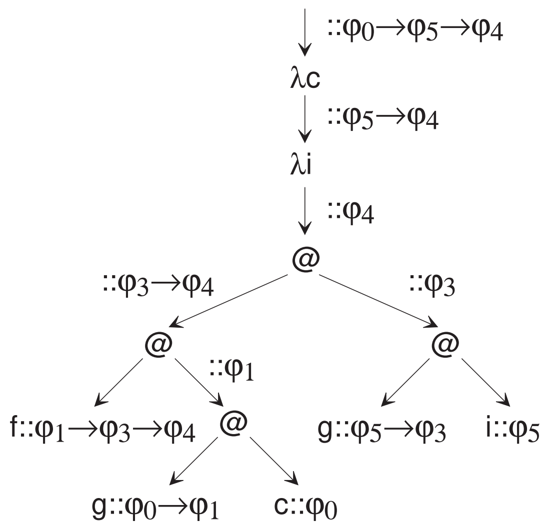

Figure 6.9 Typed tree for λc.λi.f (g c) (g i).

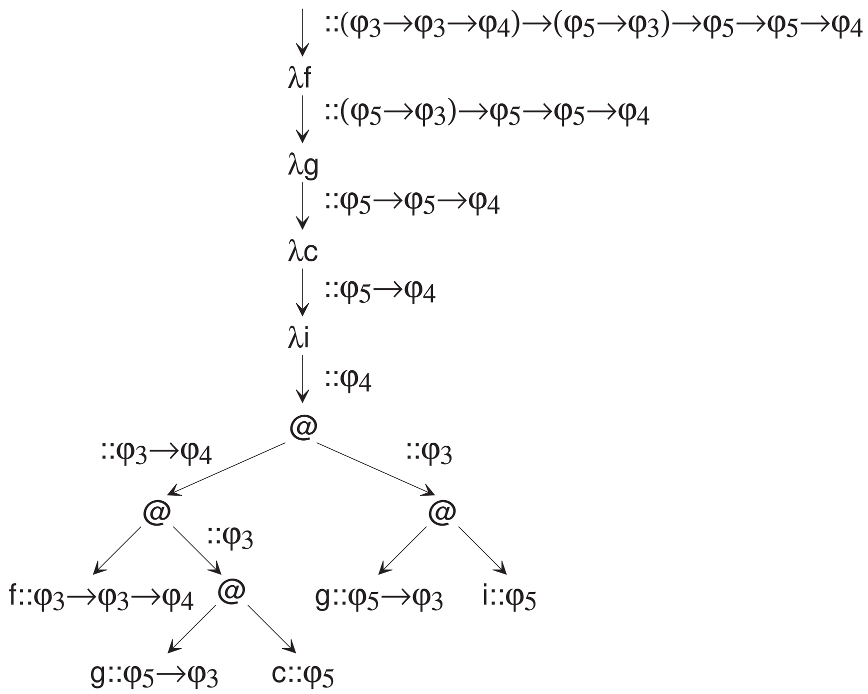

Figure 6.10 Typed tree for λf.λg.λc.λi.f (g c) (g i).

Furthermore, it can happen that applications of functions in which at run-time some arguments are not used, are typed too restrictively since at compile-time their possible use is taken into account.

Consider the following well-known definitions:

$$
\begin{aligned}
&\mathsf{S} :: (\varphi_0 \to \varphi_1 \to \varphi_2) \to (\varphi_0 \to \varphi_1) \to \varphi_0 \to \varphi_2 \\
&\mathsf{S} = \lambda x.\lambda y.\lambda z.x\ z\ (y\ z)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{K} :: \varphi_1 \to \varphi_2 \to \varphi_1 \\
&\mathsf{K} = \lambda a.\lambda b.a
\end{aligned}
$$

$$
\mathsf{S\ K\ S} :: (\varphi_3 \to \varphi_4 \to \varphi_5) \to \varphi_3 \to \varphi_4 \to \varphi_5
$$

Although S K S reduces to I, neither the Milner nor the Mycroft algorithm finds the type $\varphi_0 \to \varphi_0$, due to the imposed restrictions.

There is one final remark to make on the type assignment systems that use Milner's and Mycroft's approaches. In both systems all occurrences for a function symbol are assigned types that are substitution instances of the type provided by the environment for this symbol. One could easily think that since all substitutions are allowed on types, one could replace the type in the environment by any arbitrary substitution instance. For the system that uses Milner's approach, in which of course also types could be provided for definitions, this holds: substitution is a sound operation for terms as well as for definitions.

However, as shown above, in the Mycroft approach recursive definitions can be typed that are not typeable in Milner's. It is not very well known that there are terms, typeable in both systems for which Mycroft's approach allows a more general type than Milner's. Take for example the definitions

$$
\begin{aligned}
&\mathsf{K} :: \varphi_1 \to \varphi_2 \to \varphi_1 \\
&\mathsf{K} = \lambda a.\ \lambda b.\ a
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{R} :: (\varphi_1 \to \varphi_2 \to \varphi_1) \to (\varphi_1 \to \varphi_2 \to \varphi_1) \to \varphi_1 \to \varphi_2 \to \varphi_1 \\
&\mathsf{R} = \lambda x.\ \lambda y.\ \mathsf{R}\ (\mathsf{R}\ y\ \mathsf{K})\ x
\end{aligned}
$$

The types for the definitions (note in particular the type for R) are correct in both Milner's and Mycroft's systems. In Mycroft's system, however, it is possible to take a more general type for R; also the following is correctly typed:

$$
\begin{aligned}
&\mathsf{K} :: \varphi_1 \to \varphi_2 \to \varphi_1 \\
&\mathsf{K} = \lambda a.\ \lambda b.\ a
\end{aligned}
$$

```
R:: φ1→φ2→φ3
R = λx. λy. R (R y K) x
```

This same program can be used to illustrate that in Mycroft's system substitution is no longer a sound operation on definitions. Although the type assignment algorithm is allowed to take any substitution instance of the type $\varphi_1 \to \varphi_2 \to \varphi_3$, not every substitution instance of this type can be put in the environment. For example, using the type $\varphi_1 \to \varphi_1 \to \varphi_1$, the definition for R cannot be typed. The underlying problem here is in fact the same as, for example, in the program:

```
K = λa. λb. a
I = λc. c

I K
```

This program is typeable with the types $\varphi_1 \to \varphi_2 \to \varphi_1$ for K, and $\varphi_3 \to \varphi_3$ for I, but *not* when the type for I is replaced by $(\varphi_4 \to \varphi_4) \to \varphi_4 \to \varphi_4$. The only thing that can be said about this 'problem' is that in Mycroft's system it becomes apparent *within* definitions.

Instead of the very common idea that supplying a type for a definition is equivalent to *asking* if the definition has this type, it is in fact *giving the algorithm the instruction* that all occurrences should have types that are instances of the supplied type. So systems that allow programmers to specify types should not be called type-check systems. The notion *partial type assignment system* is better.

6.3 Type assignment for term rewriting systems

6.3.1 Extending Λ^+ to a functional programming language

It is, from the type assignment point of view, no real problem to extend the language Λ^+ to a real functional programming language. This can be done by, for example, defining the set of types as follows:

(1) all type variables φ_0, φ_1, … are types;
(2) num, real and bool are types;
(3) if σ and τ are types, then $(\sigma \to \tau)$ and $[\sigma]$ are types, where $[\sigma]$ denotes a list of objects of type σ.

The unification algorithm for such types will be almost the same as unify, extended with conditions to solve the problems with the added types. Of course unification of, for example, num and real fails.

When instead of

```
F = λa.G
```

the notation

```
F a = G
```

is used, the operand a can be given a structure, making programs written in the functional programming language look like term rewriting systems. Then the type assignment algorithm would also have to cope with pattern matching by finding the types for the given operands. In this section it will be shown that, if arbitrary patterns are allowed, the subject reduction property is lost.

Another extension is the introduction of **constructors** and **algebraic type definitions**. Constructors are symbols that, when applied to the right arguments, construct an object of a certain type. They are defined by giving an algebraic type definition. It is also customary to introduce constructors that do not take operands. Take, for example:

```
List σ   ==  Nil   |  Cons σ (List σ)
num      ==  1  |  2  |  3  |  . . .
```

The first definition is one for the algebraic type List; this definition defines two constructors, Nil and Cons. The second defines the numerals that have type num. Then Cons 1 (Cons 2 Nil) is an object of type List num.

When assigning types to definitions that use (objects of) these algebraic types, before analysing the definitions the environment must contain types for all the constructors, like

```
Nil    :: List φ
Cons   :: φ → (List φ) → (List φ)
1      :: num
2      :: num
```

In functional programming languages rule alternatives are also introduced, forcing the algorithm to unify the resulting types of alternatives such that a function always yields the same type.

To get programming convenience, the type assignment algorithm also has to deal with definitions used before they are defined (and mutually recursive definitions). This is not difficult to implement.

6.3.2 Type assignment and term rewriting systems

In this section we focus on type assignment in TRSs (see Chapter 4), since in TRSs pattern matching is closer to functional programming languages than in the λ-calculus. Such a type system can easily be transferred to FGRSs (see Chapter 8). Since the proposed type system does not make use of graph information it is best studied in TRSs. The strategy used to assign types to the tree representation of terms will be similar to the previous defined type systems for the language Λ^+. There are, however, some differences, due to the differences between a λ-term and a TRS-term (see Sections 4.1.1 and 4.1.4).

- As for Λ^+, an environment is used to store the types for the constants. However, unlike in Λ^+, there can be more than one rule that defines a constant. In finding a type assignment for the second rule, all constants will already have a type. For convenience, the choice has been made to require that the environment initially provides a type for *every* constant that appears in the TRS.
- As with Mycroft's solution it is required that the type assigned to a defining occurrence of a symbol (see Section 4.1.3) equals its type provided by the environment, while to other occurrences an instance of this type can be assigned.
- In TRSs symbols have fixed arity. Standard Curry types seem to be more appropriate for a variable arity context. So Curry types are extended with a Cartesian product (denoted as x) which is used for the types of the arguments of a function. This means that the definitions of type, unify, sub and not_in of Section 6.1.2 have to be extended accordingly (this is left to the reader).
- For type assignment on rewrite rules the left- and right-hand sides are treated as terms. A rewrite rule Lhs $\rightarrow$ Rhs is correctly typed if Lhs and Rhs are typed with the same type, and all the types assigned to one term variable (that appears in the left- and right-hand sides' bases) are one and the same.
- For type assignment on TRS-terms there is only one constraint, the **constant constraint** (C). If F is a constant symbol, and F A_1 ... A_n is a TRS-term, then there is a substitution S, and types σ_1, ..., σ_n and σ, such that S applied to the type for F provided by the environment is $(\sigma_1 \times ... \times \sigma_n) \rightarrow \sigma$. In the tree representation, typing is performed accordingly (see Figure 6.11).

(C)

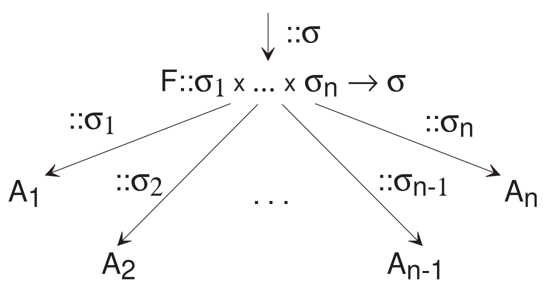

Figure 6.11 The constant constraint.

This notion of type assignment has the principal type property.

Example: stack handling. This example deals with stacks of natural numbers and contains, next to the standard stack functions Top and Pop, the function Alternate that combines two stacks. The types that are assigned are similar to the types that are assigned in the approach using sorts, as in Dershowitz and Jouannaud (1990). The main difference between the two typing approaches is that the approach presented here also allows higher order and polymorphic types. Stacks of natural numbers are represented as terms of the form

Push S_1 (Push S_2 . . .(Push S_n Empty). . .)),

where Empty is the empty stack, and the S_i denote representations of the natural numbers 0, Succ (0), Succ (Succ (0)), etc. according to the following algebraic type definitions:

```
Nat     = =  Zero   |  Succ (Nat)
Stack   = =  Empty  |  Push (Nat, Stack)
```

Push and Empty are stack constructors, and Zero and Succ are number constructors. The functions Top, Pop and Alternate are defined by:

```
Top:: Stack → Nat
Top (Push x y)   →   x

Pop:: Stack → Stack
Pop (Push x y)   →   y

Alternate:: (Stack x Stack) → Stack
Alternate Empty z        →   z
Alternate (Push x y) z   →   Push x (Alternate z y)
```

With these rules it can be shown, for example, that the term Alternate (Push (Top (Push Zero Empty)) Empty) (Pop (Push (Succ Zero) Empty)) rewrites to: Push Zero Empty.

In term rewriting systems function symbols are also allowed on other positions in left-hand sides of rules than just the left-most one (note that this is not allowed in Miranda, see also Section 8.2.3). So it is possible to define optimizations like:

```
Push x (Pop (Push z y))                  →   Push x y
Push (Top (Push x z)) y                  →   Push x y
Push (Top (Push x w)) (Pop (Push z y))   →   Push x y
```

With these rewrite rules for Push the TRS can still be typed correctly.

Currying

In term rewriting systems currying is modelled via an extra symbol for explicit curried application: Ap. Furthermore, for each symbol F a curried variant F_0 is used as an argument of such explicit applications. Finally, for each curried symbol a rewrite rule is added that transforms the curried application into an uncurried one. With respect to typing such (partly) curried systems some special actions have to be taken:

- For the symbol Ap in the environment the higher order type $((\varphi_1 \to \varphi_2) \times \varphi_1) \to \varphi_2$ has to be added. When typing occurrences of this symbol the standard constant-constraint is applicable.
- For each pair of related symbols F and F_0 the types in the environment have to be equivalent with respect to currying, as explained in Section 1.7.2.
- The added rules for transforming curried applications into uncurried applications need to be typed as defining rules for the curried symbols (and not as alternative defining rules for Ap).

Consequently, the following rules for combinator application are typeable, for example, with the given environment:

$$
\begin{array}{lll}
I & :: & \varphi_1 \to \varphi_1 \\
K & :: & (\varphi_1 \times \varphi_2) \to \varphi_1 \\
S & :: & ((\varphi_1 \to \varphi_2 \to \varphi_3) \times (\varphi_1 \to \varphi_2) \times \varphi_1) \to \varphi_3 \\
Ap & :: & ((\varphi_1 \to \varphi_2) \times \varphi_1) \to \varphi_2 \\
I_0 & :: & \varphi_1 \to \varphi_1 \\
K_0 & :: & \varphi_1 \to \varphi_2 \to \varphi_1 \\
S_0 & :: & (\varphi_1 \to \varphi_2 \to \varphi_3) \to (\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_3
\end{array}
$$

$$
\begin{array}{lll}
\text{I x} & \to & \text{x} \\
\text{K x y} & \to & \text{x} \\
\text{S x y z} & \to & \text{Ap (Ap x z) (Ap y z)}
\end{array}
$$

$$
\begin{array}{lll}
\text{Ap } I_0 \text{ x} & \to & \text{I x} \\
\text{Ap (Ap } K_0 \text{ x) y} & \to & \text{K x y} \\
\text{Ap (Ap (Ap } S_0 \text{ x) y) z} & \to & \text{S x y z}
\end{array}
$$

Subject reduction

In general, when assigning types to TRSs as defined above, the subject reduction property is lost.

Take the following rules that are well typed with the given environment:

I	::	$\varphi_1 \to \varphi_1$
K	::	$(\varphi_1 \times \varphi_2) \to \varphi_1$
S	::	$((\varphi_1 \to \varphi_2 \to \varphi_3) \times (\varphi_1 \to \varphi_2) \times \varphi_1) \to \varphi_3$
M	::	$(((\varphi_1 \to \varphi_2) \to \varphi_3) \times (\varphi_1 \to \varphi_2)) \to \varphi_2$
Ap	::	$((\varphi_1 \to \varphi_2) \times \varphi_1) \to \varphi_2$
I_0	::	$\varphi_1 \to \varphi_1$
K_0	::	$\varphi_1 \to \varphi_2 \to \varphi_1$
S_0	::	$(\varphi_1 \to \varphi_2 \to \varphi_3) \to (\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_3$
M_0	::	$((\varphi_1 \to \varphi_2) \to \varphi_3) \to (\varphi_1 \to \varphi_2) \to \varphi_2$

I x	$\to$	x
K x y	$\to$	x
S x y z	$\to$	Ap (Ap x z) (Ap y z)
M (Ap (Ap S_0 x) y)	$\to$	Ap (Ap S_0 I_0) y

Ap I_0 x	$\to$	I x
Ap (Ap K_0 x) y	$\to$	K x y
Ap (Ap (Ap S_0 x) y) z	$\to$	S x y z
Ap M_0 x	$\to$	M x

Then, the term M (Ap (Ap S_0 K_0) I_0) is typeable with the type $(\varphi_4 \to \varphi_5) \to \varphi_5$. A rewrite rule matches this term, so it can be rewritten to Ap (Ap S_0 I_0) I_0. This last term, however, is not typeable with the type $(\varphi_4 \to \varphi_5) \to \varphi_5$. In fact, it is not typeable at all.

It is possible to filter out the rewrite rules that do not satisfy the subject reduction property. Rules are called **safe** if they satisfy the following condition which is needed and sufficient for the required property (see Bakel *et al.*, 1992):

- If (basis, σ) is the principal pair for the left-hand side term, then it is possible to type the right-hand side with σ such that all term variables are typed with the same type as in basis, so if (basis, σ) is the principal pair for the left-hand side term, then (basis, σ) is a pair for the right-hand side.

The Miranda type system satisfies this safety criterion due to the fact that in Miranda in left-hand sides functions can appear as left most symbols only, in combination with the way algebraic type definitions are specified.

6.3.3 The type check algorithm

The goal of the type check algorithm presented below is to determine whether a safe type assignment can be constructed such that all the conditions on type assignment of term rewrite systems are satisfied. The main function of the algorithm, type_rules, has a set of rules, as well as an environment, as parameters. It returns a boolean that indicates whether the construction of the type assignment to all rules was successful and safe. The algorithm type_term returns the principal pair for every term.

```
trs      = =  ([rule] , trsterm)
rule     = =  (trsterm, trsterm)
trsterm  :: = Var var  |  Const const [trsterm]  |  DefSymb const [trsterm]
const    = =  name
```

The last alternative for trsterm (defining the constructor DefSymb) is used to indicate that the corresponding (sub)term starts with the defined symbol. So using DefSymb only makes sense when specifying the left-hand side of a rewrite rule. Of course, it would have been very possible to determine the defined symbol of a certain left-hand side of a rewrite rule without such a construct. This, however, would have made the algorithm somewhat more complicated and less clear.

The type of a constant is either an instance of the type for that constant given by the environment (in the case of a Const) or that type itself (in the case of a DefSymb). The distinction between the two is determined by the function type_term.

The defining node of a rewrite rule can only be typed with *one* type. So any substitution resulting from a unification is forbidden to change this type. This is solved by using as the unification algorithm not unify but unify_types, an extension of Robinson's unification algorithm which is not defined here; it is capable of deciding whether or not a type variable is special (so it appears in some environment type) and it refuses to substitute this variable by other types. This extension can be encoded easily by taking corresponding negative numbers as representation for such variables. The function unify_types must be used throughout instead of unify also where unify is applied in auxiliary functions such as check_assmptn.

```
type_trs:: freshtypevars -> environment -> trs -> typeresults
type_trs ftvs env (rules, term)
                 =  type_term ftvs env term,   if type_rules ftvs env rules

type_rules:: freshtypevars -> environment -> [rule] -> bool
type_rules ftvs env rules  =  foldr (&) True (map (type_rule ftvs env) rules)
```

```
type_rule:: freshtypevars -> environment -> rule -> bool
type_rule ftvs env (lhs, rhs)
= equal_bases (substitute (s2 . s1) basl) basl
  where s2  = unify_basis (substitute s1 basl ++ substitute s1 basr)
        s1  = unify_types σ τ
        (ftvs', basl, σ)   = type_term ftvs env lhs
        (ftvs'', basr, τ)  = type_term ftvs' env rhs

type_term:: freshtypevars -> environment -> trsterm -> typeresults
type_term (φ:ftvs) env (Var var)
        = (ftvs, [(var, Typevar φ)], Typevar φ)
type_term ftvs env (DefSymb const args)
        = type_func_app ftvs env (env const) args
type_term ftvs env (Const const args)
        = type_func_app ftvs' env typeinstance args
          where (ftvs', typeinstance) = fresh_instance ftvs (env const)

type_func_app:: freshtypevars -> environment -> type -> [trsterm]->typeresults
type_func_app ftvs env σ [ ]    = (ftvs, [ ], σ)
type_func_app (φ:ftvs) env σ terms
        = (ftvs', substitute s bas, s (Typevar φ))
          where s              = unify_types (Arrow τ (Typevar φ)) σ
                (ftvs', bas, τ) = type_args ftvs env terms

type_args:: freshtypevars -> environment -> [trsterm] -> typeresults
type_args ftvs env [ t ]      = type_term ftvs env t
type_args ftvs env (t : ts)   = (ftvs'', bas ++ rbas, CartProd σ1 σ2)
                                where (ftvs', bas, σ1)   = type_term ftvs env t
                                      (ftvs'', rbas, σ2) = type_args ftvs' env ts

unify_basis:: basis -> substitution
unify_basis [ ]             = identsub
unify_basis ((x, σ): rest)  = unify_basis bas . s
                              where (bas, s) = check_assmptn (x, σ) rest
```

The algorithm type_rule returns TRUE only for rewrite systems that are safely typeable. For every term variable occurring in a rewrite rule, all the types assigned to this variable in the bases for both the left- and the right-hand sides should be the same. This is provided in type_rule by unifying all those types after a rule has been type checked. These unifications are performed by the function unify_basis. The function type_rule also takes care of checking the safety constraint, by checking, using equal_bases, if the unification of left- and right-hand sides of a rewrite rule has changed the left-hand side basis. This last function is not specified, because its definition is rather tedious while its intention is clear.

The function type_func_app looks for the type of a function application. It calls type_args, which is an auxiliary function that derives the types for all arguments of a node and constructs an arrow type out of these types. type_args assumes that the definition of type is extended with a binary constructor CartProd denoting a Cartesian product. Of course, such an extension also requires an adjustment of several other function definitions used in this chapter.

Summary

- Type assignment systems are used to perform a compile-time check on the type consistency of a program. Most type assignment algorithms for functional programming languages are based upon the *Curry system* for the λ-calculus.
- Type assignment in the Curry system takes place just by looking at the *syntactic structure* of a λ-term: there is one type assignment constraint for a *λ-application* (the arrow elimination) and one for a *λ-abstraction* (the arrow introduction). All instances of the same variable are assumed to have the same type.
- Type assignment in the Curry system is *decidable* and the principal type for a typeable term is found.
- *Self-application* cannot be typed with the consequence that the Curry system cannot be used to type recursive functions.
- By *naming* λ-terms *recursive* definitions can be introduced. Functions can be *polymorphic*.
- With a *fixed point construction* it is possible to find a type for *any (recursive) function definition*, if one exists. But whether there exists a type or not is *undecidable*.
- Milner's solution is a decidable algorithm that finds a type for *many* recursive definitions. It requires that all types used for a recursive definition can be *unified*. All instances of the *same variable* have the *same type*.
- Mycroft's solution checks on consistency of *given* types and is also decidable. It can approve more types than can be derived with the Milner algorithm.
- Neither Milner's nor Mycroft's solution can type *self-application* nor can they handle *internal polymorphism*.
- Type assignment in term rewriting systems is a *natural extension* of type assignment in the λ-calculus. But not all properties of type assignment for the λ-calculus are inherited by term rewriting systems. *Subject reduction* only holds for *safe* rewrite rules.

EXERCISES

6.1 Give a formal derivation for the type of:

(a) λab.a
(b) λxyz. x z (y z)
(c) (λxyz. x z (y z)) (λab.a)
(d) (λxy.x y) (λa.a) (λa.a)

6.2 Show that the term (λz.(λxy.x y) z z) (λa.a) cannot be typed.

6.3 Derive the types for the Push rules of the TRS in Section 6.3.2.