Science of Computer Programming 8 (1987) 147-172
North-Holland

BASIC POLYMORPHIC TYPECHECKING

Luca CARDELLI*
AT&T Bell Laboratories, Murray Hill, NJ 07974, U.S.A.

Communicated by R. Sethi
Received March 1985
Revised June 1986

1. Introduction

Polymorphic means to have *many forms*. As related to programming languages, it refers to data or programs which have many types, or which operate on many types. There are several arbitrary ways in which programs can have many types; we are mostly interested in a particularly orderly form of polymorphism called *parametric polymorphism*. This is a property of programs which are parametric with respect to the *type* of some of their identifiers. There are two major ways of achieving parametric polymorphism which are conceptually related but pragmatically very different: explicit and implicit polymorphism.

Parametric polymorphism is called *explicit* when parametrization is obtained by explicit type parameters in procedure headings, and corresponding explicit applications of type arguments when procedures are called. In this case, parametric polymorphism reduces to the notion of having parameters of type *type* (without necessarily adopting the notion that *type* has itself type *type*). Here is a definition of the polymorphic identity with explicit type parameters (where *fun* stands for λ-abstraction) and its application to an integer and a boolean

let *id* = **fun**(*t*: *type*)**fun**(*a*: *t*)*a*
id(*int*)(3)
id(*bool*)(*true*)

Parametric polymorphism is called *implicit* when the above type parameters and type applications are not admitted, but types can contain *type variables* which are unknown, yet to be determined, types. If a procedure parameter has a type variable or a term containing type variables as its type, then that procedure can be applied to arguments of many different types. Here is the implicit version of the polymorphic

* Current address: DEC SRC, 130 Lytton Ave, Palo Alto, CA 94301, U.S.A.

identity (where α is a type variable), and its application to an integer and a boolean.

let *id* = **fun**(a: α)a
id(3)
id(*true*)

There is considerable interplay, both theoretical and practical, between explicit and implicit polymorphism, and understanding either one can help understanding the other. This paper is largely concerned with implicit polymorphism; it is nonetheless important to dedicate the rest of this section to the relationships between the two kinds of parametric polymorphism.

Implicit polymorphism can be considered as an abbreviated form of explicit polymorphism, where the type parameters and applications have been omitted and must be rediscovered by the language processor. Omitting type parameters leaves some type-denoting identifiers unbound; and these are precisely the type variables. Omitting type arguments requires *type inference* to recover the lost information.

In fact, in implicit polymorphism one can totally omit type information by interpreting the resulting programs as having type variables associated to parameters and identifiers. The programs then appear to be type-free, but rigorous type-checking can still be performed. This is one of the most appealing properties of implicit polymorphism, which makes it particularly appealing for interactive systems and for naïve users. Here is the type-free definition of the polymorphic identity, where all the type information has been omitted.

let *id* = **fun**(a)a

Explicit polymorphism is more expressive, in that it can type programs which cannot be typed by implicit polymorphism, but it is more verbose. In practice, even in explicit polymorphism one may want to omit some type information, and this creates a grey region between fully explicit and fully implicit polymorphism. In this grey region, the type-inference techniques used for implicit polymorphism can be useful, and this is a good reason for studying implicit polymorphism even in the context of explicit polymorphism. For example, a reasonable compromise could be to adopt explicit-type function declarations, but then use implicit-style function applications, using inference to recover the missing information:

let *id* = **fun**(t: *type*)**fun**(a: t)a
id(3)
id(*true*)

Implicit polymorphism can be understood in its own right, both at the semantic and type-inference levels. But it is, in a sense, ambiguous: the same implicitly polymorphic program may correspond to different explicitly polymorphic programs. This ambiguity can be critical in some extensions of the basic type system, noticeably in presence of side-effects. Although no critical ambiguities will arise in the context of this paper, an understanding of the relations between implicit and explicit polymorphism may be necessary when extending implicit polymorphic systems in certain ways.

2. A bit of history

Polymorphic typing of programs was envisioned by Strachey; his lecture notes on fundamental concepts in programming languages [16] already contains much of the notation and terminology used today.

Polymorphic types were already known as *type schemas* in combinatory logic [5]. Extending Curry's work, and collaborating with him, Hindley introduced the idea of a *principal type schema*, which is the most general polymorphic type of an expression, and showed that if a combinatorial term has a type, then it has a principal type [9]. In doing so, he used a result by Robinson about the existence of most general unifiers in the unification algorithm [14]. These results contained all the germs of polymorphic typechecking, including the basic algorithms. The existence of principal types means that a type inference algorithm will always compute a unique 'best' type for a program; moreover, unification can be used to perform this computation. However, these results did not immediately influence the programming language community, because of their theoretical setting.

Influenced by Strachey, and independently from Hindley, Milner rediscovered many of these ideas in the context of the LCF proof generation system [8], which included the first version of the ML language [13]. He introduced a crucial extension to Hindley's work: the notion of generic and non-generic type variables, which is essential for handling declarations of polymorphic functions. Milner implemented the first practical polymorphic typechecker, and showed that the type system is sound [12]. More recently, Milner and Damas proved the principal-type property for the extended system [6] which implies that the type system is decidable. With minor refinements, this is the state of the art exposed in the rest of this paper. This style of polymorphic typechecking was soon adopted by Hope [2], and more recently has been incorporated in other functional languages.

During that initial development of ML, it was found that the introduction of side-effects made the type system unsafe [8, p. 52]. This was resolved in a rather ad-hoc way; the situation was later improved by Damas, but the smooth merging of side-effects and implicit polymorphic typechecking should still be considered an open problem.

Much theoretical work has followed. Coppo showed how to define a coherent type system which is more flexible than ML's [4], although the type system is undecidable. The *ideal* model of types [10] is the model which more directly embodies the idea of implicit polymorphic types, and has its roots in Scott, and in Milner's original paper.

Explicit polymorphism has also its own story, see [1] for an extensive treatment and references, and [3] for examples. The relations between implicit and explicit polymorphism are actively being investigated, see for example [11].

3. Pragmatic motivation

Parametric polymorphic type systems share with Algol 68 properties of compile-time checking, static typing and treatment of higher-order functions, but are more flexible in their ability to define functions which work uniformly on arguments of many types.

Polymorphism in languages comes from the interaction of two contrasting programming language design goals: static typing and reusability. Static typing is the ability to determine the absence of certain classes of run-time faults by static inspection of a program. Static typing is firmly established as a fundamental tool in building large, highly structured and reliable software systems. Reusability is the ability to write routines for an open-ended collection of applications; in particular, we may want to write routines which can be reused when new data types are defined. Reusability is also an important aid in building large programs, as it helps in defining abstractions and leads to better system structuring.

These design goals are in contrast as static typing tends to prevent reusability, and reusable programs are not easy to check statically. A Pascal routine to sort integers cannot be generalized to sort strings and other ordered sets, as the Pascal type system will not allow to parametrize on the type of ordered sets. On the other hand, a Lisp routine to sort integers can be reused on many different kinds of ordered sets, but can also be misused on just about any data structure, with unpredictable results.

Polymorphic type systems try to reconcile these two goals by providing all the safety of statically typed languages, and most (but not all) the flexibility of untyped languages. In this paper we discuss Milner's polymorphic typechecking algorithm, which has proved very successful: it is sound, efficient, and supports a very rich and flexible type system.

Great progress has been made recently in polymorphic languages, but one feature remains unique to Milner's algorithm: its ability to infer types in the absence of type declarations. This feature comes for free. In the attempt to deal with programs which can be reused on many types, the algorithm searches for the *best* (most abstract) type of a program. Such best type is independent of type declarations, which can only be used to reduce the generality of the most abstract type.

This property makes Milner's algorithm particularly suitable for interactive language (ML itself is an interactive compiled language). Interactive users rarely have to bother writing down type information, which is automatically inferred and checked. This strongly contributes to ML's feel of care-free, quick-turnaround language, which is wrongly associated only with interpretive, untyped languages.

The pragmatics of polymorphic typechecking has so far been restricted to a small group of people. The only published description of the algorithm is the one in [12] which is rather technical, and mostly oriented towards the theoretical background. In the hope of making the algorithm accessible to a larger group of people, we present an implementation (in the form of a Modula-2 program) which is very close to the one used in LCF, Hope and ML [8, 2, 13]. Although clarity has sometimes been preferred to efficiency, this implementation is reasonably efficient and quite usable in practice for typechecking large programs.

Only the basic cases of typechecking are considered, and many extensions to common programming language constructs are fairly obvious. The major non-trivial extensions which are known so far (and not discussed here) concern overloading, abstract data types, exception handling, updatable data, and labeled record and union types. Many other extensions are being studied.

We present two views of typing, as a system of type equations and as a type inference system, and attempt to relate them informally to the implementation.

4. A simple applicative language

We do not deal here with ML directly, which is a full-size programming language; instead we considered a simple typed λ-calculus with constants, constituting what can be considered the kernel of the ML language. (The evaluation mechanism (call-by-name or call-by-value) is immaterial for the purpose of typechecking.)

The concrete syntax of expressions is given below, where *Ide* are identifiers, *Exp* are expressions, *Decl* are declarations and **fun** stands for λ. All identifiers declared in the same *Decl* must be distinct. The corresponding abstract syntax is given by the types Exp and Decl in the program in Appendix A (parsers and printers are not provided).

Exp ::=
    Ide |
    "**if**" *Exp* "**then**" *Exp* "**else**" *Exp* |
    "**fun**" "(" *Ide* ")" *Exp* |
    "**let**" *Decl* "**in**" *Exp* |
    "(" *Exp* ")"

Decl ::=
    Ide "=" *Exp* |
    Decl "**then**" *Decl* |
    "**rec**" *Decl* |
    "(" *Decl* ")"

Data types can be introduced into the language simply by having a predefined set of identifiers in the initial environment; this way there is no need to change the syntax or, more importantly, the typechecking program when extending the language.

As an example, the following program defines the factorial function and applies it to zero, assuming that the initial environment contains integer constants and operations:

```
let rec factorial =
  fun(n)
    if zero(n)
    then succ(0)
    else times(n)(factorial(pred(n)))
in factorial(0)
```

5. Types

A type can be either a type variable α, β, etc., standing for an arbitrary type, or a type operator. Operators like *int* (integer type) and *bool* (boolean type) are nullary type operators. Parametric type operators like $\rightarrow$ (function type) or $\times$ (cartesian product type) take one or more types as arguments. The most general forms of the above operators are $\alpha \rightarrow \beta$ (the type of any function) and $\alpha \times \beta$ (the type of any pair of values); α and β can be replaced by arbitrary types to give more specialized function and pair types. Types containing type variables are called *polymorphic*, while types not containing type variables are *monomorphic*. All the types found in conventional programming languages, like Pascal, Algol 68 etc. are monomorphic.

Expressions containing several occurrences of the same type variable, like in $\alpha \rightarrow \alpha$, express contextual dependencies, in this case between the domain and the codomain of a function type. The typechecking process consists in matching type operators and instantiating type variables. Whenever an occurrence of a type variable is instantiated, all the other occurrences of the same variable must be instantiated to the same value: legal instantiations of $\alpha \rightarrow \alpha$ are $int \rightarrow int$, $bool \rightarrow bool$, $(\beta \times \gamma) \rightarrow (\beta \times \gamma)$, etc. This contextual instantiation process is performed by *unification* [14] and is at the basis of polymorphic typechecking. Unification fails when trying to match two different type operators (like *int* and *bool*) or when trying to instantiate a variable to a term containing that variable (like α and $\alpha \rightarrow \beta$, where a circular structure would be built). The latter situation arises in typechecking self-application (e.g. **fun**(x) $x(x)$), which is therefore considered illegal.

Here is a trivial example of typechecking. The identity function $Id =$ **fun**(x) x has type $\alpha \rightarrow \alpha$ because it maps any type onto itself. In the expression $Id(0)$ the type of 0 (i.e. *int*) is matched to the domain of the type of *Id*, yielding $int \rightarrow int$, as the specialized type of *Id* in that context. Hence the type of $Id(0)$ is the codomain of the type of *Id*, which is *int* in this context.

In general, the type of an expression is determined by a set of type combination rules for the language constructs, and by the types of the primitive operators. The initial type environment could contain the following primitives for booleans, integers, pairs and lists (where $\rightarrow$ is the function type operator, $\times$ is cartesian product, and *list* is the list operator):

true, *false*	: *bool*	*snd*	: $(\alpha \times \beta) \to \beta$
0, 1, ...	: *int*	*nil*	: α *list*
succ, *pred*	: $int \to int$	*cons*	: $(\alpha \times \alpha\ list) \to \alpha\ list$
zero	: $int \to bool$	*hd*	: $\alpha\ list \to \alpha$
pair	: $\alpha \to (\beta \to (\alpha \times \beta))$	*tl*	: $\alpha\ list \to \alpha\ list$
fst	: $(\alpha \times \beta) \to \alpha$	*null*	: $\alpha\ list \to bool$

The type α *list* is the type of homogeneous list, whose elements all have type α.

6. The type of length

Before describing the typechecking algorithm, let us discuss the type of a simple recursive program which computes the length of a list:

```
let rec length =
  fun(l)
    if null(l)
    then 0
    else succ(length(tl(l)))
in ...
```

The type of *length* is $\alpha\ list \to int$; this is a polymorphic type as *length* can work on lists of any kind. The way we deduce this type can be described in two ways. In principle, typechecking is done by setting up a system of type constraints, and then solving it with respect to the type variables. In practice, typechecking is done by a bottom-up inspection of the program, matching and synthesizing types while proceeding towards the root; the type of an expression is computed from the type of its subexpressions and the type constraints imposed by the context, while the type of the predefined identifiers is already known and contained in the initial environment. It is a deep property of the type system and of the typechecking algorithm that the order in which we examine programs and carry out the matching does not affect the final result and solves the system of type constraints.

The system of type constraints for *length* is:

(1)	*null*	: $\alpha\ list \to bool$
(2)	*tl*	: $\beta\ list \to \beta\ list$
(3)	0	: *int*
(4)	*succ*	: $int \to int$
(5)	*null*(*l*)	: *bool*
(6)	0	: γ
(7)	*succ*(*length*(*tl*(*l*)))	: γ
(8)	**if** *null*(*l*) **then** 0 **else** *succ*(*length*(*tl*(*l*)))	: γ

(9)	*null*	$: \delta \to \varepsilon$
(10)	*l*	$: \delta$
(11)	*null*(*l*)	$: \varepsilon$
(12)	*tl*	$: \phi \to \chi$
(13)	*l*	$: \phi$
(14)	*tl*(*l*)	$: \chi$
(15)	*length*	$: \eta \to \iota$
(16)	*tl*(*l*)	$: \eta$
(17)	*length*(*tl*(*l*))	$: \iota$
(18)	*succ*	$: \kappa \to \lambda$
(19)	*length*(*tl*(*l*))	$: \kappa$
(20)	*succ*(*length*(*tl*(*l*)))	$: \lambda$
(21)	*l*	$: \mu$
(22)	**if** *null*(*l*) **then** 0 **else** *succ*(*length*(*tl*(*l*)))	$: \nu$
(23)	**fun**(*l*) **if** *null*(*l*) **then** 0 **else** *succ*(*length*(*tl*(*l*)))	$: \mu \to \nu$
(24)	*length*	$: \pi$
(25)	**fun**(*l*) **if** *null*(*l*) **then** 0 **else** *succ*(*length*(*tl*(*l*)))	$: \pi$

Lines (1)-(4) express the constraints for the predefined global identifiers, which are already known. The conditional construct imposes ((5)-(8)) that the result of a test must be boolean, and the two branches of the conditional must have the same type γ, which is also the type of the whole conditional expression. The four function applications in this program determine (9)-(20); in each case the function symbol must have a functional type (e.g. $\delta \to \varepsilon$ in (9)); its argument must have the same type as the domain of the function (e.g. δ in (10)), and the result must have the same type as the codomain of the function (e.g. ε in (11)). The **fun** expression (23) has a type $\mu \to \nu$, given that its parameter has type μ (21) and its body has type ν (22). Finally the definition construct imposes that the variable being defined (*length* (24)) has the same type as its definition (25).

Typechecking *length* consists in

(i) verifying that the above system of constraints is consistent (e.g. it does not imply $\mathit{int} = \mathit{bool}$), and

(ii) solving the constraints with respect to π.

The expected type of *length* ($\pi = \beta\ \mathit{list} \to \mathit{int}$) can be inferred as follows:

$$\begin{array}{ll}
\pi = \mu \to \nu & \text{by (25), (23)} \\
\mu = \phi = \beta\ \mathit{list} & \text{by (21), (13), (12), (2)} \\
\nu = \gamma = \mathit{int} & \text{by (22), (8), (6), (3)}
\end{array}$$

Considerably more work is needed to show that β is completely unconstrained, and that the whole system is consistent. The typechecking algorithm described in the next section systematically performs this work, functioning as a simple deterministic theorem prover for systems of type constraints.

Here is a bottom-up derivation of the type of *length* which is closer to what the typechecking algorithm really does; the consistency of the constraints (i.e. the absence of type errors) is also checked in the process:

(26)	l	$: \delta$	by (10)
(27)	$null(l)$	$: bool$	
		$\varepsilon = bool$; $\delta = \alpha\ list$;	by (11), (9), (1)
(28)	0	$: int$	
		$\gamma = int$;	by (6), (3)
(29)	$tl(l)$	$: \beta\ list$;	
		$\phi = \beta\ list$; $\chi = \beta\ list$	by (26), (27), (12)-(14), (2)
		$\beta = \alpha$;	
(30)	$length(tl(l))$	$: \iota$	
		$\eta = \beta\ list$;	by (15)-(17), (29)
(31)	$succ(length(tl(l)))$	$: int$	
		$\iota = \kappa = int$;	by (18)-(20), (4), (30)
(32)	**if** $null(l)$ **then** 0 **else** $succ(length(tl(l)))$	$: int$	by (5)-(8), (27), (28), (31)
(33)	**fun**(l) **if** $null(l)$ **then** 0 **else** $succ(length(tl(l)))$	$: \beta\ list \rightarrow int$	
		$\mu = \beta\ list$; $\nu = int$;	by (21)-(23), (26), (27), (32)
(34)	$length$	$: \beta\ list \rightarrow int$	
		$\pi = \beta\ list \rightarrow int$;	by (24)-(25), (33), (15), (30), (31)

Note that recursion is taken care of: the types of the two instances of *length* in the program (the definition and the recursive function call) are compared in (34).

7. Typechecking

The basic algorithm can be described as follows.

Case 1. When a new variable x is introduced by a **fun** binder, it is assigned a new type variable α meaning that its type must be further determined by the context of its occurrences. The pair $\langle x, \alpha \rangle$ is stored in an environment which is searched every time an occurrence of x is found, yielding α (or any intervening instantiation of it) as the type of that occurrence.

Case 2. In a conditional, the **if** component is matched to *bool*, and the **then** and **else** branches are unified in order to determine a unique type for the whole expression.

Case 3. In an abstraction **fun**(x) e the type of e is inferred in a context where x is associated to a new type variable.

Case 4. In an application $f(a)$, the type of f is unified against a type $A \rightarrow \beta$, where A is the type of a and β is a new type variable. This implies that the type of f must be a function type whose domain is unifiable to A; β (or any instantiation of it) is returned as the type of the whole application.

In order to describe the typechecking of **let** expressions, and of variables introduced by **let** binders, we need to introduce the notion of *generic* type variables.

Consider the following expression:

```
fun(f)pair(f(3))(f(true))        [Ex1]
```

In Milner's type system this expression cannot be typed, and the algorithm described above will produce a type error. In fact, the first occurrence of *f* determines a type $int \to \beta$ for *f*, and the second occurrence determines a type $bool \to \beta$ for *f*, which cannot be unified with the first one.

Type variables appearing in the type of a **fun**-bound identifier like *f* are called *non-generic* because, as in this example, they are shared among all the occurrences of *f* and their instantiations may conflict.

One could try to find a typing for *Ex*1, for example by somehow assigning it $(\alpha \to \beta) \to (\beta \times \beta)$; this would compute correctly in situations like *Ex*1(**fun**(*a*)0) whose result would be *pair*(0)(0). However this typing is unsound in general: for example *succ* has a type that matches $\alpha \to \beta$ and it would be accepted as an argument to *Ex*1 and wrongly applied to *true*. There are sound extensions of Milner's type system which can type *Ex*1, but they are beyond the scope of this discussion.

Hence there is a basic problem in typing heterogeneous occurrences of **fun**-bound identifiers. Forbidding such occurrences turns out to be tolerable in practice because expressions like *Ex*1 are not extremely useful or necessary and because a different mechanism is provided. We are going to try and do better in typing heterogeneous occurrences of let-bound identifiers. Consider:

```
let f = fun(a) a             [Ex2]
in pair(f(3))(f(true))
```

It is essential to be able to type the previous expression, otherwise no polymorphic function could be applied to distinct types in the same context, making polymorphism quite useless. Here we are in a better position than *Ex*1, because we know exactly what *f* is, and we can use this information to deal separately with its occurrences.

In this case *f* has type $\alpha \to \alpha$; type variables which, like α, occur in the type of **let**-bound identifiers (and that moreover do not occur in the type of *enclosing* **fun**-bound identifiers) are called *generic*, and they have the property of being able to assume different values for different instantiations of the **let**-bound identifier. This is achieved operationally by making a copy of the type of *f* for every distinct occurrence of *f*.

In making a copy of a type, however, we must be careful not to make a copy of non-generic variables, which must be shared. The following expression for example is as illegal as *Ex*1, and *g* has a non-generic type which propagates to *f*:

```
fun(g)                         [Ex3]
  let f = g
  in pair(f(3))(f(true))
```

Again, it would be unsound to accept this expression with a type like $(\alpha \to \beta) \to (\beta \times \beta)$ (consider applying *succ* so that it is bound to *g*).

The definition of generic variables is

> A type variable occurring in the type of an expression e is generic (with respect to e) iff it does not occur in the type of the binder of any **fun** expression enclosing e.

Note that a type variable which is found to be non-generic while typechecking within a **fun** expression may become generic outside it. This is the case in *Ex*2 where a is assigned a non-generic α, and f is assigned $\alpha \rightarrow \alpha$ where α is now generic.

To determine when a variable is generic we maintain a list of the non-generic variables at any point in the program: when a type variable is not in the list it is generic. The list is augmented when entering a **fun**; when leaving the **fun** the old list automatically becomes the current one, so that that type variable becomes generic. In copying a type, we must only copy the generic variables, while the non-generic variables must be shared. In unifying a non-generic variable to a term, all the type variables contained in that term become non-generic.

Finally we have to consider recursive declarations:

> **let rec** $f = \ldots f \ldots$ **in** $\ldots f \ldots$

which are treated as if the **rec** were expanded using a fixpoint operator Y (of type $(\alpha \rightarrow \alpha) \rightarrow \alpha$):

> **let** $f = Y($**fun**$(f) \ldots f \ldots)$
> **in** $\ldots f \ldots$

It is now evident that the instances of (the type variables in the type of) f in the recursive definition must be non-generic, while the instances following **in** are generic.

Case 5. Hence, to typecheck a **let** we typecheck its declaration part, obtaining an environment of identifiers and types which is used in the typechecking of the body of the **let**.

Case 6. A declaration is treated by checking all its definitions $x_i = t_i$, each of which introduces a pair $\langle x_i, T_i \rangle$ in the environment, where T_i is the type of t_i. In case of (mutually) recursive declarations $x_i = t_i$ we first create an environment containing pairs $\langle x_i, \alpha_i \rangle$ for all the x_i being defined, and where the α_i are new non-generic type variables (they are inserted in the list of non-generic variables for the scope of the declaration). Then all the t_i are typechecked in that environment, and their types T_i are again matched against the α_i (or their instantiations).

8. A digression on models, inference systems and algorithms

There are two basic approaches to the formal semantics of types. The most fundamental one is concerned with devising mathematical models for types, normally by mapping every type expression into a set of values (the values having that type);

the basic difficulty here is in finding a mathematical meaning for the $\rightarrow$ operator [15, 12, 10].

The other, complementary, approach is to define a formal system of axioms and inference rules, in which it is possible to prove that an expression has some type. The relationship between models and formal systems is very strong. A semantic model is often a guide in defining a formal system, and every formal system should be self-consistent, which is often shown by exhibiting a model for it.

A good formal system is one in which we can prove nearly everything we *know* is true (according to intuition, or because it is true in a model). Once a good formal system has been found, we can *almost* forget the models, and work in the usually simpler, syntactic framework of the system.

Typechecking is more strictly related to formal systems than to models, because of its syntactic nature. A typechecking algorithm, in some sense, implements a formal system, by providing a procedure for proving theorems in that system. The formal system is essentially simpler and more fundamental than any algorithm, so that the simplest presentation of a typechecking algorithm is the formal system it implements. Also, when looking for a typechecking algorithm, it is better to first define a formal system for it.

Not all formal type systems admit typechecking algorithms. If a formal system is too powerful (i.e. if we can prove many things in it), then it is likely to be undecidable, and no decision procedure can be found for it. Typechecking is usually restricted to decidable type systems, for which typechecking algorithms can be found. However in some cases undecidable systems could be treated by incomplete typechecking *heuristics* (this has never been done in practice, so far), which only attempt to prove theorems in that system, but may at some point give up. This could be acceptable in practice because there are limits to the complexity of a program: its meaning could get out of hand long before the limits of the typechecking heuristics are reached.

Even for decidable type systems, all the typechecking algorithms could be exponential, again requiring heuristics to deal with them. This has been successfully attempted in Hope [2] for the treatment of overloading in the presence of polymorphism.

The following section presents an inference system for the kind of polymorphic typechecking we have described. We have now two distinct views of typechecking: one is *solving a system of type equations*, as we have seen in the previous sections, and the other is *proving theorems in a formal system*, as we are going to see now. These views are interchangeable, but the latter one seems to provide more insights because of its connection with type semantics on one side and algorithms on the other.

9. An inference system

In the following inference system, the syntax of types is extended to type quantifiers $\forall\alpha.\tau$. In Milner's type system, all the type variables occurring in a type are intended to be implicitly quantified at the top level. For example, $\alpha \to \beta$ is really $\forall\alpha.\forall\beta.\alpha \to \beta$. However, quantifiers cannot be nested inside type expressions.

A type is called *shallow* if it has the form $\forall\alpha_1. \dots .\forall\alpha_n.\tau$ were $n \geq 0$ and no quantifiers occur in τ. Our inference system allows the construction of non-shallow types: unfortunately we do not have typechecking algorithms able to cope with them. Hence, we are only interested in inferences which involve only shallow types. We have chosen to use type quantifiers in the inference system because this helps explain the behavior of generic/non-generic type variables, which correspond exactly to free/quantified type variables. For a slightly different inference system which avoids non-shallow types, see [6].

Here is the set of inference rules. [IDE] is an axiom scheme, while the other rules are proper inferences. The horizontal bar reads *implies.* An *assumption* $x{:}\tau$ is the association of a variable x with a type τ. If A is a set of assumptions (uniquely mapping variables to types), then $A.x{:}\tau$ is the same as A except that x is associated with τ. If A and B are sets of assumptions, and B is $x_1{:}\tau_1. \dots .x_n{:}\tau_n$, then $A.B$ is the set of assumptions $A.x_1{:}\tau_1. \dots .x_n{:}\tau_n$. The notation $A \vdash e{:}\tau$ means that given a set of assumptions A, we can deduce that the expression e has type τ. The notation $A \vdash d{::}B$ means that given a set of assumptions A, we can deduce that the declaration d (introducing variables $x_1 \dots x_n$) determines a set of assumptions B (of the form $x_1{:}\tau_1. \dots .x_n{:}\tau_n$). Finally, the expression $\tau[\sigma/\alpha]$ is the result of substituting σ for all the free occurrences of α in τ.

[IDE] $\quad A.x{:}\tau \vdash x{:}\tau$

[COND] $$\frac{A \vdash e{:}bool \quad A \vdash e'{:}\tau \quad A \vdash e''{:}\tau}{A \vdash (\textbf{if } e \textbf{ then } e' \textbf{ else } e''){:}\tau}$$

[ABS] $$\frac{A.x{:}\sigma \vdash e{:}\tau}{A \vdash (\textbf{fun}(x)e){:}\sigma \to \tau}$$

[COMB] $$\frac{A \vdash e{:}\sigma \to \tau \quad A \vdash e'{:}\sigma}{A \vdash e(e'){:}\tau}$$

[LET] $$\frac{A \vdash d{::}B \quad A.B \vdash e{:}\tau}{A \vdash (\textbf{let } d \textbf{ in } e){:}\tau}$$

[GEN] $$\frac{A \vdash e{:}\tau}{A \vdash e{:}\forall\alpha.\tau} \quad (\alpha \text{ not free in } A)$$

[SPEC] $$\frac{A \vdash e{:}\forall\alpha.\tau}{A \vdash e{:}\tau[\sigma/\alpha]}$$

[BIND] $$\frac{A \vdash e{:}\tau}{A \vdash (x = e){::}(x{:}\tau)}$$

[THEN] $$\frac{A \vdash d{::}A' \quad A' \vdash d'{::}B}{A \vdash (d \textbf{ then } d'){::}B}$$

$$[\text{REC}] \quad \frac{A.B \vdash d :: B}{A \vdash (\mathbf{rec}\ d) :: B}$$

As a first example, we can deduce the most general type of the identity function: $(\mathbf{fun}(x)\ x) : \forall\alpha.\alpha \to \alpha$

$$\cfrac{\cfrac{x:\alpha \vdash x:\alpha \quad [\text{IDE}]}{\vdash (\mathbf{fun}(x)\ x) : \alpha \to \alpha \quad [\text{ABS}]}}{\vdash (\mathbf{fun}(x)\ x) : \forall\alpha.\alpha \to \alpha \quad [\text{GEN}]}$$

A specialized type for the identity function can be deduced either from the general type:

$$\frac{\vdash (\mathbf{fun}(x)\ x) : \forall\alpha.\alpha \to \alpha}{\vdash (\mathbf{fun}(x)\ x) : \mathit{int} \to \mathit{int}} \quad [\text{SPEC}]$$

or more directly:

$$\frac{x:\mathit{int} \vdash x:\mathit{int} \quad [\text{IDE}]}{\vdash (\mathbf{fun}(x)\ x) : \mathit{int} \to \mathit{int} \quad [\text{ABS}]}$$

We can extend the above inference to show $(\mathbf{fun}(x)\ x)(3) : \mathit{int}$:

$$\frac{\cfrac{3:\mathit{int}, x:\mathit{int} \vdash x:\mathit{int} \quad [\text{IDE}]}{3:\mathit{int} \vdash (\mathbf{fun}(x)\ x) : \mathit{int} \to \mathit{int} \quad [\text{ABS}]} \quad 3:\mathit{int} \vdash 3:\mathit{int} \quad [\text{IDE}]}{3:\mathit{int} \vdash (\mathbf{fun}(x)\ x)(3) : \mathit{int}} \quad [\text{COMB}]$$

Here is an example of a *forbidden* derivation using non-shallow types, which can be used to give a type to $\mathbf{fun}(x)\ x(x)$, which our algorithm is not able to type (here $\phi = \forall\alpha.\alpha \to \alpha$):

$$\cfrac{\cfrac{\cfrac{x:\phi \vdash x:\phi \quad [\text{IDE}]}{x:\phi \vdash x:\phi \to \phi \quad [\text{SPEC]}} \quad x:\phi \vdash x:\phi \quad [\text{IDE}]}{x:\phi \vdash x(x):\phi} \quad [\text{COMB}]}{\vdash (\mathbf{fun}(x)\ x(x)) : \phi \to \phi} \quad [\text{ABS}]$$

Note how $\forall\alpha.\alpha \to \alpha$ gets instantiated to $(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$ by [SPEC], substituting $\forall\alpha.\alpha \to \alpha$ for α.

We want to show now that $(\mathbf{let}\ f = \mathbf{fun}(x)\ x\ \mathbf{in}\ \mathit{pair}(f(3))(f(\mathit{true}))) : \mathit{int} \times \mathit{bool}$. Take $A = \{3: \mathit{int}, \mathit{true}:\mathit{bool}, \mathit{pair}:\forall\alpha.\forall\beta.\alpha \to (\beta \to (\alpha \times \beta))\}$ and $\phi = \forall\alpha.\alpha \to \alpha$.

$$\frac{\cfrac{A.f:\phi \vdash f:\phi \quad [\text{IDE}]}{A.f:\phi \vdash f:\mathit{int} \to \mathit{int} \quad [\text{SPEC}]} \quad A.f:\phi \vdash 3:\mathit{int} \quad [\text{IDE}]}{A.f:\phi \vdash f(3):\mathit{int}} \quad [\text{COMB}]$$

$$\frac{\cfrac{A.f:\phi \vdash f:\phi \quad [\text{IDE}]}{A.f:\phi \vdash f:\mathit{bool} \to \mathit{bool} \quad [\text{SPEC}]} \quad A.f:\phi \vdash \mathit{true}:\mathit{bool} \quad [\text{IDE}]}{A.f:\phi \vdash f(\mathit{true}):\mathit{bool}} \quad [\text{COMB}]$$

$$\cfrac{\cfrac{A.f{:}\phi \vdash \mathit{pair}{:}\forall\alpha.\forall\beta.\alpha\to(\beta\to(\alpha\times\beta))\quad [\mathrm{IDE}]}{A.f{:}\phi \vdash \mathit{pair}{:}\forall\beta.\mathit{int}\to(\beta\to(\mathit{int}\times\beta))}\ [\mathrm{SPEC}]}{A.f{:}\phi \vdash \mathit{pair}{:}\,\mathit{int}\to(\mathit{bool}\to(\mathit{int}\times\mathit{bool}))}\ [\mathrm{SPEC}]$$

$$\cfrac{\cfrac{A.f{:}\phi \vdash \mathit{pair}{:}\,\mathit{int}\to(\mathit{bool}\to(\mathit{int}\times\mathit{bool}))\quad A.f{:}\phi\vdash f(3){:}\,\mathit{int}}{A.f{:}\phi\vdash \mathit{pair}(f(3)){:}\,\mathit{bool}\to(\mathit{int}\times\mathit{bool})}\ [\mathrm{COMB}]\quad A.f{:}\phi\vdash f(\mathit{true}){:}\,\mathit{bool}}{A.f{:}\phi\vdash \mathit{pair}(f(3))(f(\mathit{true})){:}\,\mathit{int}\times\mathit{bool}}\ [\mathrm{COMB}]$$

$$\cfrac{A\vdash(\mathbf{fun}(x)\ x){:}\,\phi\quad A.f{:}\phi\vdash \mathit{pair}(f(3))(f(\mathit{true})){:}\,\mathit{int}\times\mathit{bool}}{A\vdash(\mathbf{let}\ f=\mathbf{fun}(x)\ x\ \mathbf{in}\ \mathit{pair}(f(3))(f(\mathit{true}))){:}\,\mathit{int}\times\mathit{bool}}\ [\mathrm{BIND}][\mathrm{LET}]$$

Note that from the assumption $f{:}\forall\alpha.\alpha\to\alpha$, we can independently instantiate α to *int* and *bool*; i.e., f has a generic type. Instead, in $(\mathbf{fun}(f)\ \mathit{pair}(f(3))(f(\mathit{true})))$ $(\mathbf{fun}(x)\ x)$, which is the function-application version of the above **let** expression, no shallow type can be deduced for $\mathbf{fun}(f)\ \mathit{pair}(f(3))(f(\mathit{true}))$.

A variable is generic if it does not appear in the type of the variables of any enclosing **fun**-binder. Those binders must occur in the set of assumptions, so that they can be later discarded by [ABS] to create those enclosing **fun**'s. Hence a variable is generic if it does not appear in the set of assumptions. Therefore, if a variable is generic, we can apply [GEN] and introduce a quantifier. This determines a precise relation between generic variables and quantifiers.

There is a formal way of relating the above inference system to the typechecking algorithm presented in the previous sections. It can be shown that if the algorithm succeeds in producing a type for an expression, then that type can be deduced from the inference system (see [12] for a result involving a closely related inference system). We are now going to take a different, informal approach to intuitively justify the typechecking algorithm. We are going to show how an algorithm can be extracted from an inference system. In this view a typechecking algorithm is a *proof heuristic*; i.e. it is a strategy to determine the order in which the inference rules should be applied. If the proof heuristic succeeds, we have determined that a type can be inferred. If it fails, however, it may still be possible to infer a type. In particular our heuristic will be unable to cope with expressions which require some non-shallow type manipulation, like in the deduction of

$$(\mathbf{fun}(x)\ x(x))(\mathbf{fun}(x)\ x){:}\ \forall\alpha.\alpha\to\alpha.$$

In fact, the heuristic will simply ignore type quantifier and treat all the type variables as free variables.

There are two aspects to the heuristic. The first one is how to determine the sets of assumptions, and the second is the order in which to apply the inference rules. If a language requires type declarations for all identifiers, it is trivial to obtain the sets of assumptions, otherwise we have to do *type inference.*

In carrying out type inference, **fun**-bound identifiers are initially associated with type variables, and information is gathered during the typechecking process to determine what the type of the identifier should have been in the first place. Hence, we start with these initial broad assumptions, and we build the proof by applying the inference rules in some order. Some of the rules require the types of two subexpressions to be equal. This will not usually be the case, so we *make* them equal by unifying the respective types. This results in specializing some of the types of the identifiers. At this point we can imagine repeating the same proof, but starting with the more refined set of assumptions we have just determined: this time the types of the two subexpressions mentioned above will come out equal, and we can proceed.

The inference rules should be applied in an order which allows us to build the expression we are trying to type from left to right and from the bottom up. For example, earlier we wanted to show that $(\mathbf{fun}(x)\ x)\colon \forall\alpha.\alpha\to\alpha$. Take $x{:}\alpha$ as our set of assumptions. To deduce the type of $(\mathbf{fun}(x)\ x)$ bottom-up we start with the type of x, which we can obtain by [IDE], and then we build up $(\mathbf{fun}(x)\ x)$ by [ABS].

If we proceed left to right and bottom-up then, with the exception of [GEN] and [SPEC], at any point only one rule can be applied, depending on the syntactic construct we are trying to obtain next. Hence the problem reduces to choosing when to use [GEN] and [SPEC]; this is done in conjunction with the [LET] rule. To simplify the discussion, we only consider the following special case which can be derived by combining the [LET] and [BIND] rules.

$$[\text{LETBIND}]\quad \frac{A\vdash e'{:}\sigma \quad A.x{:}\sigma\vdash e{:}\tau}{A\vdash(\mathbf{let}\ x=e'\ \mathbf{in}\ e){:}\tau}$$

Before applying [LETBIND], we derive $A\vdash e'{:}\sigma'$ for some σ' (refer to the [LETBIND] rule), and then we apply all the possible [GEN] rules, obtaining $A\vdash e'{:}\sigma$, where σ can be a quantified type. Now we can start deriving $A.x{:}\sigma\vdash e{:}\tau$, and every time we need to use [IDE] for x and σ is quantified, we immediately use [SPEC] to strip all the quantifiers, replacing the quantifier variable by a fresh type variable. These new variables are then subject to instantiation, as discussed above, which determines more refined ways of using [SPEC].

As an exercise, one could try to apply the above heuristic to infer the type of *length*, and observe how this corresponds to what the typechecking algorithm does in that case. Note how the list of non-generic variables corresponds to the set of assumptions and the application of [GEN] and [SPEC] rules.

10. Conclusions and acknowledgements

This paper presented some of the pragmatic knowledge about polymorphic typechecking, trying to relate it informally to the theoretical background. These ideas have been developed by a number of people over a number of years, and have been transmitted to me by discussions with Luis Damas, Mike Gordon, Dave MacQueen, Robin Milner and Ravi Sethi.

Appendix A. The program

The following Modula-2 program implements the polymorphic typechecking algorithm.

The types Exp and Decl form the abstract syntax of our language. A type expressions TypeExp can be a type variable or a type operator. A type variable, is *uninstantiated* when its instance field is NIL, or *instantiated* otherwise. An instantiated type variable behaves like its instantiation. A type operator (like *bool* or →) has a name and a list of type arguments (none for *bool*, two for →).

The function Prune is used whenever a type expression has to be inspected: it will always return a type expression which is either an uninstantiated type variable or a type operator; i.e. it will skip instantiated variables, and will actually prune them from expressions to remove long chains of instantiated variables.

The function OccursInType checks whether a type variable occurs in a type expression.

The type NonGenericVars is the type of lists of non-generic variables. FreshType makes a copy of a type expression, duplicating the generic variables and sharing the non-generic ones.

The function IsGeneric checks whether a given variable occurs in a list of non-generic variables. Note that a variables in such a list may be instantiated to a type term, in which case the variables contained in the type term are considered non-generic.

Type unification is now easily defined. Remember that when unifying a non-generic variable to a term, all the variables in that term must become non-generic. This is handled automatically by the lists of non-generic variables, as explained above. Hence, no special code is needed in the unification routine.

Type environments are then defined. Note that RetrieveTypeEnv always creates fresh types; some of this copying is unnecessary and could be eliminated.

Finally we have the typechecking routine, which maintains a type environment and a list of non-generic variables. Recursive declarations are handled in two passes. The first pass AnalyzeRecDeclBind simply creates a new set of non-generic type variables and associates them with identifiers. The second pass AnalyzeRecDecl analyzes the declarations and makes calls to UnifyType to ensure the recursive type constraints.

The implementation modules for ErrorMod, SymbolMod and ParseTreeMod are not provided.

```
(**************************************************************************)
(*************************** DEFINITION MODULES ***************************)
(**************************************************************************)

(**************************************************************************)
DEFINITION MODULE ErrorMod;
PROCEDURE Msg(msg: ARRAY OF CHAR);
(* Print an error message *)
END ErrorMod.
```

```
(***************************************************************************)
DEFINITION MODULE SymbolMod;
TYPE
  Ide;
PROCEDURE New(string: ARRAY OF CHAR): Ide;
(* Create a new identifier from a string *)
PROCEDURE Equal(ide1, ide2: Ide): BOOLEAN;
(* Compare two identifiers *)
END SymbolMod.

(***************************************************************************)
DEFINITION MODULE ParseTreeMod;
IMPORT SymbolMod;
FROM SymbolMod IMPORT Ide;


TYPE
  Exp = POINTER TO ExpBase;
  (* Parse tree for expressions *)


  Decl = POINTER TO DeclBase;
  (* Parse tree for declarations *)


  ExpClass = (IdeClass, CondClass, LambClass, ApplClass, BlockClass);


 ExpBase = RECORD
   CASE class: ExpClass OF
   | IdeClass: ide: Ide;
   | CondClass: test, ifTrue, ifFalse: Exp;
   | LambClass: binder: Ide; body: Exp;
   | ApplClass: fun, arg: Exp;
   | BlockClass: decl: Decl; scope: Exp;
   END;
 END;


DeclClass = (DefClass, SeqClass, RecClass);

DeclBase = RECORD
  CASE class: DeclClass OF
  | DefClass: binder: Ide; def: Exp;
  | SeqClass: first, second: Decl;
  | RecClass: rec: Decl;
  END;
END;


(* Allocation routines for Exp and Decl *)
PROCEDURE NewIdeExp(ide: Ide): Exp;
PROCEDURE NewCondExp(test, ifTrue, ifFalse: Exp): Exp;
PROCEDURE NewLambExp(binder: Ide; body: Exp): Exp;
PROCEDURE NewApplExp(fun, arg: Exp): Exp;
PROCEDURE NewBlockExp(decl: Decl; scope: Exp): Exp;
PROCEDURE NewDefDecl(binder: Ide; def: Exp): Decl;
PROCEDURE NewSeqDecl(first, second: Decl): Decl;
PROCEDURE NewRecDecl(rec: Decl): Decl;

END ParseTreeMod.
```

```
(****************************************************************************)
DEFINITION MODULE TypeMod;
IMPORT SymbolMod;
FROM SymbolMod IMPORT Ide;

TYPE
  TypeExp = POINTER TO TypeExpBase;
  (* The internal representation of type expressions *)


TypeClass = (VarType, OperType);

TypeExpBase = RECORD
  CASE class: TypeClass OF
  | VarType: instance: TypeExp;
  | OperType: ide: Ide; args: TypeList;
  END;
END;

TypeList = POINTER TO TypeListBase;

  TypeListBase = RECORD head: TypeExp; tail: TypeList; END;

PROCEDURE NewTypeVar(): TypeExp;
(* Allocate a new type variable *)
PROCEDURE NewTypeOper(ide: Ide; args: TypeList): TypeExp;
(* Allocate a new type operator *)
VAR
  Empty: TypeList;
  (* The empty type list *)
PROCEDURE Extend(head: TypeExp; tail: TypeList): TypeList;
(* Allocate a new type list *)
PROCEDURE SameType(typeExp1, typeExp2: TypeExp): BOOLEAN;
(* Compare two types for identity (pointer equality) *)
PROCEDURE Prune(typeExp: TypeExp): TypeExp;
(* Eliminate redundant instantiated variables at the top of "typeExp";
   The result of Prune is always a non-instantiated type variable or a
   type operator *)
PROCEDURE OccursInType(typeVar: TypeExp; typeExp: TypeExp): BOOLEAN;
(* Wheather an uninstantiated type variable occurs in a type expression *)
PROCEDURE OccursInTypeList(typeVar: TypeExp; list: TypeList): BOOLEAN;
(* Wheather an uninstantiated type variable occurs in a list *)
PROCEDURE UnifyType(typeExp1, typeExp2: TypeExp);
(* Unify two type expressions *)
PROCEDURE UnifyArgs(list1, list2: TypeList);
(* Unify two lists of type expressions *)

END TypeMod.

(****************************************************************************)
DEFINITION MODULE GenericVarMod;
IMPORT TypeMod;
FROM TypeMod IMPORT TypeExp;

TYPE
  NonGenericVars;
  (* Lists of non-generic type variables and their instantiations *)

VAR
  Empty: NonGenericVars;
  (* The empty list *)
```

```
PROCEDURE Extend(head: TypeExp; tail: NonGenericVars): NonGenericVars;
(* Extend a list *)
PROCEDURE IsGeneric(typeVar: TypeExp; list: NonGenericVars): BOOLEAN;
(* Whether an uninstantiated type variable is generic w.r.t. a list of
   non-generic type variables *)
PROCEDURE FreshType(typeExp: TypeExp; list: NonGenericVars): TypeExp;
(* Make a copy of a type expression; the generic varibles are copied, while
   the non-generic variables are shared *)

END GenericVarMod.

(****************************************************************************)
DEFINITION MODULE EnvMod;
IMPORT SymbolMod, TypeMod, GenericVarMod;
FROM SymbolMod IMPORT Ide;
FROM TypeMod IMPORT TypeExp;
FROM GenericVarMod IMPORT NonGenericVars;

TYPE
  Env;
  (* Environments associating type expressions to identifiers *)

VAR
  Empty: Env;
  (* The empty environment *)
PROCEDURE Extend(ide: Ide; typeExp: TypeExp; tail: Env): Env;
(* Extend an environment with an identifier-type pair *)
PROCEDURE Retrieve(ide: Ide; env: Env; list: NonGenericVars): TypeExp;
(* Search for an identifier in an environment and return a "fresh" copy of
   the associated type (using GenericVar.FreshType). The identifier must be
   bound in the environment *)

END EnvMod.

(****************************************************************************)
DEFINITION MODULE TypecheckMod;
IMPORT ParseTreeMod, TypeMod, EnvMod, GenericVarMod;
FROM ParseTreeMod IMPORT Exp, Decl;
FROM TypeMod IMPORT TypeExp;
FROM EnvMod IMPORT Env;
FROM GenericVarMod IMPORT NonGenericVars;

PROCEDURE AnalyzeExp(exp: Exp; env: Env; list: NonGenericVars): TypeExp;
(* Typecheck an expression w.r.t. an environment, and return its type *)
PROCEDURE AnalyzeDecl(decl: Decl; env: Env; list: NonGenericVars): Env;
(* Typecheck a declaration w.r.t an environment, and return an extended
   environment containing the types of the identifiers introduced by the
   declaration *)

END TypecheckMod.

(****************************************************************************)
(************************ IMPLEMENTATION MODULES ************************)
(****************************************************************************)

(****************************************************************************)
IMPLEMENTATION MODULE TypeMod;
IMPORT ErrorMod;

PROCEDURE NewTypeVar(): TypeExp;
  VAR r: TypeExp;
  BEGIN
    NEW(r, VarType); r^.class := VarType; r^.instance := NIL; RETURN r;
  END NewTypeVar;
```

```
PROCEDURE NewTypeOper(ide: Ide; args: TypeList): TypeExp;
  VAR r: TypeExp;
  BEGIN
    NEW(r, OperType); r^.class := OperType; r^.ide := ide; r^.args := args;  RETURN r;
  END NewTypeOper;

PROCEDURE Extend(head: TypeExp; tail: TypeList): TypeList;
  VAR r: TypeList;
  BEGIN
    NEW(r); r^.head := head; r^.tail := tail; RETURN r;
  END Extend;

PROCEDURE SameType(typeExp1, typeExp2: TypeExp): BOOLEAN;
  BEGIN RETURN typeExp1 = typeExp2; END SameType;

PROCEDURE Prune(typeExp: TypeExp): TypeExp;
  BEGIN
    CASE typeExp^.class OF
    | VarType:
        IF typeExp^.instance = NIL THEN
          RETURN typeExp;
        ELSE
          typeExp^.instance := Prune(typeExp^.instance);
          RETURN typeExp^.instance;
        END;
    | OperType: RETURN typeExp;
    END;
  END Prune;

PROCEDURE OccursInType(typeVar: TypeExp; typeExp: TypeExp): BOOLEAN;
  BEGIN
    CASE typeExp^.class OF
    | VarType: RETURN SameType(typeVar, typeExp);
    | OperType: RETURN OccursInTypeList(typeVar, typeExp^.args);
    END;
  END OccursInType;

PROCEDURE OccursInTypeList(typeVar: TypeExp; list: TypeList): BOOLEAN;
  BEGIN
    IF list = NIL THEN RETURN FALSE END;
    IF OccursInType(typeVar, list^.head) THEN RETURN TRUE END;
    RETURN OccursInTypeList(typeVar, list^.tail);
  END OccursInTypeList;

PROCEDURE UnifyType(typeExp1, typeExp2: TypeExp);
  BEGIN
    typeExp1 := Prune(typeExp1);
typeExp2 := Prune(typeExp2);
CASE typeExp1^.class OF
| VarType:
    IF OccursInType(typeExp1, typeExp2) THEN
      IF NOT SameType(typeExp1, typeExp2) THEN
        ErrorMod.Msg("Type clash");
      END;
    ELSE
      typeExp1^.instance := typeExp2;
    END;

| OperType:
    CASE typeExp2^.class OF
    | VarType: UnifyType(typeExp2, typeExp1);
    | OperType:
            IF SymbolMod.Equal(typeExp1^.ide, typeExp2^.ide) THEN
              UnifyArgs(typeExp1^.args, typeExp2^.args);
            ELSE
              ErrorMod.Msg("Type clash");
            END;
        END;
    END;
  END UnifyType;

PROCEDURE UnifyArgs(list1, list2: TypeList);
  BEGIN
    IF (list1 = Empty) AND (list2 = Empty) THEN RETURN; END;
    IF (list1 # Empty) OR (list2 # Empty) THEN
      ErrorMod.Msg("Type clash");
    ELSE
      UnifyType(list1^.head, list2^.head);
      UnifyArgs(list1^.tail, list2^.tail);
    END;
  END UnifyArgs;

BEGIN
  Empty := NIL;
END TypeMod.

(***************************************************************************)
IMPLEMENTATION MODULE GenericVarMod;
FROM TypeMod IMPORT TypeClass, TypeList;

TYPE
  NonGenericVars = TypeList;

PROCEDURE Extend(head: TypeExp; tail: NonGenericVars): NonGenericVars;
  BEGIN RETURN TypeMod.Extend(head, tail); END Extend;

PROCEDURE IsGeneric(typeVar: TypeExp; list: NonGenericVars): BOOLEAN;
  BEGIN RETURN NOT TypeMod.OccursInTypeList(typeVar, list); END IsGeneric;

TYPE
  CopyEnv = POINTER TO CopyEnvBase;
  CopyEnvBase = RECORD old, new: TypeExp; tail: CopyEnv; END;

PROCEDURE ExtendCopyEnv(old, new: TypeExp; tail: CopyEnv): CopyEnv;
  VAR r: CopyEnv;
  BEGIN
    NEW(r); r^.old := old; r^.new := new; r^.tail := tail; RETURN r;
  END ExtendCopyEnv;

PROCEDURE FreshVar(typeVar: TypeExp; scan: CopyEnv; VAR env: CopyEnv): TypeExp;
  VAR newTypeVar: TypeExp;
  BEGIN
    IF scan = NIL THEN
      newTypeVar := TypeMod.NewTypeVar();
      env := ExtendCopyEnv(typeVar, newTypeVar, env);
      RETURN newTypeVar;
    ELSIF TypeMod.SameType(typeVar, scan^.old) THEN
      RETURN scan^.new
    ELSE
      RETURN FreshVar(typeVar, scan^.tail, (*VAR*) env);
    END;
  END FreshVar;

PROCEDURE Fresh(typeExp: TypeExp; list: NonGenericVars; VAR env: CopyEnv): TypeExp;
  BEGIN
    typeExp := TypeMod.Prune(typeExp);
    CASE typeExp^.class OF
    | VarType:
        IF IsGeneric(typeExp, list) THEN
          RETURN FreshVar(typeExp, env, (*VAR*) env)
        ELSE
          RETURN typeExp
        END;
    | OperType:
        RETURN
          TypeMod.NewTypeOper(typeExp^.ide,
            FreshList(typeExp^.args, list, (*VAR*) env));
    END;
  END Fresh;

PROCEDURE FreshList(args: TypeList; list: NonGenericVars; VAR env: CopyEnv): TypeList;
  BEGIN
    IF args = TypeMod.Empty THEN RETURN TypeMod.Empty END;
    RETURN
      TypeMod.Extend(Fresh(args^.head, list, (*VAR*) env),
        FreshList(args^.tail, list, (*VAR*) env));
  END FreshList;

PROCEDURE FreshType(typeExp: TypeExp; list: NonGenericVars): TypeExp;
  VAR env: CopyEnv;
  BEGIN env := NIL; RETURN Fresh(typeExp, list, (*VAR*) env); END FreshType;

BEGIN
  Empty := TypeMod.Empty;
END GenericVarMod.

(****************************************************************************)
IMPLEMENTATION MODULE EnvMod;
IMPORT ErrorMod;

TYPE
  Env = POINTER TO EnvBase;
  EnvBase = RECORD ide: Ide; typeExp: TypeExp; tail: Env; END;

PROCEDURE Extend(ide: Ide; typeExp: TypeExp; tail: Env): Env;
  VAR r: Env;
  BEGIN
    NEW(r); r^.ide := ide; r^.typeExp := typeExp; r^.tail := tail; RETURN r;
  END Extend;

PROCEDURE Retrieve(ide: Ide; env: Env; list: NonGenericVars): TypeExp;
  BEGIN
    IF env = EnvMod.Empty THEN
      ErrorMod.Msg("Unbound ide");
      RETURN NIL;
    ELSIF SymbolMod.Equal(ide, env^.ide) THEN
      RETURN GenericVarMod.FreshType(env^.typeExp, list);
    ELSE
      RETURN Retrieve(ide, env^.tail, list);
    END;
  END Retrieve;

BEGIN
  Empty := NIL;
END EnvMod.

(****************************************************************************)
IMPLEMENTATION MODULE TypecheckMod;
IMPORT SymbolMod;
FROM ParseTreeMod IMPORT ExpClass, DeclClass;
FROM TypeMod IMPORT NewTypeVar, NewTypeOper, UnifyType, UnifyArgs;
```

```
VAR
  BoolType: TypeExp;

PROCEDURE FunType(dom, cod: TypeExp): TypeExp;
  BEGIN
    RETURN
      NewTypeOper(SymbolMod.New("->"),
        TypeMod.Extend(dom, TypeMod.Extend(cod, TypeMod.Empty)))
  END FunType;

PROCEDURE AnalyzeExp(exp: Exp; env: Env; list: NonGenericVars): TypeExp;
  VAR
    typeOfThen, typeOfElse, typeOfBinder, typeOfBody, typeOfFun, typeOfArg,
    typeOfRes: TypeExp;
    bodyEnv, declEnv: Env;
    bodyList: NonGenericVars;
  BEGIN
    CASE exp^.class OF
    | IdeClass: RETURN EnvMod.Retrieve(exp^.ide, env, list);
    | CondClass:
        UnifyType(AnalyzeExp(exp^.test, env, list), BoolType);
        typeOfThen := AnalyzeExp(exp^.ifTrue, env, list);
        typeOfElse := AnalyzeExp(exp^.ifFalse, env, list);
        UnifyType(typeOfThen, typeOfElse);
        RETURN typeOfThen;
    | LambClass:
        typeOfBinder := NewTypeVar();
        bodyEnv := EnvMod.Extend(exp^.binder, typeOfBinder, env);
        bodyList := GenericVarMod.Extend(typeOfBinder, list);
        typeOfBody := AnalyzeExp(exp^.body, bodyEnv, bodyList);
        RETURN FunType(typeOfBinder, typeOfBody);
    | ApplClass:
        typeOfFun := AnalyzeExp(exp^.fun, env, list);
        typeOfArg := AnalyzeExp(exp^.arg, env, list);
        typeOfRes := NewTypeVar();
        UnifyType(typeOfFun, FunType(typeOfArg, typeOfRes));
        RETURN typeOfRes;
    | BlockClass:
        declEnv := AnalyzeDecl(exp^.decl, env, list);
        RETURN AnalyzeExp(exp^.scope, declEnv, list);
    END;
  END AnalyzeExp;

PROCEDURE AnalyzeDecl(decl: Decl; env: Env; list: NonGenericVars): Env;
  BEGIN
    CASE decl^.class OF
    | DefClass:
        RETURN
          EnvMod.Extend(decl^.binder, AnalyzeExp(decl^.def, env, list), env);
    | SeqClass:
        RETURN
          AnalyzeDecl(decl^.second, AnalyzeDecl(decl^.first, env, list), list);
    | RecClass:
        AnalyzeRecDeclBind(decl^.rec, (*VAR*) env, (*VAR*) list);
        AnalyzeRecDecl(decl^.rec, env, list);
        RETURN env;
    END;
  END AnalyzeDecl;

PROCEDURE AnalyzeRecDeclBind(decl: Decl; VAR env: Env; VAR list: NonGenericVars);
  VAR newTypeVar: TypeExp;
  BEGIN
  CASE decl^.class OF
  | DefClass:
      newTypeVar := NewTypeVar();
      env := EnvMod.Extend(decl^.binder, newTypeVar, env);
      list := GenericVarMod.Extend(newTypeVar, list);
  | SeqClass:
      AnalyzeRecDeclBind(decl^.first, (*VAR) env, (*VAR*) list);
      AnalyzeRecDeclBind(decl^.second, (*VAR) env, (*VAR*) list);
  | RecClass: AnalyzeRecDeclBind(decl^.rec, (*VAR) env, (*VAR*) list);
  END;
END AnalyzeRecDeclBind;

PROCEDURE AnalyzeRecDecl(decl: Decl; env: Env; list: NonGenericVars);
  BEGIN
    CASE decl^.class OF
    | DefClass:
        UnifyType(EnvMod.Retrieve(decl^.binder, env, list),
          AnalyzeExp(decl^.def, env, list));
    | SeqClass:
        AnalyzeRecDecl(decl^.first, env, list);
        AnalyzeRecDecl(decl^.second, env, list);
    | RecClass: AnalyzeRecDecl(decl^.rec, env, list);
    END;
  END AnalyzeRecDecl;

BEGIN
  BoolType := NewTypeOper(SymbolMod.New("bool"), TypeMod.Empty);
END TypecheckMod.
```

References

[1] K.B. Bruce and R. Meyer, The semantics of second order polymorphic lambda calculus, in: *Semantics of Data Types*, Lecture Notes in Computer Science **173** (Springer, Berlin, 1984). Also to appear under the same title together with J.C. Mitchell.

[2] R. Burstall, D. MacQueen and D. Sannella, Hope: An experimental applicative language, *Conference Record of the 1980 LISP Conference*, Stanford (1980) 136-143.

[3] L. Cardelli and P. Wegner, On understanding types, data abstraction and polymorphism, *Comput. Surveys*, to appear.

[4] M. Coppo, An extended polymorphic type system for applicative languages, Lecture Notes in Computer Science **88** (Springer, Berlin, 1980) 194-204.

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

[6] L. Damas and R. Milner, Principal type-schemes for functional programs, *Proc. POPL 82* (1982) 207-212.

[7] J.-Y. Girard, Une extension de l'interprétation de Gödel a l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types, *Proc. 2nd Scandinavian Logic Symposium* (1971) 63-92.

[8] M.J. Gordon, R. Milner and C.P. Wadsworth, *Edinburgh LCF*, Lecture Notes in Computer Science **78** (Springer, Berlin, 1979).

[9] R. Hindley, The principal type scheme of an object in combinatory logic, *Trans. Amer. Math. Soc.* **146** (1969) 29-60.

[10] D.B. MacQueen, G.D. Plotkin and R. Sethi, An ideal model for recursive polymorphic types, *Proc. POPL 84*. Also to appear in *Information and Control*.

[11] N. McCracken, The typechecking of programs with implicit type structure, in: *Semantics of Data Types*, Lecture Notes in Computer Science **173** (Springer, Berlin, 1984) 301-316.

[12] R. Milner, A theory of type polymorphism in programming, *J. Comput. System Sci.* **17** (1978).

[13] R. Milner, A proposal for Standard ML, *Proc. 1984 ACM Symposium on Lisp and Functional Programming* (1984).

[14] J.A. Robinson, A machine-oriented logic based on the resolution principle, *J. ACM* **12** (1) (1965) 23–49.

[15] D.S. Scott, Data types as lattices, *SIAM J. Comput.* **4** (1976).

[16] C. Strachey, Fundamental concepts in programming languages, Lecture notes for the International Summer School in Computer Programming, Copenhagen, 1967.