Luca Cardelli, Basic Polymorphic Typechecking, AT&T Bell Laboratories Computing Science Technical Report, 1984.
Also in Polymorphism Newsletters, II.1, Jan 1984.

Basic polymorphic typechecking

Luca Cardelli

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

Polymorphic typechecking has its foundations in a type system devised by Hindley [Hindley 69], and later rediscovered and extended by Milner [Milner 78]. As implemented in ML [Gordon 79, Milner 84], this type system shares with Algol 68 properties of compile-time checking, strong typing and higher-order functions, but it is more flexible in allowing polymorphism, i.e. the ability to define functions which work uniformly on arguments of many types.

Milner's polymorphic typechecking algorithm has proved very successful: it is sound, efficient, and supports a very rich and flexible type system. It can also be used to infer the types of untyped or partially typed programs.

However, 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 [Milner 78] 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 an ML program, which is very close to the one used in LCF, Hope and ML [Gordon 79, Burstall 80, Milner 84]. 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 in the program presented below, 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 records and union types. Many other extensions are being studied, and there is a diffuse feeling that important discoveries are yet to be made, both in the theory and the practice of typechecking.

This paper presents two views of typing, as a system of type equations and as a type inference system, and attempts to relate them informally to the implementation.

2. A simple applicative language

The language considered here is a simple typed lambda 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; the corresponding abstract syntax is given by the type "Term" in the program at the end of this paper (parsers and printers are not provided).

```
Term ::=
  Identifier |
  'if' Term 'then' Term 'else' Term |
  'fun' Identifier '.' Term |
  Term Term |
  'let' Declaration 'in' Term |
```

```
'(' Term ')'

Declaration ::=
  Identifier '=' Term |
  Declaration 'and' Declaration |
  'rec' Declaration |
  '(' Declaration ')'
```

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.

For example, the following program defines the factorial function and applies it to zero:

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

3. 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**, $(\beta \times \xi) \rightarrow (\beta \times \xi)$, etc. This contextual instantiation process is performed by *unification*, [Robinson 1] 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 **I** = **fun x. x** has type $\alpha \rightarrow \alpha$ because it maps any type onto itself. In the expression **(I 0)** the type of **0** (i.e. **int**) is matched to the domain of the type of **I**, yielding **int** $\rightarrow$ **int** as the specialized type of **I** in that context. Hence the type of **(I 0)** is the codomain of the type of **I**, 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, **list** is the list operator, and $\times$ is cartesian product):

```
true, false      : bool
0, 1, ...        : int
succ, pred       : int → int
zero             : int → bool
```

pair	: $\alpha \rightarrow \beta \rightarrow (\alpha \times \beta)$
fst	: $(\alpha \times \beta) \rightarrow \alpha$
snd	: $(\alpha \times \beta) \rightarrow \beta$
nil	: α **list**
cons	: $(\alpha \times \alpha$ **list**$) \rightarrow \alpha$ **list**
hd	: α **list** $\rightarrow \alpha$
tl	: α **list** $\rightarrow \alpha$ **list**
null	: α **list** $\rightarrow$ **bool**

4. 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 ...
```

(we write **length** = **fun l. ...** instead of the equivalent but more elegant **length l** = **...** for convenience in the discussion below).

The type of **length** is α **list** $\rightarrow$ **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**	: α **list** $\rightarrow$ **bool**
[2]	**tl**	: β **list** $\rightarrow \beta$ **list**
[3]	**0**	: **int**
[4]	**succ**	: **int** $\rightarrow$ **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**	: $\delta \rightarrow \epsilon$
[10]	**l**	: δ
[11]	**null l**	: ϵ
[12]	**tl**	: $\phi \rightarrow \xi$
[13]	**l**	: ϕ
[14]	**tl l**	: ξ

```
[15]  length                    : θ → ι
[16]  tl l                      : θ
[17]  length(tl l)              : ι

[18]  succ                      : κ → λ
[19]  length(tl l)              : κ
[20]  succ(length(tl l))        : λ

[21]  l                         : μ
[22]  if null l then 0
      else succ(length(tl l))   : ν
[23]  fun l. if null l then 0
        else succ(length(tl l)) : μ → ν

[24]  length                    : π
[25]  fun l. if null l then 0
        else succ(length(tl l)) : π
```

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. $\delta \rightarrow \epsilon$ 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 lambda-expression [23] has a type $\mu \rightarrow \nu$, 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 **int** = **bool**), and (ii) solving the constraints with respect to π. The expected type of **length** (π = β **list** → **int**) can be inferred as follows:

$\pi = \mu \rightarrow \nu$	**by [25, 23]**
$\mu = \phi =$ β **list**	**by [21, 13, 12, 2]**
$\nu = \gamma =$ **int**	**by [22, 8, 6, 3]**

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                                 : δ              [10]
[27]  null l                            : bool
      ε = bool; δ = α list;                              [11, 9, 1]
[28]  0                                 : int
      γ = int;                                           [6, 3]
[29]  tl l                              : β list
      φ = β list; ξ = β list; β = α;                     [26, 27, 12-14, 2]
[30]  length(tl l)                      : ι
      θ = β list;                                        [15-17, 29]
```

```
[31]  succ(length(tl l))          : int
      ι = κ = int;                                  [18-20, 4, 30]
[32]  if null l then 0
      else succ(length(tl l))     : int             [5-8, 27, 28, 31]
[33]  fun l. if null l then 0
        else succ(length(tl l))   : β list → int
      μ = β list; ν = int;                          [21-23, 26, 27, 32]
[34]  length                      : β list → int
      π = β list → int;                             [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].

5. Typechecking

The basic algorithm can be described as follows.

1. When a new variable **x** is introduced by a lambda 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 <**x**,α> is stored in an environment (called **TypeEnv** in the program) 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.

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.

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.

4. In an application **f a**, the type of **f** is unified against a type **A** → β, 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** → β for **f**, and the second occurrence determines a type **bool** → β for **f**, which cannot be unified with the first one.

Type variables appearing in the type of a lambda-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 **Ex1**, for example by somehow assigning it (α → β) → (β × β); this would compute correctly in situations like (**Ex1 (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 α → β and it would be accepted as an argument to **Ex1** and wrongly applied to **true**. There are sound extensions of Milner's type system which can type **Ex1**, but they are beyond the scope of this discussion.

Hence there is a basic problem in typing heterogeneous occurrences of lambda-bound identifiers. This turns out to be tolerable in practice, because expressions like **Ex1** 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 **Ex1**, 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* lambda-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 **Ex1**, and **g** has a non-generic type which propagates to **f**:

```
fun g. let f = g                    [Ex3]
   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 iff it does not occur in the type of the binder
> of any lambda-expression enclosing **e**.

Note that a type variable which is found to be non-generic while typechecking within a lambda expression, may become generic outside it. This is the case in **Ex2** where **a** is assigned a non-generic α, and f is assigned $\alpha \to \alpha$ 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 lambda; when leaving the lambda 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 = ... f ...
in ... f ...
```

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

```
let f = Y fun f. ... f ...
in ... f ...
```

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.

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**.

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 $<x_i,\alpha_i>$ 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's are typechecked in that environment, and their types T_i are again matched against the α_i (or their instantiations).

6. 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 [Scott 76] [Milner 78] [MacQueen 84].

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 [Burstall 80] 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.

7. 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, $\alpha \rightarrow \beta$ is really $\forall\alpha.\ \forall\beta.\ \alpha \rightarrow \beta$. However, quantifiers cannot be nested inside type expressions.

A type is called *shallow* if it has the form $(\forall\alpha_1.\ \cdots\ \forall\alpha_n.\ \tau)$ where $n\geq 0$ 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 [Damas 82].

Here is the set of inference rules. [VAR] is an axiom scheme, while the other rules are proper inferences. The horizontal bar reads "implies". The notation $A \vdash e : \tau$ means that given a set of assumptions A, we can deduce that the expression e has type τ. An assumption is a typing of a variable which may be free in the expression e. The notation $A.x : \tau$ stands for the union of the set A with the assumption $x : \tau$; and $\tau[\sigma/\alpha]$ is the result of substituting σ for all the free occurrences of α in τ.

[VAR] $\quad A.x : \tau \vdash x : \tau$

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

[ABS] $\quad \dfrac{A.x : \sigma \vdash e : \tau}{A \vdash (\lambda x.\ e) : \sigma \to \tau}$

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

[LET] $\quad \dfrac{A \vdash e' : \sigma \quad A.x : \sigma \vdash e : \tau}{A \vdash (let\ x = e'\ in\ e) : \tau}$

[REC] $\quad \dfrac{A.x : \tau \vdash e : \tau}{A \vdash (rec\ x.\ e) : \tau}$

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

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

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

$$\dfrac{\dfrac{x : \alpha \vdash x : \alpha}{\vdash (fun\ x.\ x) : \alpha \to \alpha}\ [ABS]}{\vdash (fun\ x.\ x) : \forall \alpha.\ \alpha \to \alpha}\ [GEN]$$

[VAR]

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

$$\dfrac{\vdash (fun\ x.\ x) : \forall \alpha.\ \alpha \to \alpha}{\vdash (fun\ x.\ x) : int \to int}\ [SPEC]$$

or more directly:

$$\dfrac{x : int \vdash x : int}{\vdash (fun\ x.\ x) : int \to int}\ [ABS]$$

[VAR]

We can extend the above inference to show (*fun x.x*)(3): *int*;

$$\dfrac{\dfrac{\dfrac{}{x: int,\ 3: int \vdash x: int}\ [VAR]}{3: int \vdash (fun\ x.\ x): int \to int}\ [ABS] \qquad \dfrac{}{3: int \vdash 3: int}\ [VAR]}{3: int \vdash (fun\ x.\ x)(3): int}\ [COMB]$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:\phi \vdash x:\phi}\ [VAR]}{x:\phi \vdash x:\phi\to\phi}\ [SPEC] \qquad \dfrac{}{x:\phi \vdash x:\phi}\ [VAR]}{x:\phi \vdash x\ x:\phi}\ [COMB]}{\vdash (fun\ x.\ x\ x):\phi\to\phi}\ [ABS]$$

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 (*let f = fun x.x in pair*(*f* 3)(*f true*)): *int*×*bool*. Take A = {3: *int*, *true*: *bool*, *pair*: $\forall\alpha.\ \forall\beta.\ \alpha\to\beta\to\alpha\times\beta$} and $\phi = \forall\alpha.\ \alpha \to \alpha$.

$$\dfrac{\dfrac{}{A.f:\phi \vdash f:\phi}}{\dfrac{A.f:\phi \vdash f: int\to int \qquad A.f:\phi \vdash 3: int}{A.f:\phi \vdash f\ 3: int}}$$

$$\dfrac{\dfrac{}{A.f:\phi \vdash f:\phi}}{\dfrac{A.f:\phi \vdash f: bool\to bool \qquad A.f:\phi \vdash true: bool}{A.f:\phi \vdash f\ true: bool}}$$

$$\dfrac{\begin{array}{l} A.f:\phi \vdash f\ 3: int \\ A.f:\phi \vdash f\ true: bool \\ A.f:\phi \vdash pair: \forall\alpha.\forall\beta.\alpha\to\beta\to\alpha\times\beta \\ \qquad\qquad \cdots \end{array}}{A.f:\phi \vdash pair\ (f\ 3)(f\ true): int\times bool}$$

$$\dfrac{A \vdash fun\ x.x: \phi \qquad A.f:\phi \vdash pair\ (f\ 3)(f\ true): int\times bool}{A \vdash (let\ f = fun\ x.x\ in\ pair\ (f\ 3)(f\ true)): int\times bool}$$

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 (*fun f. pair*(*f* 3)(*f true*))(*fun x. x*), which is the function-application version of the above let expression, no shallow type can be deduced for (*fun f. pair*(*f* 3)(*f true*)).

A variable is generic if it does not appear in the type of the variables of any enclosing lambda-binder. Those binders must occur in the set of assumptions, so that they can be later discarded by [ABS] to create those enclosing lambdas. 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 [Milner 78] 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 the 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 (*fun x. x x*)(*fun x. x*): $\forall\alpha.\ \alpha\rightarrow\alpha$.

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, lambda-bound identifiers are initially associated to 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 (*fun x. x*): $\forall\alpha.\ \alpha\rightarrow\alpha$. Take $x:\alpha$ as our set of assumptions. To deduce the type of (*fun x. x*) bottom-up we start with the type of x, which we can obtain by [VAR], and then we build up (*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.

Before applying [LET], we derive $A \vdash e' : \sigma'$ (refer to the [LET] 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 [VAR] 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.

8. The program

The following ML program implements the polymorphic typechecking algorithm, and also illustrates how polymorphism is used in ML. ML syntax and semantics are described in [Milner 84]. Here are some comments on the program and the ML language; they refer to the program code.

Keywords are in boldface, identifiers in roman, and data constructors in italic. Data constructors are used in expressions to create data, and in pattern matching to analyze and select data components. Type variables are roman identifiers starting with a quote: ''a' is a type variable and is normally pronounced "alpha".

- The program begins with standard list manipulation routines, defined by pattern matching; note that no types are declared here.

- Pointers, in the sense of assignable references to values, are not predefined in ML. They can be built by 'ref' (built-in updatable references) and 'Option'. A pointer is an updatable reference to an optional value; if the value is missing ('none') we have a null pointer, otherwise we have a non-null pointer ('some'). 'Void' creates a new null pointer, 'Access' dereferences a

pointer, and 'Assign' updates a pointer. The type ''a Pointer' is parametric, but it will only be used here as a 'Type Pointer'.

- Time stamps are used to uniquely identify variables. This is an abstract type (so that time stamps cannot be faked) with an own variable 'Counter' which is incremented every time a new time stamp is needed.

- The types 'Ide', 'Term' and 'Decl' form the abstract syntax of our language. A type expressions 'Type' can be a type variable or a type operator. A type variable, identified by a unique time stamp, is *uninstantiated* when its type pointer is null, 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 'NGVars' 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.

- Type unification is now easily defined. Remember that when unifying a non-generic variable to a term, all the variables in that term become non-generic. This is handled automatically by the lists of non-generic variables, and 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.

```
{ ------ List Manipulation (standard library routines) ------ }

val rec
        map f nil = nil |
        map f (head :: tail) = (f head) :: (map f tail);

val rec
        fold f nil x = x |
        fold f (head :: tail) x = f (head, fold f tail x);

val rec
        exists p nil = false |
        exists p (head :: tail) = if p head then true else exists p tail;

{ ------ Options ------ }

type 'a Option = data none | some of 'a;

{ ------ Pointers ------ }

type 'a Pointer = data pointer of 'a Option ref;
val Void() = pointer(ref none);
val Access(pointer(ref(some V))) = V;
val Assign(pointer P, V) = P := some V;

{ ----- Time Stamps ----- }

abstype Stamp = data stamp of int;
with  local Counter = ref 0
      in    val NewStamp() = (Counter := !Counter+1; stamp(!Counter));
            val SameStamp(stamp S, stamp S') = (S = S')
      end
end;
```

```
{ ----- Identifiers ----- }

type Ide = data symbol of string;

{ ----- Expressions ----- }

type rec Term = data
        ide of Ide |
        cond of Term * Term * Term |
        lamb of Ide * Term |
        appl of Term * Term |
        block of Decl * Term

and Decl = data
        defDecl of Ide * Term |
        andDecl of Decl * Decl |
        recDecl of Decl;

{ ----- Types ----- }

type rec Type = data
        var of Stamp * Type Pointer |
        oper of Ide * Type list;

val NewTypeVar() = var(NewStamp(),Void());
val NewTypeOper(Name,Args) = oper(Name,Args);

val SameVar (var(Stamp,_),var(Stamp',_)) =
        SameStamp(Stamp,Stamp');

val rec Prune (Type: Type): Type =
        case Type of
                var(_,Instance) =>
                        (case Instance of
                                pointer(ref none). Type |
                                pointer(_).
                                        let val Pruned = Prune(Access Instance)
                                        in (Assign(Instance,Pruned); Pruned) end
                        ) |
                oper(_) => Type;

val rec OccursInType(TypeVar: Type, Type: Type): bool =
        let val Type = Prune Type
        in case Type of
                var(_) => SameVar(TypeVar,Type) |
                oper(Name,Args) =>
                        fold (fun Arg,Accum => OccursInType(TypeVar,Arg) orelse Accum) Args false
        end;

val OccursInTypeList(TypeVar: Type, TypeList: Type list): bool =
        exists (fun Type => OccursInType(TypeVar,Type)) TypeList;
```

```
{ ----- Generic Variables ----- }

type NGVars = data nonGenericVars of Type list;

val EmptyNGVars = nonGenericVars [];

val ExtendNGVars(Type: Type, nonGenericVars NGVars): NGVars =
      nonGenericVars(Type :: NGVars);

val Generic(TypeVar: Type, nonGenericVars NGVars): bool =
      not(OccursInTypeList(TypeVar,NGVars));

{ ----- Copy Type ----- }

type CopyEnv = (Type * Type) list;

val FreshType (Type: Type, NGVars: NGVars): Type =
      let val rec Fresh (Type: Type, Env: CopyEnv ref): Type =
            let val Type = Prune Type
            in case Type of
                  var(_) =>
                        if Generic(Type,NGVars) then FreshVar(Type,!Env,Env) else Type |
                  oper(Name,Args) =>
                        NewTypeOper(Name, map (fun Arg => Fresh(Arg,Env)) Args)
            end
      and FreshVar (Var: Type, Scan: CopyEnv, Env: CopyEnv ref): Type =
            if null Scan
            then  let val NewVar = NewTypeVar()
                  in (Env := (Var,NewVar)::(!Env); NewVar) end
            else  let val (OldVar,NewVar)::Rest = Scan
                  in if SameVar(Var,OldVar) then NewVar else FreshVar(Var,Rest,Env) end
      in Fresh(Type,ref []) end;

{ ----- Basic Type Operators ----- }

val BoolType =
      NewTypeOper(symbol "bool",[]);

val FunType (From: Type, Into: Type): Type =
      NewTypeOper(symbol "fun",[From;Into]);
```

```
{ ----- Type Unification ----- }

val rec UnifyType (Type: Type, Type´: Type): unit =
      let val Type = Prune Type and Type´ = Prune Type´
      in case Type of
            var(Stamp,Instance) =>
                  if OccursInType(Type,Type´)
                  then case Type´ of var(_) => () | oper(_) => escape "Unify"
                  else Assign(Instance,Type´) |
            oper(Name,Args) =>
                  case Type´ of
                        var(_) => UnifyType(Type´,Type) |
                        oper(Name´,Args´) =>
                              if Name=Name´ then UnifyArgs(Args,Args´) else escape "Unify"
      end

and   UnifyArgs ([], []) = () |
      UnifyArgs (Hd::Tl, Hd´::Tl´) = (UnifyType(Hd, Hd´); UnifyArgs(Tl, Tl´)) |
      UnifyArgs (_) = escape "Unify";

{ ----- Environments ----- }

type TypeEnv = data typeEnv of (Ide * Type) list;

val EmptyTypeEnv = typeEnv [];

val ExtendTypeEnv (Bind: Ide, Type: Type, typeEnv TypeEnv): TypeEnv =
      typeEnv((Bind,Type)::TypeEnv);

val RetrieveTypeEnv (Ide: Ide, typeEnv TypeEnv, NGVars: NGVars): Type =
      let val rec
            Retrieve ([]: (Ide * Type) list): Type = escape "Undefined identifier" |
            Retrieve ((Bind,Type)::Rest: (Ide * Type) list): Type =
                  if Ide=Bind then FreshType(Type,NGVars) else Retrieve Rest
      in Retrieve TypeEnv end;
```

```
{ ----- Typechecking ----- }

val rec
      AnalyzeTerm (ide Ide, TypeEnv, NGVars): Type =
            RetrieveTypeEnv(Ide,TypeEnv,NGVars) |
      AnalyzeTerm (cond(If,Then,Else), TypeEnv, NGVars): Type =
            let    val () = UnifyType(AnalyzeTerm(If,TypeEnv,NGVars),BoolType);
                   val TypeOfThen = AnalyzeTerm(Then,TypeEnv,NGVars);
                   val TypeOfElse = AnalyzeTerm(Else,TypeEnv,NGVars)
            in (UnifyType(TypeOfThen,TypeOfElse); TypeOfThen) end |
      AnalyzeTerm (lamb(Bind,Body), TypeEnv, NGVars): Type =
            let    val TypeOfBind = NewTypeVar();
                   val BodyTypeEnv = ExtendTypeEnv(Bind,TypeOfBind,TypeEnv);
                   val BodyNGVars = ExtendNGVars(TypeOfBind,NGVars);
                   val TypeOfBody = AnalyzeTerm(Body,BodyTypeEnv,BodyNGVars)
            in FunType(TypeOfBind,TypeOfBody) end |
      AnalyzeTerm (appl(Fun,Arg), TypeEnv, NGVars): Type =
            let    val TypeOfFun = AnalyzeTerm(Fun,TypeEnv,NGVars);
                   val TypeOfArg = AnalyzeTerm(Arg,TypeEnv,NGVars);
                   val TypeOfRes = NewTypeVar()
            in (UnifyType(TypeOfFun,FunType(TypeOfArg,TypeOfRes)); TypeOfRes) end |
      AnalyzeTerm (block(Decl,Scope), TypeEnv, NGVars): Type =
            let val DeclEnv = AnalyzeDecl(Decl,TypeEnv,NGVars)
            in AnalyzeTerm(Scope,DeclEnv,NGVars) end

and   AnalyzeDecl (defDecl(Bind,Term), TypeEnv, NGVars): TypeEnv =
            ExtendTypeEnv(Bind,AnalyzeTerm(Term,TypeEnv,NGVars),TypeEnv) |
      AnalyzeDecl (andDecl(Left,Right), TypeEnv, NGVars): TypeEnv =
            AnalyzeDecl(Right,AnalyzeDecl(Left,TypeEnv,NGVars),NGVars) |
      AnalyzeDecl (recDecl Rec, TypeEnv, NGVars): TypeEnv =
            let val TypeEnv,NGVars = AnalyzeRecDeclBind(Rec,TypeEnv,NGVars)
            in AnalyzeRecDecl(Rec,TypeEnv,NGVars) end

and   AnalyzeRecDeclBind (defDecl(Bind,Term), TypeEnv, NGVars) : TypeEnv * NGVars =
            let val Var = NewTypeVar()
            in ExtendTypeEnv(Bind,Var,TypeEnv), ExtendNGVars(Var,NGVars) end |
      AnalyzeRecDeclBind (andDecl(Left,Right), TypeEnv, NGVars) : TypeEnv * NGVars =
            let val TypeEnv,NGVars = AnalyzeRecDeclBind(Left,TypeEnv,NGVars)
            in AnalyzeRecDeclBind(Right,TypeEnv,NGVars) end |
      AnalyzeRecDeclBind (recDecl Rec, TypeEnv, NGVars) : TypeEnv * NGVars =
            AnalyzeRecDeclBind(Rec,TypeEnv,NGVars)

and   AnalyzeRecDecl (defDecl(Bind,Term), TypeEnv, NGVars): TypeEnv =
            (UnifyType(RetrieveTypeEnv(Bind,TypeEnv,NGVars),
                  AnalyzeTerm(Term,TypeEnv,NGVars));
             TypeEnv) |
      AnalyzeRecDecl (andDecl(Left,Right), TypeEnv, NGVars): TypeEnv =
            AnalyzeRecDecl(Right,AnalyzeRecDecl(Left,TypeEnv,NGVars),NGVars) |
      AnalyzeRecDecl (recDecl Rec, TypeEnv, NGVars): TypeEnv =
            AnalyzeRecDecl(Rec,TypeEnv,NGVars);
```

9. 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.

References

[Burstall 80] R.Burstall, D.MacQueen, D.Sannella: "Hope: an Experimental Applicative Language", Conference Record of the 1980 LISP Conference, Stanford, August 1980, pp. 136-143.

[Damas 82] L.Damas, R.Milner: "Principal type-schemes for functional programs", Proc. POPL 82, pp.207-212.

[Gordon 79] M.J.Gordon, R.Milner, C.P.Wadsworth: "Edinburgh LCF", Lecture Notes in Computer Science, No. 78, Springer-Verlag, 1979.

[Hindley 69] R.Hindley: "The principal type scheme of an object in combinatory logic", Transactions of the American Mathematical Society, Vol. 146, Dec 1969, pp. 29-60.

[MacQueen 84] D.B.MacQueen, R.Sethi, G.D.Plotkin: "An ideal model for recursive polymorphic types", Proc. Popl 84.

[Milner 78] R.Milner: "A theory of type polymorphism in programming", Journal of Computer and System Sciences, No. 17, 1978.

[Milner 84] R.Milner: "A proposal for Standard ML", Proc. of the 1984 ACM Symposium on Lisp and Functional Programming, Aug 1984.

[Robinson 65] J.A.Robinson: "A machine-oriented logic based on the resolution principle", Journal of the ACM, Vol 12, No. 1, Jan 1965, pp. 23-49.

[Scott 76] D.S.Scott: "Data types as lattices", SIAM Journal of Computing, 4, 1976.