First-Class Polymorphism for ML

Stefan Kahrs*

University of Edinburgh
Laboratory for Foundations of Computer Science
King's Buildings, EH9 3JZ

Abstract. Polymorphism in ML is implicit: type variables are silently introduced and eliminated. The lack of an explicit declaration of type variables restricts the expressiveness of parameterised modules (functors). Certain polymorphic functions cannot be expressed as functors, because implicit type parameters of polymorphic functions are in one respect more powerful than formal type parameters of functors.
The title suggests that this lack of expressiveness is due to a restricted ability to abstract — polymorphism is restricted. Type variables can only be abstracted from value declarations, but not from other forms of declarations, especially not from structure declarations.
The paper shows in the case of Standard ML how (syntax and) semantics can be modified to fill this language gap. This is not so much a question of programming language design as a contribution for better understanding the relationship between polymorphic functions, polymorphic types, and functors.

1 Introduction

Hindley-Milner polymorphism [6, 14] is the basis of the type systems of most modern functional programming languages. The computational, dynamic aspects of polymorphism are well-understood and the reader is assumed to be familiar with them. Less well-understood are the static aspects of polymorphism.

Polymorphism (for values) is introduced whenever type variables are abstracted in a *value* declaration. However, wide-spread functional languages like Standard ML [17] (SML for short) and Miranda[2] [8] have no corresponding introduction of polymorphism for *arbitrary* declarations, in particular polymorphism for modules is entirely missing. In practise, this is not a problem for languages which do not support parameterised modules, e.g. Haskell [9]. In the following we shall stick to SML, mainly because its formal definition [16] enables us to discuss the semantic issues involved; apart from that, this choice is not essential: the only essential ingredients for our approach are ML-style polymorphism and the presence of parameterised modules.

The lack of polymorphism for *structures* (modules in SML) can be illustrated by comparing the expressive powers of *functors* (parameterised modules in SML) and polymorphic functions.

* The research reported here was supported by SERC grant GR/J 07303.

[2] Miranda is a trademark of Research Software Ltd.

```
fun foldleft (f,n) nil = n
 |  foldleft (f,n) (x::xs) = foldleft (f,f(x,n)) xs;
```

The function `foldleft` is polymorphic; the type of the parameter `(f,n)` is $(\alpha * \beta \to \beta) * \beta$ for arbitrary types α and β. One can think of α and β as implicit type parameters of `foldleft`. We can try to make this explicit by turning `foldleft` into a functor:

```
functor FOLDLEFT (type A; type B; val f: A*B->B; val n: B) =
    struct
        local fun loop n' nil = n'
                |  loop n' (x::xs) = loop (f(x,n')) xs
        in    val foldleft = loop n
        end
    end;
```

The polymorphic function `foldleft` and the functor `FOLDLEFT` seem to have equal expressive power in the following sense: whenever we can write a value declaration that instantiates `foldleft` with its first argument, like

```
val foldinstance = foldleft exp;
```

then there is an equivalent instantiation of the functor `FOLDLEFT`:

```
local structure Aux =
    FOLDLEFT(type A = ... ; type B = ...
             val (f,n) = exp)
in    val foldinstance = Aux.foldleft
end;
```

The only problems here are the ellipses which have to be replaced by appropriate type expressions. Since these types can be inferred from `exp`, this does not seem to be a real problem. However, there is one, and we can observe it when considering the standard example of instantiation of `foldleft`, list reversal:

```
val reverse = foldleft (op ::,nil);
```

This is a *polymorphic* instantiation of `foldleft`: the defined function `reverse` is itself polymorphic. Hence, one might expect the appropriate type declarations in the corresponding instantiation of `FOLDLEFT` to be: `type A = 'a` and `type B = 'a list`. But it is unclear where the type variable `'a` comes from, and indeed type declarations of this form are illegal in SML (and Miranda) — type variables on the right-hand side of a type declaration have to be introduced on the left-hand side. The reason for this restriction is partly a concern about soundness, and partly lack of imagination (or desire for simplicity) on the side of the language designers.

I will show how this restriction can (safely) be eliminated, i.e. in what way the syntax and semantics of SML can be modified to support polymorphism for structures and functors.

2 Polymorphism vs. Functors

Parameterised modules in SML are called *functors*. A functor maps structures that match its interface (a *signature*) to structures. SML functors are similar to parameterised “scripts” in Miranda [8] and parameterised “modules” in ET [3].

SML functors generalise polymorphic functions in the following sense. Parameters of polymorphic functions are types (implicit) and values (explicit). Functor parameters can be (polymorphic) values, (parameterised) types, and structures. Therefore, a functor can be supplied with the arguments of a polymorphic function via its input signature. The example of `foldleft` and `FOLDLEFT` (motivated by section 9.3 in [18]) illustrates how a functor can simulate a polymorphic function, but also in which way this simulation is restricted. Apart from that, functors are *more* powerful than polymorphic functions, because the value parameters can themselves be polymorphic and because the type parameters can be parametric. Here is an example (adapted from [19]) that shows the extra power of functors.

```
signature MONAD =
    sig
        type 'a M
        val unitM: 'a -> 'a M
        val bindM: 'a M -> ('a -> 'b M) -> 'b M
    end;
functor Monad(include MONAD) =
    struct
        fun mapM f m = bindM m (unitM o f)
        fun joinM z = bindM z (fn x => x)
    end;
```

The functor `Monad` and the signature `MONAD` both use the formal parameterised type associated with `M` with different arguments, in particular `joinM` has type `('a M) M -> 'a M`. Analogously, the polymorphic function `bindM` is used with different type instances in the body of the functor. This cannot be expressed with polymorphic functions; $\lambda\omega$ is the weakest type system in Barendregt's λ-cube [1] that can express the `Monad` functor.

Our goal is to make functors properly more powerful than polymorphic functions — not so much because this lacking power is badly missed in programming practice, but in pursuit of a better understanding of the two concepts. We shall first look at ways to circumvent the restriction of polymorphic functor instantiations in the existing language.

It is still possible to define a uniform list reversal as an instance of `FOLDLEFT`, but not as a polymorphic *function* — we can define it as a *functor*:

```
functor REVERSE(type T) = FOLDLEFT(
            type A = T; type B = T list
            val f = op :: ; val n = nil);
```

Syntactic restrictions for functor instantiation make the usage of the functor REVERSE slightly awkward — it is not possible to instantiate a functor within an expression, for example as in `REVERSE(type T=int).foldleft[1,2,3]`; the functor instantiation has first to be assigned to a structure name via a structure declaration. Again this means that the polymorphic use of REVERSE within the definition of a polymorphic function is not possible, unless the polymorphic function is turned into a functor. In Miranda, the analogous situation is even worse, because each parameterised module is a file.

It should be emphasised that these restrictions on functor usage are not merely syntactical accidents; they are deliberate design decisions in the concerned languages to keep all type-checking at compile-time. See [4] for a discussion on this *phase distinction*.

The problem of polymorphic instantiations of a functor has been noticed before, for instance by Hinze in [7]. The solution suggested there is to make formal type parameters of the functor parametric. In our FOLDLEFT example, this concerns its type parameters A and B; with the so-modified functor we can define a polymorphic `reverse` by functor instantiation[3].

```
functor FOLDLEFT_1 (type 'a A; type 'b B;
                    val f: 'a A * 'a B -> 'a B; val n: 'b B) =
    struct local fun loop r nil = r
                   |  loop r (x::xs) = loop (f(x,r)) xs
           in    val foldleft = loop n
           end
    end;
local structure Aux = FOLDLEFT_1(type 'a A = 'a;
                                 type 'b B = 'b list;
                                 val f = op ::
                                 val n = nil)
in    val reverse = Aux.foldleft : 'a list -> 'a list
end;
```

The new functor FOLDLEFT_1 is not restricted to non-monomorphic applications, because type declarations like `type 'a A = int` that erase a type argument are legal in SML. Indeed, FOLDLEFT can be expressed as an instance of FOLDLEFT_1. But this approach has two snags. The type parameters for A and B have nothing to do with the functor itself: neither its interface nor its body apply A or B to anything different from a type variable. From a methodological point of view, the functor interface is therefore a bad place for introducing these type parameters. More importantly, we have not really solved yet the problem of how to get a polymorphic function, which is an instance of `foldleft`, by instantiating the corresponding functor — only the special case where polymorphism is restricted to *at most one* type parameter. For example, we cannot define the (fully) polymorphic function `map` by instantiating FOLDLEFT_1, because `map` is parametric in two type variables. Of course, we can again abstract a further variable and

[3] Some SML implementations struggle with this example, but it is perfectly legal.

define another functor FOLDLEFT_2, but each abstraction step makes the syntax of the functor (and its monomorphic instantiations) increasingly messy without solving the general problem.

We can observe the limitations of functor polymorphism more clearly in the Monad example. One of the classic examples for monads are continuation monads (also adapted from [19]):

```
structure Contin: MONAD =
    struct
        type 'a M = ('a -> Answer) -> Answer
        fun unitM a = fn c => c a
        fun bindM m k = fn c => m (fn a => k a c)
    end;
structure ContMon = Monad(open Contin);
```

Wadler suggests various choices for type Answer. We can express this in SML by turning the above piece of code into a functor with parameter Answer. However, this would enforce a new instance of Monad for every choice for Answer which is a bit of a waste. Instead, it would be more natural to replace Answer by a free type variable 'b, giving us a polymorphic continuation monad.

3 Polymorphism in SML

The example of foldleft shows how polymorphism is usually treated in SML and many related languages. The polymorphism of foldleft has been silently introduced; we can see this more clearly by mixing the explicit polymorphism of the type system $\lambda 2$ with ML code:

```
val rec foldleft = λα : ∗. λβ : ∗.
        fn (f:α × β → β,n:β) => fn ls: α list =>
        case ls of nil α => n
         | (op ::) α (x,xs) => foldleft α β (f,f(x,n)) xs
```

The $*$ is the universe of types, i.e. $\lambda\alpha : *$ denotes type variable abstraction in $\lambda 2$, see [1]. Instantiation of polymorphic values with types (application $t\tau$ of terms t to types τ in $\lambda 2$) is implicit in ML; similarly the introduction of types with variables like α for value expressions. On the level of types, application and abstraction of types are always explicit, for example in the declaration of type list:

```
datatype 'a list = nil | :: of 'a * 'a list
```

Here we have an explicit type variable 'a, its explicit abstraction on the left-hand side and an explicit type application,'a list on the right-hand side.

This syntactic difference in polymorphism has a semantic equivalent — SML uses two different notions of type variable abstraction for values and types, called *type schemes* and *type functions*. The type function $\Lambda\alpha.\alpha t$ is the semantic value of `list` (t is a *type name*, a kind of personal identification number for a type), the type scheme $\forall\alpha.\alpha t$ is the static semantic value of `nil`; type names have an arity and (constructed) types are formed by applying an n-ary type name to n types, type name application being written postfix. The difference between type functions and type schemes is that instantiation of bound type variables is always explicit for type functions and always implicit for type schemes. The semantic reason for distinguishing these two forms of abstraction is certain equivalences that apply to type schemes: for example, the type schemes $\forall\alpha\forall\beta.\tau$ and $\forall\beta\forall\alpha.\tau$ are equal, but the corresponding type functions $\Lambda\alpha\Lambda\beta.\tau$ and $\Lambda\beta\Lambda\alpha.\tau$ are different, provided α or β occurs in τ.

There is another important difference between polymorphism for values and polymorphism for types in SML: the effect of nested declarations. Instead of defining `foldleft` by direct recursion, we can exploit the fact that `f` is fixed throughout the recursion (cf. `FOLDLEFT` above):

```
fun foldleft (f,n) ls =
    let fun loop r nil = r
          |  loop r (x::xs) = loop (f(x,r)) xs
    in  loop n ls
    end;
```

The local function `loop` is monomorphic, it does not abstract type variables. We can see this by annotating the ML code with type abstractions and applications (exercise for the reader). However, the declaration of `loop` still contains (implicitly) a free type variable α for the instantiation of the list constructors `nil` and `::`. This type variable is introduced *in the context* of the declaration of `loop`.

Concerning type declarations, the rôle of nested declarations is different. Although a type declaration can occur in a context which contains free type variables (in SML; not in Miranda), all type variables occurring on its right-hand side *must* be declared on its left-hand side, they cannot come from the context. The language definition of SML imposes this restriction.

We want to lift this syntactic restriction, because it prevents us from writing the polymorphic functor instantiations. This raises the question how free type variables are introduced and eliminated, and in particular what the semantic equivalent of the elimination operation is.

4 Type Variable Declarations

Before we consider the introduction and elimination of type variables for arbitrary declarations, let us look at the semantic rule[4] that defines the corresponding operation for value declarations, i.e. that introduces type schemes for value variables — rule 17 in the definition of the static semantics of SML [16]:

[4] The rule presented here is a simplified version — I have removed the parts that deal with imperative type variables, as they have no particular significance in this context.

$$\frac{C + U \vdash \mathit{valbind} \Rightarrow \mathit{VE} \qquad \mathit{VE}' = \mathit{Clos}_C \mathit{VE}}{C \vdash \mathtt{val}_U\ \mathit{valbind} \Rightarrow \mathit{VE}' \text{ in } \mathit{Env}}$$

The non-terminal *valbind* stands for a value declaration. Sentences of the semantics of SML of the form $C \vdash \mathit{phrase} \Rightarrow \mathit{VE}$ can be read as: in the context C the syntactical phrase *phrase* gives rise to (*elaborates to* is the technical term) a variable environment VE. Variable environments bind identifiers to their type schemes; they also occur as components of general environments ($E \in \mathit{Env}$) that can contain other bindings as well. "VE' in Env" is a general environment containing the variable environment VE' but no other bindings.

$C+U$ is a context in which type variables from U are declared to be free and $\mathit{Clos}_C\mathit{VE}$ is a variable environment obtained from VE by abstracting all type variables not free in C; this abstraction introduces type schemes, pointwise for each variable bound in VE. It may be a bit surprising that the result of a type variable abstraction from a variable environment is not a mapping from types to variable environments, but another variable environment. The justification goes as follows: a (static) variable environment can be seen as a tuple of types (or rather type schemes), indexed by the bound identifiers. If we extend the type system $\lambda 2$ with binary products, the following types are "isomorphic":

$$\Pi\alpha : *. \, (T(\alpha) \times U(\alpha)) \quad \cong \quad (\Pi\alpha : *. \, T(\alpha)) \times (\Pi\beta : *. \, U(\beta))$$

The abstraction on the left-hand side abstracts a type variable from a tuple, the right-hand side is a tuple (type) of abstractions. For instance, the mapping from left to right can be given as the expression $\lambda x : P. \, (\lambda\alpha : *. \, \pi_1(x\alpha), \lambda\beta : *. \, \pi_2(x\beta))$ in $\lambda 2$, where P is the type on the left-hand side of $\cong$.

I write "isomorphic" in quotes, because they are only isomorphic in a weak sense as indicated by Di Cosmo in [2]: we have to impose a few equivalences to make the composition of both mappings equal to the identity. These equivalences are $=_{\beta\eta}$, surjective pairing, and the equation $(fx, gx) = (f, g)x$. This weak isomorphism is the justification (in SML) for performing the abstraction pointwise, i.e. for picking the type on the right.

Coming back to the mentioned semantic rule in SML, it contains another slightly mysterious bit. The subscript U in $\mathtt{val}_U$ *valbind* is the set of type variables *scoped at this value declaration*. In other words: the rule incorporates introduction and elimination of free type variables. For value polymorphism, the scoping of type variables is a minor issue as it deals only with type variables that occur explicitly in the text while type scheme polymorphism is mainly tacit and operates on implicit type variables. Explicit type variables are not introduced by an explicit declaration, but rather attached to a value declaration by a general principle. Allowing free type variables to occur in other forms of declarations raises the need for an explicit form of type variable declaration. For the purposes of this paper, I suggest `typevars` *tyvarseq* as an additional form of declaration. The corresponding rule in the static semantics is quite simple:

$$\frac{}{C \vdash \mathtt{typevars}\ \mathit{tyvarseq} \Rightarrow [\mathit{tyvarseq}] \text{ in } \mathit{Env}}$$

4.1 Abstraction: General Idea

Type variables are not bound to anything[5], so an environment simply contains a sequence of non-empty sequences of type variables, listing the free type variables in that environment. We shall see later why the semantic value of a type variable declaration is a *sequence of sequences* rather than a *set* of type variables. Semantically more interesting than introduction is elimination of free type variables. It seems natural to let type variable declarations follow the usual scoping rules for declarations and eliminate them at the end of their scope, for example by modifying the rule for local declarations accordingly:

$$\frac{C \vdash dec_1 \Rightarrow E_1 \qquad C \oplus E_1 \vdash dec_2 \Rightarrow E_2}{C \vdash \texttt{local}\ dec_1\ \texttt{in}\ dec_2\ \texttt{end} \Rightarrow abstract_C(U\ of\ E_1)(E_2)}$$

A local declaration `local` dec_1 `in` dec_2 `end` declares whatever dec_2 declares; the declarations in dec_1 are auxiliary for dec_2 and their scope ends at the keyword `end`. Thus, if dec_1 contains type variable declarations their scope also ends at the end of the local declaration.

The difference from SML's rule for local declaration is the application of the abstraction operator to E_2 in the conclusion, rather than taking E_2 itself as the result. It would follow more closely the style of the SML definition if the abstraction were expressed as $Clos_C E_2$, but there is a problem: the operation $Clos_C$ abstracts all type variables which are free in its argument and not free in C, but the order of abstraction is significant for type functions and hence for our generalised abstraction as well. I postpone the definition of *abstract*, until it is clearer which properties it should have. Consider an example that uses the feature of type variable declarations:

```
local typevars 'a
in    datatype list = nil | :: of 'a * list
      val foldleft = fn (f,n) =>
              let fun loop r nil = r
                    |  loop r (x::xs) = loop (f(x,r)) xs
              in loop n
              end
end;
```

Notice that the type variable `'a` in the declaration of `list` does *not* come from the left-hand side but from the context of the declaration. Within the local declaration, `list` is a monomorphic type and `foldleft` is only polymorphic in one type variable, its result type.

For merely pragmatic reasons, it is desirable to abstract type variables pointwise from the components of an environment. The components which matter in this respect are (i) type schemes, the semantic values of value variables, (ii) type functions, the semantic values of type constructors, and (iii) type names, the personal identification numbers of newly introduced constructed types.

[5] One can think of a type variable as an ordinary variable bound to $*$, as in $\lambda 2$.

4.2 Abstraction: Gory Details

We would certainly like the abstraction operator to behave on type schemes just as $Clos_C$ does, i.e. to increase the set of abstracted type variables of a type scheme: type abstraction introduces polymorphism. This principle already allows us to formulate `reverse` as an instance of `FOLDLEFT`:

```
local
    typevars 'a
    structure Aux =
        FOLDLEFT(type A = 'a
                 type B = 'a list
                 val f = op ::
                 val n = nil)
in  val reverse = Aux.foldleft
end
```

The only item that is subject to type variable abstraction in this example is the type of `reverse`, because its declaration is the only non-local one. Within the local declaration, `reverse` has the monomorphic type `'a list` $\rightarrow$ `'a list`, but `'a` can be abstracted at the end of the local declaration, introducing a polymorphic `reverse`.

Abstracting a type variable from a *type name* increases the arity of that type name: the arity of `list` inside the `local` declaration is 0, but it is 1 outside, as we want be able to instantiate `'a` with various types and as we have to distinguish these different instantiations semantically to preserve the soundness of the type system. We only need to abstract type variables on which a type name depends; in the example, `list` depends on `'a`, because `'a` occurs freely in its constructor environment. For simplicity, we can assume that all (new) type names depend on all abstracted type variables.

The change of arity of a type name slightly complicates abstraction in the variable environment case, as further components of the environment may contain that type name and are thus affected by such a change: an environment is like a *dependent* n-tuple and we need weak isomorphisms operating on dependent tuples (see [13] for an introduction to Σ-types), in the binary case as follows:

$$\Pi\alpha:*.\,\Sigma x:T(\alpha).\,U(\alpha,x) \quad\cong\quad \Sigma x:(\Pi\alpha:*.\,T(\alpha)).\,\Pi\beta:*.\,U(\beta,x\beta)$$

Notice that x on the left-hand and right-hand side of the $\cong$ has different arities; this corresponds to the different arities of a type name. Fortunately, this weak isomorphism exists and is similarly straightforward as in the non-dependent case.

One case remains: type variable abstraction for type functions. We could stick to the principle that type abstraction always introduces *implicit* polymorphism, so that a type function may have explicit *and* implicit parameters. In the example it would mean that `list` has an *implicit* parameter outside the `local` declaration and that it would be the task of type inference to compute it, similarly as it computes the implicit type parameter of `nil`. This is surely possible but seems rather unusual and involves a number of language design problems, e.g. its interaction with the module system or whether it would be possible to restrict implicit polymorphism in type expressions. I shall not pursue this approach here.

The alternative is to turn abstracted type variables into *explicit* parameters of a type function. Since these additional parameters are explicit, there is a corresponding effect on the syntactical level: the arities of *type constructors* change as well. In the example, the type constructor `list` has arity 1 outside the `local` declaration: it requires an argument when used in type expressions.

To be able to change the arity of local type names, it would be useful to explicitly keep track of local type names as an additional component of environments. Implementations do that anyway, and some arguments why the SML definition should also be explicit about it can be found in section 9.2 of [10].

SML already provides a mechanism to replace type names: these are the so-called *realisations* which are used for structure/signature matching. Basically, a realisation is a finite map from k-ary type names to k-ary type functions; it can be applied to various semantic objects by replacing the type names throughout the object. Realisation application can be seen as second-order substitution.

For the abstraction operation, such functions are more complicated, i.e. we need more structure for realisations, type functions etc.

- Analogously to type declarations, type functions can now contain free type variables.
- The arity of a type name (or a type function) in SML is a natural number n; it is convenient to generalise this to a sequence of natural numbers n^*, which notationally supports curried application of type constructors.
- A realisation in SML maps a k-ary type name to a k-ary type function; here, if we abstract a sequence α^* of non-empty sequences of type variables from an environment, the corresponding abstraction realisation maps k^*-ary type names to $n^* \cdot k^*$-ary type functions, where $\cdot$ is list concatenation and $n^* = \texttt{map length}\, \alpha^*$.

Let C be a fixed context and α^* be a fixed sequence of non-empty sequences of type variables and $n^* = \texttt{map length}\, \alpha^*$. An *abstraction realisation* φ is an injective map from k^*-ary type names (not in C) to $n^* \cdot k^*$-ary type names (also not in C). We need injectivity and disjointness from the type names in C in the result to preserve the soundness of the type system[6]. Because the arity of type names is not preserved, we have to redefine the application of abstraction realisations to constructed types:

$$\begin{array}{l} t \in \mathit{Dom}\, \varphi \Rightarrow \varphi(\tau^*\ t) = (\alpha^* \cdot \varphi(\tau^*))\ \varphi(t) \\ t \notin \mathit{Dom}\, \varphi \Rightarrow \varphi(\tau^*\ t) = \varphi(\tau^*)\ t \end{array}$$

where τ^* is a sequence of non-empty sequences of types and t is a type name. Furthermore the arity of type function changes, i.e. $\varphi(\Lambda\beta^*.\tau) = \Lambda\alpha^*\beta^*.\varphi(\tau)$.

[6] A technical remark: unfortunately, the *consistency* condition for semantic objects is too weak for this purpose.

Now we can define $abstract_C(\alpha^*)(E)$ as $Clos_C(\varphi E)$ where φ is an arbitrary abstraction realisation w.r.t. context C and type variables α^*, which is defined on all type names in E that are not in C. The closure operator $Clos_C$ introduces type schemes in variable (and constructor) environments.

From the language design point of view, the abstraction operator has another merit. It allows to separate type abstractions from datatype declarations. There are some reasons to enforce this separation as the only form for recursive type declarations. The static semantic rules for value declarations use the same structure, i.e. type variable abstraction is imposed after the recursion has been solved. Because of this, it is not possible to define structurally inductive functions (which pass SML's type-check) for certain recursive types.

Strengthening the static semantic rules for value declarations such that structurally inductive functions for all recursive datatypes are typable makes *typability* [1] undecidable, because this is equivalent to solving arbitrary semi-unification problems [11]. Although "undecidable" surprisingly does not imply "impractical" in this case (see [5]), a type-checker that is necessarily non-terminating for some inputs may make some people feel uncomfortable. Having abstraction and datatype declaration as two separate concepts allows to impose the same restrictions on recursive types as on recursive functions, such that any recursive type has its corresponding recursive functions and vice versa.

5 Imperative Features

Naive polymorphism is unsound in connection with certain imperative features, for example it is unsound to have updatable polymorphic variables (*references*). For this reason, SML has a second form of type variables, *imperative* type variables. Abstraction from imperative type variables is restricted to so-called non-expansive objects; non-expansiveness is a sufficient condition for preventing polymorphic references and exceptions.

Standard ML implements this restriction by modifying its abstraction operator *Clos* and making $Clos_C VE$ dependent on whether VE was derived from an expansive (value) declaration or not. A similar modification would be necessary for our generalised abstraction operator. For example, we could regard a sequence of declarations (a structure) as expansive if any of its elementary declarations is expansive; the expansive elementary declarations consist of exception declarations and expansive value declarations. An attempt to abstract imperative type variables from an environment which was derived from an expansive structure could then be regarded as an error.

But this is not the only difficulty. Since we can now also abstract type variables from type constructors, we have additional problems: the abstraction may put *applicative* type variables (unrestricted polymorphism) into places where they should not be, example:

```
functor EXCEPT(type t) =
    struct exception A of t end;
structure S = let typevars 'a in EXCEPT(type t = 'a) end;
```

The component type of an exception is not allowed to contain applicative type variables (for soundness reasons; see [15], page 42), but in the above example abstraction and module instantiation unfortunately outwit this restriction.

The straightforward solution is to supply type names with an additional "imperative" attribute (similar to the equality attribute) and to require realisations to assign only imperative types to imperative type names. The example would be ruled out, as `t` is imperative and `'a` is not. Even with imperative type variables there are problems here, because functor bodies can be expansive, as the example shows. One possible way of solving this problem is to require that formal imperative type names are mapped by a realisation to closed imperative types; in other words: to essentially restrict generalised type abstraction to the applicative case.

The presence of non-imperative type names has a number of other effects on the language.

- A type $\tau^*\ t$ is imperative if t is an imperative type name and all types in τ^* are imperative. This implies a restriction on specialisation of imperative polymorphism and the types of exception constructors.
- We have to compute an imperative attribute for newly introduced datatypes. This is completely analogous to the equality attribute, i.e. a new datatype is imperative if (roughly) all its constructors have imperative types as argument types.
- We need a feature to specify the imperative attribute of a formal type parameter of a functor, analogous to `eqtype` for the specification of the equality attribute. To keep the extension upwards-compatible with the existing language, we can take a type specification `type ty` to specify that `ty` is imperative and add a feature for the specification of (possibly) non-imperative types.

A nice side-effect of this approach is that the attributes for equality and imperativeness are treated completely analogous, i.e. they are attributes of type names and type variables. This fits very well with a Haskell-like understanding of these attributes as type classes.

Instead of this rather sophisticated approach to imperative features, one could instead employ the method suggested by Leroy in Chapter 6 of his thesis [12]. It can be roughly described by the slogan: "type variable abstraction introduces closures", i.e. (even implicit) type application forces re-evaluation. This method eliminates all soundness problems with imperative features, including the ones mentioned in this section.

6 Related Module Systems

In the following, we briefly discuss the (potential) rôle of type variable abstraction in the module systems of ET [3] and Miranda [8, 7].

6.1 ET

The (functional logic) programming language ET is the only programming language with a notion of polymorphic abstraction for types I am aware of. Because ET's module system is flat (no substructures) and because type declarations are only permitted at top level, ET has only one place for free type variables — instantiation of functors. Type variables can be locally free for a functor instantiation. In ML syntax, ET's functor instantiations all have the following shape:

```
local structure Aux =
          let typevars 'a
          in  FOLDLEFT(type A = 'a;  type B = 'a list;
                       val f = op ::; val n = nil)
          end;
in    val reverse = Aux.foldleft
end
```

The semantic operations that support such a notation in ET differ slightly from the method described above for SML. Type declarations in the functor argument are required to be non-parametric, i.e. the corresponding type functions are of arity 0. For the functor body, these types are treated like free type variables and are abstracted from the exported objects. A functor instance (with or without free type variables) instantiates these variables with actual types and restores the old arity of all type constructors, before all free type variables of the functor instance are abstracted.

The difference to the method described for SML is in sharing of types. The first abstraction does not depend on the module instance and is performed only once — an ET functor contains free type names, in contrast to SML where each functor instance makes a fresh copy of these type names. Thus, types obtained from different functor instances in ET are compatible and an ET functor application never generates new datatypes. In other words: ET functors are extensional — applied to the same arguments they deliver the same results.

6.2 Miranda

Miranda's module system has a similar structure to ET: modules are flat and types only exist on top level, but it does not support type variable abstraction from functor instances. Concerning type compatibility of functor instances, Miranda follows SML. Since the specification of parameterised types in functor interfaces is possible, ET's approach is not feasible anyway, because this would require third order type constructors and second order type variables, making type inference undecidable.

Miranda supports recursive functor instantiation, i.e. the exported objects of a functor can be used to provide it with its input parameters. It is possible to create recursive types by recursive functor instantiation. For the abstraction of type variables, here we have again the problem whether abstraction and recursion take place simultaneously, or whether abstraction is imposed *after* the recursion. An example (in ML-style syntax) should make it clear:

```
functor COPY (type t) =
    struct
        type u = t
    end;
structure rec C =
    let typevars 'a
    in  COPY(datatype t = nil | cons of 'a * 'a C.u)
    end;
```

The functor `COPY` exports the non-parameterised type `u`, which abstraction makes `u` parametric *before* it is stored in `C`, i.e. `C.u` is parametric and has to be provided with an argument even in recursive occurrences. One can also express abstraction *after* recursion by moving the declaration of `'a` outside:

```
local typevars 'a
in    structure rec C =
          COPY(datatype t = nil | cons of 'a * C.u)
end;
```

In this slight modification of the last example, `C.u` is non-parametric within the recursion, but becomes a parametric type at the end of the `local`.

7 Conclusion

Polymorphism in languages like Standard ML or Miranda is restricted because it allows to abstract type variables from declarations of values but not from declarations of types or structures. In this sense, polymorphism is not first-class. This restriction can be felt in the presence of parameterised modules, which in a strange way happen to be less expressive than polymorphic functions.

The syntactic cure is simple and — important from a language design point of view — easy to understand: introduce a new form of declaration, the explicit declaration of type variables. Such type variables are abstracted at the end of their scope. The semantic cure is a little bit more subtle, because type variable abstraction has to be defined for all possible environment components.

Such pointwise type variable abstraction from environment components is justified, for one can consider the global abstraction of types from a tuple to be isomorphic to the tuple with pointwise abstracted types. This isomorphism has already been exploited in the semantics of Standard ML and Miranda, but it becomes slightly more complicated in this setting as (general) environments have to be regarded as *dependent* tuples.

Acknowledgements

I would like to thank Bernd Gersdorf, Claudio Russo, Don Sannella and Andrzej Tarlecki and the ESOP referees for valuable discussions on this subject and feedback on an earlier version of this paper.

References

1. Hendrik P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Vol.2*, pages 117–309. Oxford Science Publications, 1992.
2. Roberto di Cosmo. Type isomorphisms in a type-assignment framework. In *19th ACM Symposium on Principles of Programming Languages*, pages 200–210, 1992.
3. Bernd Gersdorf. *Entwurf, formale Definition und Implementierung der funktional-logischen Programmiersprache ET*. PhD thesis, Universität Bremen, 1992. (mainly in German).
4. Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *17th ACM Symposium on Principles of Programming Languages*, pages 341–354, 1990.
5. Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.
6. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of AMS*, 146:29–60, 1969.
7. Ralf Hinze. *Einführung in die funktionale Programmierung mit Miranda*. Teubner, 1992. (in German).
8. Ian Holyer. *Functional Programming with Miranda*. Pitman, 1991.
9. P. Hudak, S. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell, a Non-strict, Purely Functional Language. Technical report, University of Glasgow, 1992. (also in SIGPLAN Notices 27(5), May 1992).
10. Stefan Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, University of Edinburgh, 1993.
11. A.J. Kfoury, J. Tiuryn, and P. Urcyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993.
12. Xavier Leroy. Polymorphic typing of an algorithmic language. Rapports de Recherche No. 1778, INRIA, 1992.
13. Per Martin-Löf. An intuitionistic theory of types: predicative part. In Rose and Shepherdson, editors, *Logic Colloquium 1973*, pages 73–118. North-Holland, 1975.
14. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
15. Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.
16. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
17. Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
18. Stefan Sokołowski. *Applicative High Order Programming*. Chapman & Hall Computing, 1991.
19. Philip Wadler. The essence of functional programming. In *19th ACM Symposion on Principles of Programming Languages*, pages 1–14, 1992.

This article was processed using the LaTeX macro package with LLNCS style