First-Class Polymorphism for ML

Stefan Kahrs*
University of Edinburgh
Laboratory for Foundations of Computer Science
King's Buildings, EH9 3JZ
email: smk@dcs.ed.ac.uk

Abstract

Polymorphism in ML is implicit: type variables are silently introduced and eliminated. This is fine in the core language of Standard ML, but 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 in Standard ML can only be abstracted from value declarations, but not from other forms of declarations, especially not from structure declarations. Analogous restrictions exist in other languages too, for instance Miranda. 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 [9, 18] 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. A lot of effort has been devoted to extending the computational polymorphism to imperative features, and the flood of recent publications on this subject (I recommend [15] as the ultimate reference) leads me to assure the reader that this paper is *not* yet another attempt in this direction. However, the combination of the generalised type variable abstraction suggested in this paper with weak polymorphism requires some care; this will briefly be discussed in section 5. 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 [21]

*The research reported here was supported by SERC grant GR/J 07303. An abridged version of this paper will appear in the proceedings of ESOP'94.

(SML for short) and Miranda[1] [11] 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. Clean [4] or Haskell [12]. In the following we shall stick to SML, mainly because its formal definition [20] 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.

```
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 $(\alpha * \beta \rightarrow \beta) * \beta$ 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);
```

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

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, 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 [11], parameterised "modules" in ET [6], even to "generic packages" in Ada[2] [5].

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 [22]) 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 [25]) 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 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.

[2] Ada is a trademark of the US DoD.

Our goal is to remove the "apart from" proviso, i.e. 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 [7] for a discussion on this *phase distinction*.

The problem of polymorphic instantiations of a functor has been noticed before, for instance by Hinze in [10]. 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

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

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 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 [25]):

```
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. Again, this violates the restrictions for type declarations.

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 [20]:

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

The non-terminal *valbind* stands for a value declaration. Sentences of the semantics of SML of the form $C \vdash phrase \Rightarrow VE$ 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 ($E \in Env$) 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 $Clos_C VE$ 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 (as it should be), 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. It is sufficient to consider the binary case. 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 [3]: 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

[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.

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:

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

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. A side remark: for other (static) semantic entities introduced in local declarations a corresponding abstraction is not necessary and for most not even meaningful. However, an abstraction of type names is for certain technical reasons quite desirable, see [13]. 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,

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

`list` is a monomorphic type and `foldleft` is only polymorphic in one type variable, its result type. Another difference to the first version is that the new type `list` does not *admit equality* (this means that the function `=` is not predefined for `list`) within the local declaration.

For merely pragmatic reasons, it is desirable to abstract type variables pointwise from the components of an environment, similarly as SML does for its variable environments. 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.

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 [17] 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 [13].

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:

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

[6] A technical remark: unfortunately, the *consistency* condition for semantic objects is too weak for this purpose. A stronger form of consistency would make SML signature elaboration undecidable, although this problem can be solved by other matters; see the section on signatures.

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. $\varphi(\Lambda\beta^*.\tau) = \Lambda\alpha^*\beta^*.\varphi(\tau)$. 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 [14]. Although "undecidable" surprisingly does not imply "impractical" in this case (see [8]), 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.

The inability to express recursive functions for certain recursive types is illustrated by the following example, a recursive datatype (terms in de Bruijn λ-calculus; credit to this example goes to Jim Hook) that is not expressible by "abstraction after recursion". Expanding the recursion for this type gives a non-regular infinite tree, see [2].

```
datatype 'a opt = One | An of 'a
datatype 'a term = Var of 'a | App of 'a term * 'a term |
                   Lam of ('a opt) term

fun freevars (Var x) = [x]
 |  freevars (App(t,u)) = freevars t @ freevars u
 |  freevars (Lam t) =
        foldleft(fn (One,xs) => xs |
                    (An x,xs) => x::xs, nil)(freevars t)
```

The function `freevars` does not type-check in SML, although a stronger type system could assign it the type scheme $\forall\alpha.\alpha$ `term` $\rightarrow$ α `list`. The type system of SML fails to type-check `freevars`, because `freevars` calls itself with a proper instance of this type scheme in the last clause. As an exercise, the reader may try to define the substitution function for these terms (it has type α `term` $\rightarrow$ $(\alpha \rightarrow \beta$ `term`$)$ $\rightarrow$ β `term`$)$ — it will also go beyond SML's type system.

Examples of types that need such a stronger type system are extremely rare in practice, so the suggested restriction hardly affects the ordinary SML programmer. However, this might just be a matter of educating programmers. In any case, types and values should follow the same pattern of recursion — it is a matter of good language design to only allow recursive types for which there are 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 [19], 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 [15]. 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 Signatures[7]

Module interfaces in SML are called *signatures* — a signature is like the type of a structure. Signatures are necessary as import interfaces for functors, but they can also be used to restrict the export of structures or functors.

A few design decisions in the static semantics and even the syntax for SML modules look unnatural, in particular the definitions of *consistency* and *equality-principality* in the semantics and syntactically the omission of type abbreviations in signatures. The rationale behind some of these decisions is explained in [19]; the main reason for them is apparently the objective of making signature elaboration decidable.

A signature does not introduce new types, it only contains *descriptions* of types, which provide the information about a type a functor body can exploit — quite similar to the type information of a function parameter. We can extend the feature of local free type variables to signature expressions. This allows the description of a parameterised type `type 'a t` to be decomposed into abstraction and type description:

```
local typevars 'a
in    type t
end
```

In SML, a type description (of a possibly parameterised type) specifies a type function. The elaboration algorithm has to correctly *guess* the specified type functions, which means that these guesses have to be adjusted in the presence of *sharing equations*. Sharing equations allow to specify certain types to be equal, which is useful if the types occur as components of structures (Paulson explains them quite nicely in [21]). To satisfy a sharing constraint `sharing type t=u`, the guesses for `t` and `u` have to be shared. If a type constructor is not constrained by sharing equations then the guessed type function will always be a fresh (η-expanded) type name. When a structure is successfully matched against a signature, a realisation assigns type functions to these fresh type names.

[7] Warning: this section is very SML-specific!

The SML definition does not explicitly require that all unshared type constructors are guessed to be η-expanded type names. This is just a consequence from several other definitions and restrictions, some of which have been stretched a little to achieve this property. For instance, the Commentary on SML contains in its appendix D a list of corrections of errors found in the SML Definition. One of these corrections is to replace a side condition from rule 86, that a certain *type* τ does not contain type variables, by the stricter condition that even the corresponding type expression ty does not contain type variables. These conditions differ, because there exist type functions which can throw away some of their arguments (and hence the type variables they contain), but this behaviour is not specifiable by type names. Thus, certain signature expressions would not have a principal signature if the weaker restriction were employed.

Concerning the guessing of type functions, there is a different choice: type descriptions could be restricted to only specify type *names*. If such a type name is chosen freshly then type variable abstraction as described earlier is a form of η-expansion.

This little semantic restriction has a severe (positive) impact on other parts of the semantics and syntax[8].

- We can allow type abbreviations within signature expressions, because the elaboration of sharing equations now only involves first-order (instead of second-order) unification, i.e. it is decidable and has most general solutions.

- We can strengthen the consistency property for sets of *type structures*.

- We can require each datatype description to *respect and maximise equality*, making it possible to strengthen the existence theorem of principal signatures [19]. Each principal signature is then equality-principal.

The restriction that type descriptions only specify (η-expanded) type names is not a real restriction in the presence of type abbreviations: in Standard ML, a specified type function is always guessed to be a type name, except when it is shared with a global type — but this can now more appropriately be expressed as a type abbreviation.

The consistency condition for type structures requires (roughly) that compatible types have the same value constructors. It does *not* require that these value constructors have the same types, because to check this property within signature expressions would require second-order unification. Again, this becomes decidable if we change the rôle of type descriptions as mentioned. Type names obtained from datatype declarations are always fresh — this is necessary for the soundness of SML's type system. The stronger consistency condition (when imposed throughout the elaboration process) would already guarantee soundness and a datatype declaration could be allowed to guess its type name; fresh type names would only be obtained from a principal elaboration of a structure expression.

A signature does not only specify the arities of type constructors, but also their equality attributes, i.e. under what conditions a type constructed with a described type constructor admits equality or not. While this attribute is Boolean for types, it is more

[8]The reader may find some of the following arguments rather difficult to understand, because I use some technical terms from SML's definition [20] and commentary [19]. These arguments are merely directed towards language designers and implementors, who are familiar with the SML semantics. The only significant change for a SML user is the inclusion of type abbreviations in signatures.

complicated for type functions: for an n-ary type function, it is a $\wedge$-preserving map in the Boolean algebra $2^n \to 2$, $2 = \{\bot, \top\}$ with $\bot \leq \top$. If a sharing equation in SML requires some described datatype to admit equality, then the types of its constructors have to meet this requirement. If they do not, we may have to increase the equality attribute of certain specified type functions. There is not always a most general solution to finding such an increased assignment of attributes. Here is an example that shows the problem:

```
signature S = sig
    type t; type 'a u
    datatype s = C of t u
    eqtype r
    sharing type s=r
end
```

This signature declaration does not elaborate in SML, the main difficulty being the inferred equality attributes for `t` and `u`. If `t` and `u` can be guessed to be arbitrary type functions then the example has no principal (most general) elaboration, because `u` could be guessed to be a type function like $\Lambda\alpha$.`int`, which makes `s` admit equality even if `t` does not. The SML definition solved this problem by permitting inconsistent equality attributes during signature elaboration and applying an algorithm to the elaboration result (the principal signature) which is supposed to repair the damage and deliver an "equality-principal" signature.

Under the restriction that type descriptions only specify type names, this problem goes away, because the equality attribute for an n-ary type name is always one of the two smallest (pointwise order) $\wedge$-preserving functions in $2^n \to 2$. It is possible then to maintain consistent equality attributes throughout elaboration. The above example would elaborate, with `t` and `u` both admitting equality.

A technical problem, pointed out to me by one of the ESOP referees, which arises from the restriction that guessed type functions are always η-expanded type names, is the restriction this induces on structure sharing. If a specified structure should be shared with a structure from the context, then our restriction may prohibit the sharing. Fortunately, there is a simple and very natural solution to this problem: allow structure abbreviations (ordinary structure declarations) in signatures! Thus, if a type or a structure is supposed to be shared with a *global* type or structure, we write a type or structure abbreviation. Sharing equations would only be used to express sharing between *specified* types or structures. A neat side-effect of this solution is that the so-called "cover" condition for structures comes for free. Considering the complications "cover" causes for higher-order modules (see [23]), not to mention that some SML implementations do not implement a cover-check anyway, this achievement in itself would justify the change.

7 Related Module Systems

In the following, we briefly discuss the (potential) rôle of type variable abstraction in the module systems of ET [6], Miranda [11, 10], Clean [4] and Haskell [12].

7.1 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.

7.2 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`.

7.3 Clean

Clean is another functional programming language with an ML-style type system. The module system of Clean closely resembles Modula-2 [26]; it has definition modules and implementation modules, but no notion of parameterised module.

The Modula-2 scenario of splitting a module into its interface and its implementation closely corresponds to so-called abstractions in MacQueen's original proposal for modules in ML [16]. Hence, Clean's definition modules play a similar rôle to signatures in functor interfaces; in particular both support a notion of abstract type.

From this point of view, an extension of Clean with parameterised modules should be not too difficult to design and implement; this extension should mainly affect linkage of modules, not their compilation. The main syntactic difference would be a more complicated `IMPORT` declaration, very much in the style of such declarations in ET or Miranda. This again would be the natural place to allow type variable abstraction.

7.4 Haskell

Haskell also has a flat module system and moreover no parameterised modules, so the described approach does not seem to have any applications there. However, Haskell has a rivalling module system in disguise, the so-called *classes*, see [24]. A class can be seen as a restricted form of signature, consisting of a specified type and several operations which (may) depend on this type. Declarations which depend on "classified" type

variables correspond to functors, because the declared objects can only be used in connection with instances of the particular classes; finally, class instantiations have a certain resemblance to functor applications.

Moreover, Haskell's class instantiations contain free type variables, because the replacement of the specified type of a class does not have to be a ground type. These type variables are locally free for the class instantiation, similarly to instances of parameterised modules in ET.

Altogether, this would be enough to simulate our running example, the `FOLDLEFT` functor and its polymorphic instance `reverse`, with classes and class instantiations. However, the Haskell definition [12] requires the replacement of the specified type of a class to be a constructed type (applied to distinct type variables). This excludes the instantiation of a class with a type variable, as it would be necessary for the `reverse` example.

Again, these restrictions are not merely syntactical accidents. Allowing arbitrary type expressions in class instantiations makes class membership much more difficult to check (though it is probably still decidable) and leads to additional problems such as preventing overlapping class instantiations.

8 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] F. Cardone and M. Coppo. Two extensions of Curry's type inference system. In P. Odifreddi, editor, *Logic and Computer Science*, pages 19–76. Academic Press, 1990.

[3] Roberto di Cosmo. Type isomorphisms in a type-assignment framework. In *19th ACM Symposium on Principles of Programming Languages*, pages 200–210, 1992.

[4] M.C.J.D van Eekelen, H. Huitema, E.G.J.M.H. Nöcker, M.J. Plasmeijer, and J.E.W. Smetsers. Concurrent Clean, language manual. Technical Report 92-18, University of Nijmegen, 1992.

[5] J. Ichbiah et. al. *Reference Manual of the Ada Programming Language*, 1983.

[6] Bernd Gersdorf. *Entwurf, formale Definition und Implementierung der funktional-logischen Programmiersprache ET.* PhD thesis, Universität Bremen, 1992. (mainly in German).

[7] 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.

[8] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[9] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of AMS*, 146:29–60, 1969.

[10] Ralf Hinze. *Einführung in die funktionale Programmierung mit Miranda.* Teubner, 1992. (in German).

[11] Ian Holyer. *Functional Programming with Miranda.* Pitman, 1991.

[12] 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).

[13] Stefan Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, University of Edinburgh, 1993.

[14] A.J. Kfoury, J. Tiuryn, and P. Urcyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993.

[15] Xavier Leroy. Polymorphic typing of an algorithmic language. Rapports de Recherche No. 1778, INRIA, 1992.

[16] David MacQueen. Using dependent types to express modular structure. In *13th ACM Symposium on Principles of Programming Languages*, pages 277–286, 1986.

[17] Per Martin-Löf. An intuitionistic theory of types: predicative part. In Rose and Shepherdson, editors, *Logic Colloquium 1973*, pages 73–118. North-Holland, 1975.

[18] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[19] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[20] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[21] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[22] Stefan Sokołowski. *Applicative High Order Programming*. Chapman & Hall Computing, 1991.

[23] Mads Tofte. Principal signatures for higher-order program modules. In *19th ACM Symposium on Principles of Programming Languages*, pages 189–199. ACM, 1992.

[24] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *16th ACM Symposion on Principles of Programming Languages*, pages 60–76, January 1989. Austin, Texas.

[25] Philip Wadler. The essence of functional programming. In *19th ACM Symposion on Principles of Programming Languages*, pages 1–14, 1992.

[26] Niklaus Wirth. *Programming in Modula-2*. Springer, 1982.