, , 1--13 ()

Simple Imperative Polymorphism

ANDREW K. WRIGHT * wright@cs.rice.edu
Department of Computer Science, Rice University, Houston, TX 77251-1892

Editor: Ian A. Mason

Abstract. This paper describes a simple extension of the Hindley-Milner polymorphic type discipline to call-by-value languages that incorporate imperative features like references, exceptions, and continuations. This extension sacrifices the ability to type every purely functional expression that is typable in the Hindley-Milner system. In return, it assigns the same type to functional and imperative implementations of the same abstraction. Hence with a module system that separates specifications from implementations, imperative features can be freely used to implement polymorphic specifications. A study of a number of ML programs shows that the inability to type all Hindley-Milner typable expressions seldom impacts realistic programs. Furthermore, most programs that are rendered untypable by the new system can be easily repaired.

Keywords: Continuations, functional programming, polymorphism, references, state

1. Polymorphism, Imperative Features, and Modules

The Hindley-Milner polymorphic type discipline [7], [12] is an elegant and flexible type system for functional programming languages. Many call-by-value languages include *imperative* features like references, exceptions, and continuations that facilitate concise and efficient programs. Several solutions to integrating imperative features with Hindley-Milner polymorphism in call-by-value languages have been devised [1], [3], [4], [8], [9], [11], [17], [18], [19]. These solutions range in complexity from Tofte's relatively simple method that Standard ML adopted to Talpin's sophisticated system that infers types, regions, and effects. All of these solutions assign types to all purely functional expressions that are typable by the Hindley-Milner system (henceforth called *HM-typable* expressions). However, they assign different types to imperative and functional polymorphic procedures that implement the same mathematical abstraction.

For example, in Standard ML [14] we may define a polymorphic procedure that sorts lists of any kind, given an ordering function for elements:

`val sort = fn less => fn list =>` ... *sort* ...

A functional implementation of this procedure has type:

$$\forall\alpha.\,(\alpha \rightarrow \alpha \rightarrow bool) \rightarrow (\alpha\ list) \rightarrow (\alpha\ list)$$

in the ordinary Hindley-Milner type system. An imperative implementation of `sort` that places elements of the list in a temporary reference cell or array may be more efficient or

* This research was supported in part by the United States Department of Defense under a National Defense Science and Engineering Graduate Fellowship.

more concise. But such an imperative version of `sort` has the following *imperative* types in Tofte's system [18], MacQueen's system [1], and Leroy's system [9], [11]:

$$\forall \_\alpha.\, (\_\alpha \rightarrow \_\alpha \rightarrow bool) \rightarrow (\_\alpha\ list) \rightarrow (\_\alpha\ list) \qquad (\textit{Tofte})$$
$$\forall \alpha^2.\, (\alpha^2 \rightarrow \alpha^2 \rightarrow bool) \rightarrow (\alpha^2\ list) \rightarrow (\alpha^2\ list) \qquad (\textit{MacQueen})$$
$$\forall \alpha LMNP.\, (\alpha \xrightarrow{L} \alpha \xrightarrow{M} bool) \xrightarrow{N} (\alpha\ list) \xrightarrow{P} (\alpha\ list) \qquad (\textit{Leroy})$$
$$\texttt{with}\ \{\alpha \triangleright M,\ (\alpha \xrightarrow{L} \alpha \xrightarrow{M} bool) \triangleright P\}$$

In each case, the extra annotations in the procedure's type reveal the use of imperative features in its implementation.

Revealing the imperative nature of a procedure in its type has serious consequences with a module system that separates specifications from implementations. Imperative procedures cannot be supplied as implementations for functional polymorphic specifications. In Standard ML, the following *signature* specifies the interface to a sorting module:

```
signature SORT = sig
    val sort : ∀α.(α → α → bool) → (α list) → (α list)
end
```

Only a functional version of `sort` can be supplied as an implementation for this signature. An imperative version of `sort` cannot be used because it does not have the correct type. Consequently, specifications that are to be implemented by imperative procedures must use the imperative type. The extra annotations in imperative types clutter specifications. Imperative types also restrict the applicability of procedures in ways that are peculiar to the type system and difficult for programmers to predict.

We present a simple solution to typing imperative features that sacrifices the ability to type all HM-typable expressions. In return, our solution assigns the same types to imperative and functional implementations of the same abstraction. This enables modules implementing polymorphic specifications to freely use imperative features. Based on a study of over 250,000 lines of ML code, we present empirical evidence that our type system seldom rejects realistic ML programs because of its inability to type all HM-typable expressions. Furthermore, when a program is rejected for this reason, simple syntactic changes like η-expansion usually suffice to restore typability.

The next section outlines the difficulty with incorporating imperative features in a Hindley-Milner type system and discusses previous solutions. We assume some familiarity with ordinary Hindley-Milner typing. Section 3 presents our solution and studies its impact on realistic programs. The fourth section discusses related work.

2. Polymorphism and References

We use references (*i.e.*, pointers) to illustrate the difficulty with typing imperative features in a Hindley-Milner setting, and Standard ML for examples. Our discussion applies equally well to exceptions and continuations.

In a call-by-value functional language without imperative features, we may explain the polymorphic let-expression:

$$\texttt{let val } x \texttt{ = } e_1 \texttt{ in } e_2 \texttt{ end} \tag{1}$$

as an abbreviation for the expansion:

$$(e_1\texttt{;}\ \ e_2[x/e_1]) \tag{2}$$

where $e_2[x/e_1]$ is the capture-avoiding substitution of e_1 for free x in e_2 [15]. Semantically, expressions (1) and (2) are equivalent. In the expansion, the first subexpression e_1 is evaluated and discarded to ensure that the expansion diverges when the let-expression does. Expressions (1) and (2) also have the same type. The expansion mimics polymorphism by replicating subexpression e_1. Each occurrence of e_1 in the expansion may be assigned a different type. The Hindley-Milner type system mimics typing let-expressions as their expansions without requiring a type checker to expand let-expressions.

This simple explanation of polymorphism fails in a language with imperative features because a let-expression and its expansion may not be semantically equivalent. For example, e_1 may create a reference cell that is shared at uses of x in e_2. The expansion will not capture this sharing. The following expression:

```
let val x = (ref 1) in  x := 2; !x  end
```

is not equivalent to the expansion:

```
((ref 1); (ref 1) := 2; !(ref 1))
```

In the let-expression, subexpression `(ref 1)` allocates a reference cell containing `1`. Subexpression `x := 2` replaces the contents of that cell with `2`, and `!x` extracts the cell's contents and returns `2`. The expansion, on the other hand, creates three distinct reference cells and returns `1`.

Ignoring this semantic difference when typing reference cells leads to trouble. A naive attempt to introduce references merely adds `ref`, `!`, and `:=` as primitive procedures with the following polymorphic types:

$$\begin{array}{lll}\texttt{ref} & : & \forall\alpha.\,\alpha \rightarrow (\alpha\ \mathit{ref}) \\ \texttt{!} & : & \forall\alpha.\,(\alpha\ \mathit{ref}) \rightarrow \alpha \\ \texttt{:=} & : & \forall\alpha.\,(\alpha\ \mathit{ref}) \rightarrow \alpha \rightarrow \mathit{unit}\end{array}$$

But consider the following well-known counter-example:

```
let val c = ref (fn x => x)        (3a)
in  c := (fn x => 1+x);            (3b)
    !c true                        (3c)
end                                (3d)
```

With these types for the reference cell operators, subexpression `ref (fn x => x)` in line (3a) has type $(\beta \rightarrow \beta)\ \mathit{ref}$ for any type β. Generalizing β, we obtain the polymorphic type $\forall\beta.\,(\beta \rightarrow \beta)\ \mathit{ref}$ for `c`. Line (3b) assigns the occurrence of `c` type $(\mathit{int} \rightarrow \mathit{int})\ \mathit{ref}$. This type is a valid instance of `c`'s polymorphic type obtained by instantiating β to *int*.

Line (3c) assigns the occurrence of `c` type $(bool \rightarrow bool)\ ref$, again a valid instance of `c`'s polymorphic type. Hence the let-expression as a whole is typable. But evaluating this expression leads to the *type error* `1+true`. This naive attempt to type reference cells is unsound.

In the above example, generalizing β is incorrect because β appears in the type of reference cell `c` that is shared throughout the let-expression's body. If β is not generalized, all occurrences of `c` will be forced to have the same type. Since this is impossible, the expression will be rejected. But not all uses of reference cells in a let-expression prevent generalization. If a let-expression is semantically equivalent to the expansion indicated above (2), then generalization can occur as usual. The following imperative function reverses a list in linear time [18]:

```
let val fastrev = fn list =>                                  (4)
    let val left = ref list and right = ref []
        in while !left <> [] do
               (right := hd(!left) :: !right;
                left := tl(!left));
           !right
        end
in ...
```

Reference cells `left` and `right` have type $\beta\ list\ ref$ but each use of `fastrev` in the outer let-expression's body allocates new cells. Hence β can be generalized by the outer let-expression to yield the polymorphic type $\forall\beta.\,(\beta\ list) \rightarrow (\beta\ list)$ for `fastrev`.

In general, some type variables that appear in the types of reference cells may be generalized by a let-expression and some may not. Exactly which type variables can be generalized is undecidable. A sound solution to typing reference cells must avoid generalizing type variables that appear in the types of shared reference cells.

2.1. Conservative solutions

The solutions devised to date [1], [3], [4], [8], [9], [11], [17], [18], [19] are *conservative* extensions of the Hindley-Milner type system. That is, they assign types to all HM-typable expressions. Conservative solutions require determining whether and to what degree a let-expression's binding uses imperative features. Hence these solutions record information about uses of imperative features in an expression's type.

Standard ML: Standard ML adopts Tofte's solution [18]. This solution assigns type $\forall\_\alpha.\,\_\alpha \rightarrow (\_\alpha\ ref)$ to the `ref` operator where $\_\alpha$ is an *imperative type variable*. Imperative type variables indicate values that may be placed in reference cells. Imperativeness is contagious: whenever a value is stored in a reference cell, any variables in the value's type become imperative. Imperative variables are only generalized by let-expressions when the binding has a syntactic shape which guarantees that it creates no new reference cells. In Standard ML the `fastrev` function defined above has type $\forall\_\alpha.\,(\_\alpha\ list) \rightarrow (\_\alpha\ list)$.

Weak Types: A system proposed by MacQueen has been used by Standard ML of New Jersey for several years [1]. Two approximations to it have recently been formalized by

Hoang, Mitchell, and Viswanathan [8] and Greiner [4]. These methods extend Tofte's method by assigning *weakness* numbers to type variables. The weakness of a type variable indicates how many arguments must be supplied to a curried procedure before it allocates a reference containing that type variable. For example, an imperative implementation of `sort` (from Section 1) has type $\forall\alpha^2.\,(\alpha^2 \rightarrow \alpha^2 \rightarrow bool) \rightarrow (\alpha^2\ list) \rightarrow (\alpha^2\ list)$. This procedure must be applied to two arguments before it allocates a cell containing a value of type α. Weak types allow partial applications of imperative polymorphic procedures that are rejected by Tofte's system.

Closure Typing: Leroy and Weis [9], [11] observed that it is only necessary to prohibit generalization of type variables that appear in the types of cells reachable after the binding has been evaluated (*i.e.*, cells that would not be reclaimed by garbage collection at this point). As cells may be reachable through the free identifiers of closures, their system records the types of the free identifiers of a procedure in the procedure's type. Therefore an imperative implementation of `sort` has type $\forall\alpha LMNP.\,(\alpha \xrightarrow{L} \alpha \xrightarrow{M} bool) \xrightarrow{N} (\alpha\ list) \xrightarrow{P} (\alpha\ list)$ `with` $\{\alpha \triangleright M,\ (\alpha \xrightarrow{L} \alpha \xrightarrow{M} bool) \triangleright P\}$. Although Leroy's original closure typing system [11] did not type all HM-typable expressions, his dissertation [9] corrects this oversight.

Damas: Damas proposed one of the earliest systems for typing references [3]. His system assigns both a type and a set Δ to each expression. The set Δ is a finite set of the types of cells that may be allocated by evaluating the expression. Polymorphic procedure types are augmented by a similar set Δ on the outermost arrow that indicates the types of cells the procedure may allocate when it is applied. Hence an imperative version of `sort` has type $\forall\alpha.\,(\alpha \rightarrow \alpha \rightarrow bool) \rightarrow (\alpha\ list) \xrightarrow{\Delta} (\alpha\ list)$ where $\Delta = \{\alpha\}$.

Effects: Several systems for typing references based on effect inference have been proposed. A system proposed by the author extends Damas's system to attach effect sets (Δ) to all function type arrows [19]. This system assigns the type $\forall\alpha\varsigma_1\varsigma_2\varsigma_3.\,(\alpha \xrightarrow{\varsigma_1} \alpha \xrightarrow{\varsigma_2} bool) \xrightarrow{\varsigma_3} (\alpha\ list) \xrightarrow{\alpha\varsigma_1\varsigma_2\varsigma_3} (\alpha\ list)$ to an imperative version of `sort`. A more sophisticated system devised by Talpin and Jouvelot infers types, effects, and effect regions for expressions [17].

3. A Simple Solution

In conservative solutions, the need to identify uses of imperative features in an expression's type stems from a desire to admit all HM-typable expressions. To assign the same types to imperative and functional implementations of the same abstraction, we must sacrifice this ability.

3.1. Limiting polymorphism to values

Our solution limits polymorphism to let-expressions where the binding is a syntactic value. That is, the expression:

```
let val x = e1 in e2 end
```

assigns a polymorphic type to x only if e_1 is a syntactic value. If e_1 is not a value, all uses of x in e_2 must have the same type. The precise definition of *syntactic value* is flexible. For ML, we take syntactic values to be constants, variables, λ-expressions, and constructors applied to values. (The `ref` operator is not considered a constructor for this purpose.) Since the evaluation of a syntactic value cannot cause any side effects, it is safe to generalize type variables in the same manner as the Hindley-Milner system when e_1 is a syntactic value. Since the determination of when to generalize depends solely on the syntactic shape of e_1 and not on its type, no special annotations are required in types. Imperative and functional implementations of the same abstraction have the same type.

References, exceptions, and continuations fit smoothly into this framework. The operators for references have the polymorphic types:

$$
\begin{array}{lll}
\texttt{ref} & : & \forall\alpha.\, \alpha \rightarrow (\alpha\ \mathit{ref}) \\
\texttt{!} & : & \forall\alpha.\, (\alpha\ \mathit{ref}) \rightarrow \alpha \\
\texttt{:=} & : & \forall\alpha.\, (\alpha\ \mathit{ref}) \rightarrow \alpha \rightarrow \mathit{unit}
\end{array}
$$

Since an application of `ref` is not a value, expressions that create reference cells are not generalized. Counter-example (3) from Section 2 is correctly rejected. As λ-expressions are values, references can be freely used in procedure bodies without inhibiting polymorphism. The `fastrev` function (4) from Section 2 has the polymorphic type $\forall\alpha.\, (\alpha\ \mathit{list}) \rightarrow (\alpha\ \mathit{list})$.

A sound type system for continuations must not allow continuations to be polymorphic [6], [20]. That is, in an expression like:

```
let val x = callcc (fn k => ...)
in  ...
```

`x` must not be assigned a polymorphic type. (Consider the explanation of let-expressions as abbreviations. The expansion replicates subexpression `callcc (fn k => ...)` throughout the let-expression's body and hence has a different meaning from the let-expression.) With polymorphism limited to values, Standard ML of New Jersey's `callcc` operator can be assigned the polymorphic type $\forall\alpha.\, (\alpha\ \mathit{cont} \rightarrow \alpha) \rightarrow \alpha$. Identifier `x` in the example above is not assigned a polymorphic type because the expression `callcc (fn k => ...)` is not a value. Similarly, exception types do not need any special restrictions.

We can establish soundness for our solution by showing that it is isomorphic to a restriction of Tofte's solution. Recall that Tofte's system has both *imperative* and *applicative* type variables and two rules for let-expressions. The rule for *non-expansive* bindings (*i.e.*, syntactic values) generalizes both kinds of type variables. The rule for *expansive* bindings (*i.e.*, non-values) generalizes only applicative type variables. If we remove applicative type variables from the system so that all types must use imperative type variables, the rule for expansive bindings will never generalize any type variables. Hence removing applicative type variables from Tofte's system yields a system isomorphic to ours.[1] The existing proofs of type soundness for Tofte's system [18], [20] establish type soundness for ours.

We can easily establish that our system possesses a type inference algorithm which finds *principal* types [2]. The algorithm is the ordinary Hindley-Milner type inference algorithm run after a simple translation. The translation merely expands a let-expression `let val` x `=` e_1 `in` e_2 `end` where e_1 is not a syntactic value to the equivalent expression `((fn` x `=>` e_2`)` e_1`)`. The translation prevents the type inference algorithm from assigning a polymorphic type to x.

3.2. Possible consequences

With polymorphism limited to values, some non-value expressions that have polymorphic types in the Hindley-Milner system are no longer polymorphic. There are three cases: expressions that never return, expressions that *compute* polymorphic procedures (as opposed to λ-expressions that just *are* procedures), and expressions that compute polymorphic data structures. We examine each in turn.

3.2.1. Expressions that never return

Expressions that never return arise in functional programs only as divergent computations. In a language with exceptions and continuations, expressions that signal exceptions and throw to continuations also do not return. Such expressions seldom appear in let-expression bindings because they yield no useful value. Hence not assigning polymorphic types to these expressions impacts few realistic programs. Nevertheless, we can extend our type system to accommodate some of these expressions. Many common expressions that never return have type α. It is safe to generalize such a type to $\forall\alpha.\,\alpha$ regardless of whether the expression is a value (provided that α is not free in the type environment, of course).

3.2.2. Expressions that compute polymorphic procedures

When a let-expression's binding computes a polymorphic procedure, the computation may be purely functional or may exercise imperative features. Suppose the computation is functional, as in the following example:

```
let val f = (fn x => x)(fn y => y)
in  f 1; f true  end
```

With Hindley-Milner typing, this expression is typable because the identity procedure `f` has polymorphic type $\forall\alpha.\,\alpha \to \alpha$. But with polymorphism limited to values, `f` is not assigned a polymorphic type. Since the body uses `f` with two different types, the above expression is not typable. Functional computations of polymorphic procedures typically arise in realistic ML code as uses of the compose operator `o` or as partial applications of curried procedures like `map`.

When the computation of a polymorphic procedure is functional, we can easily restore polymorphism by η-expanding the binding, as with the example above:

```
let val f = fn z => (fn x => x)(fn y => y)z
in  f 1; f true  end
```

Since the binding is now a value, `f` is assigned polymorphic type $\forall\alpha.\,\alpha \to \alpha$. Some care is needed in η-expanding bindings because this transformation can affect the algorithmic behavior of the program. The polymorphic procedure is now recomputed each time it is used.

Suppose the computation of a polymorphic procedure exercises imperative features. Then η-expansion may not be possible without altering the semantics of the program. For example, the following procedure `mkCountF` takes a procedure `f` as argument and constructs a procedure `f2` that behaves like `f` but also counts the number of times it is called:

```
val mkCountF = fn f =>
    let val x = ref 0
        val f2 = fn z => (x := !x + 1; f z)
    in  counter := x;
        f2
    end
```

The integer reference cell `x` that counts calls to `f2` is exported by assignment to the global variable `counter`. When `mkCountF` is applied to a polymorphic procedure like `map`:

```
val map2 = mkCountF map
```

the resulting procedure `map2` is not polymorphic because `mkCountF map` is not a value. Restoring polymorphism by η-expansion does not work because it causes a new counter to be allocated each time `map2` is applied. If `map2` must be polymorphic, potentially awkward global changes to the program are required. Leroy gives several other examples where it may be desirable to use imperative features in computing a polymorphic procedure [10].

3.2.3. Expressions that compute polymorphic data structures

Expressions that compute polymorphic data structures also require global program modifications to restore polymorphism, whether the computation is functional or imperative. For example, the following expression computes the polymorphic empty list:

```
val empty = (fn x => x) []
```

The Hindley-Milner system assigns the polymorphic type $\forall\alpha.(\alpha\ \mathit{list})$ to `empty`. But with polymorphism limited to values, `empty` cannot be assigned a polymorphic type. Furthermore, there is no analog to η-expansion that can be used to restore polymorphism to polymorphic data structures even when the computation is functional.

3.3. *Practical impact*

Limiting polymorphism to values is practical only if the inability to compute polymorphic procedures and data structures seldom impacts real programs. To determine how often this problem might arise, we modified Standard ML of New Jersey to use our type system. We gathered an extensive collection of ML programs and compiled them with the modified compiler (see Table 1). We found that most ML programs either satisfy the restriction of polymorphism to values already, or they can be modified to do so with a few η-expansions. In other words:

1. *Realistic ML code seldom computes polymorphic procedures or data structures. Furthermore,*

2. *When polymorphic procedures are computed, the computation is almost always functional.*

The only non-functional computations of polymorphic procedures we found were several uses in the New Jersey compiler of the unsafe procedure `c_function` that performs dynamic linking. The only computations of polymorphic data structures we found were a construction of the polymorphic empty vector in the Edinburgh ML Library and two constructions of polymorphic events in eXene. All were simple to fix. In no case did the modifications cause any detectable difference in performance.

Reppy's Concurrent ML implementation [16] illustrates the benefit of assigning the same types to functional and imperative procedures. Concurrent ML makes extensive use of Standard ML of New Jersey's first-class continuations to implement threads. To avoid assigning weak types to several of Concurrent ML's procedures, Reppy's implementation uses a version of `callcc` with type $\forall\alpha.\,(\alpha\ cont \to \alpha) \to \alpha$. This type for `callcc` is unsafe in the New Jersey compiler's weak type system; the correct type is $\forall\alpha^1.\,(\alpha^1\ cont \to \alpha^1) \to \alpha^1$. Reppy justifies the use of the unsafe type for `callcc` by a manual proof of type soundness for Concurrent ML. However, with polymorphism limited to values, $\forall\alpha.\,(\alpha\ cont \to \alpha) \to \alpha$ is the correct type for `callcc`. The troublesome procedures are automatically assigned the desired polymorphic types and no separate soundness proof is necessary.

3.4. *Integration with Standard ML's modules*

In the following structure (implementation module):

```
structure Foo = struct
    val flatten = map hd
end
```

the application `map hd` has type $(\alpha\ list\ list) \to (\alpha\ list)$. With polymorphism limited to values, α is not generalized and is free in the type of the structure. Standard ML does not allow the type of a structure to contain free type variables, hence this code is rejected.[2] A

Table 1. Practical impact of limiting polymorphism to values

Program	Size in Lines	Features Used References Exceptions Continuations	Changes Required
Standard ML of New Jersey (version 93)	62,100	R E C	4 η-expansions 4 casts in unsafe bootstrap code
SML/NJ Library	6,400	R E C	none
ML Yacc	7,300	R E C	2 η-expansions 2 η-expansions in generated code
ML Lex	1,300	R E C	none
ML Twig	2,200	R E	none
ML Info	100	E	none
Source Group (version 3.0)	8,100	R E C	none
Concurrent ML (John Reppy)	3,000	R E C	1 η-expansion added `never` for eXene
eXene X window toolkit (Reppy and Gansner)	20,200	R E	6 η-expansions 1 `(choose [])` changed to `never` 1 declaration moved
Edinburgh ML Library	15,400	R E	1 η-expansion 1 `(vector [])` changed to `#[]`
ML Kit Interpreter	38,000	R E	5 η-expansions
Isabelle Theorem Prover (version 92)	18,600	R E	2 η-expansions
Hol90 Theorem Prover (version 90.5)	83,100	R E	4 η-expansions
Lazy Streams (Thomas Breuel)	200	R E C	none
Version Arrays (Thomas Breuel)	100	R E	none
Doubly Linked Lists (Olin Shivers)	400	R E	2 η-expansions
3d Wireframe Graphics (Olin Shivers)	2,200	R E	none
Hilbert (Thomas Yan)	500	R	1 type declaration
Grobner Basis (Thomas Yan)	1,000	R E	3 η-expansions

structure like this one must be fixed by η-expansion if `flatten` is to have polymorphic type. Alternatively, a *type constraint* can be used to instantiate the free type variable if `flatten` is needed for only one specific type:

```
structure Foo = struct
    val flatten : int list list -> int list
                = map hd
end
```

Existing ML code frequently combines structures with a signature that constrains the types of their definitions:

```
signature FOO = sig
    val flatten : int list list -> int list
end

structure Foo : FOO = struct
    val flatten = map hd
end
```

The *signature constraint* “`: FOO`” constrains `flatten` to have a type with no free type variables. Unfortunately, Standard ML still insists that `structure Foo` have a closed type before it is constrained to match `signature FOO`. Rather than force programmers to add redundant type constraints to such structures, we permit free type variables of a structure to be defined by a signature constraint (or functor result constraint). In the absence of a signature constraint, structures with free type variables are still rejected.[3]

4. Related Work

Several authors [5], [10] have suggested using call-by-name semantics for let-expressions in order to combine imperative features and polymorphism. That is, in the expression:

`let val` $x = e_1$ `in` e_2 `end`

the subexpression e_1 is not evaluated until it is needed, and it is re-evaluated at each use of x in e_2. Adapting this solution to ML would involve introducing two syntactically different forms of let-expressions: a polymorphic call-by-name form, and a non-polymorphic call-by-value form. While this solution would allow imperative procedures to have polymorphic types, it would drastically alter the call-by-value nature of ML. When call-by-name let-expressions are desired, they can be simulated in our call-by-value system by introducing dummy abstractions and applications. The expression:

`let val` x `= fn _ =>` e_1 `in` e_2`[`x`/(`x `())] end`

simulates `let name` x `=` e_1 `in` e_2 `end`, where e_2`[`x`/(`x `())]` denotes the substitution of `(`x `())` for free x in e_2.

5. Conclusion

Limiting polymorphism to values yields a simple type system that smoothly incorporates imperative features. The restriction of polymorphism to values is seldom a hindrance in realistic programs. In return, this restriction enables functional and imperative implementations of the same abstraction to be assigned the same types.

A patch for Standard ML of New Jersey (Version 0.93) that limits polymorphism to values and eliminates weak types is available by anonymous FTP from `cs.rice.edu` in file `public/wright/vsml.93.tar.Z`. The related file `vsml.tools` describes the required changes for important tools like ML Yacc and Concurrent ML.

Acknowledgments

I am indebted to Bruce Duba and the referees for their comments and suggestions. I thank Thomas Breuel, Olin Shivers, and Thomas Yan for contributed programs.

Notes

1. This observation is due to Stefan Kahrs.
2. See rules 100-102 of the Definition [14] and the footnote on page 55 of the Commentary [13].
3. This extension to ML's module system appears to be sound [Bob Harper, personal communication, February 1993]. It may be possible to allow free type variables even in the absence of a signature constraint, but we have not investigated this more flexible extension.

References

1. ''Standard ML of New Jersey release notes (version 0.93),'' AT&T Bell Laboratories (November 1993).
2. Damas, L. M. M. *Principal Type Schemes for Functional Programs*, In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages* (January 1982) 207--212.
3. Damas, L. M. M. *Type Assignment in Programming Languages*, PhD thesis, University of Edinburgh (1985).
4. Greiner, J. ''Standard ML weak polymorphism can be sound,'' Technical Report CMU-CS-93-160R, Carnegie Mellon University (September 1993).
5. Harper, R. and M. Lillibridge. ''Explicit polymorphism and CPS conversion,'' In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages* (January 1993) 206--219.
6. Harper, R., B. F. Duba, and D. MacQueen. ''Typing first-class continuations in ML,'' *Journal of Functional Programming 3*, 4 (October 1993), 465--484.
7. Hindley, R. ''The principal type-scheme of an object in combinatory logic,'' *Transactions of the American Mathematical Society*, 146 (December 1969) 29--60.
8. Hoang, M., J. Mitchell, and R. Viswanathan. ''Standard ML-NJ weak polymorphism and imperative constructs,'' In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science* (June 1993) 15--25.
9. Leroy, X. *Typage polymorphe d'un langage algorithmique*, PhD thesis, L'Universite Paris 7 (1992).
10. Leroy, X. ''Polymorphism by name for references and continuations,'' In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages* (January 1993) 220--231.
11. Leroy, X. and P. Weis. ''Polymorphic type inference and assignment,'' In *Proceedings of the 18th Annual Symposium on Principles of Programming Languages* (January 1991) 291--302.

12. Milner, R. ''A theory of type polymorphism in programming,'' *Journal of Computer and System Sciences*, 17 (1978) 348--375.
13. Milner, R. and M. Tofte. *Commentary on Standard ML*, MIT Press, Cambridge, Massachusetts (1991).
14. Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML*, MIT Press, Cambridge, Massachusetts (1990).
15. Ohori, A. ''A simple semantics for ML polymorphism,'' In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (September 1989) 281--292.
16. Reppy, J. H. *Higher-order Concurrency*, PhD thesis, Cornell University (1991).
17. Talpin, J.-P. and P. Jouvelot. ''The type and effect discipline,'' In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science* (June 1992) 162--173.
18. Tofte, M. ''Type inference for polymorphic references,'' *Information and Computation*, 89, 1 (November 1990) 1--34.
19. Wright, A. K. ''Typing references by effect inference,'' In *Proceedings of the 4th European Symposium on Programming*, Springer-Verlag Lecture Notes in Computer Science 582 (1992) 473--491.
20. Wright, A. K. and M. Felleisen. ''A Syntactic Approach to Type Soundness,'' Technical Report 91-160, Rice University (April 1991). To appear in *Information and Computation*, 1994.