# Polymorphism for Imperative Languages without Imperative Types

Andrew K. Wright*

Rice University Technical Report TR93-200
February 18, 1993

**Abstract**

The simple and elegant Hindley/Milner polymorphic type discipline is the basis of the type system of Standard ML, but ML's imperative features are a blight on this otherwise clean landscape. Polymorphism and imperative features cannot freely coexist without compromising type safety, hence Standard ML assigns *imperative* types of limited polymorphism to procedures that use references, exceptions, or continuations. Several other solutions exist, but all introduce new kinds of types that complicate the type system, contaminate module signatures, and violate abstraction by revealing the pure or imperative nature of a procedure in its type.

We propose a seemingly radical alternative: by restricting polymorphism to values, imperative procedures have the *same* types as their behaviorally equivalent functional counterparts. Although the resulting type system does not accept all expressions typable in the purely functional sublanguage, this limitation is seldom encountered in practice. The vast majority of ML code already satisfies the restriction of polymorphism to values, and simple syntactic modifications fix the few non-conforming programs.

## 1 The Problem with Polymorphic References

The simple and elegant Hindley/Milner polymorphic type discipline [7] is the basis of the type system of Standard ML [9], but ML's imperative features are a blight on this otherwise clean landscape. Polymorphism and imperative features like references cannot freely coexist in a call-by-value language without compromising type safety. The following oft-repeated example illustrates why:

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

The `let`-expression binds `c` to a reference cell initially containing the identity procedure `(fn x => x)`. Hence the expression `ref (fn x => x)` has type $(\alpha \rightarrow \alpha)\,ref$ for some type $\alpha$. If both polymorphism and references are unconstrained, $\alpha$ can be any type, and `c` has polymorphic type $\forall\alpha.\,(\alpha \rightarrow \alpha)\,ref$. The body of the `let`-expression then replaces the contents of the cell with an integer procedure that increments its argument. In this assignment expression, the occurrence of `c` has type $(int \rightarrow int)\,ref$, which is a valid instance of `c`'s polymorphic type, obtained by substituting $int$ for $\alpha$. Subsequently,

*Department of Computer Science, Rice University, Houston, TX 77251-1892, or wright@cs.rice.edu. This research was supported in part by the United States Department of Defense under a National Defense Science and Engineering Graduate Fellowship.

the contents of `c` are extracted (by the dereference operator `!`) and applied to `true`. In typing the dereference expression, the second occurrence of `c` has type $(bool \rightarrow bool)\ ref$, and the expression as a whole has type *bool*. However, evaluating this expression results in an attempt to add `1` to `true`. By typing this expression, the type system has failed to prevent a type error.

The problem is that in typing the expression:

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

if evaluating $e_1$ causes side effects, type variables involved in these side effects must not be generalized in assigning a type to $x$ [14]. The existing solutions attempt to determine when $e_1$ will cause undesirable side effects by recording information about potential side effects in an expression's type [1, 3, 4, 6, 13, 14, 15]. For example, Standard ML assigns the type $\forall \_\alpha \_\beta. (\_\alpha \rightarrow \_\beta) \rightarrow \_\alpha\ list \rightarrow \_\beta\ list$ to the following imperative version of `map`:

```
fun imperativeMap f l =
    let val a = ref l and b = ref nil
    in
        while not (null (!a)) do
            (b := f (hd (!a)) :: (!b);
             a := tl (!a));
        rev (!b)
    end
```

The *imperative type variables* $\_\alpha$ and $\_\beta$ indicate that values of this type are placed in reference cells. Unfortunately, these solutions violate abstraction by revealing the pure or imperative nature of a procedure in its type. This violation of abstraction has serious consequences for the construction of modular programs. Although `imperativeMap` is behaviorally equivalent to the usual purely functional definition of `map`, `imperativeMap` cannot be supplied as an implementation for a module signature specifying the pure type $\forall \alpha\beta. (\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta\ list$.

## 1.1 Polymorphism for Values Only

The need to identify potential side effects in an expression's type stems from a desire to admit all expressions typable in the purely functional sublanguage. The existing solutions all yield a polymorphic type when the bound expression $e_1$ does not use imperative features.[1] We propose a simple alternative. The expression:

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

assigns $x$ a polymorphic type only if the bound expression $e_1$ is a syntactic value, *i.e.*, a constant, variable, or $\lambda$-expression. Since the evaluation of a value cannot cause any side effects, it is not necessary to assign special types to expressions that use references, exceptions, or continuations. Imperative procedures like `imperativeMap` have the same types as their behaviorally equivalent functional counterparts.

Restricting polymorphism to values means that some non-value expressions lose polymorphism. There are only two such kinds of expressions: expressions that compute polymorphic procedures, and expressions that diverge. Expressions that diverge are uninteresting,but what about expressions that compute polymorphic procedures? Let us examine this limitation according to whether the computation is purely functional or uses imperative features.

Suppose the bound expression involves purely functional computation, like the following example:

[1] Leroy's original closure typing system [6] did not type all such functional expressions. His dissertation [4] corrects this oversight.

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

With pure Hindley/Milner typing, this expression would be typable by assigning to `f` the polymorphic type $\forall\alpha.\, \alpha \rightarrow \alpha$. But with polymorphism restricted to values, `f` is not polymorphic, and this expression is rejected. Expressions like this typically arise in realistic ML code as uses of the compose operator (`o`), or as partial applications of curried procedures, like `map`. However, polymorphism can always be restored by the simple syntactic device of $\eta$-expansion:[2]

```
let val f = fn z => (fn x => x)(fn y => y) z
in  ...
```

Since the bound expression is now a value, it can be assigned a polymorphic type. So when polymorphic procedures are computed in a purely functional manner, restricting polymorphism to values necessitates minor syntactic modifications to recover polymorphism.

Now let us consider actively computing a polymorphic procedure, where the computation uses imperative features. For example, the procedure `makeCountFun` takes a procedure `f` as argument, and returns both a counting version of `f` and a procedure `rd` that checks the counter:

```
fun makeCountFun f =
    let val x = ref 0
        fun f' z = (x := !x + 1; f z)
        fun rd () = ! x
    in (f',rd)
    end
```

When applied to a polymorphic procedure like `map`:

```
let val (map',rdmap) = makeCountFun map
in  ...
```

the resulting procedure `map'` should be polymorphic. Indeed, it is polymorphic in Standard ML's type system and the other systems for combining references and polymorphism, but with polymorphism restricted to values, `map'` is not polymorphic. Leroy gives several other academic examples of computing polymorphic procedures [5]. Determining whether the result of a computation can be polymorphic involves determining whether the computation has visible side effects. This problem is undecidable, hence the existing systems for typing references use different techniques to achieve conservatively approximate solutions. These solutions are generally incompatible [13, 15], and determining whether an expression will yield a polymorphic type requires sophisticated reasoning processes that are usually beyond the programmer's capability. Perhaps because of the difficulty of predicting these type systems, programmers avoid computing polymorphic procedures in an imperative manner. We claim that in practice such cases are extremely rare.

That unrestricted types for imperative features can be obtained by restricting polymorphism to values is not an astonishing idea. What is surprising is how well this idea works in practice. In a sample of realistic ML programs, we found that:

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

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

Hence most ML programs either already satisfy the restriction of polymorphism to values, or can be modified to do so with a few $\eta$-expansions.

[2] In ML, exactly 1 $\eta$-expansion is necessary, since all procedures take exactly 1 argument.

| | |
|---|---|
| (*Var*) | $\Gamma \vdash x : \tau \;\text{ if }\; \Gamma(x) \succ \tau$ |
| (*Fun*) | $\dfrac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$ |
| (*App*) | $\dfrac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$ |
| (*Let*) | $\dfrac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \mathit{Close}(\tau_1, \Gamma)] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$ |

$$\begin{array}{rcl} \mathit{Close}(\tau, \Gamma) & = & \forall \alpha_1 \ldots \alpha_n . \tau \\ \text{where } \{\alpha_1, \ldots, \alpha_n\} & = & \mathit{FTV}(\tau) \setminus \mathit{FTV}(\Gamma) \end{array}$$

Figure 1: Pure Type Inference Rules

### 1.2 Outline

In the following section, we give a brief formal characterization of Hindley/Milner polymorphism for a purely functional language. Section 3 adds references to this language, presents the type system that restricts polymorphism to values, and discusses some practical extensions for Standard ML. We also summarize the other methods of combining polymorphism and imperative features. Section 4 illustrates the effectiveness of limiting polymorphism to values by discussing our implementation.

## 2 Pure Hindley/Milner Polymorphism

Let us define an idealized version of ML with the following abstract syntax:

| | |
|---|---|
| (*Expressions*) | $e ::= v \mid (e_1\ e_2) \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ |
| (*Values*) | $v ::= x \mid \lambda x.e$ |

The meta-variable $x$ ranges over a countably infinite set of *identifiers*. Programs are expressions that are closed with respect to an initial environment that provides bindings for the program's free variables.

Programs are evaluated according to the usual semantics for a call-by-value language [10]. The let-expression is semantically equivalent to the expression $((\lambda x.e_2)\ e_1)$.

Types for this language are defined as follows:

| | |
|---|---|
| (*Types*) | $\tau ::= \mathit{int} \mid \mathit{bool} \mid \mathit{unit} \mid \tau_1 \rightarrow \tau_2 \mid \alpha$ |
| (*Type Schemes*) | $\sigma ::= \forall \alpha^* . \tau$ |

where the meta-variable $\alpha$ (or $\beta$) ranges over a countably infinite set of *type variables*. *Type schemes* denote polymorphic types. A type $\tau$ is an instance of type scheme $\sigma = \forall \alpha_1 \ldots \alpha_n . \tau'$, written $\sigma \succ \tau$, if there exists a substitution $S$ (mapping type variables to types) with domain $\{\alpha_1, \ldots, \alpha_n\}$ such that $S\tau' = \tau$. We identify the type scheme $\forall . \tau$ with the type $\tau$.

The type system is a deductive proof system that assigns types to expressions. A *type judgment* $\Gamma \vdash e : \tau$ for $e$ is the conclusion of a deduction constructed according to the rules in Figure 1. The judgment $\Gamma \vdash e : \tau$ reads "in type environment $\Gamma$, expression $e$ has type $\tau$". Type environments are mappings from identifiers to type schemes; the notation $\Gamma[x \mapsto \sigma]$ denotes the extension or update of $\Gamma$ at $x$ to $\sigma$; the function FTV yields the free type variables of a type or type environment. Programs are typed in an initial type environment $\Gamma_0$ that provides types for the primitive operators.

# 3 Polymorphism and References

Standard ML includes reference cells as first-class values. We model reference cells by including the operators ref, !, and := in the initial environment. When applied to a value, the ref operator creates a reference cell containing that value; applying ! to a reference cell extracts the contents of the cell; and := changes the contents of a cell.

## 3.1 Polymorphism for Values

To define a type system that allows polymorphism for values only, we replace the inference rule for let-expressions in Figure 1 with the following rules:

$$(\mathit{Let\text{-}Val}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \mathit{Close}(\tau_1, \Gamma)] \vdash e_2 : \tau_2 \quad e_1 \in \mathit{Values}}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$(\mathit{Let\text{-}Exp}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2 \quad e_1 \notin \mathit{Values}}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

We introduce a type constructor for reference cells containing values of type $\tau$:

$$\tau ::= \ldots \mid \tau\ \mathit{ref}$$

and types for the reference operators:

$$\begin{array}{rcl} \Gamma_0(\mathsf{ref}) & = & \forall\alpha.\, \alpha \to \alpha\ \mathit{ref} \\ \Gamma_0(!) & = & \forall\alpha.\, \alpha\ \mathit{ref} \to \alpha \\ \Gamma_0(:=) & = & \forall\alpha.\, \alpha\ \mathit{ref} \to \alpha \to \mathit{unit} \end{array}$$

The resulting type system assigns the same type to behaviorally equivalent procedures, regardless of whether their implementation uses references.

For Standard ML, it is convenient to extend the definition of values to include constructions, *i.e.*, applications of constructors to values. This permits declarations like:

```
val empty = (nil,nil)
```

to be polymorphic. Primitive and user-defined data constructors already have special status in ML for pattern matching, so this extension seems reasonable. Note that `ref` is not considered a constructor in this context.

One other extension is useful in practice. The definition of Standard ML does not allow the type of a value inside a structure to contain an unbound type variable. In Standard ML, this can only occur when the unbound type variable is imperative. For example, in the following structure:

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

`flatten` has type $\_\alpha\ \mathit{list}\ \mathit{list} \to \_\alpha\ \mathit{list}$ where $\_\alpha$ cannot be generalized, so this structure is rejected.[3] Although this situation seldom arises, a similar situation occurs more often with polymorphism for values. In this structure:

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

[3] See rules 100-102 of the Definition [9], and the footnote on page 55 of the Commentary [8].

`flatten` has type $\alpha$ *list list* $\rightarrow \alpha$ *list*, but since `map hd` is not a value, $\alpha$ cannot be generalized, and this structure must be rejected. However, such structures are often combined with a monomorphic signature:

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

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

As such examples occur fairly often in realistic ML code, we permit the unbound type variables of a structure to be defined by a signature constraint (or functor result constraint). In the absence of a signature constraint, unbound type variables in structures are still illegal.[4]

### 3.2 Other Imperative Features

The difficulties of typing other imperative features like exceptions and continuations are similar to the difficulties of typing references [16].

A type system that allows capturing a polymorphic continuation is unsound.[5] In an expression like:

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

the captured continuation (which we can represent as a term with a hole: `let val x = [] in` $e$) accepts a value and binds it to `x`. Problems arise if this continuation is polymorphic, *i.e.*, if it accepts values of more that one type. But with polymorphism limited to values, the application `callcc (fn k => ...)` is not a value, so `x` and the continuation are not polymorphic. Hence `callcc` may be assigned the fully polymorphic type $\forall\alpha. (\alpha\ \mathit{cont} \rightarrow \alpha) \rightarrow \alpha$.

The Commentary illustrates how a naïve typing for exceptions is unsound [8: page 42]. With polymorphism limited to values, in:

```
let val (e',f) =
        let exception E of α in ... end
in ...
```

the subexpression `let exception E of` $\alpha$ `in ... end` is not a value, so $\alpha$ will not be generalized by the outer `let`-expression. Therefore, exceptions can also be assigned fully polymorphic types.

### 3.3 Other Type Systems

Several other systems for combining polymorphism and references have been devised. With the exception of call-by-name polymorphism, all violate abstraction by revealing the imperative nature of a polymorphic procedure in its type.

---

[4] This extension to ML's module system appears to be sound [Bob Harper, personal communication, February 1993]. It may be possible to allow unbound type variables even in the absence of a signature constraint, but we have not investigated this more flexible extension.

[5] This was discovered by Robert Harper and Mark Lillibridge [`sml` electronic mailing list, July 8, 1991]. For further details, see [2] or [16].

**Standard ML:** Standard ML adopts Tofte's method of typing imperative features [14]. Tofte's solution generalizes the types of values as above, but in addition is able to generalize some type variables when the bound expression is not a value. To determine which type variables of such an *expansive* expression may be generalized, Tofte's system defines a subset of *imperative type variables*, denoted $\_\alpha$ (or $\_\beta$). Imperative type variables are introduced the `ref` operator, which has type $\forall \_\alpha.\, \_\alpha \rightarrow \_\alpha\ \mathit{ref}$. Any type substituted for an imperative type variable must contain only imperative type variables, and imperative type variables may not be generalized when the bound expression is not a value.

**Weak Types:** A system proposed by MacQueen has been implemented by Standard ML of New Jersey [12] for several years, and has recently been formalized by Hoang, Mitchell, and Viswanathan [3]. This system extends Tofte's method by assinging "weakness" numbers to type variables. The weakness of a type variable indicates how many arguments must be supplied to a procedure before it allocates a reference containing that type variable. This extension allows partial applications of polymorphic imperative procedures that are otherwise rejected by Tofte's system. For example, the `imperativeMap` procedure from the introduction has the weak type $\forall \alpha^2 \beta^2.\, (\alpha^2 \rightarrow \beta^2) \rightarrow \alpha^2\ \mathit{list} \rightarrow \beta^2\ \mathit{list}$. When partially applied:

```
let val flatten = imperativeMap hd
```

the resulting procedure `flatten` has polymorphic type $\forall \alpha^1.\, \alpha^1\ \mathit{list\ list} \rightarrow \alpha^1\ \mathit{list}$. In Tofte's system, `flatten` may not be used polymorphically.

**Damas:** Damas proposed one of the earliest systems for typing references [1]. His system derives type judgments of the form:

$$\Gamma \vdash e : \tau * \Delta$$

where $\Delta$ is a finite set of the types of cells that may be allocated by evaluating $e$. Type schemes that represent function types are augmented by a similar set $\Delta$ on the outermost arrow, *i.e.*, $\forall \alpha^*.\, \tau_1 \xrightarrow{\Delta} \tau_2$. Within such a type scheme, $\Delta$ indicates the types of reference cells that the procedure may allocate *when it is applied*.[6]

**Closure Typing:** Leroy and Weis [4, 6] propose a closure typing system based on the observation that it is only necessary to prohibit generalization of type variables appearing in the types of cells reachable after the bound expression has been evaluated (*i.e.*, cells that would not be reclaimed by a garbage collection at this point). As cells may be reachable through the free identifiers of closures, the system records the types of the free identifiers of a procedure in the procedure's type. For example, the type of `imperativeMap` is $\forall \alpha\beta LMN.\, (\alpha \xrightarrow{L} \beta) \xrightarrow{M} \alpha\ \mathit{list} \xrightarrow{N} \beta\ \mathit{list}$ with $\alpha \xrightarrow{L} \beta \triangleright N$.

**Effects:** Several systems for typing references based on effect inference have been proposed. A system proposed by the author in essence extends Damas's system to attach effect sets ($\Delta$) to all function type arrows [15]. In this system, `imperativeMap` has type $\forall \alpha\beta\varsigma_1\varsigma_2\varsigma_3.\, (\alpha \xrightarrow{\varsigma_1} \beta) \xrightarrow{\varsigma_2} \alpha\ \mathit{list} \xrightarrow{\varsigma_1\varsigma_3\alpha\beta} \beta\ \mathit{list}$.

A more sophisticated system devised by Talpin and Jouvelot infers types, effects, and effect regions for expressions [13]. This system assigns to both `map` and `imperativeMap` the same type, $\forall \alpha\beta\varsigma_1\varsigma_2\varsigma_3.\, (\alpha \xrightarrow{\varsigma} \beta) \xrightarrow{\varsigma_2} \alpha\ \mathit{list} \xrightarrow{\varsigma_1\varsigma_3} \beta\ \mathit{list}$. In effect, this system is able to prove that the side effects with `imperativeMap` are local. But due to the undecidable nature of the problem, there exist more complicated procedures whose type reveals their imperative nature.

[6] Damas uses a slightly different notation.

**Call-by-name Polymorphism:** Several authors [2, 5] have suggested using call-by-name semantics for let-expressions in order to combine imperative features and polymorphism. That is, in the expression:

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$$

the subexpression $e_1$ is not evaluated until it is needed, and it is re-evaluated each time it is used. 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 fully polymorphic types, it would drastically alter the call-by-value nature of ML. Call-by-name let-expressions are easily simulated in our call-by-value system by introducing dummy abstractions and applications:

$$\mathsf{let\ name}\ x = e_1\ \mathsf{in}\ e_2 \quad \Rightarrow \quad \mathsf{let}\ x = \lambda d. e_1\ \mathsf{in}\ e_2[x \mapsto (x\ \mathsf{0})]$$

where $d$ is a fresh variable and $e_2[x \mapsto (x\ \mathsf{0})]$ means the capture-avoiding substitution of $(x\ \mathsf{0})$ for free $x$ in $e_2$.

# 4 Polymorphism for Values in Practice

Restricting polymorphism to values rejects some expressions that are typable in Standard ML. This solution is practical only if such expressions arise infrequently.

To determine how often such examples arise in practice, we modified Standard ML of New Jersey as discussed in Section 3.1. We gathered an extensive collection of ML programs and compiled them with the modified compiler. Our findings are summarized in Table 1. The results are surprising: realistic ML code rarely *computes* polymorphic values. Furthermore, of the few cases where this does happen, almost all involve simple, purely functional computation. The only non-functional examples we found were three uses in the New Jersey compiler of the unsafe procedure `c_function` that performs dynamic linking, a construction of the polymorphic empty vector in the Edinburgh ML Library, and two constructions of polymorphic events in eXene. The other cases were partial applications of procedures like `map` or the function composition operator, `o`. All were trivial to fix.

Reppy's Concurrent ML implementation [11] illustrates the benefit of assigning fully polymorphic types to imperative procedures. Concurrent ML makes extensive use of Standard ML of New Jersey's first-class continuations [2] to implement threads. In order to assign fully polymorphic types to several primitives within the New Jersey compiler's weak type system, Reppy's implementation uses an unsafe version of `callcc` that has type $\forall\alpha. (\alpha\ \mathit{cont} \rightarrow \alpha) \rightarrow \alpha$, rather than the safe but weak type $\forall\alpha^1. (\alpha^1\ \mathit{cont} \rightarrow \alpha^1) \rightarrow \alpha^1$. The use of unsafe `callcc` is justified by a separate, manual proof of soundness for Concurrent ML. However, with polymorphism for values, `callcc` is fully polymorphic, and the trouble causing primitives are automatically assigned fully polymorphic types. Hence the soundness of Concurrent ML is a consequence of the soundness of the underlying type system, and no separate proof is necessary.

We submit that limiting polymorphism to values offers the best compromise for integrating polymorphism with imperative features. Other systems for typing imperative features must be justified against our empirical evidence that their power is not needed.

**Announcement:** A patch for Standard ML of New Jersey (Version 0.93) that restricts polymorphism to values and eliminates weak types is available from titan.cs.rice.edu in directory public/wright, file vsml.93.tar.Z.

# Acknowledgments

I am indebted to Bruce Duba for his comments and suggestions.

| Program | Size in Lines | Imperative Features[a] | Changes Required |
|---|---|---|---|
| Standard ML of New Jersey (version 93) | 62,100 | R E C | 4 $\eta$-expansions<br>4 casts in unsafe bootstrap code |
| SML/NJ Library | 6,400 | R E C | none |
| ML Yacc | 7,300 | R E C | 2 $\eta$-expansions<br>2 $\eta$-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 $\eta$-expansion<br>added `never` for eXene |
| eXene X windows toolkit (Reppy and Gansner) | 20,200 | R E | 6 $\eta$-expansions<br>1 `(choose [])` changed to `never`<br>1 declaration moved |
| Edinburgh ML Library | 15,400 | R E | 1 $\eta$-expansion<br>1 `(vector [])` changed to `#[]` |
| ML Kit Interpreter | 38,000 | R E | 5 $\eta$-expansions |
| Isabelle Theorem Prover | 18,600 | R E | 2 $\eta$-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 $\eta$-expansions |
| 3d Wireframe Graphics (Olin Shivers) | 2200 | R E | none |
| Hilbert (Thomas Yan) | 500 | R | 1 type declaration |
| Grobner Basis (Thomas Yan) | 1000 | R E | 3 $\eta$-expansions |

[a] R = references; E = exceptions; C = continuations.

Table 1: Polymorphism for Values in Practice

# References


[1] Damas, L. M. M. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1985.

[2] Harper, R., Duba, B. F., and MacQueen, D. Typing first-class continuations in ML. *Journal of Functional Programming* (to appear 1993). Preliminary version in *Proc. 18th Annual Symposium on Principles of Programming Languages*, 1991, pp. 163–173.

[3] Hoang, M., Mitchell, J., and Viswanathan, R. Standard ML weak polymorphism and imperative constructs. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science* (June 1993). To appear.

[4] Leroy, X. *Typage polymorphe d'un langage algorithmique.* PhD thesis, L'Université Paris 7, 1992.

[5] 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), pp. 220–231.

[6] Leroy, X., and Weis, P. Polymorphic type inference and assignment. In *Proceedings of the 18th Annual Symposium on Principles of Programming Languages* (January 1991), pp. 291–302.

[7] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (1978), 348–375.

[8] Milner, R., and Tofte, M. *Commentary on Standard ML.* MIT Press, Cambridge, Massachusetts, 1991.

[9] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[10] Plotkin, G. D. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science 1* (1975), 125–159.

[11] Reppy, J. H. *Higher-order Concurrency.* PhD thesis, Cornell University, 1991.

[12] Standard ML of New Jersey release notes (version 0.75). AT&T Bell Laboratories, November 1991.

[13] Talpin, J.-P., and Jouvelot, P. The type and effect discipline. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science* (June 1992), pp. 162–173.

[14] Tofte, M. Type inference for polymorphic references. *Information and Computation 89*, 1 (November 1990), 1–34.

[15] Wright, A. K. Typing references by effect inference. In *Proceedings of the 4th European Symposium on Programming, LNCS 582* (1992), Springer-Verlag, pp. 473–491.

[16] Wright, A. K., and Felleisen, M. A syntactic approach to type soundness. Tech. Rep. 91-160, Rice University, April 1991. To appear in: *Information and Computation.*