Proc. 20th Symp. Principles of Programming Languages, 1993, pages 220–231.

Polymorphism by name for references and continuations

Xavier Leroy

École Normale Supérieure and INRIA Rocquencourt*

Abstract

This article investigates an ML-like language with by-name semantics for polymorphism: polymorphic objects are not evaluated once for all at generalization time, but re-evaluated at each specialization. Unlike the standard ML semantics, the by-name semantics works well with polymorphic references and polymorphic continuations: the naive typing rules for references and for continuations are sound with respect to this semantics. Polymorphism by name leads to a better integration of these imperative features into the ML type discipline. Practical experience shows that it retains most of the efficiency and predictability of polymorphism by value.

1 Introduction

Polymorphic type disciplines like that of ML fit well within purely applicative languages. However, polymorphism becomes problematic when imperative features are added to a purely applicative kernel. In this paper, we consider two important imperative features: references (data structures that can be modified in-place); and first-class continuations (objects that capture the control state of the evaluator). In the ML community, it has long been known that the naive polymorphic typing for references is unsound; strong typing restrictions must be put on polymorphic references to ensure soundness [19]. Recently, it has been shown that the natural polymorphic typing for continuations [4] is unsound for similar reasons [8, 9].

In this paper, we show that these difficulties are not inherent to the polymorphic typing of references and continuations, but specific to the ML semantics for generalization and specialization (the two constructs that introduce polymorphism): when generalization and specialization are given alternate semantics, the simple polymorphic typings for references and continuations becomes sound. We call the ML semantics *polymorphism by value*, and the alternate semantics studied here *polymorphism by name.*

*Author's address: INRIA Rocquencourt, projet Formel, B. P. 105, 78153 Le Chesnay, France. E-mail: `Xavier.Leroy@inria.fr`.

Generalization is the operation that transforms a term of type $\tau[\alpha]$, where α is a type variable standing for an unknown type, into a term of type $\forall\alpha.\, \tau[\alpha]$. Specialization is the operation that transforms a term of type $\forall\alpha.\, \tau[\alpha]$ into a term of type $\tau[\sigma]$, for some given type expression σ. With ML's polymorphism by value, generalization has strict semantics: it evaluates its argument once and for all, and the resulting value is shared between all specializations of the polymorphic term produced. With polymorphism by name, generalization has lazy semantics: it suspends the evaluation of its argument, and each specialization re-evaluates this suspension in the current context. Polymorphism by name is used in Quest [2]; viewed as a restricted form of polymorphism, the "generics" of Clu or Ada also follows this semantics. Drawing the parallel with function application, polymorphism by value is analogous to call by value, and polymorphism by name is analogous to call by name.

These two semantics for polymorphism cannot be distinguished in a purely applicative language without recursion: lambda-calculus with polymorphic typing is strongly normalizing. This is no longer true when we add imperative features to the core language. In particular, polymorphism by value makes it possible to access the same reference object or continuation object with two different types, which can compromise type safety; while this cannot happen with polymorphism by name, since specializations to the two types return two different references or continuations.

This semantic difference has major consequences on the polymorphic typing discipline. With polymorphism by value, some typing restrictions must be put on generalization and/or on the constructs that build references and continuations to avoid the inconsistent use of one reference or continuation object with different types. Several polymorphic type systems have been proposed that achieve this goal [3, 19, 15, 13, 20, 18], but they are either overly restrictive (many useful polymorphic functions that use references or continuations are rejected), or complicated and hard to understand from the programmer's standpoint. In contrast, with polymorphism by name, the simple, intuitive polymorphic type disci-

Semantics	Implicit syntax	Explicit syntax	Naive polymorphic typing
Polymorphism by value	ML	FX-87 [7]	unsound
Polymorphism by name	This work	Quest, Clu, Ada	sound

Figure 1: Four approaches to polymorphism

plines can be extended straightforwardly to references and continuations, and provide excellent support for the imperative programming style.

This fact is folk lore, and the author does not claim originality for noticing it. It is briefly mentioned in several discussions of imperative languages with polymorphic type systems [7, 2]. This fact is also apparent in Harper and Lillibridge's recent work on CPS conversion for polymorphic languages [9, 10], which shows that an ML-like language (i.e. one with call by value and polymorphism by value) does not admit any type-preserving CPS transform, in contrast to languages with call by name or polymorphism by name.

The first aim of the present paper is to formally state this folklore result, by giving a soundness proof for Milner's type system with respect to the by-name semantics, including references and first-class continuations. This proof improves over previous soundness proofs for references in the setting of structural operational semantics [19] by using only elementary techniques instead of more involving techniques such as co-induction.

The second aim is to clarify a confusion that appears in some of the works mentioned above: the confusion between the fact that polymorphism is given by-name semantics and the fact that polymorphism is explicit in the syntax. The explicit presentation of polymorphism consists in providing special syntactic constructs for generalization and specialization: for instance, abstraction over a type variable ($\Lambda\alpha.\, e$) and application of a term to a type ($e\langle\tau\rangle$) in Girard's and Reynold's second-order lambda-calculus. Ada, Clu, Quest follow this approach. The alternate presentation consists in leaving these operations implicit in the source program, and to perform them silently at conventional program points. In the ML language, generalization is performed by the `let` construct, and specialization by referencing a variable. All existing languages with polymorphism by name have polymorphism explicit in the syntax. This leads Cardelli to write [2]:

> Mutability [in Quest] interacts very nicely with all the quantifiers, including polymorphism [...] The problems encountered in ML are avoided by the use of explicit polymorphism.

This is misleading: mutable objects cause no difficulties in Quest because polymorphism follows the by-name semantics, not because it is explicit in the syntax. Indeed, the two semantics and the two syntactic presentations for polymorphism can be combined independently (see figure 1); but the fact that the simple polymorphic typing is sound is specific to the by-name semantics. To clearly make this point, this paper investigates a calculus with implicit polymorphism and by-name semantics for generalization and specialization. This calculus remains very close to the ML language, much closer than languages with explicit polymorphism, yet it safely supports the polymorphic typing of references and continuations without putting complicated restrictions over typing.

Our final aim is to discuss the practicality of polymorphism by name. Polymorphism by name is sometimes rejected a priori on the grounds of inefficiency [7], and also on the grounds that an imperative language with non-strict constructs is error-prone. The author's experience with a prototype implementation of ML with polymorphism by name suggests that these problems are minor in practice: polymorphism by name can be compiled just as efficiently as polymorphism by value in the most frequent cases; and programs that behave differently under the two semantics are quite rare.

The remainder of this paper is organized as follows. Section 2 informally introduces polymorphism by name, and shows how it interacts with references and continuations. Section 3 gives an operational semantics for this calculus, and shows the soundness of Milner's type system with respect to this semantics. Section 4 reports on the practicality of polymorphism by name.

2 Informal development

2.1 The `let name` binding

The `let` binding plays two roles in ML: type generalization and sharing of subcomputations. On the one hand, the expression `let` $x = a$ `in` b generalizes the type of a, allowing x to be used in b with different instances of the type of a. On the other hand, this expression evaluates a once and for all, and shares the resulting value between all occurrences of x in b. The other ML binding construct, function abstraction, also performs sharing on the value of the function argument, but does not generalize its type (for reasons relevant to the type inference problem).

MLN, the variant of ML with polymorphism by name proposed in this paper, separates type generalization from value sharing. The `let` construct is replaced by the `let name` construct:

$$\texttt{let name } x = a \texttt{ in } b.$$

This construct generalizes the type of a, giving a polymorphic type to x in b. In contrast with the usual `let` binding, the `let name` binding does not evaluate a immediately. Instead, the evaluation of a is suspended and is restarted each time x is referenced in b. In other terms, the MLN expression `let name` $x = a$ `in` b behaves exactly like the ML expression

$$\texttt{let } x = \lambda().\, a \texttt{ in } b\{x \leftarrow x()\},$$

or equivalently like the textual substitution

$$b\{x \leftarrow a\}.$$

Hence, in MLN, the two roles of the ML `let` are provided by separate constructs: the `let name` binding performs type generalization but does not share the corresponding value, while function abstraction performs value sharing, but does not generalize the corresponding type. The parallel is more apparent if we introduce the derived form

$$\texttt{let val } x = a \texttt{ in } b,$$

which is syntactic sugar for $(\lambda x.\, b)(a)$. The `let val` binding shares the value of a between all occurrences of x in b, but does not generalize the type of a; hence, all occurrences of x in b are given the same type.

The semantics for `let name` are consistent with a well-known property of the ML type system [16]: that the expression `let` $x = a$ `in` b has type τ if and only if the substitution $b\{x \leftarrow a\}$ has type τ, provided x occurs in b. The `let name` construct carries this equivalence one step further: `let name` $x = a$ `in` b not only typechecks but also behaves like the textual substitution $b\{x \leftarrow a\}$.

What have we gained by separating type generalization and value sharing into distinct constructs? In a purely applicative setting, nothing. Worse, we lose the ability to define polymorphic objects that are computed only once and shared among all their invocations. We shall discuss later how serious this restriction is for purely applicative programs. However, when we add imperative features such as references and continuations, the restriction has important benefits, as we shall see.

2.2 References

References are indirection cells whose contents can be physically updated. They model data structures that can be modified in-place. References are presented through the primitive operations $\texttt{ref}(a)$, to create a fresh reference to a; and $!a$ to access the contents of reference a; and $a := b$ to update the contents of reference a by b. For typechecking, we introduce the type τ `ref` of references containing an object of type τ. The obvious typings for the operations over references are:

$\tau \rightarrow \tau\ \texttt{ref}$	for creation
$\tau\ \texttt{ref} \rightarrow \tau$	for access
$\tau\ \texttt{ref} \times \tau \rightarrow \tau$	for update.

for all types τ.

It is well known that these typings are not sound in a language with polymorphism by value. Here is a classical example:

```
let r = ref(λx. x) in
    r := (λx. x + 1);
    if (!r)(true) then ... else ...
```

With the typings above, the reference r is given type $\forall\alpha.\,(\alpha \rightarrow \alpha)$ `ref`. Hence, we can use `r` with type $(\texttt{int} \rightarrow \texttt{int})$ `ref` and assign it the successor function. We can then consider `r` with type $(\texttt{bool} \rightarrow \texttt{bool})$ `ref`; hence (`!r`)(`true`) has type `bool`, and the `if` statement is well-typed. However, evaluating (`!r`)(`true`) causes 1 to be added to `true`, which is a run-time type violation.

With polymorphism by value, type safety can be compromised when a reference is given a non-trivial polymorphic type. Several restrictions of the ML type system have been proposed that rule out this situation [3, 19, 13, 20, 18]. However, finding "the" right type system for ML with references is still an active research topic. The main difficulty is to give a type system that is correct but not overly restrictive.

On the one hand, it is not easy to statically control the propagation of references in a program. Since references are first-class values, they can be returned as function results, stored into data structures, passed through polymorphic functions, and even hidden inside function closures as in the following example.

```
let functional_ref =
  λx. let r = ref x
      in ((λ(). !r), (λy. r := y))
```

The function `functional_ref` creates a reference and disguises it as two functions, one for access, the other for update. The two functions returned seem totally unrelated, and there is not even a `ref` in their types. It is difficult to keep track of the reference hidden in these two functions.

On the other hand, the type system should not put overly strong restrictions over references whose types contain type variables. Otherwise, many useful polymorphic functions that use references are rejected, and the imperative programming style is poorly supported.

Consider the following function, which reverses a list iteratively:

```
let reverse =
  λl. let arg = ref l in
      let res = ref [] in
      while not null(arg) do
        res := head(!arg) :: !res;
        arg := tail(!arg)
      done;
      !res
```

Because of the two local references that hold intermediate results, most extensions of the ML type system give a more restrictive type to this function than to its purely applicative counterpart, even though the two functions compute exactly the same result. The most advanced extensions [13, 18] succeed in giving the same type to the two functions, but they require complex type algebras, where types reflect many operational properties of the functions. This conflicts with the use of types as partial specifications in module interfaces.

Polymorphism by name provides an indirect way to resolve this tension. With polymorphism by name, it is semantically impossible to create a reference and consider it with two different types. Consider again the first example above, with `let` replaced by `let name`:

```
let name r = ref(λx. x) in
    r := (λx. x + 1);
    if (!r)(true) then ... else ...
```

The test `if (!r)(true)` re-evaluates `ref({λ}x{. }x)`, resulting in a new reference to the identity function, different from the reference that was assigned the successor function in the previous line. Hence `!r` evaluates to the identity function, and no run-time type violation occurs. With `let val` instead of `let name`, the reference r would be given type $(\alpha \rightarrow \alpha)$ `ref`, where α is not generalized. Hence α is instantiated to `int` when typing the assignment, and typing the test leads to a static type error.

The `functional_ref` example proceeds similarly: the only way to do something harmful with `functional_ref` is to apply it to a polymorphic object (e.g. the identity function), bind the two returned functions to variables, and use them with two different instantiations of their types:

```
let (read, write) = functional_ref(λx. x) in ...
```

Either the `let` is a `let val`, and identifiers `read` and `write` remain monomorphic; or the `let` is a `let name`, and `functional_ref({λ}x{. }x)` is re-evaluated each time `read` or `write` are referenced, creating fresh, non-aliased references to the identity function each time.

These examples show that polymorphism by name avoids type violations when polymorphic references are inconsistently used. However, it perfectly supports the consistent sharing of references inside polymorphic functions, even if these references have statically unknown types. For instance, in the case of the function `reverse`, the desired behavior is obtained by:

```
let name reverse =
  λl. let val arg = ref l in
      let val res = ref [] in
      while not null(arg) do
        res := head(!arg) :: !res;
        arg := tail(!arg)
      done;
      !res
```

The `let val` bindings for `arg` and `res` ensure that these identifiers are bound to the same references throughout the `while` loop. This causes no typing difficulties because `arg` and `res` are consistently used with the same type inside the loop. The outermost `let name` binding ensures that `reverse` is polymorphic. The fact that the closure representing this function will be rebuilt each time `reverse` is used, instead of being shared between all uses, is semantically transparent.

2.3 First-class continuations

We now consider the addition of continuations as first-class values to the core ML language. We closely follow the presentation adopted in the Standard ML of New Jersey implementation. This presentation and some alternatives are thoroughly discussed by Duba, Harper and MacQueen [4], to which the reader is referred for a more gentle introduction to continuations in ML.

In SML-NJ, continuations are presented through the type τ `cont` of continuations expecting a value of type τ, and the two operations `callcc`(a), to capture the current continuation and pass it to the function a, and `throw`(a_1, a_2), to restart the continuation a_1 on the value a_2. The typings for these operations are, for all types τ and τ':

$$
\begin{array}{ll}
(\tau\ \texttt{cont} \rightarrow \tau) \rightarrow \tau & \text{for } \texttt{callcc} \\
\tau\ \texttt{cont} \times \tau \rightarrow \tau' & \text{for } \texttt{throw}
\end{array}
$$

These typings are sound in a simply-typed language [4]. It had long been believed that they were also sound in the polymorphic type system of ML, until Harper and Lillibridge came up with a counterexample [8]:

```
let later =
  callcc(λk. (λx. x,
              λf. throw(k, (f, λg. ()))))
in print(first(later)("Hello world"));
   second(later)(λx. x + 1)
```

Everything typechecks with the typings above: `later` is given type $\forall\alpha,\beta.\,(\alpha \rightarrow \alpha) \times (\alpha \rightarrow \alpha) \rightarrow \beta$, and therefore can be used with α instantiated to `string` in the first part of the sequence and with α instantiated to `int` in the second part. At run-time, the `callcc` construct binds k to the continuation $\texttt{later} \mapsto \texttt{print(first(later)}\ldots)$. Execution proceeds by applying the identity function to the string "`Hello world`", then by restarting the continuation k on the value $v = (\{\lambda\}\texttt{x}\{.\}\texttt{x}+1), (\{\lambda\}\texttt{g}\{.\,()\})$. A run-time type violation follows when trying to apply the first component of v to the string "`Hello world`". The reason is that the type of value v is less general than the type statically assumed for `later` in the body of the `let`.

This situation is quite similar to the first example with references above: in both cases, static typing assumptions about identifiers bound to polymorphic objects are violated after the binding has changed, either because a reference was updated, or because a continuation was restarted.

This problem with polymorphic continuations can be avoided by typing restrictions similar to those for polymorphic references [21, 12] — with the same drawbacks: the corresponding type systems are either too restrictive or too complicated. Again, polymorphism by name provides an alternate solution. With by-name semantics for generalization, it is impossible to capture a continuation that generalizes the type of the value received by the continuation, as in Harper and Lillibridge's example, because polymorphism by name does not generalize the type of values, but only of suspended expressions. In the setting of polymorphism by name, Harper and Lillibridge's example becomes:

```
let name later =
  callcc(λk. (λx. x,
              λf. throw(k, (f, λg. ()))))
in print(first(later)("Hello world"));
   second(later)(λx. x + 1)
```

(Binding `later` with a lambda abstraction would result in a static type error.) The `callcc` expression defining `later` is evaluated twice, once for each reference to `later`, capturing different continuations than in the case of by-value semantics. In particular, the application `second(later)` binds `k` to the continuation $\texttt{later} \mapsto \texttt{second(later)}(\{\lambda\}\texttt{x}\{.\}\texttt{x}+1)$, and the `throw k` restarts the program at that point, with `second(later)` bound to the harmless function $\{\lambda\}\texttt{g}\{.\,()\}$. No run-time type violation occurs.

3 Formalization

In this section, we formalize the calculus presented above, give its operational semantics, and show the soundness of Milner's typing rules with respect to the semantics.

3.1 Syntax

The language we consider is the core ML language enriched with references and continuations. We assume given a countable set of variable identifiers, ranged over by x. The terms of the calculus, ranged over by a, are described by the grammar below.

Expressions:

a	::=	i	integer constant
	|	x	variable identifier
	|	$\lambda x.\, a$	function abstraction
	|	$a_1(a_2)$	function application
	|	`let name` $x = a_1$ `in` a_2	suspended binding
	|	$op(a)$	unary operator application
	|	$op(a_1, a_2)$	binary operator application

Operators:

op	::=	`ref`	creation of a reference
	|	`deref`	access to a reference
	|	`assign`	modification of a reference
	|	`callcc`	capture the current continuation
	|	`throw`	invocation of a continuation

For the sake of simplicity, we have omitted conditional constructs and fixpoint operators in the calculus above. It is straightforward to add typing and evaluation rules for conditionals. As for recursive functions, there is no need to introduce a built-in fixpoint operator, since recursive functions can be defined (albeit painfully) in terms of references and continuations. For instance, the expression below computes the fixpoint $Y(\lambda f.\, \lambda x.\, a)$:

```
let val r = ref(λx. throw(k, 1)) in
  r := λx. a{f ← !r}; !r
```

(Here, `k` is assumed to be some dummy integer continuation previously captured. The function $\{\lambda\}\texttt{x}\{.\}\texttt{throw(k,1)}$ has type $\tau \to \tau'$ for all types τ and τ', and therefore provides an initial value with the right type for the reference.)

3.2 Operational semantics

We now give a continuation semantics to the language above, in structured operational style. The rules in figure 3 define the evaluation predicate $e/s \vdash a; k \Rightarrow r$, meaning "in evaluation environment e and initial store s, the evaluation of the term a followed by the continuation k terminates on the answer r". The rules also define the auxiliary predicate $s \vdash v \triangleright k \Rightarrow r$, meaning "in the initial store s, the value v passed to the continuation k produces the answer r". An answer is either a value and a modified store, or the constant `wrong` denoting a run-time type error. Evaluation environments map variables to values. These semantic objects are represented

Values:	v	$::=$	i	integer constant
		$\mid$	(x, a, e)	function closure
		$\mid$	ℓ	store location
		$\mid$	k	continuation
Answers:	r	$::=$	v/s	normal answer
		$\mid$	`wrong`	run-time type error
Environments:	e	$::=$	$[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n]$	finite mapping from variables to values
Continuations:	k	$::=$	`stop`	end of the program
		$\mid$	$\texttt{apply1}(a, e, k)$	function part of an application
		$\mid$	$\texttt{apply2}(x, a, e, k)$	argument part of an application
		$\mid$	$\texttt{unop}(op, k)$	argument of a unary operation
		$\mid$	$\texttt{binop1}(op, a, e, k)$	first argument of a binary operation
		$\mid$	$\texttt{binop2}(op, v, k)$	second argument of a binary operation
Stores:	s	$::=$	$[\ell_1 \leftarrow v_1, \ldots, \ell_n \leftarrow v_n]$	finite mapping from locations to values

Figure 2: The semantic objects

$$s \vdash v \triangleright \texttt{stop} \Rightarrow v/s \qquad \frac{s \vdash i \triangleright k \Rightarrow r}{e/s \vdash i; k \Rightarrow r} \qquad \frac{x \in \mathrm{Dom}(e) \qquad s \vdash e(x) \triangleright k \Rightarrow r}{e/s \vdash x; k \Rightarrow r}$$

$$\frac{s \vdash (x, a, e) \triangleright k \Rightarrow r}{e/s \vdash \lambda x.\, a; k \Rightarrow r} \qquad \frac{e/s \vdash a_2\{x \leftarrow a_1\}; k \Rightarrow r}{e/s \vdash \texttt{let name } x = a_1 \texttt{ in } a_2; k \Rightarrow r}$$

$$\frac{e/s \vdash a_1; \texttt{apply1}(a_2, e, k) \Rightarrow r}{e/s \vdash a_1(a_2); k \Rightarrow r} \qquad \frac{e/s \vdash a_2; \texttt{apply2}(x, a_1, e_1, k) \Rightarrow r}{s \vdash (x, a_1, e_1) \triangleright \texttt{apply1}(a_2, e, k) \Rightarrow r} \qquad \frac{e[x \leftarrow v]/s \vdash a; k \Rightarrow r}{s \vdash v \triangleright \texttt{apply2}(x, a, e, k) \Rightarrow r}$$

$$\frac{e/s \vdash a; \texttt{unop}(op, k) \Rightarrow r}{e/s \vdash op(a); k \Rightarrow r} \qquad \frac{e/s \vdash a_1; \texttt{binop1}(op, a_2, e, k) \Rightarrow r}{e/s \vdash op(a_1, a_2); k \Rightarrow r} \qquad \frac{e/s \vdash a_2; \texttt{binop2}(op, v_1, k) \Rightarrow r}{s \vdash v_1 \triangleright \texttt{binop1}(op, a_2, e, k) \Rightarrow r}$$

$$\frac{\ell \notin \mathrm{Dom}(s) \qquad s[\ell \leftarrow v] \vdash \ell \triangleright k \Rightarrow r}{s \vdash v \triangleright \texttt{unop}(\texttt{ref}, k) \Rightarrow r} \qquad \frac{\ell \in \mathrm{Dom}(s) \qquad \vdash s(\ell) \triangleright k \Rightarrow r}{s \vdash \ell \triangleright \texttt{unop}(\texttt{deref}, k) \Rightarrow r}$$

$$\frac{\ell \in \mathrm{Dom}(s) \qquad s[\ell \leftarrow v] \vdash v \triangleright k \Rightarrow r}{s \vdash v \triangleright \texttt{binop2}(\texttt{assign}, \ell, k) \Rightarrow r}$$

$$\frac{e[x \leftarrow k]/s \vdash a; k \Rightarrow r}{s \vdash (x, a, e) \triangleright \texttt{unop}(\texttt{callcc}, k) \Rightarrow r} \qquad \frac{s \vdash v \triangleright k_1 \Rightarrow r}{s \vdash v \triangleright \texttt{binop2}(\texttt{throw}, k_1, k) \Rightarrow r}$$

If none of the conclusions of the rules above match:

$$e/s \vdash a; k \Rightarrow \texttt{wrong} \qquad s \vdash v \triangleright k \Rightarrow \texttt{wrong}$$

Figure 3: The evaluation rules

$$E \vdash i : \mathtt{int} \qquad \frac{E(x) = \forall \alpha_1 \ldots \alpha_n.\, \tau}{E \vdash x : \tau\{\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n\}}$$

$$\frac{E[x : \tau'] \vdash a : \tau}{E \vdash \lambda x.\, a : \tau' \to \tau} \qquad \frac{E \vdash a_1 : \tau' \to \tau \qquad E \vdash a_2 : \tau'}{E \vdash a_1(a_2) : \tau}$$

$$\frac{E \vdash a_1 : \tau' \qquad FV(\tau') \setminus FV(E) = \{\alpha_1 \ldots \alpha_n\} \qquad E[x : \forall \alpha_1 \ldots \alpha_n.\, \tau'] \vdash a_2 : \tau}{E \vdash \mathtt{let\ name}\ x = a_1\ \mathtt{in}\ a_2 : \tau}$$

$$\frac{E \vdash a : \tau}{E \vdash \mathtt{ref}(a) : \tau\ \mathtt{ref}} \qquad \frac{E \vdash a : \tau\ \mathtt{ref}}{E \vdash \mathtt{deref}(a) : \tau} \qquad \frac{E \vdash a_1 : \tau\ \mathtt{ref} \qquad E \vdash a_2 : \tau}{E \vdash \mathtt{assign}(a_1, a_2) : \tau}$$

$$\frac{E \vdash a : \tau\ \mathtt{cont} \to \tau}{E \vdash \mathtt{callcc}(a) : \tau} \qquad \frac{E \vdash a_1 : \tau' \qquad E \vdash a_2 : \tau'\ \mathtt{cont}}{E \vdash \mathtt{throw}(a_1, a_2) : \tau}$$

Figure 4: The typing rules

as terms from the algebra defined by the grammar in figure 2. Store locations, ranged over by ℓ, are taken from a given infinite set of locations.

The evaluation rules can almost be read as the transitions of an abstract machine such as the CEK-machine [17, 6] enriched with a store: the terms a represent the code component of the machine, the environments e represent the environment component, and the continuation terms k represent the stack. The judgement $e/s \vdash a; k \Rightarrow r$ can be read as "initiate the computation of a, pushing the current state of the computation on k if necessary". The judgement $s \vdash v \rhd k \Rightarrow r$ can similarly be read as "resume the saved computation on top of k over the value v".

The only rule that does not correspond closely to a transition of a CEK-like machine is the rule for `let name`. For the sake of simplicity, we have expressed the evaluation of $\mathtt{let\ name}\ x = a_1\ \mathtt{in}\ a_2$ as the evaluation of the textual substitution $a_2\{x \leftarrow a_1\}$. An alternate presentation, closer to an actual execution model, is to bind x to the suspension (a_1, e) during the evaluation of a_2, and to evaluate this suspension each time x is referenced; this presentation is detailed in [12, chapter 6].

3.3 Type system

We now apply Milner's polymorphic type discipline to the language above. The typing rules are well known; we recall them in figure 4. The rules define the predicate $E \vdash a : \tau$, meaning "under the assumptions E, the expression a has type τ". Type expressions, type schemes and typing environments are defined by the grammar:

Type expressions:

τ	$::=$	`int`	the type of integers
	$\mid$	α	type variable
	$\mid$	$\tau_1 \to \tau_2$	function type
	$\mid$	τ `ref`	reference type
	$\mid$	τ `cont`	continuation type

Type schemes:

$\sigma \;::=\; \forall \alpha_1 \ldots \alpha_n.\, \tau$

Typing environments:

$E \;::=\; [x_1 : \sigma_1, \ldots, x_n : \sigma_n]$

3.4 Type soundness

We are now in a position to show a Milner's style soundness result for the proposed calculus: no closed, well-typed term can evaluate to `wrong`. That is, a well-typed term either diverges or terminates with a normal response, but does not terminate on a run-time type violation.

Proposition 1 *If* $[\,] \vdash a : \tau$ *and* $[\,]/[\,] \vdash a; \mathtt{stop} \Rightarrow r$, *then* $r \neq \mathtt{wrong}$.

A simple, indirect proof of this claim is as follows. By rewriting all `let name` nodes in a by the rule

$$\begin{array}{l} \mathtt{let\ name}\ x = a_1\ \mathtt{in}\ a_2 \\ \quad \Rightarrow\ \mathtt{let}\ x = \lambda().\, a_1\ \mathtt{in}\ a_2\{x \leftarrow x()\} \end{array}$$

we transform a into a term a' of ML with polymorphism by value that evaluates to `wrong` if and only if a evaluates to `wrong`. Moreover, the translation a' is well-typed in Tofte's type system [19], with all type variables taken to be imperative type variables. The main reason is that all `let` expressions bind non-expansive expressions. Then, proposition 1 follows from the "well-known fact" that Tofte's type system is sound for the core ML language extended with references and continuations.

Unfortunately, while the soundness of Tofte's system has been shown for ML plus references [19, 21] and separately for ML plus continuations [21], no formal soundness proof has been given for the combination of references and continuations. Hence this indirect argument is not satisfactory. A direct proof of the soundness claim is given in appendix. The proof is a considerable simplification over previous proofs of type soundness in the presence of a store, which rely either on complicated domain constructions [3], or on definitions by greatest fixpoints and proofs by co-induction [19].

The key idea, due to Tofte, is to appeal to the typing relation to define what it semantically means for a functional value to belong to a function type: instead of the usual condition "closure (x, a, e) belongs to type $\tau_1 \rightarrow \tau_2$ iff it maps values of type τ_1 to values of type τ_2", we take that "closure (x, a, e) belongs to type $\tau_1 \rightarrow \tau_2$ iff we can derive the typing judgement $E \vdash \lambda x.\, a : \tau_1 \rightarrow \tau_2$ for some typing environment E that agrees with the evaluation environment e".

We have extended this idea to the semantic typing of continuations: instead of the usual condition "continuation k belongs to type τ `cont` iff it never produces `wrong` when applied to a value of type τ", we use structural induction over k and appeal to the typing rules. The resulting proof is elementary: it proceeds only by structural induction over the terms representing values and evaluation derivations. In particular, there is no need for proofs by co-induction [19, 14]. The proofs also easily extend to other polymorphic type systems for references and continuations [12].

4 Assessment

Polymorphism by name supports references and continuations in a type-safe way, while retaining the ML type algebra and typing rules, that are familiar and easy to understand. This is a strong advantage over the restricted type systems proposed for references and continuations in the setting of by-value semantics, which generally use richer type algebras and more complex typing rules. MLN, the variant of ML with polymorphism by name proposed in this paper, is therefore an interesting alternative to ML when imperative features are considered.

However, MLN is not semantically equivalent to ML: since value sharing and type generalization are performed by distinct constructs in MLN, it is not possible to share the value of a polymorphic object. In this section, we discuss the practical consequences of this fact.

4.1 Differences in semantics

First of all, programs where polymorphic objects are computed by expressions with observable side-effects do not behave the same in ML and in MLN: the side-effects are performed once at creation-time in ML, but several times in MLN — once for each specialization. Example:

```
let name f = print("Hi!"); λx. x in f(f(f))
```

Evaluating this MLN phrase prints "`Hi!`" three times; the corresponding ML phrase prints "`Hi!`" only once. Here is another example, which assumes defined a stamp generator `gensym`:

```
let stamper =
  let stamp = gensym() in λx. (x, stamp)
in ...
```

In ML, the `stamper` function takes arguments of arbitrary types and pairs them with the stamp obtained — the same stamp for all applications of `stamper`. The straightforward translation to MLN behaves differently:

```
let name stamper =
  let val stamp = gensym() in λx. (x, stamp)
in ...
```

Each application of `stamper` re-evaluates the expression defining `stamper`, and therefore calls `gensym` each time; hence, a different stamp is paired with each argument. To preserve the original behavior, the program must be rewritten as follows:

```
let val stamp = gensym() in
let name stamper = λx. (x, stamp)
in ...
```

For more complex examples, deeper transformations might be required. However, these examples are rather artificial. In practice, polymorphic objects are most often defined by expressions that are side-effect free and that do not depend on the state, such as lambda-abstractions. In this case, the translation from ML to MLN is straightforward: it suffices to replace the `let` bindings by `let name` for polymorphic objects and `let val` otherwise, and the behavior of the program is preserved. For the experiments described below, the author translated about 10000 lines of ML programs to MLN this way, without encountering a single case where non-trivial transformations (as in the `stamper` example) were required.

4.2 Differences in efficiency

Even in the cases where it is safe to re-evaluate the expressions defining polymorphic objects, we may fear that this recomputation is a major source of inefficiency.

In practice, this is not the case, because in most programs the vast majority of polymorphic objects are defined as functions $\lambda x.\, a$. The evaluation of these objects reduces to the construction of a closure, which is cheap. Moreover, the standard uncurrying techniques [1, section 6.2] can be used to avoid re-building the function closure when a polymorphic function is immediately applied. Consider the typical code fragment:

```
let name f = λx. a
in ... f(1) ... f(true) ...
```

This MLN program is compiled exactly as the following ML program:

```
let f = λ(). λx. a
in ... f()(1) ... f()(true) ...
```

That is, `f` is compiled as a curried function with two arguments, `()` and `x`. After uncurrying, the two curried applications are transformed into simple calls to a function with two arguments, which is as efficient as the direct application `f(1)` or `f(true)`. (The extra cost of passing the unit argument can easily be avoided.)

However, there are some situations where a polymorphic object is expensive to compute. These situations correspond to the partial application of a curried function that performs a significant amount of computation between the passing of its first and second arguments. Binding the function returned by the partial application allows the sharing of this computation between all subsequent applications of the function. With polymorphism by name, this sharing is impossible if the result of the partial application must remain polymorphic.

Here is an example of this situation. Consider a function that sorts key-data pairs in increasing order for the keys. Assume that the keys and the associated data are not provided together, as a list of pairs, but separately, as a list of keys and a list of associated items. The function result is the permuted list of items. To take advantage of partial applications, the clever way to write this function is to compute the sorting permutation (e.g. a list of integers) as soon as the list of keys is given, and to return a function that simply applies the sorting permutation to the given list of items:

```
let weird_sort =
  λorder. λkeys.
    let permut = ...
    in λitems. apply_permut(permut, items)
```

This is more efficient if several lists of items are to be ordered on the same list of keys:

```
let f = weird_sort (<) (lots_of_integers) in
  ... f(lots_of_strings) ...
  ... f(lots_of_booleans) ...
```

Here, the intermediate function `f` is polymorphic (with type $\forall\alpha.\,\alpha$ `list` $\rightarrow \alpha$ `list`), and therefore can be applied to item lists of different types — without sorting again the list of keys each time. This last point holds in ML, but not in MLN. If `f` is to remain polymorphic, it must be bound by a `let name` construct; then, each application `f(l)` evaluates as

```
weird_sort (<) (lots_of_integers) (l)
```

Hence, the benefits of partial application are lost.

4.3 Experimental results

To evaluate more precisely the impact of polymorphism by name on real programs, the author has implemented a prototype compiler for ML with polymorphism by name, derived from his Caml Light system [5], which implements polymorphism by value. Deriving an MLN compiler from an ML compiler is straightforward: it suffices to transform the MLN expressions

$$\texttt{let name } x = a \texttt{ in } b$$

into ML expressions

$$\texttt{let } x = \lambda\texttt{()}.\, a \texttt{ in } b\{x \leftarrow x\texttt{()}\}$$

early in the compilation process; the remainder of the compiler need not be modified. Good performance crucially depend on the efficiency of the subsequent uncurrying phase, however. In the case of the Caml Light execution model [11, chap. 3], the very same code is generated for the MLN expression

```
let name f = λx. a in ... f(a′) ... f(a″) ...
```

and for the corresponding ML expression

```
let f = λx. a in ... f(a′) ... f(a″) ...
```

which is about the best we can expect.

Figure 5 gives some preliminary benchmark results for the Caml Light-based ML and MLN compilers. The test programs comprise, in addition to the usual toy programs, two medium-sized programs performing mostly symbolic processing, Boyer's simplified theorem prover and an implementation of the Knuth-Bendix completion algorithm, and two pieces of the Caml Light environment, adapted to polymorphism by name and bootstrapped, the lexical analyzer generator (1000 lines) and the compiler itself (8000 lines). Some of these programs are completely monomorphic (Fibonacci, word count). Others use polymorphic functions intensively (Church integers). The more realistic programs operate mostly on monomorphic data structures, but make frequent use of generic functions over lists, hash tables, etc.

The experimental results show that all programs, even the purely monomorphic ones that do not use

Test	ML	MLN	Slowdown MLN/ML	Amount of polymorphism
Fibonacci	5.9 s	6.3 s	6%	none
Church integers	2.5 s	2.9 s	16%	high
Sieve	3.2 s	3.4 s	6%	moderate
Word count	6.4 s	6.7 s	4%	none
Boyer	16.0 s	18.0 s	12%	low
Knuth-Bendix	7.9 s	8.3 s	5%	moderate
Lexer generator	2.1 s	2.3 s	9%	low
The Caml Light compiler	7.1 s	8.3 s	16%	low

Figure 5: Experimental comparison between ML and MLN, in the Caml Light system

`let name` at all, are slowed down by about 5% in MLN. The reason is that a minor optimization in the Caml Light execution model, which relies on the fact that curried functions are always applied to at least one argument, applies to ML, but not to MLN. This slowdown is specific to the Caml Light execution model; it should not occur with more conventional uncurrying techniques. In addition to this general slowdown, the tests exhibit a slowdown by 1% to 10%, which represents the actual cost of polymorphism by name with respect to polymorphism by value.

Adapting such an ML compiler to MLN is straightforward: it suffices to transform $\texttt{let name}\ x = a\ \texttt{in}\ b$ expressions into $\texttt{let val}\ x = \lambda\texttt{()}.\,a\ \texttt{in}\ b\{x \leftarrow x\texttt{()}\}$ early in the compilation process, and let the uncurrying mechanisms work on this intermediate form.

These experimental results are encouraging: they show that an ML compiler with uncurrying mechanisms can easily be adapted to polymorphism by name, without major efficiency loss. The author believes that these results are not specific to the Caml Light implementation, but should apply to any ML compiler equipped with uncurrying mechanisms. Polymorphism by name cannot be dismissed easily on the grounds of inefficiency.

5 Conclusions

We have shown that by-name semantics for polymorphism can be integrated with an ML-like language, resulting in a simple solution to the problems raised by the polymorphic typing of references and continuations.

This solution has two major advantages over the traditional "restricted polymorphism by value" approach, which consists in keeping polymorphism-by-value semantics and putting suitable typing restrictions over references and continuations. First of all, the type system assigns the same types to applicative and imperative implementations of the same function, which is crucial in the context of modular programming (in Standard ML, for instance, most polymorphic functions specified with an applicative type cannot be implemented in an imperative style). Moreover, this result is achieved while keeping the simple, familiar ML type algebra, in contrast with the most recent type systems for restricted polymorphism-by-value, such as effect systems [18] and closure typing [13, 12], which achieve the same results at the cost of complicated type algebras where type expressions are so informative that their use as specifications in module interfaces becomes problematic.

It is true that the polymorphism-by-name approach has some unfortunate drawbacks, such as its inability to express the sharing of some subcomputations of polymorphic values, but these drawbacks seem relatively minor in practice, compared with the drawbacks of the restricted polymorphism-by-value approach. We have presented some experimental evidence of this fact; more experience with polymorphism by name is needed to confirm the practicality of this approach.

A Proof of soundness

In this appendix, we sketch the proof of soundness of Milner's type system with respect to the semantics given in section 3.2.

We first formalize what it means for a value to semantically belong to some type, and for a continuation to semantically accept values of some type. We write these conditions $S \models v : \tau$ and $S \models k :: \tau$, respectively. The hypothesis S is a store typing, that is, a partial mapping from locations to type expressions. The store typing is needed to take into account the sharing of values introduced by the store. The semantic typing relations are defined by structural induction over the terms representing values, continuations and evaluation environments, as follows:

- $S \models i : \texttt{int}$
- $S \models (x, a, e) : \tau_1 \rightarrow \tau_2$ if there exists a typing environment E such that $E \vdash \lambda x.\, a : \tau_1 \rightarrow \tau_2$ and $S \vdash e : E$.

- $S \models \ell : \tau\ \texttt{ref}$ if $\ell \in \text{Dom}(S)$ and $S(\ell) = \tau$.
- $S \models k : \tau\ \texttt{cont}$ if $S \models k :: \tau$.
- $S \models \texttt{stop} :: \tau$ for all types τ.
- $S \models \texttt{apply1}(a, e, k) :: \tau_1 \rightarrow \tau_2$ if there exists a typing environment E such that $E \vdash a : \tau_1$ and $S \models e : E$ and $S \models k :: \tau_2$.
- $S \models \texttt{apply2}(x, a, e, k) :: \tau$ if there exists a typing environment E and a type τ' such that $E \vdash \lambda x.\, a : \tau \rightarrow \tau'$ and $S \models e : E$ and $S \models k :: \tau'$.
- $S \models \texttt{unop}(\texttt{ref}, k) :: \tau$ if $S \models k :: \tau\ \texttt{ref}$.
- $S \models \texttt{unop}(\texttt{deref}, k) :: \tau\ \texttt{ref}$ if $S \models k :: \tau$.
- $S \models \texttt{binop1}(\texttt{assign}, a, e, k) :: \tau\ \texttt{ref}$ if there exists a typing environment E such that $E \vdash a : \tau$ and $S \models e : E$ and $S \models k :: \tau$.
- $S \models \texttt{binop2}(\texttt{assign}, v, k) :: \tau$ if $S \models v : \tau\ \texttt{ref}$ and $S \models k :: \tau$.
- $S \models \texttt{unop}(\texttt{callcc}, k) :: \tau\ \texttt{cont} \rightarrow \tau$ if $S \models k : \tau$.
- $S \models \texttt{binop1}(\texttt{throw}, a, e, k) :: \tau\ \texttt{cont}$ if there exists a typing environment E such that $E \vdash a : \tau$ and $S \models e : E$.
- $S \models \texttt{binop2}(\texttt{throw}, v, k) :: \tau$ if $S \models v : \tau\ \texttt{cont}$.
- $S \models e : E$ if for all $x \in \text{Dom}(e)$, $E(x)$ is a simple type τ such that $S \models e(x) : \tau$.
- $\models s : S$ if $\text{Dom}(s) = \text{Dom}(S)$, and for all $\ell \in \text{Dom}(s)$, we have $S \models s(\ell) : S(\ell)$.

A store typing S' extends a store typing S if $\text{Dom}(S') \supseteq \text{Dom}(S)$, and $S'(\ell) = S(\ell)$ for all $\ell \in \text{Dom}(S)$. The semantic typing relations defined above are obviously stable under store extension; that is, if $S \models v : \tau$ and S' extends S, then $S' \models v : \tau$, and similarly for the other semantic typing relations.

Proposition 2 *Let a be a term, E a typing environment, τ a type expression, S a store typing, k a continuation, s a store, and r an answer.*

1. *If $E \vdash a : \tau$ and $S \models e : E$ and $S \models k :: \tau$ and $\models s : S$ and $e/s \vdash a; k \Rightarrow r$, then $r \neq \texttt{wrong}$.*
2. *If $S \models v : \tau$ and $S \models k :: \tau$ and $\models s : S$ and $s \vdash v \triangleright k \Rightarrow r$, then $r \neq \texttt{wrong}$.*

Proof. The proof is a simple, but tedious induction on the height of the evaluation derivation (the derivation of $e/s \vdash a; k \Rightarrow r$ for (1); the derivation of $s \vdash v \triangleright k \Rightarrow r$ for (2)), and case analysis on a and k, respectively. We give the main cases; the remaining cases are similar.

Case (1) when a is $\texttt{let name}\ x = a_1\ \texttt{in}\ a_2$. From the typing derivation of $E \vdash a : \tau$, we can construct a typing derivation of $E \vdash a_2\{x \leftarrow a_1\} : \tau$ [16, section 4.7.2]. The result follows from induction hypothesis (1) applied to the evaluation of $a_2\{x \leftarrow a_1\}$.

Case (1) when a is $a_1(a_2)$. Let $\tau' \rightarrow \tau$ be the type of a_1. By definition of $\models$ over $\texttt{apply1}$ continuations, we have $S \models \texttt{apply1}(a_2, e, k) :: \tau' \rightarrow \tau$, taking E for the required typing environment. Applying the induction hypothesis (1) to the evaluation $e/s \vdash a_1; \texttt{apply1}(a_2, e, k) \Rightarrow r$, we get $r \neq \texttt{wrong}$ as expected.

Case (2) when k is $\texttt{apply1}(a_2, e_2, k)$. By hypothesis $S \models k :: \tau$, we have $\tau = \tau_1 \rightarrow \tau_2$ and $E_2 \vdash a : \tau_1$ and $S \models e_2 : E_2$ and $S \models k :: \tau_2$, for some τ_1, τ_2, E_2. By hypothesis $S \models v : \tau$, the value v is a closure (x, a_0, e_0), and (3) $E_0 \vdash \lambda x.\, a_0 : \tau_1 \rightarrow \tau_2$ and (4) $S \vdash e_0 : E_0$ for some E_0. Since v is a closure, the elimination rule for $\texttt{apply1}$ closures matches. From (3) and (4), we get $S \models \texttt{apply2}(x, a_0, e_0, k) :: \tau_1$. The expected result follows from induction hypothesis (1) applied to the evaluation $e/s \vdash a_2; \texttt{apply2}(x, a_0, e_0, k) \Rightarrow r$.

Case (2) when k is $\texttt{apply2}(x, a, e, k)$. By hypothesis $S \models k :: \tau$, we have (5) $E \vdash \lambda x.\, a : \tau \rightarrow \tau'$ and (6) $S \models e : E$ and $S \models k :: \tau'$ for some E and some τ'. Consider the environments $e_1 = e[x \leftarrow v]$ and $E_1 = E[x \leftarrow \tau]$. By (6) and the hypothesis $S \models v : \tau$, we have $S \models e_1 : E_1$. Moreover, $E_1 \vdash a : \tau'$ since this is the premise of the only typing rule that concludes (5). Hence we can apply induction hypothesis (1) to the evaluation $e_1/s \vdash a; k \Rightarrow r$.

Case (2) when k is $\texttt{unop}(\texttt{ref}, k')$. Let ℓ be the new location chosen by the evaluation rule. Consider the store typing $S' = S[\ell \leftarrow \tau]$. We have $\models s[\ell \leftarrow v] : S'$. Since $\ell \notin \text{Dom}(s) = \text{Dom}(S)$, the store typing S' extends S. Hence $S \models k :: \tau$ implies $S' \models k :: \tau$, which means $S' \models k' :: \tau\ \texttt{ref}$. The result follows by induction hypothesis (2) applied to the evaluation $s[\ell \leftarrow v] \models \ell \triangleright k' \Rightarrow r$ and to the store typing S'.

Case (2) when k is $\texttt{binop2}(\texttt{assign}, v, k')$. By hypothesis $S \models k :: \tau$, we have $S \models k :: \tau$, and the value v is a location ℓ such that $S(\ell) = \tau$. Since $\models s : S$, it follows that $\ell \in \text{Dom}(s)$. By hypothesis $S \models v : \tau$, we have $\models s[\ell \leftarrow v] : S$. Hence we can apply the induction hypothesis (2) to the evaluation $s[\ell \leftarrow v] \vdash v \triangleright k \Rightarrow r$. It follows that $r \neq \texttt{wrong}$, as desired.

Case (2) when k is $\texttt{unop}(\texttt{callcc}, k')$. By hypothesis $S \models k :: \tau$, we have $\tau = \tau'\ \texttt{cont} \rightarrow \tau'$ and $S \models k' :: \tau'$. By hypothesis $S \models v : \tau$, we have $v = (x, a, e)$, with $E \vdash \lambda x.\, a : \tau'\ \texttt{cont} \rightarrow \tau'$ and $S \vdash e : E$ for some typing environment E. Consider the environments $e' = e[x \leftarrow k]$ and $E' = E[x \leftarrow \tau'\ \texttt{cont}]$. We have $S \models e' : E'$ and, given the typing rule for function abstraction, $E' \vdash a : \tau'$. The result follows by induction hypothesis (1) applied to the evaluation of $e'/s \vdash a; k' \Rightarrow r$.

Case (2) when k is `binop2`$(\texttt{throw}, k', k'')$. By hypothesis $S \models k :: \tau$, it follows that $S \models k' :: \tau$. Applying the induction hypothesis (2) to the evaluation $S \models v \triangleright k' \Rightarrow r$, we get the expected result. $\square$

Proposition 1 in section 3.4 immediately follows from property (1) in proposition 2, taking the empty map for e, E, s and S.

Acknowledgments

The author is indebted to Didier Rémy for many discussions of the ideas in this paper. Many thanks to Benjamin Pierce for his editorial help.

References

[1] A. W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[2] L. Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, 1989.

[3] L. Damas. *Type assignment in programming languages.* PhD thesis, University of Edinburgh, 1985.

[4] B. F. Duba, R. Harper, and D. MacQueen. Typing first-class continuations in ML. In *18th symposium Principles of Programming Languages*, pages 163–173. ACM Press, 1991.

[5] X. L. *et al.* The Caml Light system, release 0.6. Software and documentation distributed by anonymous FTP on `ftp.inria.fr`, 1993.

[6] M. Felleisen and D. P. Friedman. Control operators, the SECD machine and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 131–141. North-Holland, 1986.

[7] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *13th symposium Principles of Programming Languages*, pages 28–38. ACM Press, 1986.

[8] R. Harper and M. Lillibridge. ML with `callcc` is unsound. Message sent to the `sml` mailing list, July 1991.

[9] R. Harper and M. Lillibridge. Polymorphic type assignment and CPS conversion. In *1992 SIGPLAN Continuations Workshop*, 1992.

[10] R. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In *20th symposium Principles of Programming Languages.* ACM Press, 1993.

[11] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[12] X. Leroy. *Typage polymorphe d'un langage algorithmique.* Doctoral dissertation (in French), Université Paris 7, 1992.

[13] X. Leroy and P. Weis. Polymorphic type inference and assignment. In *18th symposium Principles of Programming Languages*, pages 291–302. ACM Press, 1991.

[14] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Comput. Sci.*, 87:209–220, 1991.

[15] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* The MIT Press, 1990.

[16] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B*, pages 367–458. The MIT Press/Elsevier, 1990.

[17] J. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740. ACM Press, 1972.

[18] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science 1992.* IEEE Computer Society Press, 1992.

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

[20] A. K. Wright. Typing references by effect inference. In *European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science.* Springer-Verlag, 1992.

[21] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. Technical report TR91-160, Rice University, 1991.