Reference Polymorphism in ML

Dave King
The Pennsylvania State University
dhking@cse.psu.edu

John Hannan
The Pennsylvania State University
hannan@cse.psu.edu

Abstract

ML is a family of functional type-safe programming languages with both polymorphic types and mutable state. The naive combination of these two features leads to type violations, resulting in an unsound language. The value restriction as proposed by Wright and adopted by Standard ML restores soundness to the language, but restricts all references to be monomorphic. However, allocating and updating polymorphic data structures is important in applications such as dynamic software updating; as long as the old interface is still supported, such an update is intuitively sound. We present a new type system for ML with references that allows mutable data stored in references to be safely used polymorphically, prove its soundness and conservativity, and give an algorithm for automatic type inference.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning About Programs

General Terms
Languages, Type-Checking

Keywords
ML, references, polymorphism, reference polymorphism, alias types

1. Introduction

The interaction of type polymorphism and mutable state is a long-studied problem [1, 15, 5, 19, 8, 4, 20, 2, 14]. Both are important and desirable language features. Polymorphism supports generic programming; code that is written once can be used at many different types within the code. This flexibility is critical to authors of large systems. Imperative features such as references have conceptually easy semantics and allow programmers to write more efficient implementations of data structure. Combining these features has been difficult, and solutions that have been adopted so far limit references to be monomorphic. However, there is sometimes a need for data stored in references to be treated in a polymorphic manner. For example, in a dynamic software updating system [6], a pointer to a polymorphic data structure may need to be replaced by a new implementation which fixes bugs or includes new features.

We present a type system that permits the creation, update, and use of references to polymorphic data. We prove its soundness, demonstrating that well-typed programs do not go wrong, show that it is conservative with respect to Standard ML’s type system, and give a practical type inference algorithm for ML programs. Our type system retains the character of ML-style polymorphism, but extends the use of polymorphism to values in the store. We refer to this language feature as reference polymorphism.

Traditional type systems for ML cannot handle reference polymorphism. The naive approach is unsound; consider the following example, due to Tofte [15]:

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

In the above code, the variable `r` is bound to a reference to the polymorphic identity function, which is then updated with the successor function on integers. Finally, we dereference `r` and apply it to the boolean value `true`. In the naive type system, the reference `r` is given the polymorphic type `∀α. (α → α) ref` and so `r` can be treated as both a value of type `int → int` for the assignment statement and a value of type `bool → bool` ref for the dereference statement. The above code thus type-checks; however, the evaluation of the expression `(!r) true` attempts to increment a boolean value, causing a runtime type error. The naive way of typing references is therefore unsound.

Tofte identifies this as a problem of generalization: namely, the type of the reference to the identity function should not have been generalized. The value restriction, which limits polymorphism to syntactic let-bound values, is the solution adopted by Standard ML [20]. In practice, most programs do not violate the value restriction and those that do can be modified to be legal. This restricts references to be monomorphic.

We instead identify the above counter-example as a problem of assignment. The function stored in reference `r` is used as a function on both integers and booleans. However, the reference `r` is also assigned to function on integers. The above code is therefore unsound because it is updated with a value of a type more restrictive than all of its uses. If references may hold polymorphic values, we cannot necessarily store a value of type `τ` inside a reference that we assign the traditional ML type `τ ref`.

The key difficulty in prohibiting illegal assignments is keeping track of references in a static way. We cannot statically determine
how many references might be created during the run of a program. However, there are only a finite number of points in a program where a reference can be created. For example, the expression `ref e` might be used inside a loop to create an unknown number of references, but each reference it creates at that point will have similar type behavior. We assign each reference an alias $M$ and type it to $\text{ref}[M]$. This alias allows us to statically keep track of its usage and thus restrict illegal assignments.

As an example of what our type system is capable of, consider the following code segment:

```ml
let val K = ref (fn x => fn y => x)
in
K := (fn x => fn y => y); (!K) 3 4; (!K) true false;
end
```

In the above example, $K$ is a reference to the $K$ combinator from combinatory logic: it takes two arguments and returns the first. We update it with the function that takes two arguments and returns the second, and then use it at two different instances (with integers and booleans). As we will show, this example is a safe use of reference polymorphism.

The rest of the paper is structured as follows: we give more motivating examples and the foundations of our system in Section 2, give commentary on related work in Section 5 and give a brief discussion of applications and future work in Section 6.

2. Understanding Reference Polymorphism

To gain a better understanding of reference to polymorphic objects we consider how references can be manipulated, what goes wrong with counter-examples (such as Tofte’s), and what goes right with successful examples. This leads us to a new approach for typing references. We refer to ML’s unsafe type system (prior to Tofte) as ML’s naive type system.

ML supports three operations on references: creation (via `ref` expressions), update (via assignment expressions), and dereferencing. The first two should be viewed similarly, as storing a value in a reference, while the third retrieves the value stored in a reference. However, Standard ML’s typing rules treat reference creation and reference update differently. Using the naive typing rules, we could create a reference and give it a polymorphic type, but assignment requires monomorphic instances of this type. As noted above, the essence of what goes wrong in Tofte’s counter-example can be summarized as:

```
let val r = ref (fn x => x)
in
(!r) true; (!r) 3
end
```

The characteristic polytype of the reference stored in $r$ is $\forall \alpha. \alpha \rightarrow \alpha$ and $r$ will be given type $\text{ref} \ M \under$ under a context mapping $M$ to $\forall \alpha. \alpha \rightarrow \alpha$. The uses of $r$ are at instances of this type.

To see the importance of dereferences in determining the characteristic polytype, consider a slight modification of Tofte’s example:

```
let val r = ref (fn x => x)
in
r := (fn n => n + 1); (!r) 3
end
```

The above code is legal in both Standard ML and our system. Here the reference $r$ is dereferenced only to the type $\text{int} \rightarrow \text{int}$, so the assignment from the identity function to the successor function is legal. The characteristic polytype for $r$ is thus $\text{int} \rightarrow \text{int}$. This example would not type-check if the reference $r$ were dereferenced and used at any type other than $\text{int} \rightarrow \text{int}$. Similarly, note that the example with the $K$ combinator would not type-check if it contained the call `(!K) 3 true`, since there would then be no characteristic polytype $\sigma$ for $K$ that would instantiate to the type $\text{int} \rightarrow \text{bool} \rightarrow \beta$ for any $\beta$.

Consider now the following example:

```
fun makerf x = ref x
```
In traditional ML this function is given the type $\forall \alpha.\alpha \rightarrow \alpha$ ref; for all $\tau$, it can take an argument of type $\tau$ and return a reference of type $\tau$ ref. To support reference types containing quantified type variables we introduce alias polymorphism, allowing quantification over aliases. For the above example, our type system will give it the type $\forall \alpha.\forall \beta.\alpha \rightarrow \beta$ ref. This type has an intuitive reading: it can be instantiated by replacing $\alpha$ with any type that contains itself. The resulting instance is of type $\tau \rightarrow \tau$ ref.

The free aliases in a type $\sigma$ with respect to a context $\Theta$, written $FA(\sigma)_{\Theta}$ is given by

$$FA(\iota)_{\Theta} = \{\}$$
$$FA(\alpha)_{\Theta} = \{\}$$

$$FA(\tau_1 \rightarrow \tau_2)_{\Theta} = FA(\tau_1)_{\Theta} \cup FA(\tau_2)_{\Theta}$$

$$FA(M \text{ ref})_{\Theta} = \{M\} \cup FA(\Theta(M))_{\Theta}$$

$$FA(\forall \alpha.\sigma)_{\Theta} = FA(\sigma)_{\Theta}$$

$$FA(\forall \alpha.\forall \beta.\sigma)_{\Theta} = \{FA(\sigma)_{\Theta} - \{M\}\} \cup FA(\alpha)_{\Theta}$$

We extend this definition to reference contexts. $FA(\Theta) = \bigcup \{FA(\Theta(M))_{\Theta} \mid M \in \text{dom}(\Theta)\}$

To be well-formed, a context cannot map any alias to a type containing that alias free. Additionally, to maintain the decidability of ML-style polymorphism, we need to ensure that we do not allow an embedding of the polymorphic $\lambda$-calculus in our system, since type reconstruction for this language is undecidable [18]. To see how this might happen, consider the type $\forall \alpha.\forall \beta.\alpha \rightarrow \beta$ ref. If this type occurs with respect to a context $\Theta$ that maps $M$ and $N$ to proper polytypes, then we have effectively embedded polytypes inside of function types. To avoid such types we introduce the judgements, $\Theta :\tau \rightarrow \sigma okp$, and $\Theta :\tau okp$, axiomatized by the rules in Figure 2. These judgements ensure that free aliases in the type are in the domain of $\Theta$ and that polymorphism is appropriately restricted to allow only ML-style polymorphic types. For similar reasons, we do not allow generalization of aliases mapped to proper polytypes from reference contexts.

We can now state what it means for a context to be well-formed:

Definition 3. A context $\Theta$ is well-formed if for all $M \in \text{dom}(\Theta)$, $M \not\in FA(\Theta(M))_{\Theta}$ and $\Theta \vdash \Theta(M) okp$.

We justify these definitions by relating them to traditional ML simple types and polytypes:

Lemma 4. For all well-formed $\Theta$:

1. if $\Theta \vdash \tau okp$ then $FA(\tau)$ is an ML simple type;
2. if $\Theta \vdash \sigma okp$ then $FA(\sigma)$ is an ML polytype.

A useful concept for manipulating reference contexts is that of a maximal alias. Intuitively, an alias $M$ is maximal in a reference context if no type in the context contains it. Its binding can be removed from the context and the resulting context is still valid. Our type inference algorithm makes use of this definition.
Definition 5. An alias $M$ is maximal in $\Theta$ if $M:\sigma \in \Theta$ (for some $\sigma$) and $M \not\in FA(\Theta)$.

The description of characteristic polytypes that we gave above uses the notions of polytype instantiation and a partial order on polytypes in terms of generality. With traditional ML polytypes, instantiation is easily defined as

$$\forall \alpha_1, \ldots, \forall \alpha_n, \sigma \triangleright \tau[\alpha_1/\alpha_1, \ldots, \alpha_n/\alpha_n]$$

and polytypes are ordered $\sigma_1 \triangleright \sigma_2$ if for all $\tau$, $\sigma_1 \triangleright \tau$ implies $\sigma_1 \triangleright \sigma_2$ ($\sigma_1$ instantiates to any type that $\sigma_2$ instantiates to). Since our types are always interpreted relative to a context, we need judgements $\Theta \vdash \sigma \triangleright \tau$ and $\Theta \vdash \alpha_1 \triangleright \alpha_2$. These are given in Figure 3. These orderings correspond to the traditional ML orderings.

One additional useful operation is that of free type variables occurring in a type. Its definition is analogous to the one for free aliases.

Definition 6. The free type variables in a type $\sigma$ with respect to a context $\Theta$, written $FV(\sigma)_{\Theta}$ is defined as follows:

$FV(\cdot)_{\Theta} = \{\}$
$FV(\alpha)_{\Theta} = \{\alpha\}$
$FV(\tau_1 \rightarrow \tau_2)_{\Theta} = FV(\tau_1)_{\Theta} \cup FV(\tau_2)_{\Theta}$
$FV(M \text{ ref})_{\Theta} = FV(\Theta(M))_{\Theta}$
$FV(\forall \alpha.\sigma)_{\Theta} = FV(\sigma)_{\Theta} \setminus \{\alpha\}$
$FV(\forall \alpha.\tau.\sigma)_{\Theta} = FV(\tau)_{\Theta} \cup FV(\sigma)_{\Theta}$

We extend this definition to variable and reference contexts:

$$FV(\gamma)_{\Theta} = \bigcup \{FV(\gamma(x))_{\Theta} \mid x \in \text{dom}(\gamma)\}$$
$$FV(\Theta) = \bigcup \{FV(\Theta(M))_{\Theta} \mid M \in \text{dom}(\Theta)\}$$

3. A Type System With Reference Contexts

In this section we give a type system for a fragment of ML that allows values in the store to be used polymorphically. The language that we are dealing with has the following syntax for expressions $e$ unmodified from Harper’s simplified presentation of Tofte’s type system [4].

| Basic Values | $b ::= 0 | 1 | \cdots | \text{true} | \text{false}$ |
| Variables | $x ::= x_0, x_1, \ldots$ |
| Values | $v ::= b | x | (x) | x.x.e$ |
| Expressions | $e ::= v | e_1 \cdot e_2 \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{ref } e \mid e_1 ::= e_2 \mid !e$ |

3.1 Type System

Our type judgement is written $\Theta; \gamma \vdash e : \sigma$ and is read as “under reference context $\Theta$ mapping aliases to polytypes and variable context $\gamma$ mapping variables to polytypes, the expression $e$ has polymorphic type $\sigma$.” We often write $\Theta; \cdot \vdash e : \sigma$ instead of $\Theta; \cdot \vdash e : \sigma$ to indicate a judgement under an empty $\gamma$. Our type system is given in Figure 4. Our type syntax is unchanged from the types presented in Section 2. Let TypeOf be the standard function mapping basic values such as integers and booleans to their types.

Our type system supports the use of polymorphic values in four places. We have the traditional let-polymorphism of ML in the (TP-LET) rule. When storing a value in a reference, either through reference creation or reference assignment (the rules (TP-REF) and (TP-ASGN)), we can place a polymorphic value in the reference. This allows us to create references to polymorphic values and later update them with values that are also polymorphic. Similarly, we can retrieve values of polymorphic type from a reference using the (TP-DEREF) rule.

The typing of an expression is done under a fixed $\Theta$; the key difficulty in constructing an automated type inference algorithm for our type system is constructing this $\Theta$ on expressions without explicit type labels in a deterministic way. These questions are investigated in more depth in Section 4.

Note that type variables not occurring in the reference context can be freely generalized with the (TP-TGEN) rule. In order to generalize type variables free in the reference context, we must first generalize the aliases that refer to it. For example, when typing the function $\text{fn } x \Rightarrow \text{ref } x$, we type the expression to $\alpha \rightarrow M \text{ ref}$ under the alias typing $M:\sigma$, but to generalize $\alpha$ we must first use the (TP-AGEN) rule to generalize $M$. Otherwise $\alpha$ will remain free in the reference context.

The value restriction is featured in the (TP-AGEN) rule: aliases can only be generalized from the types of values. If generalization of aliases in expressions were allowed, we could type Tofte’s example in the following way. The variable $r$ is given the polymorphic type $\forall \alpha \forall M:\alpha.\alpha M \text{ ref}$ and then the entire expression can be typed under the context $\Theta$ with $\Theta(N) = \text{int} \rightarrow \text{int}$ and $\Theta(S) = \text{bool} \rightarrow \text{bool}$. We thus have to take similar precautions in the generalization of aliases.

To see how our type system avoids the difficulties of the naive system when assigning a reference a polymorphic type, recall the Tofte example.

let val $r = \text{ref } (\text{fn } x \Rightarrow x)$

$\text{in } r ::= (\text{fn } n \Rightarrow n + 1); (\text{tr} \text{ true end})$

Let $\Theta = M:\forall \alpha.\alpha \rightarrow \alpha$. We can type $\text{fn } x \Rightarrow x$ to the polymorphic type $\forall \alpha.\alpha \rightarrow \alpha$; we thus can give $x$ the type $M \text{ ref}$. However,
The principal type of the value first stored in the reference refers to the example given in the introduction. As an example of the power of this type system, consider the code

\[ \forall \alpha. \alpha \]  

\[ \text{let } K = \text{ref} \ (\text{fn } x \Rightarrow \text{fn } y \Rightarrow x) \text{ in} \]  

\[ K := (\text{fn } x \Rightarrow \text{fn } y \Rightarrow y); \]  

\( (\text{if})\] 3 4; \( (\text{if})\] true false; \end{verbatim}

Let \( s_0 \equiv \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha \) and \( \Theta(M) = s_0 \). We construct a typing for the example under context \( \Theta \).

The principal type of the value first stored in the reference refers to by the variable \( K \) is \( s_1 \equiv \forall \alpha \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha \). The relation \( \Theta \vdash s_1 \supset \Theta(M) \) holds and thus it is safe to store a value of polymorphic type \( s_1 \) into references with type \( M \text{ ref} \). The reference \( K \) can therefore be given the type \( M \text{ ref} \).

The assignment to \( K \) is to a function with type \( s_2 \equiv \forall \alpha \forall \beta. \alpha \rightarrow \beta \rightarrow \beta \). We have \( \Theta \vdash s_2 \supset \Theta(M) \) and so it is safe to update \( K \).

Lastly, when we dereference and use \( K \), it is to functions with types \( \text{int} \rightarrow \text{int} \rightarrow \text{int} \) and \( \text{bool} \rightarrow \text{bool} \), we have that \( \Theta(M) \) instantiates to both of these types, and so the dereferences are safe.

### 3.2 Language Soundness

We now present soundness and conservativity results for our type system. Language soundness is the classical result that “well-typed programs do not go wrong,” while conservativity shows that if a program is typable in the original Standard ML type system, then it is also typable by our type system.

We first state some of the more important properties for the type system; these are analogous to the usual lemmas required to prove type soundness [21].

**Lemma 7 (Weakening).** Suppose \( \Theta; \gamma \vdash e : \sigma \). If \( \Theta' \supseteq \Theta \) and \( \gamma' \supseteq \gamma \), then \( \Theta'; \gamma' \vdash e : \sigma \).

---

**Figure 4.** Type System

\[
\begin{align*}
\text{TypeOf}(b) = \&\ 1 & (\text{TP-BAS}) \\
\Theta; \gamma ; b : x & \rightarrow \Theta; \gamma ; () : \text{unit} & (\text{TP-UNIT}) \\
\gamma(x) = \sigma & \rightarrow \Theta; \gamma ; x : \sigma & (\text{TP-VAR}) \\
\Theta; \gamma ; [x : \tau_1] 
\rightarrow e : \tau_2 & \rightarrow \Theta; \gamma ; \tau_1 \text{ ok} \& \Theta ; \tau_2 \text{ ok} & (\text{TP-FN}) \\
\Theta; \gamma ; \lambda x : \tau_1 \rightarrow \tau_2 & \rightarrow \Theta; \gamma ; \tau_1 \rightarrow \tau_2 & (\text{TP-AP}) \\
\Theta; \gamma ; e_1 : \tau_1 \rightarrow \tau & \Theta; \gamma ; e_2 : \tau_1 & (\text{TP-LET}) \\
\Theta; \gamma ; \text{let } x = e_1 \text{ in } e_2 : \tau & (\text{TP-REF}) \\
\Theta; \gamma ; e : \sigma & \Theta; \gamma ; \sigma \supset \Theta(M) & (\text{TP-GEN}) \\
\Theta; \gamma ; e_1 : \text{M ref} & \Theta; \gamma ; e_2 : \sigma & \Theta; \gamma ; \sigma \supset \Theta(M) & (\text{TP-DEREF}) \\
\Theta; \gamma ; e : \text{ref} ; e : M & (\text{TP-INST}) \\
\Theta; \gamma ; e : \sigma' & \Theta; \gamma ; \sigma' \supset \sigma & (\text{TP-INST}) \\
\Theta; \gamma ; e : \text{unit} & (\text{TP-ASGN}) \\
\Theta; \gamma ; e : \text{ref} ; e : \text{M ref} & (\text{TP-DEREF}) \\
\Theta; \gamma ; e : \sigma & (\text{TP-TGEN}) \\
\Theta; \gamma ; e : \forall \alpha \exists \gamma, \alpha \not\in \text{FV}(\Theta) \cup \text{FV}(\gamma) \alpha & (\text{TP-AGEN}) \\
\text{Figure 5. Operational Semantics for ML with References}
\end{align*}
\]

**Lemma 8 (Type Substitution).** Let \( \varphi \) be a substitution on type variables. If \( \Theta; \gamma \vdash e : \sigma \), then \( \Theta; \psi(\Theta); \psi(\gamma) \vdash e : \varphi(\sigma) \).

**Lemma 9 (Alias Substitution).** Let \( \psi \) be a substitution on aliases. If \( \Theta; \gamma \vdash e : \sigma \), then \( \psi(\Theta); \psi(\gamma) \vdash e : \psi(\sigma) \).

**Lemma 10 (Specialization).** Suppose \( \Theta; \gamma \vdash e : \sigma \). If \( \Theta \vdash \sigma \supset \sigma' \), then \( \Theta; \gamma \vdash e : \sigma' \).

We first show that language soundness holds with respect to the classical evaluation rules for ML with references: if an expression \( e \) is given a type \( \tau \) under our type system, then the evaluation of \( e \) does not result in a type error.

Let \( \ell \) be a location, representing a first-class reference, and \( \mu \) be a store mapping locations to values. The operational semantics for ML with references are given in Figure 5 and are originally from Harper’s simplified presentation of Tofte’s solution [4]. The judgement is \( \mu \vdash e \Rightarrow v, \mu' \), which reads “under store \( \mu \), expression \( e \) evaluates to value \( v \) and modified store \( \mu' \).”
Like Harper [4], in order to avoid using co-induction to assign types to references [15], we introduce a store typing \( \lambda \). To support reference polymorphism our store typings map locations to polytypes, instead of mapping locations to simple types as done in previous definitions. We write \( \lambda' \supseteq \lambda \) to indicate that \( \lambda' \) is an extension of \( \lambda \), i.e. \( \text{dom}(\lambda') \subseteq \text{dom}(\lambda') \) and for all \( \ell \in \text{dom}(\lambda), \ell(\lambda) = \lambda(\ell) \), and define the functions \( \text{PV}(\lambda) \) and \( \text{FA}(\lambda) \) on \( \lambda \) in the obvious way.

We modify our typing judgement to include \( \lambda \), changing it to \( \Theta; \gamma; \lambda \vdash e : \sigma \) and adding the typing rule \((\text{TP}'-\text{LOC})\) to type locations:

\[
\lambda(\ell) = \Theta(M) \\
\Theta; \gamma; \lambda \vdash \ell : M \text{ ref} \quad \text{(TP}'-\text{LOC})
\]

Here, we can give the location \( \ell \) the type \( M \text{ ref} \) only if \( \lambda(\ell) \) is exactly the same as \( \Theta(M) \). We then modify the rules \((\text{TP}-\text{AGEN})\) and \((\text{TP}-\text{TGEN})\) to ensure that we do not generalize type variables and aliases occurring free in \( \lambda \).

\[
\Theta; \gamma; \lambda \vdash e : \alpha \not\in \text{FV}(\Theta) \cup \text{FV}(\gamma) \cup \text{FV}(\lambda) \quad \text{(TP}-\text{TGEN})
\]

\[
\Theta[M; \gamma]; \lambda \vdash v : \sigma \quad \Theta \vdash \tau \text{ ok}_\sigma \\
\Theta; \gamma; \lambda \vdash \forall M : \tau, \sigma \\
\Theta[M; \gamma]; \lambda \vdash v : \forall M : \tau, \sigma \quad \text{(TP}-\text{AGEN})
\]

The other rules remain the same except for the inclusion of \( \lambda \) in the judgements.

The following lemma is important to connect the typings of expressions during evaluation.

**Lemma 11 (Value Substitution).** Suppose \( \Theta; \gamma[x : \sigma_0]; \lambda \vdash e : \sigma \) and \( \Theta; \gamma; \lambda \vdash \diff v : \sigma_0 \). Then \( \Theta; \gamma; \lambda \vdash e[v/x] : \sigma \).

We now give the definition of typing a store \( \mu \) with a reference context \( \Theta \) and a store typing \( \lambda \), written \( \Theta \vdash \mu : \lambda \).

**Definition 12.** Store \( \mu \) has store typing \( \lambda \) with respect to a reference context \( \Theta \), written \( \Theta \vdash \mu : \lambda \) if \( \lambda \) is of the form \( \Theta(\lambda) \) and for all \( \ell \in \text{dom}(\mu) \), \( \Theta; \lambda \vdash \mu(\ell) : \lambda(\ell) \).

The following theorem states the main result that we need to show: given an evaluation and a corresponding typing, then the result of that evaluation also has a typing.

**Theorem 13 (Semantic Consistency).** Suppose \( \Theta \vdash e : v, \mu', \Theta; \lambda \vdash e : \sigma \), and \( \Theta \vdash \mu : \lambda \); then there exists \( \lambda' \supseteq \lambda \) such that \( \Theta \vdash \mu' : \lambda' \) and \( \Theta; \lambda' \vdash : \sigma \).

**Proof.** Proof proceeds by induction on the typing derivation.

In order to show soundness of our type system, we can add a special new value \( \text{wrong} \) to the language and extend the operational semantics given in Figure 5 to include evaluating to a type error (for example, applying a non-functional value to an argument). The special value \( \text{wrong} \) necessarily has no type: it represents a failure of our type system and a violation of language soundness.

**Theorem 14 (Soundness).** If \( \Theta \vdash e : \sigma \) and \( \vdash e : v, \mu' \) then \( v \neq \text{wrong} \).

**Proof.** Suppose \( v = \text{wrong} \). By semantic consistency, there exists \( \lambda' \) such that \( \Theta \vdash \mu' : \lambda' \) and \( \Theta; \lambda' \vdash : \sigma \). However, there is no type rule to infer that \( \text{wrong} \) has a type. The result then follows by contradiction.

### 3.3 Language Conservativity

We now outline conservativity of our type system over Wright’s type system for ML with value polymorphism: if an expression \( e \) is well-typed to a traditional ML polytype \( \sigma \), then \( e \) is well-typed under our system to a related polytype \( \sigma' \) in the extended type system with alias types.

The type grammar and the type system given in Figure 6 is the type system interpretation of Wright’s suggestion of value polymorphism [20]. Note that type variables can only be generalized for values.

In order to describe a correspondence between the two systems, it is necessary to provision translations between their corresponding types. There is a function \( | \sigma | = (\Theta, \sigma') \) which represents the translation of the traditional ML polytype \( \sigma \) into an aliased polytype \( \sigma' \) with alias types in \( \Theta \) for some newly introduced aliases \( M \).

The key difficulty in defining \( | \sigma | \) lies in translating normal polytypes of the form \( \forall \alpha \sigma \) into an aliased type \( \sigma' \), since \( \alpha \) might occur free within a type of the form \( \forall \tau \sigma \) inside of \( \sigma \). In this case, aliases containing \( \alpha \) free in them also need to be generalized. For example, \( | \forall \alpha.\alpha \rightarrow \alpha \text{ ref} | = (\emptyset, \forall \alpha.\forall M : \alpha.\alpha \rightarrow M \text{ ref}) \)

The definitions for these translations are given in the Appendix.

**Theorem 15 (Conservativity).** If \( \Gamma \vdash W e : \sigma \) and \( | \Gamma | = (\Theta, \gamma) \), then there exists some \( \Theta' \supseteq \Theta \) and \( \sigma' \) such that \( | \Theta(\sigma') | = \sigma \) and \( \Theta' : \gamma \vdash e : \sigma' \).

**Proof.** Follows by induction on the typing judgement \( \Gamma \vdash W e : \sigma \).

### 4. Algorithm

We have developed an inference algorithm based on Milner’s Algorithm \( \mathcal{W} \) [10], though a number of issues must be handled to support reference contexts. The algorithm not only generates types and substitutions, but also reference contexts and sets of instantiation constraints. Familiar operations, such as unification, are extended to support aliases.

To start we introduce a canonical form for polymorphic types which simplifies subsequent definitions. The grammar for polymorphic alias types allows interleaving of type variable abstraction and alias abstraction. Our algorithm manipulates polytypes in a restricted form.

**Definition 16.** A polytype is canonical if it is of the form \( \forall \alpha_1. \cdots. \forall \alpha_n. \forall M_1 : \tau_1. \cdots. \forall M_m : \tau_m. \tau \).

Any polytype has an equivalent canonical type (where equivalence is defined in terms of polytype ordering).
Figure 6. Typing Rules for ML with Value Polymorphism

\[
\begin{align*}
\frac{\tau ::= \alpha \mid \tau \to \tau \mid \tau \text{ ref}}{\Gamma \vdash e : \tau} & \quad (\text{WTP-REF}) \\
\frac{\sigma ::= \tau \mid \forall \alpha.\sigma}{\Gamma \vdash \text{ref} : \tau} & \quad (\text{WTP-DEREF}) \\
\frac{\Gamma(x) = \sigma}{\Gamma \vdash \text{let } x = e : \sigma} & \quad (\text{WTP-LAM}) \\
\frac{\Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} & \quad (\text{WTP-APP}) \\
\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash e : \tau} & \quad (\text{WTP-INST})
\end{align*}
\]

Type variables free in a type and not free in a context:

\[\text{type variables free in a type and not free in a context:} \]
\[\alpha \]
\[\text{free type variables occurring in the reference context. We can}
\]
\[\text{not free in the context}
\]
\[\text{noted above, these are typically the type variables free in a type}
\]
\[\text{and}
\]
\[\text{reference context and to support abstraction of aliases.}
\]
\[\text{simply generalize the type to}
\]
\[\text{does not occur free) and a type}
\]
\[\text{generalization of type}
\]
\[\text{between polytypes in the reference context and generalization of}
\]
\[\text{expressions.}
\]
\[\text{Our inference algorithm generalizes the types of references where}
\]
\[\text{possible and thus sacrifices some possible generalization of type}
\]
\[\text{variables. For the second example above, our algorithm gives}
\]
\[\text{M}
\]
\[\text{variables.}
\]
\[\text{Our inference algorithm incomplete, as our algorithm may fail to type}
\]
\[\text{example if}
\]
\[\text{used polymorphically.}
\]
\[\text{We define the abstraction of aliases containing certain type}
\]
\[\text{variables free as a way of removing occurrences of these type}
\]
\[\text{variables from a reference context. The context}\]
\[\Theta
\]
\[\text{is the one obtained by}
\]
\[\text{removing the binding for}\]
\[M
\]
\[\text{from}\]
\[\Theta
\]
\[\text{Definition 17. Given a set}\]
\[\pi
\]
\[\text{of type variables, a reference context}\]
\[\Theta
\]
\[\text{and a polytype}\]
\[\sigma
\]
\[\text{, the alias abstraction operation}\]
\[\pi[(\Theta, \sigma)]
\]
\[\text{is given by}
\]
\[\pi[(\Theta, \sigma)] = \text{[\pi[(\Theta\backslash\{M\}, \forall M:\tau.\sigma)] for } M, \tau, \alpha \in \Theta \text{ such that}
\]
\[\text{M is rightmost and maximal with some}
\]
\[\alpha
\]
\[\text{such that}
\]
\[\Theta, \alpha \text{ if } \forall M : \tau \in \Theta \text{ such that } M \text{ is rightmost and}
\]
\[\text{maximal with some}\]
\[\alpha
\]
\[\text{Note that this operation}
\]
\[\text{quantifies over aliases in } \Theta \text{ that have}
\]
\[\text{simple types.}
\]
\[\text{The resulting context returned by this function may still have}
\]
\[\text{free occurrences of some of the } \alpha.
\]
\[\text{The choice of the}
\]
\[\text{rightmost alias is done to make this operation deterministic.}
\]
\[\text{The definition of}\]
\[\text{CLOSE}\]
\[\text{can now be given. We parameterize it with}
\]
\[\text{respect to the expression whose type is being closed so that}
\]
\[\text{we
can account for the value restriction: only values can have aliases abstracted.

**Definition 18.** Given a type \( \tau \), reference context \( \Theta \), and variable context \( \gamma \), closing \( \tau \) with respect to \( \Theta \) and \( \gamma \), is given by two cases:

\[
\text{CLOSE}_{\Theta, \gamma}(\tau, v) = (\Theta', \forall \beta, \forall \gamma \beta) \text{ where } \{\alpha_1, \ldots, \alpha_n\} = \text{FV}(\tau)_{\Theta} \setminus \text{FV}(\gamma)_{\Theta}, \\
\{\Theta', \beta'\} = \text{Σ}(\Theta, \tau) \text{ and } \\
\{\beta_1, \ldots, \beta_k\} = \{\alpha_1, \ldots, \alpha_n\} \setminus \text{FV}(\Theta')
\]

**Definition 19.** Let \( \Theta(\alpha, \tau, \Theta) = ID[\alpha \mapsto \tau] \) if \( \alpha \notin \text{FV}(\tau)_{\Theta} \), 
FAIL otherwise.

The first case generalizes the type of a value and the second case handles all other expressions.

The instantiation operation, \( \text{INST}(\sigma) \), instantiates a polytype with fresh type variables and aliases, producing a reference context and a simple type.

\[
\text{INST}(\forall \alpha_0, \ldots, \forall \alpha_n, \forall M_1: \tau_1, \ldots, \forall M_m: \tau_m, \tau) = \\
\text{let } \phi = [\alpha'_1 \mapsto \alpha_1, \ldots, \alpha'_n \mapsto \alpha_n, \\
M'_1 \mapsto M_1, \ldots, M'_m \mapsto M_m] \\
\text{for fresh } \alpha'_1, \ldots, \alpha'_n, M'_1, \ldots, M'_m \\
in \{\{M'_1 : \phi(\tau_1), \ldots, M'_m : \phi(\tau_m)\}, \phi(\tau)\}
\]

The unify function, \( \text{UNIFY}(\tau, \tau', \Theta) \), produces a substitution \( \phi \) (if one exists) that unifies the two types relative to \( \Theta \). Otherwise the operation fails. Substitutions map type variables to simple types and aliases to other aliases. The operation \( \Theta(\phi) \) yields the substitution given by

\[(\Theta(\phi))(x) = \Theta(\phi(x)) \text{ for } x \text{ a type variable or alias}
\]

The operation \( \Theta(\phi) \) refers to the reference context given by

\[(\Theta(\phi))(M) = \Theta(\phi(M)).
\]

We also need a specialized form of unification called \( \text{UNIFYM} \). This version is for unifying a type that should be an alias type. The function just needs to match a type \( \tau \) with the pattern \( M \) ref.

The unify functions are given in Figure 7. Note that unification operates only on simple types. If in the process of trying to unify two references we obtain a polytype via \( \Theta \), the algorithm fails.

**Lemma 19.** If \( \text{UNIFY}(\tau, \tau', \Theta) = \phi \) then \( \phi \tau = \phi \tau' \) and
\[
\Theta(\phi)(\Theta(\tau)) = \Theta(\phi)(\Theta(\tau')).
\]

In addition to the unify function, we also need a function that given two polytypes returns a polytype less general than both of them. Again, this must be done with respect to a reference context. The **common polytype** operation \( \text{COMPOLY}(\sigma_1, \sigma_2, \Theta) \) returns such a polytype \( \sigma \) if one exists. Its definition is given in Figure 8.

As a simple example of this operation, recall the \( K \) combinator example. In it we store values with types \( \forall \alpha, \forall \beta, \alpha \rightarrow \beta \rightarrow \alpha \) and \( \forall \alpha, \forall \beta, \alpha \rightarrow \beta \rightarrow \beta \) in the reference. Applying \( \text{COMPOLY} \) operation on these types (with any \( \Theta \)) returns the type \( \forall \alpha, \alpha \rightarrow \alpha \rightarrow \alpha \) (and the empty substitution).

The union operation, \( \Theta_1 \uplus \Theta_2 \), merges two reference contexts, performing the \( \text{COMPOLY} \) operation when both contexts have a binding for the same alias. The definition is given in Figure 9. In each case, the alias \( M \) is maximal in its context. In the second and third cases, \( M \) occurs in the domain of only one context.

**Lemma 20.** If \( \Theta_1 \uplus \Theta_2 = (\Theta, \phi) \) then for all \( \sigma \),

1. \( \phi \Theta_1 \vdash \phi \sigma \Rightarrow \phi \Theta_1(M) \) implies \( \Theta \vdash \phi \sigma \Rightarrow \Theta(M) \) for all \( M \in \text{dom}(\Theta_1) \); and
2. \( \phi \Theta_2 \vdash \phi \sigma \Rightarrow \Theta_2(M) \) implies \( \Theta \vdash \phi \sigma \Rightarrow \Theta(M) \) for all \( M \in \text{dom}(\Theta_2) \).

Our type inference algorithm generates reference contexts \( \Theta \) and constraint contracts \( \mathcal{C} \). A contract is the set of all the instantiation constraints \( \Theta \vdash \Theta(M) \Rightarrow \tau \) that must be satisfied so far. These
are represented in \( C \) as simply \( M: \tau \). Note that a contract can contain multiple constraints for an alias. We introduce the notion of a context satisfying contracts.

**Definition 21.** \( \Theta \models C \) if for all \( M \) such that \( \Theta(M) = \sigma \) and \( M: \tau \in C \), \( \Theta \vdash \sigma : \tau \).

Our algorithm needs to check that a reference context satisfies a contract. For that we have the operation \( \text{CHECK} \). The function ensures (if possible) that the context satisfies the contract, introducing new alias bindings and substitutions (due to unification) where necessary.

**Definition 22.** The function \( \text{CHECK}(\Theta, C) \) is given by:

\[
\text{CHECK}(\Theta, C \cup \{ M: \tau \}) = \begin{cases} \Theta & \text{if } M \in \text{dom}(\Theta) \\
\text{let } (-, \phi) = \text{COMPOLY}(\Theta(M), \tau, \Theta) \\
\text{in } \text{CHECK}(\Theta, (C \cup (\phi \circ \phi))) \\
\text{else } \text{CHECK}(\Theta, C) \end{cases}
\]

\[
\text{CHECK}(\Theta, \{ \}) = \text{ID}
\]

The above will fail if the call to \( \text{COMPOLY} \) fails; specifically, if there is no entry \( M: \tau \) in the contract \( C \) such that \( \Theta(M) \) does not instantiate to \( \tau \).

**Lemma 23.** If \( \text{CHECK}(\Theta, C) = \phi \) then \( \phi \Theta \models \phi \).

We also need to check that a type is simple, and if not, make it simple by replacing polytypes in a context with simple types. The following function does that.

**Definition 24.** The function \( \text{ok}_{\alpha}(\Theta, \tau) \) is given by:

\[
\text{ok}_{\alpha}(\Theta, \tau) = \begin{cases} \Theta & \text{if } \alpha \not\succ \tau' \in \text{dom}(\Theta) \\
\text{let } (-, \phi) = \text{COMPOLY}(\Theta(M), \tau, \Theta) \\
\text{in } \text{ok}_{\alpha}(\Theta, \tau') \\
\text{else } \text{ok}_{\alpha}(\Theta, \tau) \end{cases}
\]

\[
\text{ok}_{\alpha}(\Theta, \text{ref}) = \begin{cases} \Theta & \text{if } M \in \text{dom}(\Theta) \\
\text{let } (-, \phi) = \text{COMPOLY}(\Theta(M), \tau, \Theta) \\
\text{in } \text{ok}_{\alpha}(\Theta, \Phi[M:\tau]) \\
\text{else } \text{ok}_{\alpha}(\Theta, \tau) \end{cases}
\]

**Lemma 25.** If \( \text{ok}_{\alpha}(\Theta, \tau) = \Theta' \) then \( \Theta' \vdash \tau \).

Finally, we give our inference algorithm \( W_{\alpha}M \) in Figure 10. Most of the cases are modifications of the corresponding cases in Milner’s algorithm, with the addition of constructing reference contexts, and constructing and checking contracts. Constraints of the form \( \Theta \vdash \sigma \succ \Theta(M) \) do not appear explicitly in the algorithm but are handled by the \( \oplus \) operation, and thus in turn, by the \( \text{COMPOLY} \) operation. The case for \text{ref} simply adds \( M: \sigma \) to the reference context. Subsequent uses of \( M \) may result in a context mapping \( M \) to a less general type. The case for assignment generates the singleton context \( \{ M: \sigma \} \) and then unifies it with the current context. The result will result in a context \( \Theta' \) such that \( \Theta' \vdash \sigma \succ \Theta'(M) \).

For constraints \( \Theta \vdash \Theta(M) \succ \tau \) (collected in the contract \( C \)) we need to explicitly check, any time we modify the context, that these still hold. \( \oplus \) operation might have instantiated an alias’ type such that constraints previously satisfied are no longer satisfied.

Our type inference algorithm is sound: if it returns successfully then there is a corresponding type judgement in our type system.

**Theorem 26.** If \( W_{\alpha}M(\gamma, e) = (\Theta, \phi, \tau, C) \) then \( \Theta, \phi(\gamma) \vdash e : \tau \) is derivable.

The theorem follows from a more general statement which also shows that \( C \) contains all the constraints \( \Theta(M) \succ \tau \) in the deduction, and \( \Theta \models C \).

We have a prototype implementation of our algorithm written in Standard ML/NJ v110.53.

To illustrate the behavior of our algorithm we briefly step through the inference process for three examples introduced earlier. For clarity, we will not explicitly describe the substitutions or their application.

Recall the original Tofte counter-example:

\[
\text{let val } r = \text{ref } (\text{fn } x => x) \text{ in } r := (\text{fn } n => n+1); \text{ (!r) true } \text{ end}
\]

Using the last case of our algorithm, we make the recursive call \( W_{\alpha}M(\gamma, \text{ref } (\text{fn } x => x)) \). The identity function can be given the polytype \( \forall \alpha. \alpha \rightarrow \alpha \) and the call returns the type \( M \text{ ref} \), reference context \( \Theta_1 = [M: \forall \alpha. \alpha \rightarrow \alpha] \), and the empty contract \( C_1 \). The next call to \( W_{\alpha}M \) is

\[
W_{\alpha}M(\gamma, \text{ref } (\text{fn } n => n+1); \text{ (!r) true })
\]

This call will return the reference context \( \Theta_2 = [M: \text{int } \rightarrow \text{int}] \) and contract \( C_2 = [M: \text{bool } \rightarrow \beta] \). The operation \( \Theta_1 \oplus \Theta_2 \) returns the context \( \Theta = [M: \text{int } \rightarrow \text{int}] \). The function \( \text{CHECK}(\Theta, C) \) then fails because \( \Theta \not\vdash \text{int } \rightarrow \text{int} \neq \text{bool } \rightarrow \beta \), so the original expression cannot be typed.

Next consider the example

\[
\text{let val } r = \text{ref } (\text{fn } x => x) \text{ in } \text{ (!r) true }; \text{ (!r) 3 end}
\]

Again we make the recursive call \( W_{\alpha}M(\gamma, \text{ref } (\text{fn } x => x)) \) and it returns the type \( M \text{ ref} \), reference context \( \Theta_1 = [M: \forall \alpha. \alpha \rightarrow \alpha] \), and the empty contract \( C_1 \). The next call to \( W_{\alpha}M \) is

\[
W_{\alpha}M(\gamma, \text{ref } (\text{fn } x => x))
\]

This call will return the type \( \text{int} \), empty context \( \Theta_2 = [M: \text{bool } \rightarrow \beta_1, M: \text{int } \rightarrow \beta_2] \). The operation \( \Theta_1 \oplus \Theta_2 \) returns the context \( \Theta = \Theta_1 \) and the function \( \text{CHECK}(\Theta, C) \) returns with the substitution \( \phi = \{ \beta_1 \mapsto \text{bool}, \beta_2 \mapsto \text{int} \} \). This is because \( \Theta \vdash \forall \alpha. \alpha \rightarrow \alpha \succ \phi(\text{bool } \rightarrow \beta_1) \) and \( \Theta \vdash \forall \alpha. \alpha \rightarrow \alpha \succ \phi(\text{int } \rightarrow \beta_2) \). The algorithm succeeds giving the expression the type \( \text{int} \).

Finally consider the example

\[
\text{let val } r = \text{ref } (\text{fn } x => x) \text{ in } r := (\text{fn } n => n+1); \text{ (!r) 3 end}
\]

Again we make the recursive call \( W_{\alpha}M(\gamma, \text{ref } (\text{fn } x => x)) \) and it returns the type \( M \text{ ref} \), reference context \( \Theta_1 = [M: \forall \alpha. \alpha \rightarrow \alpha] \),
\[
W^M(\gamma, b) = (\{ \}, \text{ID}, \text{TypeOf}(b), \{ \})
\]

\[
W^M(\gamma, x) = \text{let } (\Theta, \tau) = \text{INST}(\gamma(x)) \text{ in } (\Theta, \text{ID}, \tau, \{ \})
\]

\[
W^M(\gamma, \lambda x.e) = \text{let } \phi \text{ be fresh}
\begin{align*}
& (\Theta, \phi, \tau, C) = W^M(\gamma[x : a], e) \\
& \Theta = \text{ok}_C(\Theta, \phi(a) \rightarrow \tau) \\
& \phi' = \phi \circ \text{CHECK}(\Theta', C) \\
& \text{in } (\Theta', \phi', (a \rightarrow \tau), (\phi')C)
\end{align*}
\]

\[
W^M(\gamma, e_1 e_2) = \text{let } (\Theta_1, \phi_1, \tau_1, C_1) = W^M(\gamma, e_1) \\
(\Theta_2, \phi_2, \tau_2, C_2) = W^M(\phi_1 \gamma, e_2) \\
C = C_1 \cup C_2 \\
(\Theta_3, \phi_3) = \Theta_1 \oplus \Theta_2 \\
\phi = \phi_1 \circ \phi_2 \circ \phi_1 \\
\Theta_4 = \text{ok}_{C_1}(\Theta_3, \phi_1) \\
\Theta_5 = \text{ok}_C(\Theta_4, \phi_2) \\
\phi' = \text{CHECK}(\Theta_3, \phi_3) \circ \phi \\
\text{let } \beta \text{ be fresh}
\begin{align*}
& \phi_4 = \text{UNIFY}(\phi' \tau_1, \phi' \tau_2 \rightarrow \beta, \phi' \Theta_5) \circ \phi' \\
& \text{in } (\phi_4 \Theta_5, \phi_4, (\beta, \phi_3))C)
\end{align*}
\]

\[
W^M(\gamma, \text{let } x = e_1 \text{ in } e_2) = \text{let } (\Theta_1, \phi_1, \tau_1, C_1) = W^M(\gamma, e_1) \\
(\Theta_2, \phi_2, \tau_2, C_2) = W^M(\phi_1 \gamma, e_2) \\
\Theta'_{\phi_1} = \text{Close}_{\phi_1}(\Theta_2, \phi_1 \gamma)[x : \sigma], e_2) \\
C = C_1 \cup C_2 \\
(\Theta, \phi_1) = \phi_2 \Theta_1 \oplus \Theta_2 \\
\phi = \phi_3 \circ \phi_2 \circ \phi_1 \\
\phi' = \text{CHECK}(\Theta, \phi_3) \circ \phi \\
\text{in } (\Theta', \phi', \phi'/\Theta_2, C')
\]

**Figure 10.** Inference Algorithm \(W^M\)

and the empty contract \(C_1\). The next call to \(W^M\) is

\[
W^M([x : M \text{ ref}], r := (\text{fn } n \Rightarrow n + 1) ; (@ x) 3).
\]

This call will return the context \(\Theta_2 = [M : \text{int} \rightarrow \text{int}]\) and contract \(C_2 = [M : \text{int} \rightarrow \beta]\). The operation \(\Theta_1 \oplus \Theta_2\) returns the context \(\Theta = \Theta_2\) and the function \(\text{CHECK}(\Theta, C)\) succeeds with substitution \(\phi = \{ \beta \rightarrow \text{int} \}\) because \(\Theta \vdash \text{int} \rightarrow \phi(\text{int} \rightarrow \beta)\). The algorithm succeeds giving the expression the type \text{int}.

5. Related Work

Because both polymorphism and imperative features are very desirable features to include in a language, there has been a large amount of work in combining them in a safe way and it would be a monumental task to summarize the entire field. In this section, we give a brief survey of the most important works and those most relevant to ours.

The work most directly related to ours is in an unpublished draft by Fritz Hengellein [5], where he suggests typing references to regions \(\rho\), where \(\rho\) is a region of memory. Function types are extended to have syntax \((\Theta, \gamma \rightarrow \tau)\), where \(\Theta\) indicates the changes that a function might cause to the store, while regions are generalized with the syntax \(\gamma \rightarrow \beta\). Restrictions on \(\rho\) are thus placed in the function’s type, as opposed to being part of the syntax of the polytype. On monomorphic references, the type system presented is essentially equivalent to ours. For example, the typing rules for the imperative features of ML in the monomorphic presentation of the system are:

\[
\Theta(\rho) = \tau \\
\Theta, \gamma \vdash x : \tau \vdash \text{ref } x : \rho \quad \text{(HENG-TP-REF)}
\]

\[
\Theta(\rho) = \tau \\
\Theta, \gamma \vdash x : \rho \vdash \text{let } x := y : \text{unit} \quad \text{(HENG-TP-ASGN)}
\]

This has a close correspondence with the monomorphic version of our rules (TP-REF), (TP-DEREF), and (TP-ASGN). Our type system with reference polymorphism properly subsumes this work.

Tofte classifies type variables as either applicative of imperative [15]. For values, let-polymorphism allows full generalization of both sorts of variables. However, let-polymorphism generalizes applicative type variables for expressions, but restricts imperative type variables to be monomorphic. Harper later presents a simplified syntactic presentation of the soundness of Tofte’s system, avoiding the use of co-induction in the proof of language soundness [4].

Our work is similar in that free type variables in the reference context correspond with imperative type variables. For values, first aliases containing type variables can be generalized followed by the type variables. Once the aliases that refer to a type variable \(\alpha\) are generalized, then the type variable can be freely generalized. We therefore believe that the our type system with reference monomorphism is equivalent to Tofte’s system.
Previous versions of the Standard ML of NJ Compiler modify type variables to have strength $\alpha^n$ called weak polymorphism [2, 11]. Imperative type variables have strength 0, applicative variables have strength $\infty$, while other type variables have strength $n$, indicating that after $n$ function applications, a reference will be created with type $\alpha$. During generalization of the types of expressions, variables of strength greater than 0 are allowed to be generalized. For example, the function $\text{fn } x \Rightarrow x \text{ref } x$ has type $\alpha^{\infty} = \alpha^n$. After application to an argument of type $\beta$ list (it is convention not to explicitly show the strength of infinite variables), the result has type $\beta^n \text{list ref}$, thus preventing the generalization of this type. This system is stronger than our system for monomorphic references; for example, it permits generalization of partial applications of the function $\text{fn } f \Rightarrow \text{fn } x \Rightarrow \text{ref } (f \times x)$. However, displaying the strength of variables to the user reveals implementation details and results in more confusing error messages.

Leroy [8, 7] marks type variables that occur inside of a type $\tau \text{ref}$ as dangerous. He then keeps track of the free variables inside of functions to ensure that no functions which might have a hidden reference such as $\text{fn } x \Rightarrow (y := x \times x)$ are generalized. This approach treats generalization of the types of expressions and the types of values identically. However, the extra type information included in function types complicates showing conservativity for even the functional fragment of ML.

Our work bears some similarity with the area of effect systems [9, 16] in methodology. We use aliases in the same way that regions are used in effect systems: each represents a way of statically approximating the store. In our work, we are concerned with ensuring that creation, dereferencing, and updates of specially marked segments of code, the references, are done in a type-safe fashion. Most of the issues relevant to the effect community such as safe allocation and deallocation are orthogonal to this paper. Because we deal with the same base ML language, as long as a program using polymorphic references has been judged as type-safe, how these references are allocated and deallocated can be done independently from our type system.

6. Final Comments

We now discuss some areas of future work.

6.1 Explicit Type Declaration and The Algorithm

A key difficulty that we face when writing an algorithm is the need to choose when a reference should be treated as polymorphic or monomorphic. If the polymorphic types of references were explicitly declared, we believe that it should be possible to prove the completeness of our algorithm over our type system. This should provide a simpler way to verify the correctness of programs with reference polymorphism written in other languages where types are explicitly declared.

We will also explore notions of most general types and investigate completeness issues that remain in unlabeled ML. Our type system does not currently have most general types due to a tradeoff between polymorphism in the store and polymorphism in expressions using the store. Through modifications to both the type system and the type inference algorithm, we hope to show a principal typing result in a language supporting reference polymorphism.

6.2 Dynamic Software Updating

Safely updating pointers to polymorphic functions is of key interest in the area of dynamic software updating [6, 12]. Using dynamic software updating, a long-running program can be patched to include current bug fixes and security violations. However, there is a classical difficulty in updating pointers to polymorphic functions. It is our belief that the techniques used in this paper will be helpful in developing new techniques for dynamic software update for programs using polymorphic functions for languages such as C.

We briefly outline how our system supports, at a high level, dynamic software updating of polymorphic data structures. Suppose we have the signature for a binary tree in a reference `btree_ref`, containing functions to create, search, and expand trees. This basic interface is captured in a record type $B$:

\[
B = \{ \text{create : } 'a \rightarrow 'a \text{ tree; } \text{insert : } 'a \text{ tree } \rightarrow 'a \rightarrow 'a \text{ tree; } \text{search : } 'a \text{ tree } \rightarrow 'a \rightarrow \text{bool} \}
\]

Let $T$ be the alias given to the reference storing the implementation of the binary tree. Under a label typing $\Theta$, we have $\Theta(T) = B$. Later, we write a new implementation of the binary tree with new functionality for keeping track of the size of the tree:

\[
B' = B + \{ \text{size : } 'a \text{ tree } \rightarrow \text{int} \}
\]

We have the instantiation relation $\Theta \vdash B' \succ B$ since a value of type $B'$ can do everything that a value of type $B$ can do, and so in our system it is safe to update the old implementation with the new implementation. However, our type system does not currently allow the updated code to use new features: when we assign the new implementation of the tree, it is forced to act as if it had characteristic type $B$. In order to use these new features, we would need to create a linear type system that supported reference polymorphism [17, 3]. This is an area of future work.

6.3 Modules and Signatures

One of the key problems with using weak polymorphism as implemented in the Standard ML of NJ compiler for modules was that the strength of type variables revealed implementation details of functions [20]. It is therefore of interest to ensure that the extra complexity of our type system can be easily hidden from the user.

Signatures using alias types would need to additionally have a reference context giving meaning to any reference types in it. For example, if we have in traditional ML a signature $S$ that contains a member `foo` : int ref -> int, the aliased equivalent of this would be the signature $S'$ with the member `foo` : $M$ ref -> int along with the reference context $M$ : int. Because we prohibit polymorphic references occurring in function types, any aliases occurring inside the types of functions in signatures will be strictly monomorphic. There is thus a straightforward translation of an aliased signature into a traditional ML signature, except in the case when a signature contains a reference to a polymorphic function of the form $M$ ref where the signature requires $M$ to be polymorphic. This case can be handled by explicitly declaring such references in the signature.
6.4 Conclusion

We have presented a type system and algorithm that integrates polymorphism and references, supporting the traditional let polymorphism, but also reference and assignment polymorphism. The use of characteristic reference types and alias types provides a natural and intuitive explanation for the behavior of references in the presence of polymorphism. A form of value restriction persists, but now it is isolated to the generalization of aliases (which, however, may restrict the generalization of type variables). Our type system is sound with respect to a typical operational semantics and conservative with respect to Standard ML. The types used in our system is sound with respect to a typical operational semantics and conservative with respect to Standard ML. The types used in our system are reasonably intuitive and a translation exists between them and classical Standard ML types. We believe that our work represents a significant step forward towards languages supporting reference polymorphism.

References


A. Encoding ML Polymotypes

Definition 27 (Canonical Type and Context Translations).

1. An ML Polymtype $\sigma$ translates to the unique pair $(\Theta, \sigma')$ as follows:

   $|\varepsilon| = (\emptyset, \varepsilon)$
   $|\alpha| = (\emptyset, \alpha)$
   $|\tau_1 \to \tau_2| = (\Theta \cup \Theta_2, \tau_1' \to \tau_2' \oplus \Theta_1, \tau_1')$
   $|\tau_1 \ref\tau_2| = (\Theta \cup \{\tau' \mapsto \tau_1\}, \Theta_2, \tau_1)$
   $|\tau \ref\tau_1| = (\Theta \cup \{\tau' \mapsto \tau_1\}, M', \Theta_1, \tau')$
   $|\forall \alpha.\sigma| = (\Theta', \forall \alpha.\sigma')$
   $|\alpha| = (\Theta_1, \sigma_1)$

2. Context $\Gamma$ translates to the unique pair $(\Theta, \gamma)$ as follows:

   $|\cdot| = (\emptyset, \gamma)$
   $|\Gamma, x:\sigma| = (\Theta_1 \cup \Theta_2, \gamma, x:\sigma')$
   $|\Gamma| = (\Theta_1, \gamma)$

Because of how we pick the names of aliases we add to $\Theta$, we will never have an alias $M \in \text{dom}(\Theta_1) \cap \text{dom}(\Theta_2)$ such that $\Theta_1(M) \neq \Theta_2(M)$ when we combine $\Theta_1$ and $\Theta_2$. 

Information and Computation (1997).