

Use-Based Inference of Reference Polymorphism

Dave King and John Hannan

Pennsylvania State University, University Park PA 16802, USA

Abstract

Polymorphism is an important language feature, allowing generic code to operate on many different types. However, adding mutable state to a language places many constraints on how polymorphism interacts with references. Current solutions either restrict references to be used monomorphically or require explicit type declaration and thus limit the polymorphism of data retrieved from the store. We present a type system and type inference algorithm which allows the safe polymorphic use of data in a mutable store, as well as functions which take and return such references to the store. We refer to this language feature as *reference polymorphism*.

1 INTRODUCTION

Combining polymorphism with mutable state is a long-studied problem. Polymorphic code can be written generically once and then later instantiated to operate on many different types. A mutable store allows programmers to create references to values that can change throughout the execution of the problem. However, the naive way to extend polymorphism to operate on reference values violates language soundness. Existing solutions restrict polymorphism on data retrieved from the store in various ways.

The first class of solutions treats references to polymorphic objects as second-class types; they allow creating a reference to a polymorphic value, but they do not allow retrieving a polymorphic value from that reference [13, 8, 15, 16, 12, 5]. The goal in this line of research has been on restoring soundness to the language rather than enabling polymorphism of the store. Wright's value restriction, which limits polymorphism to syntactic values, is a critical key in many of these systems.

The second class of solutions allows the creation of references to polymorphic values that can be used polymorphically, but require explicitly declaring the type of the reference [3]. For larger values and complicated objects, this can be prohibitive for the programmer. Additionally, the presence of the value restriction limits how polymorphic data retrieved from the store can be use. Using semi-explicit polymorphism, it is not always possible to restore polymorphism to data retrieved from

the store using η -expansion. This is the case when the data retrieved is a record of methods, as in functional object encodings.

Both classes of solutions restrict references to polymorphic values in severe ways. Either we cannot treat the data in the store as polymorphic at all, or we cannot treat the reference to that polymorphic data like any other data. By focusing on inferring the types of references and adding polymorphism to the imperative subset of the language, we can gain more expressivity. In this paper, we present a type system that allows references to polymorphic values to be created, dereferenced, and updated. Polymorphic behavior of references is inferred based on how they are used; specifically, the types of values stored and retrieved in the store. Finally, we present an algorithm for automatic type inference.

Throughout the paper, we refer to a reference that can be used polymorphically as a *polymorphic reference*; this is distinct from past systems combining polymorphism with references, where the terminology is used to describe references to polymorphic values that become monomorphic when accessed through the reference.

1.1 Motivating Examples

We will work within the framework of the ML language. Each of the existing solutions combining references and polymorphism does not completely infer the behavior of code creating and dereferencing polymorphic references. Consider the following code in a dialect of ML extended with operations for creating, dereferencing, and assigning polymorphic references.

```
let val rK = polyref (fn x => fn y => x)
in
  (polyderef rK) 3 4;
  (polyderef rK) true false;
  polyasgn rK := (fn x => fn y => y);
  (polyderef rK) [1,3,7] [4,8,6];
  (polyderef rK) (fn n => n * n) (fn n => n + 1)
end
```

The above code gives a very simple example of how a reference can be safely used polymorphically in our system. Here, the polymorphic K -combinator is stored in a cell that is bound to the variable rK . We use the syntax `polyref` to indicate that the value being created is a polymorphic reference; similarly, polymorphic assignment (`polyasgn`) and polymorphic dereference (`polyderef`) operators here work on polymorphic references¹. Next, we use rK at the types $int \rightarrow int \rightarrow int$ and

¹If a language supports both traditional ML references and polymorphic references, it is important each have their own creation, assignment, and dereference primitives. Otherwise, the type inference algorithm is no longer syntax-directed.

$bool \rightarrow bool \rightarrow bool$. Later, rK is updated to a combinator that takes two arguments and returns the second. The value stored in the reference is then again used at the two types:

$$\begin{aligned} & int\ list \rightarrow int\ list \rightarrow int\ list \\ & (int \rightarrow int) \rightarrow (int \rightarrow int) \rightarrow (int \rightarrow int) \end{aligned}$$

We can see that the reference which rK is bound to is used in a safe way: at any time, the types of values which are retrieved from it using dereference are instances of the types stored in it through assignment or reference creation. As our inference procedure keeps track of what polymorphic types are stored in and retrieved from reference cells, we refer to it as *use-based*.

In order to check the safety of code using polymorphic references, we check the spots at which the reference is used, separating each into two categories: storage and retrieval. If all of the types of values stored in a reference can be instantiated to all of the types of values retrieved from a reference, then that reference is used consistently. For example, if we store a value with most general type $\forall \alpha. \alpha \rightarrow \alpha$, it is consistent to later retrieve a value of type $int \rightarrow int$ from it. On the other hand, if we were to store a value of type $bool \rightarrow bool$ in a reference, it would not be consistent to later retrieve a value of type $int \rightarrow int$ from that same reference. Using this intuition, we can see that the reference rK is used consistently in the above code.

In order to keep track of polymorphic references, we type them to the distinct type $M\ pref$. Here, M is an alias by which we can keep track of how that reference is used [11]. We use capital letters M, N, S, T, \dots to indicate distinct aliases. There is a 1-1 correspondence between reference statements and aliases; each new reference statement is represented by a different alias. In the above example, let rK be assigned type $M\ pref$.

The way that code uses a reference with an alias type is given by constraints c . There are two types of constraints in our system: storage constraints, written $\sigma \succ M$, and retrieval constraints, written $M \succ \tau$. A storage constraint $\sigma \succ M$ indicates that a value with polymorphic type σ is stored in the reference cell with type $M\ pref$. A retrieval constraint $M \succ \tau$ indicates that a value with type τ has been retrieved from a reference with type $M\ pref$. The constraints on the above code are thus:

$$C = \left\{ \begin{array}{l} M \succ int \rightarrow int \rightarrow int, M \succ bool \rightarrow bool \rightarrow bool, \\ M \succ int\ list \rightarrow int\ list \rightarrow int\ list, \\ M \succ (int \rightarrow int) \rightarrow (int \rightarrow int) \rightarrow (int \rightarrow int) \\ \forall \alpha \beta. \alpha \rightarrow \beta \rightarrow \alpha \succ M, \forall \alpha \beta. \alpha \rightarrow \beta \rightarrow \beta \succ M \end{array} \right\}$$

One can verify that each of the storage constraints on M instantiates to each of the retrieval constraints on M . If, for example, C included the constraint $int\ list \succ M$, indicating that an $int\ list$ was stored inside a reference with the alias M , then C would be inconsistent since we also retrieve a functional type from M . The above

example would thus not type correctly. Formal definitions for the above are given in Section 2.

1.2 Alias Polymorphism

Functions which take and return references require some additional machinery in order to incorporate polymorphism over aliases. For example, consider the following code:

```
let val choose_first = polyref (fn x => fn y => x)
    val choose_second = polyref (fn x => fn y => y)
    val do_choose = fn (r, x, y) => (polyderef r) x y
in
    do_choose (choose_first, 3, 4);
    do_choose (choose_second, true, false)
end
```

In order for the function `do_choose` to be fully polymorphic, it should be able to accept either `choose_first` or `choose_second` as arguments. Therefore, we must have some way to abstract away the alias in the type of the function `do_choose`, similar to the abstraction of type variables in traditional ML let-polymorphism.

Suppose the variable r is assigned type $N\text{pref}$. Before generalization, the function `do_choose` has the most general type: $N\text{pref} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$. This type is read as follows: given the reference to a polymorphic function to choose between two values of the same type, we return one of those values.

However, we cannot generalize α without also generalizing the alias N used for typing `do_choose`. In order to generalize the alias in this function, we use a form of bounded polymorphism which places the constraints on a polymorphic reference in the polymorphic type of the function. The only constraint which the function `do_choose` requires is $N \succ \alpha \rightarrow \alpha \rightarrow \alpha$, i.e. all that `do_choose` does is retrieve a value of type $\alpha \rightarrow \alpha \rightarrow \alpha$ from N . We can only abstract N and generalize α if this constraint is discharged into the polymorphic type of the function. With this idea, the function `do_choose` is given the polymorphic type

$$\forall \alpha. \forall N: \{N \succ \alpha \rightarrow \alpha \rightarrow \alpha\}. N\text{pref} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$

The above polymorphic type is read as follows: for all type variables β , aliases P , and constraint sets S such that S satisfies the constraint $P \succ \beta \rightarrow \beta \rightarrow \beta$, the function `do_choose` can take on the type $P\text{pref} \rightarrow \beta \rightarrow \beta \rightarrow \beta$.

We encode constraint sets inside polytypes in this way in order to force the propagation of how polymorphic references are used inside polymorphic functions. For example, consider the following code:

```

let val f = fn x => polyref x
in
  f [1,2,3]; f true
end

```

At the time of generalization, `f` is assigned the polymorphic type $\forall\alpha\forall M:\{\alpha \succ M\}.\alpha \rightarrow M \text{ pref}$. In order for this polymorphic type to be instantiated to operate on an integer list and a boolean, there must be aliases O, P such that the top-level environment satisfies both $\text{int list} \succ O$ and $\text{bool} \succ P$.

To avoid inference complications, we restrict the types of values that can be stored and retrieved from polymorphic references to not contain polymorphic references of their own. So while we can create polymorphic references to the identity function and other first-class polymorphic functions, we cannot create a polymorphic reference to the polymorphic assignment function. This tradeoff simplifies the presentation of our system and prevents the system from allowing higher-order polymorphism through polymorphic references, while restricting the class of typeable programs. We believe that this encompasses most of the cases where reference polymorphism is needed.

1.3 Generalization of Values and Expressions

The value restriction is an important tool to ensure soundness in the presence of side effects. It should be no surprise that the value restriction is essential to soundness in our system as well. We are free to generalize both type variables and aliases in syntactic values, which do not cause side effects when evaluated. For other expressions, we can only generalize type variables not free in the context and constraint set; we are not allowed to generalize aliases. For example, consider the following code:

```

let a = (fn x => polyref x)
let r = a (fn y => y)
let c = (fn x => fn y => r) 3
in
  ...
end

```

In the above code, `a` is given the type $\forall\alpha.M:\{\alpha \succ M\}.\alpha \rightarrow M \text{ pref}$. The variable `r` must be given a monomorphic type; otherwise, the body of the `let` expression could assign `r` to two different types, creating a 0 reference at run-time. By restricting alias polymorphism to values, we avoid unsoundness. Finally, because type variables not appearing free in the constraint set or context are generalizable, we can give `c` the polymorphic type $\forall\beta.\beta \rightarrow M \text{ pref}$.

Locations	l	$::=$	l_0, l_1, \dots
Variables	x	$::=$	x, y, z, \dots
Values	v	$::=$	$b \mid l \mid \lambda x. e \mid ()$
Expressions	e	$::=$	$x \mid v \mid e_1 e_2 \mid$ $\text{polyref } e \mid \text{polyderef } e \mid \text{polyasgn } e_1 := e_2$

FIGURE 1. Syntax for ML extended with polymorphic references

1.4 Structure of the Paper

We present the formal semantics and typing rules for our system in Section 2. We outline an algorithm for automatic type inference in Section 3. Finally, we provide comments about this method’s application to other languages along with related and future work in Section 4.

2 TYPE SYSTEM

In this section we present the typing judgment for our system. We work within a fragment of the ML language containing let-polymorphism, function abstraction, application, and new operations to operate on references containing polymorphic values.

2.1 Abstract Syntax and Types

ML expressions and values are represented by e and v , respectively. The language syntax is given in Figure 1. Note that the expressions that manipulate references are designed to only work with references that can be used polymorphically. If traditional ML-style references are desired in the language, then to maintain the syntax-directed nature of ML inference, we need separate syntax for both sorts of references. How a reference is created thus affects how the type system treats it.

We next define the syntax for constraints, polytypes, and types. This is given in Figure 2. We use the notation $FV(\tau)$ to represent the free type variables appearing in a type. We overload this notation to represent the free variables of a polytype, a constraint, a constraint set, a context, and so on. Similarly, the notation $FA(\tau)$ represents the free aliases appearing in a type, $FA(\sigma)$ the free aliases in a polytype (that are not bound by a quantifier), and so on. In order to ensure that our constraints do not include polymorphic types that contain their own constraints, we introduce the notion of a simple type, a simple polytype, and a simple constraint. We say that a type is *simple* if it does not contain any types of the form $M \text{ pref}$. Similarly, a poly-

Type Variables	$\alpha ::= \alpha, \beta, \dots$
Aliases	$M ::= M, N, S, T, \dots$
Types	$\tau ::= \alpha \mid \tau_1 \rightarrow \tau_2 \mid \text{unit} \mid M \text{ pref}$
Polymorphic Types	$\sigma ::= \tau \mid \forall \alpha. \sigma \mid \forall M:C. \sigma$
Constraints	$c ::= \sigma \succ M \mid M \succ \sigma$
Constraint Sets	$C ::= \{c_0, \dots, c_l\}$

FIGURE 2. Type Syntax

$$\begin{array}{l}
\text{(IMPL-AX)} \quad \frac{c \in C}{C \supset c} \\
\text{(IMPL-TRANS)} \quad \frac{C \supset M \succ \sigma' \quad C \vdash \sigma \succ \sigma'}{C \supset M \succ \sigma} \\
\text{(INST-REFL)} \quad C \vdash \sigma \succ \sigma \\
\text{(INST-TRANS)} \quad \frac{C \vdash \sigma \succ \sigma_0 \quad C \vdash \sigma_0 \succ \sigma'}{C \vdash \sigma \succ \sigma'} \\
\text{(INST-TVAR)} \quad C \vdash \forall \alpha. \sigma \succ \sigma[\tau/\alpha] \\
\text{(INST-ALIAS)} \quad \frac{C \supset C_0[N/M]}{C \vdash \forall M:C_0. \sigma \succ \sigma[N/M]}
\end{array}$$

FIGURE 3. Instantiation and Implication Judgments

morphic type is simple if it contains no polytypes of form $\forall M:C. \sigma$. A constraint c is simple if the types and polymorphic types occurring in it are all simple. Finally, a constraint set C is simple if it contains only simple constraints. In this paper, we only consider simple constraint sets.

We have extended the notion of what a type and a polytype mean, so we must now extend the notion of what instantiation of a polytype to another polytype means. Constraints occurring inside polymorphic types set act as guards based on how they can be instantiated. Therefore, instantiation is done under a constraint context. The instantiation judgment $C \vdash \sigma \succ \sigma'$ has the following meaning: under constraint set C , polytype σ instantiates to polytype σ' . The intuition for this judgment is that constraints are generalize in σ must be satisfied by C before instantiation. We simultaneously define the judgment $C \supset c$: under constraint set C , constraint c holds. We write $C \supset C_0$ to mean that for all $c' \in C_0$, $C \supset c'$; for every constraint $c \in C_0$, either $c \in C$ or c is implied by a more general constraint c' in C . These judgments are given in Figure 3.

In this paper, we consider an extended class of substitutions ϕ that map type variables to types and aliases to other aliases.

Lemma 2.1. *Let ϕ be a substitution on type variables and aliases.*

1. *If $C \vdash \sigma \succ \sigma'$ then $\phi(C) \vdash \phi(\sigma) \succ \phi(\sigma')$.*
2. *If $C \supset C_0$ then $\phi(C) \supset \phi(C_0)$.*

2.2 Consistency of Constraints

Type judgments for expressions are conducted under a set of constraints C , a variable context γ that maps variables to polytypes, and a location-alias mapping ρ that maps locations to aliases. It is important that the constraint set C be *consistent*: the storage and retrieval constraints must match up.

Definition 2.2 (Consistency). *A constraint set C is consistent if for all pairs of storage and retrieval constraints $(\sigma \succ M, M \succ \sigma')$ such that $C \supset \sigma \succ M$ and $C \supset M \succ \sigma'$, then $C \vdash \sigma \succ \sigma'$.*

Intuitively, C is consistent if the uses of C behave as if there is some polymorphic type σ_M for each alias M and everywhere that M is used, σ_M could have been used instead². A technical note is that we do not enforce consistency on the storage constraint level, i.e. the constraint set $\{int \succ M, bool \succ M\}$ is consistent under this definition. Type soundness is still preserved under this weaker consistency condition and it simplifies the task of type inference slightly.

We state two lemmas relating to consistent constraint sets that will be useful later.

Lemma 2.3 (Implication and Consistency). *Suppose C is consistent and $C \supset C_0$. Then C_0 is consistent.*

Lemma 2.4 (Constraint Set Consistency Under Substitutions). *Suppose C is consistent. Then:*

1. *If ϕ is a substitution on type variables, then $\phi(C)$ is consistent.*
2. *If ψ is a substitution on aliases such that $\text{range}(\psi) \cap \text{dom}(C) = \emptyset$, then $\psi(C)$ is consistent.*

²The reason that we choose a constraint-based approach rather than a direct mapping from aliases to polytypes is to ensure that functions such as $\lambda x.\text{polyderef } x$ have a principal type. It is not possible to know what a reference's type should ultimately be at reference creation. Constraints avoid this difficulty by only keeping track of what types are put in or taken out of an alias.

(TP-UNIT)	$C; \gamma; \rho \vdash () : \text{unit}$
(TP-VAR)	$\frac{\gamma(x) = \sigma}{C; \gamma; \rho \vdash x : \sigma}$
(TP-LOC)	$\frac{l \in \text{dom}(\rho)}{C; \gamma; \rho \vdash l : \rho(l) \text{ pref}}$
(TP-LAM)	$\frac{C; \gamma[x : \tau_1]; \rho \vdash e : \tau_2}{C; \gamma; \rho \vdash \lambda x. e : \tau_1 \rightarrow \tau_2}$
(TP-APP)	$\frac{C; \gamma; \rho \vdash e_1 : \tau_1 \rightarrow \tau \quad C; \gamma; \rho \vdash e_2 : \tau_1}{C; \gamma; \rho \vdash e_1 e_2 : \tau}$
(TP-LET)	$\frac{C; \gamma; \rho \vdash e_1 : \sigma_1 \quad C; \gamma[x : \sigma_1]; \rho \vdash e_2 : \sigma}{C; \gamma; \rho \vdash \text{let } x = e_1 \text{ in } e_2 : \sigma}$
(TP-POLYREF)	$\frac{C; \gamma; \rho \vdash e : \sigma \quad C \supset \sigma \succ M}{C; \gamma; \rho \vdash \text{polyref } e : M \text{ pref}}$
(TP-POLYASGN)	$\frac{C; \gamma; \rho \vdash e_1 : M \text{ pref} \quad C; \gamma; \rho \vdash e_2 : \sigma \quad C \supset \sigma \succ M}{C; \gamma; \rho \vdash \text{polyasgn } e_1 := e_2 : \text{unit}}$
(TP-POLYDEREF)	$\frac{C; \gamma; \rho \vdash e : M \text{ pref} \quad C \supset M \succ \sigma}{C; \gamma; \rho \vdash \text{polyderef } e : \sigma}$
(TP-GEN-TVAR)	$\frac{C; \gamma; \rho \vdash e : \sigma \quad \alpha \notin \text{FV}(\gamma) \cup \text{FV}(C)}{C; \gamma; \rho \vdash e : \forall \alpha. \sigma}$
(TP-GEN-ALIAS)	$\frac{C \cup C_0; \gamma; \rho \vdash v : \sigma \quad M \notin \text{dom}(C) \cup \text{FA}(\gamma) \cup \text{range}(\rho) \quad \text{dom}(C_0) = \{M\}}{C; \gamma; \rho \vdash v : \forall M. C_0. \sigma}$
(TP-INST)	$\frac{C; \gamma; \rho \vdash v : \sigma \quad C \vdash \sigma \succ \sigma'}{C; \gamma; \rho \vdash v : \sigma'}$

FIGURE 4. Typing Rules for an extension of ML with Reference Polymorphism

2.3 Type System and Theorems

We give the typing rules for our system in Figure 4. The location-label mapping ρ provides a map from locations l to aliases M , and allows us to type locations. The constraint set C and the location label map ρ fulfill the role of the typically monomorphic store typing λ from the literature.

The standard properties all apply to type judgments in our system: it is preserved under weakening, substitution on types and aliases, and substitution of values for variables.

Lemma 2.5 (Weakening). *Suppose $C; \gamma; \rho \vdash e : \sigma$. If $C \subseteq C'$ and $\gamma \subseteq \gamma'$, then $C'; \gamma'; \rho \vdash e : \sigma$.*

Lemma 2.6 (Type Substitution). *Let ϕ be a substitution on type variables and*

aliases and suppose $C; \gamma; \rho \vdash e : \sigma$. Then $\phi(C); \phi(\gamma); \phi(\rho) \vdash e : \phi(\sigma)$.

Lemma 2.7 (Value Substitution). *Suppose $C; \gamma[x:\sigma_0]; \rho \vdash e : \sigma$ and $C; \gamma; \rho \vdash v : \sigma_0$. Then $C; \gamma; \rho \vdash e[v/x] : \sigma$.*

2.4 Soundness

The operational semantics for polymorphic references are identical to the operational semantics for traditional references [5]. We use the evaluation judgment $\mu \vdash e \hookrightarrow v, \mu'$: “under store μ , expression e evaluates to value v with modified store μ' ”.

We now connect the type judgment for our language to the operational semantics. We first need a notion saying that a store and a constraint set are related: the facts in the constraint set should correspond to the values contained in the store. Soundness of any language with references depends on the type system accurately capturing what values in the store can do once they. To this end, we need the retrieval constraints to match up with the values in the store μ .

Definition 2.8 (Typing a Store). *We say that under alias-location mapping ρ , store μ satisfies constraint set C , written $\rho \vdash \mu : C$, if:*

1. $\text{dom}(\rho) = \text{dom}(\mu)$
2. For each retrieval constraint $M \succ \sigma \in C$, then for all l such that $\rho(l) = M$, $C; \rho \vdash \mu(l) : \sigma$.

We are now ready to state the semantic consistency result, which connects the operational semantics with the typing judgment. It requires that there be a typing of the expression e under a consistent constraint set C .

Theorem 2.9 (Semantic Consistency). *Let C be consistent and suppose $C; \rho \vdash e : \sigma$, $\mu \vdash e \hookrightarrow v, \mu'$, and $\rho \vdash \mu : C$. Then there exists $\rho' \supseteq \rho$ such that $C; \rho' \vdash v : \sigma$ and $\rho' \vdash \mu' : C$.*

Proof. Proof follows from induction on the structure of the typing judgment $C; \rho \vdash e : \sigma$. The full details of the semantic consistency proof are given in our companion technical report [7]. The key cases in the proof are for the (TP-POLYREF), (TP-POLYASGN), and (TP-POLYDEREF) cases. In each of these cases, a value is stored or retrieved from the store μ . For the polymorphic dereference case, the consistency of C implies that the v has the type σ . For the assignment and reference creation cases, the storage constraint being satisfied implies that the store can be safely updated. \square

Soundness of polymorphic references is a direct corollary of the Theorem 2.9.

3 TYPE INFERENCE

In this section we outline a type inference algorithm for polymorphic references. As soundness in the previous section depended on the consistency of the constraint set C , our principal task in type inference is to construct a constraint set C which details all of the polymorphic types at which polymorphic references are used and ensures that their use is consistent. We restrict our type inference algorithm to operate on expressions that do not contain locations. Our type inference algorithm `INFER` takes a context and an expression and returns a type, a substitution, and a consistent constraint set. The complete details of our algorithm are given in our companion technical report [7]. We briefly outline the important components of automatic type inference here.

Computational Generalization: We introduce a generalization function `GEN`, which takes arguments τ , C , γ , and e . If e is a syntactic value, then `GEN` generalizes all of the aliases and type variables in the type of τ which are not free in γ and returns a polytype and a reduced constraint set. Otherwise, `GEN` only generalizes type variables which are not free in γ and C .

Instantiation: In order to type variables which map to polytypes, we define an instantiation function `INSTANTIATE(σ)`. The instantiation function substitutes new type variables and aliases for generalized ones in σ , returning the substituted type, an empty substitution, and the constraints on the newly introduced aliases.

Unification: The core difficulty of type inference with polymorphic references is ensuring that different parts of code use the same mutable data consistency. For example, in the application expression, we infer information about how e_1 and e_2 both use polymorphic references. If $\gamma(x) = M \text{ pref}$, we must ensure that both e_1 and e_2 use x (through the alias M) in a consistent way; we cannot have e_1 assigning x to an integer and e_2 retrieving a boolean from x . Of course, we must still perform the normal checks of the \mathcal{W} algorithm such as ensuring that function applications are being performed with function types.

Our inference algorithm uses a unification algorithm as in traditional type inference presentations along with two inference algorithms for constraint resolution. The unification function `UNIFY(τ_1, τ_2, C)` returns a substitution of type variables and aliases ϕ such that $\phi(\tau_1) = \phi(\tau_2)$ and $\phi(C)$ is also consistent. The function `UNIFYCONSTRAINTS(C)` returns a substitution on type variables ϕ such that $\phi(C)$ is consistent, while the function `UNIFYINST(σ, τ)` returns a substitution ϕ such that $\vdash \phi\sigma \succ \phi\tau$; this is a non-structural subtyping inference algorithm [10]. The `UNIFYINST` function ensures ensure that the storage and retrieval constraints for a single alias are consistent.

Type Inference: The type inference algorithm calls each of the above functions. Its primary goal is to perform the actions of the traditional \mathcal{W} algorithm along

with the appropriate calls to `UnifyConstraints`. Our inference algorithm is given in Figure 5.

Theorem 3.1 (Correctness of Infer). *If $\text{INFER}(e, \gamma) = (\tau, \phi, C)$ and γ is consistent, then $C; \phi(g) \vdash e : \tau$ and C is consistent.*

Proof. Proof proceeds by induction on the size of e and follows from the correctness of `INSTANTIATE`, `GEN`, `UNIFY`, `UNIFYCONSTRAINTS`, `UNIFYINST`. \square

4 DISCUSSION

4.1 Application to Other Languages

Languages extended with a use-based inference system allow more flexibility when dealing with data structures that store and retrieve polymorphic objects. For example, in Java, a number of containers implement the `List` interface. In Java 1.5, programmers can parameterize Lists with type arguments, such as `List<String>`. However, it is not permissible to use subtyping within the type argument. Suppose $A \succ B$ and consider the following code examples in a Java-like language:

```
List<A> aList = { list of A's }
List<B> bList = { list of B's }
bList = aList           // dangerous
B b = bList.getFirst() // bList might now hold A's
                        // that are not B's
```

```
List<A> aList' = { list of A's }
List<B> bList' = { list of B's }
aList' = bList'
A a = aList.getFirst() // still safe
```

The traditional Java type system prevents both of the above approaches. With a use-based approach to inference, we can safely add the inference rule $\vdash \text{List} \langle A \rangle \succ \text{List} \langle B \rangle$. The first list reassignment in the example code is not allowed, since we cannot restrict the polymorphic type stored in a reference cell. However, the second reassignment of `aList'` to `bList'` is permitted.

For a completely use-based approach, we could design a type system where a programmer could just write code like the following:

```
List bList = { list of B's }
```

```
A a = l.getFirst();
B b = l.getFirst();
```

The inference engine would then attempt to infer the type parameter for the list `bList`. In order to ensure that this is being done safely, we will need to guard the internal state of the `List` object just as we guard other references; specifically, that what we are putting into that reference has the same type as what we are taking out. A polymorphic `List` class would then have an extended (and inferred) constraint-based signature:

```
T PolyList :
  state : M
  T getFirst()      : (M > T)
  void putFirst(T) : (T > M)
```

In the above, in order to retrieve a value of type `T` from a polymorphic list, we need the internal state (maybe an array) to be able to retrieve a value of type `T` from it. Similarly, to add an item to a polymorphic list, we need the internal state to be able to store a value of type `T`.

This type system would require annotating each of the user libraries with these usage constraints. Type inference would then involve propagating these constraints throughout a program to ensure that each polymorphic reference cell is used safely. The results would be similar to variance-based subtyping for parameteric types [6], except with more of an inference flavor to the language.

4.2 Verifying Code Security

Security-typed languages ensure that programs do not violate an explicitly written security policy [14]. We can decorate traditional polytypes with security levels l ; for example, an integer represents a user's bank account might have type `int{secret}`. Privacy levels can be safely raised, so all public data can be classified upward to secret data. We can represent this in our type system by extending the $C \vdash \sigma \succ \tau$ relation to include security types; i.e. $C \vdash \sigma\{\text{public}\} \succ \sigma\{\text{secret}\}$ and $C \vdash \sigma\{l\} \succ \sigma'\{l\}$ assuming $C \vdash \sigma \succ \sigma'$. This subtyping relation ensures that we cannot convert secret data into public data by laundering it through the store. This extension to the type system is not fine-grained enough to prevent implicit information flows without additional program decoration.

4.3 Related Work

As previously mentioned, combining polymorphism with mutable state is a large problem; we do not give a complete survey of the work done in this field. Earlier work has focused on finding typing semantics for references that were sound [9, 1, 13, 5], and later work has worked to increase the amount that polymorphism can be combined with mutable state [15, 4, 12]. With the adoption of the value restriction, Standard ML avoids more complicated type systems in favor of simpler error messages and easier type-checking.

Semi-explicit polymorphism allows for programmers to explicitly annotate polymorphic types on functions, that are then propagated through the rest of the program using inference methods [2]. Our system differs from theirs with respect to references in two key ways: firstly, we do not require explicit polymorphic type declarations. Secondly, the value restriction prevents the results of a dereference from being used polymorphically. The standard technique of η -expansion restores polymorphism in some cases, but not in others. In our generalization schema, we are able to safely generalize type variables (but not aliases) in the types of expressions.

4.4 Future Work

We are working on a prototype implementation of our system within Standard ML and hope to extend this implementation to an object-oriented language such as Java. We additionally are investigating modifying this system into a linear type system, wherein references to polymorphic objects can be created, assigned to a different type, and then used at that new type. Such a type system would allow for more safety in dynamic programming languages such as scripting languages.

4.5 Conclusion

In this paper we have investigated reference polymorphism, a language feature where polymorphic data in the store can be created, updated, and then dereferenced and used polymorphically in that context. We have given a proof of the type soundness of our approach in a superset of ML extended with new language features to manipulate polymorphic references. In the full paper, we will give our automatic type inference algorithm and discuss some applications of a use-based inference approach to other languages.

REFERENCES

- [1] DAMAS, L., AND MILNER, R. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1982), ACM Press, pp. 207–212.
- [2] GARRIGUE, J., AND REMY, D. Extending ML with semi-explicit higher-order polymorphism. In *Theoretical Aspects of Computer Software* (1997), pp. 20–46.
- [3] GARRIGUE, J., AND REMY, D. Semi-explicit first-class polymorphism for ML. *Information and Computation* 155, 1-2 (1999), 134–169.
- [4] GREINER, J. Weak polymorphism can be sound. *Journal of Functional Programming* 6, 1 (1996), 111–141.
- [5] HARPER, R. A simplified account of polymorphic references. *Information Processing Letters* 51, 4 (1994), 201–206.
- [6] IGARASHI, A., AND VIROLI, M. Variant parametric types: A flexible subtyping scheme for generics. *ACM Trans. Program. Lang. Syst.* 28, 5 (2006), 795–847.
- [7] KING, D., AND HANNAN, J. Use-based inference of reference polymorphism. Tech. rep., Pennsylvania State University, 2007.
- [8] LEROY, X., AND WEIS, P. Polymorphic type inference and assignment. In *18th symposium Principles of Programming Languages* (1991), ACM Press, pp. 291–302.
- [9] MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17 (1978), 348–375.
- [10] PALSBERG, J., WAND, M., AND O'KEEFE, P. Type inference with non-structural subtyping. *Formal Aspects of Computing* 9, 1 (1997), 49–67.
- [11] SMITH, F., WALKER, D., AND MORRISETT, J. G. Alias types. In *ESOP '00, 9th European Symposium on Programming, Berlin, Germany, March 25–April 2* (2000), Springer-Verlag, NY, pp. 366–381.
- [12] SMITH, G., AND VOLPANO, D. Polymorphic typing of variables and references. *ACM Transactions on Programming Languages and Systems* 18, 3 (May 1996), 254–267.
- [13] TOFTE, M. Type inference for polymorphic references. *Information and Computation* 89, 1 (1990), 1–34.
- [14] VOLPANO, D., SMITH, G., AND IRVINE, C. A sound type system for secure flow analysis. *Journal of Computer Security* 4, 3 (1996), 167–187.
- [15] WRIGHT, A. K. Typing references by effect inference. In *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, B. Krieg-Bruckner, Ed., vol. 582. Springer-Verlag, New York, N.Y., 1992, pp. 473–491.
- [16] WRIGHT, A. K. Simple imperative polymorphism. *Lisp and Symbolic Computation* 8, 4 (1995), 343–355.

$$\begin{aligned}
\text{INFER}(b, \gamma) &= (\text{TYPEOF}(b), \emptyset, \emptyset) \\
\text{INFER}(x, \gamma) &= \text{INSTANTIATE}(\gamma(x)) \\
\text{INFER}(\lambda x. e, \gamma) &= \text{let } \alpha \text{ be a fresh type variable} \\
&\quad (\tau, \phi, C) = \text{INFER}(e, \gamma[x:\alpha]) \\
&\quad \text{in } (\phi(\alpha) \rightarrow \tau, \phi, C) \\
\text{INFER}(e_1 e_2, \gamma) &= \text{let } (\tau_1, \phi_1, C_1) = \text{INFER}(e_1, \gamma) \\
&\quad (\tau_2, \phi_2, C_2) = \text{INFER}(e_2, \phi_1(\gamma)) \\
&\quad C = \phi_2(C_1) \cup C_2 \\
&\quad \text{let } \beta \text{ be a fresh type variable and } \phi' = \phi_2 \circ \phi_1 \\
&\quad \phi_u = \text{UNIFY}(\phi'(\tau_1), \phi'(\tau_2 \rightarrow \beta), C) \circ \phi' \\
&\quad \phi' = \text{UNIFYCONSTRAINTS}(\phi_u(C)) \circ \phi_u \\
&\quad \text{in } (\phi'(\beta), \phi', \phi'(C)) \\
\text{INFER}(\text{polyref } e, \gamma) &= \text{let } (\tau, \phi, C) = \text{INFER}(e, \gamma) \\
&\quad \text{let } M \text{ be a new alias} \\
&\quad (C', \sigma) = \text{GEN}(\tau, C, \gamma, e) \\
&\quad \text{if } \sigma \text{ is not simple then raise exception} \\
&\quad \text{in } (M \text{ pref}, \phi, C' \cup \{\sigma \succ M\}) \\
\text{INFER}(\text{polyderef } e, \gamma) &= \text{let } (\tau, \phi, C) = \text{INFER}(e, \gamma) \\
&\quad \text{let } M \text{ be a new alias and } \alpha \text{ a new type variable} \\
&\quad \phi_u = \text{UNIFY}(\tau, M \text{ pref}, C) \\
&\quad \text{if } \phi_u(\alpha) \text{ is not simple then raise exception} \\
&\quad C' = C \cup \{\phi_u(M) \succ \phi_u(\alpha)\} \\
&\quad \phi_c = \text{UNIFYCONSTRAINTS}(C') \\
&\quad \phi' = \phi_c \circ \phi_u \circ \phi \\
&\quad \text{in } (\phi'(\alpha), \phi', \phi'(C')) \\
\text{INFER}(\text{polyasgn } e_1 := e_2, \gamma) &= \text{let } (\tau_1, \phi_1, C_1) = \text{INFER}(e_1, \gamma) \\
&\quad \text{let } M \text{ be a new alias} \\
&\quad \phi_u = \text{UNIFY}(\tau_1, M \text{ pref}, C_1) \\
&\quad (\tau_2, \phi_2, C_2) = \text{INFER}(e_2, \phi_u(\phi_1(\gamma))) \\
&\quad (C_g, \sigma) = \text{GEN}(\tau_2, C_2, \phi_u(\phi_1(\gamma)), e) \\
&\quad \phi' = \phi_u \circ \phi_2 \circ \phi_1 \\
&\quad \text{if } \sigma \text{ is not simple then raise exception} \\
&\quad C' = \phi'(C_1) \cup C_g \cup \{\sigma \succ \phi'(M)\} \\
&\quad \phi_c = \text{UNIFYCONSTRAINTS}(C') \\
&\quad \text{in } (\text{unit}, \phi_c \circ \phi', \phi_c(C')) \\
\text{INFER}(\text{let } x = e_1 \text{ in } e_2, \gamma) &= \text{let } (\tau_1, \phi_1, C_1) = \text{INFER}(e_1, \gamma) \\
&\quad (C_g, \sigma) = \text{GEN}(\tau, C_1, \phi_1(\gamma), e_1) \\
&\quad (\tau, \phi_2, C_2) = \text{INFER}(e_2, \gamma[x:\sigma]) \\
&\quad C' = C_2 \cup \phi_2(C_g) \\
&\quad \phi_c = \text{UNIFYCONSTRAINTS}(C') \\
&\quad \phi' = \phi_c \circ \phi_2 \circ \phi_1 \\
&\quad \text{in } (\phi'(\tau), \phi', \phi'(C'))
\end{aligned}$$

FIGURE 5. The INFER Algorithm for Type Inference