A Simplified Account of Polymorphic References

Robert Harper
June, 1993
CMU-CS-93-169

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A proof of the soundness of Tofte's imperative type discipline with respect to a structured operational semantics is given. The presentation is based on a semantic formalism that combines the benefits of the approaches considered by Wright and Felleisen, and by Tofte, leading to a particularly simple proof of soundness of Tofte's type discipline.

This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.
1 Introduction

The extension of Damas and Milner's polymorphic type system for pure functional programs [2] to accommodate mutable cells has proved to be problematic. The naive extension of the pure language with operations to allocate a cell, and to retrieve and modify its contents is unsound [11]. The problem has received considerable attention, notably by Damas [3], Tofte [10, 11], and Leroy and Weiss [7]. Tofte's solution is based on a greatest fixed point construction to define the semantic typing relation [11] (see also [8]). This method has been subsequently used by Leroy and Weiss [7] and Talpin and Jouvelot [9]. It was subsequently noted by Wright and Felleisen [13] that the proof of soundness can be substantially simplified if the argument is made by induction on the length of an execution sequence, rather than on the structure of the typing derivation. Using this method they establish the soundness of a restriction of the language to require that let-bound expressions be values. In this note we present an alternative proof of the soundness of Tofte's imperative type discipline using a semantic framework that is intermediate between that of Wright and Felleisen and that of Tofte. The formalism considered admits a very simple and intuitively appealing proof of the soundness of Tofte's type discipline, and may be of some use in subsequent studies of this and related problems.

2 A Language with Mutable Data Structures

The syntax of our illustrative language is given by the following grammar:

```
expressions  e ::= x | l | unit | ref e | e_1 := e_2 | ! e | \lambda x . e | e_1 e_2 | let x be e_1 in e_2
values       t ::= x | l | unit | \lambda x . e
```

The meta-variable `x` ranges over a countably infinite set of variables, and the meta-variable `l` ranges over a countably infinite set of locations. In the above grammar `unit` is a constant, `ref` and `!` are one-argument primitive operations, and `:=` is a two-argument primitive operation. Capture-avoiding substitution of a value `v` for a free variable `x` in an expression `e` is written `[v/x]e`.

The syntax of type expressions is given by the following grammar:

```
monotypes  \tau ::= t | unit | \tau ref | \tau_1 \tau_2
polytypes  \sigma ::= \tau | \forall \tau . \sigma
```

The meta-variable `\tau` ranges over a countably infinite set of type variables. The symbol `unit` is a distinguished base type, and types of the form `\tau ref` stand for the type of references to values of type `\tau`. The set `FTV(\tau)` of type variables occurring freely in a polytype `\sigma` is defined as usual, as is the operation of capture-avoiding substitution of a monotype `\tau` for free occurrences of a type variable `t` in a polytype `\sigma`, written `[\tau/t]\sigma`.

A variable typing is a function mapping a finite set of variables to polytypes. The meta-variable `\gamma` ranges over variable typings. The polytype assigned to a variable `x` in a variable typing `\gamma` is `\gamma(x)`, and the variable typing `\gamma[x/\sigma]` is defined so that the variable `x` is assigned the polytype `\sigma`, and a variable `x'` is not assigned the polytype `\gamma(x')`. The set of type variables occurring freely in a variable typing `\gamma`, written `FTV(\gamma)`, is defined to be `\bigcup_{t \in \text{dom}(\gamma)} FTV(\gamma(t))`. A location typing is a function mapping a finite set of locations to monotypes. The meta-variable `\lambda` ranges over location typings. Notational conventions similar to those for variable typings are used for location typings.

Polymorphic type assignment is defined by a set of rules for deriving judgements of the form `\lambda; \gamma \vdash e : \tau`, with the intended meaning that the expression `e` has type `\tau` under the assumption that the locations in `e` have the monotypes ascribed by `\lambda`, and the free variables in `e` have the polytypes ascribed by `\gamma`. The rules of inference are given in Table 1. These rules make use of two auxiliary notions. The polymorphic instance relation `\sigma \geq \tau` is defined to hold iff `\sigma` is a polytype of the form `\forall \tau_1 . . . \forall \tau_n . \tau'` and `\tau` is a monotype of the form `\tau_1 . . . \tau_n / \tau_1' . . . \tau_n'`, where `\tau_1 , . . . , \tau_n` are monotypes. This relation is extended to polytypes by defining `\sigma \geq \sigma'` iff `\sigma \geq \tau` whenever `\sigma' \geq \tau`. The polymorphic generalization of a monotype `\tau` relative to a location typing `\lambda` and variable typing `\gamma`, `Close_{\lambda,\gamma}(\tau)`, is the polytype `\forall \tau_1 . . . \forall \tau_n . \tau`, where `FTV(\tau) \setminus (FTV(\lambda) \cup FTV(\gamma)) = \{ \tau_1 , . . . , \tau_n \}`. As a notational convenience, we sometimes write `\lambda \vdash e : \tau` for `\lambda; \emptyset \vdash e : \tau` and `Close_{\lambda}(\tau)` for `Close_{\lambda,\emptyset}(\tau)`.

The following lemma summarizes some important properties of the type system:
\[
\lambda; \gamma \vdash x : \tau \quad (\gamma(x) \supset \tau)
\]

\[
\lambda; \gamma \vdash \{ l : \tau\text{ref} \quad (\lambda(l) = \tau)
\]

\[
\lambda; \gamma \vdash \text{unit} : \text{unit}
\]

\[
\lambda; \gamma \vdash e : \tau
\]

\[
\lambda; \gamma \vdash \text{ref} e : \tau\text{ref}
\]

\[
\lambda; \gamma \vdash e_1 : \tau\text{ref} \quad \lambda; \gamma \vdash e_2 : \tau
\]

\[
\lambda; \gamma \vdash e_1 := e_2 : \text{unit}
\]

\[
\lambda; \gamma \vdash e : \tau\text{ref}
\]

\[
\lambda; \gamma \vdash \text{ref} e : \tau
\]

\[
\lambda; \gamma \vdash \{ x : \tau_1 \} \vdash e : \tau_2
\]

\[
(\lambda \notin \text{dom}(\gamma))
\]

\[
\lambda; \gamma \vdash \lambda x. e : \tau_1 \to \tau_2
\]

\[
\lambda; \gamma \vdash e_1 : \tau_2 \to \tau \quad \lambda; \gamma \vdash e_2 : \tau_2
\]

\[
\lambda; \gamma \vdash e_1 e_2 : \tau
\]

\[
\lambda; \gamma \vdash \{ x : \text{Close}_{\lambda \gamma}(\tau_1) \} \vdash e_2 : \tau_2
\]

\[
(\lambda \notin \text{dom}(\gamma))
\]

\[
\lambda; \gamma \vdash \text{let } x \text{ be } e_1 \text{ in } e_2 : \tau_2
\]

Table 1: Polymorphic Type Assignment
$$\text{Lemma 2.1}$$

1. (Weakening) Suppose that $\lambda; \gamma \vdash e : \tau$. If $l \notin \text{dom}(\lambda)$, then $\lambda[l:=e] ; \gamma \vdash e : \tau$, and if $x \notin \text{dom}(\gamma)$, then $\lambda ; \gamma[x:=e] \vdash e : \tau$.

2. (Substitution) If $\lambda ; \gamma \vdash e : \tau$ and $\lambda ; \gamma[x:=e'] \vdash e' : \tau'$, and if $(\text{Close}_{\lambda}\tau) \geq \sigma$, then $\lambda ; \gamma \vdash [e/x]e' : \tau'$.

3. (Specialization) If $\lambda ; \gamma \vdash e : \tau$ and $\text{Close}_{\lambda}\tau \geq \tau'$, the $\lambda ; \gamma \vdash e : \tau'$.

The proofs are routine inductions on the structure of typing derivations. Substitution is stated only for values, in recognition of the fact that in a call-by-value language only values are ever substituted for variables during evaluation.

### 3 Semantics and Soundness

A memory $\mu$ is a partial function mapping a finite set of locations to values. The contents of a location $l \in \text{dom}(\mu)$ is the value $\mu(l)$, and we write $\mu[l:=v]$ for the memory which assigns to location $l$ the value $v$ and to a location $l' \notin l$ the value $\mu(l')$. Notice that the result may either be an update of $\mu$ (if $l \in \text{dom}(\mu)$) or an extension of $\mu$ (if $l \notin \text{dom}(\mu)$).

The operational semantics of the language is defined by a collection of rules for deriving judgments of the form $\mu \vdash e \Rightarrow v, \mu'$, with the intended meaning that the closed expression $e$, when evaluated in memory $\mu$, results in value $v$ and memory $\mu'$. The rules of the semantics are given in Table 2.

The typing relation is extended to memories and location typings by defining $\mu : \lambda$ to hold iff $\text{dom}(\mu) = \text{dom}(\lambda)$, and for every $l \in \text{dom}(\mu)$, $\lambda \vdash l : \lambda(l)$. Notice that the typing relation is defined so that $\lambda(l)$ may mention locations whose type is defined by $\lambda$. (Compare Tofte’s account [11].) For example, suppose that $\mu$ is the memory sending location $l_0$ to $\lambda x.x + 1$, and location $l_1$ to $\lambda y.(l_0)y + 1$, and suppose that $\lambda$ is the location typing assigning the type int—int to both $l_0$ and $l_1$. The verification that $\mu : \lambda$ requires checking that $\lambda \vdash \lambda y.(l_0)y + 1 : \text{int—int}$, which requires determining the type assigned to location $l_0$ by $\lambda$.

As pointed out by Tofte [11], the memory $\mu'$ which assigns $\mu(l_0)$ to both $l_0$ and $l_1$ can arise as a result of an assignment statement. To verify that $\mu' : \lambda$ requires checking that $\lambda \vdash \mu(l_0) : \lambda(l_0)$, which itself relies on $\lambda(l_0)$! Tof te employs a “greatest fixed point” construction to account for this possibility, but no such machinery is needed here. This is the principal advantage of our formalism. (A similar advantage accrues to Wright and Felleisen’s approach [13] and was suggested to us by them.)

We now turn to the question of soundness of the type system.
Conjecture 3.1 If \( \mu \vdash e \Rightarrow v, \mu' \), and \( \lambda \vdash e : \tau \), with \( \mu : \lambda \), then there exists \( \lambda' \) such that \( \lambda \subseteq \lambda' \), \( \mu' : \lambda' \), and \( \lambda' \vdash v : \tau \).

The intention is to capture the preservation of typing under evaluation, taking account of the fact that evaluation may allocate storage, and hence introduce "new" locations that are not governed by the initial location typing \( \lambda \). Thus the location typing \( \lambda' \) is to be constructed as a function of the evaluation of \( e \), as will become apparent in the sequel.

A proof by induction on the structure of the derivation of \( \mu \vdash e \Rightarrow v, \mu' \) goes through for all cases but \textsc{bind}.

For example, consider the expression \( \text{ref} \ e \). We have \( \mu \vdash \text{ref} \ e \Rightarrow \tau, \mu'[t := v] \) by \textsc{alloc}, \( \lambda \vdash \text{ref} \ e : \tau \text{ref} \) by \textsc{ref}, and \( \mu : \lambda \). It follows from the definition of \textsc{alloc} that \( \mu \vdash e \Rightarrow v, \mu' \), and from the definition of \textsc{ref} that \( \lambda \vdash e : \tau \). So by induction there is a location typing \( \lambda' \supseteq \lambda \) such that \( \mu' : \lambda' \) and \( \lambda' \vdash v : \tau \). To complete the proof we need only check that the location typing \( \lambda'' = \lambda'[t := \tau] \) satisfies the conditions that \( \mu''[t := v] : \lambda'' \) and that \( \lambda'' \vdash t : \tau \text{ref} \), both of which follow from the assumptions and Lemma 2.1(1). The other cases follow a similar pattern, with the exception of rule \textsc{bind}.

To see where the proof breaks down, let us consider the obvious attempt to carry it through. Our assumption is that \( \mu \vdash \text{let} \ z \ = e \ in \ e_2 \Rightarrow v, \mu' \) by \textsc{bind}, \( \lambda \vdash \text{let} \ z \ = e \ in \ e_2 \Rightarrow \tau_2 \) by \textsc{let}, and \( \mu : \lambda \). It follows that \( \mu \vdash e_1 \Rightarrow v_1, \mu_1 \) for some value \( v_1 \) and some memory \( \mu_1 \), and that \( \mu_1 \vdash [t/z]e_2 \Rightarrow v, \mu' \). We also have that \( \lambda \vdash e_1 : \tau_1 \) for some monotype \( \tau_1 \), and that \( \lambda' \vdash \text{Close}_{\lambda}(\tau_1) \Rightarrow e_2 : \tau_2 \) for some monotype \( \tau_2 \). By induction there is a location typing \( \lambda_1 \supseteq \lambda \) such that \( \mu_1 : \lambda_1 \) and \( \lambda_1 \vdash v_1 : \tau_1 \). To complete the proof it suffices to show that \( \lambda_1 \vdash [t/z]e_2 : \tau_2 \). This would follow from the typing assumptions governing \( v_1 \) and \( e_2 \) by an application of Lemma 2.1(2), provided that we could show that \( \text{Close}_{\lambda_1}(\tau_1) \supseteq \text{Close}_{\lambda}(\tau_1) \). But this holds iff \( \text{FTV}(\lambda_1) \subseteq \text{FTV}(\lambda) \), which does not necessarily obtain. For example, if \( e_1 \equiv \text{ref} \ (\lambda \ x) \) and \( \tau_1 \) has the form \( (\mu \ :- \ \tau) \) ref, where \( \mu \) does not occur in \( \lambda \), then \( \text{Close}_{\lambda_1}(\tau_1) \) generalizes \( t \), whereas \( \text{Close}_{\lambda}(\tau_1) \) does not. (This observation is due to Tofte, who also goes on to provide a counterexample to the theorem [11].)

The simplest approach to recovering soundness is to preclude polymorphic generalization on the type of a let-bound expression unless that expression is a value. Under this restriction the proof goes through for all cases but \textsc{bind}.

We can readily see that \( \text{let} \ z \ = e \ in \ e_2 \Rightarrow v, \mu' \) iff \( \text{let} \ z \ = e \ in \ e_2 \Rightarrow v, \mu' \) for some monotype \( v, \mu' \), and that \( \lambda \vdash e_1 : \tau_1 \) for some monotype \( \tau_1 \). Thus the location typing \( \lambda_1 \) is to be constructed as a function of the evaluation of \( e_2 \), as will become apparent in the sequel.

Conjecture 3.1 If \( \mu \vdash e \Rightarrow v, \mu' \), and \( \lambda \vdash e : \tau \), with \( \mu : \lambda \), then there exists \( \lambda' \) such that \( \lambda \subseteq \lambda' \), \( \mu' : \lambda' \), and \( \lambda' \vdash v : \tau \).

The requirement that polymorphic let's bind values is rather restrictive. Following ideas of MacQueen (unpublished) and Damas [3], Tofte introduced a modification to the type system that admits a more flexible use of polymorphism, without sacrificing soundness. Tofte's idea is to employ a marking of type variables so as to maintain the invariant that if a type variable can occur in the type of a location in the store, then generalization on that type variable is suppressed. The set of type variables is divided into two countably infinite disjoint subsets, the \textit{imperative} and the \textit{applicative} type variables. A monotype is called \textit{imperative} iff all type variables occurring within it are imperative. The typing rule for \textit{ref} is constrained so that the type \( \tau \) of \( e \) in rule \textsc{ref} is required to be imperative. Polymorphic generalization must preserve the imperative/applicative distinction, and polymorphic instantiation is defined so that an imperative type variable may only be instantiated to an imperative monotype. In addition a restricted form of generalization, written \( \text{AppClose}_{\lambda}(\tau) \), is defined similarly to \( \text{Close}_{\lambda}(\tau) \), with the exception that only applicative type variables are generalized in the result: any imperative type variables remain free.

With the machinery of applicable and imperative types in hand, Tofte replaces the \textsc{bind} rule with the following two rules:

\[
\frac{\lambda ; \gamma \vdash e_1 : \tau_1 \quad \lambda ; \gamma \vdash [x: \text{Close}_{\lambda}(\tau_1)] \Rightarrow e_2 : \tau_2 \quad (x \not\in \text{dom}(\gamma))}{\lambda ; \gamma \vdash \text{let} \ z \ = e_1 \ in \ e_2 : \tau_2} \quad \text{(BIND-VAL)}
\]

\[
\frac{\lambda ; \gamma \vdash e_1 : \tau_1 \quad \lambda ; \gamma \vdash [x: \text{AppClose}_{\lambda}(\tau_1)] \Rightarrow e_2 : \tau_2 \quad (x \not\in \text{dom}(\gamma))}{\lambda ; \gamma \vdash \text{let} \ z \ = e_1 \ in \ e_2 : \tau_2} \quad \text{(BIND-ORD)}
\]
Thus if the let-bound expression is a value, it may be used polymorphically without restriction: otherwise only the applicative type variables may be generalized.

The idea behind these modifications is to maintain a conservative approximation to the set of type variables that may occur in the type of a value stored in memory. This is achieved by ensuring that if a type variable occurs freely in the memory, then it is imperative. The converse cannot, of course, be effectively maintained since the location typing in the soundness theorem is computed as a function of the evaluation trace. We say that a location typing is imperative if the type assigned to every location is imperative.

**Theorem 3.2** If \( \mu \vdash e \Rightarrow v, \mu' \), and \( \lambda \vdash e : \tau, \) with \( \mu : \lambda \) and \( \lambda \) imperative, then there exists \( \lambda' \) such that \( \lambda' \) is imperative, \( \lambda' \subseteq \lambda', \mu' : \lambda', \) and \( \lambda' \vdash v : \tau. \)

The proof proceeds by induction on the structure of the derivation of \( \mu \vdash e \Rightarrow v, \mu' \). Consider the evaluation rule \texttt{ALLOC}. The restriction on rule \texttt{REF} ensures that if \( \texttt{ref e : \tau \texttt{ref}, then } \tau \) is imperative. Consequently, the location typing \( \lambda' = \lambda'[x : \tau] \) is imperative since, by supposition, \( \lambda \) is imperative, and, by induction, \( \lambda' \) is imperative. The significance of maintaining the imperative invariant on location typings becomes apparent in the case of the \texttt{BIND} rules. The rule \texttt{BIND-VAL} is handled as sketched above: since \( v_1 \) is a value, it is non-expansive, consequently \( \lambda_1 = \lambda \), which suffices for the proof. The rule \texttt{BIND-ORD} is handled by observing that regardless of whether \( \lambda_1 \) is a proper extension of \( \lambda \) or not, we must have \( \text{Close}_{\lambda_1}(\tau_1) \geq \text{AppClose}_{\lambda}(\tau_1) \). for if a type variable \( t \) occurs freely in \( \lambda_1 \) but not in \( \lambda \), it must be (by induction hypothesis) imperative; and hence is not generalized in \( \text{AppClose}_{\lambda}(\tau_1) \) (by definition of \text{AppClose}). This is sufficient to complete the proof.

### 4 Conclusion

We have presented a simplified proof of the soundness of Toffe's type discipline for combining polymorphism and mutable references in ML. The main contribution is the elimination of the need for the maximal fixed point argument used by Toffe [11]. The methods considered here have been subsequently employed by Greiner to establish the soundness of the "weak polymorphism" type discipline implemented in the Standard ML of New Jersey compiler [1]. Our approach was influenced by the work of Wright and Felkeisen [13] who pioneered the use of reduction semantics to prove soundness of type assignment systems.

Several important studies of the problem of combining polymorphic type inference and computational effects (including mutable references) have been conducted in recent years. The interested reader is referred to the work of Gifford, Jouvelot and Talpin [6, 9], Leroy and Weiss [7], Wright [12], Hoang, Mitchell, and Viswanathan [5], and Greiner [4] for further details and references.

The author is grateful to Matthias Felkeisen, Andrew Wright, and John Greiner for their comments and suggestions.

### References


