A Simplified Account of Polymorphic References

Robert Harper
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

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.

Keywords: formal semantics, functional programming, programming languages, type theory, references and assignment.

1 Introduction

The extension of Damas and Milner's polymorphic type system for pure functional programs [2] to accomodate mutable cells has proved to be problematic. The naïve 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:

$$\begin{array}{llcl} \textit{expressions} & e & ::= & x \mid l \mid \mathsf{unit} \mid \mathsf{ref}\, e \mid e_1 := e_2 \mid\, !\, e \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{let}\, x\, \mathsf{be}\, e_1\, \mathsf{in}\, e_2 \\ \textit{values} & v & ::= & x \mid l \mid \mathsf{unit} \mid \lambda x.e \end{array}$$

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:

$$\begin{array}{llcl} \textit{monotypes} & \tau & ::= & t \mid \mathsf{unit} \mid \tau\, \mathsf{ref} \mid \tau_1 \to \tau_2 \\ \textit{polytypes} & \sigma & ::= & \tau \mid \forall t.\sigma \end{array}$$

The meta-variable t ranges over a countably infinite set of *type variables*. The symbol unit is a distinguished base type, and types of the form $\tau\,\mathsf{ref}$ stand for the type of references to values of type τ. The set $\mathrm{FTV}(\sigma)$

$$\lambda; \gamma \vdash x : \tau \quad (\gamma(x) \geq \tau) \tag{VAR}$$

$$\lambda; \gamma \vdash l : \tau\ \mathsf{ref} \quad (\lambda(l) = \tau) \tag{LOC}$$

$$\lambda; \gamma \vdash \mathsf{unit} : \mathsf{unit} \tag{TRIV}$$

$$\frac{\lambda; \gamma \vdash e : \tau}{\lambda; \gamma \vdash \mathsf{ref}\ e : \tau\ \mathsf{ref}} \tag{REF}$$

$$\frac{\lambda; \gamma \vdash e_1 : \tau\ \mathsf{ref} \qquad \lambda; \gamma \vdash e_2 : \tau}{\lambda; \gamma \vdash e_1 := e_2 : \mathsf{unit}} \tag{ASSIGN}$$

$$\frac{\lambda; \gamma \vdash e : \tau\ \mathsf{ref}}{\lambda; \gamma \vdash\ !e : \tau} \tag{RETRIEVE}$$

$$\frac{\lambda; \gamma[x{:}\tau_1] \vdash e : \tau_2}{\lambda; \gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \quad (x \notin \mathrm{dom}(\gamma)) \tag{ABS}$$

$$\frac{\lambda; \gamma \vdash e_1 : \tau_2 \to \tau \qquad \lambda; \gamma \vdash e_2 : \tau_2}{\lambda; \gamma \vdash e_1\ e_2 : \tau} \tag{APP}$$

$$\frac{\lambda; \gamma \vdash e_1 : \tau_1 \qquad \lambda; \gamma[x{:}\,\mathrm{Close}_{\lambda;\gamma}(\tau_1)] \vdash e_2 : \tau_2}{\lambda; \gamma \vdash \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 : \tau_2} \quad (x \notin \mathrm{dom}(\gamma)) \tag{LET}$$

Table 1: Polymorphic Type Assignment

of type variables occurring freely in a polytype σ is defined as usual, as is the operation of capture-avoiding substitution of a monotype τ for free occurrences of a type variable t in a polytype σ, written $[\tau/t]\sigma$.

A *variable typing* is a function mapping a finite set of variables to polytypes. The meta-variable γ ranges over variable typings. The polytype assigned to a variable x in a variable typing γ is $\gamma(x)$, and the variable typing $\gamma[x{:}\sigma]$ is defined so that the variable x is assigned the polytype σ, and a variable $x' \neq x$ is assigned the polytype $\gamma(x')$. The set of type variables occuring freely in a variable typing γ, written $\mathrm{FTV}(\gamma)$, is defined to be $\bigcup_{x \in \mathrm{dom}(\gamma)} \mathrm{FTV}(\gamma(x))$. A *location typing* is a function mapping a finite set of locations to monotypes. The meta-variable λ 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 τ under the assumption that the locations in e have the monotypes ascribed by λ, and the free variables in e have the polytypes ascribed by γ. 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 σ is a polytype of the form $\forall t_1. \ldots. \forall t_n.\tau'$ and τ is a monotype of the form $[\tau_1, \ldots, \tau_n/t_1, \ldots, t_n]\tau'$, where $\tau_1, \ldots, \tau_n$ 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 τ relative to a location typing λ and variable typing γ, $\mathrm{Close}_{\lambda;\gamma}(\tau)$, is the polytype $\forall t_1. \ldots. \forall t_n.\tau$, where $\mathrm{FTV}(\tau) \setminus (\mathrm{FTV}(\lambda) \cup \mathrm{FTV}(\gamma)) = \{ t_1, \ldots, t_n \}$. As a notational convenience, we sometimes write $\lambda \vdash e : \tau$ for $\lambda; \emptyset \vdash e : \tau$ and $\mathrm{Close}_{\lambda}(\tau)$ for $\mathrm{Close}_{\lambda;\emptyset}(\tau)$.

The following lemma summarizes some important properties of the type system:

Lemma 2.1

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

$$\mu \vdash v \Rightarrow v, \mu \tag{VAL}$$

$$\frac{\mu \vdash e \Rightarrow v, \mu'}{\mu \vdash \mathsf{ref}\, e \Rightarrow l, \mu'[l{:=}v]} \quad (l \notin \mathrm{dom}(\mu')) \tag{ALLOC}$$

$$\frac{\mu \vdash e \Rightarrow l, \mu'}{\mu \vdash \,!\, e \Rightarrow \mu'(l), \mu'} \tag{CONTENTS}$$

$$\frac{\mu \vdash e_1 \Rightarrow l, \mu_1 \qquad \mu_1 \vdash e_2 \Rightarrow v, \mu_2}{\mu \vdash e_1 := e_2 \Rightarrow \mathsf{unit}, \mu_2[l{:=}v]} \tag{UPDATE}$$

$$\frac{\mu \vdash e_1 \Rightarrow \lambda x.e_1', \mu_1 \qquad \mu_1 \vdash e_2 \Rightarrow v_2, \mu_2 \qquad \mu_2 \vdash [v_2/x]e_1' \Rightarrow v, \mu'}{\mu \vdash e_1\, e_2 \Rightarrow v, \mu'} \tag{APPLY}$$

$$\frac{\mu \vdash e_1 \Rightarrow v_1, \mu_1 \qquad \mu_1 \vdash [v_1/x]e_2 \Rightarrow v_2, \mu_2}{\mu \vdash \mathsf{let}\, x\, \mathsf{be}\, e_1\, \mathsf{in}\, e_2 \Rightarrow v_2, \mu_2} \tag{BIND}$$

Table 2: Operational Semantics for References

2. *(Substitution) If* $\lambda; \gamma \vdash v : \tau$ *and* $\lambda; \gamma[x{:}\sigma] \vdash e' : \tau'$, *and if* $\mathrm{Close}_{\lambda;\gamma}(\tau) \geq \sigma$, *then* $\lambda; \gamma \vdash [v/x]e' : \tau'$
3. *(Specialization) If* $\lambda; \gamma \vdash e : \tau$ *and* $\mathrm{Close}_{\lambda;\gamma}(\tau) \geq \tau'$, *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* μ is a partial function mapping a finite set of locations to values. The *contents* of a location $l \in \mathrm{dom}(\mu)$ 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 $l' \neq l$ the value $\mu(l')$. Notice that the result may either be an *update* of μ (if $l \in \mathrm{dom}(\mu)$) or an *extension* of μ (if $l \notin \mathrm{dom}(\mu)$).

The operational semantics of the language is defined by a collection of rules for deriving judgements of the form $\mu \vdash e \Rightarrow v, \mu'$, with the intended meaning that the closed expression e, when evaluated in memory μ, results in value v and memory μ'. 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 $\mathrm{dom}(\mu) = \mathrm{dom}(\lambda)$, and for every $l \in \mathrm{dom}(\mu)$, $\lambda \vdash l : \lambda(l)$. Notice that the typing relation is defined so that $\mu(l)$ may mention locations whose type is defined by λ. (Compare Tofte's account [11].) For example, suppose that μ 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 λ is the location typing assigning the type $\mathsf{int}{\to}\mathsf{int}$ to both l_0 and l_1. The verification that $\mu : \lambda$ requires checking that $\lambda \vdash \lambda y.(!\, l_0)\, y + 1 : \mathsf{int}{\to}\mathsf{int}$, which requires determining the type assigned to location l_0 by λ. As pointed out by Tofte [11], the memory μ' which assigns $\mu(l_1)$ 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 $\lambda(l_0)$! Tofte 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* λ' *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 λ. Thus the location typing λ' 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 BIND. For example, consider the expression $\mathsf{ref}\, e$. We have $\mu \vdash \mathsf{ref}\, e \Rightarrow l, \mu'[l{:=}v]$ by ALLOC, $\lambda \vdash \mathsf{ref}\, e : \tau\, \mathsf{ref}$ by REF, and $\mu : \lambda$. It follows from the definition of ALLOC that $\mu \vdash e \Rightarrow v, \mu'$, and from the definition of 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 $\lambda'' = \lambda'[l{:=}\tau]$ satisfies the conditions that $\mu'[l{:=}v] : \lambda''$ and that $\lambda'' \vdash l : \tau\, \mathsf{ref}$, both of which follow from the assumptions and Lemma 2.1(1). The other cases follow a similar pattern, with the exception of rule BIND. To see where the proof breaks down, let us consider the obvious attempt to carry it through. Our assumption is that $\mu \vdash \mathsf{let}\, x\, \mathsf{be}\, e_1\, \mathsf{in}\, e_2 \Rightarrow v, \mu'$ by BIND, $\lambda \vdash \mathsf{let}\, x\, \mathsf{be}\, e_1\, \mathsf{in}\, e_2 : \tau_2$ by LET, and $\mu : \lambda$. It follows that $\mu \vdash e_1 \Rightarrow v_1, \mu_1$ for some value v_1 and some memory μ_1, and that $\mu_1 \vdash [v_1/x]e_2 \Rightarrow v, \mu'$. We also have that $\lambda \vdash e_1 : \tau_1$ for some monotype τ_1, and that $\lambda; x{:}\mathrm{Close}_\lambda(\tau_1) \vdash e_2 : \tau_2$ for some monotype τ_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 $\lambda_1 \vdash [v_1/x]e_2 : \tau_2$. 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 $\mathrm{Close}_{\lambda_1}(\tau_1) \geq \mathrm{Close}_\lambda(\tau_1)$. But this holds iff $\mathrm{FTV}(\lambda_1) \subseteq \mathrm{FTV}(\lambda)$, which does not necessarily obtain. For example, if $e_1 = \mathsf{ref}\,(\lambda x.x)$ and τ_1 has the form $(t{\rightarrow}t)\,\mathsf{ref}$, where t does not occur in λ, then $\mathrm{Close}_\lambda(\tau_1)$ generalizes t, whereas $\mathrm{Close}_{\lambda_1}(\tau_1)$ 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 we can readily see that if $\mu \vdash v \Rightarrow v', \mu'$, then $v' = v$ and $\mu' = \mu$, and that if $\mu : \lambda$ and $\mu : \lambda_1$, with $\lambda_1 \supseteq \lambda$, then $\lambda_1 = \lambda$. Consequently, $\mathrm{Close}_{\lambda_1}(\tau_1) = \mathrm{Close}_\lambda(\tau_1)$ in the above proof sketch, and this is sufficient to complete the proof. Following Tofte [11], we deem an expression e *non-expansive* iff $\mu \vdash e \Rightarrow v, \mu'$ implies $\mu' = \mu$. By restricting the BIND rule so that e_1 is non-expansive, we ensure that $\lambda_1 = \lambda$, which suffices for the proof. Unfortunately in any interesting language this condition is recursively undecidable, and hence some conservative approximation must be used. Tofte chooses the simple and memorable condition that e_1 be a (syntactic) value.

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 *imperative* and the *applicative* type variables. A monotype is called *imperative* iff all type variables occurring within it are imperative. The typing rule for ref is constrained so that the type τ of e in rule 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 $\mathrm{AppClose}_{\lambda;\gamma}(\tau)$, is defined similarly to $\mathrm{Close}_{\lambda;\gamma}(\tau)$, with the exception that only applicative type variables are generalized in the result; any imperative type variables remain free.

With the machinery of applicative and imperative types in hand, Tofte replaces the BIND rule with the following two rules:

$$\frac{\lambda;\gamma \vdash v_1 : \tau_1 \qquad \lambda;\gamma[x{:}\mathrm{Close}_{\lambda;\gamma}(\tau_1)] \vdash e_2 : \tau_2}{\lambda;\gamma \vdash \mathsf{let}\, x\, \mathsf{be}\, v_1\, \mathsf{in}\, e_2 : \tau_2} \quad (x \notin \mathrm{dom}(\gamma)) \qquad (\textsc{bind-val})$$

$$\frac{\lambda;\gamma \vdash e_1 : \tau_1 \qquad \lambda;\gamma[x{:}\mathrm{AppClose}_{\lambda;\gamma}(\tau_1)] \vdash e_2 : \tau_2}{\lambda;\gamma \vdash \mathsf{let}\, x\, \mathsf{be}\, e_1\, \mathsf{in}\, e_2 : \tau_2} \quad (x \notin \mathrm{dom}(\gamma)) \qquad (\textsc{bind-ord})$$

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 iff 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 λ imperative, then there exists λ' such that λ' 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 ALLOC. The restriction on rule REF ensures that if $\mathsf{ref}\, e : \tau\, \mathsf{ref}$, then τ is imperative. Consequently, the location typing $\lambda'' = \lambda'[x : \tau]$ is imperative since, by supposition, λ is imperative, and, by induction, λ' is imperative. The significance of maintaining the imperative invariant on location typings becomes apparent in the case of the BIND rules. The rule 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 BIND-ORD is handled by observing that regardless of whether λ_1 is a proper extension of λ or not, we must have $\mathrm{Close}_{\lambda_1}(\tau_1) \geq \mathrm{AppClose}_{\lambda}(\tau_1)$, for if a type variable t occurs freely in λ_1 but not in λ, it must be (by induction hypothesis) imperative, and hence is not generalized in $\mathrm{AppClose}_{\lambda}(\tau_1)$ (by definition of AppClose). This is sufficient to complete the proof.

4 Conclusion

We have presented a simplified proof of the soundness of Tofte'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 Tofte [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 Felleisen [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 Felleisen, Andrew Wright, and John Greiner for their comments and suggestions.

References

[1] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[2] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[3] Luis Manuel Martins Damas. *Type Assignment in Programming Languages.* PhD thesis, Edinburgh University, 1985.

[4] John Greiner. Standard ml weak polymorphism can be sound. Technical Report CMU–CS–93–160, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1993.

[5] My Hoang, John Mitchell, and Ramesh Viswanathan. Standard ML-NJ weak polymorphism and imperative constructs. In *Eighth Symposium on Logic in Computer Science*, 1993.

[6] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Eighteenth ACM Symposium on Principles of Programming Languages*, pages 303–310, 1991.

[7] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Eighteenth ACM Symposium on Principles of Programming Languages*, pages 291–302, Orlando, FL, January 1991. ACM SIGACT/SIGPLAN.

[8] Robin Milner and Mads Tofte. Co-induction in relational semantics. Technical Report ECS-LFCS-88-65, Laboratory for the Foundations of Computer Science, Edinburgh University, Edinburgh, October 1988.

[9] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Seventh Symposium on Logic in Computer Science*, pages 162–173, 1992.

[10] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Edinburgh University, 1988. Available as Edinburgh University Laboratory for Foundations of Computer Science Technical Report ECS–LFCS–88–54.

[11] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.

[12] Andrew Wright. Typing references by effect inference. In *Proceedings of the European Symposium on Programming*, 1992.

[13] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91–160, Department of Computer Science, Rice University, July 1991. To appear, *Information and Computation*.