A Type System for a Lambda Calculus with Assignments

Kung Chen and Martin Odersky
Research Report YALEU/DCS/RR-963
May, 1993

This work is supported by DARPA grant N00014-91-J-4043.

A Type System for a Lambda Calculus with Assignments

Kung Chen and Martin Odersky

Department of Computer Science
Yale University
Box 2158 Yale Station
New Haven, CT 06520

May 12, 1993

Abstract

We present a Hindley/Milner-style polymorphic type system for λ_{var}, an extension of the call-by-name λ-calculus with mutable variables and assignments. To match λ_{var}'s explicit distinction between functional and imperative worlds, the type universe is stratified into two layers: One for applicative expressions and one for imperative state transformers. In inferring types for λ_{var}-terms, the type system perform a simple effect analysis to statically verify the safety of coercing a state transformer to a pure value. We prove the soundness of the type system with respect to λ_{var}'s untyped reduction semantics so that any well-typed program will evaluate to an answer if the evaluation halts. We also discuss some practical aspects of the type system and present a type checker based on the well-known W algorithm.

1 Introduction

We study a type system for λ_{var}, an imperative extension of λ-calculus. Unlike other such extensions [Rey88, WF93, SRI92], which are monomorphically typed, we have studied the theory of λ_{var} initially in an untyped setting [ORH93]. In this paper, we propose a polymorphic type system in the Hindley/Milner style, give a soundness result, and discuss type reconstruction.

Polymorphic type systems for functional/imperative language combinations usually treat the types of mutable variables specially: unlike the types of immutable variables they can be instantiated only once. Languages such as ML, in which every expression can have side effects, make finding the types of mutable variables non-trivial. Several type systems with various degrees of accuracy and complexity have been proposed for imperative ML [Dam85, Tof90, LW91, Wri92]. The more recent proposals have many things in common with effect-systems [TJ93].

In λ_{var}, we face a different situation. Side effects can be caused only by terms that occur in a state transformer and the results of those terms are always λ-bound. The evaluation of a **let**-bound value, on the other hand, never involves side effects. This makes the purely functional Hindley/Milner system, which generalizes only the types of **let**-bound variables a good match for λ_{var}. As a consequence, typing the **let**-construct is considerably simpler in λ_{var} than in imperative versions of ML. Like Mairson [Mai93] for functional ML we can type a program with **let**'s by expanding all **let**'s and typing the resulting program in a monomorphic type system.

On the other hand, the type system for λ_{var} has to solve a problem not present in either functional languages or ML: It has to guarantee statically that a state transformer enclosed in a **pure** construct is referentially transparent. We present here a simple method to verify this fact. The resulting type system is sound with respect to the untyped reduction semantics of λ_{var}. Similar to the approach of [WF92], this is shown by a syntactic method, based on a subject reduction theorem and a characterization of the types of irreducible terms. We also discuss some practical aspects of an implementation scheme derived from our type system.

2 Preliminaries

In this section, we briefly review the syntax and semantics of λ_{var} and refer the readers to [ORH93] for a more detailed description.

2.1 Term Syntax of λ_{var}

The syntax of λ_{var} is given in Figure 1. On the first line are productions for primitive function symbols f, data constructors c^n, identifiers x, function abstractions $x.M$[1], function applications $M_1\ M_2$ and a **let** construct. The second line adds constructs for modelling *state transformers*. A state transformer is a function that maps an initial state to some result value and a final state. States are composed of mutable variables (or: *tags*) v which are bound in terms $\mathbf{var}v.M$. Tags (denoted by the letters u, v, w) are different from immutable variables (denoted here by x, y, z), but are like them subject to α-renaming. Primitive operations on tags v are assignments $M =: v$ and reads $v?$. These operations can be combined into larger state transformers using the "bind"

[1]Note the absence of a leading λ; this is done to make reduction rules simpler to read.

$$
\begin{array}{rcl}
M & ::= & f \mid c^n \mid x \mid x.M \mid M_1\ M_2 \mid \textbf{let}\ y = N\ \textbf{in}\ M \\
& \mid & v \mid \textbf{var}\ v.M \mid M? \mid M_1 =: M_2 \mid M_1 \rhd x.M_2 \\
& \mid & \textbf{return}\ M \mid \textbf{pure}\ M
\end{array}
$$

Figure 1: Syntax of λ_{var}

β	$(x.M)\ N$	$\rightarrow$	$[N/x]\ M$	
δ	$f\ V$	$\rightarrow$	$\delta(f, V)$	($\delta(f, V)$ defined)
let	$\textbf{let}\ y = N\ \textbf{in}\ M$	$\rightarrow$	$[N/y]\ M$	
$\rhd\rhd$	$(M_1 \rhd x.M_2) \rhd y.M_3$	$\rightarrow$	$M_1 \rhd x.(M_2 \rhd y.M_3)$	
$r\rhd$	$(\textbf{return}\ N) \rhd x.M$	$\rightarrow$	$(x.M)\ N$	
$v\rhd$	$(\textbf{var}\ v.M) \rhd x.N$	$\rightarrow$	$\textbf{var}\ v.(M \rhd x.N)$	
$=:\rhd$	$(M_1 =: M_2) \rhd x.M_3$	$\rightarrow$	$M_1 =: M_2\ ;\ (x.M_3)\ ()$	$(x \in fv\ M_3)$
f	$N =: v\ ;\ v? \rhd x\ .\ M$	$\rightarrow$	$N =: v\ ;\ (x.M)\ N$	
b_1	$N =: v\ ;\ w? \rhd x\ .\ M$	$\rightarrow$	$w? \rhd x\ .\ N =: v\ ;\ M$	$(v \neq w)$
b_2	$\textbf{var}\ v\ .\ w? \rhd x\ .\ M$	$\rightarrow$	$w? \rhd x\ .\ \textbf{var}\ v\ .\ M$	$(v \neq w)$
$\bot$	$\textbf{var}\ v\ .\ v? \rhd x\ .\ M$	$\rightarrow$	$\textbf{var}\ v\ .\ v? \rhd x\ .\ M$	
p_c	$\textbf{pure}\ (S[\textbf{return}\ c^n\ M_1\ ...\ M_k])$	$\rightarrow$	$c^n\ (\textbf{pure}\ (S[\textbf{return}\ M_1]))\ ...\ (\textbf{pure}\ (S[\textbf{return}\ M_k]))$	$(k \leq n)$
p_λ	$\textbf{pure}\ (S[\textbf{return}\ x.M])$	$\rightarrow$	$x\ .\ \textbf{pure}\ (S[\textbf{return}\ M])$	
p_f	$\textbf{pure}\ (S[\textbf{return}\ f])$	$\rightarrow$	f	

Figure 2: Reduction rules for λ_{var}.

---

expression $M_1 \rhd x.M_2$ [Wad90]. We take ($\rhd$) to be right-associative and often employ the following abbreviation:

$$N\ ;\ M \stackrel{def}{=} N \rhd x.M \qquad (x \notin fv\ M).$$

The productions on the last line mediate between values and state transformers. **return** M is the state transformer that yields M without modifying the state. **pure** M reduces to the final result of state transformer M, provided M has no global side effects.

The type system for λ-var turns state transformers into a Kleisli monad with ($\rhd$) as the bind operator and with **return** as unit. Untyped λ-var on the other hand, satisfies only two of the three laws of a Kleisli monad ($\rhd$ is associative and **return** is a left-unit).

2.2 Operational Semantics of λ_{var}

The reduction rules given in Figure 2 define an operational semantics for λ_{var}.

We use $bv\ M$ ($fv\ M$) to denote the bound (free) variables and tags in a term M. A term is *closed* if $fv\ M = \emptyset$. We use $M \equiv N$ for syntactic equality of terms (modulo α-renaming) and reserve $M = N$ for convertibility. We assume everywhere the "hygiene" convention of Barendregt [Bar84] that bound and free variables and tags in a term are distinct.

A *value* V is a λ-abstraction, a primitive function, or a (possibly applied) constructor. An *observable value* (or *answer*) A is an element of some nonempty subset of the basic constants. A *context* is a term with a hole [ ] in it. A *prefix* P is a context of one of the forms

$$P \ ::= \ [\,] \ \mid \ \mathbf{var}\ v.P \ \mid \ M \triangleright x.P\,.$$

A *pre-state prefix* R is a "normalized" prefix

$$R \ ::= \ [\,] \ \mid \ \mathbf{var}\ v.R \ \mid \ M =: v\ ;\ R\,,$$

and a *state prefix* S is a pre-state prefix that satisfies in addition the requirement that $wr\ S \subseteq bv\ S$. The set of variables *written* in S, $wr\ S$, is defined as follows:

$$\begin{array}{lcl} wr\ [\,] & = & \emptyset \\ wr\ (\mathbf{var}\ v.S) & = & wr\ S \\ wr\ (M =: v\ ;\ S) & = & \{v\} \cup wr\ S. \end{array}$$

Rules (β), (δ), and (*let*) are the standard rules for reducing terms in an applied λ-calculus. Rules ($\triangleright\triangleright$) and ($r\triangleright$) represent two of the three laws of a Kleisli monad. Rule ($v\triangleright$) extends the scope of a tag over a ($\triangleright$) to the right. Rule ($=:\triangleright$) passes (), the result value of an assignment, to the term that follows the assignment. Rules (f), (b_1) and (b_2) deal with assignments. The fusion rule (f) reduces a pair of an assignment and a dereference with the same tag. The bubble rules (b_1) and (b_2) allow variable-readers to "bubble" to the left past assignments and introductions involving other tags. Rule ($\perp$) is not present in [ORH93]; it is added here to force the evaluation of a term with uninitialized variables to diverge. This is necessary, since we do not attempt to detect unititialized variables statically.

The final three rules implement "effect masking", by which local state manipulation can be isolated for use in a purely functional context. The context condition for state prefixes S that assigned variables must be locally bound ensures that evaluation of the argument to **pure** neither affects nor observes global storage.

In [ORH93] it was shown that this notion of reduction generates a confluent equational theory which admits a standard reduction order, derived using the *evaluation contexts* E:

$$\begin{array}{rcl} E & ::= & [\,] \ \mid \ E\ M \ \mid \ f\ E \ \mid \ \mathbf{pure}\ E \ \mid \ \mathbf{pure}\ S[\mathbf{return}\ E] \\ & \mid & \mathbf{var}\ v.E \ \mid \ E? \ \mid \ M =: E \ \mid \ E \triangleright x.M \ \mid \ M =: v \triangleright x.E \end{array}$$

Such a left-to-right reduction, denoted by $\longmapsto$, defines a deterministic evaluator, $eval_{var}$, for λ_{var} in the standard manner.

3 The Type System for λ_{var}

3.1 Types

Let $\underline{\alpha}$ and α range over two disjoint denumerable sets of type variables. The syntax of λ_{var} types is as follows:

$$\begin{array}{lrcl} \text{Applicative types} & \tau & ::= & \underline{\alpha} \mid () \mid \tau \to \tau \mid \kappa^n\ \tau_1\ \ldots\ \tau_n \\ \text{Command types} & \theta & ::= & \alpha \mid \tau \mid \theta \to \theta \mid \kappa^n\ \theta_1\ \ldots\ \theta_n \mid \mathit{Cmd}\ \theta \mid \mathit{Var}\ \theta \end{array}$$

The type system is stratified into two layers. The *applicative* layer τ contains the types of state-independent terms. These applicative types include applicative type variables $\underline{\alpha}$, the unit type () and are closed under the function space and algebraic data type constructions. The *command* layer θ extends the applicative layer to type state transformers. The type $\mathit{Var}\ \theta$ is used to type store tags, while $\mathit{Cmd}\ \theta$ types state transformers with type θ. The command layer includes type variables α, all applicative types and is closed under the type constructors $\to$, κ^n, *Var* and *Cmd*.

Note that applicative type variables $\underline{\alpha}$ serve a different purpose than either *imperative* type variables [Tof90] or *weak* type variables [AM87] used in Standard ML. They are used to type polymorphic **pure** terms, not to solve the problem with polymorphic references.

3.2 Typing Rules

Rules for inferring types for λ_{var}-terms are given in Figure 3. A type judgement $\Gamma \vdash M : \theta$ states that term M has type τ in type environment Γ. A type environment is a finite set of type assumptions of the forms $x : \theta$ and $v : \mathit{Var}\ \theta$, with no variable (tag) occurring twice. We write $\mathit{dom}(\Gamma)$ for the set of variables and tags occur in Γ, and $\Gamma(a)$ for the unique θ with $a{:}\theta$ in Γ, if it exists. Assuming $a \notin \mathit{dom}(\Gamma)$, we write $\Gamma.a{:}\theta$ for $\Gamma \cup \{a : \theta\}$.

Basically, each rule corresponds to one syntactic construct in the term language. Primitive functions and data constructors are collectively called constants c. Rule (*const*) uses the function $\mathit{TypeOf} : \mathit{Const} \mapsto$ Set of *applicative types* to assign types to constants. We assume that the types associated with a constant are the set of substitution instances a single, polymorphic type. Furthermore, we impose the usual δ-typability condition to ensure type soundness for an unspecified set of constants:

$$\tau' \to \tau \in \mathit{TypeOf}(f) \text{ and } \vdash V : \tau' \ \Rightarrow\ \delta(f, V) \text{ is defined and } \vdash \delta(f, V) : \tau$$

The rules for the purely functional part are standard. **let**-expressions are typed through term substitution (in Section 5, we will modify this to gain better average case efficiency of the type checker). All the state transformers have type $\mathit{Cmd}\ \theta$, where θ is the type inferred from their constituent(s). For tag reader $v?$, θ is derived from v's type, and, for assignments, θ becomes the unit type (). The rules for introducing new tags (*block*) and binding state transformers (*bind*) bear structures similar to those of function introduction (*abs*) and function application (*app*).

The most complex part of the type system has to do with typing the **pure** construct. This is not surprising, since the type system has to ensure that the dynamic side conditions on the reduction of a term **pure** M are always satisfied, which requires some amount of effect analysis. We adopt the following conservative conditions: First, we require that $M \equiv P[\mathbf{return}\ N]$ to ensure that

$$(const) \qquad \Gamma \vdash c : \tau \qquad \tau \in TypeOf(c)$$

$$(var) \qquad \Gamma \vdash x : \theta \qquad x : \theta \in \Gamma$$

$$(tag) \qquad \Gamma \vdash v : Var\ \theta \qquad v : Var\ \theta \in \Gamma$$

$$(abs) \qquad \frac{\Gamma.x : \theta' \vdash M : \theta}{\Gamma \vdash x.M : \theta' \to \theta}$$

$$(app) \qquad \frac{\Gamma \vdash M : \theta' \to \theta \qquad \Gamma \vdash N : \theta'}{\Gamma \vdash M\ N : \theta}$$

$$(let) \qquad \frac{\Gamma \vdash N : \theta' \qquad \Gamma \vdash [N/y]\ M : \theta}{\Gamma \vdash \mathbf{let}\ y = N\ \mathbf{in}\ M : \theta}$$

$$(reader) \qquad \frac{\Gamma \vdash M : Var\ \theta}{\Gamma \vdash M? : Cmd\ \theta}$$

$$(assign) \qquad \frac{\Gamma \vdash M_1 : \theta \qquad \Gamma \vdash M_2 : Var\ \theta}{\Gamma \vdash M_1 =: M_2 : Cmd\ ()}$$

$$(block) \qquad \frac{\Gamma.v : Var\ \theta' \vdash M : Cmd\ \theta}{\Gamma \vdash \mathbf{var}\ v.M : Cmd\ \theta}$$

$$(bind) \qquad \frac{\Gamma \vdash x.M_2 : \theta' \to Cmd\ \theta \qquad \Gamma \vdash M_1 : Cmd\ \theta'}{\Gamma \vdash M_1 \triangleright x.M_2 : Cmd\ \theta}$$

$$(unit) \qquad \frac{\Gamma \vdash M : \theta}{\Gamma \vdash \mathbf{return}\ M : Cmd\ \theta}$$

$$(pure) \qquad \frac{\Gamma_\tau \vdash M : Cmd\ \tau}{\Gamma_\tau \vdash \mathbf{pure}\ M : \tau} \quad M \equiv P[\mathbf{return}\ N]$$

Figure 3: Typing Rules for λ_{var}-terms

something is returned from the state transformer M. Second, to ensure that the evaluation of M neither affects nor observes global storage we require that the type environment for M be applicative, *i.e.*, $\forall x \in fv(M)$, $\Gamma(x)$ is applicative. This condition is expressed by writing Γ_τ in the premise and conclusion of the rule. Third, to ensure that the state prefix can be stripped from the result, we require that the result type is applicative.

4 Type Soundness

In the section, we prove the soundess of our type system with respect to λ_{var}'s operational semantics. We show that every λ_{var}-program, if not diverging, will be evaluated to an answer value.

First, we show that all the reduction rules are type-preserving, i.e., *subject reduction* holds in our

system. The following auxiliary lemmas help to establish the subject reduction property.

Lemma 4.1 (Assumption Weaking) If $\Gamma(a) = \Gamma'(a)$ for all $a \in fv(M)$, then $\Gamma \vdash M : \theta$ iff $\Gamma' \vdash M : \theta$.

Lemma 4.2 (Term Substitution) If $\Gamma.x : \theta \vdash M : \theta'$ and $x \notin dom(\Gamma)$ and $\Gamma \vdash N : \theta$ then $\Gamma \vdash [N/x]M : \theta'$.

Lemma 4.3 (Subject Reduction) If $\Gamma \vdash M_1 : \theta$ and $M_1 \to M_2$ then $\Gamma \vdash M_2 : \theta$.

Proof: The proof proceeds by case analysis on $M_1 \to M_2$. We show some typical cases here; the remainder of them follow similarly.

Case $(x.M)N \to [N/x]M$. From $\Gamma \vdash (x.M)N : \theta$ we have $\Gamma \vdash x.M : \theta' \to \theta$ and $\Gamma \vdash N : \theta'$ by (*app*). From the former, $\Gamma.x{:}\theta' \vdash M : \theta$ follows by (*abs*). Hence $\Gamma \vdash [N/x]M : \theta$ by Lemma 4.2.

Case $N =: v \ ; \ v? \triangleright x.M \to N =: v \ ; \ (x.M)N$. From $\Gamma \vdash N =: v \ ; \ v? \triangleright x.M \ : \mathit{Cmd}\ \theta_3$ we have $\Gamma \vdash z.(v? \triangleright x.M) \ : () \to \mathit{Cmd}\ \theta_3$ for some fresh variable z and $\Gamma \vdash N =: v \ : \mathit{Cmd}\ ()$ by (*assign*) and (*bind*).

From the former, $\Gamma.z{:}() \vdash x.M : \theta_2 \to \mathit{Cmd}\ \theta_3$ and $\Gamma.z{:}() \vdash v? \ : \mathit{Cmd}\ \theta_2$ by (*abs*) and (*bind*). Hence $\Gamma.z{:}() \vdash v : \mathit{Var}\ \theta_2$ by (*reader*).

From the latter, $\Gamma \vdash N : \theta_1$ and $\Gamma \vdash v : \mathit{Var}\ \theta_1$ by (*assign*). Hence $\Gamma.z : () \vdash v : \mathit{Var}\ \theta_1$ by *Lemma* 4.1.

So

$\theta_1 = \theta_2$	
$\Gamma.z{:}() \vdash N \ : \theta_1$	*by Lemma* 4.1
$\Gamma.z{:}() \vdash (x.M)N \ : \mathit{Cmd}\ \theta_3$	by (*app*)
$\Gamma \vdash z.(x.M)N \ : () \to \mathit{Cmd}\ \theta_3$	by (*abs*)
$\Gamma \vdash N =: v \triangleright z.(x.M)N \ : \mathit{Cmd}\ \theta_3$	by (*bind*).

Case $\mathbf{pure}\ (S[\mathbf{return}\ x.M]) \to x\ .\ \mathbf{pure}\ (S[\mathbf{return}\ M])$. From $\Gamma \vdash \mathbf{pure}\ (S[\mathbf{return}\ x.M]) : \tau$, we have $\Gamma_\tau \vdash S[\mathbf{return}\ x.M] \ : \mathit{Cmd}\ \tau$ by (*pure*). It is obvious from the definition of state prefix and rules (*block*) and (*bind*) that $\tau = \tau_1 \to \tau_2$.

By a straightforward induction on pre-state prefix R's structure, we can show that

$$\text{If } \Gamma \vdash R[\mathbf{return}\ x.M] : \mathit{Cmd}\ (\tau_1 \to \tau_2) \text{ then } \Gamma.x{:}\tau_1 \vdash R[\mathbf{return}\ M] : \mathit{Cmd}\ \tau_2.$$

Hence $\Gamma.x{:}\tau_1 \vdash S[\mathbf{return}\ M] \ : \mathit{Cmd}\ \tau_2$. Now since $fv(S[\mathbf{return}\ M]) \subseteq fv(S[\mathbf{return}\ x.M]) \cup \{x\}$, we must have $\Gamma.x{:}\tau_1 \vdash \mathbf{pure}\ (S[\mathbf{return}\ M]) \ : \tau_2$ by (*pure*). Hence $\Gamma \vdash x.\mathbf{pure}(S[\mathbf{return}\ M]) : \tau$ by (*abs*).

■

Next, we examine the "positive" effects the typing rules have on the evaluation of a program. We assume that there is a nullary type constructor **Out** that represents exactly the set of all answers. Programs in λ_{var} are, therefore, closed terms of type **Out**. The goal is to show that the evaluation of a program will never get stuck without producing an answer value.

Lemma 4.4 If $\vdash M : \mathbf{Out}$ and M is irreducible (with respect to $\longmapsto$) then $M = A$ for some answer A.

This lemma is actually a corollary of the following more general one:

Lemma 4.5 Let Γ be $\{v_1 : Var\ \theta_1, \ldots, v_n : Var\ \theta_n\}$, $n \geq 0$. If $\Gamma \vdash M : \theta$ and M is irreducible then $M \in \Lambda$, the language described by the following grammar:

$$\begin{array}{lcl} \Lambda & ::= & f \mid x.M \mid c^n\ M_1\ \ldots\ M_k\ \ (0 \leq k \leq n) \mid v \mid v? \triangleright x.M \mid \Theta \\ \Theta & ::= & v? \mid M =: v \mid \mathbf{return}\ M \mid \mathbf{var}\ v.\ \Theta \mid M =: v\ ;\ \Theta \end{array}$$

Proof: By induction on the possible structure of the term M:

Case $M \equiv f$, c^n or $x.M$. $M \in \Lambda$.

Case $M \equiv M_1\ M_2$. Apply the induction hypothesis to M_1. Since M_1 is of $\rightarrow$ type, we need to consider only the following possibilities:

- $M_1 \equiv f$. Impossible. Since M_2 is of some *applicative* type, we must have $M_2 = \mathbf{V}$ for some value $\mathbf{V}$ by applying the induction hypothesis to M_2. Then M would be reducible by rule (δ) according to the $\delta - typability$ assumption.
- $M_1 \equiv (c^n\ M_1\ \ldots\ M_k)$, $0 \leq k \leq n$. Since $M \equiv M_1\ M_2$ is well-typed under Γ, we must have $0 \leq k < n$. Hence $M \in \Lambda$.
- $M_1 \equiv x.N$. Impossible, since M would be reducible by rule (β).

Case $M \equiv \mathbf{let}\ y = M_1\ \mathbf{in}\ M_2$. Impossible, since M would be reducible by rule (let).

Case $M \equiv v$. Since $\Gamma \vdash M : \theta$, we must have $v \equiv v_i$, for some i. Hence $M \in \Lambda$.

Case $M \equiv N?$. Apply the induction hypothesis to N, which is of Var type. We must have $N \equiv v_i$, for some i. Hence $M \in \Lambda$.

Case $M \equiv M_1 =: M_2$. Apply the induction hypothesis to M_2, which is of Var type. We must have $M \equiv v_i$, for some i. Hence $M \in \Lambda$.

Case $M \equiv \mathbf{return}\ N$. $M \in \Lambda$.

Case $M \equiv \mathbf{var}\ v.N$. From $\Gamma \vdash M : \theta$, we have $\Gamma.v : Var\ \theta' \vdash N : \theta$. Now apply the induction hypothesis to N, which is of Cmd type. Either $N \in \Theta$ and hence $M \in \Lambda$, or $N \equiv u? \triangleright x.M'$ rendering M reducible by rule (b_2) or $(\perp)$.

Case $M \equiv M_1 \triangleright x.M_2$. Apply the induction hypothesis to M_1, which is of Cmd type. We need to consider the following possibilities:

- $M_1 \equiv v? \triangleright y.N$. Impossible, since M would be reducible by rule $(\triangleright\triangleright)$.
- $M_1 \equiv \mathbf{return}\ N$. Impossible, since M would be reducible by rule $(r\triangleright)$.
- $M_1 \equiv \mathbf{var}\ v.N$. Impossible, since M would be reducible by rule $(v\triangleright)$.
- $M_1 \equiv v?$. $M \in \Lambda$.

- $M_1 \equiv N =: v$. Since M is irreducible, we must have $x \notin fv(M_2)$, for otherwise rule $=:\triangleright$ would have been applicable before M_1 became $N =: v$. Moreover, from $\Gamma \vdash M : \theta$, we have $\Gamma \vdash x.M_2 : () \to \theta$ by (*bind*). Hence $\Gamma \vdash M_2 : \theta$ by Lemma 4.1. Now apply the induction hypothesis to M_2, which is of *Cmd* type. Either $M_2 \in \Theta$ and hence $M \in \Lambda$, or $M_2 \equiv u?\triangleright y.M_3$ rendering M reducible by rule (f) or (b_1).

Case $M \equiv$ **pure** P[**return** N] . Impossible, since M would not be irreducible as we will argue. Suppose otherwise, then P[**return** N] must be irreducible, too. Moreover, from $\Gamma \vdash$ **pure** P[**return** N] : τ, we have $\vdash P$[**return** N] : *Cmd* τ by (*pure*) and Lemma 4.1. Apply the induction hypothesis to P[**return** N], which is a closed term of *Cmd* type. We must have $P \equiv S$ for some state prefix S. Hence N must be irreducible, and $\Gamma' \vdash N : \tau$ for some $\Gamma' = \{u_1 : Var\ \nu_1, \ldots, u_k : Var\ \nu_k\}$, $u_i \in bv(S)$. Now apply the induction hypothesis to N, which is of some *applicative* type. We must have $N \equiv \mathbf{V}$ for some value $\mathbf{V}$, rendering M reducible by p_c, p_λ, or p_f.

■

With Lemma 4.3 and Lemma 4.4, the soundness theorem follows:

Theorem 4.6 (Type Soundness) If $\vdash M :$ **Out** then either $M \Uparrow$ or $eval_{var}(M) = A$ for some answer A.

5 Type Checking

In this section, we sketch an implementation scheme of our type system. Basically, a syntactic-directed type checker along the lines of [DM82] can be employed (see appendix). The major concern here is how to type **let**-expressions without explicit term substitution.

As usual, we introduce type schemes to avoid such term substitution. In doing so, the major change is to modify the rule (*pure*) for typing **pure** terms. We first extend our type system with type schemes in the standard manner:

$$\text{Type Schemes} \quad \sigma ::= \theta \mid \forall \underline{\alpha}.\sigma \mid \forall \alpha.\sigma$$

In addition to the standard rules for generalizing type variables, we need the following two rules to instantiate type variables.

$$(inst_p) \quad \frac{\Gamma \vdash M : \forall \alpha.\sigma}{\Gamma \vdash M : [\theta/\alpha]\,\sigma} \qquad (inst'_p) \quad \frac{\Gamma \vdash M : \forall \underline{\alpha}.\sigma}{\Gamma \vdash M : [\tau/\underline{\alpha}]\,\sigma}$$

All typing rules other than (*let*) and (*pure*) remain unchanged. Rule (*let*) is replaced by rule (let_p).

$$(let_p) \quad \frac{\Gamma \vdash N : \sigma \qquad \Gamma.y{:}\sigma \vdash e : \tau}{\Gamma \vdash \textbf{let } y = N \textbf{ in } M : \tau}$$

The new, polymorphic formulation of (let_p) requires some changes in rule (*pure*). Recall that, in typing **pure** M, we demand the type environment for M be applicative. In other words, only free

variables of applicative type are allowed in M. Now since we will extend the type environment with the polymorphic typescheme assumption $(y : \sigma)$ in (and only in) typing **let**-expressions, it seems that we need to extend our stratification of types to type schemes. However, it is conceivable that any such extension would make rule (*pure*) overly restrictive by rejecting many harmless procedures introduced by **let**. For instance:

$$\begin{array}{l}\textbf{let } swap = x.y.\ x?\ \triangleright z.\ y?\ \triangleright\ w.\ z =: y\ ;\ w =: x \\ \textbf{in pure } (\ \textbf{var } u.\ \textbf{var } v.\ \ 0 =: u\ ;\ 5 =: v\ ;\ \ swap\ u\ v\ ;\ u?\ \triangleright x.\ \textbf{return } x\ )\end{array}$$

would be rejected because of the reference to the non-local *swap*, whose most general type is $\forall\alpha.\, Var\ \alpha \rightarrow Var\ \alpha \rightarrow Cmd\ ()$, in the body of the **pure**. However, this term is perfectly safe, since *swap* affects only its arguments x, y. If we use the old rule, (*let*), the problem does not arise since then the body of *swap* is expanded *within* the scope of the **pure**. To correct this deficiency, we keep track of all possible non-applicative types as if all **let**'s in a term had been expanded.

Definition. Let $LB(M)$ be the set of **let**-bound identifiers in a term M and for any **let**-bound y let $defn(y)$ be the defining term of y. Then the *closure set* of free variables and tags of a term M is given by:

$$fv^*(M) \stackrel{def}{=} fv(M) \cup \bigcup_i \{fv^*(defn(y_i)) \mid y_i \in LB(M)\}.$$

Furthermore, $fv^+(M)$, the *λ-var-closure set*, denotes the subset of all λ and **var** bound identifiers in $fv^*(M)$.

In other words, the λ-var-closure set of a term is the union of its free variables and tags with all the variables and tags that will be introduced through the reduction of **let**-expressions at outer levels. Since all these variables and tags are λ or **var** bound and hence are monomorphically typed in the type environment, we can adapt rule (*pure*) as follows:

$$(pure_p)\quad \frac{\Gamma \vdash M :\ Cmd\ \tau}{\Gamma \vdash \textbf{pure } M\ :\tau}\quad M \equiv P[\textbf{return } N] \text{ and } \textbf{Safe}\ \Gamma|_{fv^+(M)}$$

where $\Gamma\,|_V$ is the restriction of Γ to the variables in V, and $\textbf{Safe}\,\Gamma'$ iff $\forall x \in dom(\Gamma')$, $\Gamma'(x)$ is applicative.

According to the modified side condition, for a **pure** term to be well-typed it is necessary to perform the applicative type check over its present free variables and tags as well as the ones that will be introduced in reducing outer **let**-expressions. Hence the preceding example program is well-typed in the new system with rules (let_p) and $(pure_p)$, since neither *swap* nor the **pure**-body contains any unsafe free variables and tags.

Clearly, this modified side condition is derived from the substitution semantics of **let**-expressions and corresponds to the one we have for rule (*pure*). Furthermore, such λ-var-closure sets of variables and tags can be easily collected as we traverse and type check the source term. One possible scheme is to have, in type checking, an extra variables/tags environment, which maps each **let**-bound identifier to the λ-var-closure set associated with its defining term, yet map λ and **var** bound identifiers to a singleton set consisting of themselves. Thus, as we traverse the source term to any of its **pure** subterms, we can easily obtain its λ-var-closure set to perform the required safe type check.

Figure 4 contains a type checker for λ_{var}. The algorithm is called W_v because of its structural similarity to the algorithm W [DM82]. However, there are two major differences between W_v and W: First, W_v takes an extra variables/tags environment $\mathcal{V}$ to facilitate computing the λ-var-closure sets for **pure** subterms as we have just described. Second, since the type system has been stratified into two layers, we need to use a different unification algorithm. Indeed, the stratified type system corresponds naturally to an order-sorted signature of two sorts, *app* and *cmd*, with $app \leq cmd$. Hence any order-sorted unification algorithm will serve our purpose.

6 Conclusion

We have presented a Hindley/Milner-style type system for λ_{var}, an imperative extension of the λ-calculus. In our type system the distinction between state-dependent and state-independent terms are manifest in their types, clearly reflecting their intended meanings in λ_{var}. The conversion between state transformers and pure expressions is moderated by the type system. We have proved the soundness of our type system with respect to λ_{var}'s operational semantics. Hence any invalid coercion from a state-dependent term to a pure value will be a static error along with other common type errors. In addition, we have presented a type checker based on a straightforward extension of the W algorithm.

An interesting and usefule sequel to the current system is to improve its precision in typing **pure**-terms. For example, the following term is ill-typed because of the reference of the non-local tag reader, *rdr*, in the body of **pure**, though no run-time error would arise in its evaluation.

$$\begin{array}{l}\textbf{return}\ (x.x?) \triangleright rdr.\\ \quad \textbf{let}\ res = \textbf{pure}\ (\textbf{var}\ v.5 =: v\ ;\ rdr\ v \triangleright z.\textbf{return}\ z)\\ \quad\ \textbf{in return}\ res\end{array}$$

Apparently, as long as no global storage is accessed in evaluating the argument term to **pure**, the coercion should be safe. Therefore, to better type **pure**-terms, we plan to investigate the feasibility of incorporating a *region analysis* similar to that of [TJ94] to our system.

Acknowledgements

The use of a stratified type system of applicative types and command types was suggested by Uday Reddy. Thanks also to Vincent Dornic for his comments on an earlier draft of this paper.

References

[AM87] Andrew W. Appel and David B. MacQueen. A standard ML compiler. In *Functional Programming and Computer Architecture*, pages 301–324, August 1987. LNCS 274.

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[Dam85] Luis Damas. *Type assignment in programming languages.* PhD thesis, Department of Computer Science, Edinburgh University, April 1985.

[DM82] L. Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[LW91] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Proc. of the 18th ACM Symp. on Principles of Programming Languages*, pages 291–302, January 1991.

[Mai93] Harry G. Mairson. Quantifier elimination and parametric polymorphism in programming languages. *To appear in the J. of Functional Programming*, 1993.

[ORH93] Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignment, and the lambda calculus. In *Proc. of 20th ACM Symp. on Principles of Programming Languages*, January 1993.

[Rey88] John C. Reynolds. Preliminary design of the programming language forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[SRI92] V. Swarup, U. Reddy, and E. Ireland. Assignments for applicative languages. In J. Hughes, editor, *Proc. 5th ACM Conf. on Functional Programming and Computer Architecture*, pages 192–214, August 1992. LNCS 582.

[TJ93] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *To appear in the J. of Functional Programming*, 1993.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *To appear in the J. of Information and Computation*, 1994.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, November 1990.

[Wad90] Philip Wadler. Comprehending monads. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 61–78, June 1990.

[WF92] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Department of Computer Science, Rice University, March 1992.

[WF93] Stephen Weeks and Matthias Felleisen. On the orthogonality of assignments and procedures in Algol. In *Proc. of 20th ACM Symp. on Principles of Programming Languages*, January 1993.

[Wri92] Andrew K. Wright. Typing references by effect inferences. In B. Krieg-Brückner, editor, *Prod. European Symposium on Programming*, pages 473–491, February 1992. LNCS 582.

$W_v\ (M, \Gamma, \mathcal{V})\ =$ **case** M **of**

$x\ \Rightarrow\ inst\ (S(\Gamma x), id)$

$x.N\ \Rightarrow$
  let α a fresh type variable
    $(\theta_1, S_1)\ =\ W_v\ (N, \Gamma.x{:}\alpha, \mathcal{V}.x{:}\{x\})$
  in $(S_1\alpha \rightarrow \theta_1, S_1)$

$M_1\ M_2\ \Rightarrow$
  let $(\theta_1, S_1)\ =\ W_v\ (M_1, \Gamma, \mathcal{V})$
    $(\theta_2, S_2)\ =\ W_v\ (M_2, S_1\Gamma, \mathcal{V})$
    α a fresh type variable
    $S_3\ =\ mgu\ S_2\theta_1\ (\theta_2 \rightarrow \alpha)$
  in $(S_3\alpha, S_3S_2S_1)$

let $y = M_1$ **in** $M_2\ \Rightarrow$
  let $(\theta_1, S_1)\ =\ W_v\ (M_1, \Gamma, \mathcal{V})$
    $\sigma\ =\ Clos(\theta_1, S_1\Gamma)$
  in $W_v\ (M_2,\ S_1\Gamma.x{:}\sigma,\ \mathcal{V}.y{:}fvc(fv\ M_2, \mathcal{V}))$

pure $M_1\ \Rightarrow$
  let $M_1 \equiv P[\mathbf{return}\ N]$
    $(\theta_1, S_1)\ =\ W_v\ (P[\mathbf{return}\ N)], \Gamma, \mathcal{V})$
    $\underline{\alpha}$ a fresh applicative type variable
    $S_2\ =\ mgu\ \theta_1\ (Cmd\ \underline{\alpha})$
    $(y_1, \ldots, y_n)\ =\ \bigcup_i \{\mathcal{V}x_i \mid x_i \in fv\ M_1\}$
  in *if* $\bigwedge_i\ ((S_2S_1\Gamma)y_i$ be *applicative*)
    then $(S_2\alpha, S_2S_1)$
    else **fail**

var $v.N\ \Rightarrow$
  let α, β fresh type variables
    $(\theta_1, S_1)\ =\ W_v\ (N, \Gamma.v{:}\ Var\ \alpha, \mathcal{V}.v{:}\{v\})$
    $S_2\ =\ mgu\ \theta_1\ (Cmd\ \beta)$
  in $(Cmd\ S_2\beta, S_2S_1)$

$M_1 \triangleright x.M_2\ \Rightarrow$
  let α, β fresh type variables
    $(\theta_1, S_1)\ =\ W_v\ (M_1, \Gamma, \mathcal{V})$
    $(\theta_2, S_2)\ =\ W_v\ (x.M_2, S_1\Gamma, \mathcal{V})$
    $S_3\ =\ mgu\ S_2\theta_1\ (Cmd\ \alpha)$
    $S_4\ =\ mgu\ S_3\theta_2\ (S_3\alpha \rightarrow Cmd\ \beta)$
  in $(Cmd\ S_4\beta, S_4S_3S_2S_1)$

$N?\ \Rightarrow$
  let $(\theta_1, S_1)\ =\ W_v\ (N, \Gamma, \mathcal{V})$
    α a fresh type variable
    $S_2\ =\ mgu\ S_1\theta_1\ (Var\ \alpha)$
  in $(Cmd\ S_2\alpha, S_2S_1)$

$M_1 =: M_2\ \Rightarrow$
  let $(\theta_1, S_1)\ =\ W_v\ (M_1, \Gamma, \mathcal{V})$
    $(\theta_2, S_2)\ =\ W_v\ (M_2, S_1\Gamma, \mathcal{V})$
    $S_3\ =\ mgu\ \theta_2\ (Var\ S_2\theta_1)$
  in $(Cmd\ (), S_3S_2S_1)$

return $N\ \Rightarrow$
  let $(\theta_1, S_1)\ =\ W_v\ (N, \Gamma, \mathcal{V})$
  in $(Cmd\ \theta_1, S_1)$

where $inst\ (\forall\underline{\alpha}::\Gamma.\sigma, S)\ =$ *let* $\underline{\beta}$ a fresh applicative type variable
  in $inst\ ([\underline{\beta}/\underline{\alpha}]\ \sigma, S)$

$inst\ (\forall\alpha::\Gamma.\sigma, S)\ =$ *let* β a fresh type variable
  in $inst\ ([\beta/\alpha]\ \sigma, S)$

$inst\ (\theta, S)\ = (\theta, S)$

$fvc\ (\{\}, \mathcal{V})\ = \{\}$

$fvc\ (\{x\} \cup L,\ \mathcal{V})\ = \mathcal{V}x\ \cup\ fvc\ (L, \mathcal{V})$

Figure 4: A type checker for λ_{var}