Polymorphic Typing of Variables and References

GEOFFREY SMITH
Florida International University
and
DENNIS VOLPANO
Naval Postgraduate School

In this article we consider the polymorphic type checking of an imperative language. Our language contains *variables*, first-class *references* (pointers), and first-class functions. Variables, as in traditional imperative languages, are implicitly dereferenced, and their addresses (L-values) are not first-class values. Variables are easier to type check than references and, in many cases, lead to more general polymorphic types. We present a polymorphic type system for our language and prove that it is sound. Programs that use variables sometimes require weak types, as in Tofte's type system for Standard ML, but such weak types arise far less frequently with variables than with references.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Assignment, references, variables

1. INTRODUCTION

Polymorphic type checking of a language with first-class *references* (pointers) is a difficult problem, as can be seen by the many type systems proposed for typing references in Standard ML [Damas 1985; Greiner 1993; Hoang et al. 1993; Leroy 1993; Leroy and Weis 1991; Talpin and Jouvelot 1992; Tofte 1990; Wright 1995]. But many imperative programs do not require the power of first-class references—they merely manipulate values, other than pointers, as the contents of local variables. Unfortunately, if local variables must be created using first-class references, then whatever mechanism is used to enforce the correct typing of references is likely to

This material is based upon activities supported by the National Science Foundation under Agreements No. CCR-9414421 and CCR-9400592. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Authors' addresses: G. Smith, School of Computer Science, Florida International University, Miami, FL 33199; email: smithg@cs.fiu.edu; D. Volpano, Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943; email: volpano@cs.nps.navy.mil.

adversely affect the typing of programs that really only need variables. Thus, it is beneficial to introduce an additional `letvar` construct to allocate *variables*, which are implicitly dereferenced and whose addresses (L-values) are not first-class values.

Aside from their typing benefits, variables are also of interest because their implicit dereferencing is a syntactic convenience, and because they are at the core of mainstream imperative languages.

The idea of including variables in a polymorphic language is not new. In fact, Edinburgh LCF ML [Gordon et al. 1979] had a `letvar` construct, which it called `letref`. But it did not have first-class references, and according to Tofte [1990], its type system was never proved sound.

2. AN INFORMAL DESCRIPTION OF THE TYPE SYSTEM

The language we consider is the core ML of Damas and Milner [1982] together with first-class references, created by `ref`, variables, created by `letvar`, and imperative constructs such as `while` loops. The construct `letvar x := a in b` binds `x` to a new cell initialized to the value of `a`. The scope of the binding is `b`, and the lifetime of the cell is unbounded. Conversion of L-values to R-values is implicit, so that `letvar x := e in x` is equivalent to `e`.

The types of our system are stratified into three levels. There are the ordinary τ (data types) and σ (type schemes) type levels of Damas and Milner's system and a new level called *phrase types* containing σ types and types of the form τ *var* for variables. Unlike references, variables are not first-class values. As in Tofte's system for Standard ML [Tofte 1990], type variables are partitioned into *weak* and *strong* variables.[1] Strong type variables are written α and weak ones $\_\alpha$. A weak type variable cannot be instantiated with a type containing strong type variables. As in Tofte's system, a weak type variable can be generalized only when it appears in the type of a *syntactic value*, that is, an identifier, a literal, or a λ-abstraction.

Because variable addresses are not first-class values, it is easier to keep track syntactically of operations on variables than operations on references. As a result, many useful functions that use `letvar` can be given fully polymorphic types. For example, imperative list reversal can be defined as

```
fun irev l = letvar a := l in
  letvar b := [] in
    while not (null a) do
      ( b := (hd a) :: b;
        a := tl a);
    b
  end end
```

Two local variables `a` and `b` are declared, yet the function is assigned fully polymorphic type $\forall\alpha\,.\,\alpha\ list \rightarrow \alpha\ list$ in our system. Thus `irev[]` is a polymorphic list of type $\forall\alpha\,.\,\alpha\ list$. Consider a definition of `irev` in Standard ML:

```
fun irev l = let val a = ref l in
  let val b = ref [] in
```

[1] Tofte actually calls them *imperative* and *applicative* variables, respectively.

```
      while not (null (!a)) do
        ( b := (hd (!a)) :: (!b);
          a := tl (!a));
      !b
    end end
```

Now the use of local variables is reflected in the type of `irev`. Standard ML would give it type $\forall \_\alpha\,.\,\_\alpha\ \mathit{list} \rightarrow \_\alpha\ \mathit{list}$, where $\_\alpha$ is an imperative type variable, and Standard ML of New Jersey would give it type $\forall \alpha^1\,.\,\alpha^1\ \mathit{list} \rightarrow \alpha^1\ \mathit{list}$ where α^1 is a weak type variable. The weak variable indicates that applying `irev` once may create a reference whose type involves α. In each case, consequently, `irev[]` is not a polymorphic list.

One has the option of defining `irev` in our language using `ref` instead of `letvar`, but this would needlessly constrain polymorphism. Our system would then give it the Standard ML type $\forall \_\alpha\,.\,\_\alpha\ \mathit{list} \rightarrow \_\alpha\ \mathit{list}$, and the application `irev[]` would no longer be polymorphic. In fact, if one always uses `let` and `ref` in our system rather than `letvar`, then our system "degenerates" to Tofte's system for Standard ML.

Our system also does well on programs that cause problems for the "syntactic values" type system advocated by Wright [1995]. Consider `makeCountFun` which takes a function f as input and returns both a counting version of f and a function to read the counter:

```
fun makeCountFun f = letvar x := 0 in
    (fn z => x := x + 1; f z,
     fn () => x)
end
```

Our system gives `makeCountFun` type

$$\forall \alpha, \beta\,.\,(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta) \times (\mathit{unit} \rightarrow \mathit{int}),$$

and an application such as `makeCountFun hd` is polymorphic. If `makeCountFun` is written using `let` and `ref`, then `makeCountFun hd` is also polymorphic in Tofte's system. But in Wright's system, `makeCountFun hd` is monomorphic because only syntactic values are polymorphic, and a function application is not a syntactic value.

Programs that use `letvar` but not `ref` may still require weak types. The rule is that a `letvar`-bound identifier must be given a weak type if it occurs in a λ-abstraction within its scope. This rule comes into play when functions create "objects" or "own variables." For example, consider a function that creates a stack object with push and pop operations accessing a shared stack:

```
fun makestack x =
    letvar stk := x in
      (fn v => stk := v :: stk,
       fn () => stk := tl stk)
    end
```

It is unsound to give `makestack` the strong polymorphic type

$$\forall \alpha\,.\,\alpha\ \mathit{list} \rightarrow (\alpha \rightarrow \mathit{unit}) \times (\mathit{unit} \rightarrow \mathit{unit})$$

because, if the application `makestack []` were polymorphic, the resulting push operation could be called with values of different types, leading to a nonhomogeneous stack. In our system, since `stk` occurs inside a λ-abstraction within its scope, it must be given a weak type. This allows `makestack` to be given only the weak polymorphic type

$$\forall \_\alpha \,.\, \_\alpha\ \mathit{list} \rightarrow (\_\alpha \rightarrow \mathit{unit}) \times (\mathit{unit} \rightarrow \mathit{unit}).$$

The ability to give `makestack` a weak polymorphic type makes our system substantially better than Edinburgh LCF ML on cases of this kind. In LCF ML, a `letvar`-bound identifier must be given a *monotype* (i.e., a type with no variables) if it is assigned to within a λ-abstraction within its scope (restriction 2*ib* [Gordon et al. 1979, p. 49]). Hence, since `stk` is assigned to within the push and pop operations, LCF ML requires `stk` to be annotated with a monotype. This forces `makestack` to be monomorphic.

Finally, typings of purely functional programs in our system are preserved as they are in the type systems for Standard ML and Standard ML of New Jersey. No labels or other annotations are required on arrow types as they are in closure [Leroy and Weis 1991] and effect [Talpin and Jouvelot 1992; Wright 1992] typing.

3. A FORMAL TREATMENT OF THE TYPE SYSTEM

The syntax of our language is given below. Following Tofte [1990], we distinguish a subset of the expressions called *Values*. Evaluating a value does not allocate any new cells; this property is exploited by the type system.

$$\begin{array}{lll}
(\mathit{Expressions}) & e ::= & v \mid l \mid e_1\, e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \\
 & & \mathbf{letvar}\ x := e_1\ \mathbf{in}\ e_2 \mid e_1 := e_2 \mid \\
 & & \mathbf{ref}\, e \mid *e \mid \\
 & & e_1; e_2 \mid \mathbf{while}\ e_1\ \mathbf{do}\ e_2 \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \\
(\mathit{Values}) & v ::= & x \mid c \mid r \mid \lambda x.\, e
\end{array}$$

Metavariable x ranges over identifiers, and metavariable c ranges over literals, such as **true**, **false**, and **unit**. Metavariables l and r range over *variable locations* and *reference locations*, respectively.[2] Notice that unlike reference locations, variable locations are not values. The $*$ operator is used to dereference a reference; it is similar to ! in Standard ML. Finally, we remark that the sequential composition $e_1; e_2$ could be taken as syntactic sugar for **let** $z = e_1$ **in** e_2, where z is new.

The types of the language are stratified as follows.

$$\begin{array}{lr}
\tau ::= \alpha \mid \mathit{bool} \mid \mathit{unit} \mid \tau\ \mathit{ref} \mid \tau \rightarrow \tau' & (\mathit{data\ types}) \\
\sigma ::= \forall \alpha \,.\, \sigma \mid \tau & (\mathit{type\ schemes}) \\
\rho ::= \sigma \mid \tau\ \mathit{var} & (\mathit{phrase\ types})
\end{array}$$

Metavariable α ranges over *type variables*. Type variables are partitioned into *weak* and *strong* type variables, written $\_\alpha$ and α respectively. These variables correspond to the imperative and applicative type variables respectively of Tofte's system. We

[2]Locations will not in fact occur in user programs. They are included as expressions solely for the purpose of simplifying the semantics, as will become clear in Section 4.

say a data type τ is weak iff every type variable occurring in it is weak. Type τ *ref* (τ *var*) is the type of *reference* (*variable*) locations storing values of type τ.

The rules of the type system are formulated as they are in Harper's system [Harper 1994] and are given in Figure 1. It is a deductive proof system used to assign types to expressions. Typing judgments have the form

$$\lambda; \gamma \vdash e : \rho$$

meaning that expression e has type ρ assuming that the free identifiers and locations of e have the types prescribed by γ and λ, respectively. More precisely, metavariable γ ranges over *identifier typings*, which are finite functions mapping identifiers to phrase types; $\gamma(x)$ is the phrase type assigned to x by γ, and $\gamma[x : \rho]$ is a modified identifier typing that assigns phrase type ρ to x and assigns phrase type $\gamma(x')$ to any identifier x' other than x. Metavariable λ ranges over *location typings*, which are finite functions mapping locations to data types. The notational conventions for location typings are similar to those for identifier typings.

The *generalization* of a data type τ relative to λ and γ, written $\mathit{Close}_{\lambda;\gamma}(\tau)$, is the type scheme $\forall\bar{\alpha} \,.\, \tau$, where $\bar{\alpha}$ is the set of all type variables occurring free in τ but not in λ or in γ. We write $\lambda \vdash e : \tau$ and $\mathit{Close}_{\lambda}(\tau)$ when $\gamma = \emptyset$. A restricted form of generalization, written $\mathit{AppClose}_{\lambda;\gamma}(\tau)$, is defined to be the same as $\mathit{Close}_{\lambda;\gamma}(\tau)$ except that only strong type variables are generalized; any weak ones remain free.

A *substitution* is a mapping S from type variables to data types such that if $\_\alpha$ is in the domain of S, then $S(\_\alpha)$ is weak. Substitutions extend homomorphically to data types.

We say that τ' is a *generic instance* of $\forall\bar{\alpha} \,.\, \tau$, written $\forall\bar{\alpha} \,.\, \tau \geq \tau'$, if there exists a substitution S with domain $\bar{\alpha}$ such that $S\,\tau = \tau'$. We extend this definition to type schemes by saying that $\sigma \geq \sigma'$ if for all τ, $\sigma' \geq \tau$ implies $\sigma \geq \tau$.

Finally, we write $\lambda; \gamma \vdash e : \sigma$ iff $\lambda; \gamma \vdash e : \tau$ whenever $\sigma \geq \tau$.

Rules (L-VAL) and (ASSIGN) should be contrasted with the analogous rules in Standard ML. In our system, if $e : \tau$ *ref*, then $*e : \tau$ *var*; in Standard ML, $!e : \tau$. In our system, the left-hand side of an assignment must have a type of the form τ *var*; in Standard ML, it must have a type of the form τ *ref*. Hence if x : *int ref*, then one increments the cell that x points to by writing

$$*x := *x + 1$$

in our system and

$$x := !x + 1$$

in Standard ML.

We do not adopt Standard ML's typings of references because this would lead to ambiguity. For suppose that we had *two* rules for typing assignments: rule (ASSIGN) and Standard ML's rule,

$$\frac{\lambda; \gamma \vdash e_1 : \tau\ \mathit{ref}, \quad \lambda; \gamma \vdash e_2 : \tau}{\lambda; \gamma \vdash e_1 := e_2 : \mathit{unit}}.$$

Then the expression

$$\textbf{letvar}\ p := \textbf{ref}\ 0\ \textbf{in}\ \lambda x.\, p\ :=\ x$$

(IDENT) $\lambda;\gamma \vdash x : \tau \qquad \gamma(x) \geq \tau$

(VAR-ID) $\lambda;\gamma \vdash x : \tau\ var \qquad \gamma(x) = \tau\ var$

(REFLOC) $\lambda;\gamma \vdash r : \tau\ ref \qquad \lambda(r) = \tau$

(VARLOC) $\lambda;\gamma \vdash l : \tau\ var \qquad \lambda(l) = \tau$

(LIT) $\lambda;\gamma \vdash \mathbf{true} : bool$

$\lambda;\gamma \vdash \mathbf{false} : bool$

$\lambda;\gamma \vdash \mathbf{unit} : unit$

($\rightarrow$-INTRO) $\dfrac{\lambda;\gamma[x : \tau_1] \vdash e : \tau_2}{\lambda;\gamma \vdash \lambda x.\, e : \tau_1 \rightarrow \tau_2}$

($\rightarrow$-ELIM) $\dfrac{\lambda;\gamma \vdash e_1 : \tau_1 \rightarrow \tau_2,\ \ \lambda;\gamma \vdash e_2 : \tau_1}{\lambda;\gamma \vdash e_1\, e_2 : \tau_2}$

(LET-VAL) $\dfrac{\lambda;\gamma \vdash v : \tau_1,\ \ \lambda;\gamma[x : Close_{\lambda;\gamma}(\tau_1)] \vdash e : \tau_2}{\lambda;\gamma \vdash \mathbf{let}\ x = v\ \mathbf{in}\ e : \tau_2}$

(LET-ORD) $\dfrac{\lambda;\gamma \vdash e_1 : \tau_1,\ \ \lambda;\gamma[x : AppClose_{\lambda;\gamma}(\tau_1)] \vdash e_2 : \tau_2}{\lambda;\gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}$

(LETVAR) $\dfrac{\lambda;\gamma \vdash e_1 : \tau_1,\ \ \lambda;\gamma[x : \tau_1\ var] \vdash e_2 : \tau_2 \quad \text{If } x \text{ occurs in a } \lambda\text{-abstraction in } e_2, \text{ then } \tau_1 \text{ is weak.}}{\lambda;\gamma \vdash \mathbf{letvar}\ x := e_1\ \mathbf{in}\ e_2 : \tau_2}$

(R-VAL) $\dfrac{\lambda;\gamma \vdash e : \tau\ var}{\lambda;\gamma \vdash e : \tau}$

(ASSIGN) $\dfrac{\lambda;\gamma \vdash e_1 : \tau\ var,\ \ \lambda;\gamma \vdash e_2 : \tau}{\lambda;\gamma \vdash e_1 := e_2 : unit}$

(REF) $\dfrac{\lambda;\gamma \vdash e : \tau,\ \ \tau \text{ is weak}}{\lambda;\gamma \vdash \mathbf{ref}\, e : \tau\ ref}$

(L-VAL) $\dfrac{\lambda;\gamma \vdash e : \tau\ ref}{\lambda;\gamma \vdash *e : \tau\ var}$

(COMPOSE) $\dfrac{\lambda;\gamma \vdash e_1 : \tau_1,\ \ \lambda;\gamma \vdash e_2 : \tau_2}{\lambda;\gamma \vdash e_1; e_2 : \tau_2}$

(WHILE) $\dfrac{\lambda;\gamma \vdash e_1 : bool,\ \ \lambda;\gamma \vdash e_2 : \tau}{\lambda;\gamma \vdash \mathbf{while}\ e_1\ \mathbf{do}\ e_2 : unit}$

(IF) $\dfrac{\lambda;\gamma \vdash e_1 : bool,\ \ \lambda;\gamma \vdash e_2 : \tau,\ \ \lambda;\gamma \vdash e_3 : \tau}{\lambda;\gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}$

Fig. 1. Rules of the type system.

in which p has type *int ref var* would be ambiguous. If x had type *int ref*, then the assignment would make p point to a new cell. On the other hand, if x had type *int*, then an R-value conversion of p could give it type *int ref*, and the assignment would change the contents of the cell to which p points. But with our rules there is no ambiguity. Just as in C, we write $p := x$ to make p point to a new cell and $*p := x$ to change the contents of the cell to which p points.

Note also how the type stratification and typing rules force variables to be implicitly dereferenced, except when they occur as the left-hand side of an assignment. Consider, for example, the typing of **letvar** $x := e_1$ **in** e_2. Rule (LETVAR) forces e_2 to be given a *data type* τ_2, not a *phrase type.* So if e_2 is x (with type, say, τ_1 *var*), then we are forced to use rule (R-VAL) to derive the typing $x : \tau_1$ before we can type the entire **letvar**. Indeed, one can readily see that the only expressions that can get types of the form τ *var* are identifiers, variable locations, and expressions of the form $*e$.

4. SEMANTICS AND SOUNDNESS

In this section, we establish the soundness of our type system using the framework of Harper [1994], who built upon the earlier work of Tofte [1990], Wright and Felleisen [1994], and Leroy and Weis [1991].

First we give a structured operational semantics for our language. An expression is evaluated relative to a *memory* μ, which is a finite function from locations to values. The contents of a location $l \in dom(\mu)$ is the value $\mu(l)$, and we write $\mu[l := v]$ for the memory that assigns value v to location l, and value $\mu(l')$ to a location $l' \neq l$. Note that $\mu[l := v]$ is an *update* of μ if $l \in dom(\mu)$ and an *extension* of μ if $l \notin dom(\mu)$.

Our evaluation rules are given in Figure 2. They allow us to derive judgments of the form

$$\mu \vdash e \Rightarrow v, \mu'$$

which is intended to assert that evaluating closed expression e in memory μ results in value v and new memory μ'. We write $[e'/x]e$ to denote the capture-avoiding substitution of e' for all free occurrences of x in e. The use of substitutions in rules (APPLY), (BIND), and (BINDVAR) allows us to avoid environments and closures in the semantics, so that the result of evaluating an expression is just another expression.

We now turn to soundness. The basic idea is to show that if $\vdash e : \tau$ and $\vdash e \Rightarrow v, \mu'$, then $\vdash v : \tau$, a property called *subject reduction.* But since e can allocate locations and since these locations can occur in v, the conclusion must actually be that there exists a location typing λ' such that $\lambda' \vdash v : \tau$ and such that $\mu' : \lambda'$. The latter condition asserts that λ' is consistent with μ'; more precisely, we say that $\mu : \lambda$ if $dom(\mu) = dom(\lambda)$ and for every $l \in dom(\mu)$, $\lambda \vdash \mu(l) : \lambda(l)$.

It is the location typing λ' that makes soundness delicate. As observed by Tofte, we may generalize a type variable α in typing $\vdash e : \tau$, only to find that α occurs (free) in λ', and therefore cannot be generalized in typing $\lambda' \vdash v : \tau$.

(VAL) $$\mu \vdash v \Rightarrow v, \mu$$

(APPLY) $$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow \lambda x.\, {e_1}', \mu_1 \\ \mu_1 \vdash e_2 \Rightarrow v_2, \mu_2 \\ \mu_2 \vdash [v_2/x]{e_1}' \Rightarrow v, \mu'\end{array}}{\mu \vdash e_1\ e_2 \Rightarrow v, \mu'}$$

(BIND) $$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow v_1, \mu_1 \\ \mu_1 \vdash [v_1/x]e_2 \Rightarrow v_2, \mu_2\end{array}}{\mu \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow v_2, \mu_2}$$

(BINDVAR) $$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow v_1, \mu_1 \\ l \notin \mathit{dom}(\mu_1) \\ \mu_1[l := v_1] \vdash [l/x]e_2 \Rightarrow v_2, \mu_2\end{array}}{\mu \vdash \textbf{letvar}\ x := e_1\ \textbf{in}\ e_2 \Rightarrow v_2, \mu_2}$$

(CONTENTS) $$\mu \vdash l \Rightarrow \mu(l), \mu$$

(UPDATE) $$\frac{\mu \vdash e \Rightarrow v, \mu'}{\mu \vdash l := e \Rightarrow \textbf{unit}, \mu'[l := v]}$$

$$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow r, \mu_1 \\ \mu_1 \vdash e_2 \Rightarrow v, \mu_2\end{array}}{\mu \vdash *e_1 := e_2 \Rightarrow \textbf{unit}, \mu_2[r := v]}$$

(ALLOC) $$\frac{\begin{array}{l}\mu \vdash e \Rightarrow v, \mu' \\ r \notin \mathit{dom}(\mu')\end{array}}{\mu \vdash \textbf{ref}\, e \Rightarrow r, \mu'[r := v]}$$

(DEREF) $$\frac{\mu \vdash e \Rightarrow r, \mu'}{\mu \vdash *e \Rightarrow \mu'(r), \mu'}$$

(SEQ) $$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow v_1, \mu_1 \\ \mu_1 \vdash e_2 \Rightarrow v_2, \mu_2\end{array}}{\mu \vdash e_1; e_2 \Rightarrow v_2, \mu_2}$$

(LOOP) $$\frac{\mu \vdash e_1 \Rightarrow \textbf{false}, \mu'}{\mu \vdash \textbf{while}\ e_1\ \textbf{do}\ e_2 \Rightarrow \textbf{unit}, \mu'}$$

$$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow \textbf{true}, \mu_1 \\ \mu_1 \vdash e_2 \Rightarrow v, \mu_2 \\ \mu_2 \vdash \textbf{while}\ e_1\ \textbf{do}\ e_2 \Rightarrow \textbf{unit}, \mu'\end{array}}{\mu \vdash \textbf{while}\ e_1\ \textbf{do}\ e_2 \Rightarrow \textbf{unit}, \mu'}$$

(BRANCH) $$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow \textbf{true}, \mu_1 \\ \mu_1 \vdash e_2 \Rightarrow v, \mu'\end{array}}{\mu \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Rightarrow v, \mu'}$$

$$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow \textbf{false}, \mu_1 \\ \mu_1 \vdash e_3 \Rightarrow v, \mu'\end{array}}{\mu \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Rightarrow v, \mu'}$$

Fig. 2. The evaluation rules.

The approach taken by Tofte [1990][3] is to prevent the generalization of any type variables that may occur in λ'. In contrast, our system permits some type variables that occur in λ' to be generalized. For example, an expression such as

$$\textbf{letvar } x := [\,] \textbf{ in } x$$

allocates a location l of type α *list*; nevertheless, we *are* allowed to generalize α. The reason it is sound to do so is that the expression evaluates to $[\,]$, and (since l does not occur in $[\,]$) we do not *need* the typing of l to derive a type for $[\,]$. In general, our system keeps track of which variable locations may occur in values and prevents the generalization of type variables that occur in the types of these locations.

We now proceed with the formal development of the subject reduction theorem. First we introduce some useful lemmas.

Lemma 4.1 (SUPERFLUOUSNESS). Suppose that $\lambda; \gamma \vdash e : \tau$. If $l \notin dom(\lambda)$, then $\lambda[l : \tau']; \gamma \vdash e : \tau$ and if $r \notin dom(\lambda)$, then $\lambda[r : \tau']; \gamma \vdash e : \tau$. Also, if $x \notin dom(\gamma)$, then $\lambda; \gamma[x : \rho] \vdash e : \tau$.

Lemma 4.2 (SUBSTITUTION). If $\lambda; \gamma \vdash v : \sigma$ and $\lambda; \gamma[x : \sigma] \vdash e : \tau$, then $\lambda; \gamma \vdash [v/x]e : \tau$. Also, if $\lambda; \gamma \vdash l : \tau\ var$ and $\lambda; \gamma[x : \tau\ var] \vdash e : \tau'$, then $\lambda; \gamma \vdash [l/x]e : \tau'$.

Lemma 4.3 ($\forall$-INTRO). If $\lambda; \gamma \vdash e : \sigma$ and α does not occur free in λ or in γ, then $\lambda; \gamma \vdash e : \forall\alpha\,.\,\sigma$.

The preceding three lemmas are straightforward variants of the lemmas given in Harper [1994]. We also need another lemma:

Lemma 4.4. If $\lambda[l : \tau]; \gamma \vdash e : \tau'$ and l does not occur in e, then $\lambda; \gamma \vdash e : \tau'$.

Finally, we say that l *occurs in the range of* μ if l occurs in $\mu(l')$ for some l' or in $\mu(r)$ for some r. We can now give the subject reduction theorem:

Theorem 4.5. Suppose that $\mu \vdash e \Rightarrow v, \mu'$, $\lambda \vdash e : \tau$, $\mu : \lambda$, and λ assigns weak types to all reference locations in its domain and to all variable locations that occur in the range of μ or in a λ-abstraction in e. Then there exists λ' such that $\lambda \subseteq \lambda'$, $\mu' : \lambda'$, $\lambda' \vdash v : \tau$, and λ' assigns weak types to all reference locations in its domain and to all variable locations that occur in the range of μ' or in v.

PROOF. The proof is by induction on the structure of the derivation of $\mu \vdash e \Rightarrow v, \mu'$. For brevity, we present only the two most interesting cases: (BIND), when e_1 is not a value, and (BINDVAR).

In the (BIND) case, where e_1 is not a value, the evaluation must end with

$$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow v_1, \mu_1\\ \mu_1 \vdash [v_1/x]e_2 \Rightarrow v_2, \mu_2\end{array}}{\mu \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \Rightarrow v_2, \mu_2}$$

[3]The same approach is taken by Wright [1992; 1995] and SML/NJ [Greiner 1993; Hoang et al. 1993], but not by Leroy and Weis [Leroy 1992; Leroy and Weis 1991].

while the typing must end with

$$\frac{\begin{array}{l}\lambda \vdash e_1 : \tau_1 \\ \lambda; [x : \mathit{AppClose}_\lambda(\tau_1)] \vdash e_2 : \tau_2\end{array}}{\lambda \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2}.$$

Also, we have $\mu : \lambda$, and λ assigns weak types to all reference locations in its domain and to all variable locations that occur in the range of μ or in a λ-abstraction in e_1 or in e_2.

By induction, there exists λ_1 such that $\lambda \subseteq \lambda_1$, $\mu_1 : \lambda_1$, $\lambda_1 \vdash v_1 : \tau_1$, and λ_1 assigns weak types to all reference locations in its domain and to all variable locations that occur in the range of μ_1 or in v_1.

Now to apply induction again we want to show that

$$\lambda_1 \vdash [v_1/x]e_2 : \tau_2.$$

By Lemma 4.1 we have

$$\lambda_1; [x : \mathit{AppClose}_\lambda(\tau_1)] \vdash e_2 : \tau_2,$$

so we can apply Lemma 4.2 to get what we want provided that we can show

$$\lambda_1 \vdash v_1 : \mathit{AppClose}_\lambda(\tau_1).$$

Now, applying Lemma 4.3 to $\lambda_1 \vdash v_1 : \tau_1$ we can get $\lambda_1 \vdash v_1 : \mathit{AppClose}_{\lambda_1}(\tau_1)$, but this is not good enough, because λ_1 may contain free strong type variables that are not free in λ. To proceed, we exploit our knowledge about what locations can occur in v_1.

Let λ_1^- be formed by removing from λ_1 any typings $l : \tau$ such that τ is not weak. By the above use of induction, this process does not remove any typings of locations that occur in v_1, as all such locations have weak types. So by Lemma 4.4, $\lambda_1^- \vdash v_1 : \tau_1$. Hence, by Lemma 4.3, $\lambda_1^- \vdash v_1 : \mathit{AppClose}_\lambda(\tau_1)$, since λ_1^- contains no strong type variables. Lemma 4.1 then gives $\lambda_1 \vdash v_1 : \mathit{AppClose}_\lambda(\tau_1)$, and finally by Lemma 4.2 we get $\lambda_1 \vdash [v_1/x]e_2 : \tau_2$.

By the use of induction above, λ_1 assigns weak types to all reference locations in its domain and to all variable locations that occur in the range of μ_1. Furthermore, if a variable location l occurs in a λ-abstraction in $[v_1/x]e_2$, then either l occurs in v_1, or l occurs in a λ-abstraction in e_2. In the first case, $\lambda_1(l)$ is weak by the above use of induction; in the second case, $\lambda(l)$ is weak by the hypothesis, and so $\lambda_1(l)$ is weak since $\lambda \subseteq \lambda_1$.

Hence we can use induction a second time to show that there exists λ' such that $\lambda_1 \subseteq \lambda'$, $\mu_2 : \lambda'$, $\lambda' \vdash v_2 : \tau_2$, and λ' assigns weak types to all reference locations in its domain and to all variable locations that occur in the range of μ_2 or in v_2. Since $\lambda \subseteq \lambda_1 \subseteq \lambda'$, we are done.

As for the (BINDVAR) case, the evaluation must end with

$$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow v_1, \mu_1 \\ l \notin \mathit{dom}(\mu_1) \\ \mu_1[l := v_1] \vdash [l/x]e_2 \Rightarrow v_2, \mu_2\end{array}}{\mu \vdash \textbf{letvar } x := e_1 \textbf{ in } e_2 \Rightarrow v_2, \mu_2}$$

while the typing must end with

$$\frac{\begin{array}{l}\lambda \vdash e_1 : \tau_1 \\ \lambda; [x : \tau_1\ var] \vdash e_2 : \tau_2 \\ \text{If } x \text{ occurs in a } \lambda\text{-abstraction in } e_2 \text{ then } \tau_1 \text{ is weak.}\end{array}}{\lambda \vdash \textbf{letvar}\ x := e_1\ \textbf{in}\ e_2 : \tau_2}.$$

Also, we have $\mu : \lambda$, and λ assigns weak types to all reference locations in its domain and to all variable locations that occur in the range of μ or in a λ-abstraction in e_1 or in e_2.

By induction, there exists λ_1 such that $\lambda \subseteq \lambda_1$, $\mu_1 : \lambda_1$, $\lambda_1 \vdash v_1 : \tau_1$, and λ_1 assigns weak types to all reference locations in its domain and to all variable locations that occur in the range of μ_1 or in v_1.

Since $l \notin dom(\lambda_1)$, $\lambda_1 \subseteq \lambda_1[l : \tau_1]$.

Since $\lambda_1[l : \tau_1] \vdash l : \tau_1\ var$ and (by Lemma 4.1) $\lambda_1[l : \tau_1]; [x : \tau_1\ var] \vdash e_2 : \tau_2$, we can apply Lemma 4.2 to get

$$\lambda_1[l : \tau_1] \vdash [l/x]e_2 : \tau_2.$$

Also, $\mu_1[l := v_1] : \lambda_1[l : \tau_1]$ by Lemma 4.1.

Next, by the use of induction above, $\lambda_1[l : \tau_1]$ assigns weak types to all reference locations in its domain and to all variable locations that occur in the range of $\mu_1[l := v_1]$. Now suppose that a variable location l' occurs in a λ-abstraction in $[l/x]e_2$. Then either l' occurs in a λ-abstraction in e_2, or else $l' = l$ and x occurs in a λ-abstraction in e_2. In the first case, by the hypothesis $\lambda(l')$ is weak, and so $\lambda_1[l : \tau_1](l')$ is weak. In the second case, by the restriction on the (LETVAR) rule, τ_1 is weak, and so $\lambda_1[l : \tau_1](l')$ is weak.

So by a second use of induction, there exists λ' such that $\lambda_1[l : \tau_1] \subseteq \lambda'$, $\mu_2 : \lambda'$, $\lambda' \vdash v_2 : \tau_2$, and λ' assigns weak types to all reference locations in its domain and to all variable locations that occur in the range of μ_2 or in v_2. Since $\lambda \subseteq \lambda_1 \subseteq \lambda_1[l : \tau_1] \subseteq \lambda'$, we are done. □

Type soundness actually involves more than the subject reduction property. However, it is straightforward to extend the subject reduction theorem to show that well-typed programs cannot suffer run-time type errors. This requires an easy *canonical forms* lemma about the type system, that tells us that a closed value of some type has the proper form. For example, a closed value of type $\tau \rightarrow \tau'$ must be a λ-abstraction. Harper [1996] discusses this more fully.

5. DISCUSSION

One of our primary objectives has been to simplify the types of imperative programs as much as possible. It is often argued that too much information in types makes them unsuitable as specifications in module interfaces. This has also been a goal of Wright's system based on syntactic values. His system is a restriction of Tofte's system in that *all* type variables are considered imperative, regardless of whether references are used.[4] To restore polymorphism in practice, often η-expansion will do. However, there are cases when η-expansion does not work, in particular, when

[4]Indeed, the technical report [Wright 1993] describing this system would be more accurately titled "Polymorphism for Imperative Languages without *Applicative* Types."

computing polymorphic procedures with imperative features. For example, in his system, `makeCountFun`, expressed using `let` and `ref`, is effectively given type

$$\forall \_\alpha, \_\beta \,.\, (\_\alpha \rightarrow \_\beta) \rightarrow (\_\alpha \rightarrow \_\beta) \times (\mathit{unit} \rightarrow \mathit{int}).$$

Consequently, the application `makeCountFun hd` is not polymorphic, and restoring it by η-expansion will not work, since each time the expansion is called a new counter is created. Wright argues that in practice when polymorphic procedures are computed, the computation is almost always functional, so polymorphism can easily be restored by η expansion. Even if η expansion does work, there is also the issue of call-by-name inefficiency as there is in Leroy's proposal for call-by-name polymorphism [Leroy 1993]. Shared intermediate polymorphism through partial application of curried functions is lost. In view of these deficiencies, our system with `letvar` is an attractive alternative. It is relatively simple and greatly reduces the need for weak types.

Our system is not perfect, however. The restriction on rule (LETVAR) sometimes forces variables to be given weak types unnecessarily. For example, consider the following function that computes the Cartesian product of two lists:

```
fun icart xs ys = letvar a := xs in
  letvar b := [] in
    while not (null a) do
      ( b := (map (fn y => (hd a, y)) ys) @ b;
        a := tl a);
    b
  end end
```

The mere occurrence of variable `a` in `(fn y => (hd a, y))` forces it to be given a weak type. Hence the best type we can give `icart` is

$$\forall \_\alpha, \beta \,.\, \_\alpha \ \mathit{list} \rightarrow \beta \ \mathit{list} \rightarrow (\_\alpha \times \beta) \ \mathit{list},$$

even though it should be fully polymorphic.

Similarly, the functional style of programming that codes loops using tail recursion leads our system to assign weak types unnecessarily. This is why we have included the **while** loop as a primitive in our language.

We conjecture[5] that the restriction in the (LETVAR) rule can be relaxed to

If x is *assigned to* within a λ-abstraction in e_2, then τ_1 is weak.

This is similar to Edinburgh LCF's restriction 2*ib* [Gordon et al. 1979, p. 49]. Proving soundness now requires a different strategy than the one used here, because now variable locations with strong types *can* occur in values, as in examples like **letvar** $x := [\,]$ **in** $\lambda y.\, x$.

Under the relaxed restriction, function `icart` can be fully polymorphic, because variable `a` is not assigned to within `(fn y => (hd a, y))`. However, even the relaxed restriction can force weak types to be introduced unnecessarily. For example, a faster version of `icart` can be obtained by eliminating list concatenation:

[5]This conjecture has now been established for a core language with variables but no references [Volpano and Smith 1995].

```
fun fast_icart xs ys = letvar a := xs in
   letvar b := [] in
      while not (null a) do
         ( map (fn y => b := (hd a, y) :: b) ys;
           a := tl a);
      b
   end end
```

The relaxed restriction forces both `xs` and `ys` to have weak type, although it is safe for them to have strong type.

6. CONCLUSIONS

The type system presented here is appealing in its combination of expressiveness and simplicity. It also clarifies the relationship between variables and references. For example, C has the conversion operator & for taking the address of a variable or array element. We can introduce & by including the typing rule

$$\frac{\begin{array}{l}\lambda; \gamma \vdash e : \tau \ \mathit{var} \\ \tau \text{ is weak}\end{array}}{\lambda; \gamma \vdash \& e : \tau \ \mathit{ref}}$$

which is nicely symmetric to rule (L-VAL) [Volpano and Smith 1996]. Finally, this work has provided a basis for polymorphic typing in the C programming language [Smith and Volpano 1996].

REFERENCES

Damas, L. 1985. Type assignment in programming languages. Ph.D. thesis, Univ. of Edinburgh.

Damas, L. and Milner, R. 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages.* ACM, New York, 207–212.

Gordon, M., Milner, R., and Wadsworth, C. 1979. *Edinburgh LCF.* Lecture Notes in Computer Science, vol. 78. Springer-Verlag, Berlin.

Greiner, J. 1993. Standard ML weak polymorphism can be sound. Tech. Rep. CMU-CS-93-160, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa. May.

Harper, R. 1994. A simplified account of polymorphic references. *Inf. Process. Lett. 51*, 201–206.

Harper, R. 1996. A note on “A simplified account of polymorphic references.” *Inf. Process. Lett. 57*, 15–16.

Hoang, M., Mitchell, J., and Viswanathan, R. 1993. Standard ML/NJ weak polymorphism and imperative constructs. In *Proceedings of the 8th IEEE Symposium on Logic in Computer Science.* IEEE, New York.

Leroy, X. 1992. Polymorphic typing of an algorithmic language. Ph.D. thesis, INRIA-Rocquencourt Res. Rep. 1778, Le Chesnay, France.

Leroy, X. 1993. Polymorphism by name for references and continuations. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages.* ACM, New York, 220–231.

Leroy, X. and Weis, P. 1991. Polymorphic type inference and assignment. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages.* ACM, New York, 291–302.

Smith, G. and Volpano, D. 1996. Towards an ML-style polymorphic type system for C. In *Proceedings of the 6th European Symposium on Programming.* Lecture Notes in Computer Science. Springer-Verlag, Berlin. To appear.

Talpin, J.-P. and Jouvelot, P. 1992. The type and effect discipline. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science.* IEEE, New York, 162–173.

Tofte, M. 1990. Type inference for polymorphic references. *Inf. Comput. 89*, 1–34.

Volpano, D. and Smith, G. 1995. A type soundness proof for variables in LCF ML. *Inf. Process. Lett. 56*, 141–146.

Volpano, D. and Smith, G. 1996. A note on typing variables and references. Tech. Rep. NPS-CS-96-003, Computer Science Dept., Naval Postgraduate School, Monterey, Calif.

Wright, A. 1992. Typing references by effect inference. In *Proceedings of the 4th European Symposium on Programming.* Lecture Notes in Computer Science, vol. 582. Springer-Verlag, Berlin, 473–491.

Wright, A. 1993. Polymorphism for imperative languages without imperative types. Tech. Rep. TR 93-200, Dept. of Computer Science, Rice Univ., Houston, Tex.

Wright, A. 1995. Simple imperative polymorphism. *J. Lisp Symb. Comput. 8,* 4 (Dec.), 343–356.

Wright, A. and Felleisen, M. 1994. A syntactic approach to type soundness. *Inf. Comput. 115,* 1 (Nov.), 38–94.

Received March 1995; revised September 1995; accepted January 1996