Towards an ML-Style Polymorphic Type System for C*

Geoffrey Smith[1] and Dennis Volpano[2]

[1] School of Computer Science, Florida International University, Miami, FL 33199, USA, email: smithg@fiu.edu
[2] Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943, USA, email: volpano@cs.nps.navy.mil

Abstract. Advanced polymorphic type systems have come to play an important role in the world of functional programming. But, curiously, these type systems have so far had little impact upon widely-used imperative programming languages like C and C++. We show that ML-style polymorphism can be integrated smoothly into a dialect of C, which we call *Polymorphic C*. It has the same pointer operations as C, including the address-of operator &, the dereferencing operator *, and pointer arithmetic. Our type system allows these operations in their full generality, so that programmers need not give up the flexibility of C to gain the benefits of ML-style polymorphism. We prove a type soundness theorem that gives a rigorous and useful characterization of well-typed Polymorphic C programs in terms of what can go wrong when they are evaluated.

1 Introduction

Much attention has been given to developing sound polymorphic type systems for languages with imperative features. Most notable is the large body of work surrounding ML [GMW79, Tof90, LeW91, SML93, Wri95, VoS95]. However, none of these efforts addresses the polymorphic typing of variables, arrays and pointers (first-class references), which are essential ingredients of any traditional imperative language. As a result, they cannot be directly applied to get ML-style polymorphic extensions of widely-used languages like C and C++.

This paper presents a provably-sound type system for a polymorphic dialect of C, called Polymorphic C. It has the same pointer operations as C, including the address-of operator &, the dereferencing operator *, and pointer arithmetic. The type system allows these operations without any restrictions on them so that programmers can enjoy C's pointer flexibility and yet have type security

* 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.

and polymorphism as in ML. Our type system demonstrates that ML-style polymorphism can be brought cleanly and elegantly into the realm of traditional imperative languages.

We establish a type soundness theorem that gives a rigorous and useful characterization of well-typed Polymorphic C programs in terms of what can go wrong when they are evaluated. Our approach uses a natural-style semantics and a formulation of subject reduction based on Harper's syntactic approach [Har94]. It is simple and does not require a separate type semantics. We expect it to be useful in proving type soundness for a wide variety of imperative languages having first-class pointers and mutable variables and arrays.

We begin with an overview of Polymorphic C in the next section. Then we formally describe its syntax, type system, and semantics. Then, in Section 4 we establish the soundness of the type system.

2 An Overview of Polymorphic C

Polymorphic C is intended to be as close to the core of Kernighan and Ritchie C [KR78] as possible. In particular, it is stack-based with variables, pointers, and arrays. Pointers are dereferenced explicitly using $*$, while variables are dereferenced implicitly. Furthermore, pointers are first-class values, but variables are not. Polymorphic C has the same pointer operations as C. A well-typed Polymorphic C program in our system may still suffer from dangling reference and illegal address errors. Our focus has not been on eliminating such pointer insecurities, which would require weakening C's expressive power, but rather on adding ML-style polymorphism to C, so that programmers can write polymorphic functions naturally and soundly as they would in Standard ML, rather than by parameterizing functions on data sizes or by using pointers of type `void *`.

Syntactically, Polymorphic C uses a flexible syntax similar to that of core-ML of Damas and Milner [DaM82]. For example, here is a Polymorphic C function that reverses the elements of an array:

$$
\begin{array}{l}
\textbf{let } \mathit{swap} = \lambda x, y.\, \textbf{letvar } t := *x \textbf{ in } \; *x := *y; \;\; *y := t \\
\textbf{in} \\
\textbf{let } \mathit{reverse} = \lambda a, n.\, \textbf{letvar } i := 0 \textbf{ in} \\
\qquad \textbf{while } i < n - 1 - i \textbf{ do} \\
\qquad\quad \mathit{swap}(a + i,\ a + n - 1 - i); \\
\qquad\quad i := i + 1 \\
\textbf{in} \ldots
\end{array}
$$

The construct **letvar** $x := e_1$ **in** e_2 binds x to a new cell initialized to the value of e_1; the scope of the binding is e_2 and the lifetime of the cell ends after e_2 is evaluated. Variable x is dereferenced implicitly. This is achieved via a typing rule that says that if e has type $\tau\ \mathit{var}$, then it also has type τ.

As in C, the call to *swap* in *reverse* could equivalently be written as

$$\mathit{swap}(\&a[i],\ \&a[n - 1 - i])$$

and also as in C, array subscripting is syntactic sugar: $e_1[e_2]$ is equivalent to $*(e_1+e_2)$. Arrays themselves are introduced by the construct **letarr** $x[e_1]$ **in** e_2, which binds x to a pointer to an uninitialized array whose size is the value of e_1; the scope of x is e_2, and the lifetime of the array ends after e_2 is evaluated.

The type system of Polymorphic C assigns types of the form τ *var* to variables, and types of the form τ *ptr* to pointers.[3] Functions *swap* and *reverse* given above are polymorphic; *swap* has type

$$\forall\alpha\,.\,\alpha\ ptr \times \alpha\ ptr \rightarrow \alpha$$

while *reverse* has type

$$\forall\alpha\,.\,\alpha\ ptr \times int \rightarrow unit$$

Notice that pointer and array types are unified as in C. Also, variable and pointer types are related by symmetric typing rules for & and $*$: if $e : \tau\ var$, then $\&e : \tau\ ptr$, and if $e : \tau\ ptr$, then $*e : \tau\ var$. Note that dereferencing in Polymorphic C differs from dereferencing in Standard ML, where if $e : \tau\ ref$, then $!e : \tau$.

Polymorphic C's types are stratified into three levels. There are the ordinary τ (data types) and σ (type schemes) type levels of Damas and Milner's system [DaM82], and a new level called *phrase types* containing σ types and variable types of the form τ *var*. This stratification enforces the "second-class" status of variables: for example, the return type of a function must be a data type, so that one cannot write a function that returns a variable. On the other hand, pointer types are included among the data types, making pointers first-class values.

Polymorphic C has been designed to ensure that function calls can be implemented on a stack without the use of static links or displays. In traditional imperative languages, this property has been achieved by rigidly fixing the syntactic structure of programs. For example, in C, functions can only be defined at top level. But such syntactic restrictions are often complex and unnecessarily restrictive. In contrast, Polymorphic C adopts a completely free syntax, as in core-ML. The ability to implement Polymorphic C on a stack, without static links or displays, is achieved by imposing one key restriction on lambda abstractions: *the free identifiers of any lambda abstraction must be declared at top level.* Roughly speaking, a top-level declaration is one whose scope extends all the way to the end of the program. For example, in the program

let $f = \ldots$ **in**
letvar $x := \ldots$ **in**
letarr $a[\ldots]$ **in** $f(\ldots)$

the identifiers declared at top level are f, x, and a. Although they are severely restricted, Polymorphic C's anonymous lambda abstractions are convenient at times. For example, we can write $map(\lambda n.\, n + 1, [4, 2, 5])$ without having to declare a named successor function. Nevertheless, one might prefer a different syntax for Polymorphic C; it should be noted that there would be no obstacle to adopting a more C-like syntax.

[3] We use *ptr* rather than *ref* to avoid confusion with C++ and ML references.

2.1 The Issue of Type Soundness in Polymorphic C

Much effort has been spent trying to develop sound polymorphic type systems for imperative extensions of core-ML. Especially well-studied is the problem of typing Standard ML's first-class references [Tof90, LeW91, SML93, Wri95]. The problem is easier in a language with variables but no references, such as Edinburgh LCF ML, but subtle problems still arise [GMW79]. The key problem is that a variable can escape its scope via a lambda abstraction as in

$$\textbf{letvar } stk := [\,] \textbf{ in } \lambda v.\, stk := v :: stk$$

In this case, the type system must not allow type variables that occur in the type of *stk* to be generalized. Different mechanisms have been proposed for dealing with this problem [GMW79, VoS95]

In the context of Polymorphic C, however, we can adopt an especially simple approach. Because of the restriction on the free identifiers of lambda abstractions, Polymorphic C does not allow a polymorphic value to be *computed* in an interesting way; for example, we cannot write curried functions. For this reason, we suffer essentially no loss of language expressiveness by limiting polymorphism to *syntactic values*, that is, identifiers, literals, and lambda abstractions [Tof90].[4]

Limiting polymorphism to syntactic values ensures the soundness of polymorphic generalizations, but pointers present new problems for type soundness. If one is not careful in formulating the semantics, then the subject reduction property may not hold. For example, if a program can dereference a pointer to a cell that has been deallocated and then reallocated, then the value obtained may have the wrong type. Our semantics is designed to catch all pointer errors.

3 The Polymorphic C Language

The syntax of Polymorphic C is given below. For the sake of describing the type system, we need to distinguish a subset of the expressions, called *Values*, which are the syntactic values [Tof90, Wri95] of the language:

$$\begin{aligned}
(\textit{Expr})\quad e ::= {} & v \mid e(e_1,\ldots,e_n) \mid e_1 := e_2 \mid \\
& \&e \mid *e \mid e_1 + e_2 \mid e_1[e_2] \mid e_1; e_2 \mid \\
& \textbf{while } e_1 \textbf{ do } e_2 \mid \\
& \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \mid \\
& \textbf{let } x = e_1 \textbf{ in } e_2 \mid \\
& \textbf{letvar } x := e_1 \textbf{ in } e_2 \mid \\
& \textbf{letarr } x[e_1] \textbf{ in } e_2 \mid \\
& (a, 1) \\
(\textit{Values})\, v ::= {} & x \mid c \mid \lambda x_1,\ldots,x_n.e \mid (a, 0)
\end{aligned}$$

[4] In the context of a language with first-class functions, limiting polymorphism to syntactic values does limit the expressiveness of the language. But Wright argues that even then the loss of expressiveness is not a problem in practice [Wri95].

Meta-variable x ranges over identifiers, c over literals (such as integer literals and **unit**), and a over addresses. All free identifiers of every lambda abstraction must be declared at *top level*; this restriction can be precisely defined by an attribute grammar.

The expressions $(a, 1)$ and $(a, 0)$ are *variables* and *pointers*, respectively. These will not actually occur in user programs; they are included in the language solely for the purpose of simplifying the semantics, as will become clear in Section 3.2. Notice that pointers are values, but variables are not; this reflects the fact that variables are implicitly dereferenced, while pointers are not.

The + operator here denotes only pointer arithmetic. In the full language, + would be overloaded to denote integer addition as well.

A subtle difference between C and Polymorphic C is that the formal parameters of a Polymorphic C function are *constants* rather than local variables. Hence the C function `f(x) { b }` is equivalent to

$$\textbf{let } f = \lambda x.\, \textbf{letvar } x := x \textbf{ in } b \textbf{ in } \ldots$$

in Polymorphic C. Also, Polymorphic C cannot directly express C's internal static variables. For example, the C declaration

```
f(x) { static int n = 0;  b }
```

corresponds directly to the Polymorphic C expression

$$\textbf{let } f = \textbf{letvar } n := 0 \textbf{ in } \lambda x.\, b \textbf{ in } \ldots$$

but this violates the restriction on lambda abstractions if n is free in b. Such functions must be transformed to eliminate static variables in favor of uniquely-renamed global variables:

$$\textbf{letvar } n := 0 \textbf{ in let } f = \lambda x.\, b \textbf{ in } \ldots$$

3.1 The Type System of Polymorphic C

The types of Polymorphic C are stratified as follows.

$$\begin{array}{lll} \tau ::= \alpha \mid \mathit{int} \mid \mathit{unit} \mid \tau\ \mathit{ptr} \mid \tau_1 \times \cdots \times \tau_n \to \tau & & (\textit{data types}) \\ \sigma ::= \forall \alpha.\, \sigma \mid \tau & & (\textit{type schemes}) \\ \rho ::= \sigma \mid \tau\ \mathit{var} & & (\textit{phrase types}) \end{array}$$

Meta-variable α ranges over *type variables*. Compared to the type system of Standard ML, all type variables in Polymorphic C are imperative [Tof90].

The rules of the type system are formulated as they are in Harper's system [Har94] and are given in Figure 1.[5] It is a deductive proof system used to assign types to expressions. Typing judgements have the form

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

[5] For brevity, we have omitted typing rules for sequential composition, **if**, and **while**.

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

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

(PTR) $\lambda;\gamma \vdash ((i,j),0) : \tau\ ptr \qquad \lambda(i) = \tau$

(VAR) $\lambda;\gamma \vdash ((i,j),1) : \tau\ var \qquad \lambda(i) = \tau$

(LIT) $\lambda;\gamma \vdash c : int \qquad c$ is an integer literal

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

(→-INTRO) $$\frac{\lambda;\gamma[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash e : \tau}{\lambda;\gamma \vdash \lambda x_1, \ldots, x_n.\, e : \tau_1 \times \cdots \times \tau_n \to \tau}$$

(→-ELIM) $$\frac{\lambda;\gamma \vdash e : \tau_1 \times \cdots \times \tau_n \to \tau,\ \lambda;\gamma \vdash e_i : \tau_i,\ \ 1 \leq i \leq n}{\lambda;\gamma \vdash e(e_1, \ldots, e_n) : \tau}$$

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

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

(LETVAR) $$\frac{\lambda;\gamma \vdash e_1 : \tau_1,\ \ \lambda;\gamma[x : \tau_1\ var] \vdash e_2 : \tau_2}{\lambda;\gamma \vdash \mathbf{letvar}\ x := e_1\ \mathbf{in}\ e_2 : \tau_2}$$

(LETARR) $$\frac{\lambda;\gamma \vdash e_1 : int,\ \ \lambda;\gamma[x : \tau_1\ ptr] \vdash e_2 : \tau_2}{\lambda;\gamma \vdash \mathbf{letarr}\ x[e_1]\ \mathbf{in}\ e_2 : \tau_2}$$

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

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

(ADDRESS) $$\frac{\lambda;\gamma \vdash e : \tau\ var}{\lambda;\gamma \vdash \&e : \tau\ ptr}$$

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

(ARITH) $$\frac{\lambda;\gamma \vdash e_1 : \tau\ ptr,\ \ \lambda;\gamma \vdash e_2 : int}{\lambda;\gamma \vdash e_1 + e_2 : \tau\ ptr}$$

(SUBSCRIPT) $$\frac{\lambda;\gamma \vdash e_1 : \tau\ ptr,\ \ \lambda;\gamma \vdash e_2 : int}{\lambda;\gamma \vdash e_1[e_2] : \tau\ var}$$

Fig. 1. Rules of the Type System

meaning that expression e has type ρ, assuming that γ prescribes phrase types for the free identifiers of e and λ prescribes data types for the variables and pointers in e. More precisely, meta-variable 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.

Meta-variable λ ranges over address typings, which are needed in typing the values produced by programs. One might expect that addresses would just be natural numbers, but that would not allow the semantics to detect invalid pointer arithmetic. So instead an address is a pair of natural numbers (i, j) where i is the *segment number* and j is the *offset*. Intuitively, we put each variable or array into its own segment. Thus a simple variable has address $(i, 0)$, and an n-element array has addresses

$$(i, 0), (i, 1), \ldots, (i, n-1)$$

Pointer arithmetic involves only the offset of an address, and dereferencing nonexistent or dangling pointers is detected as a "segmentation fault". An *address typing* then is a finite function mapping segment numbers to data types. The reason it does not map addresses to data types is that nonexistent pointers can be produced as values of programs, and such pointers must therefore be typable if subject reduction is to hold. For example, the program

$$\textbf{letarr } a[10] \textbf{ in } a + 17$$

is well typed and evaluates to $((0, 17), 0)$, a nonexistent pointer. The notational conventions for address typings are similar to those for identifier typings.

The *generalization* of a data type τ relative to λ and γ, written $Close_{\lambda;\gamma}(\tau)$, 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 $Close_{\lambda}(\tau)$ when $\gamma = \emptyset$. 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 $\sigma \geq \tau$ whenever $\sigma' \geq \tau$. Finally, we say that $\lambda; \gamma \vdash e : \sigma$ if $\lambda; \gamma \vdash e : \tau$ whenever $\sigma \geq \tau$.

The type system has the property that the *type* of a value determines the *form* of the value; also, an expression of type τ *var* can have only two possible forms:

Lemma 1 (Correct Form). *Suppose* $\lambda \vdash v : \tau$. *Then*

- *if* τ *is int, then* v *is an integer literal,*
- *if* τ *is unit, then* v *is* **unit**,
- *if* τ *is* τ' *ptr, then* v *is of the form* $((i, j), 0)$, *and*
- *if* τ *is* $\tau_1 \times \cdots \times \tau_n \to \tau_{n+1}$, *then* v *is of the form* $\lambda x_1, \ldots, x_n.e$.

Furthermore, if $\lambda \vdash e : \tau$ *var, then* e *is of the form* $((i, j), 1)$ *or of the form* $*e'$.[6]

Proof. Immediate from inspection of the typing rules. □

[6] Note that this assumes that array subscripting is syntactic sugar.

A consequence of the last part of this lemma is that if $\lambda \vdash e : \tau$ and e is not of the form $((i,j),1)$ or $*e'$, then derivation of the typing judgement cannot end with rule (R-VAL). So the typing rules, for the most part, remain syntax directed. The fact that variables can have only two possible forms is exploited in our structured operational semantics, specifically within rules (REF) and (UPDATE).

3.2 The Semantics of Polymorphic C

We give a structured operational semantics. A closed expression is evaluated relative to a *memory* μ, which is a finite function from addresses to values. It may also map an address to **dead** or **uninit**, indicating that the cell with that address has been deallocated or is uninitialized. The contents of an address $a \in dom(\mu)$ is the value $\mu(a)$, and we write $\mu[a := v]$ for the memory that assigns value v to address a, and value $\mu(a')$ to an address $a' \neq a$; $\mu[a := v]$ is an *update* of μ if $a \in dom(\mu)$ and an *extension* of μ if $a \notin dom(\mu)$.

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

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

which asserts 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. Note the use of substitution in rules (APPLY), (BIND), (BINDVAR), and (BINDARR). It allows us to avoid environments and closures in the semantics, so that the result of evaluating a Polymorphic C expression is just another expression in Polymorphic C. This is made possible by the flexible syntax of the language and the fact that all expressions are closed, including lambda abstractions.

4 Semantic Soundness

In this section, we establish the soundness of our type system. We begin by using the framework of Harper [Har94] to show *subject reduction*, which basically asserts that if $\vdash e : \tau$ and $\vdash e \Rightarrow v, \mu'$, then $\vdash v : \tau$. But since e can allocate addresses and they can occur in v, the conclusion must actually be that there exists an address 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 $\mu : \lambda$ if

1. $dom(\lambda) = \{i \mid (i,0) \in dom(\mu)\}$, and
2. for all (i,j), $\lambda \vdash \mu((i,j)) : \lambda(i)$ if $\mu((i,j))$ is a value.

Note that λ must give a type to uninitialized and dead addresses of μ, but the type can be anything.

Before giving the subject reduction theorem, we require a number of lemmas that establish some useful properties of the type system. We begin with a fundamental type substitution lemma:

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

(CONTENTS) $$\frac{a \in dom(\mu) \text{ and } \mu(a) = v}{\mu \vdash (a,1) \Rightarrow v, \mu}$$

(DEREF) $$\frac{\mu \vdash e \Rightarrow (a,0), \mu' \quad a \in dom(\mu') \text{ and } \mu'(a) = v}{\mu \vdash *e \Rightarrow v, \mu'}$$

(REF) $\mu \vdash \&(a,1) \Rightarrow (a,0), \mu$

$$\frac{\mu \vdash e \Rightarrow (a,0), \mu'}{\mu \vdash \& * e \Rightarrow (a,0), \mu'}$$

(OFFSET) $$\frac{\mu \vdash e_1 \Rightarrow ((i,j),0), \mu_1 \quad \mu_1 \vdash e_2 \Rightarrow n, \mu' \ (n \text{ an integer})}{\mu \vdash e_1 + e_2 \Rightarrow ((i, j+n), 0), \mu'}$$

(APPLY) $$\frac{\mu \vdash e \Rightarrow \lambda x_1, \ldots, x_n.\, e', \mu_1 \quad \mu_1 \vdash e_1 \Rightarrow v_1, \mu_2 \quad \ldots \quad \mu_n \vdash e_n \Rightarrow v_n, \mu_{n+1} \quad \mu_{n+1} \vdash [v_1, \ldots, v_n/x_1, \ldots, x_n]e' \Rightarrow v, \mu'}{\mu \vdash e(e_1, \ldots, e_n) \Rightarrow v, \mu'}$$

(UPDATE) $$\frac{\mu \vdash e \Rightarrow v, \mu' \quad a \in dom(\mu') \text{ and } \mu'(a) \neq \textbf{dead}}{\mu \vdash (a,1) := e \Rightarrow v, \mu'[a := v]}$$

$$\frac{\mu \vdash e_1 \Rightarrow (a,0), \mu_1 \quad \mu_1 \vdash e_2 \Rightarrow v, \mu_2 \quad a \in dom(\mu_2) \text{ and } \mu_2(a) \neq \textbf{dead}}{\mu \vdash *e_1 := e_2 \Rightarrow v, \mu_2[a := v]}$$

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

(BINDVAR) $$\frac{\mu \vdash e_1 \Rightarrow v_1, \mu_1 \quad (i,0) \notin dom(\mu_1) \quad \mu_1[(i,0) := v_1] \vdash [((i,0),1)/x]e_2 \Rightarrow v_2, \mu_2}{\mu \vdash \textbf{letvar } x := e_1 \textbf{ in } e_2 \Rightarrow v_2, \mu_2[(i,0) := \textbf{dead}]}$$

(BINDARR) $$\frac{\mu \vdash e_1 \Rightarrow n, \mu_1 \ (n \text{ a positive integer}) \quad (i,0) \notin dom(\mu_1) \quad \mu_1[(i,0), \ldots, (i, n-1) := \textbf{uninit}, \ldots, \textbf{uninit}] \vdash [((i,0),0)/x]e_2 \Rightarrow v_2, \mu_2}{\mu \vdash \textbf{letarr } x[e_1] \textbf{ in } e_2 \Rightarrow v_2, \mu_2[(i,0), \ldots, (i, n-1) := \textbf{dead}, \ldots, \textbf{dead}]}$$

Fig. 2. The Evaluation Rules

Lemma 2 (Type Substitution). *If $\lambda;\gamma \vdash e : \tau$, then for any substitution S, $S\lambda; S\gamma \vdash e : S\tau$, and the latter typing has a derivation no longer than the former.*

Lemma 3 (Superfluousness). *Suppose that $\lambda;\gamma \vdash e : \tau$. If $i \notin dom(\lambda)$, then $\lambda[i : \tau'];\gamma \vdash e : \tau$, and if $x \notin dom(\gamma)$, then $\lambda;\gamma[x : \rho] \vdash e : \tau$.*

Lemma 4 (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 $\lambda;\gamma \vdash (a,1) : \tau\ var$ and $\lambda;\gamma[x : \tau\ var] \vdash e : \tau'$, then $\lambda;\gamma \vdash [(a,1)/x]e : \tau'$.*

The preceding lemma does not hold for arbitrary expression substitution.

Lemma 5 (∀-intro). *If $\lambda;\gamma \vdash e : \tau$ and $\alpha_1, \ldots, \alpha_n$ do not occur free in λ or in γ, then $\lambda;\gamma \vdash e : \forall\alpha_1, \ldots, \alpha_n . \tau$.*

We can now give the subject reduction theorem:

Theorem 6 (Subject Reduction). *If $\mu \vdash e \Rightarrow v, \mu'$, $\lambda \vdash e : \tau$, and $\mu : \lambda$, then there exists λ' such that $\lambda \subseteq \lambda'$, $\mu' : \lambda'$, and $\lambda' \vdash v : \tau$.*

Proof. By induction on the structure of the derivation of $\mu \vdash e \Rightarrow v, \mu'$. Here we just show the (BINDVAR) and (BIND) cases.

(BINDVAR). The evaluation must end with

$$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow v_1, \mu_1 \\ (i,0) \notin dom(\mu_1) \\ \mu_1[(i,0) := v_1] \vdash [((i,0),1)/x]e_2 \Rightarrow v_2, \mu_2\end{array}}{\mu \vdash \textbf{letvar } x := e_1 \textbf{ in } e_2 \Rightarrow v_2, \mu_2[(i,0) := \textbf{dead}]}$$

while the typing must end with (LETVAR):

$$\frac{\begin{array}{l}\lambda \vdash e_1 : \tau_1 \\ \lambda;[x : \tau_1\ var] \vdash e_2 : \tau_2\end{array}}{\lambda \vdash \textbf{letvar } x := e_1 \textbf{ in } e_2 : \tau_2}$$

and $\mu : \lambda$. By induction, there exists λ_1 such that $\lambda \subseteq \lambda_1$, $\mu_1 : \lambda_1$, and $\lambda_1 \vdash v_1 : \tau_1$. Since $\mu_1 : \lambda_1$ and $(i,0) \notin dom(\mu_1)$, also $i \notin dom(\lambda_1)$. So $\lambda_1 \subseteq \lambda_1[i : \tau_1]$. By rule (VAR),

$$\lambda_1[i : \tau_1] \vdash ((i,0),1) : \tau_1\ var$$

and by Lemma 3,

$$\lambda_1[i : \tau_1];[x : \tau_1\ var] \vdash e_2 : \tau_2$$

So we can apply Lemma 4 to get

$$\lambda_1[i : \tau_1] \vdash [((i,0),1)/x]e_2 : \tau_2$$

Also, $\mu_1[(i,0) := v_1] : \lambda_1[i : \tau_1]$. So by a second use of induction, there exists λ' such that $\lambda_1[i : \tau_1] \subseteq \lambda'$, $\mu_2 : \lambda'$, and $\lambda' \vdash v_2 : \tau_2$.

It only remains to show that $\mu_2[(i,0) := \textbf{dead}] : \lambda'$. But this follows immediately from $\mu_2 : \lambda'$.

Remark. What would go wrong if we simply removed the deallocated address $(i, 0)$ from the domain of the final memory, rather than marking it **dead**? Well, with the current definition of $\mu : \lambda$, we would then be forced to remove i from the final address typing. But then $\mu_2 - i : \lambda' - i$ would fail, if there were any dangling pointers $((i, j), 0)$ in the range of $\mu_2 - i$. If, instead, we allowed λ' to retain the typing for i, then the next time that $(i, 0)$ were allocated we would have to *change* the typing for i, rather than *extend* the address typing.

(BIND). If e_1 is a value v_1, then the evaluation must end with

$$\frac{\mu \vdash v_1 \Rightarrow v_1, \mu \quad \mu \vdash [v_1/x]e_2 \Rightarrow v_2, \mu'}{\mu \vdash \textbf{let } x = v_1 \textbf{ in } e_2 \Rightarrow v_2, \mu'}$$

while the typing must end with (LET-VAL):

$$\frac{\lambda \vdash v_1 : \tau_1 \quad \lambda; [x : \mathit{Close}_\lambda(\tau_1)] \vdash e_2 : \tau_2}{\lambda \vdash \textbf{let } x = v_1 \textbf{ in } e_2 : \tau_2}$$

and $\mu : \lambda$. By Lemma 5, $\lambda \vdash v_1 : \mathit{Close}_\lambda(\tau_1)$, and so by Lemma 4, $\lambda \vdash [v_1/x]e_2 : \tau_2$. So by induction, there exists λ' such that $\lambda \subseteq \lambda'$, $\mu' : \lambda'$, and $\lambda' \vdash v_2 : \tau_2$.

The case when e_1 is not a value is similar, but Lemma 5 is not required, and induction is used twice. □

The subject reduction property does not by itself ensure that a type system is sensible. For example, a type system that assigns *every* type to *every* expression trivially satisfies the subject reduction property, even though such a type system is useless. The main limitation of subject reduction is that it only applies to well-typed expressions that evaluate successfully. Really we would like to be able say something about what happens when we *attempt* to evaluate an arbitrary well-typed expression.

One approach to strengthening subject reduction (used by Gunter [Gun92], for example) is to augment the evaluation rules with rules specifying that certain expressions evaluate to a special value, **TypeError**, which has no type. For example, an attempt to dereference a value other than a pointer would evaluate to **TypeError**. Then, by showing that subject reduction holds for the augmented evaluation rules, we get that a well-typed expression cannot evaluate to **TypeError**. Hence any of the errors that lead to **TypeError** cannot occur in the evaluation of a well-typed expression. Aside from the drawback of requiring us to augment the evaluation rules, this approach does not give us as much information as we would like. It tells us that certain bad things will not happen during the evaluation of well-typed expression, but says nothing about what *other* bad things can happen.

We now present a different approach leading to a type soundness theorem that characterizes precisely everything that may go wrong when we attempt to evaluate a well-typed expression. First, we note that a successful evaluation always produces a *value*:

Lemma 7. *If $\mu \vdash e \Rightarrow v, \mu'$, then v is a value and μ' is a memory.*

Roughly speaking, the combination of the subject reduction theorem and the correct forms lemma (Lemma 1) allows us to characterize the forms of expressions that will be encountered during the evaluation of a well-typed expression. This will allow us to characterize what can go wrong during the evaluation.

To get a handle on the "progress" of an attempted evaluation, it is helpful to recast the evaluation rules as a recursive evaluation function, *eval*. For example, the (UPDATE) rules correspond to the clauses

$$
\begin{array}{l}
\mathit{eval}(\mu, (a,1) := e) = \\
\quad \textbf{let } (v, \mu') = \mathit{eval}(\mu, e) \textbf{ in} \\
\quad\quad \textbf{if } a \in \mathit{dom}(\mu') \textbf{ and } \mu'(a) \neq \textbf{dead then} \\
\quad\quad\quad (v, \mu'[a := v]) \\
\quad\quad \textbf{else} \\
\quad\quad\quad \mathit{fail};
\end{array}
$$

$$
\begin{array}{l}
\mathit{eval}(\mu, *e_1 := e_2) = \\
\quad \textbf{let } ((a,0), \mu_1) = \mathit{eval}(\mu, e_1) \textbf{ in} \\
\quad \textbf{let } (v, \mu_2) = \mathit{eval}(\mu_1, e_2) \textbf{ in} \\
\quad\quad \textbf{if } a \in \mathit{dom}(\mu_2) \textbf{ and } \mu_2(a) \neq \textbf{dead then} \\
\quad\quad\quad (v, \mu_2[a := v]) \\
\quad\quad \textbf{else} \\
\quad\quad\quad \mathit{fail};
\end{array}
$$

Introducing *eval* allows us to talk about type soundness in terms of what happens when *eval* is called on a well-typed program.

Definition 8. A call $\mathit{eval}(\mu, e)$ is *well typed* iff there exist λ and τ such that $\mu : \lambda$ and $\lambda \vdash e : \tau$.

Definition 9. An activation of *eval aborts directly* if the activation *itself* aborts. Note that an activation does not abort directly if it makes a recursive call that aborts or does not terminate.

We can now show the key result for type soundness:

Theorem 10. *Suppose that an activation $\mathit{eval}(\mu, e)$ is well typed. Then every recursive call made by the activation is well typed. Furthermore, if the activation aborts directly, it aborts due to one of the following errors:*

E1. An attempt to read or write to a dead address (i, j).
E2. An attempt to read or write to a nonexistent address (i, j). Address $(i, 0)$ always will exist, so the problem is that the offset j is invalid.
E3. An attempt to read an uninitialized address (i, j).
E4. An attempt to declare an array of size less than or equal to 0.

Proof. We just consider all possible forms of expression e. Here we just give the case $e_1 := e_2$; the other cases are quite similar.

If $eval(\mu, e_1 := e_2)$ is well typed, then there exist λ and τ such that $\mu : \lambda$ and $\lambda \vdash e_1 := e_2 : \tau$. The latter typing must be by (ASSIGN):

$$\frac{\lambda \vdash e_1 : \tau\ var \qquad \lambda \vdash e_2 : \tau}{\lambda \vdash e_1 := e_2 : \tau}$$

By Lemma 1, e_1 must be of the form $((i,j),1)$ or else of the form $*e_1'$. So, simplifying notation a bit, we are left with two cases: $(a,1) := e$ and $*e_1 := e_2$. Note that there is a clause of *eval* that applies to each of these. We consider the two cases in turn.

If the activation is $eval(\mu, (a,1) := e)$, where $\mu : \lambda$ and $\lambda \vdash (a,1) := e : \tau$, then the typing must end with (ASSIGN):

$$\frac{\lambda \vdash (a,1) : \tau\ var \qquad \lambda \vdash e : \tau}{\lambda \vdash (a,1) := e : \tau}$$

So by (VAR), $\lambda(i) = \tau$, where $a = (i,j)$.

Also, the recursive call $eval(\mu, e)$ is well typed. If this call fails to return, then the parent activation $eval(\mu, (a,1) := e)$ doesn't abort directly. If the call succeeds, then by Lemma 7 it returns a value v and a memory μ', so the pattern-match '**let** $(v, \mu') = eval(\mu, e)$' doesn't abort.

By the subject reduction theorem, there exists λ' such that $\lambda \subseteq \lambda'$, $\mu' : \lambda'$, and $\lambda' \vdash v : \tau$. Hence $\lambda'(i) = \tau$, and so $(i,0) \in dom(\mu')$.

So the only way for the activation $eval(\mu, (a,1) := e)$ to abort directly is if $(i,j) \notin dom(\mu')$ or $\mu'((i,j)) = \mathbf{dead}$. And since $(i,0) \in dom(\mu')$, we know that if the first case holds, the error is in the offset j.

If the activation is $eval(\mu, *e_1 := e_2)$, where $\mu : \lambda$ and $\lambda \vdash *e_1 := e_2 : \tau$, then the typing must end with (L-VAL) followed by (ASSIGN):

$$\frac{\dfrac{\lambda \vdash e_1 : \tau\ ptr}{\lambda \vdash *e_1 : \tau\ var} \qquad \lambda \vdash e_2 : \tau}{\lambda \vdash *e_1 := e_2 : \tau}$$

So the recursive call $eval(\mu, e_1)$ is well typed. If this call fails to return, then the parent activation $eval(\mu, *e_1 := e_2)$ doesn't abort directly. If the call succeeds, then by Lemma 7 it returns a value v_1 and a memory μ_1.

By the subject reduction theorem, there exists λ_1 such that $\lambda \subseteq \lambda_1$, $\mu_1 : \lambda_1$, and $\lambda_1 \vdash v_1 : \tau\ ptr$. So by the Correct Form lemma, v_1 is of the form $((i,j),0)$ hence the pattern-match '**let** $((a,0),\mu_1) = eval(\mu, e_1)$' doesn't abort. Also, by (PTR), $\lambda_1(i) = \tau$.

By the Superfluousness Lemma, $\lambda_1 \vdash e_2 : \tau$, so the recursive call $eval(\mu_1, e_2)$ is also well typed. If this call fails to return, then the parent activation doesn't

get stuck. If it succeeds, then it returns a value v and a memory μ_2, so the pattern-match '**let** $(v, \mu_2) = eval(\mu_1, e_2)$' doesn't abort. By the subject reduction theorem, there exists λ' such that $\lambda_1 \subseteq \lambda'$, $\mu_2 : \lambda'$, and $\lambda' \vdash v : \tau$. Hence $\lambda'(i) = \tau$, and so $(i, 0) \in dom(\mu_2)$.

So the only way for the activation $eval(\mu, *e_1 := e_2)$ to abort directly is if $(i, j) \notin dom(\mu_2)$ or $\mu_2((i, j)) = \mathbf{dead}$. And since $(i, 0) \in dom(\mu_2)$, we know that if the first case holds, the error is in the offset j. □

Corollary 11 (Type Soundness). *If $\lambda \vdash e : \tau$ and $\mu : \lambda$, then $eval(\mu, e)$ either*

1. *succeeds (producing a value of type τ), or*
2. *fails to halt, or*
3. *aborts due to one of the errors E1, E2, E3, or E4.*

Proof. Any call must either succeed, fail to halt, or abort.

If the call aborts, then one of its recursive activations must abort directly. Now this activation must have been reached by a finite path of recursive calls from the root call $eval(\mu, e)$. Since the root call is well typed, by Theorem 10 all the calls on the path are well typed. So the activation that aborts directly is well typed. Hence by Theorem 10 it aborts due to one of the errors *E1–E4*. □

5 Discussion

The semantics specifies that an implementation is under no obligation to preserve the contents of variables beyond their scope, which in turn *justifies* a stack-based implementation. Further, there is no need for static links since all functions in Polymorphic C are closed with respect to top-level declarations. It is also interesting to note that in light of this closure property, there would be no need to specify in the semantics that a variable dies at the end of its scope if there were no & operator. The variable would simply be unreachable in this case.

To maintain subject reduction, the semantics also ensures that any program with pointer errors does not produce a value. This requires a number of mechanisms, for example, keeping track of cells that have been deallocated, that we do not expect to see in any realistic implementation of the semantics. We believe that an implementation, for the sake of efficiency, should be able to do whatever it likes on programs that do not yield values, and hence are in error, according to the semantics. For example, the semantics does not prescribe a value for dereferencing a dangling pointer. So it would be acceptable, upon an attempt to dereference such a pointer, for an implementation to merely return the last value stored there, as in C, rather than detect an error.

Given that a real implementation would not catch pointer errors, what then is the practical significance of our type soundness theorem? Two things can be said. First, the theorem gives a characterization of the *source* of errors—it tells us that when a program crashes with a "Segmentation fault—core dumped" message, what causes the crash is one of the errors *E1–E4* and not, for example, an invalid polymorphic generalization. Second, by directly implementing our semantics, one can get a robust "debugging" implementation that flags all pointer errors.

6 Conclusion

Advanced polymorphic type systems have come to play a central role in the world of functional programming, but so far have had little impact on traditional imperative programming. We assert that an ML-style polymorphic type system can be applied fruitfully to a "real-world" language like C, bringing to it both the expressiveness of polymorphism as well as a rigorous characterization of the behavior of well-typed programs.

Future work on Polymorphic C includes the development of a type inference algorithm (preliminary work indicates that this can be done straightforwardly), the development of an efficient implementation (perhaps using the work of [Le92, ShA95, HaM95]), and extending the language to include other features of C, especially structures.

References

[DaM82] Damas, L. and Milner, R., Principal Type Schemes for Functional Programs, *Proc. 9th ACM Symposium on Principles of Programming Languages*, pp. 207–212, 1982.

[GMW79] Gordon, M., Milner, R. and Wadsworth, C., Edinburgh LCF, *Lecture Notes in Computer Science* **78**, Springer-Verlag, 1979.

[Gun92] Gunter, C., *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992.

[Har94] Harper, R., A Simplified Account of Polymorphic References, *Information Processing Letters*, 51, pp. 201–206, August 1994.

[HaM95] Harper, R. and Morrisett, G., Compiling Polymorphism Using Intensional Type Analysis, *Proc. 22nd ACM Symposium on Principles of Programming Languages*, pp. 130–141, 1995.

[KR78] Kernighan, B. and Ritchie, D., *The C Programming Language*, Prentice-Hall, 1978.

[LeW91] Leroy, X. and Weis, P., Polymorphic Type Inference and Assignment, *Proc. 18th ACM Symposium on Principles of Programming Languages*, pp. 291–302, 1991.

[Le92] Leroy, X., Unboxed Objects and Polymorphic Typing, *Proc. 19th ACM Symposium on Principles of Programming Languages*, pp. 177–188, 1992.

[ShA95] Shao, Z. and Appel, A., A Typed-Based Compiler for Standard ML, *Proc. 1995 Conf. on Programming Language Design and Implementation*, pp. 116–129, 1995.

[SML93] Standard ML of New Jersey, Version 0.93, February 15, 1993.

[Tof90] Tofte, M., Type Inference for Polymorphic References, *Information and Computation*, 89, pp. 1–34, 1990.

[VoS95] Volpano, D. and Smith, G., A Type Soundness Proof for Variables in LCF ML, *Information Processing Letters*, 56, pp. 141–146, November 1995.

[Wri95] Wright, A., Simple Imperative Polymorphism, *Lisp and Symbolic Computation* 8, 4 pp. 343–356, December 1995.