A Simple Interpretation of OOP in a Language with State

Jonathan Eifrig* Scott Smith* Valery Trifonov* Amy Zwarico

Department of Computer Science
The Johns Hopkins University †

May 22, 1993

Abstract

Giving a complete semantics to strongly typed object-oriented programming is a well-known research problem. Recent work has made significant strides toward solving this problem. However, in most of this work a purely functional, call-by-name view of objects is taken. In this paper we give meaning to a call-by-value, typed object language with updatable instance variables, and prove the type system given is sound; *i.e.*, well-typed programs do not experience "message not understood" errors.

The semantics is given by a translation of the object-based language into a state-based language with a simple type discipline and with a new notion of a once-assignable cell.

1 Introduction

Giving full and faithful meaning to typed object-oriented programming languages is a well-known research problem. There has been considerable recent activity in defining powerful type systems for OOP [CHC90, CCH+89, BM92, Bru93, Mit90, PT93]. These approaches have a natural model in $F^{\omega}_{\leq}$ and admit a very powerful type theory utilizing F-bounded quantification. One major shortcoming of all these approaches is they take a functional view of objects. Thus, the internal state of an object must be explicitly threaded through the control structure of the program. This threading however damages the very property that object-based programming techniques were designed to address: the localized partitioning of a global state.

What we accomplish here is an interpretation of objects in a PCF-like language extended to have state operations. Our objects are thus objects that much more closely correspond to objects in a real implementation, with real instance variables that may be updated in place.

We begin by defining a representative state-based typed object-oriented language LOOP, and show how the semantics of this language may be defined by translating it into a simple imperative language, SOOP. SOOP is roughly the call-by-value lambda calculus extended with records and references, and an additional a new construct, the *single-assignment reference*, for interpreting self-referential objects. This new construct is not strictly necessary, but we argue why it is more useful to have it than to suffer the consequences of its absence.

The translation of expressions is similar to other interpretations of records as objects developed for untyped OOP [CP89, Red88]. The main difference is use of the single assignment reference in

*Partially supported by NSF grant CCR-9109070

†Authors' email addresses: eifrig@cs.jhu.edu, scott@cs.jhu.edu, trifonov@cs.jhu.edu, amy@cs.jhu.edu

place of the Y-combinator to avoid copying of code. In a functional world the two are equivalent, but not in the presence of state.

What we do not accomplish here is the incorporation of the powerful type language of F-bounded polymorphism into SOOP. In order to make the state problem tractable, we have elected to use a simpler typing scheme based on an approach requiring inheritance to conform with subtyping. Subtyping in SOOP is the standard record subsumption model following Cardelli [Car84]. Since many real-world OOP languages enforce this criterion, it is not wholly unreasonable, and may ultimately prove the most tractable approach if the type-checking and type-inference problems of F-bounded polymorphism prove intractable [Pie92].

We define a translation of any typed LOOP program into a typed SOOP program, and together with a proof of the type soundness of SOOP this gives us type soundness and lack of run-time type errors for LOOP. The result is a state-based object-oriented language with a formally defined semantics and provably sound type theory. Also, the tools we use are simple operational interpreters and type systems, so this can also serve as a complete, comprehensible specification of the language. The same can not be said for F-bounded polymorphism. One measure of the complexity of that type system is that it cannot be given meaning in second-order logic, whereas the language we study has a semantics expressible in Peano Arithmetic.

Some other type-safe systems have been designed that are not based on F-bounded polymorphism [GJ90, SCB+86], but these require method bodies to be re-type checked when subclasses are defined. In a language supporting first-class class definitions, in which classes may be extended dynamically, this re-checking needs to be performed during program execution and type-checking of programs becomes essentially a run-time task.

Section 2 presents the syntax and type system of LOOP along with an informal discussion of its semantics. Sections 3 and 4 introduce the SOOP language and type system, its formal operational semantics, and proofs of type soundness. Section 5 contains the translation of LOOP into SOOP, thus giving a rigorous semantics to LOOP programs. A type soundness result for the LOOP type system is presented in Section 5.

2 The LOOP Language

We now define LOOP (Little Object-Oriented Programming language). LOOP is a strongly-typed extension of call-by-value PCF with just enough additional constructs to capture the spirit of most Smalltalk-derived languages.

Identifiers of LOOP are divided into four sorts: $v \in K_{\text{par}}$ are function parameters, $s \in K_{\text{class}}$ are bound class names, $m \in K_{\text{meth}}$ are method names, and $x \in K_{\text{inst}}$ are instance variables. These sorts are present for convenience only; the ability to syntactically distinguish identifiers will enable the translation from LOOP to SOOP to be purely syntactic in nature. The LOOP expressions are the least set

$$\begin{array}{rcll} e & \in & \mathcal{E} & ::= \quad v \mid 0 \mid 1 \mid \ldots \mid \texttt{true} \mid \texttt{false} \mid \texttt{pred}(e) \mid \texttt{succ}(e) \mid \texttt{is\_zero}(e) \\ & & & \quad \mid \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{fn}\ (v)\ \texttt{=>}\ e \mid e_0(e_1) \mid \texttt{empty} \\ & & & \quad \mid \texttt{class}\ s\ \texttt{is}\ \ e : \tau\ \texttt{with}\ \{\{x_1 = e_1, \ldots, x_n = e_n\}, \{m_1 = e'_1, \ldots, m_{n'} = e'_{n'}\}\} \\ & & & \quad \mid \texttt{new}\ e \mid s \mid e.x \mid e.x\ \texttt{:=}\ e \mid e\ \texttt{<-}\ m \mid e.m \end{array}$$

where τ is a type, defined below.*

Many of the LOOP expressions are familiar PCF terms; the remainder serve to represent classes and objects. The constant `empty` denotes the class with no instance variables or methods; other

*Explicit typing of class expressions enables our translation to be completely syntactic.

classes are created by extending an existing class with additional instance variables and methods. The expression

$$\texttt{class}\ s\ \texttt{is}\ e : \tau\ \texttt{with}\ \{\{x_1 = e_1, \ldots, x_n = e_n\}, \{m_1 = e'_1, \ldots, m_{n'} = e'_{n'}\}\}$$

extends the class e by adding the new instance variables and methods indicated by the x_i and m_j ($1 \leq i \leq n, 1 \leq j \leq n'$) respectively. With each new instance variable x_i is associated an expression e_i, defining the initial value of the instance variable upon object creation. In a strongly typed framework the Smalltalk paradigm of not giving variables initial values makes no sense because it admits the possibility that an instance variable may be used when it is undefined. Within the bodies of the method definitions and instance variable initializations, the name s is bound and plays the role of `self`: $s.x$ denotes the *value* of the instance variable x, $s.x$ `:=` e updates the value of the instance variable with the value of the expression e, and $s.m$ sends the message m to the self. In LOOP, classes are first-class terms. `new` e creates an object of the class denoted by e and returns that object as its value, while e `<-`m is the familiar "send message" construct. Note that sending messages to self inside a class definition, $s.m$, and sending a message to an object, e `<-`m, have different notation in order to make the translation into SOOP purely syntactic.

Recursive functions are definable in LOOP via the implicit recursion present in class definitions: each method defined in a class can be used in the body of any of the method definitions. Thus, all terms definable in PCF are definable in LOOP. With this in mind, we will treat the usual arithmetic functions on integers as macros of LOOP.

A formal definition of the meaning of a LOOP program is given in Section 5, in which we *define* a translation of LOOP programs into another language, SOOP. Class-based languages (of which LOOP is an example) have no syntax for objects themselves; thus, programs like `new empty` compute to a value which is inexpressible within the language itself.

2.1 The Loop Type System

The syntax of LOOP types is as follows.

$$\begin{array}{rcrcl} M & \in & \mathcal{M} & ::= & \{m_1 : \tau_1, \ldots, m_k : \tau_k\} \\ I & \in & \mathcal{I} & ::= & \{x_1 : \tau_1, \ldots, x_k : \tau_k\} \\ C & \in & \mathcal{C} & ::= & (I, M) \\ \tau & \in & \mathcal{T} & ::= & \texttt{Bool} \mid \texttt{Num} \mid \tau \rightarrow \sigma \mid \texttt{TObj}\{M\} \mid \texttt{TOpenObj}\{C\} \mid \texttt{TClass}\{C\} \end{array}$$

Terms denoting objects have types of the form `TObj`$\{M\}$ or `TOpenObj`$\{C\}$, depending on whether the object is being viewed from the "outside" or the "inside." Objects viewed from the outside (e.g., expressions of the form `new` e) have type `TObj`$\{M\}$ (where M is a *method context*, a mapping from method names to types); these terms respond to the messages listed in M, while the instance variables remain hidden. Object terms viewed from the inside (i.e., the name s bound within a `class` expression) have type `TOpenObj`$\{(I, M)\}$ (where I is an *instance context*, analogous to the method context above); such terms have both their instance variables and their method names visible. Terms of type `TClass`$\{(I, M)\}$ represent classes. For $e \in$ `TClass`$\{(I, M)\}$, `new` e creates an object of type `TObj`$\{M\}$, and `class` s `is` $e : \tau$ `with` $\{\{x_1 = e_1, \ldots, x_n = e_n\}, \{m_1 = e'_1, \ldots, m_{n'} = e'_{n'}\}\}$ extends e to a subclass with added and overridden methods and instance variables.

LOOP types and contexts are partially ordered by the subtyping relation $\leq$, axiomatized by the rules in Figure 1. PCF types are ordered in the usual way [Car84]. Two instance contexts I_1 and I_2 are ordered, $I_1 \leq I_2$, only if every instance variable name occurring in I_2 occurs in I_1 with the same type, although I_1 may map additional names as well. Method contexts have a slightly

$$(TRefl)\quad \frac{}{\vdash \tau \leq \tau} \qquad (TTrans)\quad \frac{\vdash \rho \leq \sigma \quad \vdash \sigma \leq \tau}{\vdash \rho \leq \tau} \qquad (TFunc)\quad \frac{\vdash \tau' \leq \tau \quad \vdash \sigma \leq \sigma'}{\vdash \tau \to \sigma \leq \tau' \to \sigma'}$$

$$(TInst)\quad \frac{n' \leq n \quad \tau_i = \sigma_i \ \ (\text{for each } 1 \leq i \leq n')}{\vdash \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \leq \{x_1 : \sigma_1, \ldots, x_{n'} : \sigma_{n'}\}}$$

$$(TMeth)\quad \frac{n' \leq n \quad \vdash \tau_i \leq \sigma_i \ \ (\text{for each } 1 \leq i \leq n')}{\vdash \{m_1 : \tau_1, \ldots, m_n : \tau_n\} \leq \{m_1 : \sigma_1, \ldots, m_{n'} : \sigma_{n'}\}}$$

$$(TClass)\quad \frac{\vdash I \leq I' \quad \vdash M \leq M'}{\vdash (I, M) \leq (I', M')} \qquad (TObj)\quad \frac{\vdash M \leq M'}{\vdash \texttt{TObj}\{M\} \leq \texttt{TObj}\{M'\}}$$

$$(TOpenObj)\quad \frac{\vdash C \leq C'}{\vdash \texttt{TOpenObj}\{C\} \leq \texttt{TOpenObj}\{C'\}}$$

Figure 1: LOOP Subtyping Rules

stronger ordering since methods are not mutable: two method contexts M_1 and M_2 are ordered $M_1 \leq M_2$ if an only if $M_1(m) \leq M_2(m)$, for each method name m mapped in M_2. Two object types are ordered if their corresponding method contexts are ordered; there is no non-trivial ordering on class types.

In practice, these orderings restrict the ways in which classes can be extended. A class can be extended by the addition of new instance variables and methods, but overriding an existing method can only be done with a new method body whose type is a subtype of the one overridden. This restriction, critiqued in [CHC90], is necessary in our simple interpretation; if methods could be overridden with new methods of arbitrary types, the correctness of the other methods of the class (which may make use of the now-overridden method name) could not be guaranteed. The use of a semantics based on F-bounded polymorphism would allow for a more generous solution, but that is beyond the scope of this work.

The typing rules for LOOP terms are listed in Figure 2. Many of the rules are standard PCF rules. Rules (*Inst*), (*Assn*), and (*Self*) allow for instance variable access, update, and method selection of an object based on its "internal" view. Rule (*Mesg*) allows for method selection of an object based on its "external" view. Rule (*Empty*) types the `empty` expression, and rule (*New*) types object creation; note how instance variable types are hidden at this point.

The most interesting rule is for forming new class expressions, (*Class*). The class name s in the `class` expression is bound within the bodies of method definitions and instance variable initializations, and plays the role of "`self`". In this rule $\oplus$ is the concatenation operation on classes. It makes explicit the restriction on class extension to subtypes.

DEFINITION 2.1 $(I, M) \oplus \{\overline{x_i : \tau_i}, \overline{m_i : \sigma_i}\}$ denotes the class context $C' = (I', M')$ such that $I'(x) = \tau_i$ if $x = x_i$ for some i and $I'(x) = I(x)$ otherwise, and $M'(x) = \sigma_i$ if $m = m_i$ for some i and $M'(x) = M(x)$ otherwise. Furthermore, the above is considered well-formed only when for each i, if $x_i \in dom(C)$ then $\tau_i = C(x_i)$, and if $m_i \in dom(C)$, $\sigma_i \leq C(m_i)$.

The type system given here for the language is by no means the strongest one possible; in fact,

$$(Sub)\quad \frac{\vdash \tau' \leq \tau \quad E \vdash e : \tau'}{E \vdash e : \tau} \qquad (Var)\quad \frac{E(v) = \tau}{E \vdash v : \tau}$$

$$(Num)\quad \frac{n \text{ is a numeral}}{E \vdash n : \texttt{Num}} \qquad (Pred)\quad \frac{E \vdash e : \texttt{Num}}{E \vdash \texttt{pred}(e) : \texttt{Num}}$$

$$(Succ)\quad \frac{E \vdash e : \texttt{Num}}{E \vdash \texttt{succ}(e) : \texttt{Num}} \qquad (IsZero)\quad \frac{E \vdash e : \texttt{Num}}{E \vdash \texttt{is\_zero}(e) : \texttt{Bool}}$$

$$(True)\quad \frac{}{E \vdash \texttt{true} : \texttt{Bool}} \qquad (False)\quad \frac{}{E \vdash \texttt{false} : \texttt{Bool}}$$

$$(Cond)\quad \frac{E \vdash e_1 : \texttt{Bool} \quad E \vdash e_2 : \tau \quad E \vdash e_3 : \tau}{E \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau} \qquad (Abs)\quad \frac{E, v : \tau \vdash e : \sigma}{E \vdash \texttt{fn}\,(v)\ \texttt{=>}\ e_1 : \tau \rightarrow \sigma}$$

$$(App)\quad \frac{E \vdash e_1 : \tau \rightarrow \sigma \quad E \vdash e_2 : \tau}{E \vdash e_1(e_2) : \sigma} \qquad (Mesg)\quad \frac{E \vdash e : \texttt{TObj}\{M\} \quad M(m) = \tau}{E \vdash e\texttt{<-}m : \tau}$$

$$(Inst)\quad \frac{E \vdash e : \texttt{TOpenObj}\{I, M\} \quad I(x) = \tau}{E \vdash e.x : \tau} \qquad (Assn)\quad \frac{E \vdash e_1 : \texttt{TOpenObj}\{I, M\} \quad I(x) = \tau \quad E \vdash e_2 : \tau}{E \vdash e_1.x\texttt{:=}e_2 : \tau}$$

$$(Self)\quad \frac{E(e) = \texttt{TOpenObj}\{I, M\} \quad M(m) = \tau}{E \vdash e.m : \tau} \qquad (Empty)\quad \frac{}{E \vdash \texttt{empty} : \texttt{TClass}\{\{\}, \{\}\}}$$

$$(New)\quad \frac{E \vdash e : \texttt{TClass}\{I, M\}}{E \vdash \texttt{new}\ e : \texttt{TObj}\{M\}} \qquad (Mesg)\quad \frac{E \vdash e : \texttt{TObj}\{M\} \quad M(m) = \tau}{E \vdash e\texttt{<-}m : \tau}$$

$$(Class)\quad \frac{\begin{array}{l}\tau = \texttt{TClass}\{C_0\} \\ E \vdash e_0 : \texttt{TClass}\{C_0\} \\ E, s : \texttt{TOpenObj}\{C_0 \oplus \{\overline{x : \tau}, \overline{m : \sigma}\}\} \vdash e_k : \tau_k, \text{ (for each } e_k \in \overline{e}) \\ E, s : \texttt{TOpenObj}\{C_0 \oplus \{\overline{x : \tau}, \overline{m : \sigma}\}\} \vdash f_k : \sigma_k, \text{ (for each } f_k \in \overline{f})\end{array}}{E \vdash \texttt{class}\ s\ \texttt{is}\ e_0 : \tau\ \texttt{with}\ \{\overline{x = e}, \overline{m = f}\} : \texttt{TClass}\{C_0 \oplus \{\overline{x : \tau}, \overline{m : \sigma}\}\}}$$

Figure 2: LOOP Typing Rules

the notable lack of recursive types severely restricts the expressibility of the language. The system is merely illustrative of the kinds of type theories possible for the language, and the ways in which these type theories can be shown sound via our operational translation of the LOOP language.

We will hereafter use the syntactic abbreviations `let` $n = e_1$ `in` e_2 to mean (`fn` (n) `=>` $e_2)(e_1)$, e_1; e_2 to mean (`fn` (a) `=>` (`fn` (b) `=>` $b)$)$(e_1)(e_2)$, the type `unit` to mean `TObj{}`, and `()` to be an element of this type, namely `new empty`.

2.2 Programming in LOOP

Although LOOP does not have an explicit recursion operator or updatable reference cells, both of these constructs are definable within LOOP.

A simple LOOP program is the following:

```
let Point1 = class self is empty:TClass{{},{}} with {
   {x = 0},
   {getx = fn a => self.x,
    setx = fn n => self.x := n; (),
    align = fn p => p<-setx (self.getx ())}}
in let Point2 = class self is Point1:τ1 with {{y=0},
      {gety = fn a => self.y,
       sety = fn n => self.y := n; (),
       align2 = fn p => self.align p; p<-sety (self.gety ())}}
in let p1 = new Point2 in let p2 = new Point2 in p1<-align2 p2
```

where the types of `Point1` and `Point2`, τ_1 and τ_2, are

```
τ1 = TClass{{x:Num},{getx:unit -> Num, setx:Num -> unit,
     align:TObj{setx:Num -> unit} -> unit}}
τ2 = TClass{{x:Num,y:Num},{getx:unit -> Num, setx:Num -> unit,
     gety:unit -> Num, sety:Num -> unit, align:TObj{setx:Num -> unit} -> unit,
     align2:TObj{sety:Num -> unit, align:TObj{setx:Num -> unit} -> unit}} -> unit}}
```

`Point1` is a class of simple one-dimensional points that can move about and in addition can respond to an `align` message that moves a second point to its position. Class `Point2` further refines this idea by adding a second coordinate and a new alignment method; note that this new method uses one inherited from the parent class `Point1`. Observe that the `align` and `align2` methods may be typed in our language even though they may take objects of their own class as arguments.

Simulating this program in a functional interpretation of OOP such as [Bru93, PT93] requires an explicit threading of state in some fashion. In particular, `align2` modifies the state of its argument object while executing. The simple encodings of object methods as state transformers, taking the state of their object as an additional parameter and returning their modified state as an additional result, are insufficient here. To properly account for this the states of *every* object in the system would need to be threaded through the flow of control, since it will not in general be known what object is being modified by the `align` method.

3 The SOOP Language

We define the meaning of LOOP programs in terms of a lower-level "implementation" language SOOP (Semantics for Object-Oriented Programming), a call-by-value language which offers simple operations on records and reference cells in addition to most of the standard PCF constructs. The only significant departure from standard notions is the single-assignment reference (SAR) cell, which allows for a natural and type-sound interpretation of objects in the presence of effects.

In standard encodings of object-oriented languages [CP89, Red88] an object is formed by taking a fixed point of a function that represents its class, referred to below as the "class function." This results in a recursive record, and the methods of the object—fields of the record—gain access to its "self" by unrolling the fixed point expression. Consider as an informal illustration the case of an object `o` of class `c` with method `m` whose definition refers to the value of the object:

```
c = class ...
  methods {m = ...self...}
o = new c
```

translates to

$$c = \lambda self.\, \{m = \ldots self \ldots\}$$
$$o = Y(c)$$

where Y is the fixed point combinator. In purely functional languages this encoding produces perfectly adequate results, but the situation changes with the introduction of effects and the natural for them (though not intrinsic [CF91]) call-by-value semantics. As pointed out in [Wan89, CHC90, PT93] the fixed point combinator is then only well-defined on functionals, and classes do not correspond to functionals.

One possible solution is to "freeze" the access of an object to its "self," e.g.

$$c = \lambda self.\, \lambda().\, \{m = \ldots self() \ldots\}$$
$$o = Y(c)()$$

but it is not completely satisfactory because a desirable feature of a state-based object-oriented language is the support of mutable instance variables, private to each object. It is also desirable to allow the initial value of each instance variable or method to include initialization code for allocation of mutable cells. However, creating cells in the record fields above would cause new cells to be allocated at each unrolling of the fixed point, i.e. at each access of `m` to "self."

Two approaches to solving this problem are discussed in [Hen91]; the one for which the author gives a denotational semantics is based on the interpretation of the fixed points of class functions as state transformers. This bears some similarity to the above translation, but the state is represented as an explicit parameter of certain semantic functions, which allows the reallocation to be avoided by reusing the same state for each unrolling of the fixed point. The presence of imperative constructs in our target language renders this method inapplicable to SOOP.

Alternatively the cells may be created before the fixed point is taken, but inside another abstraction of the class function, since otherwise they will be shared by all instances of the class. In our informal notation an object of class `c` with instance variable `x` is then

```
c = class ...
  variables {x = ...}
  methods {m = ...self...}
o = new c
```

yielding

$$c = \lambda().\, \mathsf{let}\ x = \mathsf{ref}\ \ldots\ \mathsf{in}$$
$$\qquad \lambda self.\, \lambda().\, \{x = x,\ m = \ldots self() \ldots\}$$
$$o = Y(c())()$$

However we can only do this under the otherwise unjustified restriction that the initial values of the instance variables may not refer to *self*, even if they are not strict in it. It is possible to overcome

this limitation by initializing all private variables to some default values of their respective types, and having a special method automatically invoked after the fixed point is computed to assign the required values to them, but this would further complicate the semantics of class extension and inheritance.

Our solution is to define a fixed point combinator in terms of mutable cells, in the spirit of Landin's suggestion [Lan64]. Operationally it is equivalent to

$$Y_r = \lambda f.\, \mathsf{let}\ r = \mathsf{ref}\ \mathsf{null}\ \mathsf{in}\ (\mathsf{set}\ (r,\ f(r));\ !\ r)$$

where null is a "dummy" initial value, and set $(r,\ v)$ and $!\ r$ are the assignment to and dereferencing of the mutable cell r. The argument f now must operate on a reference cell which is to be assigned the value of the fixed point. This combinator evaluates f only once, and therefore it can be applied to class functions with side effects, in particular allocating new mutable cells, to obtain objects with the intuitively expected behavior.

The use of an ordinary mutable cell r in the definition of Y_r however makes impossible the type-sound class extensions: the class function of the extended class must first pass its argument (now a pointer to the future object) to the class function of the superclass, and then modify or add new fields to the resulting record. But since a class function is now of type τ Ref -> τ, where τ is the type of the object, it is a type error to apply it to an argument of type τ' Ref, where τ' is the type of the objects of the extended class, even though τ' is a subtype of τ (i.e. any value of type τ' is also a value of type τ), because τ' Ref cannot be a subtype of τ Ref in a sound typing system.

This is the rationale for the introduction of the SAR cells. Intuitively the SAR x is both allocated and assigned to via the new construct

$$\mathsf{let}\ x\ \mathsf{be\ SAR\ to}\ e.$$

The cell is allocated before but assigned the value of e after e has been evaluated, and the whole construct then returns the same value. No other assignment to x is possible and this makes a (covariant) subtyping rule for SAR types sound. The value pointed to by x is accessible from within e via dereferencing (for SARs), e.g. $\uparrow x$; this returns the value of e itself once the assignment has been performed. Thus the evaluation of let x be SAR to e is similar to taking the fixed point of $\lambda x.\, e$ (if we define $\uparrow x$ as x). The important difference is that the body e is not reevaluated each time $\uparrow x$ is computed, resolving the problem of allocating cells for instance variables encountered in the fixed point translation.

During the evaluation of e the result of dereferencing x is "undefined," and the special value null is introduced in the operational semantics for this case. In effect the value of the expression let x be SAR to e is undefined if e is strict in x, which in terms of LOOP happens exactly when an object refers to its own value during the initialization of its instance variables and methods. Note that the fixed point translation of such an object results in an expression whose evaluation in a call-by-value language leads to infinite recursion; hence a LOOP program translated to SOOP yields null only if its translation under the fixed point semantics fails to terminate. This property of the SOOP operational interpreter is similar to the detection of "black holes" by some interpreters for lazy languages [Wad92].

3.1 SOOP Syntax

Let x range over the countable set of variables *Var* and l range over the countable set of labels *Lab*. The values $v \in Val$ and expressions $e \in Exp$ of SOOP are the least inductive collections defined as

$$\text{(SubRecord)}\quad \frac{\tau_i' \le \tau_i \text{ for all } i \in \{1, \ldots, m\} \qquad m \le m'}{\{\!|l_1 : \tau_1', \ldots, l_{m'} : \tau_{m'}'|\!\} \le \{\!|l_1 : \tau_1, \ldots, l_m : \tau_m|\!\}}$$

$$\text{(SubFun)}\quad \frac{\tau_1 \le \tau_1' \qquad \tau_2' \le \tau_2}{\tau_1' \text{ -> } \tau_2' \le \tau_1 \text{ -> } \tau_2}$$

$$\text{(SubSAR)}\quad \frac{\tau' \le \tau}{\tau' \text{ SAR} \le \tau \text{ SAR}}$$

Figure 3: Subtyping rules of SOOP.

follows:

$$\begin{array}{rcl} v & ::= & x \mid n \mid b \mid \mathsf{null} \mid \lambda x.\, e \mid \{\!|l_1 = v_1, \ldots, l_m = v_m|\!\} \\ e & ::= & v \mid e_1(e_2) \mid \mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\ & & \mid \mathsf{is\_zero}\ (e') \mid \mathsf{succ}\ (e') \mid \mathsf{pred}\ (e') \\ & & \mid \{\!|l_1 = e_1, \ldots, l_m = e_m|\!\} \mid e'.l \\ & & \mid \mathsf{ref}\ e' \mid !\, e' \mid \mathsf{set}\ (e_1, e_2) \\ & & \mid \mathsf{let}\ x\ \mathsf{be\ SAR\ to}\ e' \mid \uparrow e' \end{array}$$

where $n \in \mathit{Num} \stackrel{\text{def}}{=} \{0, 1, \ldots\}$ and $b \in \mathit{Bool} \stackrel{\text{def}}{=} \{\mathsf{true}, \mathsf{false}\}$. Both $\lambda x.\, e$ and $\mathsf{let}\ x\ \mathsf{be\ SAR\ to}\ e$ bind x in e; $\uparrow e'$ dereferences the SAR cell which is the value of e'.

3.2 Type System

The set of types of SOOP programs $\tau \in \mathit{Typ}$ is the least inductive collection satisfying

$$\tau \ ::= \ \mathsf{Num} \mid \mathsf{Bool} \mid \tau' \text{ -> } \tau'' \mid \{\!|l_1 : \tau_1, \ldots, l_m : \tau_m|\!\} \mid \tau' \text{ Ref} \mid \tau' \text{ SAR}$$

As in LOOP there is a subtyping relation $\le$ on types of SOOP—it is the least reflexive transitive relation generated by the rules in Figure 3; these include the standard rules for record and function types [Car84]. Non-trivial subtyping between reference types is unsound; however the discipline of once-assigning values to SAR cells allows us to treat the SAR type constructor as covariant without losing soundness. This covariance is crucial for the type-checking of the translation of LOOP class extensions into SOOP.

A *type environment* Γ is a finite map in $\mathit{Env} \stackrel{\text{def}}{=} \mathit{Var} \to \mathit{Typ}$; a *typing judgement* for expressions is a triple from $\mathit{Env} \times \mathit{Exp} \times \mathit{Typ}$ written in the form $\Gamma \vdash e : \tau$. The typing rules (whose inductive closure are the judgements) are given in Figure 4. Most of the rules reflect the usual typing properties of PCF terms, records and reference cells in the presence of subtyping. The rule (Self) gives the expected relation between the type of a SAR cell and the value it points to. The rule for the new construct (SAR) requires the body e to be of (a subtype of) the type of the value that x points to by assumption — intuitively this guarantees that e can be safely assigned to x.

4 Semantics of SOOP

We now give the semantics of SOOP and prove type soundness. Readers may want to skip to the translation of LOOP to SOOP, in Section 5, on first reading.

$$(\textsc{Sub})\quad \frac{\Gamma \vdash e : \tau \qquad \tau \leq \tau'}{\Gamma \vdash e : \tau'} \qquad (\textsc{Var})\quad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$(\textsc{Num})\quad \frac{}{\Gamma \vdash n : \mathsf{Num}} \qquad (\textsc{Bool})\quad \frac{}{\Gamma \vdash b : \mathsf{Bool}}$$

$$(\textsc{Cond})\quad \frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau} \qquad (\textsc{IsZero})\quad \frac{\Gamma \vdash e : \mathsf{Num}}{\Gamma \vdash \mathsf{is\_zero}\ (e) : \mathsf{Bool}}$$

$$(\textsc{Succ})\quad \frac{\Gamma \vdash e : \mathsf{Num}}{\Gamma \vdash \mathsf{succ}\ (e) : \mathsf{Num}} \qquad (\textsc{Pred})\quad \frac{\Gamma \vdash e : \mathsf{Num}}{\Gamma \vdash \mathsf{pred}\ (e) : \mathsf{Num}}$$

$$(\textsc{App})\quad \frac{\Gamma \vdash e_1 : \tau \text{ -> } \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1(e_2) : \tau'} \qquad (\textsc{Abs})\quad \frac{\Gamma \| \{x \mapsto \tau\} \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : \tau \text{ -> } \tau'}$$

$$(\mathrm{SAR})\quad \frac{\Gamma \| \{x \mapsto \tau\ \mathsf{SAR}\} \vdash e : \tau}{\Gamma \vdash \mathsf{let}\ x\ \mathsf{be\ SAR\ to}\ e : \tau} \qquad (\textsc{Self})\quad \frac{\Gamma \vdash e : \tau\ \mathsf{SAR}}{\Gamma \vdash \uparrow e : \tau}$$

$$(\textsc{Ref})\quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ref}\ e : \tau\ \mathsf{Ref}} \qquad (\textsc{Deref})\quad \frac{\Gamma \vdash e : \tau\ \mathsf{Ref}}{\Gamma \vdash !\, e : \tau}$$

$$(\textsc{Set})\quad \frac{\Gamma \vdash e_1 : \tau\ \mathsf{Ref} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{set}\ (e_1,\ e_2) : \tau} \qquad (\textsc{Select})\quad \frac{\Gamma \vdash e : \{\!|l : \tau|\!\}}{\Gamma \vdash e.l : \tau}$$

$$(\textsc{Record})\quad \frac{\Gamma \vdash e_i : \tau_i \text{ for all } i \in \{1, \ldots, m\}}{\Gamma \vdash \{\!|l_1 = e_1, \ldots, l_m = e_m|\!\} : \{\!|l_1 : \tau_1, \ldots, l_m : \tau_m|\!\}}$$

Figure 4: Typing rules of SOOP.

4.1 Operational Interpreter for SOOP

We give semantics of SOOP in the general framework of [FH92, MT91] developed further in [CF91, WF91]. The operational interpreter of the language is specified by binary relations between SOOP terms in memory environments, which represent the notion of computation. As in [WF91] the memory is not treated as a syntactic context but instead as a function defined on variables, because the let x be SAR to e construct cannot be represented as a pair of independent allocation and assignment without loss of type soundness, and therefore a memory context may need more than one "holes" for each of the bodies of these constructs being evaluated. The basic schema needs certain adjustments in order to accommodate the two kinds of reference cells in the language (Ref and SAR).

We use the notation $L + R$ for the disjoint sum (co-product) S of two sets L and R, with the injection $inl_S \in L \to S$, projection $outl_S \in S \to L$ and discriminator functions $isl_S \in S \to Bool$ (and the respective for R) defined for it (the subscripts will be dropped when the context determines S). The flattening projection $out_S \in S \to L \cup R$ is defined by

$$out_S(v) = \begin{cases} outl_S(v), \text{ if } isl_S(v) \\ outr_S(v), \text{ otherwise.} \end{cases}$$

The notation $\{x \mapsto v\}$ will be used for the map defined only on x with value v; the concatenation $\mu \| \mu'$ of two maps μ and μ' satisfies

$$(\mu \| \mu')(x) = \begin{cases} \mu'(x), \text{ if } x \in dom(\mu') \\ \mu(x), \text{ otherwise.} \end{cases}$$

$\langle \Sigma,\ R[(\lambda x.\, e)(v)]\rangle$	$\mapsto_1$	$\langle \Sigma,\ R[\{v/x\}e]\rangle$	
$\langle \Sigma,\ R[\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2]\rangle$	$\mapsto_1$	$\langle \Sigma,\ R[e_1]\rangle$	
$\langle \Sigma,\ R[\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2]\rangle$	$\mapsto_1$	$\langle \Sigma,\ R[e_2]\rangle$	
$\langle \Sigma,\ R[\mathsf{is\_zero}\ (n)]\rangle$	$\mapsto_1$	$\langle \Sigma,\ R[\mathsf{true}]\rangle$,	if $[\![n]\!] = 0$
$\langle \Sigma,\ R[\mathsf{is\_zero}\ (n)]\rangle$	$\mapsto_1$	$\langle \Sigma,\ R[\mathsf{false}]\rangle$,	if $[\![n]\!] \neq 0$
$\langle \Sigma,\ R[\mathsf{succ}\ (n)]\rangle$	$\mapsto_1$	$\langle \Sigma,\ R[n']\rangle$,	where $[\![n']\!] = [\![n]\!] + 1$
$\langle \Sigma,\ R[\mathsf{pred}\ (n)]\rangle$	$\mapsto_1$	$\langle \Sigma,\ R[n']\rangle$,	where $[\![n']\!] = [\![n]\!] - 1$
$\langle \Sigma,\ R[\{\!|l_1 = v_1,\ \ldots,\ l_m = v_m|\!\}.l_k]\rangle$	$\mapsto_1$	$\langle \Sigma,\ R[v_k]\rangle$,	if $k \in \{1,\ \ldots,\ m\}$
$\langle \Sigma,\ R[\mathsf{ref}\ v]\rangle$	$\mapsto_1$	$\langle \Sigma \| \{x \mapsto inl(v)\},\ R[x]\rangle$,	where $x \notin dom(\Sigma)$
$\langle \Sigma,\ R[!\ x]\rangle$	$\mapsto_1$	$\langle \Sigma,\ R[out(\Sigma(x))]\rangle$,	if $isl(\Sigma(x))$
$\langle \Sigma,\ R[\mathsf{set}\ (x,\ v)]\rangle$	$\mapsto_1$	$\langle \Sigma \| \{x \mapsto inl(v)\},\ R[v]\rangle$,	if $isl(\Sigma(x))$
$\langle \Sigma,\ R[\uparrow x]\rangle$	$\mapsto_1$	$\langle \Sigma,\ R[out(\Sigma(x))]\rangle$,	if $isr(\Sigma(x))$ and $outr(\Sigma(x)) \neq \mathsf{null}$
$\langle \Sigma,\ R[\uparrow x]\rangle$	$\mapsto_1$	$\langle \emptyset,\ \mathsf{null}\rangle$,	if $isr(\Sigma(x))$ and $outr(\Sigma(x)) = \mathsf{null}$
$\langle \Sigma,\ R[\mathsf{let}\ x\ \mathsf{be\ SAR\ to}\ e]\rangle$	$\mapsto_1$	$\langle \Sigma \| \{x' \mapsto inr(\mathsf{null})\},\ R[\mathsf{let}\ x'\ \mathsf{be\ SAR\ to}\ \{x'/x\}e]\rangle$,	
		if $x \notin dom(\Sigma)$ or $isl(\Sigma(x))$, and where $x' \notin dom(\Sigma)$	
$\langle \Sigma,\ R[\mathsf{let}\ x\ \mathsf{be\ SAR\ to}\ v]\rangle$	$\mapsto_1$	$\langle \Sigma \| \{x \mapsto inr(v)\},\ R[v]\rangle$,	if $isr(\Sigma(x))$

Figure 5: The single-step computation.

A *memory* Σ is a finite map in $Mem \stackrel{\text{def}}{=} Var \to (Val + Val)$. The intended interpretation is: if $x \in dom(\Sigma)$, then x is the identifier of a reference cell; $isl(\Sigma(x))$ is true if it is a Ref cell, and $isr(\Sigma(x))$ is true if the cell is a SAR. In both cases $out(\Sigma(x))$ is the value contained in the cell. An expression e is *closed* in a memory Σ if $FV(e) \subseteq dom(\Sigma)$; here $FV(e)$ denotes the set of free variables of the expression e. A memory Σ is *consistent* if $out(\Sigma(x))$ is closed in Σ for all $x \in dom(\Sigma)$.

A *computation state* $\langle \Sigma,\ e\rangle$ is a pair of a memory and an expression; it is closed if Σ is consistent and e is closed in Σ. We consider two states $\langle \Sigma,\ e\rangle$ and $\langle \Sigma',\ e'\rangle$ identical if they only differ in the names of the free variables, i.e. if there exists a bijection μ from $dom(\Sigma)$ to $dom(\Sigma')$ and a corresponding substitution $\sigma = \{\mu(x)/x$ for all $x \in dom(\Sigma)\}$ such that $\sigma e = e'$ and $\sigma\Sigma(x) = \Sigma'(\mu(x))$ for all $x \in dom(\Sigma)$.

A *reduction context* $R[\circ]$ with respect to a memory Σ is recursively defined as

$$
\begin{array}{rcl}
R[\circ] & ::= & \circ \mid R'[\circ](e) \mid v(R'[\circ]) \mid \mathsf{if}\ R'[\circ]\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
& \mid & \mathsf{is\_zero}\ (R'[\circ]) \mid \mathsf{succ}\ (R'[\circ]) \mid \mathsf{pred}\ (R'[\circ]) \\
& \mid & \{\!|l_1 = v_1,\ \ldots,\ l_{k-1} = v_{k-1},\ l_k = R'[\circ],\ l_{k+1} = e_{k+1},\ \ldots,\ l_m = e_m|\!\} \mid R'[\circ].l \\
& \mid & \mathsf{ref}\ R'[\circ] \mid !\ R'[\circ] \mid \mathsf{set}\ (R'[\circ],\ e) \mid \mathsf{set}\ (v,\ R'[\circ]) \\
& \mid & \mathsf{let}\ x\ \mathsf{be\ SAR\ to}\ R'[\circ],\ \text{if}\ isr(\Sigma(x)) \mid \uparrow R'[\circ]
\end{array}
$$

where $R'[\circ]$ is a reduction context with respect to Σ. $R[e]$ is the result of substituting e for $\circ$ in $R[\circ]$; it may involve name-capture.

The *single-step computation* $\mapsto_1$ is the least binary relation on $Mem \times Exp$ which relates all closed computation states of the forms shown in Figure 5, where $R[\circ]$ is a reduction context with respect to Σ, and $[\![\cdot]\!] \in Num \to \mathbb{N}$ is the obvious interpretation of the numerals; the arguments to R in the left column are *redeces*. We write $\{e'/x\}e$ for the name-capture avoiding substitution of the term e' for x in e; in particular it satisfies

$$\text{if } x' \in FV(e'),\ \text{then } \{e'/x\}\lambda x'.\, e = \lambda x''.\, \{e'/x\}\{x''/x'\}e,\ \text{where } x'' \notin FV(e) \cup FV(e') \cup \{x\}.$$

By induction on the structure of terms and inspection of $\mapsto_1$ it is easy to demonstrate the correctness of the following.

LEMMA 4.1 $\mapsto_1$ defines a partial function on *Exp*, i.e. for any state $\langle \Sigma,\ e\rangle$ there exists at most one $\langle \Sigma',\ e'\rangle$ such that $\langle \Sigma,\ e\rangle \mapsto_1 \langle \Sigma',\ e'\rangle$.

LEMMA 4.2 If $\langle \Sigma,\ e\rangle \mapsto_1 \langle \Sigma',\ e'\rangle$, then $\langle \Sigma',\ e'\rangle$ is closed.

The *computation* $\mapsto^*$ is the reflexive transitive closure of $\mapsto_1$. A closed computation state $\langle \Sigma,\ e\rangle$ is *stuck* (written $\langle \Sigma,\ e\rangle\wr$) if $e \notin Val$ and there is no state $\langle \Sigma',\ e'\rangle$ such that $\langle \Sigma,\ e\rangle \mapsto_1 \langle \Sigma',\ e'\rangle$. A state $\langle \Sigma,\ e\rangle$ *diverges* (written $\langle \Sigma,\ e\rangle \uparrow$) if $\langle \Sigma,\ e\rangle \mapsto_1 \langle \Sigma',\ e'\rangle$ for some $\langle \Sigma',\ e'\rangle$, and for each $\langle \Sigma',\ e'\rangle$ such that $\langle \Sigma,\ e\rangle \mapsto^* \langle \Sigma',\ e'\rangle$ there exists a state $\langle \Sigma'',\ e''\rangle$ such that $\langle \Sigma',\ e'\rangle \mapsto_1 \langle \Sigma'',\ e''\rangle$.

4.2 Soundness of the Type System of SOOP

A memory Σ is *typable* in a type environment Γ if for all $x \in dom(\Sigma)$

(i) $isl(\Sigma(x))$ and $\Gamma \vdash x : \tau$ Ref and $\Gamma \vdash outl(\Sigma(x)) : \tau$ for some type τ; or

(ii) $isr(\Sigma(x))$ and either $\Gamma \vdash x : \tau$ SAR and $\Gamma \vdash outr(\Sigma(x)) : \tau$ for some type τ, or $outr(\Sigma(x)) =$ null

A typing judgement for computation states is a quadruple of the form $\Gamma \vdash \langle \Sigma,\ e\rangle : \tau$ in which $\langle \Sigma,\ e\rangle$ is a closed computation state, Σ is typable in Γ, and $\Gamma \vdash e : \tau$. A state $\langle \Sigma,\ e\rangle$ has a type if $\Gamma \vdash \langle \Sigma,\ e\rangle : \tau$ for some type environment Γ and type τ.

LEMMA 4.3 (REPLACEMENT) If $\Gamma \vdash \langle \Sigma,\ R[e]\rangle : \tau$, $\Gamma \vdash e : \tau'$, $\Gamma \vdash e' : \tau'$ and $FV(e') \subseteq FV(e)$, then $\Gamma \vdash \langle \Sigma,\ R[e']\rangle : \tau$.

Proof: By induction on the structure of the proof of $\Gamma \vdash \langle \Sigma,\ R[e]\rangle : \tau$. □

LEMMA 4.4 (SUBJECT REDUCTION) If $\Gamma \vdash \langle \Sigma,\ e\rangle : \tau$ and $\langle \Sigma,\ e\rangle \mapsto_1 \langle \Sigma',\ e'\rangle$, then either $e' =$ null, or $\Gamma' \vdash \langle \Sigma',\ e'\rangle : \tau$ for some type environment Γ'.

Proof: Based on a case analysis of the single-step computation. In most cases the statement is a direct corollary of Lemma 4.3 and the typability of Σ in Γ. The type environment Γ' is an extension of Γ in the cases when the reduction extends the domain of the memory (corresponding to the allocation of a cell). □

LEMMA 4.5 (STUCK EXPRESSIONS ARE UNTYPABLE) If $\langle \Sigma,\ e\rangle\wr$, then $\langle \Sigma,\ e\rangle$ has no type.

Proof: Note that the proof rules for types of SOOP terms have as antecedents judgements about the types of all of their subexpressions, hence all subterms of a typable term must be typable. Also note that there is exactly one typing rule for each language construct, in addition to the (SUB) rule which however can only be applied if a type for the term is already deduced. Suppose now that $\langle \Sigma,\ e\rangle\wr$ and let $e = R[e_0]$ be a decomposition of e such that R is a reduction context with respect to Σ and $e_0 \notin Val$ and for every further decomposition $e_0 = R'[e']$ either $R'[\circ] = \circ$ or $e' \in Val$; such R and e_0 exist since $e \notin Val$, and Val forms the inductive basis for *Exp*. The rest of the proof employs analysis of the structure of e_0; we will illustrate the ideas for the case $e_0 = e_1(e_2)$.

Both e_1 and e_2 must be values, otherwise further nontrivial decomposition of e_0 is possible. But e_1 cannot be of the form $\lambda x.\, e'$, because then it is a redex, and $\langle \Sigma,\ R[e_0]\rangle \mapsto_1 \langle \Sigma,\ R[\{e_2/x\}e']\rangle$, which contradicts $\langle \Sigma,\ e\rangle\wr$. Hence e_1 is a non-functional value, and an inspection of the possible types for these shows that none of them matches the antecedent of the only effectively applicable typing rule (APP); it must be noted here that each free variable x of e is in $dom(\Sigma)$, and the typability of Σ implies that the type of x is either τ Ref or τ SAR for some type τ. Therefore e_0 is untypable, and so is e. □

As a corollary of Lemmas 4.4 and 4.5 we have

$$
\begin{array}{rcl@{\qquad}rcl}
[\![\{\overline{x_i : \tau_i}\}]\!] & = & \{|\overline{x_i : [\![\tau_i]\!]\ \mathsf{Ref}}|\} & [\![\{\overline{m_i : \tau_i}\}]\!] & = & \{|\overline{m_i : [\![\tau_i]\!]}|\} \\
[\![\texttt{TObj}\{M\}]\!] & = & [\![M]\!] & [\![(I, M)]\!] & = & \{|\mathsf{inst} : [\![I]\!], \mathsf{meth} : [\![M]\!]|\} \\
[\![\texttt{TClass}\{C\}]\!] & = & [\![C]\!]\ \mathsf{SAR} \texttt{ -> } [\![C]\!] & [\![\texttt{TOpenObj}\{C\}]\!] & = & [\![C]\!]\ \mathsf{SAR}
\end{array}
$$

Figure 6: Translation of LOOP Types to SOOP Types

THEOREM 4.6 (SOUNDNESS OF THE TYPE SYSTEM) If $\Gamma \vdash \langle \Sigma,\ e\rangle : \tau$ for some Γ, then either $\langle \Sigma,\ e\rangle \uparrow$, or $\langle \Sigma,\ e\rangle \mapsto^* \langle \emptyset,\ \mathsf{null}\rangle$, or $\langle \Sigma,\ e\rangle \mapsto^* \langle \Sigma',\ e'\rangle$ for some Γ' and $\langle \Sigma',\ e'\rangle$ such that $e' \in \mathit{Val}$ and $\Gamma' \vdash \langle \Sigma',\ e'\rangle : \tau$.

5 Translating LOOP to SOOP

The translation of LOOP terms to SOOP terms may be defined by a simple induction on term structure. In order to create a value corresponding to a LOOP object, a single-assignment cell is first allocated and its name added to the current environment; this name will then play the rôle of `self` in the object's methods. A pair of records, corresponding to the sets of instance variables and methods of the object, is then constructed, with fields incrementally added from the chain of inheritance of the object. Finally, this completed record is stored into the created cell; since the name of this cell is (presumably) used in the construction of the record, this assignment "ties the knot" and constructs the self-referential object.

We begin with a translation of LOOP types and contexts to SOOP types; the translation is shown in Figure 6. In this framework, objects types translate into record types, and classes translate into functions from single-assignment cells to records. Some motivations for this encoding are to be found in the previous section.

We may translate LOOP terms into SOOP terms via the translation function $[\![\cdot]\!]$ of Figure 7. PCF terms translate to themselves, as SOOP is just another extension of PCF. As discussed in the previous section, objects are interpreted as records; therefore, the "send message" expression is translated into a record selection from the corresponding SOOP term. Terms of type `TOpenObj` (i.e., the "self" of an object) are translated into single-assignment cells that point to the object in question.

Terms of type `TClass` represent classes, which as mentioned above translate to functions from a SAR cell name to a record. The `empty` expression therefore is translated as a function returning the record denoting the class with no instance variables or methods. Classes created by the `class` expression are translated into functions in a similar way. These functions first apply the superclass to their bound parameter, creating a record denoting an instance of the superclass. This record is then modified by the addition or replacement of one or more fields, and returned. Note that extend ... with ... is not part of the SOOP syntax, it is metanotation defined as follows.

DEFINITION 5.1

$$
\begin{array}{l}
\mathsf{extend}\ (\texttt{TClass}(I, M))\ e\ \mathsf{with} \\
\{|\mathsf{inst} = \{|\overline{x_i = e_i}|\}, \mathsf{meth} = \{|\overline{m_i = f_i}|\}|\}
\end{array}
\quad \stackrel{def}{=} \quad
\begin{array}{l}
\{|\mathsf{inst} = \{|\overline{x_i' = e.\mathsf{inst}.x_i'}, \overline{x_i = e_i}|\} \\
\mathsf{meth} = \{|\overline{m_i' = e.\mathsf{meth}.m_i'}, \overline{m_i = f_i}|\}|\}
\end{array}
$$

for each x_i', m_i' such that $x_i' \in \mathit{dom}(I)$ but $x_i' \notin \overline{x_i = e_i}$ and $m_i' \in \mathit{dom}(M)$ but $m_i' \notin \overline{m_i = f_i}$.

$$
\begin{array}{rcl}
[\![e\texttt{<-}m]\!] & = & [\![e]\!].m \\
[\![s]\!] & = & \uparrow s \\
[\![e_1.x]\!] & = & !\,([\![e_1]\!].\mathsf{inst}).x \\
[\![e_1.x := e_2]\!] & = & \mathsf{set}\,(([\![e_1]\!].\mathsf{inst}).x,\ [\![e_2]\!]) \\
[\![e_1.m]\!] & = & ([\![e_1]\!].\mathsf{meth}).m \\
[\![\texttt{new}\ e]\!] & = & (\mathsf{let}\ s\ \mathsf{be\ SAR\ to}\ [\![e]\!]\ s).\mathsf{meth} \\
[\![\texttt{empty}]\!] & = & \lambda s.\,\{|\mathsf{inst} = \{|\},\mathsf{meth} = \{|\}|\} \\
[\![\texttt{class}\ s\ \texttt{is}\ e_0 : \tau\ \texttt{with} & & \lambda s.\,\mathsf{extend}\ (\tau)([\![e_0]\!](s))\ \mathsf{with} \\
\quad \{\{\overline{x = e}\}\{\overline{m = f}\}\}]\!] & = & \quad \{|\mathsf{inst} = \{|\overline{x = \mathsf{ref}\ [\![e]\!]}|\},\mathsf{meth} = \{|\overline{m = [\![f]\!]}|\}|\}
\end{array}
$$

Figure 7: Translation of LOOP Terms to SOOP Terms

The `new` expression creates a SAR cell and passes it as a parameter to the corresponding class expression, resulting in a new record. The "public" half of this record is then selected and returned, thereby hiding the instance variables from view.

This translation gives a semantics for LOOP programs in terms of their corresponding SOOP programs; for any closed terms e_1 and e_2, we have an evaluation relation $e_1 \Rightarrow e_2$ if and only if $[\![e_1]\!] \mapsto^* [\![e_2]\!]$. Similarly, the translation gives a definition of the types of LOOP programs:

DEFINITION 5.2 A LOOP program e has type τ in environment E, written $E \models e : \tau$, if and only if $[\![E]\!] \vdash_{\mathrm{SOOP}} [\![e]\!] : [\![\tau]\!]$.

With this definition, we can easily show that the type system of LOOP is sound:

LEMMA 5.3 (TYPE SOUNDNESS) For any LOOP term e and environment E, if $E \vdash_{\mathrm{LOOP}} e : \tau$, then $E \models e : \tau$.

As an obvious corollary to this lemma and Theorem 4.6 is

THEOREM 5.4 Well typed LOOP programs do not get stuck; i.e., for all LOOP programs e, if $\vdash_{\mathrm{LOOP}} e : \tau$ for some type τ, then the SOOP program $[\![e]\!]$ does not get stuck.

To illustrate these ideas, the example LOOP program of Section 2.2 translated into SOOP is shown below.

```
let Point1 = λs.extend {|inst = {||}, meth={||}|} with {|
  inst = {|x = ref 0|},
  meth = {|getx = λa. !((↑s).inst.x),
            setx = λn. set((↑s).inst.x, n); (),
            align = λp. p.setx ((↑s).meth.getx ()) |}|}
in let Point2 = λs.extend Point1 s with {|
  inst={|y = ref 0|},
  meth = {|gety = λa. !((↑s).inst.y),
            sety = λn. set((↑s).inst.y, n); (),
            align2 = λp. (↑s).meth.align p; p.sety ((↑s).meth.gety ()) |}|}
in let p1 = (let s be SAR to Point2 s).meth
in let p2 = (let s be SAR to Point2 s).meth
in p1.align2 p2
```

6 Conclusions

We give an operational interpretation of an object-oriented language with mutable instance variables, similar in flavor to the "wrapper semantics" of [CP89, Hen91], but in a strongly typed framework. This requires the solution of two basic semantic problems: avoiding the reallocation of instance variable cells resulting from an object's self-reference (in the fixed point encoding), and the type-correct extension of classes (based on the subtyping relation between object types). Both are solved by the introduction of a new language construct: the single-assignment reference cell.

There are several obvious directions to extend this approach with different degrees of polymorphism. Adding Damas-Milner style polymorphism to the purely functional portion of the language (restricting instance variables to have monomorphic types) is straightforward. More expressive polymorphic systems, such as F-bounded quantification and various imperative type systems, will likely require additional machinery.

Acknowledgement

We are grateful to an anonymous referee for several constructive comments and bringing related work to our attention.

References

[BM92] K. Bruce and J. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 316–327, 1992.

[Bru93] K. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 285–298, 1993.

[Car84] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *Lecture notes in Computer Science*, pages 51–67. Springer-Verlag, 1984.

[CCH+89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded polymorphism for object-oriented progrmming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[CF91] E. Crank and M. Felleisen. Parameter-passing and the lambda calculus. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1991.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1990.

[CP89] W. Cook and J. Parlsberg. A denotational semantics of inheritence and its correctness. In *OOPSLA '89 Proceedings*, pages 433–443, 1989.

[FH92] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992.

[GJ90] J. Graver and R. Johnson. A type system for Smalltalk. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[Hen91] A. V. Hense. Wrapper semantics of an object-oriented programming language with state. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, volume 526 of *Lecture notes in Computer Science*, pages 548–567. Springer-Verlag, 1991.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[Mit90] J. Mitchell. Towards a typed foundation for method specialization and inheritence. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990.

[MT91] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[Pie92] B. Pierce. Bounded quantification is undecidible. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 305–315, 1992.

[PT93] B. Pierce and D. N. Turner. Object-oriented programming without recursive types. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 299–312, 1993.

[Red88] U. S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proceeding of the ACM Conference on Lisp and Functional Programming*, pages 289–297, 1988.

[SCB+86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *Proceedings of OOPSLA Conference*, pages 9–16, 1986.

[Wad92] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, 1992.

[Wan89] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92–97. IEEE, 1989.

[WF91] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Rice University Department of Computer Science, 1991. (revised June 1992).