, , 1–41 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

An Interpretation of Typed OOP in a Language with State

JONATHAN EIFRIG* ** eifrig@cs.jhu.edu
SCOTT SMITH* scott@cs.jhu.edu
VALERY TRIFONOV* ** trifonov@cs.jhu.edu
AMY ZWARICO** amy@cs.jhu.edu
Department of Computer Science, The Johns Hopkins University

Received May 1, 1991

Editor: Ian A. Mason, Martin Odersky

Abstract. In this paper we give semantics to LOOP, an expressive typed object-oriented programming language with updatable instance variables. LOOP has a rich type system that allows for the typing of methods operating over an open-ended "self" type. We prove the type system given is sound; *i.e.*, well-typed programs do not experience "message not understood" errors. The semantics of LOOP is given by a translation into a state-based language, SOOP, that contains reference cells, records, and a form of F-bounded polymorphic type.

Keywords: Object-oriented programming, type systems, state in programming

1. Introduction

Developing full and faithful type systems for object-oriented programming languages is a well-known and difficult research problem. To frame the problem we desire a static type system that preserves all of the classic features of (untyped) class-based OOP, including treatment of two particularly difficult issues: binary methods and object subsumption. Significant steps have recently been made toward solving this problem [10], [7], [4], [3], [22]. One major gap in the aforementioned works is they take a functional view of objects, whereas objects are inherently state-carrying entities.

What we accomplish here is a provably sound interpretation of a typed imperative OOP language. We define a representative language, LOOP, providing notation for class definitions, subclassing, multiple inheritance, binary methods, protection of instance variables from outside access, dynamic creation of objects of a class, and message send to "self" or other objects. Thus, the language is similar on the surface to a "sugar free" version of C++ or Object Pascal. The main improvement is LOOP's richer type system, which allows for typing of class methods that take and return objects of an open-ended "self-type." We base our approach on the ideas

* Partially supported by NSF grant CCR-9109070
** Partially supported by AFOSR grant F49620-93-1-0169

of [10]. In addition, we show how these notions may be combined with subtype-based inheritance, which is sometimes more powerful. We call the F-bounded view of typing inheritance the *open-self* view, and subtype-based inheritance found in languages such as C++ the *fixed-self* view because the type of methods does not change upon inheritance. We show that both have strengths and weaknesses: it is always possible to lift objects up the inheritance hierarchy in the fixed-self approach, a property that sometimes fails for the open-self approach. Any object programmer knows this *object subsumption* feature is very useful. It is an open problem of how the advantages of both approaches may be combined in one language.

We give the semantics of Loop by translation into a typed imperative language, Soop. We take a translational approach to semantics rather than the direct approach, because Soop itself is a useful foundation for various object coding ideas and is simpler to understand than the more complex object typing systems. Untyped Soop is the call-by-value lambda calculus extended with records and references. The type system of Soop includes records, subtyping and F-bounded polymorphism for the interpretation of inheritance. We give a collection of type rules for Soop and show the rules sound with respect to the operational semantics: no run-time type errors occur.

We develop independent translations of Loop terms and types. The translation of terms interprets objects as a form of record and classes as object generators as in [9], [18]. The main difference is use of a special memory-based "weak" fixed-point combinator in place of the normal Y. In a functional world the two are equivalent, but not in the presence of state. The most important feature of the type translation is the use of F-bounded polymorphism in Soop to interpret the open-ended notion of "self" type needed for the class types of Loop. This means we obtain the proper typings of methods that take objects of "self" type as argument [10].

Section 2 presents the syntax and type rules for Loop, and concludes with an example that both illustrates how to program in Loop, and shows some of the problems involved in typing binary methods and the competing "fixed-self" and "open-self" solutions. Section 3 defines the Soop language and type system, gives its formal operational semantics, and contains a full proof of Soop type soundness. Section 4 contains the translation of Loop into Soop and a proof of the soundness of this translation, yielding the fact that typed Loop programs do not experience "message not understood" errors. This paper is a significantly extended version of [12].

2. The Loop Language and Type System

We begin our discussion by defining a representative object-oriented programming language, Loop (Little Object-Oriented Programming language). Loop is an extension of call-by-value PCF, with added syntax for classes, objects, object creation, message send, and instance variable access.

Our convention for use of metavariables to describe the syntax of Loop is as follows. We let v range over the set of identifiers, and also use identifiers s for

$$
\begin{array}{rcll}
\mathit{Num} & \ni & n & ::= \texttt{0} \mid \texttt{1} \mid \ldots \\
\mathit{Bool} & \ni & b & ::= \texttt{true} \mid \texttt{false} \\
\mathit{Exp} & \ni & e & ::= v \mid n \mid b \mid \texttt{pred}(e) \mid \texttt{succ}(e) \mid \texttt{is\_zero}(e) \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \\
 & & & \quad \mid \texttt{fn } v \texttt{ => } e \mid e(e) \mid e.x \mid e.x \texttt{:=} e \mid e \texttt{<-} m \mid \texttt{new } e \\
 & & & \quad \mid \texttt{class } s \texttt{ super } \overline{u_i \texttt{ of } e_i} \texttt{ inst } \overline{x_j = e_j} \texttt{ meth } \overline{m_k = e_k}
\end{array}
$$

Figure 1. Loop Syntax

$$
\begin{array}{rcll}
\mathit{Typ} & \ni & \tau & ::= t \mid \texttt{Bool} \mid \texttt{Nat} \mid \tau \rightarrow \tau' \mid \texttt{Obj}(t)M \mid \texttt{Class}(t)(I; M) \\
 & & & \quad \mid \texttt{PreObj}\, M \mid \texttt{Self}(I; t) \mid \texttt{Super}(I; M) \\
 & & I & ::= \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \\
 & & M & ::= \{m_1 : \tau_1, \ldots, m_n : \tau_n\}
\end{array}
$$

Figure 2. Loop Types

bound class names, and u for bound superclass names. There is a set of labels K_{lab}, with x and m ranging over this set, used for instance variable names and method names, respectively. The "vector notation" $\overline{A_i}$ will be used to indicate a comma-separated sequence of elements, ranged over by the indicated index variable: $\overline{u_i \texttt{ of } e_i} \stackrel{\text{def}}{=} u_1 \texttt{ of } x_1, \ldots, u_n \texttt{ of } x_n$.

The syntax of Loop expressions is given in Figure 1. Classes are created by extending a (possibly empty) set of existing classes with additional instance variables and methods. The expression $\texttt{class } s \texttt{ super } \overline{u_i \texttt{ of } e_i} \texttt{ inst } \overline{x_j = e'_j} \texttt{ meth } \overline{m_k = e''_k}$ extends the class(es) e_i by adding the new instance variables and methods indicated. Within the body of the class expression, the name s is bound and plays the role of "self," while the u_i provide access to the parent class definitions. Use of the u_i to access parent classes is analogous to the use of `super` in Smalltalk programs, but since we allow multiple inheritance a different name for each superclass is needed. Note that Loop does not provide for implicit inheritance of superclass members; *every* method of a class must be explicitly listed in its declaration. Thus, $m_j = u_i \texttt{<-} m_j$ denotes that m_j is inherited from superclass u_i. This serves to resolve any ambiguities with multiple inheritance. Classes consist of collections of mutable instance variables and immutable methods. To simplify our presentation, we shall restrict ourselves to considering only "protected" instance variables and "public" methods (using C++ terminology.) Instance variables are visible throughout the body of their enclosing class expression as well as any later extensions of the class, but are not visible in the objects generated by these classes. Methods, on the other hand, are visible in the generated objects as well. The current value of an instance variable is accessed by the expression $e.x$, and updated by $e.x \texttt{:=} e'$. Methods are accessed by $e \texttt{<-} m$.

The syntax of Loop types is given in Figure 2. The familiar function, number, and Boolean types need no explanation. I and M are *instance signatures* and *method signatures* respectively, associating types with instance variable and method

names. Equivalently, they may be viewed as finite mappings from $K_{lab} \to Typ$; the distinction will be clear in context.

Terms of type $\texttt{Obj}(t)M$ denote objects; the signature M specifies the "interface" of the object, listing the names and types of the messages which the object recognizes. The type variable t is bound in $\texttt{Obj}(t)M$ and represents the type of "self" within the object's interface. Terms e of type $\texttt{Class}(t)(I\,;\,M)$ represent classes, which may be used to generate new objects via $\texttt{new}\ e$, as well as extended by another `class` expression. The type variable t is bound within both I and M, and represents the type of objects generated by this class. $\texttt{Self}(I\,;\,t)$, $\texttt{Super}(I\,;\,M)$, and $\texttt{PreObj}\,M$ are used in type checking class expressions. The type $\texttt{Self}(I\,;\,t)$ is the type of a class object viewed from inside a class definition; there are in fact no terms of this type, since this internal view cannot escape. Types $\texttt{Super}(I\,;\,M)$ are the types of superclass objects viewed from inside a class definition. The internal view given by $\texttt{Self}(I\,;\,t)$ and $\texttt{Super}(I\,;\,M)$ exposes the instance variables, allowing them to be read and set. $\texttt{PreObj}\,M$ is used inside a class definition as the view of what objects will look like from the outside, with the instance variables removed. The "self-type" t may occur free in $\texttt{PreObj}\,M$, $\texttt{Self}(I\,;\,t)$, and $\texttt{Super}(I\,;\,M)$; such free type variables are bound by type constraints of the form $t \leq \texttt{PreObj}\,M$ when typing classes. It may help to understand the differences between a $\texttt{PreObj}\,M$ and an $\texttt{Obj}(t)M$ type to note the former will basically translate (in Section 4 below) to a record of methods, and the latter will basically be a recursive record type. The $\texttt{PreObj}\,M$ types will also be used in the subtyping rules for bookkeeping purposes.

We use some syntactic abbreviations to aid in program readability: $\texttt{let}\ v = e\ \texttt{in}\ e'$ stands for $(\texttt{fn}\ v \texttt{=>} e')(e)$, and sequencing $(e_1\,;\,e_2)$ for $(\texttt{fn}\ x \texttt{=>} \texttt{fn}\ y \texttt{=>} y)(e_1)(e_2)$. These give the expected behavior since functions are call-by-value. We also write $e()$ for the application of e to the "empty" object `new (class self super inst meth)` of type $\mathbf{1} \stackrel{\text{def}}{=} \texttt{Obj(t)\{\}}$.

2.1. The Loop Rules

Figure 3 axiomatizes the subtyping relation between Loop types. A type constraint system C consists of a sequence of type constraints $t \leq \texttt{PreObj}\,M$; C may equivalently be viewed as a mapping from type variables t to their corresponding upper bounds. $C \,\|\, t \leq \texttt{PreObj}\,M$ denotes the constraint system C extended by mapping t to $\texttt{PreObj}\,M$.

Some basic notions of subtyping are now reviewed [8], with special attention to issues that arise in Loop subtyping. The subtype relation on types $\tau \leq \tau'$ at first approximation corresponds to subset. Note in particular that objects in this view are functions from the label set to the methods, and an object type with added fields is then a *subtype* of its smaller ancestor, e.g.

$$\tau_{\texttt{ab}} \stackrel{\text{def}}{=} \texttt{Obj(SelfType)\{a:1→Nat, b:1→Bool\}} \leq \texttt{Obj(SelfType)\{a:1→Nat\}} \stackrel{\text{def}}{=} \tau_{\texttt{a}},$$

since $\tau_{\texttt{a}}$ places no constraint on the type of the `b` method. It may at first seem unlikely that this is provable from the subtyping rules of Figure 3, for there is no

$$(\mathit{Refl})\ \frac{}{C \vdash \tau \leq \tau} \qquad (\mathit{Trans})\ \frac{C \vdash \tau \leq \tau', \quad \tau' \leq \tau''}{C \vdash \tau \leq \tau''}$$

$$(\mathit{Hyp})\ \frac{t \in dom(C)}{C \vdash t \leq C(t)} \qquad (\mathit{Fun})\ \frac{C \vdash \tau_1' \leq \tau_1, \quad \tau_2 \leq \tau_2'}{C \vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

$$(\mathit{PreObj})\ \frac{C \vdash M(m) \leq M'(m),\ \forall m \in dom(M')}{C \vdash \mathtt{PreObj}\, M \leq \mathtt{PreObj}\, M'}$$

$$(\mathit{Fold})\ \frac{}{C \vdash \mathtt{PreObj}\, M[\mathtt{Obj}(t)\, M/t] \leq \mathtt{Obj}(t)\, M}$$

$$(\mathit{Fix})\ \frac{C \,\|\, t \leq \mathtt{PreObj}\, M' \vdash \mathtt{PreObj}\, M \leq \mathtt{PreObj}\, M'}{C \vdash \mathtt{Obj}(t)\, M \leq \mathtt{PreObj}\, M'[\mathtt{Obj}(t)\, M/t]}$$

Figure 3. LOOP subtyping rules

rule for directly comparing object types. However, there are many ways by which conclusions of the form $C \vdash \mathtt{Obj}(t)\, M \leq \mathtt{Obj}(t')\, M'$ may be proven. We focus on the three most common varieties. First and second, object folding and unfolding rules are derivable from (*Fix*) and (*Fold*), $\mathtt{Obj}(t)(M[\mathtt{Obj}(t)\, M/t]) \lesseqgtr \mathtt{Obj}(t)\, M$. Third, the (*Fold*) and (*Fix*) rules allow for an "inductive" comparison of object types in the spirit of [2] via the following derived rule:

$$(\mathit{ObjFix})\ \frac{C \,\|\, t \leq \mathtt{PreObj}\, M'[\mathtt{Obj}(t')\, M'/t'] \vdash \mathtt{PreObj}\, M \leq \mathtt{PreObj}\, M'[\mathtt{Obj}(t')\, M'/t']}{C \vdash \mathtt{Obj}(t)\, M \leq \mathtt{Obj}(t')\, M'}$$

Using this derived rule, $\tau_{\mathtt{ab}} \leq \tau_{\mathtt{a}}$ may be proven. There is an analogous derived rule for recursive types in SOOP, and in Section 3 the relationship between this rule and other rules in the literature will be discussed in more detail.

In the classic function subtyping rule (*Fun*) from [8], τ_1 and τ_1' are reversed since a "smaller" input τ_1' leads to "more" type-correct outputs possible. For instance, if a function has the domain $\tau_{\mathtt{a}}$ above, it can easily accept an argument of the smaller type $\tau_{\mathtt{ab}}$ and ignore the extra `b` method.

Serious difficulties arise when subtyping and use of `SelfType` in an object type combine. Consider the following two object types:

$$\tau_{\mathtt{abc}} \stackrel{\text{def}}{=} \mathtt{Obj(SelfType)\{a:1\to Nat,\ b:1\to Bool,\ c:SelfType\to SelfType\}}$$
$$\not\leq$$
$$\mathtt{Obj(SelfType')\{a:1\to Nat,\ c:SelfType'\to SelfType'\ \}} \stackrel{\text{def}}{=} \tau_{\mathtt{ac}}$$

By using the derived (*ObjFix*) rule above, we will need to show $\tau \leq \mathtt{SelfType}$ under the assumption $\mathtt{SelfType} \leq \tau$ for some τ, but this is clearly unprovable. It is not difficult to establish that if $\tau_{\mathtt{abc}} \leq \tau_{\mathtt{ac}}$ was provable, the type system would not be sound. Under the "open-self" view of class typing discussed in Section 2.2

below, it will turn out that $\tau_{\mathtt{abc}}$ could be the type of an object generated by a subclass of the class generating $\tau_{\mathtt{ac}}$. Observe then that these subobjects would not be subtypes. Note that if `SelfType` only occurs *positively* (or not at all) in the types of the methods, the (*ObjFix*) rule can be used to show subobjects are subtypes. This illustrates some potential problems in typing objects with binary methods, discussed in more detail in Section 2.2 below.

The type rules for Loop programs are listed in Figure 4. A type environment Γ is a finite map from variables to types, and $\Gamma \| x : \tau$ is its extension mapping x to type τ. The (*Class*) rule is the most important and complex rule. When type checking the bodies of class members (instances in the third antecedent of the rule, methods in the fourth), the final type of the objects that incorporate these definitions are not known, since the class currently being type-checked may have been extended by inheritance. We thus want to view the "self-type" here as open-ended and parametric. For this we introduce a new type variable t as the "self-type;" all that is known about t is that it must satisfy the constraint $t \leq \mathtt{PreObj}\, M$, i.e. it contains at least the methods M. This means the self-references in M (and in I) are to the open-ended "self." The class name s can be assumed to have type $\mathtt{Self}(I; t)$, superclass names are of type $\mathtt{Super}(I; M)$. The first antecedent forces the superclasses to have the types asserted for them, and the second one requires the class to be a parametric extension of each superclass.

Rules (*Self*) and (*Super*) type the uses of class and superclass names as expressions. Note the rules are slightly different. Rule (*Self*) gives the type t to the class name s; this type variable is constrained within C by $t \leq \mathtt{PreObj}\, M$ and thus (by rule (*Sub*)) s is also of type $\mathtt{PreObj}\, M$. The weaker type $\mathtt{PreObj}\, M$ is not given to s as s may be returned by a method, and all methods of the returned value should be visible, not merely the ones defined in this class.

Rule (*New*) is the obvious rule for typing objects. When a new object is created, its instance variables are hidden, and the "self" type t becomes fixed.

We now address how method update, modifying a method of an already existing object, may be encoded in Loop. The language itself has no features for method update, but a simple schema allows for it to be encoded. In a more complete Loop language, this would be an atomic feature. Suppose we had an arbitrary class with a single method m, intended to be updateable:

$$\mathtt{class}\ s\ \mathtt{super}\ \overline{u_i\ \mathtt{of}\ e_i}\ \mathtt{inst}\ \overline{x_j = e'_j}\ \mathtt{meth}\ m = e, \overline{m_k = e''_k}$$

This class is then encoded as

$$\begin{array}{l}\mathtt{class}\ s\ \mathtt{super}\ \overline{u_i\ \mathtt{of}\ e_i}\ \mathtt{inst}\ m\mathtt{code} = e, \overline{x_j = e'_j} \\ \quad \mathtt{meth}\ m = \mathtt{fn}\ v \Rightarrow s.m\mathtt{code}(v), \mathtt{set}m = \mathtt{fn}\ v \Rightarrow s.m\mathtt{code} := v, \overline{m_k = e''_k}\end{array}$$

Note the fact that the instance variable initialization e may refer to the "self" via s is critical to the translation. If the original class had type $\mathtt{Class}(t)(I; M)$, where $M(m) = \tau$, the modified class can be shown to have type $\mathtt{Class}(t)(m\mathtt{code}{:}\tau, I; \mathtt{set}m{:}\tau{\rightarrow}\tau, M)$. The presence of an updateable method will make inheritance from this class more restrictive—the negative occurence of τ in the type of the $\mathtt{set}m$ method forces the type of m to be preserved in all extensions of the class, i.e.

$$(Sub)\ \frac{C;\Gamma \vdash e : \tau \quad C \vdash \tau \leq \tau'}{C;\Gamma \vdash e : \tau'} \qquad (Var)\ \frac{v \in dom(\Gamma)}{C;\Gamma \vdash v : \Gamma(v)}$$

$$(Num)\ \frac{}{C;\Gamma \vdash n : \texttt{Nat}} \qquad (Bool)\ \frac{}{C;\Gamma \vdash b : \texttt{Bool}}$$

$$(Pred)\ \frac{C;\Gamma \vdash e : \texttt{Nat}}{C;\Gamma \vdash \texttt{pred}(e) : \texttt{Nat}} \qquad (Succ)\ \frac{C;\Gamma \vdash e : \texttt{Nat}}{C;\Gamma \vdash \texttt{succ}(e) : \texttt{Nat}}$$

$$(IsZero)\ \frac{C;\Gamma \vdash e : \texttt{Nat}}{C;\Gamma \vdash \texttt{is\_zero}(e) : \texttt{Bool}} \qquad (Cond)\ \frac{C;\Gamma \vdash e_1 : \texttt{Bool}, \quad e_2 : \tau, \quad e_3 : \tau}{C;\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau}$$

$$(App)\ \frac{C;\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad C;\Gamma \vdash e_2 : \tau}{C;\Gamma \vdash e_1(e_2) : \tau'} \qquad (Abs)\ \frac{C;\Gamma \,||\, v : \tau \vdash e : \tau'}{C;\Gamma \vdash \texttt{fn } v \texttt{ => } e : \tau \rightarrow \tau'}$$

$$(Mesg)\ \frac{C;\Gamma \vdash e : \texttt{PreObj}\, M}{C;\Gamma \vdash e\texttt{<-}m : M(m)} \qquad (New)\ \frac{C;\Gamma \vdash e : \texttt{Class}(t)(I\,;\,M)}{C;\Gamma \vdash \texttt{new } e : \texttt{Obj}(t)M}$$

$$(Self)\ \frac{\Gamma(s) = \texttt{Self}(I\,;\,t)}{C;\Gamma \vdash s : t} \qquad (Super)\ \frac{\Gamma(u) = \texttt{Super}(I\,;\,M)}{C;\Gamma \vdash u : \texttt{PreObj}\, M}$$

$$(SelfInst)\ \frac{\Gamma(s) = \texttt{Self}(I\,;\,t)}{C;\Gamma \vdash s.x : I(x)} \qquad (SupInst)\ \frac{\Gamma(u) = \texttt{Super}(I\,;\,M)}{C;\Gamma \vdash u.x : I(x)}$$

$$(SelfAssn)\ \frac{\Gamma(s) = \texttt{Self}(I\,;\,t) \quad C;\Gamma \vdash e : I(x)}{C;\Gamma \vdash s.x\texttt{:=}e : I(x)}$$

$$(Class)\ \frac{\dfrac{\dfrac{\dfrac{\dfrac{}{C;\Gamma \vdash e_i : \texttt{Class}(t_i)(I_i\,;\,M_i)}}{C' \vdash t \leq \texttt{PreObj}\, M_i[t/t_i]}}{C';\Gamma' \vdash e'_j : I(x_j)}}{C';\Gamma' \vdash e''_k : M(m_k)} \text{ where } \begin{cases} \Gamma' = \Gamma \,||\, s : \texttt{Self}(I\,;\,t) \,||\, \overline{u_i : (\texttt{Super}(I_i\,;\,M_i))[t/t_i]} \\ C' = C \,||\, t \leq \texttt{PreObj}\, M \\ \overline{I(x) = I_i(x)[t/t_i],\ \forall x \in dom(I_i)} \end{cases}}{C;\Gamma \vdash \texttt{class } s \texttt{ super } \overline{u_i \texttt{ of } e_i} \texttt{ inst } \overline{x_j = e'_j} \texttt{ meth } \overline{m_k = e''_k} : \texttt{Class}(t)(I\,;\,M)}$$

Figure 4. Loop Type Rules

an updateable method behaves more like an instance variable. Making all methods updateable in LOOP would hence be a design mistake.

Updating the method m of an object o' of type $\texttt{Obj}(t)M'$, generated by an extension of the above class to method signature M', is accomplished by $o'\texttt{<-set}m(f(o'))$ where the function f of type $(t{\rightarrow}\tau)[\texttt{Obj}(t)(\texttt{set}m\texttt{:}\tau{\rightarrow}\tau\texttt{,}\ M')/t]$ represents the new method code. The parameter to f is the external view of the object itself, since allowing f to have access to the instance variables would expose them indirectly. Note that the type of f is fixed to the type of the objects of this particular class, and so f cannot in general be used to update m of an object from a subclass. We expect it would be possible to define such an "open-ended" update function in SOOP, but LOOP's type system is currently too weak to express this.

2.2. LOOP Programming and Self

The purpose of this section is two-fold, one to give an example of programming in LOOP and typing LOOP programs, and two to explore in detail the problems involved in typing binary methods.

The main complexity of typing object-oriented languages arises from the interplay of subtyping and inheritance in the presence of binary methods. A first approximation to the typing of inheritance is the principle that inheritance is subtyping and subclasses correspond to subtypes. This principle is currently at the core of most of the commonly used typed object-oriented programming languages, including C++ and Object Pascal. However, problems arise in the presence of methods that take and/or return objects of "self-type," the type of the object containing the method itself [10], [3].

There are two common views of how "self" should be treated when typing a class expression. In one view, its type is *fixed* and thus denotes an object of the current class only. Alternatively, its type can be considered *open-ended*, meaning it denotes an object created from the current class *or* some future extension. It will be shown in this section how neither view subsumes the other.

To make this idea concrete, consider the example LOOP program in Figure 5. We define a class `Num`, with an instance variable `value` representing the value of a number, and methods `dec` to decrease this value, `isZero` to test whether it is zero, and `diff` to "destructively" compute the difference between `self` and another `Num`. Note that `Num` does not inherit from any existing classes, as indicated by the empty `super` declaration. The objects `n` and `n'` are instances of `Num`, created by the `new` operation. The class `CNum` extends `Num`, adding the new variable `cnt` and method `click`, and overriding `Num`'s `isZero` method. Within the body of the new `isZero` implementation the inherited version is accessed via `number<-isZero`, since in the `super` declaration `number` is bound to an object of class `Num`.

What are the proper LOOP types to give to these classes and objects? Let us denote by `NumObj` the type of `Num` objects `n` and `n'`. Intuitively, their method `diff` takes a `NumObj` as argument, and its result is also a `NumObj`. Since the methods are the only visible features of an object (cf. Section 2), `NumObj` should be

```
let Num = class self
  super
  inst
    value = 0
  meth
    dec = fn dummy => self.value:=pred(self.value),
    isZero = fn dummy => is_zero(self.value),
    diff = fn other =>
      if self<-isZero() then other
      else if other<-isZero() then self
      else (self<-dec(); other<-dec(); self<-diff(other))
in let n = new Num
in let n' = new Num
in let CNum = class self
  super
    number of Num
  inst
    cnt = 0,
    value = number.value
  meth
    click = fn dummy => self.cnt:=succ(self.cnt),
    dec = number<-dec,
    isZero = fn dummy => (self<-click(); number<-isZero()),
    diff = number<-diff
in let cn = new CNum
in let cn' = new CNum
in ...
```

Figure 5. `Num` and `CNum` Classes

$$\texttt{NumObj} \stackrel{\text{def}}{=} \texttt{Obj(SelfType)\{dec:1}\rightarrow\texttt{Nat, isZero:1}\rightarrow\texttt{Bool,}$$
$$\texttt{diff:SelfType}\rightarrow\texttt{SelfType\}}$$

The bound type variable `SelfType` refers to "the type of the object itself" — and indeed if `n` is of type `NumObj` then `n<-diff` can be observed to be of type `NumObj`$\rightarrow$`NumObj` by applying rules (*Mesg*), (*Sub*), and (*Fix*) of the Loop type system.

We now present the two views of typing "self" in classes, first the open-ended view and then the fixed view.

2.2.1. The open-ended-self typing

First, we consider what type the class expression `Num` should be given in the open-ended view, the view most naturally representable in Loop. One possible type to give `Num` in Loop is

```
NumClass ≝ Class(SelfType)({value:Nat};
                           {dec:1→Nat, isZero:1→Bool,
                            diff:SelfType→SelfType})
```

This typing follows from Loop's (*Class*) typing rule as follows. The first two hypotheses of the (*Class*) rule are vacuous since there are no superclasses. The most interesting aspect of the typing proof is how the `diff` method is typechecked as part of the fourth hypothesis of the rule. For this method, under the assumption `other` is of type `SelfType`, we must show the body is of type `SelfType`. The constraint system has added constraint `SelfType` $\leq$ `PreObj` M for M being the methods of `NumClass`, and there is an added assumption that `self` is of type `Self(`I`;`t`)` for I being the instances of `NumClass`. Consider the most problematic message send, `self <- diff(other)`. Using the (*Self*) rule and aforementioned assumption we may conclude `self` is of type `SelfType`. With (*Sub*) using the constraint `SelfType` $\leq$ `PreObj` M, `self` is proved to be of type `PreObj` M. Thus by (*Mesg*), `diff` is of type `SelfType→SelfType`, and since `other` is of type `SelfType` by assumption, the message send type-checks with result type `SelfType`.

Using the (*New*) rule, the objects `n` and `n'` are of type `NumObj`, as expected.

As alluded to at the beginning of this section, the difficult question is the treatment of the type of the binary method `diff`. Here we give it the function type `SelfType→SelfType`, but what exactly is `SelfType`? The methods defined for `Num` objects may be inherited in subsequent extensions of this class (subclasses), hence the type system must ensure that these methods operate properly even when applied to objects of these subclasses. For this reason the object denoted by `self` within the body of the class definition must be considered as generated by either the class being defined, or by one of its descendants. Thus the type `SelfType` is not fixed when type checking the class definition. All we know is that `self` responds to the messages that may be sent objects in any class extending `Num` objects, and these may define additional methods. In Loop we take an *open-ended, parametric* view of `SelfType`: it is the type of "self" at object creation time, and thus the object by then could contain more methods via inheritance.

The subclass `CNum` can be given the Loop type

```
CNumClass ≝ Class(SelfType)({value, cnt:Nat};
                            {click, dec:1→Nat, isZero:1→Bool,
                             diff:SelfType→SelfType})
```

by the (*Class*) rule. Note the typing of this subclass will require showing `Num` is of type `NumClass` (the first hypothesis of the (*Class*) rule), which was previously shown. The added assumption that `number` has a `Self` type with the methods and instances of `NumClass` allows the inherited methods `dec` and `diff` to be shown to be of the proper type. The generated objects by (*New*) are then of type

```
CNumObj ≝ Obj(SelfType){click,dec:1→Nat, isZero:1→Bool,
                        diff:SelfType→SelfType}
```

2.2.2. The fixed-self typing

This is not the only typing that may be given to the example in LOOP. It is possible also to *fix* the type of "self" to only contain fields of an object of the current class. Taking this view, we can give the `diff` method the type `NumObj`→`NumObj`, and give the class expression `Num` the type

```
FixNumClass ≝ Class(SelfType)({value:Nat};
                              {dec:1→Nat, isZero:1→Bool,
                               diff:NumObj→NumObj})
```

where the open-ended `SelfType` is never used. A `Num` object generated using the `FixNumClass` typing for `Num` has the type `NumObj`, the same type as `NumClass` objects. Note however that the (*New*) rule applied to `Num` of type `FixNumClass` will produce an object not precisely of type `NumObj`, but in a once-unrolled form. The (*Fold*) rule will then be needed to show `Num` is of type `NumObj`. The (*Fold*) rule is thus critical to typing programs in the fixed-self view. The type of `CNum` can be proven to be

```
CFixNumClass ≝ Class(SelfType)({value,cnt:Nat};
                               {click,dec:1→Nat, isZero:1→Bool,
                                diff:NumObj→NumObj})
```

and the generated objects `cn` and `cn'` are of type

```
CFixNumObj ≝ Obj(SelfType){click,dec:1→Nat, isZero:1→Bool,
                           diff:NumObj→NumObj}
```

Observe `CFixNumObj` is the same as `NumObj` except that it has an extra `click` method; in particular the `diff` methods have identical types. This means that `CFixNumObj` is a subtype of `NumObj` according to the subtyping rules (the (*Fold*), (*Fix*), and (*PreObj*) rules in particular). In general, in the fixed-self view subclass objects will always be of subtypes of superclass objects' types, so inheritance *is* subtyping. Also, observe that this typing would not allow `diff` to be overridden if the new `diff` sent `other` a `click` message. The type of `diff` is fixed at a point in the hierarchy. This is the first problem with the fixed-self view: some forms of method override are disallowed.

2.2.3. Inheritance vs subtyping

Consider the following two possible conclusions to the example program:

(1) `in (cn<-diff(cn'))<-click()`,

(2) `in (cn<-diff(n))<-isZero()`.

(1) is type-correct using the open-ended type for `self`. In this case, `diff` has type `CNumObj`→`CNumObj`, and `cn'` has type `CNumObj`, so the application is sound and the result is of type `CNumObj` and thus can respond to a `click` message.

However, (1) will not typecheck using the fixed type for `self`: there, `diff` has type `NumObj`→`NumObj` and `cn'` has type `CFixNumObj`. Now, since `CFixNumObj` is a subtype of `NumObj` (as mentioned above), `cn'` also has type `NumObj` by (*Sub*), and hence cn<-diff(cn') is well-typed. Unfortunately, the result is of type `NumObj`, so the `click` method of `cn'` is lost and cannot be invoked (but note that the invocation `(cn<-diff(cn'))<-isZero()` would type-check in the fixed view).

In contrast, the opposite situation occurs in (2). Under the open-ended typing of `self`, `cn` is of type `CNumObj` so `diff` has type `CNumObj`→`CNumObj`. However, `n` is only of type `NumObj`, and `NumObj` $\not\leq$ `CNumObj` since `SelfType` occurs negatively as alluded to above. Thus, cn<-diff(n) is ill-typed. Clearly (2) type-checks in the fixed-self view, since in this view inheritance is subtyping: the `diff` method of `cn` accepts arguments of type `NumObj`.

2.2.4. Conclusions

What this example shows is that neither the pure fixed- nor open-self typing scheme is completely adequate; there is a tension between allowing more inheritance and allowing more subtyping. We believe therefore it may be best to let the programmer choose which scheme is appropriate on a case-by-case basis. If a method only *takes* objects of the "self" type and is never overridden, the fixed-self typing will always be adequate; the open-ended typing can be used whenever no subsumption of object types up the inheritance hierarchy is needed.

The fixed and open schemes are only two extreme points of a continuum of options. It is possible to create a class hierarchy that gives "self" initially an open-ended type, but then at some point down the hierarchy fixes it and keeps it fixed below that point. In the lower portions of the hierarchy, objects may then be "lifted" freely up the inheritance tree. In addition, a single class type may contain multiple occurrences of "self-type"; some can be fixed at this level while others remain open-ended. In particular, it is possible to have positive occurrences of `SelfType` be open-ended and negative occurrences fixed, preserving some open-endedness and at the same time conforming to the subclasses-generate-subtypes principle.

The terms *covariance* and *contravariance* are sometimes used in the literature when discussing this topic [15]. The open-ended "self" is a *covariant* view of method argument types (upon subclassing they may become objects with more methods), whereas the fixed-self view is a *contravariant* view (in subclasses method argument types may be replaced by supertypes, in accordance with rule (*Fun*); in our example we kept the types invariant).

3. The Soop Language, Semantics and Soundness

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. Let x range over the countably infinite set of variables *Var* and l range over the countable set of labels *Lab*. The values *Val* and expressions *Exp* of SOOP are the least inductive collections defined as follows:

$$\begin{array}{rcl} \mathit{Val} \ni v & ::= & x \mid n \mid b \mid \lambda x.\, e \mid \{\overline{l_i = v_i}\} \\ \mathit{Exp} \ni e & ::= & v \mid e(e) \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{is\_zero}(e) \mid \mathsf{succ}(e) \mid \mathsf{pred}(e) \\ & & \mid \{\overline{l_i = e_i}\} \mid e.l \mid \mathsf{ref}\ e \mid !\ e \mid \mathsf{set}\,(e,\, e) \end{array}$$

where $n \in \mathit{Nat} \stackrel{\text{def}}{=} \{0,\, 1,\, \ldots\}$ and $b \in \mathit{Bool} \stackrel{\text{def}}{=} \{\mathsf{true},\, \mathsf{false}\}$.

In addition to the basic types and type constructors, the type system of SOOP provides recursive types and polymorphic function types:

$$\begin{array}{rcl} \mathit{Typ} \ni \theta, \tau & ::= & t \mid \mathsf{Nat} \mid \mathsf{Bool} \mid \{\overline{l_i : \tau_i}\} \mid \tau\ \mathsf{Ref} \mid \mu t.\, \tau \mid \varphi \\ \mathit{FnTyp} \ni \varphi & ::= & \tau \text{->} \tau' \mid \forall[\overline{t_i \leq \theta_i}].\, \varphi \end{array}$$

where t and t_i range over the countable set *TVar* of type variables. All labels of a record type must be distinct; two record types are considered identical if they only differ in the order of their label-type pairs. The type expression $\forall[\overline{t_i \leq \theta_i}].\, \varphi$ binds the type variables t_i in each of their *upper bounds* θ_i as well as in the body φ; this allows a form of F-bounded polymorphism [7]. A *recursive type* is of the form $\mu t.\, \tau$; the type variable t is bound in τ. By convention $\forall$ and μ have lower precedence than the arrow and Ref. We use $FTV(\tau)$ to denote the set of free type variables of the type expression τ.

The SOOP types are partially ordered by the subtyping relation $\leq$; the following definitions will be necessary for its introduction. As in LOOP, a *constraint system* C is a finite function in $\mathit{Con} \stackrel{\text{def}}{=} \mathit{TVar} \rightarrow \mathit{Typ}$, mapping type variables to their respective upper bounds. A type expression τ is *closed* in C if $FTV(\tau) \subseteq \mathit{dom}(C)$; C is *consistent* if $C(t)$ is closed in C for all $t \in \mathit{dom}(C)$. Note that the requirement for consistency allows type variables to appear free in their own bounds.

The valid *subtyping judgements* are triples from $\mathit{Con} \times \mathit{Typ} \times \mathit{Typ}$, written in the form $C \vdash \tau \leq \tau'$, which for a consistent C and closed in it τ and τ' can be derived from the rules in Figure 6. This system is an extension of the standard record subtyping [8] with recursive and F-bounded polymorphic types.

SOOP is not a second order calculus (there are no type abstractions and applications), therefore we use subtyping to instantiate the type of a polymorphic function. The rules for subtyping polymorphic types reflect the view that the set of values (ideal [20]) corresponding to a polymorphic type is the intersection of the sets corresponding to all of its instances. Thus rule (INST) specifies that a polymorphic type is a subtype of its instances with all of the top level type variables t_i replaced simultaneously by type expressions τ_i, provided the latter can be shown to be subtypes of the corresponding bounds θ_i instantiated with the same substitution σ. Rule (GEN) provides for e.g. comparing two polymorphic types; the intuition behind the rule is that a type φ is a subtype of a polymorphic type τ if φ is a subtype of all instances of τ. The standard rule for (F-)bounded quantification [23]

$$\text{(S-ALL)} \quad \frac{C \| t \leq \theta_2 \vdash t \leq \theta_1, \quad C \| t \leq \theta_2 \vdash \tau_1 \leq \tau_2, \quad t \notin FTV(C)}{C \vdash \forall[t \leq \theta_1].\, \tau_1 \leq \forall[t \leq \theta_2].\, \tau_2}$$

$$(\textsc{refl})\ \frac{}{C \vdash \tau \le \tau} \qquad (\textsc{record})\ \frac{C \vdash \overline{\tau_i \le \tau_i''}}{C \vdash \{\overline{l_i : \tau_i},\ \overline{l_j' : \tau_j'}\} \le \{\overline{l_i : \tau_i''}\}}$$

$$(\textsc{trans})\ \frac{C \vdash \tau \le \tau', \quad \tau' \le \tau''}{C \vdash \tau \le \tau''} \qquad (\textsc{fold})\ \frac{}{C \vdash \tau[\mu t.\,\tau/t] \le \mu t.\,\tau}$$

$$(\textsc{var})\ \frac{t \in dom(C)}{C \vdash t \le C(t)} \qquad (\textsc{fix})\ \frac{C \| t \le \theta \vdash \tau \le \theta, \quad \tau \downarrow t, \quad t \notin FTV(C)}{C \vdash \mu t.\,\tau \le \theta[\mu t.\,\tau/t]}$$

$$(\textsc{fun})\ \frac{C \vdash \tau_1' \le \tau_1, \quad \tau_2 \le \tau_2'}{C \vdash \tau_1 \text{->} \tau_2 \le \tau_1' \text{->} \tau_2'} \qquad (\textsc{inst})\ \frac{C \vdash \overline{\tau_i \le \sigma\theta_i} \text{ where } \sigma = [\overline{\tau_i/t_i}]}{C \vdash \forall[\overline{t_i \le \theta_i}].\,\varphi \le \sigma\varphi}$$

$$(\textsc{gen})\ \frac{C \| \overline{t_i \le \theta_i} \vdash \varphi \le \varphi', \quad \{\overline{t_i}\} \cap (FTV(C) \cup FTV(\varphi)) = \emptyset}{C \vdash \varphi \le \forall[\overline{t_i \le \theta_i}].\,\varphi'}$$

Figure 6. Soop subtyping rules.

is derived in Soop by

(1) applying (INST) to the derivation of $C \| t \le \theta_2 \vdash t \le \theta_1$, thus proving $C \| t \le \theta_2 \vdash \forall[t \le \theta_1].\,\tau_1 \le \tau_1$
(2) applying (TRANS) to the previous result and $C \| t \le \theta_2 \vdash \tau_1 \le \tau_2$, and
(3) applying (GEN) to the conclusion of (TRANS).

The subtyping rules for recursive types are (FOLD) and (FIX). In the latter, τ must be *contractive* in t (in the sense of [20]), written $\tau \downarrow t$; in our system this means that τ cannot be of the form $\mu t_1.\ldots\mu t_m.\,t$ for any $m \ge 0$. Note that a rule for unfolding recursive types (proving $C \vdash \mu t.\,\tau \le \tau[\mu t.\,\tau/t]$) is derivable from (FIX) for $\theta = \tau$. Thus we establish that a recursive type is equivalent to its unrolling with respect to subtyping; this relation is weaker than the type equivalence defined by Amadio and Cardelli in [2], and as a result some judgements, provable in the system of [2], are not provable in Soop (e.g. $C \vdash \mu$t. Nat -> t $\le \mu$t. Nat -> Nat -> t). However (FIX) is stronger than the rule for subtyping recursive types of [2]

$$(\mu)\ \frac{C \| t \le t,\ t' \le t \vdash \tau' \le \tau, \quad t \notin FTV(\tau'),\ t' \notin FTV(\tau),\ \{t, t'\} \cap FTV(C) = \emptyset}{C \vdash \mu t'.\,\tau' \le \mu t.\,\tau}$$

In fact, (μ) is a meta-rule in the Soop type system: each valid subtyping derivation using (μ) can be transformed by

(1) replacing t by $\mu t.\,\tau$ in the derivation of the antecedent of (μ), producing a proof of $C \| t' \le \mu t.\,\tau \vdash \tau' \le \tau[\mu t.\,\tau/t]$;
(2) applying (FOLD) and (TRANS) to obtain $C \| t' \le \mu t.\,\tau \vdash \tau' \le \mu t.\,\tau$;
(3) applying (FIX).

Furthermore (FIX) can be used to prove subtyping between recursive types with negative occurrences of the bound variable, which is impossible with (μ) alone, e.g.

$$C \vdash \mu\text{t. \{f: \{x: Nat\} -> t, x: Nat\}} \le \mu\text{t. \{f: t -> t, x: Nat\}}$$

Types are assigned to Soop terms by the system of proof rules in Figure 7. Here

$$\text{(Sub)}\quad \frac{C;\Gamma\vdash e:\tau \quad C\vdash \tau\le\tau'}{C;\Gamma\vdash e:\tau'} \qquad \text{(IsZero)}\quad \frac{C;\Gamma\vdash e:\mathsf{Nat}}{C;\Gamma\vdash \mathsf{is\_zero}(e):\mathsf{Bool}}$$

$$\text{(Var)}\quad \frac{x\in dom(\Gamma)}{C;\Gamma\vdash x:\Gamma(x)} \qquad \text{(Succ)}\quad \frac{C;\Gamma\vdash e:\mathsf{Nat}}{C;\Gamma\vdash \mathsf{succ}(e):\mathsf{Nat}}$$

$$\text{(Nat)}\quad \frac{}{C;\Gamma\vdash n:\mathsf{Nat}} \qquad \text{(Pred)}\quad \frac{C;\Gamma\vdash e:\mathsf{Nat}}{C;\Gamma\vdash \mathsf{pred}(e):\mathsf{Nat}}$$

$$\text{(Bool)}\quad \frac{}{C;\Gamma\vdash b:\mathsf{Bool}} \qquad \text{(Ref)}\quad \frac{C;\Gamma\vdash e:\tau}{C;\Gamma\vdash \mathsf{ref}\ e:\tau\ \mathsf{Ref}}$$

$$\text{(Record)}\quad \frac{C;\Gamma\vdash \overline{e_i:\tau_i}}{C;\Gamma\vdash \{\overline{l_i=e_i}\}:\{\overline{l_i:\tau_i}\}} \qquad \text{(Deref)}\quad \frac{C;\Gamma\vdash e:\tau\ \mathsf{Ref}}{C;\Gamma\vdash\ !\ e:\tau}$$

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

$$\text{(Cond)}\quad \frac{C;\Gamma\vdash e:\mathsf{Bool},\quad e_1:\tau,\quad e_2:\tau}{C;\Gamma\vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2:\tau} \qquad \text{(App)}\quad \frac{C;\Gamma\vdash e_1:\tau\text{->}\tau',\quad e_2:\tau}{C;\Gamma\vdash e_1(e_2):\tau'}$$

$$\text{(Abs)}\quad \frac{C\|\overline{t_i\le\theta_i};\ \Gamma\|x:\tau\vdash e:\tau',\quad \{\overline{t_i}\}\cap FTV(C)=\emptyset}{C;\Gamma\vdash \lambda x.\,e:\forall[\overline{t_i\le\theta_i}].\,\tau\text{->}\tau'}$$

Figure 7. Soop typing rules.

a *type environment* Γ is a finite map in $Env \stackrel{\text{def}}{=} Var \to Typ$; it is *closed* in a constraint system C if $\Gamma(x)$ is closed in C for each $x \in dom(\Gamma)$. The valid *typing judgements* derived by these rules are quadruples from $Con \times Env \times Exp \times Typ$ of the form $C;\Gamma\vdash e:\tau$ where C is consistent and Γ is closed in C. Since we only allow function types to be polymorphic, rule (Abs) provides also for type abstraction.

The type system of Soop is powerful enough to allow for a type-correct translation of Loop programs (and more); yet it was weakened to not include certain types and proof rules which, if added, would further complicate the proof of soundness of the system. Some restrictions are:

- Only functions are allowed to be of polymorphic types. Extending polymorphism over records is not a conceptual problem, but note that a polymorphic record of functions would be equivalent to a record of polymorphic functions; similarly a recursive type could be unfolded under the universal quantifier to get an equivalent type. We impose this restriction in order to avoid having polymorphic Ref types of the form $\forall[t\le\theta].\,\tau$ Ref with t free in τ; this type cannot be given to any value in a sound system. For the same reason we combine type generalization with abstraction, instead of using a separate generalization rule — the abstraction prevents evaluation, and frozen references can be polymorphic, however e.g. a field of a record should not be of a polymorphic Ref type. It is possible to have a more flexible language and still avoid these types; we believe this would be a considerable complication with little reward. Also excluded from the language are "bottom" types (e.g. $\forall[t\le\theta].\,t$), whose pres-

ence would introduce a number of special cases in the most technical part of our proof, and would not affect noticeably other aspects of the language.

- The omission of rules to distribute quantifiers over type constructors, e.g.

$$C \vdash \ \forall[\overline{t_i \leq \theta_i}].\, \tau \text{->} \varphi \ \leq \ \tau \text{->} \forall[\overline{t_i \leq \theta_i}].\, \varphi, \quad \text{if } \{\overline{t_i}\} \cap FTV(\tau) = \emptyset$$

  Since the reversed subtyping relation between these types is provable in Soop using rule (GEN), they would be equivalent if this rule was adopted.

Type inference in Soop seems unlikely, considering the problems in similar systems [23]; the possibility of having it in versions of the language is the topic of ongoing research.

3.1. Operational Semantics of Soop

We present semantics of Soop in the general framework of [14], [21], [11], [26]. The operational interpreter of the language is specified by representing the notion of computation as a binary relation between Soop terms in memory environments. As in [26] the memory is treated as a function from variables to values.

We use the notation $[\, x \mapsto v \,]$ for the map defined only on x with value v, and $f||g$ for the functional extension of f by g.

A *memory* Σ is a finite map in $Mem \stackrel{\text{def}}{=} Var \rightarrow Val$; if $x \in dom(\Sigma)$, then x is the identifier of a reference cell, and $\Sigma(x)$ is the value contained in the cell. Define $FV(\Sigma) = \bigcup_{x \in dom(\Sigma)} FV(\Sigma(x))$.

A *computation state* $\langle \Sigma,\, e \rangle$ is a pair of a memory and an expression; it is *closed* if $FV(\Sigma) \cup FV(e) \subseteq dom(\Sigma)$.

A *reduction context* R is recursively defined as

$$\begin{aligned} R \ ::= \ & \circ \mid R(e) \mid v(R) \mid \mathsf{if}\ R\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{is\_zero}(R) \mid \mathsf{succ}(R) \mid \mathsf{pred}(R) \\ & \mid \{l_1 = v_1,\, \ldots,\, l_{k-1} = v_{k-1},\, l_k = R,\, l_{k+1} = e_{k+1},\, \ldots,\, l_m = e_m\} \mid R.l \\ & \mid \mathsf{ref}\ R \mid\ !\ R \mid \mathsf{set}\,(R,\, e) \mid \mathsf{set}\,(v,\, R) \end{aligned}$$

and $R[e]$ is the result of substituting e for $\circ$ in R. The reduction context isolates the subterm of the expression in which the next reduction step is to be performed.

The *single-step computation relation* $\mapsto_1$ is the least binary relation on $Mem \times Exp$ which relates all computation states of the forms shown in Figure 8, where R is a reduction context, and $[\![\cdot]\!] \in Nat \rightarrow \mathbb{N}$ is the obvious interpretation of the numerals; the arguments to R in the left column are *redexes*.

The *computation relation* $\mapsto^*$ is the reflexive transitive closure of $\mapsto_1$. A closed computation state $\langle \Sigma,\, e \rangle$ is *stuck* 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 *gets stuck* if $\mapsto^*$ relates it to a stuck state. A state $\langle \Sigma,\, e \rangle$ *diverges* if 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 $\langle \Sigma',\, e' \rangle \mapsto_1 \langle \Sigma'',\, e'' \rangle$. An example of a stuck state is $\langle \Sigma,\, 3(5) \rangle$; a diverging state is $\langle \Sigma,\, (\lambda x.\, x(x))(\lambda x.\, x(x)) \rangle$ — it is related to itself by $\mapsto_1$, thus representing a non-terminating computation.

$$
\begin{aligned}
\langle \Sigma,\ R[(\lambda x.\, e)(v)] \rangle &\mapsto_1 \langle \Sigma,\ R[e[v/x]] \rangle \\
\langle \Sigma,\ R[\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2] \rangle &\mapsto_1 \langle \Sigma,\ R[e_1] \rangle \\
\langle \Sigma,\ R[\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2] \rangle &\mapsto_1 \langle \Sigma,\ R[e_2] \rangle \\
\langle \Sigma,\ R[\mathsf{is\_zero}(0)] \rangle &\mapsto_1 \langle \Sigma,\ R[\mathsf{true}] \rangle \\
\langle \Sigma,\ R[\mathsf{is\_zero}(n)] \rangle &\mapsto_1 \langle \Sigma,\ R[\mathsf{false}] \rangle, && \text{if } n \neq 0 \\
\langle \Sigma,\ R[\mathsf{succ}(n)] \rangle &\mapsto_1 \langle \Sigma,\ R[n'] \rangle, && \text{where } [\![n']\!] = [\![n]\!] + 1 \\
\langle \Sigma,\ R[\mathsf{pred}(n)] \rangle &\mapsto_1 \langle \Sigma,\ R[n'] \rangle, && \text{where } [\![n']\!] = [\![n]\!] \dot{-} 1 \\
\langle \Sigma,\ R[\{\overline{l_i = v_i}\}.l_k] \rangle &\mapsto_1 \langle \Sigma,\ R[v_k] \rangle, && \text{if } k \in \{1, \ldots, m\} \\
\langle \Sigma,\ R[\mathsf{ref}\ v] \rangle &\mapsto_1 \langle \Sigma[\![\, x \mapsto v \,]\!],\ R[x] \rangle, && \text{where } x \notin dom(\Sigma) \\
\langle \Sigma,\ R[!\ x] \rangle &\mapsto_1 \langle \Sigma,\ R[\Sigma(x)] \rangle, && \text{if } x \in dom(\Sigma) \\
\langle \Sigma,\ R[\mathsf{set}\,(x,\ v)] \rangle &\mapsto_1 \langle \Sigma[\![\, x \mapsto v \,]\!],\ \Xi \rangle R[v], && \text{if } x \in dom(\Sigma)
\end{aligned}
$$

Figure 8. The single-step computation relation.

Directly from these definitions and by induction on the structure of terms we have the following Lemmas.

LEMMA 1 *If* $\langle \Sigma,\ e \rangle \mapsto_1 \langle \Sigma',\ e' \rangle$ *and* $\langle \Sigma,\ e \rangle$ *is closed, then* $\langle \Sigma',\ e' \rangle$ *is closed.*

LEMMA 2 (DETERMINISTIC COMPUTATION) $\mapsto_1$ *defines a partial function on the closed computation states.*

COROLLARY 1 *A computation state computes to a value, or diverges, or gets stuck.*

3.2. Soundness of the SOOP Type System

Following [26] we establish soundness of the type system of SOOP with respect to the computation relations by proving that a subject reduction property holds for this type system, and that the stuck states cannot be given types. The first implies that the type of a computation state is preserved by the single-step computation relation; together with the second this means that the stuck states are unreachable from a typable initial state.

In comparison with proofs of subject reduction for the λ-calculus [25] and ML-style polymorphic languages [26], the proof for the SOOP type system is complicated by the rich subtyping relation. The only analog of subtyping in the ML type system is that an expression of a polymorphic type scheme is also of each type produced by any instantiation of the type variables of the scheme. This allows a relatively straightforward proof of subject reduction for the β-conversion rewriting rules. In contrast the instantiation of a polymorphic type in SOOP is only correct if the substituted types satisfy certain subtyping relations; as a result the typing derivation of the β-contractum of an expression e can be produced only after a significant transformation of the typing derivation of e. Further complications are introduced by the non-trivial subtyping on most of the basic SOOP types.

$$
\begin{array}{rcl}
s & ::= & \textsc{var}\ t \mid \textsc{fun}\ (S,\ S) \mid \textsc{record}\ (\langle\overline{l_i}\rangle,\ \langle\overline{l'_j}\rangle,\ \langle\overline{S_i}\rangle,\ \langle\overline{\tau'_j}\rangle) \\
 & & \mid \textsc{fold}\ (t,\ \tau) \mid \textsc{fix}\ (t,\ \theta,\ S) \\
 & & \mid \textsc{inst}\ (\langle\overline{t_i}\rangle,\ \langle\overline{\theta_i}\rangle,\ \varphi,\ \langle\overline{S_i}\rangle) \mid \textsc{gen}\ (\langle\overline{t_i}\rangle,\ \langle\overline{\theta_i}\rangle,\ S) \\
\mathit{SubDer} \ni S & ::= & \textsc{refl}\ \tau \mid \langle\overline{s_i}\rangle \text{ where } i \in \{1, \ldots, m\},\ m \geq 1 \\
d & ::= & \textsc{Var}\ x \mid \textsc{Nat}\ n \mid \textsc{Bool}\ b \mid \textsc{Record}\ (\langle\overline{l_i}\rangle,\ \langle\overline{D_i}\rangle) \mid \textsc{Sel}\ (l,\ D) \\
 & & \mid \textsc{Cond}\ (D,\ D,\ D) \mid \textsc{IsZero}\ D \mid \textsc{Succ}\ D \mid \textsc{Pred}\ D \\
 & & \mid \textsc{Ref}\ D \mid \textsc{Deref}\ D \mid \textsc{Set}\ (D,\ D) \\
 & & \mid \textsc{App}\ (D,\ D) \mid \textsc{Abs}\ (C,\ x,\ \tau,\ D) \\
\mathit{TypDer} \ni D & ::= & \langle d,\ S\rangle
\end{array}
$$

Figure 9. The language of subtyping and typing derivations.

To overcome these difficulties we employ a technique of explicit proof manipulation. We introduce a language of typing and subtyping derivations for Soop programs, and its semantics in terms of validity of a derivation and the elements of its conclusion. After establishing some properties of the derivations we apply them in proving the subject reduction theorem. The proof of the main soundness result concludes this section.

3.2.1. Canonical Forms of Derivations

We begin by representing the proof derivations in canonical forms, encoded as expressions of the language shown in Figure 9.

Most of the rules in Figure 6 have their obvious counterparts, e.g. the *SubDer* expression $\langle\textsc{fun}\ (S_1,\ S_2)\rangle$ represents the subtyping derivation resulting from applying rule (FUN) to the two derivations represented in *SubDer* by S_1 and S_2. The $\overline{l_i}$ and $\overline{S_i}$ elements of the RECORD expression correspond to the labels of the "larger" type in the conclusion of rule (RECORD) and the derivations of their respective types (the antecedents), while $\overline{l'_j}$ and $\overline{\tau'_j}$ are the "extra" labels and types of the "smaller" type. There is no construct corresponding to (TRANS); if a derivation has a non-empty subtree of applications of (TRANS) at its root, the corresponding *SubDer* expression is the sequence of derivations found at the leaves of this subtree, each with an application of a rule other than (TRANS) and (REFL) at its root — since (TRANS) is associative, the exact way in which the subtree is formed is immaterial, and any (REFL) can be removed from the sequence, unless it is the only element.

The only typing rule with no direct correspondence is (Sub). It can be applied to any Soop term; we therefore choose to only consider typing derivations in which it is applied exactly once to each subterm. For each derivation we can find an equivalent one (i.e. one proving the same judgement) with this property; in particular it suffices to insert an application of (Sub) (with an instance of (REFL) for the subtyping antecedent) for each subterm where there is none, and combine subsequent applications of (Sub) into one, with the subtyping antecedent being the transitive

s	$\langle\!\langle s \rangle\!\rangle_C$	$[\![s]\!]_C$
VAR t	t	$C(t)$
FUN $(S_1,\ S_2)$	$[\![S_1]\!]_C$ -> $\langle\!\langle S_2 \rangle\!\rangle_C$	$\langle\!\langle S_1 \rangle\!\rangle_C$ -> $[\![S_2]\!]_C$
RECORD $(\langle \overline{l_i} \rangle,\ \langle \overline{l'_j} \rangle,\ \langle \overline{S_i} \rangle,\ \langle \overline{\tau'_j} \rangle)$	$\{\overline{l_i : \langle\!\langle S_i \rangle\!\rangle_C},\ \overline{l'_j : \tau'_j}\}$	$\{\overline{l_i : [\![S_i]\!]_C}\}$
FOLD $(t,\ \tau)$	$\tau[\mu t.\, \tau / t]$	$\mu t.\, \tau$
FIX $(t,\ \theta,\ S)$	$\mu t.\, \langle\!\langle S \rangle\!\rangle_{C \| t \leq \theta}$	$\theta[\mu t.\, [\![S]\!]_{C \| t \leq \theta} / t]$
INST $(\langle \overline{t_i} \rangle,\ \langle \overline{\theta_i} \rangle,\ \varphi,\ \langle \overline{S_i} \rangle)$	$\forall[\overline{t_i \leq \theta_i}].\, \varphi$	$\varphi[\overline{\langle\!\langle S_i \rangle\!\rangle_C / t_i}]$
GEN $(\langle \overline{t_i} \rangle,\ \langle \overline{\theta_i} \rangle,\ S)$	$\langle\!\langle S \rangle\!\rangle_C$	$\forall[\overline{t_i \leq \theta_i}].\, [\![S]\!]_{C \| \overline{t_i \leq \theta_i}}$

$$\langle\!\langle \text{REFL } \tau \rangle\!\rangle_C = \tau \qquad [\![\text{REFL } \tau]\!]_C = \tau$$
$$\langle\!\langle \langle s_1,\ \ldots,\ s_m \rangle \rangle\!\rangle_C = \langle\!\langle s_1 \rangle\!\rangle_C \qquad [\![\langle s_1,\ \ldots,\ s_m \rangle]\!]_C = [\![s_m]\!]_C$$

Figure 10. Reconstructing types from subtyping derivations.

sequence of the respective premises. This leads to a proof tree representable in *TypDer*: at each node there is an application of one typing rule determined by the syntax of the Soop term, and a subtyping derivation.

Not all elements of *SubDer* or *TypDer* correspond to valid derivations. To define the latter we introduce partial functions recovering the information in the conclusion of a derivation, given the context (for subtyping — the constraint system, for typing — also the environment). Thus $\langle\!\langle S \rangle\!\rangle_C$ is the "smaller" type in the conclusion of the subtyping derivation corresponding to S with constraint system C, while $[\![S]\!]_C$ is the "larger" type; similarly $[\![D]\!]^\Gamma_C$ is the type in the conclusion of D in context $C; \Gamma$, and $\mathcal{E}(D)$ is the Soop term whose type is derived by D. To avoid extra notation we define the respective functions also on the subterms s and d of *SubDer* and *TypDer* expressions. The definitions are in Figures 10 and 11 respectively.

The validity of a derivation is formally determined by the rules given in Figures 12 and 13, where the meaning of $C \triangleright S$ is "S is a valid subtyping derivation in context C", and similarly for ${}^\Gamma_C \triangleright D$.

In summary we have the following Lemma.

LEMMA 3

(i) *$C \vdash \tau \leq \tau'$ is a provable subtyping judgement if and only if there exists $S \in$ SubDer such that $C \triangleright S$ and $\langle\!\langle S \rangle\!\rangle_C = \tau$ and $[\![S]\!]_C = \tau'$.*

(ii) *$C; \Gamma \vdash e : \tau$ is a valid typing judgement if and only if there exists $D \in$ TypDer such that ${}^\Gamma_C \triangleright D$ and $\mathcal{E}(D) = e$ and $[\![D]\!]^\Gamma_C = \tau$.*

Example: The subtyping derivation

d	$\mathcal{E}(d)$	$[\![d]\!]^\Gamma_C$
VAR x	x	$\Gamma(x)$
NAT n	n	Nat
BOOL b	b	Bool
RECORD $(\langle\overline{l_i}\rangle, \langle\overline{D_i}\rangle)$	$\{\overline{l_i = \mathcal{E}(D_i)}\}$	$\{\overline{l_i : [\![D_i]\!]^\Gamma_C}\}$
SEL (l, D)	$\mathcal{E}(D).l$	τ if $[\![D]\!]^\Gamma_C = \{l:\tau\}$
COND (D_1, D_2, D_3)	if $\mathcal{E}(D_1)$ then $\mathcal{E}(D_2)$ else $\mathcal{E}(D_3)$	$[\![D_3]\!]^\Gamma_C$
ISZERO D	is_zero$(\mathcal{E}(D))$	Bool
SUCC D	succ$(\mathcal{E}(D))$	Nat
PRED D	pred$(\mathcal{E}(D))$	Nat
REF D	ref $\mathcal{E}(D)$	$[\![D]\!]^\Gamma_C$ Ref
DEREF D	! $\mathcal{E}(D)$	τ if $[\![D]\!]^\Gamma_C = \tau$ Ref
SET (D_1, D_2)	set$(\mathcal{E}(D_1), \mathcal{E}(D_2))$	$[\![D_2]\!]^\Gamma_C$
APP (D_1, D_2)	$\mathcal{E}(D_1)(\mathcal{E}(D_2))$	τ if $[\![D_1]\!]^\Gamma_C = [\![D_2]\!]^\Gamma_C$ -> τ
ABS $(\overline{t_i \le \theta_i}, x, \tau, D)$	$\lambda x.\, \mathcal{E}(D)$	$\forall[\overline{t_i \le \theta_i}].\, \tau$ -> $[\![D]\!]^{\Gamma\| x:\tau}_{C\|\overline{t_i \le \theta_i}}$
$\mathcal{E}(\langle d, S\rangle) = \mathcal{E}(d)$		$[\![\langle d, S\rangle]\!]^\Gamma_C = [\![S]\!\!]_C$

Figure 11. Reconstructing terms and types from typing derivations.

$$
\begin{array}{rl}
C \triangleright \textsc{var}\ t & \text{iff } t \in dom(C) \\
C \triangleright \textsc{fun}\ (S_1, S_2) & \text{iff } C \triangleright S_1 \text{ and } C \triangleright S_2 \\
C \triangleright \textsc{record}\ (\langle\overline{l_i}\rangle, \langle\overline{l'_j}\rangle, \langle\overline{S_i}\rangle, \langle\overline{\tau'_j}\rangle) & \text{iff } \overline{C \triangleright S_i} \text{ and } \{\overline{l_i}\} \cap \{\overline{l'_j}\} = \emptyset \text{ and } \overline{\tau'_j} \text{ closed in } C \\
C \triangleright \textsc{fold}\ (t, \tau) & \text{iff } \mu t.\, \tau \text{ is closed in } C \\
C \triangleright \textsc{fix}\ (t, \theta, S) & \text{iff } C' \triangleright S \text{ and } [\![S]\!\!]_{C'} = \theta \text{ and } \langle\!\langle S \rangle\!\rangle_{C'} \downarrow t \\
 & \quad \text{where } C' = C \| t \le \theta, \text{ and } t \notin FTV(C) \\
C \triangleright \textsc{inst}\ (\langle\overline{t_i}\rangle, \langle\overline{\theta_i}\rangle, \varphi, \langle\overline{S_i}\rangle) & \text{iff } \overline{C \triangleright S_i} \text{ and } \overline{[\![S_i]\!\!]_C = \sigma\theta_i} \text{ where } \sigma = [\overline{\langle\!\langle S_i \rangle\!\rangle_C / t_i}] \\
C \triangleright \textsc{gen}\ (\langle\overline{t_i}\rangle, \langle\overline{\theta_i}\rangle, S) & \text{iff } C' \triangleright S \text{ and } \{\overline{t_i}\} \cap (FTV(C) \cup FTV(\langle\!\langle S \rangle\!\rangle_{C'})) = \emptyset \\
 & \quad \text{and} \{\langle\!\langle S \rangle\!\rangle_{C'}, [\![S]\!\!]_{C'}\} \subseteq Fn\,Typ \\
 & \quad \text{where } C' = C \| \overline{t_i \le \theta_i}
\end{array}
$$

$$
\begin{array}{rl}
C \triangleright \textsc{refl}\ \tau & \text{iff } \tau \text{ is closed in } C \\
C \triangleright \langle s_1, \ldots, s_m \rangle & \text{iff } C \triangleright s_i \text{ for } 1 \le i \le m, \\
 & \quad \text{and } [\![s_i]\!\!]_C = \langle\!\langle s_{i+1} \rangle\!\rangle_C \text{ for } 1 \le i < m
\end{array}
$$

Figure 12. Valid subtyping derivations.

$$
\begin{array}{rl}
{}^{\Gamma}_{C}\triangleright \textsc{Var}\ x & \text{iff } x \in dom(\Gamma) \\
{}^{\Gamma}_{C}\triangleright \delta & \text{if } \delta \in \{\textsc{Nat}\ n,\ \textsc{Bool}\ b\} \\
{}^{\Gamma}_{C}\triangleright \textsc{Record}\ (\langle \overline{l_i} \rangle,\ \langle \overline{D_i} \rangle) & \text{iff } \overline{{}^{\Gamma}_{C}\triangleright D_i} \\
{}^{\Gamma}_{C}\triangleright \textsc{Sel}\ (l,\ D) & \text{iff } {}^{\Gamma}_{C}\triangleright D \text{ and } \exists \tau : [\![D]\!]^{\Gamma}_{C} = \{l\!:\!\tau\} \\
{}^{\Gamma}_{C}\triangleright \textsc{Cond}\ (D_1,\ D_2,\ D_3) & \text{iff } \overline{{}^{\Gamma}_{C}\triangleright D_i},\ [\![D_1]\!]^{\Gamma}_{C} = \mathsf{Bool} \text{ and } [\![D_2]\!]^{\Gamma}_{C} = [\![D_3]\!]^{\Gamma}_{C} \\
{}^{\Gamma}_{C}\triangleright \delta D & \text{iff } {}^{\Gamma}_{C}\triangleright D \text{ and } [\![D]\!]^{\Gamma}_{C} = \mathsf{Nat} \\
 & \quad \text{for } \delta \in \{\textsc{IsZero},\ \textsc{Succ},\ \textsc{Pred}\} \\
{}^{\Gamma}_{C}\triangleright \textsc{Ref}\ D & \text{iff } {}^{\Gamma}_{C}\triangleright D \\
{}^{\Gamma}_{C}\triangleright \textsc{Deref}\ D & \text{iff } {}^{\Gamma}_{C}\triangleright D \text{ and } \exists \tau : [\![D]\!]^{\Gamma}_{C} = \tau\ \mathsf{Ref} \\
{}^{\Gamma}_{C}\triangleright \textsc{Set}\ (D_1,\ D_2) & \text{iff } {}^{\Gamma}_{C}\triangleright D_1,\ {}^{\Gamma}_{C}\triangleright D_2 \text{ and } [\![D_1]\!]^{\Gamma}_{C} = [\![D_2]\!]^{\Gamma}_{C}\ \mathsf{Ref} \\
{}^{\Gamma}_{C}\triangleright \textsc{App}\ (D_1,\ D_2) & \text{iff } {}^{\Gamma}_{C}\triangleright D_1,\ {}^{\Gamma}_{C}\triangleright D_2 \text{ and } \exists \tau : [\![D_1]\!]^{\Gamma}_{C} = [\![D_2]\!]^{\Gamma}_{C} \mathsf{->} \tau \\
{}^{\Gamma}_{C}\triangleright \textsc{Abs}\ (\overline{t_i \leq \theta_i},\ x,\ \tau,\ D) & \text{iff } {}^{\Gamma \| x:\tau}_{C \| \overline{t_i \leq \theta_i}}\triangleright D \text{ and } \tau,\ \overline{\theta_i} \text{ are closed in } C\|\overline{t_i \leq \theta_i} \\
\hline
{}^{\Gamma}_{C}\triangleright \langle d,\ S \rangle & \text{iff } {}^{\Gamma}_{C}\triangleright d,\ C \triangleright S \text{ and } [\![d]\!]^{\Gamma}_{C} = \langle\!\langle S \rangle\!\rangle_{C}
\end{array}
$$

Figure 13. Valid typing derivations.

$$
\begin{array}{lrll}
(1)\ C\|\mathsf{t} \leq \mathsf{t} \mathsf{->} \mathsf{Nat} \vdash & \mathsf{t} \mathsf{->} \mathsf{Nat} \leq \mathsf{t} \mathsf{->} \mathsf{Nat} & & \text{by } (\textsc{refl}) \\
(2)\ C \vdash & \mu \mathsf{t}.\ \mathsf{t} \mathsf{->} \mathsf{Nat} \leq (\mu \mathsf{t}.\ \mathsf{t} \mathsf{->} \mathsf{Nat}) \mathsf{->} \mathsf{Nat} & & (\textsc{fix}) \text{ of } (1) \\
(3)\ C \vdash & \forall [\mathsf{t} \leq \mathsf{t} \mathsf{->} \mathsf{Nat}].\ \mathsf{t} \mathsf{->} \mathsf{Nat} \leq (\mu \mathsf{t}.\ \mathsf{t} \mathsf{->} \mathsf{Nat}) \mathsf{->} \mathsf{Nat} & & (\textsc{inst})\ (2) \\
(4)\ C \vdash & (\mu \mathsf{t}.\ \mathsf{t} \mathsf{->} \mathsf{Nat}) \mathsf{->} \mathsf{Nat} \leq \mu \mathsf{t}.\ \mathsf{t} \mathsf{->} \mathsf{Nat} & & (\textsc{fold}) \\
(5)\ C \vdash & \forall [\mathsf{t} \leq \mathsf{t} \mathsf{->} \mathsf{Nat}].\ \mathsf{t} \mathsf{->} \mathsf{Nat} \leq \mu \mathsf{t}.\ \mathsf{t} \mathsf{->} \mathsf{Nat} & & (\textsc{trans})\ (3,4) \\
(6)\ C \vdash & \mathsf{Nat} \leq \mathsf{Nat} & & (\textsc{refl}) \\
(7)\ C \vdash & (\mu \mathsf{t}.\ \mathsf{t} \mathsf{->} \mathsf{Nat}) \mathsf{->} \mathsf{Nat} \leq (\forall [\mathsf{t} \leq \mathsf{t} \mathsf{->} \mathsf{Nat}].\ \mathsf{t} \mathsf{->} \mathsf{Nat}) \mathsf{->} \mathsf{Nat} & & (\textsc{fun})\ (5,6) \\
(8)\ C \vdash & \forall [\mathsf{t} \leq \mathsf{t} \mathsf{->} \mathsf{Nat}].\ \mathsf{t} \mathsf{->} \mathsf{Nat} \leq (\forall [\mathsf{t} \leq \mathsf{t} \mathsf{->} \mathsf{Nat}].\ \mathsf{t} \mathsf{->} \mathsf{Nat}) \mathsf{->} \mathsf{Nat} & & (\textsc{trans})\ (3,7)
\end{array}
$$

is encoded as $\langle s_0,\ \textsc{fun}\ (\langle s_0,\ \textsc{fold}\ (\mathsf{t},\ \mathsf{t} \mathsf{->} \mathsf{Nat})\rangle,\ \textsc{refl}\ \mathsf{Nat})\rangle$
where $s_0 = \textsc{inst}\ (\langle \mathsf{t} \rangle,\ \langle \mathsf{t} \mathsf{->} \mathsf{Nat} \rangle,\ \mathsf{t} \mathsf{->} \mathsf{Nat},\ \langle\langle \textsc{fix}\ (\mathsf{t},\ \mathsf{t} \mathsf{->} \mathsf{Nat},\ \textsc{refl}\ (\mathsf{t} \mathsf{->} \mathsf{Nat}))\rangle\rangle\rangle)$. □

Lemma 4

(i) *Let C' be a proper extension of C, i.e. $dom(C') \supseteq dom(C)$ and $C'(t) = C(t)$ for all $t \in dom(C)$; let S be a subtyping derivation, and $C \triangleright S$. Then $C' \triangleright S$, $\langle\!\langle S \rangle\!\rangle_{C'} = \langle\!\langle S \rangle\!\rangle_{C}$ and $[\![S]\!\rangle_{C'} = [\![S]\!\rangle_{C}$.*

(ii) *Similarly for environments and typing derivations.*

Proof: By structural induction on the derivations. ■

Definition 1 *Two subtyping derivations are* equivalent *in a context if they prove the same judgement in this context:*

$$S \simeq_C S' \text{ iff } C \triangleright S,\ C \triangleright S',\ \langle\!\langle S \rangle\!\rangle_{C} = \langle\!\langle S' \rangle\!\rangle_{C} \text{ and } [\![S]\!\rangle_{C} = [\![S']\!\rangle_{C}.$$

Let $q_1|q_2$ denote the concatenation of two arbitrary sequences q_1 and q_2. We define $S_1 \| S_2$ (the "concatenation" of two subtyping derivations) as $S_1|S_2$, if both S_1 and S_2 are sequences, and as the other argument, if one of them is a REFL (the ambiguity here disappears for valid derivations, see Lemma 5). Furthermore for $D_1 = \langle d_1,\, S_1\rangle \in \mathit{TypDer}$ and $S_2 \in \mathit{SubDer}$ we define $D_1 \| S_2 = \langle d_1,\, S_1 \| S_2\rangle \in \mathit{TypDer}$. Directly from the definitions we have

LEMMA 5 *If $C \triangleright S_1$, $C \triangleright S_2$, and $[\![S_1]\!\rangle_C = \langle\!\langle S_2]\!]_C$, then $C \triangleright S_1 \| S_2$.*

COROLLARY 2 *If ${}^{\Gamma}_{C} \triangleright D$, $C \triangleright S$, and $[\![D]\!]^{\Gamma}_{C} = \langle\!\langle S]\!]_C$, then ${}^{\Gamma}_{C} \triangleright D \| S$.*

LEMMA 6 *If C and $C\|\overline{t_i \le \theta_i}$ are consistent, $C\|\overline{t_i \le \theta_i} \triangleright S$, and $\{\overline{S_i}\}$ are subtyping derivations satisfying $\overline{C \triangleright S_i}$ and $\overline{[\![S_i]\!\rangle_C = \sigma\theta_i}$, where the substitution σ is $[\overline{\langle\!\langle S_i]\!]_C / t_i}]$, then there exists $S' \in \mathit{SubDer}$ such that $C \triangleright S'$ and $\langle\!\langle S']\!]_C = \sigma \langle\!\langle S]\!]_{C\|\overline{t_i \le \theta_i}}$ and $[\![S']\!\rangle_C = \sigma [\![S]\!\rangle_{C\|\overline{t_i \le \theta_i}}$.*

Proof: Assume that all type variables bound in subterms of S are renamed to not appear free in $\overline{S_i}$. To obtain S' from S, we extend σ to a map σ^S on subtyping derivations by applying σ to all type terms in the derivation, and replacing occurrences of VAR t_i by S_i; thus for instance

$$\begin{aligned}
\sigma^S\langle s_1,\, \ldots,\, s_m\rangle &= \sigma^s s_1 \| \ldots \| \sigma^s s_m \\
\sigma^s(\text{VAR } t_i) &= S_i \\
\sigma^s(\text{VAR } t) &= \langle \text{VAR } t\rangle, \text{ if } t \notin \mathit{dom}(\sigma) \\
\sigma^s(\text{INST } (\langle\overline{t'_j}\rangle,\, \langle\overline{\theta'_j}\rangle,\, \tau,\, \langle\overline{S'_j}\rangle)) &= \langle \text{INST } (\langle\overline{t'_j}\rangle,\, \langle\overline{\sigma\theta'_j}\rangle,\, \sigma\tau,\, \langle\overline{\sigma^S S'_j}\rangle)\rangle, \quad \text{etc.}
\end{aligned}$$

By induction on the structure of S the derivation $S' = \sigma^S S$ is valid in C, and it is easy to see that the types reconstructed from S' are related to their counterparts in S via σ. ■

The system of subtyping rules, excluding (REFL) and (TRANS) as in the definition of *SubDer*, would be deterministic in some sense (e.g. only one rule may be applied to infer $C \vdash \tau \le \tau'$ given the outermost type constructor of τ) if not for the rules (FOLD) and (FIX). The following Lemmas show that in certain cases this ambiguity can be dealt with by eliminating the occurrences of the corresponding *SubDer* constructors from the "top level" of a sequence, thus converting it to canonical form.

We begin by showing that applications of FIX can be moved deeper into the derivation tree so that only the special case UNFOLD $(t,\, \tau) \stackrel{\text{def}}{=}$ FIX $(t,\, \tau,$ REFL $\tau)$ is left at top-level.

Given a constraint system C and a type variable t, define inductively $\mathcal{N}_C(t)$ as the set of types of the form $\mu t_1.\, \ldots \mu t_n.\, t_0$ $(n \ge 0)$ such that $t_0 = t$ or $C(t_0) \in \mathcal{N}_C(t)$.

LEMMA 7 *For every C and t, $\mathcal{N}_C(t)$ is downward closed, i.e. if $C \vdash \tau' \le \tau$ and $\tau \in \mathcal{N}_C(t)$, then $\tau' \in \mathcal{N}_C(t)$.*

Proof: By induction on the structure of the derivation S of $C \vdash \tau' \leq \tau$. The statement is trivial if $S = \text{REFL}\,\tau$; we now assume $S = \langle s_1, \ldots, s_m \rangle$ and proceed by induction on the length of S. Our goal is to prove $\langle\!\langle s_m \rangle\!\rangle_C \in \mathcal{N}_C(t)$. From Figure 10 the type $[\![S]\!\rangle_C$ can have the form $\mu t_1. \ldots \mu t_n. t_0$ only if the outermost constructor of s_m is one of VAR, FOLD or FIX. The first two are base cases of an induction on the depth of S. The proof is immediate in the first case; in the second one we have $n \geq 1$, and $\langle\!\langle s_m \rangle\!\rangle_C$ is either $\mu t_2. \ldots \mu t_n. \tau$, if $t_0 = t_1$, or $\mu t_2. \ldots \mu t_n. t_0$ otherwise, but both of these are in $\mathcal{N}_C(t)$. For the inductive case $s_m = \text{FIX}\ (t', \theta', S')$ we assume that the statement of the Lemma is true for all derivations of lesser depth than S, and let C' denote $C \| t' \leq \theta'$. There are two subcases:

(i) $\theta' = \mu t_1. \ldots \mu t_k. t'$ where $k < n$; this leads to a contradiction: by inductive hypothesis $\langle\!\langle S' \rangle\!\rangle_{C'} \in \mathcal{N}_{C'}(t')$, which is only possible if $\langle\!\langle S' \rangle\!\rangle_{C'} = \mu t_{k+2}. \ldots \mu t_n. t'$ (because $t' \notin FTV(C)$, t' does not occur in $C'(t'')$ for any t'' except possibly t' itself), but this type is not contractive in t', and FIX could not be applied.

(ii) $\theta' = \tau$, hence $[\![S']\!\rangle_{C'} = \tau$ and $\langle\!\langle s_m \rangle\!\rangle_C = \mu t'. \langle\!\langle S' \rangle\!\rangle_{C'}$. By inductive hypothesis we have $\langle\!\langle S' \rangle\!\rangle_{C'} \in \mathcal{N}_{C'}(t)$, therefore $\langle\!\langle S' \rangle\!\rangle_{C'}$ is of the form $\mu t'_1. \ldots \mu t'_k. t'_0$, where $t'_0 \neq t'$ (otherwise $\langle\!\langle S' \rangle\!\rangle_{C'}$ is not contractive in t'); and since $t' \notin FTV(C)$ we have $\langle\!\langle S' \rangle\!\rangle_{C'} \in \mathcal{N}_C(t)$, which implies $\langle\!\langle s_m \rangle\!\rangle_C \in \mathcal{N}_C(t)$. ■

COROLLARY 3 *If $C \triangleright S$ where $S = \text{FIX}\ (t, \theta, \langle \overline{s_j} \rangle)$ (or $S = \text{GEN}\ (\langle \overline{t_i} \rangle, \langle \overline{\theta_i} \rangle, \langle \overline{s_j} \rangle)$), then none of s_j is* VAR t *(resp.* VAR t_i*).*

Proof: Suppose first that S is a FIX, $s_j = \text{VAR}\,t$ for some j, and let $C' = C \| t \leq \theta$. If $j = 1$, then $\langle\!\langle s_1 \rangle\!\rangle_{C'} = t$; if $j > 1$, we have $[\![s_{j-1}]\!\rangle_{C'} = t \in \mathcal{N}_{C'}(t)$, and by Lemma 7 (for the derivation $\langle s_1, \ldots, s_{j-1} \rangle$) it follows that $\langle\!\langle s_1 \rangle\!\rangle_{C'} \in \mathcal{N}_{C'}(t)$, which is only possible for $\langle\!\langle s_1 \rangle\!\rangle_{C'} = \mu t_1. \ldots \mu t_n. t$ (since $t \notin FTV(C)$). In neither case is $\langle\!\langle s_1 \rangle\!\rangle_{C'}$ contractive in t, contradicting the validity of the application of FIX.

The proof for the case when S is a GEN is quite similar; observe that $\mathcal{N}_C(t) \cap \mathit{FnTyp} = \emptyset$ for any C and t. ■

DEFINITION 2 *Define the* root set $\mathcal{R}(S)$ *of a subtyping derivation $S = \langle \overline{s_j} \rangle$ as the least set with the following property: $s \in \mathcal{R}(S)$ if either $s \in \{\overline{s_j}\}$, or $s \in \{\overline{s'_k}\}$ where $\text{GEN}\,(\langle \overline{t_i} \rangle, \langle \overline{\theta_i} \rangle, \langle \overline{s'_k} \rangle) \in \mathcal{R}(S)$ for some $\overline{t_i}$, $\overline{\theta_i}$; define $\mathcal{R}(\text{REFL}\,\tau) = \emptyset$. We refer to the sequence $\langle \overline{s'_k} \rangle$ (in the antecedent of a (GEN) rule) as to a* floor *of the root set.*

COROLLARY 4 *If $C \triangleright S$ where $S = \text{FIX}\ (t, \theta, S')$ (or $S = \text{GEN}\ (\langle \overline{t_i} \rangle, \langle \overline{\theta_i} \rangle, S')$), then* VAR $t \notin \mathcal{R}(S')$ *(resp. $\{\overline{\text{VAR}\ t_i}\} \cap \mathcal{R}(S') = \emptyset$).*

Proof: By Corollary 3 and Definition 2. ■

LEMMA 8 *If $C \triangleright S$, then there exists $S' \simeq_C S$ such that if $s \in \mathcal{R}(S')$ and $s =$ FIX (t, θ, S_0) for some t, θ, and S_0, then $S_0 =$ REFL θ, i.e. $s =$ UNFOLD (t, θ).*

Proof: By inductions on the depth of S. Consider an element $s \in \mathcal{R}(S)$ of the form FIX (t, θ, S_0), and let $\tau = \langle\!\langle S \rangle\!\rangle_{C \| t \leq \theta}$ (hence $\langle\!\langle \langle s \rangle \rangle\!\rangle_C = \mu t. \tau$); our inductive

hypothesis is that S_0 has no occurrences of FIX in its root set, except as UNFOLDs. Observe that S_0 and $\langle s \rangle$ satisfy the conditions of Lemma 6, namely $C||t \leq \theta \triangleright S_0$ and $C \triangleright \langle s \rangle$ and $[\![\langle s \rangle]\!]_C = \sigma\theta$, where $\sigma = [\mu t.\, \tau/t]$. By that Lemma we have a derivation S_0' such that $C \triangleright S_0'$ with $\langle\!\langle S_0']\!]_C = \sigma\tau$ and $[\![S_0']\!]_C = \sigma\theta$; moreover the only structural differences between S_0 and S_0' are at the occurrences of VAR t, now replaced by $\langle s \rangle$. But none of these are in the root set of S_0 (by Corollary 4), and by inductive hypothesis there were no (proper) FIXes there either, so the root set of S' is FIX-free. The last step of the proof is to note that UNFOLD $(t,\, \tau)[\![S' \simeq_C \langle s \rangle$; thus we can replace each FIX by its expansion prefixed by an UNFOLD. ■

LEMMA 9 *Let $S \in SubDer$ be such that $C \triangleright S$ and $[\![S]\!]_C$ is not a recursive type. Then there exists $S' \simeq_C S$ with the property that none of the elements of its root set has* FOLD *as its outermost constructor.*

Proof: By Lemma 8 there exist subtyping derivations, equivalent to S in C, in the root set of which FIX occurs only as UNFOLD. Let S' be a derivation with these properties and with the least number of FOLDs in its root set, and suppose this number is not zero. Choose a floor $\langle s_1', \ldots, s_m' \rangle$ with FOLDs in it, and let s_j' be the FOLD with largest index. Then $[\![s_j']\!]_C$ is a recursive type, and therefore it cannot be the case that $j = m$ since $[\![s_m']\!]_C$ is not recursive — either by assumption, if $S' = \langle \overline{s_i'} \rangle$, or by Definition 2 and Figure 12 (the largest type on the floor must be in $FnTyp$). Also by assumption S' is a valid derivation ($C \triangleright S'$), hence $\langle\!\langle s_{j+1}']\!]_C = [\![s_j']\!]_C$ which (by inspection of Figure 10) is only possible if s_{j+1}' is FOLD or FIX. But s_j' was the rightmost FOLD in the sequence, so s_{j+1}' must be FIX; by assumption the only form in which FIX occurs in the root set of S' is UNFOLD, so we have also $\langle\!\langle s_j']\!]_C = [\![s_{j+1}']\!]_C$. Therefore the pair of s_j' and s_{j+1}' can be removed from S', yielding an equivalent valid derivation with less FOLDs, which contradicts the assumption about S'. ■

LEMMA 10 (CANONICAL SUBTYPING PROOFS) *If $C \triangleright S$ and $[\![S]\!]_C$ is not recursive, then*

(i) *if $\langle\!\langle S]\!]_C$ is one of* Nat, Bool *or τ* Ref *(for some τ), then $S \simeq_C$* REFL $\langle\!\langle S]\!]_C$;

(ii) *if $\langle\!\langle S]\!]_C = \{\overline{l_i : \tau_i}\}$, then $S \simeq_C \langle s \rangle$ where s is a* RECORD;

(iii) *if $\langle\!\langle S]\!]_C = \forall[\overline{t_i \leq \theta_i}].\, \tau_1 \text{->} \tau_2$ and $[\![S]\!]_C$ is not polymorphic, then $S \simeq_C \langle s_1,\, s_2 \rangle$ where s_1 is an* INST *and s_2 is a* FUN.

Proof:

(i) By Lemma 9 there is a valid derivation $S' \simeq_C S$ not involving FOLD in its root set. Suppose S' is a sequence with first element s_1'. Since $\langle\!\langle s_1']\!]_C$ is a Nat, Bool or τ Ref, an inspection of Figure 10 shows that s_1' can only be a FOLD. But by assumption no element of $\mathcal{R}(S')$ is a FOLD. It follows that S' is not a sequence, hence $S' =$ REFL $\langle\!\langle S]\!]_C$.

(ii) If $S' = \langle \overline{s'_j} \rangle$ corresponds to S by Lemma 9, inspecting Figure 10 we see that $\langle\!\langle s'_1 \rangle\!\rangle_C = \{\overline{l_i : \tau_i}\}$ only if s'_1 is a RECORD, and hence $[\![s']\!\rangle_C$ is also a record type; by induction on the length of S' all of its elements are RECORDs. Furthermore it is easy to check that

$$\langle \overline{\text{RECORD}\ (L_i,\ L'_i,\ \langle S_i^1, \ldots, S_i^{n_i} \rangle,\ T_i)} \rangle \simeq_C$$
$$\langle \text{RECORD}\ (L_m,\ L'_1 | \ldots | L'_m,\ \langle (S_1^1 [\!] \ldots [\!] S_m^1), \ldots, (S_1^{n_m} [\!] \ldots [\!] S_m^{n_m}) \rangle,\ T_1 | \ldots | T_m) \rangle$$

(we let i range from 1 to m, and assume without loss of generality that the label sequence L_m is an initial subsequence of each of L_i.)

Alternatively, if S' is not a sequence, we can use the equivalence

$$\text{REFL}\ \{\overline{l_i : \tau_i}\} \simeq_C \langle \text{RECORD}\ (\langle \overline{l_i} \rangle, \langle\rangle, \langle \overline{\text{REFL}\ \tau_i} \rangle, \langle\rangle) \rangle$$

(iii) We first show that we can eliminate all occurrences of GEN from the top level of S. Let S' be a derivation corresponding to S by Lemma 9 (in this case $\langle\!\langle S]\!]_C \neq [\![S \rangle\!\rangle_C$, hence S' must be some sequence $\langle \overline{s'_i} \rangle$); suppose it has a minimal number of occurrences of GEN at top level, and $s'_j = \text{GEN}\ (\langle \overline{t_i} \rangle, \langle \overline{\theta_i} \rangle, S_0)$ with $[\![s'_j \rangle\!\rangle_C = \forall [\overline{t_i \leq \theta_i}].\, \varphi$ is the GEN with largest index at the top level of S'. It cannot be the last element of the sequence, since $[\![S \rangle\!\rangle_C$ is not polymorphic by assumption, and from Figure 10 the only possible forms of s'_{j+1} are FOLD, GEN, and INST; but there are no FOLDs in the root set (and hence in the top level) of S', and s'_j was the rightmost GEN, so $s'_{j+1} = \text{INST}\ (\langle \overline{t_i} \rangle, \langle \overline{\theta_i} \rangle, \varphi, \langle \overline{S_i} \rangle)$ (since $\langle\!\langle s'_{j+1}]\!]_C = [\![s'_j \rangle\!\rangle_C$). Note that S_0 and $\overline{S_i}$ satisfy the conditions of Lemma 6 (cf. Figure 12 for validity of INST and definition of the substitution σ). Thus we have a derivation S'_0 such that $C \triangleright S'_0$ with $\langle\!\langle S'_0]\!]_C = \sigma \langle\!\langle S_0]\!]_C$ and $[\![S'_0 \rangle\!\rangle_C = \sigma\varphi$; note also that $\mathcal{R}(S'_0)$ has the same number of occurrences of GEN as $\mathcal{R}(S_0)$ (by Corollary 4 none of the VARs replaced in S_0 are in its root set). But $\sigma \langle\!\langle S_0]\!]_C = \langle\!\langle S_0]\!]_C = \langle\!\langle s'_j]\!]_C$, since the variables $\overline{t_i}$ are not free in it; also $[\![s'_{j+1} \rangle\!\rangle_C = \sigma\varphi$. Thus we can replace $\langle s'_j,\ s'_{j+1} \rangle$ by S'_0 in the top level of S'; the resulting derivation has one GEN less than S' in its root set, contradicting our assumption.

Thus there exists $\langle \overline{s'_i} \rangle \simeq_C S$ with no occurrences of FOLD or GEN at top level; therefore (by inspection of Figure 10) $\langle\!\langle s'_1]\!]_C$ can be a polymorphic type only if s'_1 is an INST, and hence $[\![s'_1 \rangle\!\rangle_C$ is a function type $\tau \text{->} \tau'$. If the sequence has more than one element, then s'_2 can only be a FUN therefore $\langle\!\langle s'_2]\!]_C = [\![s'_1 \rangle\!\rangle_C$ is again a function type; therefore (similarly to case (ii) of this Lemma) the rest of the sequence is equivalent to a single FUN:

$$\langle \overline{\text{FUN}\ (S_i,\ S'_i)} \rangle \simeq_C \langle \text{FUN}\ (S_1 [\!] \ldots [\!] S_m,\ S'_1 [\!] \ldots [\!] S'_m) \rangle.$$

If s'_1 is the last element, we can append $\langle \text{FUN}\ (\text{REFL}\ \tau,\ \text{REFL}\ \tau') \rangle$ to obtain the required form. ■

3.2.2. Subject Reduction Theorem

The main idea in proving subject reduction is that the single-step computation relation induces a similar relation on derivations, i.e. that we can construct a type derivation for the contractum given the derivation for the redex, and it can then be replaced in the reduction context's derivation.

We define typability of memories and computation states.

DEFINITION 3 *A memory* Σ *is* typable *in* Γ *if for each* $x \in dom(\Sigma)$ *there is a closed type* τ *such that* $\Gamma(x) = \tau$ Ref *and* $\emptyset; \Gamma \vdash \Sigma(x) : \tau$.

DEFINITION 4 $\Gamma \vdash \langle \Sigma, e \rangle : \tau$ *if and only if* $dom(\Gamma) = dom(\Sigma)$, *the memory* Σ *is typable in* Γ, *and* $\emptyset; \Gamma \vdash e : \tau$.

LEMMA 11 *If* $\Gamma \vdash \langle \Sigma, e \rangle : \tau$, *then* $\langle \Sigma, e \rangle$ *is closed.*

Proof: By Definitions 4 and 3 $\Gamma \vdash \langle \Sigma, e \rangle : \tau$ implies that there exist valid in Γ type derivations for all expressions in the memory, as well as for e. By induction on *TypDer* all free variables in an expression with a type derivation valid in Γ must be in $dom(\Gamma)$, and by Definition 4 $dom(\Gamma) = dom(\Sigma)$. ■

LEMMA 12 (REPLACEMENT) *If* $\Gamma \vdash \langle \Sigma, R[e] \rangle : \tau$, *where* R *is a reduction context, then*

(i) $\emptyset; \Gamma \vdash e : \tau'$ *for some* τ', *and*

(ii) *if* $\emptyset; \Gamma \vdash e' : \tau'$, *then* $\Gamma \vdash \langle \Sigma, R[e'] \rangle : \tau$.

Proof: By Lemma 3 and induction on the structure of the derivation D of type τ for $R[e]$ in Γ; note that by definition R binds no variables. ■

LEMMA 13 (INSTANTIATION) *Let* $D \in TypDer$ *be such that* $\frac{\Gamma \| x : \tau}{C \| \overline{t_i \leq \theta_i}} \triangleright D$ *where* Γ *is closed in* C *and* C *is consistent. If* $\overline{S_i}$ *are subtyping derivations satisfying* $\overline{C \triangleright S_i}$ *and* $\overline{[\![S_i]\!]_C = \sigma\theta_i}$ *where the substitution* σ *is* $[\overline{(\!|S_i|\!)_C / t_i}]$, *then there exists* $D' \in TypDer$ *satisfying* $\mathcal{E}(D') = \mathcal{E}(D)$ *and* $\frac{\Gamma \| x : \sigma\tau}{C} \triangleright D'$ *and* $[\![D']\!]_C^{\Gamma \| x : \sigma\tau} = \sigma [\![D]\!]_{C \| \overline{t_i \leq \theta_i}}^{\Gamma \| x : \tau}$.

Proof: Similar to the proof of Lemma 6. Extend σ to σ^D on typing derivations by applying σ to all type terms and σ^S (defined as in Lemma 6) to all *SubDer* terms, e.g. mapping $\langle \text{ABS}\ (C_0, x_0, \tau_0, D_0), S_0 \rangle$ to $\langle \text{ABS}\ (\sigma \circ C_0, x_0, \sigma\tau_0, \sigma^D D_0), \sigma^S S_0 \rangle$. By induction on the structure of D and Lemma 6 the derivation $D' = \sigma^D D$ is valid in C and $\Gamma \| x : \sigma\tau$ (which in turn is closed in C), and the type derived by D' is as promised; note also that $\mathcal{E}(D') = \mathcal{E}(D)$, since σ^D does not touch *TypDer* constructors. ■

LEMMA 14 (SUBSTITUTION) *Let* D *be a derivation for the* SOOP *term* e *satisfying* $\frac{\Gamma \| x : \tau'}{C} \triangleright D$, *and* D' *be a derivation for* e' *such that* $\frac{\Gamma}{C} \triangleright D'$ *and* $[\![D']\!]_C^\Gamma = \tau'$. *Then there exists a derivation* D_s *for* $e[e'/x]$ *for which* $\frac{\Gamma}{C} \triangleright D_s$ *and* $[\![D_s]\!]_C^\Gamma = [\![D]\!]_C^{\Gamma \| x : \tau'}$.

Proof: To obtain D_s from D we replace all free occurrences of VAR x in D by D' (after renaming of bound variables as per $[e'/x]$). This results in a derivation for $e[e'/x]$; its validity in Γ and the preservation of the derived type is proved by induction on the structure of D. ■

THEOREM 1 (SUBJECT REDUCTION) *Let* $\Gamma \vdash \langle \Sigma, e \rangle : \tau$ *and* $\langle \Sigma, e \rangle \mapsto_1 \langle \Sigma', e' \rangle$. *Then there exists* Γ' *such that* $\Gamma' \vdash \langle \Sigma', e' \rangle : \tau$.

Proof:
By Lemma 11 $\langle \Sigma, e \rangle$ is closed, so by Lemma 1 $\langle \Sigma', e' \rangle$ is closed as well.
By Definition 4 and Lemma 3 in terms of the language of derivations we have that

- Σ is typable in Γ, hence there exists a function $D^\Sigma \in \mathit{Var} \to \mathit{TypDer}$ such that $\mathit{dom}(D^\Sigma) = \mathit{dom}(\Sigma)$ and for each $x \in \mathit{dom}(\Sigma)$ it holds that ${}^{\Gamma}_{\emptyset} \triangleright D^\Sigma(x)$, $\mathcal{E}(D^\Sigma(x)) = \Sigma(x)$, and $[\![D^\Sigma(x)]\!]^\Gamma_\emptyset = \tau'$, where $\Gamma(x) = \tau'$ Ref;
- $\emptyset; \Gamma \vdash e : \tau$, hence there exists a derivation D such that ${}^{\Gamma}_{\emptyset} \triangleright D$, $\mathcal{E}(D) = e$, and $[\![D]\!]^\Gamma_\emptyset = \tau$.

We need to construct a new environment Γ', a function $D^{\Sigma'}$ and a derivation D' with the same properties with respect to the new state.

By definition of the single-step computation we have $e = R[e_0]$ where e_0 is a redex; then $e' = R[e'_0]$, where e'_0 is the corresponding contractum. Let D_0 be the subterm of D deriving the type of e_0; by Lemma 12(i) it follows that ${}^{\Gamma}_{\emptyset} \triangleright D_0$.

In cases (i)–(viii) and (x) below there are no side effects of the reduction (i.e. $\Sigma' = \Sigma$), so in these cases we set $\Gamma' = \Gamma$ and $D^{\Sigma'} = D^\Sigma$. By Lemma 12(ii) we then have ${}^{\Gamma}_{\emptyset} \triangleright D'$ and $\mathcal{E}(D') = e'$ and $[\![D']\!]^\Gamma_\emptyset = [\![D]\!]^\Gamma_\emptyset$ (implying $\emptyset; \Gamma \vdash e' : \tau$) if D' is obtained by replacing D_0 in D by a derivation D'_0 such that ${}^{\Gamma}_{\emptyset} \triangleright D'_0$ and $\mathcal{E}(D'_0) = e'_0$ and $[\![D'_0]\!]^\Gamma_\emptyset = [\![D_0]\!]^\Gamma_\emptyset$.

Depending on the structure of e we have the following possibilities:

(i) $e_0 = (\lambda x.\, e_b)(v)$; then $e'_0 = e_b[v/x]$. Since $\mathcal{E}(D_0) = e_0$, by definition of $\mathcal{E}$ (Figure 11) the only possible form of D_0 is

$$D_0 = \langle \text{APP}\ (\langle \text{ABS}\ (\overline{t_i \leq \theta_i},\ x,\ \tau',\ D_b),\ S_f \rangle,\ D_v),\ S \rangle$$

for some $\overline{t_i}$, $\overline{\theta_i}$, and τ'; here D_b derives a type for e_b, and D_v derives a type for v. Let $C_b = \overline{t_i \leq \theta_i}$ and $\Gamma_b = \Gamma \| x : \tau'$.

From ${}^{\Gamma}_{\emptyset} \triangleright D_0$ we have $\emptyset \triangleright S_f$, $\langle\!\langle S_f \rangle\!\rangle_\emptyset = \forall[\overline{t_i \leq \theta_i}].\, \tau' \text{->} [\![D_b]\!]^{\Gamma_b}_{C_b}$, and $[\![S_f]\!\rangle_\emptyset = [\![D_v]\!]^\Gamma_\emptyset \text{->} \langle\!\langle S \rangle\!\rangle_\emptyset$. Applying Lemma 10(iii) to S_f, we get $S_f \simeq_\emptyset \langle s_1,\ s_2 \rangle$ where

$$\begin{aligned} s_1 &= \text{INST}\ (\langle \overline{t_i} \rangle,\ \langle \overline{\theta_i} \rangle,\ \tau' \text{->} [\![D_b]\!]^{\Gamma_b}_{C_b},\ \langle \overline{S_i} \rangle) \\ s_2 &= \text{FUN}\ (S'_1,\ S'_2) \end{aligned}$$

for some $\langle \overline{S_i} \rangle$, S'_1 and S'_2. Let $\tau_1 \text{->} \tau_2 = [\![s_1]\!\rangle_\emptyset = \langle\!\langle s_2 \rangle\!\rangle_\emptyset$ be the intermediate type of this sequence. Taking into account that ${}^{\Gamma}_{\emptyset} \triangleright D_0$ implies ${}^{\Gamma_b}_{C_b} \triangleright D_b$, and $\emptyset \triangleright S_f$

implies $\emptyset \triangleright s_1$ and therefore $\overline{\emptyset \triangleright S_i}$, by Lemma 13 from D_b and $\overline{S_i}$ we can obtain a derivation D_b' for e_b satisfying $[\![D_b']\!]_\emptyset^{\Gamma \| x:\tau_1} = \tau_2$.

We now use ${}^{\Gamma}_{\emptyset}\triangleright D_0$ once again to infer ${}^{\Gamma}_{\emptyset}\triangleright D_v$ and $[\![S_f]\!]_\emptyset = [\![D_v]\!]_\emptyset^\Gamma$ -> $\langle\!\langle S \rangle\!\rangle_\emptyset$. Recalling that $\langle\!\langle s_2 \rangle\!\rangle_\emptyset = \tau_1$ -> τ_2 and $[\![s_2]\!]_\emptyset = [\![S_f]\!]_\emptyset$, we can conclude that $\langle\!\langle S_1' \rangle\!\rangle_\emptyset = [\![D_v]\!]_\emptyset^\Gamma$ and $[\![S_1']\!]_\emptyset = \tau_1$. Therefore D_b' and $D_v[\![S_1'$ satisfy the conditions of Lemma 14, and hence there is a derivation D_s for $e_0' = e_b[v/x]$ such that ${}^{\Gamma}_{\emptyset}\triangleright D_s$ and $[\![D_s]\!]_\emptyset^\Gamma = \tau_2$. Now we can use $\langle\!\langle S_2' \rangle\!\rangle_\emptyset = \tau_2$, $[\![S_2']\!]_\emptyset = \langle\!\langle S \rangle\!\rangle_\emptyset$ and Corollary 2 to construct the necessary derivation D_0' as $D_s[\![S_2'[\![S$.

(ii) $e_0 =$ if true then e_t else e_f, hence $e_0' = e_t$. It must be the case that $D_0 = \langle \textsc{Cond}\ (D_c,\ D_t,\ D_f),\ S\rangle$, where $\mathcal{E}(D_t) = e_t$; from ${}^{\Gamma}_{\emptyset}\triangleright D_0$ follow ${}^{\Gamma}_{\emptyset}\triangleright D_t$ and $[\![D_t]\!]_\emptyset^\Gamma = \langle\!\langle S \rangle\!\rangle_\emptyset$, therefore $D_0' = D_t[\![S$.

(iii) $e_0 =$ if false then e_t else e_f; this case is similar to (ii).

(iv) $e_0 =$ is_zero(0); then $e_0' =$ true. Here we have $D_0 = \langle \textsc{IsZero}\ \langle \textsc{Nat}\ 0,\ S'\rangle,\ S\rangle$ hence $\langle\!\langle S \rangle\!\rangle_\emptyset =$ Bool. Therefore $D_0' = \langle \textsc{Bool}\ \mathsf{true},\ S\rangle$ is the required derivation.

(v) $e_0 =$ is_zero(n) where $n \neq 0$ is similar to (iv).

(vi) $e_0 =$ succ(n), and $e_0' = n'$. Here D_0 must be of the form $\langle \textsc{Succ}\ \langle \textsc{Nat}\ n,\ S'\rangle,\ S\rangle$, hence $\langle\!\langle S \rangle\!\rangle_\emptyset =$ Nat, and so $D_0' = \langle \textsc{Nat}\ n',\ S\rangle$ is a valid derivation satisfying $[\![D_0']\!]_\emptyset^\Gamma = [\![D_0]\!]_\emptyset^\Gamma$.

(vii) $e_0 =$ pred(n) is similar to (vi).

(viii) $e_0 = \{\overline{l_i = v_i}\}.l$; then $e_0' = v_k$ where $l = l_k \in \{\overline{l_i}\}$. Therefore

$$D_0 = \langle \textsc{Sel}\ (l_k,\ \langle \textsc{Record}\ (\langle\overline{l_i}\rangle,\ \langle\overline{D_{v_i}}\rangle),\ S'\rangle),\ S\rangle$$

where $\mathcal{E}(D_{v_i}) = v_i$. From ${}^{\Gamma}_{\emptyset}\triangleright D_0$ we get $\langle\!\langle S' \rangle\!\rangle_\emptyset = \{\overline{l_i : [\![D_{v_i}]\!]_\emptyset^\Gamma}\}$ and $[\![S']\!]_\emptyset = \{l_k : \langle\!\langle S \rangle\!\rangle_\emptyset\}$ — hence by Lemma 10(ii)

$$S' \simeq_\emptyset \langle \textsc{record}\ (\langle l_k\rangle,\ \langle\overline{l_j}\rangle,\ \langle S_k'\rangle,\ \langle\overline{[\![D_{v_j}]\!]_\emptyset^\Gamma}\rangle)\rangle$$

for some subtyping derivation S_k'; here the index j ranges over $\{1, \ldots, m\} \setminus \{k\}$. Since

$$\langle\!\langle \textsc{record}\ (\langle l_k\rangle,\ \langle\overline{l_j}\rangle,\ \langle S_k'\rangle,\ \langle\overline{[\![D_{v_j}]\!]_\emptyset^\Gamma}\rangle) \rangle\!\rangle_\emptyset = \{l_k : \langle\!\langle S_k' \rangle\!\rangle_\emptyset,\ \overline{l_j : [\![D_{v_j}]\!]_\emptyset^\Gamma}\}$$

it follows that $\langle\!\langle S_k' \rangle\!\rangle_\emptyset = [\![D_{v_k}]\!]_\emptyset^\Gamma$; on the other hand $[\![S_k']\!]_\emptyset = \langle\!\langle S \rangle\!\rangle_\emptyset$, so we can build the required derivation as $D_0' = D_{v_k}[\![S_k'[\![S$.

(ix) $e_0 =$ ref v. In this case $\Sigma' = \Sigma \| [\, x' \mapsto v\,]$ and $e_0' = x'$, where $x' \notin dom(\Sigma)$. The derivation D_0 is of the form $\langle \textsc{Ref}\ D_v,\ S\rangle$, where $\mathcal{E}(D_v) = v$. Let $\Gamma' = \Gamma \| x' : \langle\!\langle S \rangle\!\rangle_\emptyset$. Since by Definition 4 $dom(\Gamma) = dom(\Sigma)$, it follows that $x' \notin dom(\Gamma)$, and by Lemma 4(ii) we have ${}^{\Gamma'}_{\emptyset}\triangleright D^\Sigma(x)$ for all $x \in dom(\Sigma)$, and ${}^{\Gamma'}_{\emptyset}\triangleright D$. In particular this also implies (by Lemma 12(i)) that ${}^{\Gamma'}_{\emptyset}\triangleright D_v$ and $\langle\!\langle S \rangle\!\rangle_\emptyset = [\![D_v]\!]_\emptyset^\Gamma$ Ref =

$[\![D_v]\!]^{\Gamma'}_{\emptyset}$ Ref. Therefore D_v is the derivation we need to type the new memory in the new environment: let $D^{\Sigma'} = D^{\Sigma}|||[\, x' \mapsto D_v \,]$. Furthermore, by Lemma 12(ii) we obtain a valid in Γ' derivation for e' if we replace D_0 in D by $D'_0 = \langle \textsc{Var}\ x',\ S \rangle$.

(x) $e_0 = !\, x$ where $x \in dom(\Sigma)$. Then $e'_0 = \Sigma(x)$ and $D_0 = \langle \textsc{Deref}\ \langle \textsc{Var}\ x,\ S' \rangle,\ S \rangle$; also $x \in dom(\Gamma)$ with $\Gamma(x)$ being a reference type. From $\overset{\Gamma}{\emptyset} \triangleright D_0$ it follows that $\langle\!\langle S']\!]_{\emptyset} = \Gamma(x)$ and $[\![S' \rangle\!\rangle_{\emptyset} = \langle\!\langle S]\!]_{\emptyset}$ Ref, and hence by Lemma 10(i) applied to S' we have $\Gamma(x) = \langle\!\langle S]\!]_{\emptyset}$ Ref.

Since $x \in dom(\Sigma)$ we have $\mathcal{E}(D^{\Sigma}(x)) = \Sigma(x)$, $\overset{\Gamma}{\emptyset} \triangleright D^{\Sigma}(x)$, and $[\![D^{\Sigma}(x)]\!]^{\Gamma}_{\emptyset} = \langle\!\langle S]\!]_{\emptyset}$. Therefore $D'_0 = D^{\Sigma}(x)[\![S$ is the necessary derivation.

(xi) $e_0 = \mathsf{set}(x,\ v)$ where $x \in dom(\Sigma)$; then $\Sigma' = \Sigma|||[\, x \mapsto v \,]$ and $e'_0 = v$. It must be the case that $D_0 = \langle \textsc{Set}\ (\langle \textsc{Var}\ x,\ S' \rangle,\ D_v),\ S \rangle$, and from $\overset{\Gamma}{\emptyset} \triangleright D_0$ follow $\overset{\Gamma}{\emptyset} \triangleright D_v$ and $\Gamma(x) = [\![D_v]\!]^{\Gamma}_{\emptyset}$ Ref (applying Lemma 10(i) to S'). Therefore Σ' is still typable in $\Gamma' = \Gamma$ — to wit, the necessary derivations are $D^{\Sigma'} = D^{\Sigma}|||[\, x \mapsto D_v \,]$. The derivation for e'_0 is $D'_0 = D_v[\![S$. ■

3.2.3. Soundness of the Soop *Type System*

Lemma 15 *If the type environment* Γ *is such that for each* $x \in dom(\Gamma)$ $\Gamma(x) = \tau$ Ref *for some type* τ*, then*

(i) $C; \Gamma \vdash v : \mathsf{Nat}$ *if and only if* $v \in Nat$;

(ii) $C; \Gamma \vdash v : \mathsf{Bool}$ *if and only if* $v \in Bool$;

(iii) $C; \Gamma \vdash v : \tau$ Ref *only if* $v \in Var$;

(iv) $C; \Gamma \vdash v : \{l : \tau\}$ *only if* $v = \{\overline{l_i = v_i}\}$ *for some* $\overline{v_i}$*, and* $l \in \{\overline{l_i}\}$;

(v) $C; \Gamma \vdash v : \tau' \text{->} \tau''$ *only if* $v = \lambda x.\, e$ *for some* x *and* e.

Proof: Directly from Lemma 10; to illustrate we prove (i). Suppose $v \notin Nat$. Then $v \in Bool$ or $v \in Var$ or $v = \{\overline{l_i : v_i}\}$ or $v = \lambda x.\, e$; hence d in a type derivation $D = \langle d,\ S \rangle$ for v must be respectively a Bool, a Var, a Record, or an Abs, and from $[\![d]\!]^{\Gamma}_{C} = \langle\!\langle S]\!]_{C}$, $[\![S \rangle\!\rangle_{C} = \mathsf{Nat}$ (i.e. not a recursive type) and the corresponding cases of Lemma 10 we get a contradiction for S. ■

Lemma 16 (Stuck States Are Not Typable) *If* $\Gamma \vdash \langle \Sigma,\ e \rangle : \tau$*, then either* $e \in Val$*, or there exists a state* $\langle \Sigma',\ e' \rangle$ *such that* $\langle \Sigma,\ e \rangle \mapsto_1 \langle \Sigma',\ e' \rangle$.

Proof: By induction on the structure of e:

Base Case: $e = x$ or $e = n$ or $e = b$ or $e = \lambda x.\, e'$ or $e = \{\overline{l_i : v_i}\}$ — in all cases $e \in Val$.

Inductive Hypothesis: If $e' \notin Val$ is a proper subterm of e, $\langle \Sigma, e' \rangle$ is closed, $\emptyset; \Gamma \vdash e' : \tau'$ for some type τ', Σ is typable in Γ, and $dom(\Gamma) = dom(\Sigma)$, then there exist a reduction context R' and a redex e_0 such that $e' = R'[e_0]$.

Inductive Step: We prove that if $e \notin Val$, $\langle \Sigma, e \rangle$ is closed, $\emptyset; \Gamma \vdash e : \tau$, Σ is typable in Γ, and $dom(\Gamma) = dom(\Sigma)$, then $e = R[e_0]$ for some reduction context R and redex e_0.

Depending on the structure of e we have the following cases:

(i) $e = e_1(e_2)$. By definition e is typable in $\emptyset; \Gamma$ (i.e. has a valid type derivation — cf. Figure 13) only if e_1 is typable in $\emptyset; \Gamma$ (no new bound variables are introduced by the application). If $e_1 \notin Val$, by the inductive hypothesis $e_1 = R'[e_0]$, and therefore $e = R[e_0]$ for $R = R'(e_2)$. Similarly if $e_1 \in Val$ but $e_2 \notin Val$ we have $e_2 = R'[e_0]$, and then $e = R[e_0]$ for $R = e_1(R')$.

If both e_1 and e_2 are values, we note that the only form of a valid type derivation for e is an APP, which requires that $\emptyset; \Gamma \vdash e_1 : \tau' \text{->} \tau''$ for some τ' and τ''; then by Lemma 15(v) we have $e_1 = \lambda x.\, e'$, and hence $e = R[e_0]$ for $R = \circ$ and $e_0 = e$.

(ii) $e =$ if e' then e_1 else e_2. From the typability of e (in $\emptyset; \Gamma$) follows $\emptyset; \Gamma \vdash e' : \mathsf{Bool}$; if $e' \notin Val$, from the inductive hypothesis we have $e = R[e_0]$ for $R =$ if R' then e_1 else e_2 where $e' = R'[e_0]$. Otherwise ($e' \in Val$) Lemma 15(ii) gives us that $e' \in Bool$, hence e is itself a redex.

(iii) $e = \mathsf{is\_zero}(e')$; therefore $\emptyset; \Gamma \vdash e' : \mathsf{Nat}$. If $e' \notin Val$, inductively $e' = R'[e_0]$, and hence $R = \mathsf{is\_zero}(R')$ is the reduction context necessary for $e = R[e_0]$; otherwise ($e' \in Val$) by Lemma 15(i) it follows that $e' \in Nat$ and therefore e is a redex.

(iv) $e = \mathsf{succ}(e')$, and

(v) $e = \mathsf{pred}(e')$ are similar to (iii).

(vi) $e = \{l_1 = e_1, \ldots, l_m = e_m\}$ where not all of e_i are values (otherwise $e \in Val$). Let k be the smallest index for which $e_k \notin Val$; then $e_k = R'[e_0]$ by inductive hypothesis, and for $R = \{l_1 = e_1, \ldots, l_{k-1} = e_{k-1}, R', l_{k+1} = e_{k+1}, \ldots, l_m = e_m\}$ we have $e = R[e_0]$.

(vii) $e = e'.l$; therefore $\emptyset; \Gamma \vdash e' : \{l : \tau'\}$. If $e' \notin Val$, the inductive hypothesis yields $e' = R'[e_0]$, hence $e = R[e_0]$ for a reduction context $R = R'.l$. If e' is a value, Lemma 15(iv) states that e' is a record (of values) and one of the labels in it is l; thus e itself is a redex.

(viii) $e =$ ref e' — if $e' \notin Val$, inductively $e' = R'[e_0]$, and $e = R[e_0]$ for $R =$ ref R'; otherwise e is a redex.

(ix) $e = !\ e'$; hence we have $\emptyset; \Gamma \vdash e' : \tau'$ Ref for some τ'. The case $e' \notin Val$ is handled by induction. If e' is a value, by Lemma 15(iii) it is a variable x, and hence it is typable only if $x \in dom(\Gamma)$. By assumption $dom(\Gamma) = dom(\Sigma)$ and Σ is typable in Γ, so $\Gamma(x)$ is a reference type if and only if $x \in dom(\Sigma)$. Therefore e is a redex.

(x) $e = \mathsf{set}(e_1, e_2)$ — if $e_1 \notin Val$, then by the inductive hypothesis $e_1 = R'[e_0]$, and so $e = R[e_0]$ for $R = \mathsf{set}(R', e_2)$. If $e_1 \in Val$ but $e_2 \notin Val$, then $e = R[e_0]$ for $R = \mathsf{set}(e_1, R')$ where $e_2 = R'[e_0]$. Finally if both e_1 and e_2 are values, similarly to the previous case we have $e_1 = x \in dom(\Sigma)$, hence e is a redex. ■

THEOREM 2 (SOUNDNESS OF THE SOOP TYPE SYSTEM) *If* $\Gamma \vdash \langle \Sigma, e \rangle : \tau$, *then either* $\langle \Sigma, e \rangle$ *diverges, or* $\langle \Sigma, e \rangle \mapsto^* \langle \Sigma', v \rangle$ *where* $\emptyset; \Gamma' \vdash v : \tau$ *for some* Γ'.

Proof: Directly from Theorem 1 and Lemma 16. ■

4. LOOP Semantics and Soundness

In this section we show how LOOP terms and types may be given meaning by translation into SOOP, the result being LOOP programs experience no "message not understood" errors upon execution.

4.1. Semantics of LOOP terms

The semantics of LOOP terms is defined via a translation into SOOP terms; this translation is given in Figure 14.

In many encodings of object-oriented languages [9], [18] an object is defined as a record formed by taking a fixed point of a class function, a function mapping records to records. 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...meth 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 good results, but the situation changes with the introduction of effects and call-by-value semantics. The associated fixed-point operator is then only well-defined on functionals, but as pointed out for instance in [10], classes do not correspond to functionals.

Several solutions with various limitations have been proposed. One possible solution is to "freeze" the access of an object to its "self," as in

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

Further complications arise if the language is to support mutable instance variables belonging to objects—in the straightforward implementations their allocation takes place either "too early" (they are shared by all objects of the class) or "too late" (new cells are being allocated at each access to the object).

Two approaches to solving this problem are discussed in [17]; 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 ...
  inst x = ...
  meth m = ...self...
o = new c
```

yielding

$$\begin{array}{l} c = \lambda().\,\mathsf{let}\ y = \mathsf{ref}\ \ldots\ \mathsf{in} \\ \quad \lambda self.\,\lambda().\,\{x = y,\ m = \ldots self() \ldots\} \\ o = Y(c())() \end{array}$$

However we can only do this with the restriction that the initial values of the instance variables may not refer to *self*. We believe *self* should be allowed in instance variable initializations, for it makes possible method update in objects. A method stored in an instance variable (as a function) may be updated by setting the instance variable to a different function. So if instance variable initializations could not refer to *self*, the initial value of an updateable method could not refer to *self* and would thus be of limited use. See the end of Section 2.1 for an implementation of updateable methods in LOOP.

Our approach allows using *self* in instance variable initializations by defining a "weak" fixed-point combinator (a fixed-point operator on functionals, i.e. expressions of the form $\lambda x.\,\lambda y.\,e$ [21]) in terms of mutable cells in the spirit of Landin [19]. A first approximation to this is

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

where null is some "dummy" initial value. The class function f now maps a cell containing a record to a record, and this cell will be assigned the value of the "fixed point" (the object being created). This combinator evaluates $f(r)$ only once before "tying the knot" on the object, therefore it can be applied to class functions with side effects, in particular allocating mutable cells, to obtain objects with the intuitively expected behavior.

However, this definition of Y_r is lacking: the class function f takes a reference cell as argument, and there is no nontrivial subtyping on reference cells in SOOP. This means inheritance from classes would not type-check. More precisely, when a class function f is passed a reference r to the future object, and f inherits features from a superclass defined by f', f must apply f' to r (since the "self" of the object must be shared between inherited and new methods). However, since a reference is not a subtype of any other reference type, either f and f' have the same type (false if f extends f'), or one of $f(r)$ and $f'(r)$ must not be type-correct. In either case, inheritance will not work.

$$
\begin{aligned}
[\![\ v\ ]\!] &= v \\
[\![\ n\ ]\!] &= n \\
[\![\ b\ ]\!] &= b \\
[\![\ \texttt{succ}(e)\ ]\!] &= \mathsf{succ}([\![e]\!]) \\
[\![\ \texttt{pred}(e)\ ]\!] &= \mathsf{pred}([\![e]\!]) \\
[\![\ \texttt{is\_zero}(e)\ ]\!] &= \mathsf{is\_zero}([\![e]\!]) \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ ]\!] &= \mathsf{if}\ [\![e_1]\!]\ \mathsf{then}\ [\![e_2]\!]\ \mathsf{else}\ [\![e_3]\!] \\
[\![\ \texttt{fn}\ v\ \texttt{=>}\ e\ ]\!] &= \lambda v.\, [\![e]\!] \\
[\![\ e(e')\ ]\!] &= [\![e]\!]([\![e']\!]) \\
[\![\ e.x\ ]\!] &= !\ ([\![e]\!](\{\}).\mathsf{inst}.x) \\
[\![\ e.x := e'\ ]\!] &= \mathsf{set}\,([\![e]\!](\{\}).\mathsf{inst}.x,\ [\![e']\!]) \\
[\![\ e\texttt{<-}m\ ]\!] &= [\![e]\!](\{\}).\mathsf{meth}.m \\
[\![\ \texttt{new}\ e\ ]\!] &= Y'_r([\![e]\!]) \\
[\![\ \texttt{class}\ s\ \texttt{super}\ \overline{u_i\ \texttt{of}\ e_i}\ \ \texttt{inst}\ \overline{x_j = e'_j}\ \texttt{meth}\ \overline{m_k = e''_k}\ ]\!] &= \lambda s.\ \mathsf{let}\ \overline{u_i = [\![e_i]\!](s)}\ \mathsf{in} \\
&\qquad (\lambda \mathsf{x}.\ \lambda \mathsf{y}.\ \mathsf{x})(\{\ \mathsf{inst} = \{\overline{x_j = \mathsf{ref}\ [\![e'_j]\!]}\}, \\
&\qquad\qquad \mathsf{meth} = \{\overline{m_k = [\![e''_k]\!]}\}\ \})
\end{aligned}
$$

Figure 14. Translation of Loop Terms to Soop Terms

There is no inherent need for general references here; the cell is set only once, and after that it is read-only. So, the idea is to define a "cell accessor function" of type `{}->{...}` that when applied will return the result in the cell. f then takes this cell accessor function as argument, and will thus not have a Ref type argument. The following "weak" fixed-point combinator achieves this effect.

$$Y'_r = \lambda f.\, \mathsf{let}\ r = \mathsf{ref}\ (\lambda x.\, \Omega)\ \mathsf{in}\ (\mathsf{set}\,(r,\ f(\lambda x.\, (!\ r)(x)))\mathsf{;}\ !\ r)$$

where $\Omega = (\lambda x.\, x(x))(\lambda x.\, x(x))$ is a diverging computation. The initial value placed in the cell is $\lambda x.\, \Omega$ since it has all function types and thus imposes no additional type constraint on the cell. It is worth noting that $Y'_r(f)$ is the fixed point of f when f is a functional, for its application evaluates without side effects [21].

Let us now examine the encoding of Figure 14 more carefully. For uniformity, we interpret classes as functions from frozen records `{}->{...}` to frozen records, and objects as frozen records. These records in turn have two fields, inst which is a record containing the instance variables, and meth a record containing the methods. The `new` operation then applies Y'_r to the class to make an object. Note the instance variables are not hidden in Soop. We model instance variable hiding by crippling the Loop type system to not allow typing of access to the instance variables of objects by removing them from object types.

Each class function is a wrapper around the class functions of its superclasses. Within its body, the function first applies its superclass functions to its own cell accessor; the resulting record holds the instance variables and methods defined by the superclass whose "self" name has been bound to the subclass's cell accessor. These

inherited values are then available for use in the body of methods and instances of the child class.

It is important to recognize the implications of this inheritance mechanism. Each class function will be applied to several different kinds of cell accessors, one for each of its subclasses. This fact has important consequences in the Loop type system, and motivates the F-bounded polymorphism of Soop.

Since Loop is given semantics by translation, no evaluation relation for Loop computations is defined. In a more complete presentation we would define such a primitive evaluation mechanism and relate it to Soop evaluation, but a translation is sufficient for our purpose of characterizing erroneous Loop executions. By using a translation it is also more easy to relate our language with other work in the literature that is based on standard notions of function, record, and reference. So, for practical reasons we now find it more fruitful to pursue a translative approach to meaning. The risk is the possibility that a translation will miss some key idea; we see no evidence for this as of yet but that does not rule out the possibility. See [1] for an alternative, primitive theory of object execution and typing.

Definition 5 *A* Loop *program e becomes* stuck *if its corresponding* Soop *translation in empty memory $\langle \emptyset, [\![e]\!] \rangle$ gets stuck.*

In particular, note that a "message not understood" error in Loop corresponds to an attempt to access a nonexistent field of a record in Soop. Such attempts lead to stuck Soop states, and thus Loop programs that have such translations are considered erroneous under this definition.

4.2. Semantics of Loop Types

The translation of Loop types to Soop types is given in Figure 15, following the general technique sketched above. The most important feature not discussed is how the open-ended "self-type" may be represented by F-bounded quantification. Classes are functions from frozen records to frozen records, taking as a parameter a representation of the final object created by this class and any further extensions (the "self"). The final type of the objects created by this class are not known at this point; indeed, a class can be part of several distinct class heirarchies, each producing objects of different types. Thus, classes are polymorphic in the type of their object's public interface(s), implemented by means of the F-bounded quantification of Soop. When t occurs free in M, the methods implemented by this class have open-ended notions of "self-type."

4.3. An Example Translation

To illustrate the translation of Loop to Soop, we translate the Loop example of Section 2.2 into Soop. The translation of the Loop program of Figure 5 using the translation of Figure 14 is given in Figure 16. As can be seen, each Loop `class`

Types:

$$
\begin{aligned}
[\![\ \texttt{Bool}\ ]\!] &= \mathsf{Bool} \\
[\![\ \texttt{Nat}\ ]\!] &= \mathsf{Nat} \\
[\![\ \tau{\to}\tau'\ ]\!] &= [\![\tau]\!] \texttt{->} [\![\tau']\!] \\
[\![\ t\ ]\!] &= \{\}\texttt{->}\{\mathsf{meth}{:}t\} \\
[\![\ \texttt{PreObj}\,M\ ]\!] &= \{\}\texttt{->}\{\mathsf{meth}{:}[\![M]\!]\} \\
[\![\ \texttt{Obj}(t)\,M\ ]\!] &= \{\}\texttt{->}\{\mathsf{meth}{:}\mu t.\,[\![M]\!]\} \\
[\![\ \texttt{Class}(t)(I;\,M)\ ]\!] &= \forall[t \le [\![M]\!]].\,[\![\texttt{Self}(I;\,t){\to}\texttt{Super}(I;\,M)]\!] \\
[\![\ \texttt{Self}(I;\,t)\ ]\!] &= \{\}\texttt{->}\{\mathsf{inst}{:}[\![I]\!],\ \mathsf{meth}{:}t\} \\
[\![\ \texttt{Super}(I;\,M)\ ]\!] &= \{\}\texttt{->}\{\mathsf{inst}{:}[\![I]\!],\ \mathsf{meth}{:}[\![M]\!]\}
\end{aligned}
$$

Instance Variables: $[\![\ \{\overline{x_i:\tau_i}\}\ ]\!] = \{\overline{x_i:[\![\tau_i]\!]\ \mathsf{Ref}}\}$

Methods: $[\![\ \{\overline{m_i:\tau_i}\}\ ]\!] = \{\overline{m_i:[\![\tau_i]\!]}\}$

Figure 15. Translation of LOOP Types

```
let empty = Y'_r (λself. let in (λx. λy. x)({inst = {}, meth = {}}))
in let Num = λself.
     let in (λx. λy. x) ({
       inst = {value = ref 0},
       meth = {
         dec = λdummy. set(self({}).inst.value, pred(! (self({}).inst.value))),
         isZero = λdummy. is_zero(! (self({}).inst.value)),
         diff = λother.
           if self({}).meth.isZero(empty) then other
           else if other({}).meth.isZero(empty) then self
           else (
             self({}).meth.dec(empty);
             other({}).meth.dec(empty);
             self({}).meth.diff(other) ) } })
in let n = Y'_r (Num)
in let n' = Y'_r (Num)
in let CNum = λself.
     let number = Num(self)
     in (λx. λy. x) ({
       inst = {value = ! (number({}).inst.value) },
       meth = {
         click = counter({}).meth.click,
         dec = number({}).meth.dec,
         isZero = λdummy. (self({}).meth.click(empty); number({}).meth.isZero(empty) ),
         diff = number({}).meth.diff } })
in let cn = Y'_r (CNum)
in let cn' = Y'_r (CNum)
in (cn({}).meth.diff(cn'))({}).meth.click;
   (cn({}).meth.diff(n))({}).meth.isZero
```

Figure 16. Translation of the LOOP program of Figure 5 to SOOP.

$$\begin{array}{ll}
\textbf{Constraints:} & [\![\, t \leq \texttt{PreObj}\, M \,]\!] = t \leq [\![M]\!] \\
\textbf{Hypotheses:} & [\![\, v : \tau \,]\!] = v : [\![\tau]\!] \\
\textbf{Judgements:} & [\![\, C \vdash \tau \leq \tau' \,]\!] = [\![C]\!] \vdash [\![\tau]\!] \leq [\![\tau']\!] \\
 & [\![\, C; \Gamma \vdash e : \tau \,]\!] = [\![C]\!]; [\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau]\!]
\end{array}$$

Figure 17. Translation of Loop Judgements

translates into a function that generates objects, and `new` creates an object of the class via Y'_r.

Consider the open-self typings; recall from Section 2.2 the Loop types `NumClass`, `NumObj`, `CNumClass`, and `CNumObj`. They are translated to the following Soop types.

$$\begin{array}{rcl}
[\![\texttt{NumClass}]\!] & = & \forall[\mathsf{SelfType} \leq M_N].\, \langle\!\langle I_N,\, \mathsf{SelfType} \rangle\!\rangle \to \langle\!\langle I_N,\, M_N \rangle\!\rangle \\
[\![\texttt{NumObj}]\!] & = & \{\} \to \{\mathsf{meth}: \mu\mathsf{SelfType}.\, M_N\} \\
I_N & = & \{\mathsf{value}: \mathsf{Nat}\} \\
M_N & = & \{\mathsf{dec}: \mathsf{unit} \to \mathsf{Nat},\ \mathsf{isZero}: \mathsf{unit} \to \mathsf{Bool}, \\
 & & \quad \mathsf{diff}: (\{\} \to \{\mathsf{meth}: \mathsf{SelfType}\}) \to \{\} \to \{\mathsf{meth}: \mathsf{SelfType}\}\}
\end{array}$$

$$\begin{array}{rcl}
[\![\texttt{CNumClass}]\!] & = & \forall[\mathsf{SelfType} \leq M_{CN}].\, \langle\!\langle I_{CN},\, \mathsf{SelfType} \rangle\!\rangle \to \langle\!\langle I_{CN},\, M_{CN} \rangle\!\rangle \\
[\![\texttt{CNumObj}]\!] & = & \{\} \to \{\mathsf{meth}: \mu\mathsf{SelfType}.\, M_{CN}\} \\
I_{CN} & = & \{\mathsf{value},\ \mathsf{cnt}: \mathsf{Nat}\} \\
M_{CN} & = & \{\mathsf{click},\ \mathsf{dec}: \mathsf{unit} \to \mathsf{Nat},\ \mathsf{isZero}: \mathsf{unit} \to \mathsf{Bool}, \\
 & & \quad \mathsf{diff}: (\{\} \to \{\mathsf{meth}: \mathsf{SelfType}\}) \to \{\} \to \{\mathsf{meth}: \mathsf{SelfType}\}\}
\end{array}$$

where $\langle\!\langle \tau_I,\, \tau_M \rangle\!\rangle \stackrel{\text{def}}{=} \{\} \to \{\mathsf{inst}: \tau_I,\ \mathsf{meth}: \tau_M\}$, and $\mathsf{unit} \stackrel{\text{def}}{=} \{\} \to \{\mathsf{meth}: \{\}\}$.

4.4. Soundness of the Loop Type System

We now may show Loop type derivations are sound by translation into Soop. Translation of Loop type judgements into Soop type judgements is as shown in Figure 17. For the most part, the translation is straightforward, as the translation of judgements is homomorphic in its treatment of expressions and terms. Its translation of typing constraints deserves some additional commentary, however.

If a Loop type constraint $t \leq \texttt{PreObj}\, M$ were translated in a homomorphic fashion following the scheme for translating types given in Figure 15, the result would be

$$\{\} \to \{\mathsf{meth}: t\} \leq \{\} \to \{\mathsf{meth}: [\![M]\!]\}$$

This is not a valid Soop type constraint, since it does not relate a type variable to its upper bound. However, such a subtyping relation is equivalent to Soop type constraint $t \leq [\![M]\!]$, as is obvious by examination of the Soop subtyping rules.

Lemma 17 *The translation of each* Loop *subtyping rule is a provable* Soop *subtyping derivation.*

Proof: Examining each rule in turn:

(i) Rules (*Refl*), (*Trans*), and (*Fun*) translate directly into their SOOP equivalents.

(ii) The soundness of (*Hyp*) is essentially the derivation of

$$t \leq [\![M]\!] \vdash \texttt{\{\}->\{meth:}t\texttt{\}} \leq \texttt{\{\}->\{meth:}[\![M]\!]\texttt{\}}$$

which follows from rules (VAR), (RECORD), and (FUN).

(iii) The translation of (*PreObj*) is provable by rules (RECORD) and (FUN) in a similar way.

(iv) (*Fold*) and (*Fix*) translate into derivations involving rules (FUN), (RECORD), and (FOLD). Translating the result of substituting $\texttt{Obj(}t\texttt{)}\,M$ (which translates to $\texttt{\{\}->\{meth:}\mu t.\,[\![M]\!]\texttt{\}}$) for t (which translates to $\texttt{\{\}->\{meth:}t\texttt{\}}$) in $\texttt{PreObj}\ M$ is the same as substituting $\mu t.\,[\![M]\!]$ for t in $\texttt{\{\}->\{meth:}[\![M]\!]\texttt{\}}$. ■

LEMMA 18 (LOOP SUBTYPING) *If $C \vdash \tau \leq \tau'$ is derivable via the* LOOP *subtyping rules, then $[\![C]\!] \vdash [\![\ \tau\ ]\!] \leq [\![\ \tau'\ ]\!]$ is a provable* SOOP *judgement.*

Proof: By induction on LOOP subtyping judgements, and Lemma 17. ■

LEMMA 19 *If $C;\Gamma \vdash e : \tau$ is provable in* LOOP*, the corresponding judgement $[\![C]\!];[\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau]\!]$ is provable in* SOOP*.*

Proof: By induction on the proof of $C;\Gamma \vdash e : \tau$. Considering the rule in the proof of $C;\Gamma \vdash e : \tau$ in turn:

(i) Rules (*Var*), (*Num*), (*Bool*), (*Pred*), (*Succ*), (*IsZero*), (*Cond*), and (*App*) translate directly into their SOOP equivalents, and thus the conclusion follows by induction.

(ii) Rule (*Sub*). The conclusion then follows by Lemma 18 and rule (SUB).

(iii) Rule (*Abs*). The conclusion follows by rules (ABS), (INST), and (SUB), where rule (ABS) generalizes over an empty set of type variables.

(iv) Rule (*Mesg*). As $[\![e\texttt{<-}m]\!] = [\![e]\!](\texttt{\{\}}).\texttt{meth}.m$ and $[\![\texttt{PreObj}\ M]\!] = \texttt{\{\}->\{meth:}[\![M]\!]\texttt{\}}$, the conclusion follows by rules (APP), (RECORD), and (SELECT).

(v) (*New*) translates into the purported SOOP derived rule:

$$([\![\ New\ ]\!])\quad \frac{[\![C]\!];[\![\Gamma]\!] \vdash [\![e]\!] : \forall[t \leq [\![M]\!]].\,[\![\texttt{Self(}I\texttt{;}\,t\texttt{)}\rightarrow\texttt{Super(}I\texttt{;}\,M\texttt{)}]\!]}{[\![C]\!];[\![\Gamma]\!] \vdash Y'_r([\![e]\!]) : \texttt{\{\}->\{meth:}\mu t.\,[\![M]\!]\texttt{\}}}$$

where $Y'_r = \lambda f.\,\mathsf{let}\ r = \mathsf{ref}\ (\lambda x.\,\Omega)\ \mathsf{in}\ (\mathsf{set}\,(r,\ f(\lambda x.\,(!\,r)(x)));!\,r)$. First, note that $\Omega = (\lambda x.\,x(x))(\lambda x.\,x(x))$ can be given any type τ, since $\lambda x.\,x(x)$ can be given type $\forall[\mathsf{t} \leq \mathsf{t}\texttt{->}\tau].\,\mathsf{t}\texttt{->}\tau$ by rules (VAR), (SUB), (APP), and (ABS); we choose

$\tau = \{\mathsf{inst}: [\![I]\!],\ \mathsf{meth}: \mu t.\,[\![M]\!]\}$, and hence $\lambda x.\,\Omega$ has type $\tau' \stackrel{\text{def}}{=} \{\}\text{->}\tau$. Then $\lambda x.\,(!\,r)(x)$ has type τ' under the assumption that r is of type τ' Ref. Thus, if f is given type $\tau'\text{->}\tau'$, then Y'_r will be of type $(\tau'\text{->}\tau')\text{->}\tau'$.

Notice also that $C \vdash \mu t.\,[\![M]\!] \leq [\![M]\!][\mu t.\,[\![M]\!]/t]$ by rule (FIX); thus $[\![e]\!]$ can be given type $\tau'\text{->}\{\}\text{->}\{\mathsf{inst}: [\![I]\!],\ \mathsf{meth}: [\![M]\!][\mu t.\,[\![M]\!]/t]\}$ (by rules (SUB) and (INST), instantiating t to $\mu t.\,[\![M]\!]$), which by (FOLD), (RECORD) and (FUN) is a subtype of $\tau'\text{->}\tau'$. The conclusion then follows by rules (SUB) and (APP).

(vi) (*Self*). The conclusion follows by rules (VAR) and (SUB), since

$$[\![\ \mathtt{Self}(I\,;\,t)\ ]\!] = \{\}\text{->}\{\mathsf{inst}: [\![I]\!],\ \mathsf{meth}: t\} \leq \{\}\text{->}\{\mathsf{meth}: t\} = [\![\ t\ ]\!]$$

by rules (FUN) and (RECORD). Rule (*Super*) follows in a similar fashion.

(vii) (*SelfInst*). If s has type $[\![\ \mathtt{Self}(I\,;\,t)\ ]\!] = \{\}\text{->}\{\mathsf{inst}: [\![I]\!],\ \mathsf{meth}: t\}$, then $[\![\ e\,.\,x\ ]\!] = !\,([\![e]\!](\{\}).\mathsf{inst}.x)$ has type $[\![I(x)]\!]$, by rules (APP), (SELECT), and (DEREF) and the way the instance variables are translated. Rule (*SupInst*) follows in an identical fashion, and rule (*SelfAssn*) to an analogous way using rule (SET) in place of (DEREF).

(viii) (*Class*). Suppose $t \leq [\![M]\!]$. If $[\![t]\!] \leq [\![\mathtt{PreObj}\ M_i[t/t_i]]\!]$ is provable under this assumption, it must have been proven by rules (FUN) and (RECORD). Thus, $t \leq [\![M_i[t/t_i]]\!]$ is provable as well under this assumption. Now by hypothesis $[\![C]\!]; [\![\Gamma]\!] \vdash [\![e_i]\!] : [\![\mathtt{Class}(t_i)(I_i\,;\,M_i)]\!]$ is provable. Since $[\![\mathtt{Class}(t_i)(I_i\,;\,M_i)]\!] = \forall[t_i \leq [\![M_i]\!]].\,[\![\mathtt{Self}(I_i\,;\,t_i)\text{→}\mathtt{Super}(I_i\,;\,M_i)]\!]$ and t satisfies this bound, by rule (SUB) and (INST) each $[\![e_i]\!]$ can be given type $[\![\mathtt{Self}(I_i[t/t_i]\,;\,t)]\!]\text{→}[\![\mathtt{Super}(I_i\,;\,M_i)[t/t_i]]\!]$

Now suppose s has type $[\![\mathtt{Self}(I\,;\,t)]\!] = \{\}\text{->}\{\mathsf{inst}: [\![I]\!],\ \mathsf{meth}: t\}$. By rules (FUN) and (RECORD) and the hypothesis that $I_i(x)[t/t_i] = I(x)$ for each $x \in dom(I)$ it can be shown that $[\![\mathtt{Self}(I\,;\,t)]\!] \leq [\![\mathtt{Self}(I_i[t/t_i]\,;\,t)]\!]$ Then by (SUB) and (APP) each $[\![e_i]\!](s)$ has type $[\![\mathtt{Super}(I_i\,;\,M_i)[t/t_i]]\!]$. If under the additional assumption that u_i has type $[\![\mathtt{Super}(I_i\,;\,M_i)[t/t_i]]\!]$ it can be shown that $[\![e'_j]\!]$ has type $[\![I(x_j)]\!]$ and each $[\![e''_k]\!]$ has type $[\![M(m_k)]\!]$, then $\{\mathsf{inst} = \{\overline{x_j = \mathsf{ref}\ [\![e'_j]\!]}\},\ \mathsf{meth} = \{\overline{m_k = [\![e''_k]\!]}\}\}$ has type $\{\mathsf{inst}: [\![I]\!],\ \mathsf{meth}: [\![M]\!]\}$. The desired judgement then follows by rule (ABS). ■

THEOREM 3 (SOUNDNESS OF LOOP TYPING) *Typed* LOOP *programs do not become stuck.*

Proof: Suppose $\emptyset;\emptyset \vdash e : \tau$. By Lemma 19, $\emptyset;\emptyset \vdash [\![e]\!] : [\![\tau]\!]$ is then a provable SOOP judgement, and by Theorem 2 $\langle \emptyset,\ [\![e]\!] \rangle$ does not get stuck. ■

5. Discussion

We have shown here how the ideas of [7] used to model typed functional OOP can be applied to LOOP, a state-based object-oriented programming language. The

problem of OOP typing cannot be said to be completely solved, however. As the example in Section 2.2 illustrates, there is a tension between inheritance and subtyping, and in LOOP the programmer is forced to choose one over the other. It would be desirable to have a solution that did not force this decision on the programmer.

Ghelli presents another solution to the fixed/open "self" problem that always preserves subtyping between subclass and superclass objects, but at the expense of requiring redefinition of a method whenever the type changes, and doing dynamic dispatching based on the type information at run-time [15]. See [10], [3] for more discussion of how the open-ended view of "self" relates to other approaches in the literature.

In constructing SOOP we developed a provably sound axiomatization of F-bounded quantification in the presence of state that is strong enough to show the LOOP type system sound, and at the same time weak enough so the SOOP type system may itself be shown sound. We believe SOOP is of independent interest as an "Object Typing Workbench": alternate schemes for typing objects may be expressed in SOOP, and this strong foundation greatly simplifies the task of formulating a sound type system for OOP languages. The type system of SOOP offers recursive types with subtyping weaker than the one defined in [2] but stronger than that of [3]. The novel (FIX) rule may be of independent interest. The presentation also differs from others because type generalization and instantiation is implicit as in ML, not explicit as in System F. The proof of subject reduction for SOOP is involved, but it may be that no significantly simpler form exists.

Bruce and van Gent [6] have defined an imperative OOP language, TOIL, which is an imperative extension of Bruce's TOOPL language [3]. TOOPL/TOIL and LOOP are closely related, since they are all ultimately based on F-bounded polymorphism; here we outline some differences. We interpret LOOP via translation to SOOP, while TOIL is given semantics directly. The TOOPL/TOIL subtyping rules do not allow for folding and unfolding of object types, and this means the fixed-self typings will not be possible there. The LOOP (*ObjFix*) rule is somewhat more general than the object subtyping rule of TOOPL/TOIL. In LOOP the initial value of an instance variable may refer to "self," and if a function is stored in an instance variable, this allows a form of method override in objects. TOIL on the other hand has a *nil* object, which is of every object type, but responds to every message with an error. LOOP provides for multiple inheritance and nested class definitions; TOOPL/TOIL does not. The rules have quite a different character; our rules are more directly inspired by how the typings translate into SOOP. We have no need for Bruce's "matching" relation $\leq_{meth}$—in our formulation subtyping alone suffices.

There are several important language issues which are not addressed here. LOOP is monomorphic, even though it is translated to a polymorphic language. It would not be difficult to lift the F-bounded polymorphism of SOOP into LOOP. Classes are not fully "first-class citizens" in LOOP; to achieve this, some notion of extensible record would need to be added to SOOP [24]. There is no analogue here to the "friend" functions or "private" class variables of C++. Also, the question of

decidable type checking or decidable type inference is not addressed here. However, an explicitly-typed version of LOOP that is closely related to the language of this paper is proven to have a decidable type-checking algorithm in [13]. A decidability proof for type-checking a variant of TOOPL is presented in [5].

Acknowledgments

We would like to thank the anonymous referees for helpful comments, in particular motivating us to come up with a cleaner translation from LOOP to SOOP, and to extend the SOOP type system with stronger rules.

References

1. M. Abadi and L. Cardelli. A semantics of object types. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 332–341, 1994.
2. R. Amadio and L. Cardelli. Subtyping recursive types. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, 1991.
3. 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.
4. 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.
5. K. B. Bruce, J. Crabtree, T. P. Murtagh, R. van Gent, A. Dimock, and R. Muller. Safe and decidable type checking in an object-oriented language. In *OOPSLA '93 Conference Proceedings*, 1993.
6. Kim B. Bruce and Robert van Gent. TOIL: A new type-safe object-oriented imperative language. Technical report, Williams College, 1993.
7. P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.
8. 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.
9. W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *OOPSLA '89 Proceedings*, pages 433–443, 1989.
10. 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.
11. 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.
12. J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. A simple interpretation of OOP in a language with state. Technical Report YALEU/DCS/RR-968, Yale University, 1993.
13. J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. Application of OOP type theory: State, decidability, integration. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1994.
14. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992.
15. Giorgio Ghelli. A static type system for message passing. In *Proc. OOPSLA*, pages 129–145, 1991.

16. C. Gunter and J. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* MIT Press, 1994.
17. 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.
18. Samuel N. Kamin and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, chapter 13, pages 464–495. MIT Press, 1994.
19. P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.
20. D. B. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.
21. I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.
22. J. Mitchell. Towards a typed foundation for method specialization and inheritance. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990.
23. B. Pierce. Bounded quantification is undecidable. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 305–315, 1992.
24. Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1994.
25. J.P. Seldin. A sequent calculus for type assignment. *Journal of Symbolic Logic*, 42:11–28, 1977.
26. A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Rice University Department of Computer Science, 1991. To appear in *Information and Computation.*