Application of OOP Type Theory: State, Decidability, Integration

Jonathan Eifrig*† Scott Smith* Valery Trifonov*† Amy Zwarico†

Department of Computer Science, The Johns Hopkins University

{eifrig, scott, trifonov, amy}@cs.jhu.edu

Abstract

Important strides toward developing expressive yet semantically sound type systems for object-oriented programming languages have recently been made by Cook, Bruce, Mitchell, and others. This paper focusses on how the theoretical work using F-bounded quantification may be brought more into the realm of actual language implementations while preserving rigorous soundness properties. We simultaneously address three of the more significant problems: adding a notion of global **state**, proving type-checking is **decidable**, and **integrating** the more widely implemented view that subclasses correspond to subtypes with the F-bounded view.

1 Introduction

Developing expressive yet semantically sound type systems for object-oriented programming languages is a well-known and difficult research problem. Many workable solutions are possible, but there has as of yet been no universally accepted solution to the 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.

Important strides toward solving this problem have recently been made [8, 5, 2, 15]. By using F-bounded quantification, these researchers can capture the open-ended nature of class definitions in the presence of inheritance: "self" refers not only to the current class being defined, but also to any future extension of it by subclassing. An important problem is to bring this work more into the realm of sound language implementations; that is the subject of this paper.

We address three problems. Nearly all previous semantic work takes a functional view of objects; however, state is of central importance to OOP and cannot be ignored. Another problem is that an implementable language requires decidable type checking. Some progress has been made in this area for functional OO languages [3], but the problem is difficult and the most general type systems may prove to be undecibible [16]. The final problem we address is the integration of type systems with different views of the type of "self." 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. On the surface the open-self interpretation appears to be superior since it allows more class definitions to be successfully type-checked. However, the fixed-self approach has the advantage of allowing all objects to be lifted up the inheritance hierarchy, a property that sometimes fails for the open-self approach. Any object programmer knows this *object subsumption* feature is very useful. There is then a problem of how the advantages of both approaches may be combined in one language.

This paper provides a solution to these three problems. We develop a provably sound interpretation of an explicitly typed imperative object language, LOOP. This language includes notation for explicitly typed 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." Soundness of the type system and decidability of type-checking are proved. We additionally show how the fixed-self and open-self notions may be combined, embedding fixed-self class definitions in our open-self language as special cases.

*Partially supported by NSF grant CCR-9109070

†Partially supported by AFOSR grant F49620-93-1-0169

$$
\begin{array}{rrl}
\mathit{Num} \ni & n ::= & \texttt{0} \mid \texttt{1} \mid \ldots \\
\mathit{Bool} \ni & b ::= & \texttt{true} \mid \texttt{false} \\
\mathit{Exp} \ni & e ::= & v \mid n \mid b \mid \texttt{pred(}e\texttt{)} \mid \texttt{succ(}e\texttt{)} \mid \texttt{is\_zero(}e\texttt{)} \\
& & \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{fn}\ v : \tau\ \texttt{=>}\ e \\
& & \mid e(e) \mid a \mid a\texttt{.}x \mid s\texttt{.}x\texttt{:=}e \mid e\texttt{<-}m \mid \texttt{new}\ e \\
& & \mid \texttt{class}\ s : t\ \texttt{super}\ \overline{u_i\ \texttt{of}\ e_i} \\
& & \quad \texttt{inst}\ \overline{x_j : \tau_j\ \texttt{=}\ e_j}\ \texttt{meth}\ \overline{m_k : \tau_k\ \texttt{=}\ e_k} \\
\mathit{Typ} \ni & \tau ::= & t \mid \texttt{Bool} \mid \texttt{Nat} \mid \tau \rightarrow \tau' \\
& & \mid \texttt{Obj(}t\texttt{)}M \mid \texttt{Class(}t\texttt{)(}I\texttt{;}M\texttt{)} \\
& & \mid \texttt{PreObj}M \mid \texttt{Self[}t\texttt{](}I\texttt{;}M\texttt{)} \\
& I ::= & \texttt{\{}x_1 : \tau_1, \ldots, x_n : \tau_n\texttt{\}} \\
& M ::= & \texttt{\{}m_1 : \tau_1, \ldots, m_n : \tau_n\texttt{\}}
\end{array}
$$

Figure 1: Loop Syntax

The semantics of Loop are defined 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 easier 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 quantification for the interpretation of inheritance. We give a collection of type rules for Soop and show the rules sound with respect to an operational semantics: no run-time type errors can occur.

To be clear, we do not address delegation-style OOP here [15], nor do we consider having dynamic dispatch based on type information known at run-time [11].

2 The Loop Language

We begin our discussion by defining a representative object-oriented programming language, Loop (Little Object-Oriented Programming language).

Identifiers of Loop are divided into five sorts: $v \in K_{par}$ are function parameters, $s \in K_{self}$ are bound class names, $u \in K_{super}$ are bound superclass names, $x \in K_{inst}$ are instance variable names, and $m \in K_{meth}$ are method names. Furthermore, the metavariable a will be used to range over $K_{self} \cup K_{super}$. Throughout this paper, the "vector" notation $\overline{A_i}$ will be used to indicate a comma-separated sequence of elements, ranged over by the indicated index variable: $\overline{u_i\ \texttt{of}\ e_i} \overset{\text{def}}{=} u_1\ \texttt{of}\ e_1, \ldots, u_n\ \texttt{of}\ e_n$.

The syntax of Loop expressions and types 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

$$\texttt{class}\ s\texttt{:}t\ \texttt{super}\ \overline{u_i\ \texttt{of}\ e_i}\ \texttt{inst}\ \overline{x_j\texttt{:}\tau'_j\ \texttt{=}\ e'_j}$$
$$\texttt{meth}\ \overline{m_k\texttt{:}\tau''_k\ \texttt{=}\ e''_k}$$

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. Type information is explicitly given with class definitions to facilitate type checking. t is the type of "self", and the τ'_j and τ''_k give the types of individual instances and methods, respectively. 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, $m_j : \tau\ \texttt{=}\ u_i\texttt{<-}m_j$ denotes that m_j is inherited from superclass u_i. This serves to resolve any ambiguities with multiple inheritance. The current value of an instance variable is accessed by the expression $a\texttt{.}x$, and updated by $s\texttt{.}x\texttt{:=}e$. Methods are accessed by $e\texttt{<-}m$.

Terms of type $\texttt{Obj(}t\texttt{)}M$ 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 $\texttt{Obj(}t\texttt{)}M$ and represents the type of "self" within the object's interface. Similarly, terms of type $\texttt{Class(}t\texttt{)(}I\texttt{;}M\texttt{)}$ denote classes; in particular, such terms generate (via the `new` operation) objects of type $\texttt{Obj(}t\texttt{)}M$. $\texttt{PreObj}M$ and $\texttt{Self[}t\texttt{](}I\texttt{;}M\texttt{)}$ are used in type checking class expressions, and will be discussed in detail in the presentation of Loop rules in Section 5. We make some syntactic abbreviations to aid in program readability: $\texttt{let}\ v = e'\ \texttt{in}\ e$ stands for $\texttt{(fn}\ v\ \texttt{=>}\ e\texttt{)(}e'\texttt{)}$, sequencing $\texttt{(}e_1\texttt{;}\ e_2\texttt{)}$ abbreviates $\texttt{(fn x => fn y => y)(}e_1\texttt{)(}e_2\texttt{)}$. We also use $e\texttt{()}$ for the application of e to the "empty" object `new (class super inst meth)` of type $\mathbf{1} \overset{\text{def}}{=} \texttt{Obj(t)\{\}}$. Before giving the semantics, we next discuss issues involved in typing "self," and in the process also give an informal introduction to programming in Loop.

3 Typing Self

The main complexity of typing object-oriented languages arises from the interplay of subtyping and inheritance. 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 [8, 2].

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. As we shall see, neither the fixed nor the open-ended view subsumes the other, and the programmer may wish to have both available and choose between the two.

To make this idea concrete, consider the example Loop program in Figure 2. 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`.

What are the proper 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

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

The bound type variable `SelfType` refers to "the type of the object itself" — and indeed by the rules of the Loop type system (formally presented in Section 5) if `n` is of type `NumObj` then `n<-diff` is of type `NumObj`→`NumObj`.

We now present the two views of typing "self" in classes, first the open-ended view and then the fixed view. All typings asserted are in fact provable in the type system of Loop, given in Section 5.

3.1 The open-ended-self typing

First, we consider what type the class expression `Num` should be given in the open-ended view. In Loop the instance variables declared in a class definition are also visible in the definitions of subclasses, hence the types of these variables, along with the types of the methods, define the type of a class expression. One possible type

```
let Num = class self : SelfType super
    inst
      value : Nat = 0
    meth
      dec : 1→Nat =
         fn dummy : 1 =>
         self.value:=pred(self.value),
      isZero : 1→Bool =
         fn dummy : 1 =>
         is_zero(self.value),
      diff : τ→τ =
         fn other : τ =>
         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 : SelfType
    super
      number of Num
    inst
      cnt : Nat = 0,
      value : Nat = number.value
    meth
      click : 1→Nat =
         fn dummy : 1 =>
         self.cnt:=succ(self.cnt),
      dec : 1→Nat = number<-dec,
      isZero : 1→Bool =
         fn dummy : 1 =>
         (self<-click(); number<-isZero()),
      diff : τ→τ = number<-diff
in let cn = new CNum
in let cn' = new CNum
...
```

Figure 2: `Num` and `CNum` Classes

to give `Num` in LOOP is

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

(Note that under this typing, τ in Figure 2 is `SelfType`.) 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 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 of `self`, referred to via the type variable `SelfType`, is not known exactly 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 this *open-ended* view of `SelfType`: it is the type of "self" at object creation time, and thus the object could contain more methods via inheritance. According to the typing rules of LOOP, the objects generated by this class via `new` are of type `NumObj`, as expected.

The subclass `CNum` can then be given the LOOP type

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

and the generated objects are then of type

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

However, one consequence of this typing is the `diff` method of `CNum` cannot be applied to an object of type `NumObj`, because it does not have all the features of the subclass and hence does not match the type of `self`. Thus, we have chosen inheritance in favor of subtyping in the open-ended case. We will consider this issue in more detail below.

3.2 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})
```

(Note that under this typing, τ in Figure 2 is `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 as for `NumClass`.

The type of `CNum` must 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 type. This means `CFixNumObj` is a subtype of `NumObj` according to the type rules of LOOP. In general, in the fixed-self view subclass objects will always be subtypes of superclass objects, and 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.

3.3 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`$\rightarrow$`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`$\rightarrow$`NumObj` and `cn'` has type `CFixNumObj`. Now, since `CFixNumObj` is a subtype of `NumObj` (as mentioned above), `cn'` also has type `NumObj`, and hence `cn<-diff(cn')` is well-typed. Unfortunately, the result is of type `NumObj`, and thus the `click` method of `cn'` is lost.

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`$\rightarrow$`CNumObj`. However, `n` is only of type `NumObj`, *not* of type `CNumObj`, and 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`.

3.4 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 [11]. 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 (more precisely, an *invariant* view, the types do not change).

4 The Soop Language

We define the meaning of Loop programs in terms of a lower-level "implementation" language Soop (Semantics for Object-Oriented Programming), a call-by-value language which offers simple operations on records and reference cells in addition to most of the standard PCF constructs.

The syntax for Soop values $v \in \mathit{Val}$, expressions $e \in \mathit{Exp}$, and types $\theta, \tau \in \mathit{Typ}$ is presented below.

$$\begin{aligned}
v &::= x \mid n \mid b \mid \lambda x.\, e \mid \{\overline{l_i = v_i}\} \\
e &::= v \mid e(e) \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
&\quad\mid \mathsf{is\_zero}\,(e) \mid \mathsf{succ}\,(e) \mid \mathsf{pred}\,(e) \\
&\quad\mid \{\overline{l_i = e_i}\} \mid e.l \mid \mathsf{ref}\ e \mid !\ e \mid \mathsf{set}\,(e,\ e) \\
\theta,\ \tau &::= \mathsf{Nat} \mid \mathsf{Bool} \mid t \mid \tau \text{->} \tau' \mid \{\overline{l_i : \tau_i}\} \\
&\quad\mid \tau\ \mathsf{Ref} \mid \forall[\overline{t_i \leq \theta_i}].\, \tau \mid \mu t.\, \tau
\end{aligned}$$

Here we let the metavariable x range over the countable set of variables Var, l over a countable set of labels, $n \in \{0,\ 1,\ \ldots\}$ and $b \in \{\mathsf{true},\ \mathsf{false}\}$. The language includes some PCF terms, record construction and selection, and reference cells.

In addition to the basic types and type constructors (including references) the type system of Soop provides polymorphic and recursive types. The metavariables 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 $\forall[\overline{t_i \leq \theta_i}].\, \tau$ binds the type variables t_i in each of their *upper bounds* θ_i as well as in the body τ; this allows a form of F-bounded polymorphism [5]. The type variable t is bound in $\mu t.\, \tau$. By convention $\forall$ and μ have lower precedence than the arrow. We use $FTV(\tau)$ to denote the set of free type variables of τ.

The Soop types are partially ordered by the subtyping relation $\leq$, formalized by the system of rules in Figure 3. The subtyping system is an extension of the standard record subtyping [6] with recursive and F-bounded polymorphic types. A *constraint system* C is a finite map in $\mathit{TVar} \rightarrow \mathit{Typ}$; in our proof rules for the subtyping relation such functions serve as systems of assumptions about the free type variables, mapping them 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 written in the form $C \vdash \tau' \leq \tau''$ which for a consistent

$$(\textsc{refl})\quad \frac{\tau \text{ is closed in } C}{C \vdash \tau \le \tau}$$

$$(\textsc{trans})\quad \frac{C \vdash \tau \le \tau' \qquad C \vdash \tau' \le \tau''}{C \vdash \tau \le \tau''}$$

$$(\textsc{var})\quad \frac{t \in \mathit{dom}(C)}{C \vdash t \le C(t)}$$

$$(\textsc{fun})\quad \frac{C \vdash \tau_1' \le \tau_1 \qquad C \vdash \tau_2 \le \tau_2'}{C \vdash \tau_1 \texttt{->} \tau_2 \le \tau_1' \texttt{->} \tau_2'}$$

$$(\textsc{record})\quad \frac{\begin{array}{c} C \vdash \overline{\tau_i' \le \tau_i} \text{ for } 1 \le i \le m \\ \overline{\tau_{m+j}'} \text{ is closed in } C \text{ for } 1 \le j \le m' \end{array}}{C \vdash \{\overline{l_i : \tau_i'},\ \overline{l_{m+j} : \tau_{m+j}'}\} \le \{\overline{l_i : \tau_i}\}}$$

$$(\textsc{inst})\quad \frac{C \vdash \overline{\tau_i \le \sigma\theta_i}}{C \vdash \forall[\overline{t_i \le \theta_i}].\, \tau \le \sigma\tau} \text{ where } \sigma = [\overline{{}^{\tau_i}/_{t_i}}]$$

$$(\textsc{fold})\quad \frac{\mu t.\, \tau \text{ is closed in } C}{C \vdash [{}^{\mu t.\, \tau}/_{t}]\tau \le \mu t.\, \tau}$$

$$(\textsc{unfold})\quad \frac{\mu t.\, \tau \text{ is closed in } C}{C \vdash \mu t.\, \tau \le [{}^{\mu t.\, \tau}/_{t}]\tau}$$

Figure 3: Subtyping rules of Soop.

$$(\textsc{Sub})\quad \frac{C; \Gamma \vdash e : \tau \qquad C \vdash \tau \le \tau'}{C; \Gamma \vdash e : \tau'}$$

$$(\textsc{Var})\quad \frac{x \in \mathit{dom}(\Gamma)}{C; \Gamma \vdash x : \Gamma(x)}$$

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

$$(\textsc{App})\quad \frac{C; \Gamma \vdash e_1 : \tau \texttt{->} \tau', \qquad C; \Gamma \vdash e_2 : \tau}{C; \Gamma \vdash e_1(e_2) : \tau'}$$

$$(\textsc{Record})\quad \frac{C; \Gamma \vdash \overline{e_i : \tau_i}}{C; \Gamma \vdash \{\overline{l_i = e_i}\} : \{\overline{l_i : \tau_i}\}}$$

$$(\textsc{Select})\quad \frac{C; \Gamma \vdash e : \{l : \tau\}}{C; \Gamma \vdash e.l : \tau}$$

$$(\textsc{Ref})\quad \frac{C; \Gamma \vdash e : \tau}{C; \Gamma \vdash \mathsf{ref}\ e : \tau\ \mathsf{Ref}}$$

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

$$(\textsc{Set})\quad \frac{C; \Gamma \vdash e_1 : \tau\ \mathsf{Ref}, \qquad C; \Gamma \vdash e_2 : \tau}{C; \Gamma \vdash \mathsf{set}\ (e_1,\ e_2) : \tau}$$

Figure 4: Selected typing rules of Soop.

C can be derived by the subtyping rules. Rule (inst) makes a polymorphic type a subtype of the instances of its body τ when the types τ_i, replacing (by substitution σ) type variables t_i, satisfy the subtyping relations specified by the constraints $t_i \le \theta_i$. The two rules (fold) and (unfold) establish the equivalence between a recursive type and its unrolling with respect to subtyping.

Along with the standard proof rules for constants and PCF primitives the type system of Soop includes those in Figure 4. In the typing rules, a *type environment* Γ is a finite map in $\mathit{Var} \to \mathit{Typ}$; it is *closed* in a constraint system C if $\Gamma(x)$ is closed in C for each $x \in \mathit{dom}(\Gamma)$. The valid *typing judgements* derived by these rules are of the form $C; \Gamma \vdash e : \tau$ where C is consistent and Γ is closed in C.

Due to space considerations the Soop operational semantics and type soundness proofs cannot be presented in detail here; complete proofs may be found in [9]. Here we must settle for a brief outline. We present semantics of Soop in the general framework of [10, 14]. The operational interpreter of the language is specified by binary relations between Soop terms in memory environments, which represent the notion of computation.

A *computation state* $\langle \Sigma, e \rangle$ is a pair of a memory $\Sigma \in \mathit{Var} \to \mathit{Val}$ and an expression e. This represents the complete state of the computation at any point in time. The memory maps cell names to values assigned to these cells. A single-step computation relation, $\mapsto_1$, is then defined on computation states, precisely capturing the execution behavior of Soop programs. The *computation* relation $\mapsto^*$ is the reflexive transitive closure of $\mapsto_1$. A closed computation state $\langle \Sigma, e \rangle$ is *stuck* if $e \notin \mathit{Val}$ 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 every state $\langle \Sigma', e' \rangle$, if $\langle \Sigma, e \rangle \mapsto^* \langle \Sigma', e' \rangle$, then $\langle \Sigma', e' \rangle \mapsto_1 \langle \Sigma'', e'' \rangle$ for some Σ'' and e''.

Typing of computation states is defined as follows: $\Gamma \vdash \langle \Sigma, e \rangle : \tau$ if $\emptyset; \Gamma \vdash e : \tau$, and $\emptyset; \Gamma \vdash \Sigma(x) : \tau_x$ where $\Gamma(x) = \tau_x\ \mathsf{Ref}$, for each $x \in \mathit{dom}(\Sigma)$.

Following [19] 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.

Theorem 4.1 (Subject Reduction) If $\Gamma \vdash \langle \Sigma, e \rangle : \tau$ and $\langle \Sigma, e \rangle \mapsto_1 \langle \Sigma', e' \rangle$, then there exists Γ' such that $\Gamma' \vdash \langle \Sigma', e' \rangle : \tau$.

Lemma 4.2 (Stuck States Are Not Typable) If $\Gamma \vdash \langle \Sigma, e \rangle : \tau$, then either $e \in \mathit{Val}$, or there exists a state $\langle \Sigma', e' \rangle$ such that $\langle \Sigma, e \rangle \mapsto_1 \langle \Sigma', e' \rangle$.

Theorem 4.3 (Soop Type Soundness) If $\Gamma \vdash \langle \Sigma, e \rangle : \tau$, then either $\langle \Sigma, e \rangle$ diverges, or $\langle \Sigma, e \rangle \mapsto^* \langle \Sigma', v \rangle$ and $\Gamma' \vdash \langle \Sigma', v \rangle : \tau$ for some Γ'.

In comparison with proofs of subject reduction for the λ-calculus [18] and ML-style polymorphic languages [19], the proof for the Soop type system is complicated by the relatively rich subtyping relation. The only analog of subtyping in the ML type system is that a term with a polymorphic type scheme can also be assigned each type produced by an 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 a term e can be constructed only after a significant transformation of the typing derivation of e. Further problems arise from the non-trivial subtyping on some basic Soop types.

The type system of Soop is powerful enough to allow for a type-correct translation of Loop objects (and more); yet it was deliberately weakened by omiting certain proof rules which would have notably complicated the proof of its soundness, or the Loop type checking algorithm (Section 6). Examples are rules for distributing quantifiers over type constructors, and the more sophisticated subtyping relation on recursive types of [1].

5 Loop Type Soundness

In this section we present the Loop type system, and 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.

5.1 The Loop Type System

The syntax of Loop types was given in Figure 1 of Section 2. We take up the discussion of Loop types from that point. In addition to object and class types $\texttt{Obj}(t)M$ and $\texttt{Class}(t)(I;M)$ discussed previously, some additional forms of object and class type are needed for bookkeeping purposes. The type $\texttt{Self}[t](I;M)$ is the type of a class viewed from inside a class definition; there are in fact no terms of this type, since this internal view cannot escape. The internal view given by $\texttt{Self}[t](I;M)$ exposes the instance variables, allowing them to be read and set. $\texttt{PreObj}M$ is the view of what objects of the class will look like from the outside; it is $\texttt{Self}[t](I;M)$ with the instance variables removed. In general, the "self-type" t may occur free in $\texttt{PreObj}M$ and $\texttt{Self}[t](I;M)$, and in the latter expression t is not bound. Such free type variables are bound by type constraints of the form $t \leq \texttt{PreObj}M$.

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

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

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

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

$$(\mathit{Unfold})\ \frac{}{C \vdash \texttt{Obj}(t)M \leq \texttt{PreObj}M[\texttt{Obj}(t)M/t]}$$

Figure 5: Loop subtyping rules

A type constraint system C consists of a sequence of such type constraints; it may equivalently be viewed as a mapping from type variables to their corresponding bounds. Figure 5 axiomatizes the subtyping relation between Loop types. Object and pre-object types can be compared via the rules (*PreObj*), (*Fold*), and (*Unfold*). Two object types are compared by first using (*Fold*)/(*Unfold*) to unwind them, and then (*PreObj*) to compare them componentwise. Thus `PreObj`M types in fact have two purposes, one alluded to above for typing classes, and another for bookkeeping purposes to allow object types to be compared.

The object typing rules for Loop are listed in Figure 6. Rules for functions, `pred`, `succ` and `is_zero` are standard and are omitted for brevity. The (*Class*) rule is the most important 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 is not known, since the class currently being type-checked may have been extended by inheritance. As discussed in Section 3, 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 $t \leq \texttt{PreObj}M$, i.e. it contains at least the methods M. The class name s can be assumed to have type $\texttt{Self}[t](I;M)$. Recall that t is not bound in this expression, so t occuring in M are constrained by the bound $t \leq \texttt{PreObj}M$. This means the self-references in M (and in I) are to the open-ended "self." A similar property holds for the superclass types. 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.

$$(Sub)\ \frac{C;\Gamma \vdash e : \tau, \qquad C \vdash \tau \leq \tau'}{C;\Gamma \vdash e : \tau'} \qquad (Self)\ \frac{\Gamma(s) = \mathtt{Self[}t\mathtt{]}(I;M)}{C;\Gamma \vdash s : t} \qquad (Mesg)\ \frac{C;\Gamma \vdash e : \mathtt{PreObj}M}{C;\Gamma \vdash e\mathtt{<-}m : M(m)}$$

$$(Var)\ \frac{v \in dom(\Gamma)}{C;\Gamma \vdash v : \Gamma(v)} \qquad (Super)\ \frac{\Gamma(u) = \mathtt{Self[}t\mathtt{]}(I;M)}{C;\Gamma \vdash u : \mathtt{PreObj}M} \qquad (Inst)\ \frac{\Gamma(a) = \mathtt{Self[}t\mathtt{]}(I;M)}{C;\Gamma \vdash a.x : I(x)}$$

$$(Cond)\ \frac{C;\Gamma \vdash e_1 : \mathtt{Bool},\ e_2 : \tau,\ e_3 : \tau}{C;\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau} \qquad (New)\ \frac{C;\Gamma \vdash e : \mathtt{Class}(t)(I;M)}{C;\Gamma \vdash \mathtt{new}\ e : \mathtt{Obj}(t)M} \qquad (Assn)\ \frac{\begin{array}{l}\Gamma(s) = \mathtt{Self[}t\mathtt{]}(I;M)\\ C;\Gamma \vdash e : I(x)\end{array}}{C;\Gamma \vdash s.x\mathtt{:=}e : I(x)}$$

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

Figure 6: Selected Loop Type Rules

Rules (*Self*) and (*Super*) type the uses of class and superclass names in the class definition. Note the rules are slightly different. (*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 s is also of type $\mathtt{PreObj}M$. The weaker type $\mathtt{PreObj}M$ is not given to s in the rule as s may be returned by a method; t is the open-ended "self" that would be returned, not merely an object containing the methods defined in this class alone. (*New*) is the obvious rule for typing object creation. When a new object is created, its instance variables are hidden, and the "self-type" t becomes fixed.

5.2 Operational Semantics

The semantics of Loop is defined via a translation of Loop terms into Soop terms; this translation is given in Figure 7. In many encodings of object-oriented languages [7, 13] 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. 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 combinator is then only well-defined on functionals, but as pointed out for instance in [8], classes do not correspond to functionals. 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).

Several solutions with various limitations have been proposed; the reader is referred to [12, 4] and the discussions in [9] for details. Our approach avoids these limitations by defining a fixed point combinator in terms of mutable cells in the spirit of Landin. A first approximation to this is

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

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. Another advantage of this encoding is that "self" may be used in instance variable initializations. This allows for a form of method override in objects (not just classes): a method stored in an instance variable (as a function) may be overridden by setting the variable to a different function.

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 typecheck. 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.

There is not an 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 fixed-point combinator achieves this effect.

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

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 type constraint on the cell.

Let us now examine the encoding of Figure 7 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.

The translation of LOOP types to SOOP types appears in figure 7. Many of the properties of the type translation were sketched above. The most important feature

Terms:

$$\begin{aligned}
\llbracket\, v \,\rrbracket &= v \\
\llbracket\, n \,\rrbracket &= n \\
\llbracket\, b \,\rrbracket &= b \\
\llbracket\, \mathtt{succ}(e) \,\rrbracket &= \mathsf{succ}\ (\llbracket e \rrbracket) \\
\llbracket\, \mathtt{pred}(e) \,\rrbracket &= \mathsf{pred}\ (\llbracket e \rrbracket) \\
\llbracket\, \mathtt{is\_zero}(e) \,\rrbracket &= \mathsf{is\_zero}\ (\llbracket e \rrbracket) \\
\llbracket \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rrbracket &= \mathsf{if}\ \llbracket e \rrbracket\ \mathsf{then}\ \llbracket e_1 \rrbracket\ \mathsf{else}\ \llbracket e_2 \rrbracket \\
\llbracket\, \mathtt{fn}\ v : \tau\ \mathtt{=>}\ e \,\rrbracket &= \lambda v.\, \llbracket e \rrbracket \\
\llbracket\, e_1(e_2) \,\rrbracket &= \llbracket e_1 \rrbracket(\llbracket e_2 \rrbracket) \\
\llbracket\, a \,\rrbracket &= a \\
\llbracket\, a.x \,\rrbracket &= !\ (a(\mathtt{\{\}}).\mathsf{inst}.x) \\
\llbracket\, s.x\mathtt{:=}e \,\rrbracket &= \mathsf{set}\ (s(\mathtt{\{\}}).\mathsf{inst}.x,\ \llbracket e \rrbracket) \\
\llbracket\, e\mathtt{<-}m \,\rrbracket &= \llbracket e \rrbracket(\mathtt{\{\}}).\mathsf{meth}.m \\
\llbracket\, \mathtt{new}\ e \,\rrbracket &= Y'_r(\llbracket e \rrbracket)
\end{aligned}$$

$$\begin{gathered}
\llbracket\, \mathtt{class}\ s : t\ \mathtt{super}\ \overline{u_i\ \mathtt{of}\ e_i} \\
\mathtt{inst}\ \overline{x_j : \tau'_j = e'_j}\ \mathtt{meth}\ \overline{m_k : \tau''_k = e''_k}\ \rrbracket \\
= \\
\lambda s.\, \mathsf{let}\ \overline{u_i = \llbracket e_i \rrbracket(s)}\ \mathsf{in} \\
(\lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \mathsf{x})(\{\mathsf{inst} = \{\overline{x_j = \mathsf{ref}\ \llbracket e'_j \rrbracket}\},\ \mathsf{meth} = \{\overline{m_k = \llbracket e''_k \rrbracket}\}\})
\end{gathered}$$

Types:

$$\begin{aligned}
\llbracket\, \mathtt{Bool} \,\rrbracket &= \mathsf{Bool} \\
\llbracket\, \mathtt{Nat} \,\rrbracket &= \mathsf{Nat} \\
\llbracket\, \tau \rightarrow \tau' \,\rrbracket &= \llbracket \tau \rrbracket\ \mathtt{->}\ \llbracket \tau' \rrbracket \\
\llbracket\, t \,\rrbracket &= \mathtt{\{\}}\ \mathtt{->}\ \{\mathsf{meth} : t\} \\
\llbracket\, \mathtt{PreObj}M \,\rrbracket &= \mathtt{\{\}}\ \mathtt{->}\ \{\mathsf{meth} : \llbracket M \rrbracket\} \\
\llbracket\, \mathtt{Obj}(t)M \,\rrbracket &= \mathtt{\{\}}\ \mathtt{->}\ \{\mathsf{meth} : \mu t.\, \llbracket M \rrbracket\} \\
\llbracket\, \mathtt{Class}(t)(I;M) \,\rrbracket &= \forall [t \leq \llbracket M \rrbracket]. \\
&\qquad \langle\!\langle \llbracket I \rrbracket,\ t \rangle\!\rangle\ \mathtt{->}\ \langle\!\langle \llbracket I \rrbracket,\ \llbracket M \rrbracket \rangle\!\rangle
\end{aligned}$$

Instance Variables and Methods:

$$\begin{aligned}
\llbracket\, \{\overline{x_i : \tau_i}\} \,\rrbracket &= \{\overline{x_i : \llbracket \tau_i \rrbracket\ \mathsf{Ref}}\} \\
\llbracket\, \{\overline{m_i : \tau_i}\} \,\rrbracket &= \{\overline{m_i : \llbracket \tau_i \rrbracket}\}
\end{aligned}$$

Constraints:

$$\llbracket\, t \leq \mathtt{PreObj}M \,\rrbracket = t \leq \llbracket M \rrbracket$$

Hypotheses:

$$\begin{aligned}
\llbracket\, v : \tau \,\rrbracket &= v : \llbracket \tau \rrbracket \\
\llbracket\, s : \mathtt{Self[}t\mathtt{]}(I;M) \,\rrbracket &= s : \langle\!\langle \llbracket I \rrbracket,\ t \rangle\!\rangle \\
\llbracket\, u : \mathtt{Self[}t\mathtt{]}(I;M) \,\rrbracket &= u : \langle\!\langle \llbracket I \rrbracket,\ \llbracket M \rrbracket \rangle\!\rangle
\end{aligned}$$

Judgements:

$$\begin{aligned}
\llbracket\, C \vdash \tau \leq \tau' \,\rrbracket &= \llbracket C \rrbracket \vdash \llbracket \tau \rrbracket \leq \llbracket \tau' \rrbracket \\
\llbracket\, C;\Gamma \vdash e : \tau \,\rrbracket &= \llbracket C \rrbracket;\llbracket \Gamma \rrbracket \vdash \llbracket e \rrbracket : \llbracket \tau \rrbracket
\end{aligned}$$

where

$$\langle\!\langle \tau_I,\ \tau_M \rangle\!\rangle \overset{\mathrm{def}}{=} \mathtt{\{\}}\ \mathtt{->}\ \{\mathsf{inst} : \tau_I,\ \mathsf{meth} : \tau_M\}$$

Figure 7: Translation of LOOP to SOOP

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. The open-endedness is found in the record passed as a parameter, for this may contain more methods than the current class defines. So, we use F-bounded quantification to specify that t is an arbitrary record with at least the methods M. When t occurs free in M, this will allow the methods to have open-ended notions of "self-type."

5.3 Type Soundness of Loop

We now sketch how type soundness may be proven for Loop. Proofs are omitted for lack of space; see [9] for complete proofs. The language studied there is an untyped version of Loop and since the translation ignores explicit type information, proofs are unchanged. The translation of Loop type expressions and judgements into Soop judgements appears in Figure 7.

The following Lemma shows the translation of each Loop rule is a derived rule of Soop.

Lemma 5.1 The translation of the conclusion of each Loop rule is provable in Soop from the translations of the antecedents of the rule.

From this, a simple induction argument allows us to conclude the correctness of Loop typing derivations.

Theorem 5.2 (Loop Typing) If $C;\Gamma \vdash e : \tau$ is provable in Loop, then $[\![C]\!];[\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau]\!]$ is provable in Soop.

Definition 5.3 A Loop program e becomes *stuck* if its corresponding Soop translation $[\![e]\!]$ becomes stuck upon computing.

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 computation states, and thus Loop programs that have such translations are considered erroneous under this definition. From Theorem 5.2 and Theorem 4.3, the soundness of Soop, we may conclude that no type-correct Loop programs may become stuck.

Theorem 5.4 (Soundness of Loop Typing) Typable Loop programs do not become stuck.

6 Decidability of Loop Type Checking

We may establish the decidability of type-checking for Loop. This construction is carried out directly in the Loop type system, because Soop type-checking is undecidable.

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

$$(\mathit{Hyp}')\ \frac{C \vdash C(t) \le \tau}{C \vdash t \le \tau} \qquad (\mathit{Fun})\ \frac{C \vdash \tau_1' \le \tau_1, \qquad \tau_2 \le \tau_2'}{C \vdash \tau_1 \to \tau_2 \le \tau_1' \to \tau_2'}$$

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

$$(\mathit{Fold}')\ \frac{C \vdash \texttt{PreObj}M' \le \texttt{PreObj}M[\texttt{Obj}(t)M/t]}{C \vdash \texttt{PreObj}M' \le \texttt{Obj}(t)M}$$

$$(\mathit{Unfold}')\ \frac{C \vdash \texttt{PreObj}M[\texttt{Obj}(t)M/t] \le \tau}{C \vdash \texttt{Obj}(t)M \le \tau}$$

Figure 8: Alternative axiomatization of Loop subtyping

6.1 Decidability of Loop Subtyping

The Loop subtyping relation, axiomatized by the rules listed in Figure 5, has a straightforward decision procedure. To show this, we first introduce an alternative set of Loop subtyping rules, listed in Figure 8.

Lemma 6.1 The set of rules in Figure 8 is equivalent to those in Figure 5.

Proof: Each alternate form of a rule is derivable from its original version and rule (*Refl*). Conversely, each original rule is derivable from its alternate form and rule (*Trans*). □

Thus, both sets of rules define the same subtyping relation. For the remainder of this section, we shall use the alternate set of rules in Figure 8.

Lemma 6.2 (Head Forms) If $C \vdash \tau_1 \le \tau_2$ is provable, then it has one of the following forms:

$$\begin{array}{l}
C \vdash \tau \le \tau \\
C \vdash t \le \texttt{PreObj}M \\
C \vdash t \le \texttt{Obj}(t')M \\
C \vdash \tau_1 \to \tau_2 \le \tau_1' \to \tau_2' \\
C \vdash \texttt{PreObj}M \le \texttt{PreObj}M' \\
C \vdash \texttt{PreObj}M \le \texttt{Obj}(t')M' \\
C \vdash \texttt{Obj}(t)M \le \texttt{PreObj}M' \\
C \vdash \texttt{Obj}(t)M \le \texttt{Obj}(t')M'
\end{array}$$

Proof: By induction on subtyping proofs. □

Lemma 6.3 If $C \vdash \tau \le \tau'$ is provable with a single use of (*Trans*) as the last rule in the proof, then it is provable without using (*Trans*).

Proof: By induction on the depth of proof trees. If the final rule used in the proof is (*Trans*), then the judgement is proven from judgements $C \vdash \tau \le \tau''$, $C \vdash$

$\tau'' \leq \tau'$, and (*Trans*), for some type τ''. The case $\tau = \texttt{Obj}(t)M$, $\tau' = \texttt{Obj}(t')M'$ will be shown to illustrate the general proof technique.

If the proof of $C \vdash \texttt{Obj}(t)M \leq \texttt{Obj}(t')M'$ has an instance of (*Trans*) as its final proof step, the antecedents of this rule have the form $C \vdash \texttt{Obj}(t)M \leq \tau''$ and $C \vdash \tau'' \leq \texttt{Obj}(t')M'$. A simple induction argument shows that τ'' must be either $\texttt{Obj}(t'')M''$ or $\texttt{PreObj}M''$. Considering each case:

(i) If τ'' is $\texttt{PreObj}M''$, then the antecedent judgements must have been proven via rules (*Unfold'*) and (*Fold'*), respectively. The antecedents of these rules in turn must have been proven via rule (*PreObj*). Thus, for each $m \in dom(M')$, we have proofs of $C \vdash M(m)[\texttt{Obj}(t)M/t] \leq M''(m)$ and $C \vdash M''(m) \leq M'(m)[\texttt{Obj}(t')M'/t']$. These can be combined by rule (*Trans*); however, the resulting proofs are shorter than the original, and hence these uses of (*Trans*) can be eliminated by induction. From these proofs we can construct a new proof of $C \vdash \texttt{Obj}(t)M \leq \texttt{Obj}(t')M'$ by rules (*PreObj*), (*Fold'*) and (*Unfold'*).

(ii) τ'' is $\texttt{Obj}(t'')M''$. Then antecedents are proven via the (*Fold'*) and (*Unfold'*) rules, and the proof proceeds in the same way as above. $\square$

A simple induction argument on proofs then proves:

THEOREM 6.4 (TRANSITIVITY ELIMINATION) If $C \vdash \tau \leq \tau'$ is provable, then it is provable without the use of (*Trans*).

Therefore, (*Trans*) may be eliminated from the system of subtyping rules without changing the subtyping relation axiomatized by the rules. Furthermore, the top-level type constructors of the conclusions of the rules are unique (ignoring rule (*Refl*)); for every judgement $C \vdash \tau \leq \tau'$ if $\tau \neq \tau'$ there is at most one rule instance that has this judgement as its conclusion.

DEFINITION 6.5 A *canonical proof* of judgement $C \vdash \tau \leq \tau'$ is a proof whose final rule is:

(i) (*Refl*), if $\tau = \tau'$
(ii) not (*Trans*), if $\tau \neq \tau'$, and whose antecedents are canonical proofs.

LEMMA 6.6 (CANONICAL SUBTYPING) If $C \vdash \tau \leq \tau'$ is provable, it has a unique canonical proof that does not "loop"; i.e., it does not contain a subproof with conclusion $C \vdash \tau \leq \tau'$.

PROOF: If $C \vdash \tau \leq \tau'$ is provable, then by Theorem 6.4 it has a proof that does not use rule (*Trans*). The proof then proceeds by induction on the length of these (*Trans*)-free proofs.

If $\tau = \tau'$, then $C \vdash \tau \leq \tau'$ has a unique canonical proof via rule (*Refl*). Otherwise, its final rule is not (*Trans*), and this rule has antecedents with unique canonical proofs by induction. This proof can be shown to be unique by case analysis on the allowable subtyping judgement forms of Lemma 6.2.

Obviously, such canonical proofs cannot have an antecedent of $C \vdash \tau \leq \tau'$, for the proof of this would yield a different, shorter canonical proof of $C \vdash \tau \leq \tau'$. $\square$

DEFINITION 6.7 If τ is a type and C a type constraint system, $S(\tau, C)$, the *subterm closure* of τ and C, is the least set of types such that:

(i) $\tau \in S(\tau, C)$
(ii) if $t \in S(\tau, C)$, then $C(t) \in S(\tau, C)$
(iii) if $\tau_1 \rightarrow \tau_2 \in S(\tau, C)$, then $\{\tau_1, \tau_2\} \subseteq S(\tau, C)$
(iv) if $\texttt{PreObj}M \in S(\tau, C)$, then $M(m) \in S(\tau, C)$ for each $m \in dom(M)$
(v) if $\texttt{Obj}(t)M \in S(\tau, C)$, then $\texttt{PreObj}M[\texttt{Obj}(t)M/t] \in S(\tau, C)$.

LEMMA 6.8 For any type constraint system C:

(i) $S(C(t), C) \subseteq S(t, C)$, for each $t \in dom(C)$.
(ii) $S(\tau, C) \cup S(\tau', C) \subseteq S(\tau \rightarrow \tau', C)$.
(iii) $S(M(m), C) \subseteq S(\texttt{PreObj}M, C)$, $\forall m \in dom(M)$.
(iv) $S(\texttt{PreObj}M[\texttt{Obj}(t)M/t], C) \subseteq S(\texttt{Obj}(t)M, C)$.

PROOF: By induction on the definition of $S(\tau, C)$. $\square$

LEMMA 6.9 For any type τ and type constraint system C, $S(\tau, C)$ is finite.

PROOF: The definition of $S(\tau, C)$ can be viewed as an algorithm for generating its elements. We may extend the language of types to include a "marked" equivalent of the $\texttt{Obj}(t)M$ type, $\underline{\texttt{Obj}}(t)M$, and change the last case in the definition of $S(\tau, C)$ to unroll $\texttt{Obj}(t)M$ to $\texttt{PreObj}M[\underline{\texttt{Obj}}(t)M/t]$. Since the marked type $\underline{\texttt{Obj}}(t)M$ is not unrolled, this defines a new set $S'(\tau, C)$ which can be shown to be finite by induction on the size of τ. Furthermore, $S(\tau, C)$ is the image of $S'(\tau, C)$ under the "unmarking" operation. Thus, $S(\tau, C)$ must be finite as well. $\square$

LEMMA 6.10 Any proof of $C \vdash \tau \leq \tau'$ uses only antecedents of the form $C \vdash \tau_1 \leq \tau_2$ where $\{\tau_1, \tau_2\} \subseteq S(\tau, C) \cup S(\tau', C)$.

PROOF: By induction on proofs, inspection of the proof rules, and Lemma 6.8. $\square$

THEOREM 6.11 (DECIDABLE SUBTYPING) It is decidable whether or not any subtyping judgement $C \vdash \tau \leq \tau'$ has a proof.

PROOF: Lemma 6.6 implies a semi-decision procedure for the subtyping relation; if $C \vdash \tau \leq \tau'$ is provable, the procedure outlined will generate a proof of it. Otherwise, for some antecedent judgement J, the procedure loops, generating J again as a subgoal, by Lemmas 6.10 and 6.9. But by Lemma 6.6, J therefore has no canonical proof. Thus, the original judgement must be unprovable. □

6.2 Minimal Types of LOOP terms

To make the type-checking system of LOOP practical, we only require that the program contains type information about each identifier, variable and method; the system has to infer the rest in a type-correct program, and reject type-incorrect ones. In some cases however a term e can be proved to have more than one type (e.g. using the (*Sub*) typing rule); therefore claiming to have an inference system makes sense only if it is able to find a typing for e from which all others follow, i.e. a type which is smaller than all other possible types for e.

DEFINITION 6.12 (MINIMAL TYPES) τ is a *minimal type* for a LOOP term e in environment $C;\Gamma$ if $C;\Gamma \vdash e : \tau$ and for each type τ', if $C;\Gamma \vdash e : \tau'$, then $C \vdash \tau \leq \tau'$.

Proving the existence of a minimal type for every typable LOOP term is therefore a prerequisite to proving the correctness of the type-checking algorithm. As it turns out, computing the minimal type is non-trivial only in the case of a conditional expression: the type τ of the result must be the least type larger than the (minimal) types τ_1 and τ_2 of the terms in both branches, i.e. it must be their *least upper bound.* Since the subtyping relation is only defined with respect to a constraint system C, so is the lub; it is a partial function on pairs of type terms — some pairs (`Nat` and `Bool`, for instance) have no common upper bound.

To prove that the lub exists whenever the types have an upper bound, we first give an algorithm for computing it (Figure 9). For brevity it is presented in the form of a nondeterministic system of rules which makes use of the symmetry of the lub to define it as a function on 2-element sets of types (identity on types is considered up to renaming of bound type variables). The ambiguity is harmless because of the Church-Rosser property exhibited by the system: if two rules can be applied, the result is independent of the order of application (this situation only arises when each of the two types is either a type variable or an object type). The additional parameters $S_\vee$ and $S_\wedge$ are provided in order to guarantee the termination of the algorithm (the argument follows in the steps of the proof of termination of the algorithm for subtype checking presented in Section 6.1). The computation fails if the arguments do not match the patterns in any rule, or if a subgoal (invocation of LUB or GLB) fails.

LEMMA 6.13 Given a constraint system C and two types τ_1 and τ_2, if $\tau_1 \vee_C \tau_2 = \tau$, then τ is a least upper bound of τ_1 and τ_2 with respect to C, and if $\tau_1 \vee_C \tau_2$ fails, then τ_1 and τ_2 have no upper bounds.

PROOF: We prove the following proposition about the function LUB:

> If $LUB\ (\{\tau_1,\tau_2\},\ S_\vee,\ S_\wedge) = \tau$, then τ is a least upper bound of τ_1 and τ_2; if $LUB\ (\{\tau_1,\tau_2\},\ S_\vee,\ S_\wedge)$ fails, then for every τ such that $C \vdash \tau_1 \leq \tau$ and $C \vdash \tau_2 \leq \tau$, for every pair of proofs of these judgements there exist types τ_1', τ_2' and τ' such that $\{\tau_1',\tau_2'\} \in S_\vee$, and $C \vdash \tau_1' \leq \tau'$ and $C \vdash \tau_2' \leq \tau'$ are intermediate steps of the respective proofs

(and simultaneously the analogous statement about GLB). The proof proceeds by induction with respect to the metric we used to demonstrate termination of these algorithms. We illustrate the argument in the following cases:

CASE $\tau_1 = t$ and $\tau_2 \neq t$:

If $\{t,\tau_2\} \notin S_\vee$ and $LUB\ (\{C(t),\tau_2\},\ S_\vee',\ S_\wedge) = \tau$, then by inductive hypothesis $C \vdash C(t) \leq \tau$ and $C \vdash \tau_2 \leq \tau$; by an application of (*Hyp*$'$) from the former we obtain $C \vdash t \leq \tau$, i.e. τ is an upper bound of t and τ_2. Suppose that τ' is also their upper bound. The proof of $C \vdash t \leq \tau'$ may not result from an application of (*Refl*), since this implies that $C \vdash \tau_2 \leq t$, which (by Lemma 6.2) is only possible if $\tau_2 = t$, contradicting our assumption. Hence $C \vdash t \leq \tau'$ is proved by an instance of (*Hyp*$'$), and therefore we must have a proof of the antecedent of this rule, namely $C \vdash C(t) \leq \tau'$. But then $C \vdash \tau \leq \tau'$ by the inductive hypothesis.

Consider the case when $\{t,\tau_2\} \notin S_\vee$ and $LUB\ (\{C(t),\tau_2\},\ S_\vee',\ S_\wedge)$ has failed. Suppose t and τ_2 have an upper bound τ, and consider any two proofs of their relations. By the same argument as above every proof of $C \vdash t \leq \tau$ contains a proof of $C \vdash C(t) \leq \tau$, and by inductive hypothesis we have some τ_1', τ_2' and τ' such that $\{\tau_1',\tau_2'\} \in S_\vee'$ and the judgements $C \vdash \tau_1' \leq \tau'$ and $C \vdash \tau_2' \leq \tau'$ appear in the respective proofs of $C \vdash C(t) \leq \tau$ and $C \vdash \tau_2 \leq \tau$. It now only remains to be shown that the case of $\tau_1' = t$ and $\tau_2' = \tau_2$ is impossible (and therefore $\{\tau_1',\tau_2'\} \in S_\vee$). Indeed, if it was, it would mean that there is no pair of shortest proofs that τ is an upper bound or t and τ_2 — since *every* such pair contains subproofs (at least one of them shorter) of the

$$
\begin{array}{ll}
\multicolumn{2}{l}{\tau_1 \vee_C \tau_2 = \mathit{LUB}\ (\{\tau_1, \tau_2\},\ \emptyset,\ \emptyset)} \\
\multicolumn{2}{l}{\text{where}} \\
\multicolumn{2}{l}{\mathit{LUB}\ (\{\tau_1, \tau_2\},\ S_\vee,\ S_\wedge) =} \\
\multicolumn{2}{l}{\quad \text{if } \tau_1 = \tau_2, \text{ then } \tau_1, \text{ else if } \{\tau_1, \tau_2\} \notin S_\vee, \text{ then}} \\
\quad\quad \mathit{LUB}\ (\{C(t), \tau_2\},\ S'_\vee,\ S_\wedge), & \text{if } \tau_1 = t \\
\quad\quad \mathit{LUB}\ (\{\texttt{PreObj}M[\tau_1/t], \tau_2\},\ S'_\vee,\ S_\wedge), & \text{if } \tau_1 = \texttt{Obj(}t\texttt{)}M \\
\quad\quad \texttt{PreObj}M, & \text{if } \tau_1 = \texttt{PreObj}M_1,\ \tau_2 = \texttt{PreObj}M_2 \\
 & \text{where } \mathit{dom}(M) = \mathit{dom}(M_1) \cap \mathit{dom}(M_2) \\
 & \text{and } M(m) = \mathit{LUB}\ (\{M_1(m), M_2(m)\},\ S'_\vee,\ S_\wedge) \\
 & \quad\quad \text{for all } m \in \mathit{dom}(M) \\
\quad\quad \mathit{GLB}\ (\{\tau'_1, \tau'_2\},\ S'_\vee,\ S_\wedge) \rightarrow \mathit{LUB}\ (\{\tau''_1, \tau''_2\},\ S'_\vee,\ S_\wedge), & \text{if } \tau_1 = \tau'_1 \rightarrow \tau''_1,\ \tau_2 = \tau'_2 \rightarrow \tau''_2 \\
\multicolumn{2}{l}{\quad\ \text{where } S'_\vee = S_\vee \cup \{\{\tau_1, \tau_2\}\}} \\
\multicolumn{2}{l}{\text{and}} \\
\multicolumn{2}{l}{\mathit{GLB}\ (\{\tau_1, \tau_2\},\ S_\vee,\ S_\wedge) =} \\
\multicolumn{2}{l}{\quad \text{if } \tau_1 = \tau_2, \text{ then } \tau_1, \text{ else if } \{\tau_1, \tau_2\} \notin S_\wedge, \text{ then}} \\
\quad\quad t, & \text{if } \tau_1 = t \text{ and } C \vdash t \leq \tau_2 \\
\quad\quad \mathit{GLB}\ (\{\texttt{PreObj}M[\tau_1/t], \tau_2\},\ S_\vee,\ S'_\wedge), & \text{if } \tau_1 = \texttt{Obj(}t\texttt{)}M \\
\quad\quad \texttt{PreObj}M, & \text{if } \tau_1 = \texttt{PreObj}M_1,\ \tau_2 = \texttt{PreObj}M_2 \\
 & \text{where } \mathit{dom}(M) = \mathit{dom}(M_1) \cup \mathit{dom}(M_2), \\
 & M(m) = M_1(m) \text{ for all } m \in \mathit{dom}(M_1) \setminus \mathit{dom}(M_2), \\
 & M(m) = M_2(m) \text{ for all } m \in \mathit{dom}(M_2) \setminus \mathit{dom}(M_1), \\
 & \text{and } M(m) = \mathit{GLB}\ (\{M_1(m), M_2(m)\},\ S_\vee,\ S'_\wedge) \\
 & \quad\quad \text{for all } m \in \mathit{dom}(M_1) \cap \mathit{dom}(M_2) \\
\quad\quad \mathit{LUB}\ (\{\tau'_1, \tau'_2\},\ S_\vee,\ S'_\wedge) \rightarrow \mathit{GLB}\ (\{\tau''_1, \tau''_2\},\ S_\vee,\ S'_\wedge), & \text{if } \tau_1 = \tau'_1 \rightarrow \tau''_1,\ \tau_2 = \tau'_2 \rightarrow \tau''_2 \\
\multicolumn{2}{l}{\quad\ \text{where } S'_\wedge = S_\wedge \cup \{\{\tau_1, \tau_2\}\}}
\end{array}
$$

$$
\begin{array}{rl}
\mathit{Type}\ (C, \Gamma;\ v) = \Gamma(v), & \text{if } v \in \mathit{dom}(\Gamma) \\
\mathit{Type}\ (C, \Gamma;\ n) = \texttt{Nat} & \\
\mathit{Type}\ (C, \Gamma;\ b) = \texttt{Bool} & \\
\mathit{Type}\ (C, \Gamma;\ \delta(e)) = \texttt{Nat}, & \text{if } \mathit{Type}\ (C, \Gamma;\ e) = \texttt{Nat},\ \delta \in \{\texttt{succ}, \texttt{pred}\} \\
\mathit{Type}\ (C, \Gamma;\ \texttt{is\_zero}(e)) = \texttt{Bool}, & \text{if } \mathit{Type}\ (C, \Gamma;\ e) = \texttt{Nat} \\
\mathit{Type}\ (C, \Gamma;\ \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2) = \mathit{Type}\ (C, \Gamma;\ e_1) \vee_C \mathit{Type}\ (C, \Gamma;\ e_2), & \text{if } \mathit{Type}\ (C, \Gamma;\ e) = \texttt{Bool} \\
\mathit{Type}\ (C, \Gamma;\ \texttt{fn}\ v : \tau\ \texttt{=>}\ e) = \tau \rightarrow \mathit{Type}\ (C, (\Gamma,\ v : \tau);\ e) & \\
\mathit{Type}\ (C, \Gamma;\ e(e')) = \tau, & \text{if } \mathit{Type}\ (C, \Gamma;\ e) = \tau' \rightarrow \tau \text{ and } C \vdash \mathit{Type}\ (C, \Gamma;\ e') \leq \tau' \\
\mathit{Type}\ (C, \Gamma;\ s) = t, & \text{if } \Gamma(s) = \texttt{Self[}t\texttt{](}I\texttt{;}M\texttt{)} \\
\mathit{Type}\ (C, \Gamma;\ u) = \texttt{PreObj}M, & \text{if } \Gamma(u) = \texttt{Self[}t\texttt{](}I\texttt{;}M\texttt{)} \\
\mathit{Type}\ (C, \Gamma;\ a\texttt{.}x) = I(x), & \text{if } \Gamma(a) = \texttt{Self[}t\texttt{](}I\texttt{;}M\texttt{)} \\
\mathit{Type}\ (C, \Gamma;\ s\texttt{.}x\texttt{:=}e) = I(x), & \text{if } \Gamma(s) = \texttt{Self[}t\texttt{](}I\texttt{;}M\texttt{)} \text{ and } C \vdash \mathit{Type}\ (C, \Gamma;\ e) \leq I(x) \\
\mathit{Type}\ (C, \Gamma;\ s\texttt{<-}m) = M(m), & \text{if } C(\mathit{Type}\ (C, \Gamma;\ s)) = \texttt{PreObj}M \\
\mathit{Type}\ (C, \Gamma;\ e\texttt{<-}m) = M(m), & \text{if } \mathit{Type}\ (C, \Gamma;\ e) = \texttt{PreObj}M \\
\mathit{Type}\ (C, \Gamma;\ \texttt{new}\ e) = \texttt{Obj(}t\texttt{)}M, & \text{if } \mathit{Type}\ (C, \Gamma;\ e) = \texttt{Class(}t\texttt{)(}I\texttt{;}M\texttt{)}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Type}\ (C, \Gamma;\ \texttt{class}\ s : t\ \texttt{super}\ \overline{u_i\ \texttt{of}\ e_i}\ \texttt{inst}\ \overline{x_j : \tau'_j \texttt{=} e'_j}\ \texttt{meth}\ \overline{m_k : \tau''_k \texttt{=} e''_k}) \\
\qquad = \texttt{Class(}t\texttt{)(}I\texttt{;}M\texttt{)},\ \text{where } I = \{\overline{x_j\ :\ \tau'_j}\},\ M = \{\overline{m_k\ :\ \tau''_k}\} \\
\qquad \text{if } \overline{\mathit{Type}\ (C, \Gamma;\ e_i) = \texttt{Class(}t_i\texttt{)(}I_i\texttt{;}M_i\texttt{)}}, \\
\qquad\quad \overline{C' \vdash \mathit{Type}\ (C', \Gamma';\ e'_j) \leq \tau'_j},\ \ \overline{C' \vdash \mathit{Type}\ (C', \Gamma';\ e''_k) \leq \tau''_k}, \\
\qquad\quad \overline{C' \vdash t \leq \texttt{PreObj}M_i[t/t_i]},\ \text{and } \overline{I(x) = I_i(x)[t/t_i]\ \forall x \in \mathit{dom}(I_i)} \\
\qquad\quad \text{where } C' = (C,\ t \leq \texttt{PreObj}M) \\
\qquad\quad\qquad \Gamma' = (\Gamma,\ s : \texttt{Self[}t\texttt{](}I\texttt{;}M\texttt{)},\ \overline{u_i : (\texttt{Self[}t_i\texttt{](}I_i\texttt{;}M_i\texttt{)})[t/t_i]})
\end{array}
$$

Figure 9: Algorithms for computing the lub of two types and for type-checking LOOP terms

same fact about some τ' — and hence no (finite) proofs exist.

In the remaining case of $\{t, \tau_2\} \in S_\vee$ the statement follows trivially (for $\tau_1' = t$, $\tau_2' = \tau_2$, and $\tau' = \tau$).

CASE $\tau_1 = \texttt{PreObj}M_1$ and $\tau_2 = \texttt{PreObj}M_2$:

By Lemma 6.2 an upper bound of these may only be an object or a pre-object type; but since proving that a pre-object type is a subtype of an object type is only possible via the (*Fold'*) rule, we may focus on upper bounds in pre-object form.

It is easy to see that $\texttt{PreObj}M$ (as defined in the corresponding rule of Figure 9), when it exists, is an upper bound of τ_1 and τ_2. Since any upper bound must define the fields in $dom(M)$, and the types of these fields are upper bounds of those of M_1 and M_2 (so that the antecedents of the only applicable rule (*PreObj*) are satisfied), it immediately follows (again by (*PreObj*) and inductive hypothesis) that any upper bound is a supertype of $\texttt{PreObj}M$. Conversely, if one of the subgoals $LUB\ (\{M_1(m), M_2(m)\},\ S_\vee',\ S_\wedge)$ fails, the second part of the proposition follows by an argument similar to the one given in the previous case.

The statement of the Lemma is a corollary of this proposition and the definition of $\tau_1 \vee_C \tau_2$. □

We are now ready to present a type-checking algorithm for Loop (Figure 9) in the form of a function *Type* $(C, \Gamma;\ e)$ which produces the minimal type of the term e in environment $C; \Gamma$, if it exists, and is undefined otherwise.

THEOREM 6.14 (DECIDABLE TYPE CHECKING) The Loop typing judgement $C; \Gamma \vdash e : \tau$ is provable if and only if *Type* $(C, \Gamma;\ e)$ is defined, and $C \vdash$ *Type* $(C, \Gamma;\ e) \leq \tau$.

PROOF: By structural induction on Loop terms. Because of the lack of non-trivial subtyping on class types, there exists at most one type for each class term (all type information for a class is explicit in the syntax); the algorithm only verifies that the class expression is type correct. As an example we present the proof in the case of a conditional expression.

Let $e = \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. An inspection of the typing rules (Figure 6) shows that the only way to prove $C; \Gamma \vdash e : \tau$ is by an application of (*Cond*) (followed possibly by applications of (*Sub*)). Hence the antecedents of (*Cond*) must also be provable, and by the inductive hypothesis *Type* $(C, \Gamma;\ e_1)$ and *Type* $(C, \Gamma;\ e_2)$ produce minimal types τ_1 and τ_2 of e_1 and e_2 respectively. Therefore the type in the conclusion of (*Cond*) is an upper bound of τ_1 and τ_2, and by Lemma 6.13 it is a supertype of $\tau_1 \vee_C \tau_2$.

The converse part (correctness of the inferred type) follows trivially. □

Termination of the algorithm follows similarly by induction on the structure of Loop terms, and by termination of the algorithms for subtype checking and computing least upper bounds.

7 Discussion

We have shown here how the ideas of [5] used to model typed functional OOP can be applied to Loop, a state-based OOP language with decidable type-checking. We also explored the deficiencies that arise from the fact that subtyping is not inheritance, and showed how fixed-self typings may thus sometimes be preferable. The problem of OOP typing cannot be said to be completely solved, however. As the example in Section 3 illustrates, there is a battle being fought between inheritance and subtyping, and in Loop the programmer is forced to take sides by taking a fixed or open view of self. What would be desirable is 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 [11]. See [8] for more discussion of how the open-ended view of "self" relates to other approaches in the literature.

Bruce and van Gent [4] have defined an imperative OOP language, TOIL, which is an imperative extension of TOOPL [2] (a decidability proof for type-checking of another variant of TOOPL is presented in [3]). TOIL and Loop are closely related, since both are based on F-bounded quantification; here we outline some differences. We interpret Loop via translation to Soop, while TOIL is given semantics directly. The TOIL subtyping rules do not allow for folding and unfolding of object types, and this means the fixed-self typings will not be possible in TOIL. On the other hand Loop does not have the subtyping rule for objects types of TOIL (adopted in a restricted form from the system of [1]), so there are programs using open-ended "self-type" typable in TOIL but not in Loop; however the TOIL rule disallows negative occurrences of "self-type" in the supertype, which limits its applicability to infrequent cases. 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; TOIL does not.

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 (though preserving decidable type-checking may be more difficult). This extension could be desirable for programming: as it stands, only the "self-type" can be open-ended, but it may also be necessary to allow a method to take as parameter an open-ended object from some other hierarchy. Classes are not fully "first-class citizens": we cannot write a function that takes an arbitrary class as argument and returns a class with some methods added. To achieve this, some notion of extensible record would need to be added to Soop [17].

References

[1] R. Amadio and L. Cardelli. Subtyping recursive types. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, 1991.

[2] 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.

[3] 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.

[4] Kim B. Bruce and Robert van Gent. TOIL: A new type-safe object-oriented imperative language. Technical report, Williams College, 1993.

[5] 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.

[6] 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.

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

[8] 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.

[9] J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. An interpretation of typed OOP in a language with state. *Lisp and Symbolic Computation*, 1995. To appear.

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

[11] Giorgio Ghelli. A static type system for message passing. In *Proc. OOPSLA*, pages 129–145, 1991.

[12] 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.

[13] Samuel N. Kamin and Uday S. Reddy. *Two Semantic Models of Object-Oriented Languages*, chapter 13, pages 464–495. MIT Press, 1994.

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

[15] 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.

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

[17] 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, 1993.

[18] J.P. Seldin. A sequent calculus for type assignment. *Journal of Symbolic Logic*, 42:11–28, 1977.

[19] 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*.