Sound Polymorphic Type Inference for Objects

Jonathan Eifrig*† Scott Smith* Valery Trifonov*†

Department of Computer Science, The Johns Hopkins University

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

Abstract

A polymorphic, constraint-based type inference algorithm for an object-oriented language is defined. A generalized form of type, polymorphic *recursively constrained* types, are inferred. These types are expressive enough for typing objects, since they generalize recursive types and F-bounded polymorphism. The well-known tradeoff between inheritance and subtyping is mitigated by the type inference mechanism. Soundness and completeness of type inference are established.

1 Introduction

Type inference, the process of automatically inferring type information from untyped programs, is originally due to Hindley and Milner [16]. These ideas have found their way into some recent innovative programming languages, including Standard ML [17]. The type inference problem for object-oriented languages is a challenging one: even simple object-oriented programs require quite advanced features to be present in the type system. One of the main sources of difficulty lies with binary methods, such as an add method on Number objects. The problems arise when classes containing binary methods are subclassed: in this process the type of "self" changes, and some notion of operator polymorphism or F-bounded polymorphism is needed to capture this behavior type-theoretically [9, 6]. In addition the object types are self-referential (*e.g.* the type of the argument to add is the same as the object's type), so a form of recursive/self-referential type is also needed.

We define a polymorphic, constraint-based type inference algorithm for an object-oriented language, I-Loop. Our language incorporates standard notions of class, multiple inheritance, object creation, message dispatch, mutable instance variables, and hiding of instance variables. Thus, there is enough of a core that the ideas should scale up to an implemented language. We define an algorithm to automatically infer a generalized form of type, *recursively constrained* (rc) types of the form $\tau \setminus C$, with " $\setminus$ " reading "where." Types of similar expressive power have been defined by Curtis [10], and Aiken and Wimmers [2]. τ here is a type, and C is a set of type constraints, each of the form $\tau_1 \leq \tau_2$. These constraints may be *recursive* in that a variable t could occur free in both τ_1 and τ_2. For example, the rc type

$$\mathsf{t} \setminus \{\mathsf{Nat}\rightarrow\mathsf{t} \leq \mathsf{t},\ \mathsf{t} \leq \mathsf{Nat}\rightarrow\mathsf{t}\}$$

is equivalent to the recursive type $\mu\mathsf{t}.\,\mathsf{Nat}\rightarrow\mathsf{t}$. This form of type is thus a generalization of recursive types. Polymorphic rc types have the form $\forall t_1, \ldots, t_n.\, \tau \setminus C$ where constraints $\tau_1 \leq \tau_2$ in C may contain type variables $t_1, \ldots t_n$ free. This type language is strong enough for typing objects, since recursive constraints generalize recursive types, and rc-polymorphism generalizes F-bounded polymorphism. The polymorphic types inferred are analogous to the so-called **let**-polymorphic types of the Hindley/Milner system. Additional expressive power is gained from the ability to have multiple upper and lower bounds on the same type variable in a rc type. These constraints can be understood as defining a restricted form of union and intersection type: $\{\tau \leq t, \tau' \leq t\}$ is conceptually equivalent to $\{\tau \vee \tau' \leq t\}$ in a language with union types $\tau \vee \tau'$; a dual relationship exists between intersections and upper bounds.

By using this form of type in an object-oriented language, detailed rc types can be inferred that are far more precise than any type a programmer would be likely to specify by hand. And, this will in turn give the programmer a degree of flexibility well beyond what is provided by current typed object-oriented languages. In particular, there is a well-known trade-off between subtyping and inheritance. Most object-oriented languages require the choice of one approach exclusive of the other. For instance in C++ and Object Pascal, inheritance is subtyping, so binary methods will not inherit properly. In PolyTOIL [5] and our Loop language [12], inheritance is not subtyping, so binary methods inherit properly but object subtyping may be limited. Our type inference al-

*Partially supported by NSF grants CCR-9109070 and CCR-9301340

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

gorithm is capable of inferring a most general type that in effect postpones the choice of subtyping or binary method inheritance to the point the objects are used, thus allowing for significantly greater flexibility without requiring the language designer or programmer to make a choice between the two.

We have previously defined a language, I-SOOP, that is the non-object-oriented analogue of I-LOOP. I-SOOP's features include rc types, **let**-polymorphism, and type inference [11]. The current paper shows how these ideas can be applied to an actual object-oriented language. We additionally consider issues of type simplification, since the number of constraints produced by the inference algorithm can be large. The main technical results of the paper are the soundness and completeness of the type inference system. The I-LOOP language and typing rules are shown sound by translation into I-SOOP. A similar methodology was carried out in a paper presented at a previous OOPSLA conference [12]. That paper was concerned with the problem of type *checking* for object-oriented languages, not type inference. There we developed a type-checking algorithm for LOOP, and proved it sound by translation into the non-object-oriented SOOP language. In both that paper and the present one, the terms and types of (I-)LOOP are translated into (I-)SOOP, and soundness means that if an (I-)LOOP program is typable, its translation into (I-)SOOP is also typable. I-SOOP has been defined and proven sound in [11]. It would also be possible to give a soundness proof for I-LOOP directly, but at this stage in the project we feel the insights gained by a translational approach are worth the costs. See for instance [1] for ideas on how objects may be directly given meaning. The second main result we obtain for I-LOOP is the completeness of type inference, establishing that the type inference algorithm will always infer a type if there is one.

1.1 Related Work

Many of the papers on constraints and type inference are not directly concerned with type inference for objects. Three classic papers on subtyping and type constraints are [24, 7, 18]. Constrained type inference algorithms that are somewhat weaker than the approach we take have been developed for conventional languages [28, 14, 25]. The stronger version of types with recursive constraints, polymorphism, and type inference was first formulated by Curtis in his purely functional PLEAT language [10]. His work has not received widespread circulation, however. We learned of these richer types through the work of Aiken and Wimmers [2], which demonstrates soundness of a full type inference algorithm for a simple functional language; Curtis only showed soundness of a weaker inference algorithm (he however conjectured soundness of a full algorithm). We first established completeness of type inference, *i.e.* that the inference algorithm will find a typing if there exists one, and were the first to incorporate state, which required a completely different proof technique for soundness [11]. Aiken and Wimmers have not addressed the problem of typing object-oriented programs, and their language also lacks important features such as state and records necessary for encoding objects. This paper presents the first object-oriented language to incorporate rc types and type inference, and shows the soundness and completeness of such a type discipline. state in particular poses difficulty to requires a new proof technique.

In the realm of object-oriented languages, Bruce *et al.* [5] have developed an expressive object-oriented programming language, PolyTOIL, with decidable type-checking (not inference), which is similar to our LOOP language [12] in that it properly types binary methods in the presence of inheritance. PolyTOIL in addition has a form of higher-order polymorphism. I-LOOP allows for inference of higher-order polymorphic types in the form of rc types. The expressivity of the two forms of polymorphism are incomparable, but both share some of the same basic functionality. There is a *matching* requirement for subclassing in PolyTOIL; interestingly, there is no such requirement for subclassing in I-LOOP, allowing more legal subclasses. This topic will be discussed in more detail in the body of the paper.

Palsberg, Schwartzbach, *et al.* have a number of significant results concerning type inference for object-oriented languages [21, 20, 22]. They have developed a novel type inference procedure that is rooted in the ideas of flow analysis. They show how their approach may be used to infer the type-correctness of object-oriented programs. Other advantages of their approach include asymptotically efficient inference algorithms, and named class types. Their system however has no polymorphism, and they take a code-expansion view of inheritance, requiring re-type-checking with each class extension. It also requires the whole program to be present, interfering with modularity. Plevyak and Chien [23] have extended this flow-based approach to incorporate a notion of polymorphism via the concept of *splitting* the flow graph. Their aims and approach are somewhat different than ours, but a comparison is nonetheless possible. Roughly speaking, both approaches are alike in that they infer very powerful types for object-oriented programs; but while their analysis requires the whole program to be present, our approach can be applied to parts separately, allowing modularity to be accommodated. They place more emphasis on implementation: their algorithm has been implemented as part of the Concurrent Aggregates language, and they have made impressive use of the type inference information to aid in the generation of optimal code. In some ways their inference algorithm also appears to be more powerful, as it allows for more than **let**-polymorphism, but at the price of accepting closed programs only. We give a more rigorous treatment—they present no type rules and establish no soundness or completeness properties. We also give a notation for types, and aim to make this information available to the programmer.

2 The I-Loop Language

I-Loop (Implicitly typed Little Object-Oriented Programming language) is intended to be a core object-oriented programming language. It includes notions of class, multiple inheritance, object creation via **new**, single dispatch of messages, and instance variables which may be read and written and which can be protected from outside access. To make the language simple we have left out friends and modules amongst other features. The grammar is as follows:[1]

$$
\begin{aligned}
e &::= x \mid n \mid \boldsymbol{\lambda} x.\, e \mid e\ e \mid \textbf{let}\ x = e\ \textbf{in}\ e \\
&\quad \mid l\ \#\ e \mid \textbf{case}\ e\ \textbf{of}\ \overline{l \Rightarrow e} \\
&\quad \mid e \leftarrow m \mid e.y \mid e.y := e \\
&\quad \mid \textbf{class}\ s\ \textbf{super}\ \overline{p_i\ \textbf{of}\, e_i}\ \textbf{inst}\ \overline{y_j = e_j}\ \textbf{meth}\ \overline{m_k = e_k} \\
&\quad \mid \textbf{new}\ e \mid \textbf{close}\ e \\
x &\in \mathit{Var} \\
n &::= 0 \mid 1 \mid 2 \mid \ldots
\end{aligned}
$$

In a **class** definition s is a variable bound to the self in the method bodies $\overline{e_j}$, and $\overline{p_i}$ refer to the *super*s in method bodies and the initial values $\overline{e_k}$ of instance variables. The following artificially simple example defines a class IntOrd with methods to read an object's value and to compare it against that of another object.

```
let
  IntOrd = class self super
    inst x = 0
    meth
      value = λ_. self.x,
      leq = λo. (self ← value ()) <= (o ← value ())
in ...
```

In this example and elsewhere we use $\boldsymbol{\lambda}\_.\, e$ to denote a method with no argument, an abbreviation for $\boldsymbol{\lambda} x.\, e$ for some x not free in e. To invoke such a method, we must apply it to some value; our convention is to apply it to the *empty object* with no methods or instance variables, written () and defined as **new** (**class** self **super inst meth**). The I-Loop injection syntax $l\ \#\ e$ tags the value of e with the label l; the **case** expression then uses the tag to determine the appropriate case to be applied to the value. The booleans and conditional are derived from variants: true and false are defined as true # () and false # (), respectively, and **if** e **then** e_1 **else** e_2 stands for **case** e **of** true⇒$\boldsymbol{\lambda}\_.\, e_1$, false⇒$\boldsymbol{\lambda}\_.\, e_2$. The **close** construct is used by a method that wishes to return the self or a super, preventing instance variables from escaping.[2]

Inheritance is explicitly expressed in I-Loop, and this disambiguates in the case of multiple inheritance. Continuing the above example, we define a class BoolOrd which inherits the value of its instance variable x and method not from the simple class BoolValue, inherits value from IntOrd, and redefines method leq.

```
...let
  BoolValue = class self super
    inst x = false
    meth not = λ_. if self.x then false else true
 in let
  BoolOrd = class self
    super bp of BoolValue, ip of IntOrd
    inst x = bp.x
    meth
      not = bp ← not,
      value = ip ← value,
      leq = λo. if self ← value () then o ← value ()
                                    else true
 in ...
```

2.1 I-Loop Types

The monomorphic types *Typ* of the language are

$$
\begin{aligned}
\tau &::= \alpha \mid \mathsf{Nat} \mid \tau \rightarrow \tau' \mid [\,\overline{l : \tau}\,] \mid \textbf{Inst}\ i\beta\ \textbf{Meth}\ m\beta \\
&\quad \mid \textbf{Class}\ st\ \textbf{Inst}\ i\beta\ \textbf{Meth}\ m\beta \\
m\beta &::= mr \mid \overline{m : \tau} \\
i\beta &::= ir \mid \overline{y : \tau}
\end{aligned}
$$

where a type variable $\alpha \in \mathit{TyVar}$ can be either *applicative*, $t \in \mathit{AppTyVar} \stackrel{\text{def}}{=} \{\mathsf{t}_1, \mathsf{t}_2, \ldots\}$ or *imperative*, $u \in \mathit{ImpTyVar} \stackrel{\text{def}}{=} \{\mathsf{u}_1, \mathsf{u}_2, \ldots\}$. This division of variables into two kinds is needed in the presence of state and **let**-polymorphism, as in Standard ML. The set of free type variables in a type τ is $FTV(\tau)$; τ is *imperative* if $FTV(\tau) \subseteq \mathit{ImpTyVar}$. Along with the conventional basic types of numerals and functions we include variant types, from which *e.g.* the type of booleans is derived:

[1] Some words on our notation. The "vector notation" $\overline{m = e}$ is shorthand for $m_1 = e_1, \ldots, m_k = e_k$ for some k; $\overline{m_i = e_i}$ is shorthand for the same and indicates i will range over the elements of the vector. We use some informal conventions with variable names. Variables x and z are generic. There are two distinct kinds of names, method names m, and instance variable names y. We assume there exist some standard built-in primitive functions with names in the set $B \subseteq \mathit{Var}$.

[2] An implemented language might insert **close** implicitly and keep it out of the official language syntax.

$\mathsf{Bool} \stackrel{\text{def}}{=} [\,\mathsf{true}:\mathbf{1}, \mathsf{false}:\mathbf{1}\,]$, where $\mathbf{1} \stackrel{\text{def}}{=} \textbf{Inst Meth}$ is the type of the empty object $()$.

The two novel types are the class types and the so-called **Inst**-**Meth** types. An **Inst**-**Meth** type should be thought of as a general type of objects. Inside a class definition, the types of self and super are of this form, with both the instance variables and methods visible. For instance, super object bp in the definition of class BoolOrd above is of type $\textbf{Inst}\ \mathsf{x}:\mathsf{Bool}\ \textbf{Meth}\ \mathsf{not}:\mathbf{1}\longrightarrow\mathsf{Bool}$, since it represents an object of class BoolValue with instance variable x visible and method not.

Objects outside their class definition have only their methods visible; they have an **Inst**-**Meth** type with no instance variables. We let $\textbf{Inst}\ i\beta$ and $\textbf{Meth}\ m\beta$ be shorthand types for an empty method or instance variable list, respectively: $\textbf{Inst}\ i\beta \stackrel{\text{def}}{=} \textbf{Inst}\ i\beta\ \textbf{Meth}$, $\textbf{Meth}\ m\beta \stackrel{\text{def}}{=} \textbf{Inst Meth}\ m\beta$. The latter are the types of objects, *e.g.* **new** BoolValue has the type $\textbf{Meth}\ \mathsf{not}:\mathbf{1}\longrightarrow\mathsf{Bool}$.

Class types have the added feature of a distinguished (applicative) type variable, by convention named st, denoting the type of self; the latter is not fixed by the class, for in the case that the class is extended, the self will in the future include more methods or instance variables. Class types are thus akin to type operators on st, giving the appropriate "open-ended" view of self [9, 12].

The type system of I-LOOP assigns expressions *recursively constrained (rc) types* of the form

$$\kappa ::= \tau \setminus C$$

where C is a set of *type constraints*—subtyping assertions about pairs of (monomorphic) types, written $\tau_1 \leq \tau_2$. This indicates that the variables in type τ are required to satisfy the constraints in C; since the rules will implicitly enforce consistency of C, it makes sense to view $\tau \setminus C$ as a type. As an example of an rc type, the type of the IntOrd class will be shown later to be

$$\begin{array}{l} \textbf{Class}\ \mathsf{st} \\ \textbf{Inst}\ \mathsf{x}:\mathsf{Nat} \\ \textbf{Meth}\ \mathsf{value}:\mathsf{t}_1\longrightarrow\mathsf{t}_2, \\ \qquad \mathsf{leq}:(\textbf{Meth}\ \mathsf{value}:\mathbf{1}\longrightarrow\mathsf{Nat})\longrightarrow\mathsf{Bool} \\ \qquad \setminus \{\mathsf{st} \leq \textbf{Inst}\ \mathsf{x}:\mathsf{t}_2,\ \mathsf{st} \leq \textbf{Meth}\ \mathsf{value}:\mathbf{1}\longrightarrow\mathsf{Nat}\} \end{array}$$

Note st has two upper bounds, and this is equivalent to a type expression

$$\mathsf{st} \leq (\textbf{Inst}\ \mathsf{x}:\mathsf{t}_2) \wedge (\textbf{Meth}\ \mathsf{value}:\mathbf{1}\longrightarrow\mathsf{Nat})$$

with $\wedge$ indicating a hypothetical greatest lower bound type operation. I-LOOP is capable of expressing some glb types, as above, as well as some lub types. Including lub and glb as general type operators presents significant difficulty [2], however, and these restricted forms fulfill most of the need for lub and glb type operations.

The *type schemes* σ are either monomorphic types or quantified rc types:

$$\sigma ::= \tau \mid \forall\overline{\alpha}.\ \kappa$$

These types are a straightforward generalization of Hindley-Milner **let**-polymorphic types to incorporate constraint sets. Since $\kappa = \tau \setminus C$ can contain an arbitrary collection of constraints C, an F-bounded quantified type [6] written $\forall t \leq \tau.\ \tau'$ may be expressed by a polymorphic rc type $\forall t.\ \tau' \setminus \{t \leq \tau\}$.

Next we need to define when a set of constraints C is consistent. If C is consistent, $\tau \setminus C$ is a sensible type. The definition of consistency we use is somewhat unusual: the constraints are considered consistent if they are not inconsistent in any "obvious" way. To motivate this definition, the following constraint system is obviously inconsistent

$$C_{\text{bad}} \stackrel{\text{def}}{=} \{\mathbf{1}\longrightarrow\mathsf{Nat} \leq \mathsf{t},\ \mathsf{t} \leq \mathbf{1}\longrightarrow\mathsf{Bool}\}$$

because by transitivity $\mathbf{1}\longrightarrow\mathsf{Nat} \leq \mathbf{1}\longrightarrow\mathsf{Bool}$, and thus by subtyping on functions, $\mathsf{Nat} \leq \mathsf{Bool}$, but this is immediately inconsistent. Because this constraint system is inconsistent, it makes no sense to write an rc type such as $\mathsf{t} \setminus C_{\text{bad}}$. The typing rules will implicitly rule out these ill-formed rc types. If no such inconsistency is found by applying these and other basic principles, the constraint set is declared consistent. We formally define consistency by first defining a closure operation on a constraint set that performs these simple deductions. The definition of a closed set of constraints C appears in Figure 1. Most of these conditions are based on standard subtyping principles [7]. In condition (v), if one instance variable is a subtype of another, the types must in fact be the same; this is because instance variables may be both read and written. In condition (vii), the self-types st are contravariantly related. This is because classes are parametric in the as-yet unknown self, so the contravariance is as for function parameter types. $Cl(C)$, the *closure* of C, is defined as the least closed superset of C. The consistent constraint sets finally may be defined as the ones in which the closure did not uncover any immediate inconsistencies.

DEFINITION 2.1 (CONSTRAINT CONSISTENCY)

(i) Constraints of the following forms are *immediately inconsistent*:

  (a) $\tau \leq \tau'$ where τ and τ' have different outermost type constructors;

  (b) $[\,\overline{l:\tau}\,] \leq [\,\overline{l':\tau'}\,]$, if $\{\overline{l}\} \not\subseteq \{\overline{l'}\}$; and

  (c) $\textbf{Inst}\ \overline{y{:}\tau_y}\ \textbf{Meth}\ \overline{m{:}\tau_m} \leq \textbf{Inst}\ \overline{y'{:}\tau'_y}\ \textbf{Meth}\ \overline{m'{:}\tau'_m}$, if $\{\overline{m}\} \not\supseteq \{\overline{m'}\}$ or $\{\overline{y}\} \not\supseteq \{\overline{y'}\}$.

(i) If $\{\tau_1 \leq \tau_2,\ \tau_2 \leq \tau_3\} \subseteq C$, then $\tau_1 \leq \tau_3 \in C$.
(ii) If $\overline{\tau_1 \rightarrow \tau'_1 \leq \tau_2 \rightarrow \tau'_2} \in C$, then $\{\tau_2 \leq \tau_1,\ \tau'_1 \leq \tau'_2\} \subseteq C$.
(iii) If $[\,\overline{l_i : \tau_i}\,] \leq [\,\overline{l_j : \tau'_j}\,] \in C$ and $\{\overline{l_i}\} \subseteq \{\overline{l_j}\}$, then $\{\overline{\tau_i \leq \tau'_i}\} \subseteq C$.
(iv) If $\textsf{Inst}\ i\beta\ \textsf{Meth}\ m\beta \leq \textsf{Inst}\ i\beta'\ \textsf{Meth}\ m\beta' \in C$, then $\{\textsf{Inst}\ i\beta \leq \textsf{Inst}\ i\beta',\ \textsf{Meth}\ m\beta \leq \textsf{Meth}\ m\beta'\} \subseteq C$.
(v) If $\textsf{Inst}\ \overline{y_i : \tau_i} \leq \textsf{Inst}\ \overline{y_j : \tau'_j} \in C$ and $\{\overline{y_i}\} \supseteq \{\overline{y_j}\}$, then $\{\overline{\tau_j \leq \tau'_j,\ \tau'_j \leq \tau_j}\} \subseteq C$.
(vi) If $\textsf{Meth}\ \overline{m_i : \tau_i} \leq \textsf{Meth}\ \overline{m_j : \tau'_j} \in C$ and $\{\overline{m_i}\} \supseteq \{\overline{m_j}\}$, then $\{\overline{\tau_j \leq \tau'_j}\} \subseteq C$.
(vii) If $\textsf{Class}\ st\ \textsf{Inst}\ i\beta_0\ \textsf{Meth}\ m\beta_1 \leq \textsf{Class}\ st'\ \textsf{Inst}\ i\beta'_0\ \textsf{Meth}\ m\beta'_1 \in C$,
then $\{\textsf{Inst}\ i\beta_0\ \textsf{Meth}\ m\beta_1 \leq \textsf{Inst}\ i\beta'_0\ \textsf{Meth}\ m\beta'_1,\ st' \leq st\} \subseteq C$.

Figure 1: Definition of a closed constraint set C.

(ii) A set of constraints C is consistent if no constraint in $Cl(C)$ is immediately inconsistent.

We define the following notion of subtyping on consistent rc types.

Definition 2.2 (Subtyping rc Types)
$\tau \setminus C \leq \tau' \setminus C'$ provided that C' is consistent and either $Cl(C \cup \{\tau \leq \tau'\}) \subseteq Cl(C')$, or $\tau = \tau'$ and $Cl(C) \subseteq Cl(C')$.

Stronger notions of subtyping could be defined, but for our purposes this definition suffices. The main shortcoming of this weaker form of subtyping is minor: it forces "garbage" constraints to be added to C. For instance, suppose we needed to show the subtyping

$$\textsf{Inst}\ x : \textsf{Nat}, x' : \textsf{Bool} \setminus \emptyset \ \leq\ \textsf{Inst}\ x : \textsf{Nat} \setminus \emptyset;$$

(for instance, if we had an expression in the former type and wished to use subsumption to show it was in the latter). We would instead use the valid subtyping

$$\begin{array}{l} \textsf{Inst}\ x : \textsf{Nat}, x' : \textsf{Bool} \setminus \emptyset \\ \leq\ \textsf{Inst}\ x : \textsf{Nat} \setminus \{\textsf{Inst}\ x : \textsf{Nat}, x' : \textsf{Bool} \leq \textsf{Inst}\ x : \textsf{Nat}\} \end{array}$$

Since the added constraint never introduces any inconsistencies, it is garbage. We will compensate for our weak notion of subtyping by having a simplification rule to remove garbage.

2.2 The I-Loop Typing Rules

The typing rules for I-Loop are given in Figure 2; notation used in sequent judgements includes the following. A *type environment* A is a mapping from variables to type schemes; we use the notation $(A, x : \sigma)$ for the environment extending A by assigning type scheme σ to the variable x. Given a type environment A, the proof system assigns to an expression e a rc type $\tau \setminus C$, written as the *type judgement* $A \vdash e : \tau \setminus C$, under the condition that C is consistent (as mentioned previously, *each rule implicitly enforces consistency on all constraint sets used in the rule*).[3] The rules differ in several ways when compared with standard typing rules. Each type assigned to a term is an rc type, with a set of constraints C. In rules with multiple assumptions, the conclusion type constraints are the union of the branches. Since all constraint systems in rule conclusions must be consistent, these unions are required to yield consistent constraint sets if the rule is to be applicable. Other presentations of constrained type systems [18, 2, 14] do not require local consistency, so the constraints in the rules have both a hypothetical and assertional component: they are hypothetical in that they may be inconsistent, and they are assertional in that they assert properties of the type if they are consistent. We believe the meaning of constraints is made clearer by implicitly enforcing their consistency, thus removing the hypothetical aspect of constraints and justifying their placement on the right of the turnstile as a positive assertion.

We illustrate the I-Loop typing rules by showing how the class IntOrd defined previously is provably typed as follows:

$A_0 \vdash_{\mathrm{L}}$ **class** self **super**
  inst x = 0
  meth
    value = λ_. self.x,
    leq = λo. (self ← value ()) <= (o ← value ())

[3] We write also $A \vdash_{\mathrm{L}} \overline{e_i : \tau_i \setminus C_i}$ to indicate several type judgements provable in the same environment. Programs are type-checked in the initial environment A_0 assigning type schemes to the built-ins in B; we let $\vdash_{\mathrm{L}} e : \kappa$ abbreviate $A_0 \vdash_{\mathrm{L}} e : \kappa$, since all typings should occur in the presence of assumptions A_0. A *substitution* on $\{\overline{\alpha}\}$ is a map $\Psi \in TyVar \rightarrow Typ$ which is the identity on $TyVar \setminus \{\overline{\alpha}\}$ and maps $ImpTyVar$ to imperative types; a *renaming* Φ of $\{\overline{\alpha}\}$ is a substitution on $\{\overline{\alpha}\}$ with $Codom(\Phi) \subseteq TyVar$. Following Tofte [27] we form type schemes by making the sets of type variables we generalize over dependent on the expansiveness of the expression: an expression is *expansive* if and only if it is not a value; in I-Loop values are the identifiers, numerals, λ-abstractions, classes, and injections of values. The definitions of these sets are

$$\begin{aligned} Clos(\tau \setminus C, A) &= (FTV(\tau) \cup FTV(C)) - FTV(A) \\ AppClos(\tau \setminus C, A) &= Clos(\tau \setminus C, A) \cap AppTyVar \end{aligned}$$

where the functionality of FTV is extended as usually to constraint systems, rc types, type schemes, and type environments.

$$\text{(Sub)}\ \frac{A \vdash_{\rm L} e : \kappa, \qquad \kappa \le \kappa'}{A \vdash_{\rm L} e : \kappa'}$$

$$\text{(Simp)}\ \frac{A \vdash_{\rm L} e : \kappa}{A \vdash_{\rm L} e : \mathit{Simplified}(\kappa,\, A)} \qquad \text{(Num)}\ \frac{}{A \vdash_{\rm L} n : \mathsf{Nat} \setminus \emptyset}$$

$$\text{(Var)}\ \frac{A(x) = \tau}{A \vdash_{\rm L} x : \tau \setminus \emptyset} \qquad \text{(PVar)}\ \frac{A(x) = \forall\overline{\alpha}.\ \kappa, \qquad \Psi \text{ is a substitution on } \{\overline{\alpha}\}}{A \vdash_{\rm L} x : \Psi\kappa}$$

$$\text{(Abs)}\ \frac{A,\, x : \tau \vdash_{\rm L} e : \tau' \setminus C}{A \vdash_{\rm L} \lambda x.\, e : \tau \to \tau' \setminus C} \qquad \text{(App)}\ \frac{A \vdash_{\rm L} e_1 : \tau_1 \to \tau_2 \setminus C_1, \qquad A \vdash_{\rm L} e_2 : \tau_1 \setminus C_2}{A \vdash_{\rm L} e_1\ e_2 : \tau_2 \setminus C_1 \cup C_2}$$

$$\text{(Inj)}\ \frac{A \vdash_{\rm L} e : \tau \setminus C}{A \vdash_{\rm L} l\,\#\,e : [\,l : \tau\,] \setminus C} \qquad \text{(Case)}\ \frac{A \vdash_{\rm L} e : [\,\overline{l_i : \tau_i}\,] \setminus C, \qquad A \vdash_{\rm L} \overline{e_i : \tau_i \to \tau \setminus C_i}}{A \vdash_{\rm L} \mathbf{case}\ e\ \mathbf{of}\ \overline{l_i \Rightarrow e_i} : \tau \setminus C \cup \bigcup_i C_i}$$

$$\text{(Read)}\ \frac{A \vdash_{\rm L} e : \mathbf{Inst}\ y : \tau \setminus C}{A \vdash_{\rm L} e.y : \tau \setminus C} \qquad \text{(Write)}\ \frac{A \vdash_{\rm L} e_1 : \mathbf{Inst}\ y : \tau \setminus C_1, \qquad A \vdash_{\rm L} e_2 : \tau \setminus C_2}{A \vdash_{\rm L} e_1.y \mathrel{\mathbf{:=}} e_2 : \tau \setminus C_1 \cup C_2}$$

$$\text{(Msg)}\ \frac{A \vdash_{\rm L} e : \mathbf{Meth}\ m : \tau \setminus C}{A \vdash_{\rm L} e \leftarrow m : \tau \setminus C} \qquad \text{(Close)}\ \frac{A \vdash_{\rm L} e : \mathbf{Inst}\ i\beta\ \mathbf{Meth}\ m\beta \setminus C}{A \vdash_{\rm L} \mathbf{close}\ e : \mathbf{Meth}\ m\beta \setminus C}$$

$$\text{(Let)}\ \frac{A \vdash_{\rm L} e : \tau \setminus C, \qquad A,\, x : \forall\overline{\alpha}.\ \tau \setminus C \vdash_{\rm L} e' : \tau' \setminus C', \qquad \Phi \text{ is a renaming of } \{\overline{\alpha}\}}{A \vdash_{\rm L} \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau' \setminus \Phi C \cup C'}$$

$$\text{where } \{\overline{\alpha}\} = \begin{cases} \text{if } e \text{ is expansive then } \mathit{AppClos}(\tau \setminus C,\, A) \\ \text{else } \mathit{Clos}(\tau \setminus C,\, A) \end{cases}$$

$$\text{(New)}\ \frac{A \vdash_{\rm L} e : \mathbf{Class}\ st\ \mathbf{Inst}\ i\beta\ \mathbf{Meth}\ m\beta \setminus C}{A \vdash_{\rm L} \mathbf{new}\ e : \mathbf{Meth}\ m\beta \setminus C \cup \{\mathbf{Inst}\ i\beta\ \mathbf{Meth}\ m\beta \le st\}}$$

$$\text{(Class)}\ \frac{\begin{array}{l} A \vdash_{\rm L} \overline{e_i : \mathbf{Class}\ st\ \mathbf{Inst}\ i\beta_i\ \mathbf{Meth}\ m\beta_i \setminus C_i} \\ A,\, \overline{p_i : \mathbf{Inst}\ i\beta_i\ \mathbf{Meth}\ m\beta_i} \vdash_{\rm L} \overline{e_j : \tau_j \setminus C'_j} \\ A,\, \overline{p_i : \mathbf{Inst}\ i\beta_i\ \mathbf{Meth}\ m\beta_i},\, s : st \vdash_{\rm L} \overline{e_k : \tau_k \setminus C''_k} \end{array}}{A \vdash_{\rm L} \mathbf{class}\ s\ \mathbf{super}\ \overline{p_i\ \mathbf{of}\ e_i}\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = e_k} : \mathbf{Class}\ st\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \setminus C}$$

provided $\overline{\tau_j}$ are imperative and $C \cup \{\mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \le st\}$ is consistent, where $C = \overline{C_i} \cup \overline{C'_j} \cup \overline{C''_k}$

Figure 2: Typing rules of I-Loop.

$$\begin{array}{l} : \mathbf{Class}\ \mathsf{st} \\ \quad \mathbf{Inst}\ \mathsf{x} : \mathsf{Nat} \\ \quad \mathbf{Meth}\ \mathsf{value} : \mathsf{t_1} \to \mathsf{t_2}, \\ \qquad\quad \mathsf{leq} : (\mathbf{Meth}\ \mathsf{value} : \mathbf{1} \to \mathsf{Nat}) \to \mathsf{Bool} \\ \setminus \{\mathsf{st} \le \mathbf{Inst}\ \mathsf{x} : \mathsf{t_2},\ \mathsf{st} \le \mathbf{Meth}\ \mathsf{value} : \mathbf{1} \to \mathsf{Nat}\} \end{array}$$

We assume that A_0 assigns the type $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool}$ to the infix function <=; for brevity we drop A_0 from the judgments in this example. Type-checking the initialization of instance variable x is trivial: by (Nat) we have $\vdash_{\rm L} \mathsf{0} : \mathsf{Nat} \setminus \emptyset$. Recall the class expression binds self in the method definitions; they are now type-checked under the assumption (self : st). Starting with the value method body, rule (Var) yields the obvious type for self:

$$\mathsf{self} : \mathsf{st} \vdash_{\rm L} \mathsf{self} : \mathsf{st} \setminus \emptyset$$

To apply (Read) we need the instance variable x to be present in the type of the object; according to definition 2.2 we can construct a supertype of $\mathsf{st} \setminus \emptyset$ by adding a constraint (provided the resulting system is consistent, which it is):

$$\mathsf{st} \setminus \emptyset \ \le\ \mathbf{Inst}\ \mathsf{x} : \mathsf{t_2} \setminus \{\mathsf{st} \le \mathbf{Inst}\ \mathsf{x} : \mathsf{t_2}\}$$

so we can lift the rc type of self to a type that has the field x using (Sub):

$$\mathsf{self} : \mathsf{st} \vdash_{\rm L} \mathsf{self} : \mathbf{Inst}\ \mathsf{x} : \mathsf{t_2} \setminus \{\mathsf{st} \le \mathbf{Inst}\ \mathsf{x} : \mathsf{t_2}\}$$

We chose a new type variable $\mathsf{t_2}$ for the type of the instance variable x because simply selecting x from self says nothing about the type of x. By (Read),

$$\mathsf{self} : \mathsf{st} \vdash_{\rm L} \mathsf{self.x} : \mathsf{t_2} \setminus \{\mathsf{st} \le \mathbf{Inst}\ \mathsf{x} : \mathsf{t_2}\}$$

Since nothing further is known about $\mathsf{t_2}$ at this point, the type variable remains unconstrained. The method parameter type $\mathsf{t_1}$ is also unconstrained since the parameter is not used. The full method body may then be typed by (Abs):

$$\mathsf{self} : \mathsf{st} \vdash_{\rm L} \boldsymbol{\lambda}\_.\, \mathsf{self.x} : \mathsf{t_1} \to \mathsf{t_2} \setminus \{\mathsf{st} \le \mathbf{Inst}\ \mathsf{x} : \mathsf{t_2}\}$$

It would seem that the natural type to give value would be $\mathbf{1}\rightarrow\mathsf{Nat}$, but the method body itself poses no requirement at all on the argument, and only requires t_2 to be the type of x, whatever that type may be.[4] Turning to the type-checking of the leq method, the message send $\mathsf{self} \leftarrow \mathsf{value}$ is typed similarly to self.x; the difference is (Msg) is used in place of (Read):

$$\mathsf{self} : \mathsf{st}, \mathsf{o} : \mathsf{t}_3 \vdash_{\mathrm{L}} \mathsf{self} \leftarrow \mathsf{value} : \mathsf{t}_4 \setminus \{\mathsf{st} \leq \textbf{Meth}\ \mathsf{value}{:}\mathsf{t}_4\}$$

The above proof was "lazy" in the sense that from inspection of the program we can see *a priori* that the type of $\mathsf{self} \leftarrow \mathsf{value}$ must be $\mathbf{1}\rightarrow\mathsf{Nat}$: it is applied to () and the result is then compared as a Nat by <=. Thus, we could have used that type in place of t_4. The inference algorithm, presented in the next section, is "completely lazy," it never looks at the context and instead always invents a new variable as above. A key result of the paper—completeness of type inference—implies that there is no harm in being completely lazy. We will at least be somewhat eager at this point by adding consistent constraint $\mathsf{t}_4 \leq \mathbf{1}\rightarrow\mathsf{Nat}$; with this constraint and (Sub), by (App) we obtain

$$\begin{array}{l}\mathsf{self} : \mathsf{st}, \mathsf{o} : \mathsf{t}_3 \vdash_{\mathrm{L}} \mathsf{self} \leftarrow \mathsf{value}\ () : \mathsf{Nat} \\ \qquad\qquad \setminus \{\mathsf{st} \leq \textbf{Meth}\ \mathsf{value} : \mathsf{t}_4,\ \mathsf{t}_4 \leq \mathbf{1}\rightarrow\mathsf{Nat}\}\end{array}$$

The proof of the type of the other argument of <= is almost identical, with the conclusion

$$\begin{array}{l}\mathsf{self} : \mathsf{st}, \mathsf{o} : \mathsf{t}_3 \vdash_{\mathrm{L}} \mathsf{o} \leftarrow \mathsf{value}\ () : \mathsf{Nat} \\ \qquad\qquad \setminus \{\mathsf{t}_3 \leq \textbf{Meth}\ \mathsf{value} : \mathsf{t}_5,\ \mathsf{t}_5 \leq \mathbf{1}\rightarrow\mathsf{Nat}\}\end{array}$$

The type of the leq method may finally be obtained by applying (App) twice, followed by (Abs).

$$\begin{array}{l}\mathsf{self} : \mathsf{st} \vdash_{\mathrm{L}} \boldsymbol{\lambda}\mathsf{o}.\ (\mathsf{self} \leftarrow \mathsf{value}\ ()) \mathrel{<=} (\mathsf{o} \leftarrow \mathsf{value}\ ()) \\ \qquad : \mathsf{t}_3 \rightarrow \mathsf{Bool} \\ \qquad\ \setminus \{\mathsf{st} \leq \textbf{Meth}\ \mathsf{value} : \mathsf{t}_4,\ \mathsf{t}_4 \leq \mathbf{1}\rightarrow\mathsf{Nat}, \\ \qquad\qquad \mathsf{t}_3 \leq \textbf{Meth}\ \mathsf{value} : \mathsf{t}_5,\ \mathsf{t}_5 \leq \mathbf{1}\rightarrow\mathsf{Nat}\}\end{array}$$

We are now able to apply the (Class) rule and give a type to the whole class expression. The first premise of (Class) guarantees the superclasses are of appropriate **Class** types, vacuously satisfied here. The second premise guarantees the instance variable initializations are type-correct under the assumption that the superclass objects p_i have the **Inst**-**Meth** types corresponding to the types of their classes; for our example this was trivial since $\vdash_{\mathrm{L}} 0 : \mathsf{Nat}$ directly.[5] The third premise is similar but concerns method bodies, where self has type *st*. We just finished proving these requirements for the two methods, value and leq. From these premises we may apply (Class) to obtain the following type for class IntOrd:

$$\begin{array}{l}\vdash_{\mathrm{L}} \textbf{class}\ \mathsf{self}\ \ldots \\ : \textbf{Class}\ \mathsf{st} \\ \quad \textbf{Inst}\ \mathsf{x} : \mathsf{Nat} \\ \quad \textbf{Meth}\ \mathsf{value} : \mathsf{t}_1 \rightarrow \mathsf{t}_2, \\ \qquad\quad \mathsf{leq} : \mathsf{t}_3 \rightarrow \mathsf{Bool} \\ \quad \setminus \{\mathsf{st} \leq \textbf{Inst}\ \mathsf{x} : \mathsf{t}_2, \\ \qquad \mathsf{st} \leq \textbf{Meth}\ \mathsf{value} : \mathsf{t}_4,\ \mathsf{t}_4 \leq \mathbf{1}\rightarrow\mathsf{Nat}, \\ \qquad \mathsf{t}_3 \leq \textbf{Meth}\ \mathsf{value} : \mathsf{t}_5,\ \mathsf{t}_5 \leq \mathbf{1}\rightarrow\mathsf{Nat}\}\end{array}$$

In rule (Class) there is a final constraint consistency proviso; this condition enforces "new-ability," *i.e.* it ensures the **new** operation will succeed for this class. Without this proviso it would be possible to define a class that used the self in incompatible ways, such as self.y when there was no instance variable y defined. This restriction is in fact not necessary in the (Class) rule, for such errors would be caught by (New). This condition is added for methodological reasons—the programmer is unlikely to define a class for which **new** always fails. However, there *are* reasons to consider allowing such incomplete classes—for instance, it may be desirable to define abstract classes which use methods that have not yet been defined (in Smalltalk, this is accomplished by defining the method but with the dummy body self subclassResponsibility which flags an error if the method is not overridden). Special class definition syntax, *e.g.* **absclass**, could be introduced for these inherently abstract classes, and the new-ability constraint dropped for them. This would increase the usefulness of programming in a mixin-style methodology in I-Loop—mixins are often fragments that are ill-formed as classes *per se*.

The (Class) rule still allows more flexibility than most class typing principles. There is no requirement that the subclass type *match* its superclass type(s) [5]. In terms of I-Loop, a subclass matches its superclass if a subtyping constraint between their class types is consistent when they share the same self type *st*; for instance, class BoolOrd matches class BoolValue since all methods and instance variables of BoolValue are inherited by BoolOrd, and their types remain unchanged (as functions of the self type) while additional methods are defined. Since we do not enforce this matching requirement, a subclass can completely change the type of, or even not define, a method or instance variable of a superclass. Both of these examples in fact occur in Smalltalk class libraries (the latter via method body self shouldNotImplement), so it does not make sense to further restrict the class rule to disallow this; however, if so desired, adding the constraints

$$\textbf{Inst}\ \overline{y_j : \tau_j}\ \textbf{Meth}\ \overline{m_k : \tau_k} \leq \textbf{Inst}\ i\beta_i\ \textbf{Meth}\ m\beta_i$$

[4] Indeed in our example committing to result type Nat would prevent the subclass BoolOrd from type-checking: its method value should return a boolean, the type of its x.

[5] Since the instance variables τ_j are mutable, their types must be imperative as per Tofte's type discipline [27]; Nat trivially satisfies this—it contains no type variables.

to the conclusion of the (Class) rule would enforce matching. Returning to our earlier example, class BoolOrd also inherits from IntOrd, but does not match it—the type of the instance variable x is changed from Nat to Bool. However the newability proviso of the (Class) rule *will* prevent an action that could be unsafe, like changing the result type of instance variable x from Nat to Bool while having another method leq directly or indirectly use x and expect a Nat result. In our example leq had to be overridden to prevent this clash from arising.

The type obtained for IntOrd can now be transformed to the more concise form given at the beginning of the section using the simplification rule (Simp); specifically this rule allows the bound of a type variable to be substituted for the variable itself when certain conditions are met. The formal definition of *Simplified*(κ, A) is technical and is found in Appendix A, but informally we can say it is safe to replace a type variable which only appears in the type in "contravariant" (or "negative") positions by its upper bound, provided the bound is unique and does not mention that variable. The exact definition of contravariant position in a rc type is rather involved, but it generalizes the conventional notion based on the observation that the function type $\tau \longrightarrow \tau'$ is contravariant in the argument type τ and covariant in the result type τ'. Thus for instance the type variable t_3 is only in contravariant position in the type

$$\textbf{Class}\ \mathsf{st}\ \textbf{Inst}\ \mathsf{x} : \mathsf{Nat}\ \textbf{Meth}\ \mathsf{value} : \mathsf{t}_1 \longrightarrow \mathsf{t}_2, \mathsf{leq} : \mathsf{t}_3 \longrightarrow \mathsf{Bool}$$

and therefore it can be replaced by its upper bound $\textbf{Meth}\ \mathsf{value} : \mathsf{t}_5$. In the resulting type

$$\begin{array}{l}\textbf{Class}\ \mathsf{st}\\ \textbf{Inst}\ \mathsf{x} : \mathsf{Nat}\\ \textbf{Meth}\ \mathsf{value} : \mathsf{t}_1 \longrightarrow \mathsf{t}_2, \mathsf{leq} : (\textbf{Meth}\ \mathsf{value} : \mathsf{t}_5) \longrightarrow \mathsf{Bool}\end{array}$$

the variable t_5 is in negative position, so we can replace it in turn by its upper bound $\mathbf{1} \longrightarrow \mathsf{Nat}$. A similar transformation involving the type variable t_4 yields the initially claimed type for IntOrd.

The type we obtained for the class IntOrd sets only upper bounds on the self type st. As we saw, each upper bound on a type variable t is added to the constraint system as a result of some *requirement* that an expression of type represented by t must meet. Dually, a lower bound on t corresponds to the type of an *implementation*, an actual value that may have to be used as a member of type t. Thus the transitivity clause (i) in Figure 1 can be viewed as enforcing the rule that implementations must meet their stated requirements. A class expression introduces requirements on self, since methods make use of features of self; but it does not constrain the implementation of self as it is open to further extensions and modifications in the possible subclasses. Thus only upper bounds are placed on *st* when typing a class, confirmed by the example. The implementation is defined when we create a new object of the class: this object must be a valid implementation of self. The rule (New) expresses this by adding a lower bound on self type *st*, ensuring the instance variables and methods defined by the class indeed are an implementation of self. If **new** is applied to the IntOrd class, it adds such a constraint on st and yields the following type (some constraints resulting from taking the closure of the constraint set are shown as well):

$$\begin{array}{l}\vdash_{\mathrm{L}} \textbf{new}\ (\textbf{class} \ldots)\\ : \textbf{Inst}\ \mathsf{x} : \mathsf{Nat}\\ \quad \textbf{Meth}\ \mathsf{value} : \mathsf{t}_1 \longrightarrow \mathsf{t}_2,\\ \qquad\quad \mathsf{leq} : (\textbf{Meth}\ \mathsf{value} : \mathbf{1} \longrightarrow \mathsf{Nat}) \longrightarrow \mathsf{Bool}\\ \ \setminus \{\mathsf{st} \leq \textbf{Inst}\ \mathsf{x} : \mathsf{t}_2,\ \mathsf{st} \leq \textbf{Meth}\ \mathsf{value} : \mathbf{1} \longrightarrow \mathsf{Nat},\\ \quad (\textbf{Inst}\ \mathsf{x} : \mathsf{Nat}\\ \quad\ \textbf{Meth}\ \mathsf{value} : \mathsf{t}_1 \longrightarrow \mathsf{t}_2,\\ \qquad\quad \mathsf{leq} : (\textbf{Meth}\ \mathsf{value} : \mathbf{1} \longrightarrow \mathsf{Nat}) \longrightarrow \mathsf{Bool}) \leq \mathsf{st},\\ \quad \mathsf{Nat} \leq \mathsf{t}_2,\ \mathsf{t}_2 \leq \mathsf{Nat},\ \mathbf{1} \leq \mathsf{t}_1\}\end{array}$$

Only now does the result type t_2 of method value obtain a lower bound of Nat; this bound will be Bool for an object of class BoolOrd which redefines the instance variable x as a boolean. But a result type Nat is also what was required here by the use of value in leq, appearing in the closure of the constraint set as an upper bound on t_2; we encourage the reader to verify that the closure is indeed consistent.

Rule (Simp) also allows for the removal of all unreachable constraints from the constraint system of an rc type. Intuitively, a constraint is unreachable if it cannot cause an inconsistency to appear in any legal use of the rc type. For example, consider a term e for which $\vdash_{\mathrm{L}} e : \mathsf{t} \setminus \{\mathsf{Nat} \leq \mathsf{t}, \mathsf{t} \leq \mathsf{Nat}\}$. The constraint system in this example is consistent—its closure also contains the (consistent) constraint $\mathsf{Nat} \leq \mathsf{Nat}$. However, once the consistency of the system is established, the constraint $\mathsf{t} \leq \mathsf{Nat}$ can be ignored. This is because inspection of the typing rules shows that the type variable t can only get new upper bounds in all possible contexts in which e can appear; these new upper bounds can only produce inconsistent constraints when compared (by transitivity) to the old lower bounds of t, and hence the old upper bounds are irrelevant. Similarly, in the IntOrd object type given above it turns out that all of the constraints on st and t_1, as well as the upper bound on t_2, are unreachable; this allows us to apply (Simp) to obtain

$$\begin{array}{l}\textbf{Inst}\ \mathsf{x} : \mathsf{Nat}\\ \textbf{Meth}\ \mathsf{value} : \mathsf{t}_1 \longrightarrow \mathsf{Nat}, \mathsf{leq} : (\textbf{Meth}\ \mathsf{value} : \mathbf{1} \longrightarrow \mathsf{Nat}) \longrightarrow \mathsf{Bool}\\ \ \setminus \emptyset\end{array}$$

The rule (Close) is used to hide the instance variables in the self. If a method returns the self s, it should explicitly return **close** s to hide the instance variables. Note this syntax leaves open the possibility of *not* hiding the instance variables when the self is returned. This may in fact be advantageous for the programmer upon occasion, but in our case an explicit hiding syntax was used because to implicitly hide would add an extra degree of complication to the language definition that we wished to avoid.

Some comments are in order about **let**-polymorphism. (Let) gives **let**-bound variables polymorphic types in the assumption list, and (PVar) in turn allows instantiation of these polymorphic types. This is a straightforward generalization of ML-style **let**-polymorphism[6] to constrained type systems [10, 14, 2]. The technical restriction of quantification over imperative types to the nonexpansive expressions is the standard solution [27] recast in this setting, first shown sound in [11]. The **let** construct is very important for defining classes. In the example, classes are **let**-defined, meaning each use of one of the classes (extension or object creation) will get a fresh set of type variables via (PVar), and this will allow the class to behave in different ways each time it is used.

2.3 Type Inference

We now define the type inference algorithm and prove it is complete, *i.e.* if a program has a type derivation the inference algorithm will infer a type for it. Without this property, the programmer runs the risk of being stuck by writing a typable program that the inference algorithm fails to find a type for. The classic method for establishing completeness is to prove a *principal type* theorem, stating that any derived type is in fact a supertype of the inferred type. We take a more primitive path, since our rules are too weak to have a principality property. We first define a set of "inference" typing rules which are special forms of the general rules constructed so as to be (almost) deterministic, in that all type derivations using these rules having the same conclusion are isomorphic. These rules then serve to define the inference algorithm. We then prove these rules are *complete*: all proofs of typing using the general rules can be transformed into proofs in the restricted inference rules, meaning the inference algorithm will infer a result. A similar strategy was used to prove completeness of a type inference algorithm for I-Soop [11]. The I-Loop inference rules appear in Figure 3. It is easy to verify by inspection that each inference rule is a restricted form of the combination of (Sub) with a single general rule. So, the inference rules are a restricted version of the general rules. Inspection also should convince the reader that given proofs of all subterms of a term, with the exception of (Simp) there is only one rule that may next be applied, and this rule may only be applied in one manner (modulo renaming of variables). So, the rules nearly define a deterministic algorithm. We get around the problem with (Simp) by showing the rule is not necessary.

Lemma 2.3 (Simplification Removal) If $A \vdash_{\mathrm{L}}^{\mathrm{inf}} e : \kappa$ then this proof may be transformed to a proof $A \vdash_{\mathrm{L}}^{\mathrm{inf}} e : \kappa'$ that does not use the (Simp) rule.

Proof Sketch: See Appendix A. □

Theorem 2.4 For all terms e and environments A, it is decidable whether there exists a κ such that $A \vdash_{\mathrm{L}}^{\mathrm{inf}} e : \kappa$.

Proof Sketch: First, by Lemma 2.3 it suffices to show it is decidable to find a (Simp)-free proof. By inspection of the rules, there is only one non-(Simp) rule for typing each expression construct. By further inspection, the only nondeterminism that may be introduced in rule application is the choice of type variables used in rules (Class), (Abs) and (PVar). We thus choose *canonical* proofs that use fresh variables in every place possible. If a proof exists, there clearly must then be a corresponding canonical proof. For expression e the canonical proof is unique modulo α-conversion. Thus a decision procedure may be defined for constructing such a canonical proof. The algorithm fails when an inconsistent constraint system is obtained when combining the constraint systems inferred for subterms, and detection of such inconsistencies is trivially decidable. □

We now relate the inference rules to the general rules.

Theorem 2.5 (Completeness of Type Inference) Given an environment A and an expression e, the typing judgement $A \vdash_{\mathrm{L}} e : \kappa$ is provable for some κ if and only if $A \vdash_{\mathrm{L}}^{\mathrm{inf}} e : \kappa'$ is provable for some κ'.

Proof Sketch: If $A \vdash_{\mathrm{L}}^{\mathrm{inf}} e : \kappa'$ is provable, $A \vdash_{\mathrm{L}} e : \kappa$ is obviously provable as well; each inference rule is a special case of a combination of (Sub) and a general rule.

Conversely, a proof of $A \vdash_{\mathrm{L}} e : \kappa$ can be transformed into a proof of $A \vdash_{\mathrm{L}}^{\mathrm{inf}} e : \kappa'$ in several stages. First, each assumption $x : \tau$ added inside the proof by the rules (Abs) or (Class) is replaced with the assumption $x : t$ for some fresh t, and the constraint $\{t = \tau\}$ (shorthand for $\{t \leq \tau,\ \tau \leq t\}$) is added to the constraint system of each judgement. A use of the (Sub) rule is then used after each (Var) rule for x to lift $x : t$ to $x : \tau$; an application of (Sub) also follows (Abs) to reduce the domain t of the

[6] Alternative systems for typing polymorphic references could be considered. For instance inference of effect constraints [26] appears to be orthogonal to the inference of subtyping constraints in I-Loop.

$$\text{(Simp)}\ \frac{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e : \kappa}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e : \mathit{Simplified}(\kappa,\, A)} \qquad \text{(Num)}\ \frac{}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} n : \mathsf{Nat} \setminus \emptyset}$$

$$\text{(Var)}\ \frac{A(x) = \tau}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} x : \tau \setminus \emptyset} \qquad \text{(PVar)}\ \frac{A(x) = \forall \overline{\alpha}.\ \kappa, \quad \Phi \text{ is a renaming of } \{\overline{\alpha}\}}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} x : \Phi\kappa}$$

$$\text{(Abs)}\ \frac{A,\, x : t \vdash^{\mathrm{inf}}_{\mathrm{L}} e : \tau \setminus C}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} \lambda x.\, e : t \to \tau \setminus C} \qquad \text{(App)}\ \frac{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e_1 : \tau_1 \setminus C_1, \quad e_2 : \tau_2 \setminus C_2}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e_1\ e_2 : t \setminus C_1 \cup C_2 \cup \{\tau_1 \leq \tau_2 \to t\}}$$

$$\text{(Inj)}\ \frac{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e : \tau \setminus C}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} l\, \#\, e : [\, l : \tau\,] \setminus C} \qquad \text{(Case)}\ \frac{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e : \tau \setminus C, \quad \overline{e_i : \tau_i \setminus C_i}}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} \textbf{case}\ e\ \textbf{of}\ \overline{l_i \Rightarrow e_i} : t \setminus C \cup \{\tau \leq [\,\overline{l_i : t_i}\,]\} \cup \bigcup_i (C_i \cup \{\tau_i \leq t_i \to t\})}$$

$$\text{(Read)}\ \frac{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e : \tau \setminus C}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e.y : t \setminus C \cup \{\tau \leq \textbf{Inst}\ y : t\}} \qquad \text{(Write)}\ \frac{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e_1 : \tau_1 \setminus C_1, \quad e_2 : \tau_2 \setminus C_2}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e_1.y := e_2 : t \setminus C_1 \cup C_2 \cup \{\tau_1 \leq \textbf{Inst}\ y : t,\ \tau_2 \leq t\}}$$

$$\text{(Msg)}\ \frac{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e : \tau \setminus C}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e \leftarrow m : t \setminus C \cup \{\tau \leq \textbf{Meth}\ m : t\}} \qquad \text{(Close)}\ \frac{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e : \tau \setminus C}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} \textbf{close}\ e : \textbf{Meth}\ mr \setminus C \cup \{\tau \leq \textbf{Meth}\ mr\}}$$

$$\text{(Let)}\ \frac{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e : \tau \setminus C, \quad A,\, x : \forall\overline{\alpha}.\ \tau \setminus C \vdash^{\mathrm{inf}}_{\mathrm{L}} e' : \tau' \setminus C'}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} \textbf{let}\ x = e\ \textbf{in}\ e' : \tau' \setminus C \cup C'} \text{ where } \{\overline{\alpha}\} = \begin{cases} \text{if } e \text{ is expansive then } \mathit{AppClos}(\tau \setminus C,\, A) \\ \text{else } \mathit{Clos}(\tau \setminus C,\, A) \end{cases}$$

$$\text{(New)}\ \frac{A \vdash^{\mathrm{inf}}_{\mathrm{L}} e : \tau \setminus C}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} \textbf{new}\ e : \textbf{Meth}\ mr \setminus C \cup \{\tau \leq \textbf{Class}\ st\ \textbf{Inst}\ ir\ \textbf{Meth}\ mr, \textbf{Inst}\ ir\ \textbf{Meth}\ mr \leq st\}}$$

$$\text{(Class)}\ \frac{A \vdash^{\mathrm{inf}}_{\mathrm{L}} \overline{e_i : \tau_i \setminus C_i}, \quad A,\, \overline{p_i : pt_i} \vdash^{\mathrm{inf}}_{\mathrm{L}} \overline{e_j : \tau_j \setminus C'_j}, \quad A,\, \overline{p_i : pt_i},\, s : st \vdash^{\mathrm{inf}}_{\mathrm{L}} \overline{e_k : \tau_k \setminus C''_k}}{A \vdash^{\mathrm{inf}}_{\mathrm{L}} \textbf{class}\ s\ \textbf{super}\ \overline{p_i\ \textbf{of}\ e_i}\ \textbf{inst}\ \overline{y_j = e_j}\ \textbf{meth}\ \overline{m_k = e_k} : \textbf{Class}\ st\ \textbf{Inst}\ \overline{y_j : u_j}\ \textbf{Meth}\ \overline{m_k : \tau_k} \setminus C}$$

provided $C \cup \{\textbf{Inst}\ \overline{y_j : u_j}\ \textbf{Meth}\ \overline{m_k : \tau_k} \leq st\}$ is consistent, where

$$C = \overline{C_i} \cup \overline{C'_j} \cup \overline{C''_k} \cup \{\overline{\tau_j \leq u_j}\} \cup \{\overline{\tau_i \leq \textbf{Class}\ st\ \textbf{Inst}\ ir_i\ \textbf{Meth}\ mr_i}\} \cup \{\overline{\textbf{Inst}\ ir_i\ \textbf{Meth}\ mr_i \leq pt_i}\}$$

Figure 3: Type inference rules of I-Loop.

λ-abstraction back to τ, and similarly with the (Class) rule. A similar transformation is then used to convert each substitution Ψ in the (PVar) rule into a renaming, replacing each type τ in the codomain of Ψ with fresh type variable t and adding the constraint $\{t = \tau\}$ as above. Finally, the proof is inductively transformed from the leaves to the root, replacing each general rule with its inference form and bubbling to the root uses of (Sub) in the proof. The final form is an inference proof of $A \vdash_{\mathrm{L}} e : \kappa$ with a final (Sub) step, and the proof with this final step removed is an inference proof. □

Thus from Theorems 2.5 and 2.4 we may conclude that every program typable under the general rules has a type inferred by the type inference algorithm.

3 Semantics of I-Loop

We have established completeness of the type inference algorithm, but more basically we need to show the I-Loop rules are sound, namely that properly typed programs never have run-time errors. We give semantics to I-Loop by translation into a more basic language, I-Soop, which lacks object-oriented features but includes records and reference cells. We first define the I-Soop language and type rules. An operational semantics and type rules for I-Soop can be found in [11], along with proofs of subject reduction and soundness of the I-Soop type system. In this paper we only present the I-Soop type rules, taking their soundness as given.

We translate I-Loop terms, types, and judgements into I-Soop by three translation functions. Then, we show a provable I-Loop judgement translates into a provable I-Soop judgement, and thus since I-Soop is type-sound, I-Loop is also type-sound. The I-Loop type rules were directly inspired by the untyped translation, so the translation serves more than merely to prove soundness, it provides for an understanding of I-Loop.

3.1 The I-Soop Language and Type System

I-Soop is very similar to I-Loop: it has rc types, and the typing rules are of a similar form. Since there are significant parallels between the two languages, the description of I-Soop will be terse. Many conventions used for I-Loop will also be used for I-Soop without explicit mention.

$$\begin{array}{lll} \mathit{Var} & \ni x & \\ \mathit{Num} & \ni n & ::= \mathsf{0} \mid \mathsf{1} \mid \mathsf{2} \mid \ldots \\ \mathit{Exp} & \ni e & ::= x \mid n \mid \lambda x.\, e \mid e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\ & & \quad \mid \{\,\overline{l = e}\,\} \mid e.l \mid \mathsf{case}\ e\ \mathsf{of}\ \overline{l \Rightarrow e} \mid l\ \#\ e \end{array}$$

The identifiers of the I-Soop built-in functions are $B = \{\mathsf{pred},\, \mathsf{succ},\, \mathsf{is\_zero},\, \mathsf{ref},\, !,\, \mathsf{set}\} \subseteq \mathit{Var}$.

$$
\begin{array}{rl}
\text{(i)} & \text{If } \{\tau_1 \le \tau_2,\ \tau_2 \le \tau_3\} \subseteq C, \text{ then } \tau_1 \le \tau_3 \in C. \\
\text{(ii)} & \text{If } \tau_1 \to \tau_1' \le \tau_2 \to \tau_2' \in C, \text{ then } \{\tau_2 \le \tau_1,\ \tau_1' \le \tau_2'\} \subseteq C. \\
\text{(iii)} & \text{If } \{\,\overline{l_i : \tau_i}\,\} \le \{\,\overline{l_j : \tau_j'}\,\} \in C \text{ and } \{\overline{l_i}\} \supseteq \{\overline{l_j}\}, \\
 & \qquad \text{then } \{\overline{\tau_j \le \tau_j'}\} \subseteq C. \\
\text{(iv)} & \text{If } [\,\overline{l_i : \tau_i}\,] \le [\,\overline{l_j : \tau_j'}\,] \in C \text{ and } \{\overline{l_i}\} \subseteq \{\overline{l_j}\}, \\
 & \qquad \text{then } \{\overline{\tau_i \le \tau_i'}\} \subseteq C. \\
\text{(v)} & \text{If } \tau_1\ \mathsf{ref} \le \tau_2\ \mathsf{ref} \in C, \text{ then } \{\tau_1 \le \tau_2,\ \tau_2 \le \tau_1\} \subseteq C.
\end{array}
$$

Figure 4: Definition of when I-SOOP constraint set C is *closed.*

The new syntax found in I-SOOP missing from I-LOOP includes records and reference cells. The syntax for records is standard. Imperative programming is allowed in I-SOOP via the "ML school," where normal variables x are immutable as in a functional language, and explicit reference cells, created by $x = \mathsf{ref}\ e$, may then be assigned to via $\mathsf{set}\ \{\,\mathsf{cell} = x, \mathsf{val} = e\,\}$ and read from via $!x$. Note reference cells are assigned by passing the set function a record; this notation is used for convenience of the technical presentation.

The monomorphic types of I-SOOP are

$$
\begin{array}{lll}
\mathit{TyVar} & \ni \alpha & ::= t \mid u \\
\mathit{Typ} & \ni \tau & ::= \alpha \mid \mathsf{Nat} \mid \tau \to \tau' \mid \{\,\overline{l : \tau}\,\} \mid [\,\overline{l : \tau}\,] \mid \tau\ \mathsf{ref}
\end{array}
$$

Definitions of closure and consistency are very similar to the related I-LOOP concepts, so only the main definitions will be given. The closure condition for I-SOOP constraint sets is defined in Figure 4.

DEFINITION 3.1 The following I-SOOP constraints are *immediately inconsistent*:

(i) $\tau \le \tau'$ where τ and τ' have different outermost type constructors;

(ii) $\{\,\overline{l : \tau}\,\} \le \{\,\overline{l' : \tau'}\,\}$, where $\{\overline{l}\} \not\supseteq \{\overline{l'}\}$; or

(iii) $[\,\overline{l : \tau}\,] \le [\,\overline{l' : \tau'}\,]$, where $\{\overline{l}\} \not\subseteq \{\overline{l'}\}$.

Definitions of rc type and subtyping for I-SOOP are direct analogues of the I-LOOP definitions. The initial type environment A_0 assigns the following type schemes to the I-SOOP built-ins:

$$
\begin{array}{l}
(\,\mathsf{is\_zero} : \mathsf{Nat} \to [\,\mathsf{true}\ : \{\,\},\ \mathsf{false}\ : \{\,\}\,], \\
\ \ \mathsf{pred} : \mathsf{Nat} \to \mathsf{Nat},\ \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat},\ \mathsf{ref} : \forall \mathsf{u}.\ \mathsf{u} \to \mathsf{u}\ \mathsf{ref}, \\
\ \ ! : \forall \mathsf{t}.\ \mathsf{t}\ \mathsf{ref} \to \mathsf{t},\ \mathsf{set} : \forall \mathsf{t}.\ \{\,\mathsf{cell} : \mathsf{t}\ \mathsf{ref}, \mathsf{val} : \mathsf{t}\,\} \to \mathsf{t}\,)
\end{array}
$$

The typing rules for I-SOOP are given in Figure 5. Notation used in the rules has the obvious analogue with I-LOOP.

3.2 The Translation Functions

First we give the semantics of I-LOOP terms by a mapping into I-SOOP. The translation of terms is at the

$$
\begin{array}{ll}
\text{(Sub)} & \dfrac{A \vdash_{\mathrm{S}} e : \kappa, \qquad \kappa \le \kappa'}{A \vdash_{\mathrm{S}} e : \kappa'} \\[2ex]
\text{(Num)} & \dfrac{}{A \vdash_{\mathrm{S}} n : \mathsf{Nat} \setminus \emptyset} \\[2ex]
\text{(Var)} & \dfrac{A(x) = \tau}{A \vdash_{\mathrm{S}} x : \tau \setminus \emptyset} \\[2ex]
\text{(PVar)} & \dfrac{A(x) = \forall \overline{\alpha}.\ \kappa, \qquad \Psi \text{ is a substitution on } \{\overline{\alpha}\}}{A \vdash_{\mathrm{S}} x : \Psi\kappa} \\[2ex]
\text{(Abs)} & \dfrac{A,\ x : \tau \vdash_{\mathrm{S}} e : \tau' \setminus C}{A \vdash_{\mathrm{S}} \lambda x.\, e : \tau \to \tau' \setminus C} \\[2ex]
\text{(App)} & \dfrac{A \vdash_{\mathrm{S}} e_1 : \tau \to \tau' \setminus C_1, \qquad A \vdash_{\mathrm{S}} e_2 : \tau \setminus C_2}{A \vdash_{\mathrm{S}} e_1\ e_2 : \tau' \setminus C_1 \cup C_2} \\[2ex]
\text{(Inj)} & \dfrac{A \vdash_{\mathrm{S}} e : \tau \setminus C}{A \vdash_{\mathrm{S}} l \,\#\, e : [\,l : \tau\,] \setminus C} \\[2ex]
\text{(Case)} & \dfrac{A \vdash_{\mathrm{S}} e : [\,\overline{l_i : \tau_i}\,] \setminus C, \qquad A \vdash_{\mathrm{S}} \overline{e_i : \tau_i \to \tau \setminus C_i}}{A \vdash_{\mathrm{S}} \mathsf{case}\ e\ \mathsf{of}\ \overline{l_i \Rightarrow e_i} : \tau \setminus C \cup \bigcup_i C_i} \\[2ex]
\text{(Sel)} & \dfrac{A \vdash_{\mathrm{S}} e : \{\,l : \tau\,\} \setminus C}{A \vdash_{\mathrm{S}} e.l : \tau \setminus C} \\[2ex]
\text{(Record)} & \dfrac{A \vdash_{\mathrm{S}} \overline{e_i : \tau_i \setminus C_i}}{A \vdash_{\mathrm{S}} \{\,\overline{l_i = e_i}\,\} : \{\,\overline{l_i : \tau_i}\,\} \setminus \bigcup_i C_i} \\[2ex]
\text{(Let)} & \dfrac{A \vdash_{\mathrm{S}} e : \tau \setminus C, \qquad A,\ x : \forall \overline{\alpha}.\ \tau \setminus C \vdash_{\mathrm{S}} e' : \tau' \setminus C'}{A \vdash_{\mathrm{S}} \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau' \setminus \Phi C \cup C'}
\end{array}
$$

where Φ is a renaming of $\{\overline{\alpha}\}$,

$$
\{\overline{\alpha}\} \subseteq \begin{cases} \text{if } e \text{ is expansive then } \mathit{AppClos}(\tau \setminus C,\ A) \\ \text{else } \mathit{Clos}(\tau \setminus C,\ A) \end{cases}
$$

Figure 5: Typing rules of I-SOOP.

top of Figure 6. The core idea of the translation is the classic encoding of objects as records, classes as functions from the self to a record, and **new** as a fixed-point operation [15]. However in order to get a general translation that works in a call-by-value setting, and incorporates instance variables and their initialization and hiding, more structure is required. Objects after translation are records of the form

$$
\lambda\_.\ \{\,\mathsf{inst} = \{\,\overline{y_j = e_j}\,\},\ \mathsf{meth} = \{\,\overline{m_k = e_k}\,\}\,\}
$$

i.e. a (frozen) record consisting of two sub-records, inst containing the instance variables and meth the methods. Observe that objects in fact contain the instances; the *type* translation will disallow access to instances in objects even though they are present. Each e_j is a reference cell. The outer freeze operation is necessary to allow a call-by-value fixed point to be taken. Given this explanation of objects, the translations of message send and instance read and write should make sense. The **close** operation is a no-op execution—just as object instances are in fact present, closing the self only restricts typings; **new** translates as a fixed point as expected.

TERM TRANSLATION:

$$\begin{aligned}
\llbracket e \leftarrow m \rrbracket &= (\llbracket e \rrbracket\ \{\,\}).\mathsf{meth}.m \\
\llbracket e.y \rrbracket &= !((\llbracket e \rrbracket\ \{\,\}).\mathsf{inst}.y) \\
\llbracket e_1.y := e_2 \rrbracket &= \mathsf{set}\ \{\mathsf{cell} = (\llbracket e_1 \rrbracket \{\,\}).\mathsf{inst}.y, \mathsf{val} = \llbracket e_2 \rrbracket\} \\
\llbracket \mathbf{new}\ e \rrbracket &= \mathsf{Y}(\llbracket e \rrbracket\ \{\,\}) \\
\llbracket \mathbf{close}\ e \rrbracket &= \llbracket e \rrbracket \\
\llbracket \lambda x.\, e \rrbracket &= \lambda x.\, \llbracket e \rrbracket \\
\llbracket e\ e' \rrbracket &= \llbracket e \rrbracket\ \llbracket e' \rrbracket \\
\llbracket \mathbf{let}\ x = e\ \mathbf{in}\ e' \rrbracket &= \mathsf{let}\ x = \llbracket e \rrbracket\ \mathsf{in}\ \llbracket e' \rrbracket
\end{aligned}$$

$$\llbracket \mathbf{class}\ s\ \mathbf{super}\ \overline{p_i\ \mathbf{of}\ e_i}\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = e_k} \rrbracket$$
$$=$$
$$\begin{aligned}
&\lambda_.\ \mathsf{let}\ \overline{p_i^0 = \llbracket e_i \rrbracket\ \{\,\}}\ \mathsf{in}\ \mathsf{let}\ \overline{p_i = p_i^0\ (\lambda_.\, \Omega)}\ \mathsf{in} \\
&\quad \mathsf{let}\ y = \{\overline{y_j = \mathsf{ref}\ \llbracket e_j \rrbracket}\}\ \mathsf{in} \\
&\qquad \lambda s.\ \mathsf{let}\ \overline{p_i = p_i^0\ (s)}\ \mathsf{in}\ \lambda_.\ \{\mathsf{inst} = y,\ \mathsf{meth} = \{\overline{m_k = \llbracket e_k \rrbracket}\}\}
\end{aligned}$$
where $\mathsf{Y} \stackrel{\text{def}}{=} \lambda \mathsf{y}.\,(\lambda \mathsf{x}.\, \mathsf{x}\ \mathsf{x})\ \lambda \mathsf{x}.\, \lambda \mathsf{z}.\, \mathsf{y}\ (\mathsf{x}\ \mathsf{x})\ \mathsf{z}$, and $\Omega \stackrel{\text{def}}{=} \mathsf{Y}\ (\lambda \mathsf{x}.\, \mathsf{x})$

TYPE TRANSLATION:

$$\llbracket \mathbf{Class}\ st\ \mathbf{Inst}\ i\beta\ \mathbf{Meth}\ m\beta \rrbracket$$
$$=$$
$$\{\,\} \rightarrow st \rightarrow \llbracket \mathbf{Inst}\ i\beta\ \mathbf{Meth}\ m\beta \rrbracket$$

$$\llbracket \mathbf{Inst}\, i\beta\, \mathbf{Meth}\, m\beta \rrbracket = \{\,\} \rightarrow \{\mathsf{inst} : \llbracket i\beta \rrbracket, \mathsf{meth} : \llbracket m\beta \rrbracket\}$$
$$\llbracket i\beta \rrbracket = \begin{cases} \alpha & \text{if } i\beta = \alpha \\ \{\overline{y_i : \llbracket \tau_i \rrbracket\ \mathsf{ref}}\} & \text{if } i\beta = \overline{y_i : \tau_i} \end{cases}$$
$$\llbracket m\beta \rrbracket = \begin{cases} \alpha & \text{if } m\beta = \alpha \\ \{\overline{m_i : \llbracket \tau_i \rrbracket}\} & \text{if } m\beta = \overline{m_i : \tau_i} \end{cases}$$

JUDGEMENT TRANSLATION:

$$\begin{aligned}
\llbracket \tau \le \tau' \rrbracket &= \llbracket \tau \rrbracket \le \llbracket \tau' \rrbracket \\
\llbracket \{\overline{\tau \le \tau'}\} \rrbracket &= \{\overline{\llbracket \tau \rrbracket \le \llbracket \tau' \rrbracket}\} \\
\llbracket \forall \overline{\alpha}.\, \tau \setminus C \rrbracket &= \forall \overline{\alpha}.\, \llbracket \tau \rrbracket \setminus \llbracket C \rrbracket \\
\llbracket \overline{x_i{:}\sigma_i} \vdash_{\mathrm{L}} e{:}\tau \setminus C \rrbracket &= \overline{x_i : \llbracket \sigma_i \rrbracket} \vdash_{\mathrm{S}} \llbracket e \rrbracket : \llbracket \tau \rrbracket \setminus \llbracket C \rrbracket
\end{aligned}$$

Figure 6: Semantics of I-LOOP by translation to I-SOOP

The least intuitive aspect of the translation is the translation of classes. To better understand, first consider the following simpler (but incorrect) version.

$$\begin{aligned}
&\llbracket \mathbf{class}\ s\ \mathbf{super}\ \overline{p_i\ \mathbf{of}\ e_i}\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = e_k} \rrbracket \stackrel{\text{def}}{=} \\
&\lambda s.\ \mathsf{let}\ \overline{p_i = p_i\ (s)}\ \mathsf{in} \\
&\qquad \lambda_.\ \{\mathsf{inst} = \{\overline{y_j = \mathsf{ref}\ \llbracket e_j \rrbracket}\},\ \mathsf{meth} = \{\overline{m_k = \llbracket e_k \rrbracket}\}\}
\end{aligned}$$

A class here is a function from self s to an object, where the object bodies have access to $\overline{p_i}$ for the superclasses. Objects are then created by taking the fixed point of such a class. The problem with this translation is objects produced will re-initialize their instance variables every time a method of the object is invoked. The actual translation hoists the initialization of the instances outside to prevent this.[7]

[7] The shallow nature of the I-SOOP type system imposes additional restrictions on the form of translation that may be used; in [12] we presented a simpler translation of classes that in addition allows s to be used in instance variable initializations. That translation could in theory be used here, but objects would necessarily have imperative type, restricting polymorphism.

The translation of types and judgements is given at the bottom of Figure 6. The base types and function types translate homomorphically, so are not given. The **Inst-Meth** types are the types of objects with instance variables visible; since objects are frozen inst-meth records, as described above, object types are translated as frozen inst-meth record types. Compared to standard interpretations of object types, this type may seem unusual: since the self type may be used in objects, object types are usually interpreted as recursive types (see *e.g.* [12, 3]). In I-SOOP there are no recursive types; the recursion is implicitly found in the constraints imposed on the individual method types, and these constraints may be circular. Classes are frozen structures that take the self as argument and return records, so class types are frozen functions from self type st to an object type. This translation gives a fixed type st to the self in the class. This might appear dangerous, as it would seem all extensions to this class and all objects made by the class would share the same self type. However classes are intended to be defined via a **let**-expression, as in the IntOrd example, and the class will thus have a polymorphic type inferred and each object and subclass will have a different version of st.

3.3 Type Soundness of I-LOOP

We now sketch how type soundness may be proven for I-LOOP. Full proofs are omitted for lack of space. The obvious statement of soundness is: each provable I-LOOP judgement translates to a provable I-SOOP judgement. However this is not true for *all* I-LOOP proofs, since I-LOOP is in some ways stronger than I-SOOP in that it has a simplification rule. What we do instead is show proofs without (Simp) translate to I-SOOP proofs, and then use the completeness property of I-LOOP to show this means all I-LOOP proofs are sound.

LEMMA 3.2 (RESTRICTED I-LOOP SOUNDNESS) If $A \vdash_{\mathrm{L}} e : \tau \setminus C$ and this proof does not use rule (Simp), then $\llbracket A \rrbracket \vdash_{\mathrm{S}} \llbracket e \rrbracket : \llbracket \tau \rrbracket \setminus \llbracket C \rrbracket \cup C'$.

PROOF SKETCH: The proof proceeds by showing that for each I-LOOP inference rule, there is a I-SOOP derivation that starts with the translations of the hypotheses of the I-LOOP rule and ends with the translation of the conclusion of the rule. For most of the I-LOOP rules this fact follows directly. The (Class) rule has the added complication of needing a typing for $\lambda x.\, \Omega$; with will require extra constraints be added to type this function, the source of the C' above. In the (New) rule, typing Y will produce similar extra constraints. □

THEOREM 3.3 (I-LOOP SOUNDNESS) If $\vdash_{\mathrm{L}}\ e:\kappa_{\mathrm{L}}$, then $\vdash_{\mathrm{S}} [\![\, e\,]\!] : \kappa_{\mathrm{S}}$ for some κ_{S}.

PROOF SKETCH: By completeness of the type inference rules (Theorem 2.5), $\vdash^{\mathrm{inf}}_{\mathrm{L}}\ e:\kappa'_{\mathrm{L}}$; then, by Lemma 2.3, $\vdash^{\mathrm{inf}}_{\mathrm{L}}\ e:\kappa''_{\mathrm{L}}$ in a proof not containing any (Simp) steps. Then this proof may be easily transformed into a non-inference proof $\vdash_{\mathrm{L}}\ e:\kappa''_{\mathrm{L}}$ without (Simp), and finally, Lemma 3.2 applied to show $\vdash_{\mathrm{S}} [\![\, e\,]\!] : \kappa_{\mathrm{S}}$ for some κ_{S}. □

Thus by soundness of I-SOOP the evaluation of $[\![\, e\,]\!]$ cannot get stuck, and hence no "message not understood" errors will occur during execution of I-LOOP programs translated into I-SOOP.

4 An Example

We now show the I-LOOP inference algorithm functioning on a larger example. Consider the following I-LOOP program.

```
let View = class self super
      inst dep = None # ()
      meth
          doall = λf.
                f(self);
                case self.dep of
                      None ⇒ λ_. (),
                      Some ⇒ λd. (d ← doall f; ()),
          setDep = λv. (self.dep := Some # v; ())
let GView = class self super p of View
      inst dep = p.dep
      meth
          doall = p ← doall,
          setDep = p ← setDep,
          draw = λ_. ()
let v1 = new View in let g1 = new GView in
let g2 = new GView in let g3 = new GView in
     g1 ← setDep v1;
     g2 ← setDep g3; g2 ← doall (λob. ob ← draw ())
```

In this oversimplified example, these classes are Views in name only; other methods and instances would be present to perform various view functions. We focus here on a dependency mechanism of the view system: one view may contain other dependent views, and it may be necessary to perform some function for each dependent of a particular view. To make it simple, here we limit ourselves to the case where each view here has at most one dependent dep. If dep is None # (), there is no dependent, and if dep is Some # obj, then obj is the dependent view object. The doall method takes a function f and applies it to the view itself before passing it to doall of its dependent view, which in turn passes it to its dependent, etc. GView is a subclass of View intended to be a graphics view with a draw method of some sort; here we simplify again—draw does nothing.

What is interesting is that it is not possible to determine in advance how the programmer intends this code to behave when the View class is extended with new functionality in subclasses. There are two possibilities. The dependent views could be heterogenous classes, and thus the function passed to doall must only work on methods defined in View. The first message send, g1 ← setDep v1, reflects this approach: v1 has no draw method, so in a message send g1 ← doall f, f cannot invoke draw on its argument. This is the "subclasses are subtypes" philosophy: the type of g1's dep has been lifted to be a View. Alternatively, the programmer may intend to have a series of homogenous nested view classes, as in the message sends to g2. The function passed to doall should be able to invoke all of the methods defined in GView, in particular draw. This is what is gained from the "inheritance is not subtyping" philosophy, not requiring the type of dep to be fixed as a View [5, 9].

In previous work [12] we gave a typing system which allowed the programmer to choose between these two behaviors by the type given to the doall method. However, in that system the programmer was forced to allow only *one* of the alternatives for the View class hierarchy, based on the type given to the class. What is needed is some sort of "principal" type that accommodates both approaches, and this is exactly what our inference algorithm accomplishes. In particular, for the above program the following type scheme is assigned to View by the rule (Let) (after applying (Simp)):

$$\forall \mathsf{st}, \mathsf{u}, \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3, \mathsf{t}_4, \mathsf{t}_5, \mathsf{t}_6, \mathsf{t}_7.$$
$$\textbf{Class}\ \mathsf{st}\ \textbf{Inst}\ \mathsf{dep}:\mathsf{u}\ \textbf{Meth}\ \mathsf{doall}:\mathsf{t}_1\!\longrightarrow\!\mathbf{1}, \mathsf{setDep}:\mathsf{t}_2\!\longrightarrow\!\mathbf{1} \setminus C$$

where C is the constraint system

$$\begin{array}{rcl}
[\,\mathsf{None}:\mathbf{1}\,] & \leq & \mathsf{u}\\
\mathsf{t}_1 & \leq & \mathsf{st}\!\longrightarrow\!\mathsf{t}_3\\
\mathsf{st} & \leq & \textbf{Inst}\ \mathsf{dep}:\mathsf{t}_4\\
\mathsf{t}_4 & \leq & [\,\mathsf{None}:\mathsf{t}_5, \mathsf{Some}:\textbf{Meth}\ \mathsf{doall}:\mathsf{t}_1\!\longrightarrow\!\mathsf{t}_6\,]\\
\mathsf{st} & \leq & \textbf{Inst}\ \mathsf{dep}:\mathsf{t}_7\\
[\,\mathsf{Some}:\mathsf{t}_2\,] & \leq & \mathsf{t}_7
\end{array}$$

The only requirements on st are on its instance varible, because *e.g.* self ← doall ... is not found anywhere. In a similar way, we can infer the type scheme inferred for GView to be

$$\forall \mathsf{st}, \mathsf{u}, \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3, \mathsf{t}_4, \mathsf{t}_5, \mathsf{t}_6, \mathsf{t}_7.$$
$$\textbf{Class}\ \mathsf{st}\ \textbf{Inst}\ \mathsf{dep}:\mathsf{u}$$
$$\textbf{Meth}\ \mathsf{doall}:\mathsf{t}_1\!\longrightarrow\!\mathbf{1}, \mathsf{setDep}:\mathsf{t}_2\!\longrightarrow\!\mathbf{1}, \mathsf{draw}:\mathsf{t}_8\!\longrightarrow\!\mathbf{1} \setminus C$$

It turns out that the constraint system in the type of GView is identical to the system of View—the type of the added method is trivial and does not introduce new constraints.

When we create instances of View and GView, we "tie the knot" between what each class *implements* (the **Inst** ... **Meth** ... portion of the class type) and what each class *requires* (the constraints on type variable st). This introduces new constraints; v1 has the inferred type

$$\textbf{Meth}\ \mathsf{doall} : \mathsf{t}_1' \to \mathbf{1}, \mathsf{setDep} : \mathsf{t}_2' \to \mathbf{1} \setminus C'$$

where C' is

$$\begin{array}{rcl} [\,\mathsf{None} : \mathbf{1}\,] & \leq & \mathsf{u}' \\ [\,\mathsf{Some} : \mathsf{t}_2'\,] & \leq & \mathsf{u}' \\ \mathsf{u}' & \leq & [\,\mathsf{None} : \mathsf{t}_5', \mathsf{Some} : \textbf{Meth}\ \mathsf{doall} : \mathsf{t}_1' \to \mathsf{t}_6'\,] \\ \mathsf{t}_2' & \leq & \textbf{Meth}\ \mathsf{doall} : \mathsf{t}_1' \to \mathsf{t}_6' \\ \mathsf{t}_1' & \leq & (\textbf{Inst}\ \mathsf{dep} : \mathsf{u}' \\ & & \ \textbf{Meth}\ \mathsf{doall} : \mathsf{t}_1' \to \mathbf{1}, \mathsf{setDep} : \mathsf{t}_2' \to \mathbf{1}) \to \mathsf{t}_3' \end{array}$$

As View and GView are let-bound, the type variables of constraint system C are generalized, and each use of View or GView gets unique instances of these variables via the renaming of the (PVar) rule. We indicate this by the primed variables; we use double primes in the inferred type of g1:

$$\textbf{Meth}\ \mathsf{doall} : \mathsf{t}_1'' \to \mathbf{1}, \mathsf{setDep} : \mathsf{t}_2'' \to \mathbf{1}, \mathsf{draw} : \mathsf{t}_8'' \to \mathbf{1} \setminus C''$$

where C'' is

$$\begin{array}{rcl} [\,\mathsf{None} : \mathbf{1}\,] & \leq & \mathsf{u}'' \\ [\,\mathsf{Some} : \mathsf{t}_2''\,] & \leq & \mathsf{u}'' \\ \mathsf{u}'' & \leq & [\,\mathsf{None} : \mathsf{t}_5'', \mathsf{Some} : \textbf{Meth}\ \mathsf{doall} : \mathsf{t}_1'' \to \mathsf{t}_6''\,] \\ \mathsf{t}_2'' & \leq & \textbf{Meth}\ \mathsf{doall} : \mathsf{t}_1'' \to \mathsf{t}_6'' \\ \mathsf{t}_1'' & \leq & (\textbf{Inst}\ \mathsf{dep} : \mathsf{u}''\ \textbf{Meth}\ \mathsf{doall} : \mathsf{t}_1'' \to \mathbf{1}, \\ & & \ \mathsf{setDep} : \mathsf{t}_2'' \to \mathbf{1}, \mathsf{draw} : \mathsf{t}_8'' \to \mathbf{1}) \quad \to \mathsf{t}_3'' \end{array}$$

Now, we may perform g1 ← setDep v1, as in the example. When we do this, new constraints are introduced on the types of the doall and setDep methods of g1; a new constraint inferred is that the type of v1 must be a subtype of the domain type of method setDep of g1:

$$\textbf{Meth}\ \mathsf{doall} : \mathsf{t}_1' \to \mathbf{1}, \mathsf{setDep} : \mathsf{t}_2' \to \mathbf{1} \leq \mathsf{t}_2''$$

Since t_2'' already had an upper bound in C'', by transitivity we have

$$\textbf{Meth}\ \mathsf{doall} : \mathsf{t}_1' \to \mathbf{1}, \mathsf{setDep} : \mathsf{t}_2' \to \mathbf{1} \leq \textbf{Meth}\ \mathsf{doall} : \mathsf{t}_1'' \to \mathsf{t}_6''$$

which implies (cf. Figure 1) $\mathsf{t}_1'' \leq \mathsf{t}_1'$. If we now attempt to invoke g1 ← doall with the argument $\boldsymbol{\lambda}$ob. ob ← draw (), whose inferred type is $(\textbf{Meth}\ \mathsf{draw} : \mathbf{1} \to \mathsf{t}) \to \mathsf{t}$, we get the additional constraint

$$(\textbf{Meth}\ \mathsf{draw} : \mathbf{1} \to \mathsf{t}) \to \mathsf{t} \leq \mathsf{t}_1''$$

which together with $\mathsf{t}_1'' \leq \mathsf{t}_1'$ and the upper bound on t_1' in C' yields

$$\begin{array}{l} (\textbf{Meth}\ \mathsf{draw} : \mathbf{1} \to \mathsf{t}) \to \mathsf{t} \\ \leq (\textbf{Inst}\ \mathsf{dep} : \mathsf{u}'\ \textbf{Meth}\ \mathsf{doall} : \mathsf{t}_1' \to \mathbf{1}, \mathsf{setDep} : \mathsf{t}_2' \to \mathbf{1}) \to \mathsf{t}_3' \end{array}$$

The argument types of these function types must then be in the reverse subtyping relation:

$$\begin{array}{l} \textbf{Inst}\ \mathsf{dep} : \mathsf{u}'\ \textbf{Meth}\ \mathsf{doall} : \mathsf{t}_1' \to \mathbf{1}, \mathsf{setDep} : \mathsf{t}_2' \to \mathbf{1} \\ \leq \textbf{Meth}\ \mathsf{draw} : \mathbf{1} \to \mathsf{t} \end{array}$$

But an object type without draw cannot be a subtype of an object type with draw—this is an inconsistent constraint (by definition 2.1). This is fortunate because if this were allowed it would produce a run-time error, as v1 is g1's dependent and it will not understand draw. However, consider the line g2 ← doall($\boldsymbol{\lambda}$ob. ob ← draw ()) from the example: it is similar to the above but since g2's dependent is a GView, the lower bound in the final constraint above will include draw, so no inconsistency will be found. The single View class will thus be useful in the case of both homogenous and heterogenous dependents.

5 Discussion

The inference algorithm presented here adds a powerful degree of expressivity to typed object-oriented programming. This paper has presented the basic ideas, but leaves upen a number of problems. Type simplification is a critical issue since the constraints may be in a non-optimal form; some initial results were presented but more work needs to be done. One of the most important related problems is developing a user interface for presenting the program text and associated type information meaningfully to the programmer. Since the types are of a different nature than standard types, significantly different approaches will be needed when compared to traditional language designs. One simple tool that is needed is an editor that shows the possible sources of a constraint inconsistency by *e.g.* highlighting the portion(s) of code where the error may have arisen. This tool puts the type system to its best advantage: much more information about the program is encoded in the types, so more precise descriptions of type errors may be given. Two language features missing from I-Loop that will be challenging to incorporate are modules and a notion of friend functions as found in C++.

This type system also makes contributions to the issue of typing object-oriented programs. The precision of the types allows the "inheritance vs subtyping" decision to be postponed, increasing flexibility. Bruce's PolyTOIL language [5] has stronger form of polymorphism that allows many of the same notions to be expressed, but the price is the programmer must come up with the complicated PolyTOIL typings, a potentially daunting task. Our system also allows some union and intersection types to be expressed, a feature absent from PolyTOIL. PolyTOIL types on the other hand do not suffer from the readability problem of I-Loop constrained types. An alternate school of thought is to use multi-methods instead of binary methods [8, 19]. The multi-method approach has the advantage of avoiding the tradeoff between inheritance and subtyping. One advantage of our rich type language is Ingalls' solution [13] can be used to encode multiple dispatch in our single-dispatch framework. A fuller discussion of these topics is found in [4].

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] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 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 Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. Technical Report TR95-08, Department of Computer Science, Iowa State University, Ames, Iowa 50011-1040 USA. `ftp://ftp.cs.iastate.edu/pub/techreports/TR95-08/TR.ps.Z`, 1995.

[5] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *ECOOP '95*, 1995.

[6] 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.

[7] 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.

[8] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3), 1995.

[9] 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.

[10] Pavel Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, XEROX Palo Alto Research Center, `CSLPubs.parc@xerox.com`, 1990.

[11] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[12] J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. Application of OOP type theory: State, decidability, integration. In *OOPSLA '94*, pages 16–30, 1994.

[13] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *OOPSLA '86 Conference Proceedings, Portland, Oregon, September 1986*, volume 21(11), pages 347–349, November 1986.

[14] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *ACM Conference on Lisp and Functional Programming*, pages 193–204, 1992.

[15] 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.

[16] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[17] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[18] J. Mitchell. Coercion and type inference (summary). In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1984.

[19] W. B. Mugridge, J. G. Hosking, and J. Hamer. Multi-methods in a statically-typed programming language. In Pierre America, editor, *ECOOP '91 Conference Proceedings, Geneva, Switzerland*, volume 512 of *Lecture notes in Computer Science*. Springer-Verlag, 1991.

[20] N. Oxhøj, J. Palsberg, and M. I. Schwartzbach. Type inference with subtypes. In *ECOOP'92 European Conference on Object-Oriented Programming*, volume 615 of *Lecture notes in Computer Science*, pages 329–349. Springer-Verlag, 1992.

[21] J. Palsberg and M. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.

[22] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference for partial types. *Information Processing Letters*, pages 175–180, 1992.

[23] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–340, 1994.

[24] J. C. Reynolds. Three approaches to type structure. In *TAPSOFT proceedings*, volume 185 of *Lecture notes in Computer Science*, pages 97–138, 1985.

[25] T. Sekiguchi and A. Yonezawa. A complete type inference system for subtyped recursive types. In *Proc. Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 667–686. Springer-Verlag, 1994.

[26] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 162–173. IEEE, 1992.

[27] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[28] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, pages 37–44. IEEE, 1987.

A Simplification

To define formally the concept of reachability, we first introduce the operators T_{+1} and T_{-1} to extract the types

$$Reachable(A,\,\tau,\,C) \stackrel{\text{def}}{=} R_{+1}(\tau) \cup \Big(\bigcup_{\tau' \in Codom(A)} R_{-1}(\tau')\Big) \cup \Big(\bigcup_{u \in FTV(\sigma),\, \sigma \in Codom(A)} (R_{+1}(u) \cup R_{-1}(u))\Big),$$

where

$$\begin{array}{rlrl}
R_p(\alpha) = & D_p(\alpha) & R_p(\tau_1 \to \tau_2) = & R_{-p}(\tau_1) \cup R_p(\tau_2) \\
R_p(\mathsf{Nat}) = & \emptyset & R_p(\textbf{Class}\ st\ \textbf{Inst}\ i\beta\ \textbf{Meth}\ m\beta) = & R_{-p}(st) \cup R_p(\textbf{Inst}\ i\beta\ \textbf{Meth}\ m\beta) \\
R_p(\mathsf{Bool}) = & \emptyset & R_p(\textbf{Inst}\ i\beta\ \textbf{Meth}\ m\beta) = & R_p(\textbf{Inst}\ i\beta) \cup R_p(\textbf{Meth}\ m\beta) \\
R_p(\textbf{Inst}\ ir) = & D_p(\textbf{Inst}\ ir) & R_p(\textbf{Inst}\ \overline{y_j : \tau_j}) = & \bigcup_j (R_{+1}(\tau_j) \cup R_{-1}(\tau_j)) \\
R_p(\textbf{Meth}\ mr) = & D_p(\textbf{Meth}\ mr) & R_p(\textbf{Meth}\ \overline{m_k : \tau_k}) = & \bigcup_k R_p(\tau_k)
\end{array}$$

$$D_p(\tau) = C_p(\tau) \cup \bigcup_{c \in C_p(\tau)} R_p(T_p(c)) \qquad C_p(\tau) = \{c \mid c \in Cl(C) \text{ and } T_{-p}(c) = \tau\}$$

Figure 7: Reachability Definition

involved in a type constraint:

$$T_{+1}(\tau_1 \le \tau_2) \stackrel{\text{def}}{=} \tau_1, \quad T_{-1}(\tau_1 \le \tau_2) \stackrel{\text{def}}{=} \tau_2$$

The set of constraints in C *reachable* from A and τ is then defined in Figure 7. In this definition the subscript p ranges over $\{-1, +1\}$. We say a type τ' occurs in a *positive* (resp. *negative*) position in $\langle A,\, \tau \setminus C\rangle$ if computing $Reachable(A,\,\tau,\,C)$ involves computing $R_{+1}(\tau')$ (resp. $R_{-1}(\tau')$). Effectively we perform a garbage collection on the closure of the set of constraints, taking into accout that further applications of typing rules can only (i) put upper bounds on the "root type" τ, hence we trace it "downwards," searching its current lower bounds; (ii) put lower bounds on the type τ' of a variable introduced by rules (Abs) (by placing τ' in the negative position of the function's type) or (Class); or (iii) put upper or lower bounds on imperative type variables free in the type scheme of a let-bound variable (by reading its value or assigning to it, respectively). The result of this garbage collection is a constraint system of a special kind: each constraint in it specifies either an upper or a lower bound on a type variable.

Further transformations of the rc type allowed by the (Simp) include:

Replacing a variable by its bound: Define $SubLower_t^A(\tau \setminus C)$ as follows: if t occurs only positively in $\langle A,\, \tau \setminus C\rangle$, and there is only one constraint of the form $\tau' \le t$ in C, and $t \notin FTV(\tau')$, then $SubLower_t^A(\tau \setminus C) = (\tau \setminus (C - \{\tau' \le t\}))[\tau'/t]$, *i.e.* the rc type with τ' substituted everywhere for t; otherwise $SubLower_t^A(\tau \setminus C) = \tau \setminus C$.

A similar simplification $SubUpper_t^A(\tau \setminus C)$ can be performed to replace negatively occurring variables with their upper bounds.

Merging of variables: Define $Merge_{t,t'}^A(\tau \setminus C)$ as follows: if $\{t \le t',\, t' \le t\} \subseteq C$ and t' is not free in A, then $Merge_{t,t'}(\tau \setminus C) = (\tau \setminus C)[t/t']$; otherwise $Merge_{t,t'}(\tau \setminus C) = \tau \setminus C$.

We omit for lack of space the descriptions of other transformations that can simplify the types. The function *Simplified* is defined as the composition of *Reachable*, *SubLower*, *SubUpper*, and *Merge*, applied for all free variables in the rc type.

Proof Sketch of Lemma 2.3: The transformation proceeds by bubbling all (Simp) uses from the leaves to the root of the proof tree. To establish this, it must be shown that (Simp) commutes with every other I-Loop inference rule. Consider first the garbage collection simplification. By induction on the definition of the *Reachable* set, we can show that (i) if the type variables t and t' are free neither in A nor the *Reachable* set, and t is free in the constraint system before applying (Simp), then the proof below the application of (Simp) can be α-converted to use t instead of t'; (ii) if a constraint c, in which τ is in a positive position, is consistent with C, then it is consistent with $Reachable(A,\,\tau,\,C)$. By (i) it follows that the union of constraint systems is consistent, if the union of their garbage-collected versions is (since we can add back the unreachable constraints while renaming variables to avoid accidental reuse and therefore possible inconsistencies). By (ii) we have that the constraint systems before the garbage collection are consistent with the new constraints added by some of the rules (since all of these constraints have the "root types" from their premises in positive positions). The (Let) and (PVar) rules complicate the matter somewhat as they can produce many copies of the garbage, and can create many copies of "root types," but these are still all in positive positions. Regarding the other transformations in *Simplified*, it suffices to notice that their conditions for applicability are monotone in C (provided type variables not free in A are renamed to avoid reuse in the new constraints), hence we can still apply them after combining several constraint systems.

At this point, all applications of (Simp) have been bubbled from the leaves to the root, so the proof has only one instance of (Simp), at the root. The last step of the proof transformation is to remove the only (Simp), observing that the I-Loop expressions in its premise and conclusion are the same. □