Electronic Notes in Theoretical Computer Science 1 (1995)
URL: http://www.elsevier.nl/locate/entcs/volume1.html 22 pages

Type Inference for Recursively Constrained Types and its Application to OOP

Jonathan Eifrig [1,2] Scott Smith [1] Valery Trifonov [1,2]

Department of Computer Science
The Johns Hopkins University
Baltimore, Maryland 21218
{eifrig, scott, trifonov}@cs.jhu.edu

Abstract

We define a powerful type inference mechanism with application to object-oriented programming. The types inferred are *recursively constrained* types, types that come with a system of constraints. These types may be viewed as generalizations of recursive types and F-bounded polymorphic types, the forms of type that are necessary to properly encode object typings. The base language we study, I-Soop, incorporates state and records, the two features critical to encode objects in a non-object-oriented language. Soundness and completeness of the type inference algorithm are established by operational means. Our method for establishing these properties is somewhat novel. We illustrate how the algorithm may be fruitfully applied to infer types of object-oriented programs.

1 Introduction

This paper addresses the problem of designing an object-oriented programming language with an effective type inference mechanism. Recently developed programming languages including Standard ML and Haskell incorporate type inference as a core component of the language. However, type inference has yet to achieve practical application to object-oriented programming languages.

We strongly feel the core type features necessary to model object-oriented programming with type inference include a notion of subtyping [8], and a notion of "recursively constrained polymorphism," a generalization of F-bounded polymorphism [9,6].

Recursively constrained types κ are types of the form $\tau \setminus C$, with " $\setminus$ " reading "where." C is a set of type constraints of the form $\tau_1 \leq \tau_2$, possibly containing free type variables. These constraints may be *recursive* in that a

[1] Partially supported by NSF grants CCR-9109070 and CCR-9301340
[2] Partially supported by AFOSR grant F49620-93-1-0169

variable t could occur free in both τ_1 and τ_2. The recursive constraint set $\{t \rightarrow \mathsf{Nat} \leq t, t \leq t \rightarrow \mathsf{Nat}\}$ expresses $t = t \rightarrow \mathsf{Nat}$, so recursively constrained types subsume recursive types. We will use *rc type* to abbreviate recursively constrained type.

Polymorphic rc types are types $\forall t_1, \ldots, t_n. \tau \setminus C$ where constraints $\tau_1 \leq \tau_2$ in C may contain type variables $t_1, \ldots t_n$ free. Polymorphic rc types generalize the more well-known bounded types [8] $\forall t \leq \tau. \tau'$ in several ways. First, they are recursive, so t could occur free in τ; this is not allowed in bounded types. Types with t occurring free in τ are the so-called F-bounded types [6]. Polymorphic rc types generalize F-bounded types by allowing more than one upper bound on a type variable, as well as allowing multiple lower-bound constraints $\tau \leq t$. This generalized form of polymorphic type is very useful in typing object-oriented programs that are otherwise untypable, irrespective of the question of type inference. An example of such a program is given in Section 5 below.

It is not difficult to see how rc polymorphism is useful in typing classes and objects, for it is at least as useful as F-bounded polymorphism. Classes may have so-called binary methods that refer to the type of objects of their own class; for instance an object with an equal method takes as parameter another object of its own type. Thus, a self-type is needed. And, this self-type needs to be open-ended since a class may be extended; we wish the type of self to be “an object with all the methods currently defined, and possibly additional ones”. Polymorphic rc types capture this notion by constraining the polymorphic “self-type” t to include the current methods, for instance

$$\forall t. \tau \setminus \{t \leq \ldots \mathsf{equal} : t \rightarrow \mathsf{Bool}, \ldots\}$$

Binary methods have proven very difficult to type in a general way; it has even been suggested that they be disallowed.

One way to understand the usefulness of lower bounds $\tau \leq t$ in rc types are as generalizations of recursive types. It is possible to write an rc type $\kappa = t \setminus \{\tau_1 \leq t \leq \tau_2\}$ where lower bound τ_1 differs from upper bound τ_2 (it is a recursive type if $\tau_1 = \tau_2$). These generalized forms are useful as intermediate results produced during the type inference process as “partial” forms of recursive types. During the type inference process, constraints are accumulated on types in a “bottom-up” fashion, and so types at the leaves of typing proofs have small constraint sets, and have fat constraint sets at the root. The lower bound τ_1 constrains the “output” of the type κ (what properties objects of type κ must have); if an object of type κ is used (*i.e.*, passed to a function of type $\tau' \rightarrow \ldots$), an additional upper-bound constraint $t \leq \tau'$ will be placed on the type by the type inference mechanism, and this could only be contradictory if $\tau_1 \leq \tau'$, which follows by transitivity, was contradictory. The upper bound is the dual of this, constraining the “input” of the type (what functions of type $t \rightarrow \ldots$ must do).

The presence of multiple upper-bound constraints or multiple lower-bound constraints can be understood as a restricted form of union and intersection type: $\{\tau \leq t, \tau' \leq t\}$ would be equivalent to $\{\tau \vee \tau' \leq t\}$ if there were union types $\tau \vee \tau'$ in the language; a dual relationship exists between intersections and upper bounds. We believe general union and intersection types cause too many problems to be worthwhile, but this implicit restricted form is quite natural.

In this paper we develop a type inference algorithm for the I-Soop language (Inference Semantics of OOP). I-Soop is not an object-oriented language; however, it has an expressive enough type system so that typed OOP may be effectively encoded within I-Soop. We take a translational approach because we find the factoring to help clarify ambiguities; however, there is also merit in studying languages where objects themselves are primitive [1], and the concepts herein should eventually be recast as primitive object typings. I-Soop's type system contains both subtyping and polymorphic rc types. We infer shallow polymorphic rc types at let-expressions as in the Hindley/Milner algorithm [17]. In addition the underlying language includes records and a notion of state, for with these features it is possible to obtain an effective encoding of object-oriented programming. Records are needed so record subtyping can be used to model object subtyping [8]. Without state, the critical state-holding property of objects is lost [11].

Our approach to establishing the soundness of constrained type inference differs from other work in the literature. In other approaches (e.g. [3,13], [25,20]), a method is given that either produces a satisfying assignment to the constraints and thus establishes their consistency, or establishes that no such solution exists and the constraints are thus inconsistent. In our approach, an rc type's constraint system is considered "consistent" if it does not contain any "obvious" contradictions such as $\mathsf{Nat} \leq \mathsf{Bool}$. We show this view is sound, *without* ever showing the "consistent" constraint systems have solutions. Instead we directly establish a subject-reduction property over a proof of typing with "consistent" rc types at each node [26,27]. We believe the standard method of finding solutions to the constraint sets can be overly restrictive, for it forces one to have a rich enough type language or type model that can express the solutions as types or sets. In our language, for instance, we expect general union and intersection types would be required to express the solution of constraints as types, but we do not wish to pay the penalty of having these types in our language.

We also take a more primitive approach to establishing the completeness of type inference, i.e. that all typable programs will successfully have some type inferred by the type inference algorithm. We first define a restricted set of typing rules, the *inference rules*, for which typing derivations are deterministic. Then these rules are shown equivalent in strength to the general form of rules, without recourse to a "principal types" property.

1.1 Related Work

A number of type inference systems have been developed that bear on the type inference problem for OOP. Papers of Reynolds [24], Cardelli [7], and Mitchell [18] are foundational papers in the field that develop the basic concepts of constraints and subtyping. Many papers have been written since; we focus on the more recent work the most relevant to ours.

Kaes [13] develops a type inference algorithm for a language containing polymorphic and recursive types and type constraints. This work incorporates subtyping constraints, recursive types, and polymorphism. Kaes writes so-called constrained types $\tau|C$ in close analogy to our rc types $\tau \setminus C$. This approach cannot solve general recursive constraints: $t \leq \tau$ generates a non-terminating unification problem in his system if t occurs free in τ, while our approach can handle such constraints without difficulty. He does allow a "fixing" of such a constraint by replacing it with a recursive type $\mu t.\tau$, but at the cost of an important loss of generality. Kaes takes the standard approach to constraint consistency, by producing a solution to the constraints. He also intends $\leq$ to model overloading, not record subtyping (his system has no record types). Sekiguchi and Yonezawa [25] take an approach similar to Kaes but interpret $\leq$ as subtyping on record types, making it more directly applicable to object-oriented programming.

Palsberg, Schwartzbach, *et. al.* have written a number of papers concerning type inference for objects [20,19,21,15]. The main feature of their work is they do not take the Hindley/Milner approach to type inference. Instead, their inference algorithm uses flow analysis to generate a set of constraints about a program, and then applies another algorithm to come up with a solution to these constraints if it exists. Their work represents the current state-of-the-art in having a practical type inference algorithm for object-oriented programming languages. 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. This lack of polymorphism has been partially addressed by Plevyak and Chien [22].

Our work is closest to that of Aiken and Wimmers [3]. They develop a type system with subtyping, union and intersection types, and a form of polymorphic type similar to polymorphic rc types. They prove soundness using the ideal model [16]. As with the previously mentioned researchers, they have an algorithm that produces a satisfying assignment to the top-level constraints to establish consistency of a constraint set. The satisfying assignment they produce is an ideal in the ideal model. We have no union, intersection, or negation types. These types prove problematic in their system, and they are in fact unnecessary for type inference — if they are not used in the types of atomic constructs, they are not generated by the inference algorithm (provided multiple upper and lower bounds to the same variable are allowed, as we do). Aiken and Wimmers have not addressed the problem of using their system for typing object-oriented programs; their language lacks important features necessary for the encoding of objects. In particular their language is a functional language without records. The ideal model cannot model languages with state, so their approach would not extend to a language with state. Aiken has implemented the type inference algorithm [2], and this implemented system has an optimized inference algorithm and an implementation of extensible records.

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

Fig. 1. Syntax of the I-Soop language.

Encoding object-oriented features within a more basic language is one possible approach to how object-oriented programming should be done [23]. We could take a similar approach by programming in an object-oriented style via the encoding of objects in I-Soop that we give in Section 5. Rémy gives a collection of extensions to ML that allow OOP to be encoded. Rémy is the only author amongst of those previously discussed who has a proof of soundness of his system in the presence of reference cells. His encoding is missing a notion of subtyping and thus lacks the core feature of object lifting: allowing subclass objects to be implicitly coerced to be superclass objects. Instead, coercion functions must be explicitly supplied. Rémy's encoding is more efficient than the encoding we use; each object creation in our encoding entails forming closures for each method of the object. If our language were to be used as a primitive OOP language, some more efficient object representations would need to be developed. Rémy's system also has a notion of extensible record, which we expect will be useful for encoding delegation-style object-oriented programming.

1.2 Outline

In Section 2 we present I-Soop and its operational semantics. Section 3 presents the I-Soop type system; sketches of the proofs of subject reduction and type inference appear in Section 4. Then, to show how OOP can be faithfully encoded, an extended example is worked in Section 5. This example also serves to illustrate the power of the type inference system. We draw some final conclusions in Section 6.

2 The I-Soop Language

We begin by defining the I-Soop language, which is roughly call-by-value PCF with records, variants, reference cells, and let-expressions (see Figure 1).

The "vector notation" $\overline{l = v}$ is shorthand for $l_1 = v_1, \ldots, l_k = v_k$ for some k; $\overline{l_i = v_i}$ is shorthand for the same and indicates that i will range over the elements of the vector. The set $B = \{\mathsf{succ}, \mathsf{pred}, \mathsf{is\_zero}, \mathsf{ref}, \mathsf{!}, \mathsf{set}\} \subset \mathit{Var}$ contains the names of built-in primitive functions on numbers and reference cells. Variants are dual to records: the *injection* $l\,\#\,e$ tags the value of e with label l, and the *match* $[\,\overline{l \Rightarrow e}\,]$ (similar to the Standard ML `fn` construct) can be applied to a tagged value to extract it. The booleans and conditional are derived from variants: $\mathtt{true}$ and false are defined as $\mathsf{true}\,\#\,\{\,\}$ and $\mathsf{false}\,\#\,\{\,\}$, respectively, and $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ stands for $[\,\mathsf{true}{\Rightarrow}\lambda\_.\,e_1, \mathsf{false}{\Rightarrow}\lambda\_.\,e_2\,]\ e$; we use $\lambda\_.\,e$ to denote $\lambda x.\,e$ for some x not free in e.

A *store* (ranged over by s) is a finite mapping from variables to values. A *configuration* $\langle s,\ e\rangle$ is a pair of a store and an expression. Computation is defined via a single-step relation $\mapsto_1$ between configurations. A *reduction context* R is an expression with a "hole" $\circ$ in it, into which one may put a subexpression via $R[e]$. Reduction contexts serve to isolate the next step of computation to be performed—it is always in the hole.

Definition 2.1 A *reduction context* is defined inductively in Figure 2.

$$\begin{aligned}
R ::= {} & \circ \mid R\ e \mid v\ R \mid \mathsf{let}\ x = R\ \mathsf{in}\ e \\
& \mid \{\, l_1 = v_1,\ \ldots, l_{i-1} = v_{i-1}, l_i = R, l_{i+1} = e_{i+1},\ \ldots, l_k = e_k \,\} \mid R.l \\
& \mid [\, l_1{\Rightarrow}v_1,\ \ldots, l_{i-1}{\Rightarrow}v_{i-1}, l_i{\Rightarrow}R, l_{i+1}{\Rightarrow}e_{i+1},\ \ldots, l_k{\Rightarrow}e_k \,] \mid l\,\#\,R
\end{aligned}$$

Fig. 2. I-Soop reduction contexts.

Definition 2.2 $\mapsto_1$ is the least relation on configurations satisfying the conditions shown in Figure 3, where

$e[e'/x]$ is the capture-free substitution of e' for x in e,
$[\,x \mapsto v\,]$ is the map defined only on x with result v,
$f||g$ is the functional extension of f by g.

$$\begin{array}{rll}
\langle s,\ R[(\lambda x.\,e)\ v]\rangle \mapsto_1 & \langle s,\ R[e[v/x]]\rangle & \\
\langle s,\ R[\mathsf{let}\ x = v\ \mathsf{in}\ e]\rangle \mapsto_1 & \langle s,\ R[e[v/x]]\rangle & \\
\langle s,\ R[\mathsf{succ}\ n]\rangle \mapsto_1 & \langle s,\ R[n']\rangle & (\text{if } n' = n + 1) \\
\langle s,\ R[\mathsf{pred}\ n]\rangle \mapsto_1 & \langle s,\ R[n']\rangle & (\text{if } n' = n \mathbin{\dot{-}} 1) \\
\langle s,\ R[\mathsf{is\_zero}\ 0]\rangle \mapsto_1 & \langle s,\ R[\mathsf{true}]\rangle & \\
\langle s,\ R[\mathsf{is\_zero}\ n]\rangle \mapsto_1 & \langle s,\ R[\mathsf{false}]\rangle & (\text{if } n \neq 0) \\
\langle s,\ R[\{\ \ldots, l = v,\ \ldots\ \}.l]\rangle \mapsto_1 & \langle s,\ R[v]\rangle & \\
\langle s,\ R[[\ \ldots, l{\Rightarrow}v,\ \ldots\ ]\ (l\,\#\,v')]\rangle \mapsto_1 & \langle s,\ R[v\ v']\rangle & \\
\langle s,\ R[\mathsf{ref}\ v]\rangle \mapsto_1 & \langle s||[\,x \mapsto v\,],\ R[a]\rangle & (x \notin Dom(s) \cup B) \\
\langle s,\ R[!x]\rangle \mapsto_1 & \langle s,\ R[s(x)]\rangle & (x \in Dom(s)) \\
\langle s,\ R[\mathsf{set}\ \{\,\mathsf{cell} = x, \mathsf{val} = v\,\}]\rangle \mapsto_1 & \langle s||[\,x \mapsto v\,],\ R[v]\rangle & (x \in Dom(s))
\end{array}$$

Fig. 3. The single-step computation relation.

Here is a sample execution.

$$\langle \emptyset,\ (\lambda x.\,\mathsf{succ}\ (!(x.\mathsf{field})))\ \{\,\mathsf{field} = \mathsf{ref}\ 5\,\}\rangle$$

$$\mapsto_1 \langle [\,y \mapsto 5\,],\ (\lambda x.\,\mathsf{succ}\ (!(x.\mathsf{field})))\ \{\,\mathsf{field} = \mathsf{y}\,\}\rangle$$

$$\mapsto_1 \langle [\,y \mapsto 5\,],\ \mathsf{succ}\ (!(\{\,\mathsf{field} = \mathsf{y}\,\}.\mathsf{field}))\rangle$$

$$\mapsto_1 \langle [\,y \mapsto 5\,],\ \mathsf{succ}\ (!y)\rangle$$

$$\mapsto_1 \langle [\,y \mapsto 5\,],\ \mathsf{succ}\ 5\rangle$$

$$\mapsto_1 \langle [\,y \mapsto 5\,],\ 6\rangle$$

Lemma 2.3

(i) $\mapsto_1$ *is deterministic: if* $\langle s,\ e\rangle \mapsto_1 \langle s',\ e'\rangle$ *and* $\langle s,\ e\rangle \mapsto_1 \langle s'',\ e''\rangle$, *then there is a uniform renaming of variables in* s' *and* e' *to those in* s'' *and* e'' *respectively.*

(ii) $\mapsto_1$ *is compositional: if* $\langle s,\ e\rangle \mapsto_1 \langle s',\ e'\rangle$, *then* $\langle s,\ R[e]\rangle \mapsto_1 \langle s',\ R[e']\rangle$ *for every reduction context* R. □

3 I-Soop Types

The monomorphic types of the language are

$$\mathit{TyVar} \ni \alpha ::= t \mid u$$

$$\mathit{Typ} \quad \ni \tau ::= \alpha \mid \mathsf{Nat} \mid \tau \rightarrow \tau' \mid \{\,\overline{l:\tau}\,\} \mid [\,\overline{l:\tau}\,] \mid \tau\ \mathsf{ref}$$

where t ranges over the *applicative* type variables $\mathit{AppTyVar} \overset{\mathrm{def}}{=} \{\mathsf{t}_1,\ \mathsf{t}_2,\ \ldots\}$, and u ranges over the *imperative* ones: $\mathit{ImpTyVar} \overset{\mathrm{def}}{=} \{\mathsf{u}_1,\ \mathsf{u}_2,\ \ldots\}$. This division of variables into two classes is similar to that of Standard ML. The set of free type variables in a type τ is $FTV(\tau)$; τ is *imperative* if $FTV(\tau) \subseteq \mathit{ImpTyVar}$.

A *type constraint* is a subtyping assertion between two (monomorphic) types, written $\tau_1 \leq \tau_2$. We will require all sets of constraints used in types and rules to be implicitly closed under obvious laws.

Definition 3.1 (Constraint System) A set of type constraints C is *closed* iff

(i) If $\tau_1 \leq \tau_2 \in C$ and $\tau_2 \leq \tau_3 \in C$, then $\tau_1 \leq \tau_3 \in C$.

(ii) If $\tau_1 \rightarrow \tau_1' \leq \tau_2 \rightarrow \tau_2' \in C$, 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 $\{\overline{l_i}\} \supseteq \{\overline{l_j}\}$, then $\{\overline{\tau_j \leq \tau_j'}\} \subseteq C$.

(iv) If $[\,\overline{l_i:\tau_i}\,] \leq [\,\overline{l_j:\tau_j'}\,] \in C$ and $\{\overline{l_i}\} \subseteq \{\overline{l_j}\}$, then $\{\overline{\tau_i \leq \tau_i'}\} \subseteq C$.

(v) If $\tau_1\ \mathsf{ref} \leq \tau_2\ \mathsf{ref} \in C$, then $\{\tau_1 \leq \tau_2,\ \tau_2 \leq \tau_1\} \subseteq C$.

A closed set of constraints is a *constraint system.*

We let C range over (implicitly closed) constraint systems, and thus will be careful to make sure any new set of constraints we form is closed. The closed union of sets of constraints is denoted by $C_1 \uplus C_2$, an operation that by inspection can be seen to be associative.

Definition 3.2 (Constraint Consistency) A constraint $\tau_1 \leq \tau_2$ is *consistent* if

(i) $\tau_1 \in \mathit{TyVar}$ or $\tau_2 \in \mathit{TyVar}$;

(ii) $\tau_1 = \tau_2 = \mathsf{Nat}$, or $\tau_1 = \tau_1' \ \mathsf{ref}$ and $\tau_2 = \tau_2' \ \mathsf{ref}$, or $\tau_1 = \tau_1' \rightarrow \tau_1''$ and $\tau_2 = \tau_2' \rightarrow \tau_2''$ (for some τ_1', τ_2', τ_1'', τ_2'');

(iii) $\tau_1 = \{\, \overline{l : \tau} \,\}$, $\tau_2 = \{\, \overline{l' : \tau'} \,\}$, and $\{\bar{l}\} \supseteq \{\bar{l'}\}$; or

(iv) $\tau_1 = [\, \overline{l : \tau} \,]$, $\tau_2 = [\, \overline{l' : \tau'} \,]$, and $\{\bar{l}\} \subseteq \{\bar{l'}\}$.

Otherwise a constraint is *inconsistent.*

For example, $\mathsf{Nat} \leq \mathsf{t} \rightarrow \mathsf{Nat}$ and $\mathsf{t} \ \mathsf{ref} \leq \{\, \mathsf{m} : \mathsf{Nat} \,\}$ are inconsistent constraints, while $\mathsf{t} \leq \mathsf{t} \rightarrow \mathsf{Nat}$, $\mathsf{t} \leq \mathsf{u}$, and $\mathsf{u} \leq \mathsf{Nat}$ are each consistent. A constraint system is consistent if all the constraints in the system are consistent. The rules will require all constraint systems to implicitly be consistent.

The type system assigns I-Soop expressions *rc types* of the form

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

to indicate an expression of type τ which is constrained by the constraints in C. Since the rules implicitly require C to be consistent, it makes sense to view κ as a type and to write C on the right side of the turnstile as part of the type.

We define the following notion of subtyping on rc types.

Definition 3.3 (Subtyping rc Types) $\tau \setminus C \leq \tau' \setminus C'$ provided that C' is consistent and $C \uplus \{\tau \leq \tau'\} \subseteq C'$.

Stronger notions of subtyping could be defined, but for our purposes this definition suffices. The type schemes σ are as follows.

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

Note that since $\kappa = \tau \setminus C$ can contain an arbitrary collection of constraints C, shallow F-bounded polymorphic types are a special case of these polymorphic rc types.

3.1 I-Soop Typing Rules

Before giving the rules we describe notation used in the rules. Notation used in sequent judgements includes the following. A *type environment* A is a mapping from variables to type schemes; we use the more intuitive notation $[\, x : \sigma \,]$ instead of $[\, x \mapsto \sigma \,]$. 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, all constraint sets C appearing in the rules implicitly *must* be consistent); we occasionally

$$\text{(Sub)}\ \frac{A \vdash e : \kappa, \quad \kappa \leq \kappa'}{A \vdash e : \kappa'} \qquad \text{(Num)}\ \frac{}{A \vdash n : \mathsf{Nat} \setminus \emptyset}$$

$$\text{(Abs)}\frac{A \| [\, x : \tau \,] \vdash e : \tau' \setminus C}{A \vdash \lambda x.\, e : \tau \to \tau' \setminus C} \qquad \text{(App)}\ \frac{A \vdash e_1 : \tau \to \tau' \setminus C_1, \quad e_2 : \tau \setminus C_2}{A \vdash e_1\ e_2 : \tau' \setminus C_1 \uplus C_2}$$

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

$$\text{(Sel)}\ \frac{A \vdash e : \{\, l : \tau \,\} \setminus C}{A \vdash e.l : \tau \setminus C} \qquad \text{(Record)}\ \frac{A \vdash \overline{e_i : \tau_i \setminus C_i}}{A \vdash \{\, \overline{l_i = e_i} \,\} : \{\, \overline{l_i : \tau_i} \,\} \setminus \biguplus_i C_i}$$

$$\text{(Inj)}\ \frac{A \vdash e : \tau \setminus C}{A \vdash l \,\#\, e : [\, l : \tau \,] \setminus C} \qquad \text{(Match)}\ \frac{A \vdash \overline{e_i : \tau_i \to \tau \setminus C_i}}{A \vdash [\, \overline{l_i \Rightarrow e_i} \,] : [\, \overline{l_i : \tau_i} \,] \to \tau \setminus \biguplus_i C_i}$$

$$\text{(Let)}\ \frac{A \vdash e : \tau \setminus C, \quad A \| [\, x : \forall \overline{\alpha}.\, \tau \setminus C \,] \vdash e' : \tau' \setminus C'}{A \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau' \setminus C \uplus C'}$$

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

Fig. 4. Typing rules of I-Soop.

may write $A \vdash e_1 : \tau_1 \setminus C_1,\ e_2 : \tau_2 \setminus C_2,\ \ldots$ to indicate several type judgements provable in the same environment. Programs are type-checked in the initial environment A_0 assigning the following type schemes to the built-ins:

$$\begin{aligned} A_0 = [\, & \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat},\ \mathsf{pred} : \mathsf{Nat} \to \mathsf{Nat},\ \mathsf{is\_zero} : \mathsf{Nat} \to \mathsf{Bool}, \\ & \mathsf{ref} : \forall \mathsf{u}.\ \mathsf{u} \to \mathsf{u\ ref},\ ! : \forall \mathsf{t}.\ \mathsf{t\ ref} \to \mathsf{t},\ \mathsf{set} : \forall \mathsf{t}.\ \{\, \mathsf{cell} : \mathsf{t\ ref}, \mathsf{val} : \mathsf{t} \,\} \to \mathsf{t} \,] \end{aligned}$$

where Bool stands for the type $[\mathsf{true}{:}\{\,\}, \mathsf{false}{:}\{\,\}]$. A *substitution* on $\{\overline{\alpha}\}$ is a map $\Psi \in \mathit{TyVar} \to \mathit{Typ}$ which is the identity on $\mathit{TyVar} \setminus \{\overline{\alpha}\}$ and maps $\mathit{ImpTyVar}$ to imperative types; a *renaming* Φ of $\{\overline{\alpha}\}$ is a substitution on $\{\overline{\alpha}\}$ with $\mathit{codom}(\Phi) \subseteq \mathit{TyVar}$. An expression is *expansive* if and only if it is not a value; following Tofte [26] we form type schemes by making the sets of type variables we generalize over dependent on the expansiveness of the expression. The definitions of these sets are

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

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

The typing rules for I-Soop are given in Figure 4. Most of the rules have obvious relation to those of standard systems with subtyping and records; as in Tofte's system [26], the typing of ref introduces imperative types. The main difference is the addition of constraints as part of types, the associated subsumption rule on these types, and the way consistent constraints accumulate from the leaves to the root of a typing proof. It is important to observe that consistency of constraints is implicitly enforced by each rule. Other presentations of constrained type systems [18,3,13] 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. For this reason they write C on the left of the turnstile, and perform some top-level consistency check before a proved typing is "true." Since constraints are never inconsistent in our rules we have no hypothetical component and constraints are thus written on the right-hand side of the turnstile.

Some justification is required for the (Let) rule, in which the constraint system of the let expression contains not only the constraints in C', necessary for typing its body, but also those in C, accumulated for the type of the bound variable. Leaving the latter constraints out (as [3] do, but corrected in [2]) results in a system unsound with respect to the standard call-by-value semantics of the let expression; C may contain constraints on type variables free in the environment, and their omission may lead to accepting programs which get stuck while evaluating the expression assigned to the bound variable. As an example, consider the expression

$$(\lambda \mathsf{x}.\, \mathsf{let\ y = !x\ in\ succ\ x})\ 5$$

By rules (PVar), (Var), (Sub), and (App) the constraint system C of the rc type of !x contains $\tau \leq \tau'$ ref for some type τ', where τ is the type associated with x by the rule (Abs). This constraint will lead to inconsistency when combined with the constraint Nat $\leq \tau$ at the outermost rule of the typing proof, (App). If it were omitted from the constraint system of the let, the other constraint on τ, namely $\tau \leq$ Nat from the body succ x, would not cause an inconsistency, and the program would type-check; however its execution obviously leads to the stuck state $\langle \emptyset$, let y = !5 in succ 5$\rangle$.

While the type language does not have recursive types, λx. x x can be given the rc type $\mathsf{t}_1 \rightarrow \mathsf{t}_2 \setminus \{\mathsf{t}_1 \leq \mathsf{t}_1 \rightarrow \mathsf{t}_2\}$. We do not have a "bottom" type, but its positive occurrences may be simulated by an unconstrained type variable, e.g. (λx. x x) λx. x x has the rc type

$$\mathsf{t}_2 \setminus \{\mathsf{t}_1 \rightarrow \mathsf{t}_2 \leq \mathsf{t}_1,\ \mathsf{t}_1 \leq \mathsf{t}_1 \rightarrow \mathsf{t}_2\}$$

An unconstrained variable can also be used instead of a "top" type in negative positions. Positive occurrences of "top" may be simulated by overconstraining from below:

$$A_0 \vdash \mathsf{if\ true\ then\ 5\ else\ \{\,\}} : \mathsf{t} \setminus \{\mathsf{Nat} \leq \mathsf{t},\ \{\,\} \leq \mathsf{t}\}$$

This constraint system is consistent. Note that not all typable programs are of this particular "top" type, but they are provably of type $t \setminus \{\mathsf{Nat} \leq t,\ \{\,\} \leq$

$t\}\uplus C$ for some C and fresh t by a single use of (Sub). Similarly overconstraining from above achieves the effect of "bottom" in negative positions.

4 Subject Reduction, Soundness, and Type Inference

We prove soundness of the type system by demonstrating a subject reduction property. First we strengthen the (Let) rule of the system to:

$$(\mathsf{Let})\ \frac{A \vdash e : \tau \setminus C, \quad A||[x : \forall\overline{\alpha}.\, \tau \setminus C\,] \vdash e' : \tau' \setminus C', \quad \Phi \text{ is a renaming of } \{\overline{\alpha}\}}{A \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau' \setminus \Phi C \uplus C'}$$

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

Obviously, any use of the original (Let) rule can be trivially transformed into a use of the stronger rule, by choosing Φ to be the identity renaming. This renaming does not add any power to the typing system; any program that is typable with the stronger (Let) rule is also typable with the original; it is introduced only to avoid certain technical complications which arise during reductions within a let expression.

Next, we extend the notion of typing to configurations:

Definition 4.1 $A \vdash \langle s,\, e\rangle : \tau \setminus C$ if and only if

1. $A \vdash e : \tau \setminus C$;
2. $\mathit{Dom}(A) = \mathit{Dom}(A_0) \cup \mathit{Dom}(s)$, $\mathit{Dom}(A_0) \cap \mathit{Dom}(s) = \emptyset$, and A agrees with A_0 on $\mathit{Dom}(A_0)$;
3. for each $x \in \mathit{Dom}(s)$ we have $A(x) = \tau_x\ \mathsf{ref}$ and $A \vdash s(x) : \tau_x \setminus C_x$ for some τ_x and $C_x \subseteq C$.

Theorem 4.2 (Subject Reduction) *If $A \vdash \langle s,\, e\rangle : \kappa$, then either $e \in \mathit{Val}$ or else $\langle s,\, e\rangle \mapsto_1 \langle s',\, e'\rangle$ and there exists an environment A' such that $A' \vdash \langle s',\, e'\rangle : \kappa$.* □

We present only a sketch of the proof in this abbreviated version. The proof proceeds in the standard fashion: given a configuration and a proof of its typability, perform one step of computation and transform the original typing proof into a proof for the new configuration. The interaction between let-polymorphism and reference cells is known to cause significant difficulty [26]; our approach to this problem derives from [27], avoiding Tofte's complex greatest fixed-point construction.

The differences between our proof and that of [27] result from the constraint systems of rc types and polymorphic rc types. Each step of computation is accompanied by a proof transformation that pushes constraints present near the top of the proof tree towards the leaves. The complications of the proof arise when these constraints are pushed through uses of the (Let) rule;

demonstrating that the type generalizations performed in the initial application of the rule remain valid is non-trivial.

This pushing of constraints from the root of the typing proof towards the leaves during reduction can be considered a lazy approach to proof canonicalization. An alternative approach would be to regularize the initial typing proof of a program to canonical form by pushing all of the constraints present at the root to the leaves before performing any computation. This would result in a more straightforward subject reduction proof, at the expense of a more complicated proof canonicalization lemma.

The soundness of the type system is a corollary of the Subject Reduction theorem:

Theorem 4.3 (Soundness) *If $A_0 \vdash e : \kappa$, then either e diverges, or e computes to a value.*

Proof. By induction on the length of computation, using Theorem 4.2. □

Note we have thus proved soundness of the constrained type system without ever having shown the systems of constraints have a solution.

4.1 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. The strategy we take to reach this desired outcome is the following.

1. Define a new set of rules (the *inference rules*) for which typing derivations are deterministic.
2. Prove the inference rules are equivalent in strength to the *general rules* we had been using previously.

The inference rules appear in Figure 5.

Theorem 4.4 *For all terms e and environments A, it is decidable whether there exists a κ such that $A \vdash_{\text{inf}} e : \kappa$.*

Proof (Sketch) By inspection of the rules, there is only one 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 (Abs) and (PVar). We thus choose *α-normal* proofs that use fresh variables in every place possible. If a proof exists, there clearly must then be a corresponding α-normal proof. For expression e the α-normal 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.

$$(\mathsf{Abs})\quad \frac{A\|[x\,:\,t] \vdash_{\text{inf}} e : \tau \setminus C}{A \vdash_{\text{inf}} \lambda x.\, e : t \to \tau \setminus C} \qquad (\mathsf{Var})\quad \frac{A(x) = \tau}{A \vdash_{\text{inf}} x : \tau \setminus \emptyset}$$

$$(\mathsf{Sel})\quad \frac{A \vdash_{\text{inf}} e : \tau \setminus C}{A \vdash_{\text{inf}} e.l : t \setminus C \uplus \{\tau \leq \{\, l : t\,\}\}} \qquad (\mathsf{Inj})\quad \frac{A \vdash_{\text{inf}} e : \tau \setminus C}{A \vdash_{\text{inf}} l\,\#\,e : [\,l : \tau\,] \setminus C}$$

$$(\mathsf{Record})\quad \frac{A \vdash_{\text{inf}} \overline{e_i : \tau_i \setminus C_i}}{A \vdash_{\text{inf}} \{\,\overline{l_i = e_i}\,\} : \{\,\overline{l_i : \tau_i}\,\} \setminus \biguplus_i C_i} \qquad (\mathsf{Num})\quad \frac{}{A \vdash_{\text{inf}} n : \mathsf{Nat} \setminus \emptyset}$$

$$(\mathsf{Match})\quad \frac{A \vdash_{\text{inf}} \overline{e_i : \tau_i \setminus C_i}}{A \vdash_{\text{inf}} [\,\overline{l_i \Rightarrow e_i}\,] : [\,\overline{l_i : t_i}\,] \to t \setminus \biguplus_i (C_i \uplus \{\tau_i \leq t_i \to t\})}$$

$$(\mathsf{App})\quad \frac{A \vdash_{\text{inf}} e_1 : \tau_1 \setminus C_1, \quad e_2 : \tau_2 \setminus C_2}{A \vdash_{\text{inf}} e_1\ e_2 : t \setminus C_1 \uplus C_2 \uplus \{\tau_1 \leq \tau_2 \to t\}}$$

$$(\mathsf{PVar})\quad \frac{A(x) = \forall \overline{\alpha}.\, \kappa, \quad \Phi \text{ is a renaming of } \{\overline{\alpha}\}}{A \vdash_{\text{inf}} x : \Phi\kappa}$$

$$(\mathsf{Let})\quad \frac{A \vdash_{\text{inf}} e : \tau \setminus C, \quad A\|[x\,:\,\forall\overline{\alpha}.\,\tau \setminus C\,] \vdash_{\text{inf}} e' : \tau' \setminus C'}{A \vdash_{\text{inf}} \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau' \setminus C \uplus C'}$$

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

Fig. 5. Type inference rules of I-Soop.

Theorem 4.5 (Completeness of Type Inference) *Given an environment A and an expression e, the typing judgement $A \vdash e : \kappa$ is provable for some κ if and only if $A \vdash_{\text{inf}} e : \kappa'$ is provable for some κ'.*

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

Conversely, typing proofs in the general set of rules may be transformed into ones using only the inference ones in a two-step process. First, the proof is transformed into *pre-inference* form, in which each rule used one of the inference rules, or possibly (Sub). In the process, certain types τ used in the proof (such as in the conclusion of rules (Var), (Sel), (App), and the like) are replaced by fresh type variables t; the corresponding type constraints $\{\tau \leq t, t \leq \tau\}$ are added to the constraint system and bubbled to the top. Similarly, each assumption $x : \tau$ is replaced with an assumption of the form $x : t$, together with the constraints $\{\tau \leq t, t \leq \tau\}$, for some fresh t. The result is a larger set of constraints mentioning these new type variables. Demonstrating the consistency of these richer constraint systems as these new constraints propagate to the root of the proof is non-trivial.

Second, the proof is reworked again, eliminating uses of rule (Sub) inductively. This transformation takes a pre-inference proof of $A \vdash e : \tau \setminus C$ and produces an inference proof of $A \vdash e : \tau' \setminus C'$, where $C' \subseteq C$ and either $\tau = \tau'$ or $\{\tau' \leq \tau\} \subseteq C$. This is possible because the antecedents of each inference rule are simply of the form $A \vdash e : \tau \setminus C$; the type τ need not be in any special form for the rule to be applicable. Essentially, this means that a use of (Sub) followed by another rule can be exchanged, thus moving the subsumptions to the root of the proof where they can be eliminated. □

Thus from Theorems 4.5 and 4.4 we may conclude that every program typable under the general rules has a type inferred by the type inference algorithm. Note we establish no principal typing property. The typing produced by the inference algorithm is indeed "minimal" in an intuitive sense, but it is not formally minimal since our definition of $\kappa \leq \kappa'$ is weak: $t \rightarrow \mathsf{Nat} \setminus \{t \leq \mathsf{Nat}\}$ is not a subtype of $\mathsf{Nat} \rightarrow \mathsf{Nat} \setminus \emptyset$, even though any term that can be given the former type can also be given the latter. We leave the question of principal typings for future study, since completeness is ultimately all the programmer desires.

5 Applications to OOP

We now illustrate how this type inference algorithm is useful for typing object-oriented programs, the main motivation for our work. We show its utility in class-based OOP; we expect it also applies to delegation-style OOP but that topic is beyond the scope of this paper. The basic OOP concepts we wish to incorporate include standard notions of object, method, instance variable, class, inheritance, method/instance hiding, and object lifting[3]. The more advanced notions we wish to account for include polymorphism, multiple inheritance and binary methods. Without binary methods (in general, methods that take objects as parameters or return objects as values), the object typing problem is not overly difficult: objects may be interpreted as records of functions (methods) and cells (instance variables), inheritance is subtyping, and object lifting is accomplished by a subsumption rule. As we show, typing becomes considerably more difficult in the presence of binary methods [9].

The ideal way to show applicability to OOP would be to define a complete OOP language, types, and inference algorithm; this is beyond the scope of this paper, however. Instead, we will show how a collection of simple macros allow OOP to be embedded into I-Soop.

The basic idea of the representation is to interpret classes as functions on records $\lambda s.\{\ \ldots\ \}$ where s is the "self"; new then takes the fixed point of a class to produce an object, in the form of a record (see [14]). We cannot quite use this encoding. First, it is difficult to create fixed points which are records in a call-by-value language. Second, when taking a fixed point via a Y-combinator, the semantics entails re-evaluating the record with each recursive access, and thus erroneously re-initialize any instance variables. In previous work [12] we avoided these problems by using a memory-based fixed point. Unfortunately

[3] Also called implicit object coercion or object subsumption.

this encoding will not work here as the use of reference cells to form the fixed point will infer imperative polymorphic types for objects. We thus opt for an encoding using a Y-combinator with an initial instance variable allocation phase. In a more complete treatment of this topic a limited form of memory-based fixed point such as the single-assignment reference (SAR) of [10] could be used. We ignore the issue of information hiding in this presentation, though it is not difficult to incorporate.

Definition 5.1 The object syntax is defined by the macros given in Figure 6, where $\mathsf{Y} \overset{\text{def}}{=} \lambda \mathsf{y}.(\lambda \mathsf{x}.\mathsf{x}\ \mathsf{x})\ \lambda \mathsf{x}.\lambda \mathsf{z}.\mathsf{y}\ (\mathsf{x}\ \mathsf{x})\ \mathsf{z}$ is a call-by-value Y-combinator, and $\Omega \overset{\text{def}}{=} \mathsf{Y}\ (\lambda \mathsf{x}.\mathsf{x})$.

Note that the class macro binds occurrences of s free in the e''_k, and those of u_i free in e'_j and e''_k.

$$\begin{array}{lrcl}
\text{(class)} & \multicolumn{3}{l}{\mathsf{class}\ s\ \mathsf{super}\ \overline{u_i\ \mathsf{of}\ e_i}\ \mathsf{inst}\ \overline{x_j = e'_j}\ \mathsf{meth}\ \overline{m_k = e''_k}} \\
 & & = & \\
 & \multicolumn{3}{l}{\lambda\_.\,\mathsf{let}\ \overline{u_i^0 = e_i\ \{\,\}}\ \mathsf{in}\ \mathsf{let}\ \overline{u_i = u_i^0\ (\lambda x.\Omega)}\ \mathsf{in}} \\
 & \multicolumn{3}{l}{\quad \mathsf{let}\ y = \overline{\{\, x_j = \mathsf{ref}\ e'_j \,\}}\ \mathsf{in}\ \lambda s.\,\mathsf{let}\ \overline{u_i = u_i^0\ (s)}\ \mathsf{in}} \\
 & \multicolumn{3}{l}{\quad \lambda\_.\{\,\mathsf{inst} = y,\ \mathsf{meth} = \{\,\overline{m_k = e''_k}\,\}} \\
\text{(new)} & \mathsf{new} & = & \lambda \mathsf{x}.\,\mathsf{Y}(\mathsf{x}\ \{\,\}) \\
\text{(message send)} & e \mathrel{\texttt{<-}} m & = & (e\ \{\,\}).\mathsf{meth}.m \\
\text{(instance read)} & e{\cdot}x & = & !((e\ \{\,\}).\mathsf{inst}.x) \\
\text{(instance write)} & e_1{\cdot}x := e_2 & = & \mathsf{set}\ \{\,\mathsf{cell} = (e_1\ \{\,\}).\mathsf{inst}.x, \mathsf{val} = e_2\,\}
\end{array}$$

Fig. 6. Macros for object syntax.

We illustrate the typing problems involved with binary methods through an example of a GcdNum class that has a binary method gcd that takes another GcdNum and recursively computes the GCD of itself and the other GcdNum. In order to keep the example very simple we assume the instance variable containing the actual number, val, is publicly accessible, and that GcdNum defines no other methods. ZGcdNum is a subclass of GcdNum with an additional unimportant method zero. Here mod is taken to be a function that computes the modulus of two numbers.

```
let GcdNum = class s super
    inst
        val = 0
    meth
        gcd = λnum. if is_zero (s·val) then s
                    else if is_zero (mod (num·val) (s·val)) then s
                    else s·val := mod (s·val) (num·val); num <- gcd s
```

The gcd method takes another GcdNum object, num, as argument. Because num is of the same type as the type of objects of the class we are currently defining, expressing the type of the gcd method will require some self-referentiality.

We first consider appropriate types for the inheritance-is-subtyping paradigm. This is known to have serious limitations [9], but is nonetheless frequently found in commercial OOP languages. In this paradigm we give GcdNum the type

$$\mathsf{GcdNum} : \mathsf{GcdType} \to \mathsf{GcdType}, \text{ where}$$

$$\mathsf{GcdType} = \mu \mathsf{t}.\,(\{\,\} \to \{\, \mathsf{val} : \mathsf{Nat\ ref}, \mathsf{gcd} : \mathsf{t} \to \mathsf{t}\,\})$$

Note that μ is the usual recursive type constructor. We use it instead of the I-Soop encoding of recursive types using recursive constraints. new GcdNum then returns an object of type GcdType. Without inheritance this type is perfectly adequate. We now look at the adequacy of this type with inheritance. We extend our example by defining ZGcdNum, a subclass of GcdNum that also includes a method that tests for zero.

```
let ZGcdNum = class s
      super
          u of GcdNum
     inst
          val = u·val
      meth
          gcd = u <- gcd,
          zero = λ_. is_zero (s·val)
```

In this case we did not override the gcd method; instead, we inherited it from GcdNum, denoted here by the superclass variable u (in this encoding we explicitly state the superclass of each inherited method). Using the inheritance-is-subtyping paradigm, the inherited instance variables and methods must have the same types as in the superclass since these types are fixed. Thus, the type of ZGcdNum must be

$$\mathsf{ZGcdNum} : \mathsf{ZGcdType} \to \mathsf{ZGcdType}, \text{ where}$$

$$\mathsf{ZGcdType} =$$

$$\mu \mathsf{t}.\,(\{\,\} \to \{\, \mathsf{val} : \mathsf{Nat\ ref}, \mathsf{gcd} : \mathsf{GcdType} \to \mathsf{GcdType}, \mathsf{zero} : \{\,\} \to \mathsf{Bool}\,\})$$

Note the gcd method still operates on GcdType, not ZGcdType. Thus if gcd were overridden in ZGcdNum with a function that used num's zero method, this typing would fail, an undesirable fact. Another problem with this typing is illustrated in the following additional code.

```
let zgnum = new ZGcdNum in (zgnum <- gcd zgnum) <- zero { }
```

The gcd method type is not parametric in the type of the object given to it. Thus it will accept an object of ZGcdType as an argument since by subtyping ZGcdType $\leq$ GcdType, but the result returned is only of GcdType, and thus is not known to have a zero method. The above code will thus not type-check, even though it executes without error.

An alternative typing is needed. Since we inherit from GcdNum, the ZGcdNum objects that eventually are created will have more methods than just gcd. To capture this, we must take a *parametric* or *open-ended* view of the self-type in GcdNum's type. The parametricity we desire in GcdNum is that t should be any subclass with at least gcd and val, and furthermore that gcd parametrically maps t to t. To express the open-ended view as a type, F-bounded quantification is used as follows.

$$\mathsf{GcdNum} : \forall \mathsf{t} \leq \mathsf{GcdTypeF(t)}.\ \mathsf{t} \rightarrow \mathsf{GcdTypeF(t)},\ \text{where}$$

$$\mathsf{GcdTypeF(t)} = \{\,\} \rightarrow \{\, \mathsf{val} : \mathsf{Nat\ ref}, \mathsf{gcd} : \mathsf{t} \rightarrow \mathsf{t} \,\}$$

ZGcdNum may then be typed as

$$\mathsf{ZGcdNum} : \forall \mathsf{t} \leq \mathsf{ZGcdTypeF(t)}.\ \mathsf{t} \rightarrow \mathsf{ZGcdTypeF(t)},\ \text{where}$$

$$\mathsf{ZGcdTypeF(t)} = \{\,\} \rightarrow \{\, \mathsf{val} : \mathsf{Nat\ ref}, \mathsf{gcd} : \mathsf{t} \rightarrow \mathsf{t}, \mathsf{zero} : \{\,\} \rightarrow \mathsf{Bool} \,\},$$

giving zgnum the type $\mu\mathsf{t}.\,\mathsf{ZGcdTypeF(t)}$. Thus the above code type-checks. In addition, it would have been possible to override gcd in ZGcdNum, impossible in the simple recursive-types view.

The F-bounded typing has a drawback, however. ZGcdNum objects can no longer be lifted to be GcdNum objects (since their types are recursive types with t occurring negatively), and thus the following code will not type-check.

```
let gnum = new GcdNum in
   let zgnum = new ZGcdNum in
      gnum <- gcd zgnum
```

Note that the recursive typing *would* allow this code to type-check.

So, both the F-bounded interpretation of inheritance and the recursive types interpretation fail to typecheck certain typable programs. Our type inference algorithm, however, infers types that will allow both of the above varieties of message send to be typed in a single program.

5.1 Types inferred in I-Soop

To simplify the presentation, we will ignore the instance variable val in the example. We will also simplify the translation scheme to reflect this, by eliminating the first line from the macro expansion of class and replacing u_i^0 by e, and defining new as Y.

First consider the types inferred for the classes GcdNum and ZGcdNum. The simplified translations are

```
let GcdNum =
         λs. λ_. { gcd = λnum. if — then s
                               else if — then s
                               else (num { }).gcd s }
in let ZGcdNum =
         λs. let u = GcdNum (s) in
```

$$\begin{array}{l} \lambda\_.\,\{\,\mathsf{gcd} = (\mathsf{u}\ \{\,\}).\mathsf{gcd}, \\ \qquad \mathsf{zero} = \lambda\_.\ \mathsf{is\_zero}\ \text{—}\ \ \} \end{array}$$

We first sketch how the inference system of rules, $\vdash_{\text{inf}}$, infers GcdNum's type. These rules are deterministic modulo α-variants so proof construction is mechanical. Starting from the leaves and using rules (Record), (App), and (Sel) in turn we obtain

$$A_0||[\,\mathsf{s} : \mathsf{t}_1,\ \_ : \mathsf{t}_a,\ \mathsf{num} : \mathsf{t}_b\,] \vdash_{\text{inf}} (\mathsf{num}\ \{\,\}).\mathsf{gcd} : \mathsf{t}_d \setminus C_1,$$

$$\text{where } C_1 = \{\mathsf{t}_b \leq \{\,\}\rightarrow\mathsf{t}_c,\ \mathsf{t}_c \leq \{\,\mathsf{gcd} : \mathsf{t}_d\,\}\}$$

Next, using (App),

$$A_0||[\,\mathsf{s} : \mathsf{t}_1,\ \_ : \mathsf{t}_a,\ \mathsf{num} : \mathsf{t}_b\,] \vdash_{\text{inf}} (\mathsf{num}\ \{\,\}).\mathsf{gcd}\ \mathsf{s} : \mathsf{t}_e \setminus C_2,$$

$$\text{where } C_2 = \{\mathsf{t}_b \leq \{\,\}\rightarrow\mathsf{t}_c,\ \mathsf{t}_c \leq \{\,\mathsf{gcd} : \mathsf{t}_d\,\},\ \mathsf{t}_d \leq \mathsf{t}_1\rightarrow\mathsf{t}_e\}$$

Next, expanding the conditional and using (Abs) and (Match) twice,

$$A_0||[\,\mathsf{s} : \mathsf{t}_1,\ \_ : \mathsf{t}_a\,] \vdash_{\text{inf}} \lambda\mathsf{num}.\ \ldots : \mathsf{t}_b\rightarrow\mathsf{t}_2 \setminus C_2 \uplus \{\mathsf{t}_e \leq \mathsf{t}_2,\ \mathsf{t}_1 \leq \mathsf{t}_2\}$$

Finally, by (Record) and (Abs) twice,

$$A_0 \vdash_{\text{inf}} \mathsf{GcdNum} : \mathsf{t}_1\rightarrow\mathsf{t}_a\rightarrow\{\,\mathsf{gcd} : \mathsf{t}_b\rightarrow\mathsf{t}_2\,\} \setminus C_2 \uplus \{\mathsf{t}_e \leq \mathsf{t}_2,\ \mathsf{t}_1 \leq \mathsf{t}_2\}$$

This is the type inferred by the inference rule system. An actual implemented type inference algorithm would automatically perform a number of simplifications on this type that do not change the meaning. Here we present these simplifications informally by giving typings deduced in the general rules that are simplified forms of the inferred types. For GcdNum, t_a is unconstrained so it may be replaced by $\{\,\}$ by subsumption. t_b has only one positive occurrence in the type, so it may be replaced with its upper bound. t_c, t_d and t_e may also each be replaced. The following type may then be deduced for GcdNum in the general rules:

$$\forall\mathsf{t}_1,\mathsf{t}_2.\ \mathsf{t}_1\rightarrow\{\,\}\rightarrow\{\,\mathsf{gcd} : (\{\,\}\rightarrow\{\,\mathsf{gcd} : \mathsf{t}_1\rightarrow\mathsf{t}_2\,\})\rightarrow\mathsf{t}_2\,\}\,\} \setminus \{\mathsf{t}_1 \leq \mathsf{t}_2\}$$

Hereafter we present the simplified forms of types only. An actual implemented type inference algorithm would automatically perform these simplifications. For ZGcdNum, the (simplified) inferred type is

$$\forall\mathsf{t}_1,\mathsf{t}_2.\ \mathsf{t}_1\rightarrow\{\,\}\rightarrow\{\,\mathsf{gcd}\!: (\{\,\}\rightarrow\{\,\mathsf{gcd}\!: \mathsf{t}_1\rightarrow\mathsf{t}_2\,\})\rightarrow\mathsf{t}_2, \mathsf{zero}\!: \{\,\}\rightarrow\mathsf{Bool}\,\}\setminus\{\mathsf{t}_1 \leq \mathsf{t}_2\}$$

Contrast these types with the F-bounded type given GcdNum in the "open-self" encoding above. Observe that the parameter num is an object with a gcd method. Since that is the only method of num that is used, no more fields are required in the inferred type. Contrast that with the F-bounded case where num has all methods of GcdNum: the open-endedness here is more precise, each method that is passed the "self" requires that self to only have the methods actually used. Note also that this is not even an F-bounded type, the constraint $\mathsf{t}_1 \leq \mathsf{t}_2$ is not recursive. Recursive constraints may not arise in classes, since the knot has not been tied yet.

Consider now the object types. gnum and zgnum have the following (simplified) types inferred:

$$\begin{array}{l}
\mathsf{gnum} : \forall \mathsf{t}_1, \mathsf{t}_2.\ \mathsf{t}_1 \setminus \{\{\,\}\rightarrow\{\,\mathsf{gcd} : (\{\,\}\rightarrow\{\,\mathsf{gcd} : \mathsf{t}_1\rightarrow\mathsf{t}_2\,\})\rightarrow\mathsf{t}_2\,\} \leq \mathsf{t}_1 \leq \mathsf{t}_2\}, \\
\mathsf{zgnum} : \forall \mathsf{t}_1, \mathsf{t}_2.\ \mathsf{t}_1 \setminus \{\{\,\}\rightarrow\{\,\mathsf{gcd} : (\{\,\}\rightarrow\{\,\mathsf{gcd} : \mathsf{t}_1\rightarrow\mathsf{t}_2\,\})\rightarrow\mathsf{t}_2, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{zero} : \{\,\}\rightarrow\mathsf{Bool}\,\} \leq \mathsf{t}_1 \leq \mathsf{t}_2\}
\end{array}$$

It is difficult to explain precisely what these types denote, except to say they are definitely not the recursive types used in both encodings for objects above.

The message sends from the example have the following constrained types.

$$\begin{array}{l}
\mathsf{zgnum} \mathrel{\mathsf{<-}} \mathsf{gcd}\ \mathsf{gnum} : \mathsf{t}_2 \setminus \{ \\
\quad \{\,\}\rightarrow\{\,\mathsf{gcd} : (\{\,\}\rightarrow\{\,\mathsf{gcd} : \mathsf{t}_{1a}\rightarrow\mathsf{t}_2\,\})\rightarrow\mathsf{t}_2, \mathsf{zero} : \{\,\}\rightarrow\mathsf{Bool}\,\} \leq \mathsf{t}_{1a}, \\
\quad \mathsf{t}_{1a} \leq \{\,\}\rightarrow\{\,\mathsf{gcd} : \mathsf{t}_{1b}\rightarrow\mathsf{t}_2\,\},\ \mathsf{t}_{1b} \leq \{\,\}\rightarrow\{\,\mathsf{gcd} : \mathsf{t}_{1a}\rightarrow\mathsf{t}_2\,\} \\
\quad \{\,\}\rightarrow\{\,\mathsf{gcd} : (\{\,\}\rightarrow\{\,\mathsf{gcd} : \mathsf{t}_{1b}\rightarrow\mathsf{t}_2\,\})\rightarrow\mathsf{t}_2\,\} \leq \mathsf{t}_{1b},\ \mathsf{t}_{1a} \leq \mathsf{t}_2, \mathsf{t}_{1b} \leq \mathsf{t}_2\}, \\
\mathsf{zgnum} \mathrel{\mathsf{<-}} \mathsf{gcd}\ \mathsf{zgnum} : \mathsf{t}'_2 \setminus \{ \\
\quad \{\,\}\rightarrow\{\,\mathsf{gcd} : (\{\,\}\rightarrow\{\,\mathsf{gcd} : \mathsf{t}'_{1a}\rightarrow\mathsf{t}'_2\,\})\rightarrow\mathsf{t}'_2, \mathsf{zero} : \{\,\}\rightarrow\mathsf{Bool}\,\} \leq \mathsf{t}'_{1a}, \\
\quad \mathsf{t}'_{1a} \leq \{\,\}\rightarrow\{\,\mathsf{gcd} : \mathsf{t}'_{1b}\rightarrow\mathsf{t}'_2\,\},\ \mathsf{t}'_{1b} \leq \{\,\}\rightarrow\{\,\mathsf{gcd} : \mathsf{t}'_{1a}\rightarrow\mathsf{t}'_2\,\}, \\
\quad \{\,\}\rightarrow\{\,\mathsf{gcd} : (\{\,\}\rightarrow\{\,\mathsf{gcd} : \mathsf{t}'_{1b}\rightarrow\mathsf{t}'_2\,\})\rightarrow\mathsf{t}'_2, \mathsf{zero} : \{\,\}\rightarrow\mathsf{Bool}\,\} \leq \mathsf{t}'_{1b}, \\
\quad \mathsf{t}'_{1a} \leq \mathsf{t}'_2, \mathsf{t}'_{1b} \leq \mathsf{t}'_2\}
\end{array}$$

Note the function upper bounds of t_{1a}, t_{1b}, t'_{1a} and t'_{1b} can be proved to never be used; a more complete set of simplification transformations would justify their removal. Each use of gnum and zgnum gives rise to fresh variables by the (PVar) rule; if these objects were not let-polymorphic, the two message sends above would share type variables and generality would be lost. Observe there are no contradictions in the constraint systems of either of these message sends. Also note the result type t_2 is in effect the union of t_{1a} and t_{1b} since it is an upper bound of these two types. This corresponds to the fact that the result of gcd could be either a gnum or a zgnum. Consider sending a zero message to the result of the second message send, $(\mathsf{zgnum} \mathrel{\mathsf{<-}} \mathsf{gcd}\ \mathsf{zgnum}) \mathrel{\mathsf{<-}} \mathsf{zero}\ \{\,\}$. The rules force $\mathsf{t}'_2 \leq \{\,\}\rightarrow\{\,\mathsf{zero} : \{\,\}\rightarrow\mathsf{Bool}\,\}$ to be added to the constraints, but this is still consistent. On the other hand, consider $(\mathsf{zgnum} \mathrel{\mathsf{<-}} \mathsf{gcd}\ \mathsf{gnum}) \mathrel{\mathsf{<-}} \mathsf{zero}\ \{\,\}$. This may give a run-time error, so should not type-check. Indeed, $\mathsf{t}_2 \leq \{\,\}\rightarrow\{\,\mathsf{zero} : \{\,\}\rightarrow\mathsf{Bool}\,\}$ by transitive closure also requires a record without zero to be a subtype of a record with zero, but this is by definition an inconsistent constraint.

Compared to other work on rigorously sound class-based object languages, neither Bruce's TOOPLE or TOIL languages [4,5], nor our Loop language [12] allows the above program to type-check; in fact we know of no static type-system for object-oriented programming that successfully type-checks this example. So, not only do we obtain object type inference, we have a richer type language where it is not required to choose between "inheritance is subtyping" and the open-ended view of self.

6 Discussion

We have given a new, powerful method for type inference for object-oriented languages that is in many ways more powerful than previously existing methods. We have hopes that the core we present here will lead to development of a full-scale object-oriented programming language incorporating type inference. What we present here only shows this method is feasible, however. Further study is necessary to see if it can be implemented efficiently in practice. There also is the question of how well other language features will combine with this inference method. Modules in particular will be a challenge. There also should be separate syntax and types added for OOP features such as class definition and message send. This will provide a uniform notion of what OOP is to all programmers, and limit incompatibility of code. Lastly, even though this system is significantly stronger than the existing Hindley/Milner-style inference algorithms, the types it produces are larger and less easily readable by programmers. Thus it is important to address both the problem of simplification of these types, and the problem of how a better descriptions of what led to a type error can be given to programmers.

Acknowledgements

We would like to acknowledge Jens Palsberg for helpful discussions on related work, and Amy Zwarico for contributions in the early phases of this project.

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. Illyria system. Available by anonymous ftp from `ftp://s2k-ftp.cs.berkeley.edu/pub/personal/aiken/`, 1994.

[3] 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.

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

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

[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] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[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] J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. A simple interpretation of OOP in a language with state. Technical Report YALEU/DCS/RR-968, Yale University, 1993.

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

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

[13] 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.

[14] 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.

[15] D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient inference of partial types. In *Foundations of Computer Science*, 1992.

[16] D. B. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

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

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

[19] 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.

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

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

[22] 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.

[23] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.

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

[27] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Rice University Department of Computer Science, 1991. To appear in *Information and Computation.*