Safe and decidable type checking in an object-oriented language *

Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent
Williams College
Williamstown, MA

Allyn Dimock, Robert Muller
Harvard University
Cambridge, MA

Abstract

Over the last several years, much interesting work has been done in modelling object-oriented programming languages in terms of extensions of the bounded second-order lambda calculus, $F_{\leq}$. Unfortunately, it has recently been shown by Pierce ([Pie92]) that type checking $F_{\leq}$ is undecidable. Moreover, he showed that the undecidability arises in the seemingly simpler problem of determining whether one type is a subtype of another.

In [Bru93a, Bru93b], the first author introduced a statically-typed, functional, object-oriented programming language, TOOPL, which supports classes, objects, methods, instance variables, subtypes, and inheritance. The semantics of TOOPL is based on $F_{\leq}$, so the question arises whether type checking in this language is decidable.

In this paper we show that type checking for TOOPLE, a minor variant of TOOPL (Typed Object-Oriented Programming Language), is decidable. The proof proceeds by showing that subtyping is decidable, that all terms of TOOPLE have minimum types (which are in fact computable), and then using these two results to show that type checking is decidable. Our algorithm fails to be polynomial in the size of the term because the size of its type can be exponential in the size of the term. Nevertheless, it performs well in practice.

*Bruce, Crabtree, Murtagh, and van Gent were partially supported by NSF grant CCR-9121778. Dimock and Muller were partially supported by DARPA grant F19628-92-C-0113.

OOPSLA'93, pp. 29-46

1 Introduction

Beginning with the influential paper, [CW85], there has been a great deal of interest in using various extensions of $F_{\leq}$, the bounded second-order lambda calculus, as a basis for a theoretical understanding of object-oriented programming languages. Papers taking this approach include [Car92, Car89, CL91, CM90, CHC90, CCH+89, CCHO89, Mit90, Bru92, BL90, BM92, BTCGS91, PT92a, PT92b]. Among others, Ghelli ([Ghe90]) implemented a type checker for $F_{\leq}$, which he initially claimed was sound and complete. The soundness was correct, but Ghelli and others later discovered that the subalgorithm for determining whether one type was a subtype of another diverged on certain inputs. After several researchers attempted to patch the algorithm, Pierce [Pie92] proved that the problem of determining whether one type was a subtype of another is undecidable, and hence so is the problem of type-checking terms of $F_{\leq}$.

While it appears that the subtyping algorithm performs well in practice (all counter-examples appear to be contrived and similar), this negative result threw into question the notion of using $F_{\leq}$ as a foundation for object-oriented programming languages. In this paper we show that the type checking problems of the full $F_{\leq}$ do not necessarily have an impact on using that language as a basis for understanding the fundamental notions of object-oriented programming languages.

A series of papers, [Bru93a, Bru93b, BCK93], introduced and proved properties of TOOPL, a statically-typed, functional, object-oriented programming language which supports classes, ob-

jects, methods, instance variables, subtypes, and inheritance. The language was designed with a semantics based on an extension of $F_{\leq}$, the so-called "F-bounded" second-order lambda calculus (see [CHC90]), with recursively defined types and elements. It was presented with a set of axioms and rules for subtyping and type checking terms, but no algorithm was provided to do the type checking.

Pierce's results raised the question as to whether there was a complete algorithm for type checking in the language. The language TOOPL does not directly support polymorphic functions or higher-order types, so it is not clear that Pierce's results apply. On the other hand, the semantics of TOOPL types are specified as fixed points of bounded functions from types to types, and the semantics of class terms, for instance, are presented as higher-order terms of an extension of the bounded second-order lambda calculus. The strength of this underlying language, involving extensions to $F_{\leq}$, raised significant doubts about the existence of a type-checking algorithm for TOOPL.

The approach taken in [CG92] to developing an algorithm to type check a language with subtyping was to design an algorithm for computing the minimum type of a term and an algorithm for determining whether one type is a subtype of another. One can then use these two algorithms in order to determine if a term, M, has type τ as follows. Find the minimum type, τ_0, of M, and then determine if τ_0 is a subtype of τ. If so, the subsumption rule can be used to show that M has type τ. If τ_0 is not a subtype of τ then τ cannot be a type of M by the definition of minimum type.

This approach failed for $F_{\leq}$ since there is no algorithm to determine if one type is a subtype of another. The subtype checking (semi-)algorithm presented in [CG92] need only converge when the first type is a subtype of the other. If not, the algorithm may diverge. In this paper, we are able to show that a similar algorithm for TOOPL always halts. While there are terms of TOOPL which do not have minimum types, we work here with a minor variant, TOOPLE (TOOPL - Enhanced), which results from adding extra type information to class terms. We show that all terms of TOOPLE have minimum types, allowing us to carry through the program outlined above for type checking terms of TOOPLE.

While we expected some difficulties in finding minimum types for terms involving message sending, we discovered that computing minimum types for conditional expressions was unexpectedly complex. One must be able to compute least upper bounds of types in order to determine the minimum type of a conditional term from the types of its "true" and "false" branches. However, the interaction of this with the implicit recursion in objects and the contravariance of subtyping in function types forced us to define and compute a generalized form of least upper bounds and greatest lower bounds that were monotonic with respect to one collection of variables and anti-monotonic with respect to another. We indicate in Section 4 where these complications arise.

We had originally hoped to develop a polynomial time algorithm for type checking. However, as shown in section 4, the minimal type of a term may be exponential in the size of the term. As a result any algorithm which constructs (infers) minimal types cannot be polynomial in the size of the term. Nevertheless, the bad examples are quite contrived, and are extremely unlikely to occur in practice. Thus we expect this type-checking algorithm to perform well in practice.

For simplicity, the main portion of the paper deals only with a restriction of TOOPLE which does not involve instance variables. We indicate in Section 5 where some of the complexities arise with instance variables.

The paper is organized as follows. In Section 2 we present a brief description of TOOPLE. In Section 3 we describe the subtyping algorithm for this restricted language. In Section 4 we describe the algorithms to find the minimum type of a term and type check it. In Section 5 we describe briefly some of the extra complexities which arise by adding instance variables. Section 6 contains comparisons of our work with those of other researchers. Finally in Section 7 we summarize our results, describe the current state of our implementation of TOOPLE and mention some work in progress on TOOPLE.

2 A Brief Introduction to TOOPLE

TOOPLE is a statically-typed, functional, object-oriented programming language. It provides full support for object-oriented features including objects, classes, methods, instance variables, dynamic method invocation, subclasses, and subtypes. Moreover, TOOPLE provides mechanisms to allow the programmer to refer to the current object (*self*), its type (*MyType*), and the record of methods of its superclass (*super*). A description of the fundamental concepts of object-oriented languages is given below. It will be followed by an introduction to the syntax of TOOPLE.

An *object* consists of a collection of *instance variables*, representing the state of the object, and a collection of *methods*, which are routines for manipulating the object. When a *message* is sent to an object, the corresponding method of the object is executed. *Classes* are extensible templates for creating objects. In particular, classes contain initial values for instance variables and the bodies for methods. All objects generated from the same class share the same methods, but may contain different values for their instance variables. A *subclass* may be defined from a class by either adding to or modifying the methods and instance variables of the original class. Restrictions on the modification of the types of methods and instance variables in subclasses are necessary in order to preserve type safety.

In this conference paper, we defer further discussion of instance variables except for a few remarks in Section 5. The extension of TOOPLE to include instance variables is described in more detail in [Bru93a, Bru93b].

All terms of the language, including both classes and objects, have associated types. A type is either a variable (from a set, $\mathcal{V}^{Tp}$, of type variables), a constant (from a set, $\mathcal{C}^{Tp}$, of type constants), or of the form $\sigma \rightarrow \tau$ (for function types), $\{m_1{:}\,\tau_1; \ldots; m_n{:}\,\tau_n\}$ (for record types), $\mathit{ObjectType}(\mathit{MyType})\tau$ (for object types) or $\mathit{ClassType}(\mathit{MyType})\tau$ (for class types). The types, τ, in object and class types must be record types.

The pre-terms of TOOPL are given in Figure 1. In the grammar, B, M, N, e, e_i, c, and o all represent pre-terms. M and N are intended to suggest general pre-terms, B a Boolean expression, e a record, c a class, and o an object. The m and m_i are labels, while τ is a type. The variable x is from a fixed collection of variables $\mathcal{V}$.

Most of the pre-terms should be self-explanatory. A term of the form

$$\mathit{class}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e$$

represents a class whose method bodies are contained in the record e with type τ. The bound variable *self* may be used in method bodies in e to refer to the object executing the method. The bound variable *MyType* refers to the type of *self*. Since the method may be inherited by subclasses, the meaning of *MyType* at execution time may actually correspond to the type of an object generated by a subclass of the class being defined. The occurrence of $\mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau$ in the class term is meant to suggest this fact (see below for the definition of $\leq_{meth}$). "*Update*" and "*extend*" terms provide ways of modifying old methods or adding new ones to a class to form a subclass. Methods not mentioned in the subclass definition are inherited from the superclass.

If c is a class then *new* c represents an object generated from c. The type-checking rules will indicate that if c has type $\mathit{ClassType}(\mathit{MyType})\tau$, then *new* c will have corresponding type $\mathit{ObjectType}(\mathit{MyType})\tau$. A term of the form $o \Leftarrow m$ represents sending the message m to object o. A few simple examples of TOOPLE expressions are given at the end of this section.

We say type σ is a *subtype* of τ if a value of type σ can be used in any context in which a value of type τ is expected. Note that subtyping depends only on the type of values, while subclass depends upon implementations. Axioms and rules describing the subtyping relation for types of TOOPLE are given in the Appendix. Most rules should be familiar with the possible exception of the subtyping rule for object types. This rule arises from the fact that object types are defined recursively (in order for *MyType* to stand for the type of the object in its type definition), and is adopted from a similar rule in [AC90] for subtyping recursive types. See [Bru93b] or [Bru93a] for further explanation. Recall also that function types are contravariant in their domains.

$$M ::= x \mid \mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ N \mid \mathit{fun}(v{:}\sigma)M \mid M\ N \mid M = N \mid \{m_1 = e_1, \ldots, m_n = e_n\} \mid e.m_i \mid \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e \mid \mathit{new}\ c \mid o \Leftarrow m \mid \mathit{update}\ c\ \mathit{by}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau; \mathit{super})\{m_1 = e_1'\} \mid \mathit{extend}\ c\ \mathit{with}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau; \mathit{super})\{m_{n+1} = e_{n+1}\}.$$

Figure 1: Pre-terms of TOOPLE

There is a separate ordering on object types which is related to types obtained by taking subclasses (see [CHC90]). This ordering is a pointwise ordering on method types, and is denoted $\leq_{meth}$. It reflects the changes which may be made in constructing subclasses. In particular, if $\mathit{ObjectType}(\mathit{MyType})\tau$ is the type of an object, o, generated from class, c, and $\mathit{ObjectType}(\mathit{MyType})\tau'$ is the type of an object generated from a subclass of c, then $\mathit{ObjectType}(\mathit{MyType})\tau' \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau$. The axioms and rules for $\leq_{meth}$ are given in the Appendix.

The terms of TOOPLE are those pre-terms which can be type checked with respect to a collection, C, of subtyping and inheritance assumptions on types (called a *restricted type constraint system*), and an assignment, E, of types to variables. The definition of restricted type constraint system is given in Definition A.1 in the Appendix. The restriction on the simple type constraints make it easier for us to determine subtypes and to derive minimum types for terms. Note that they essentially forbid forcing a type variable to be a subtype of an object type. We have found no compelling reasons to allow less restricted constraints, and more complex constraints are not introduced by our subtyping or type-checking algorithms.

The type-assignment rules for TOOPLE can be found in Figures 4 and 5 in the Appendix. A further description of the language and its type-assignment rules can be found in [Bru93b] or [Bru93a]. We provide a brief description of the type-assignment rule for classes here.

In order to show $\mathit{class}(\mathit{self} : \mathit{MyType})e$ has type $\mathit{ClassType}(\mathit{MyType})\tau$, it is sufficient to show that e has type τ under the assumption that *self* has type *MyType*. In this derivation one may not assume that $\mathit{MyType} = \mathit{ObjectType}(\mathit{MyType})\tau$, only that $\mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau$. The reason for this is that the methods in e may be inherited in a subclass (whose type is guaranteed to be $\leq_{meth}$ $\mathit{ObjectType}(\mathit{MyType})\tau$). As a result, we may only make this weaker assumption in type-checking.

In [Bru93a, Bru93b, BCK93] it was shown that the type-checking rules for TOOPLE are *safe*. For instance, in the evaluation of a term that type checks correctly, an object will never be sent a message that it does not understand. Our goal in this paper is to find an algorithm which, given C, E, M, and τ, determines if $C, E \vdash M : \tau$.

The following are simple examples of terms and types from TOOPLE.

Let $\mathit{PointType} = \mathit{ObjectType}(\mathit{MyType})\{x, y{:}\ \mathit{Int}; \mathit{eq}{:}\ \mathit{MyType} \rightarrow \mathit{Bool}\}$, the type of objects with x, y, and eq methods. The following class will generate objects of this type.

$$\mathit{PointClass} = \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{PointType})\ \{x = 0,\ y = 0,\ \mathit{eq} = \mathit{fun}(p{:}\mathit{MyType})\ ((\mathit{self} \Leftarrow x) = (p \Leftarrow x))\ \&\ ((\mathit{self} \Leftarrow y) = (p \Leftarrow y))\}$$

Note that the method eq takes a parameter, p, with the same type as *self* and compares the results of sending messages x and y to p and the results of sending the same messages to *self*. $\mathit{PtObj} = \mathit{new}\ \mathit{PointClass}$ represents a new object of type *PointType*. Thus, if $(\mathit{PtObj} \Leftarrow \mathit{eq})\ (o)$ is to be well-typed, o must be of type *PointType*, since *MyType* will be instantiated to PointType when the message eq is sent to *PtObj*. (See the type-assignment rule (*Msg*).)

Suppose we now wish to modify *PointClass* by adding a color field. Let

$$\mathit{ColorPointType} = \mathit{ObjectType}(\mathit{MyType})\{x, y{:}\ \mathit{Int}; \mathit{eq}{:}\ \mathit{MyType} \rightarrow \mathit{Bool}; c{:}\ \mathit{ColorType}\}.$$

Then the class defined by

$$\mathit{ColorPointClass} = \mathit{extend}\ \mathit{PointClass}\ \mathit{by}\ (\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ColorPointType}, \mathit{super})\ \{c = \mathit{Red}\}$$

will generate objects of that type. All of the methods of *PointClass* are inherited unchanged in *Col-*

orPointClass. Notice, however, that if the *eq* message is sent to an object of type *ColorPointType*, the parameter for the message must also be of type *ColorPointType*, not *PointType*. This is an important example of how the meaning of *MyType* may change when methods are inherited.

If we wish to change the method *eq* so that it now also checks the color components of two records, we define

$$\begin{aligned} &\mathit{NuColorPtClass} = \mathit{update}\ \mathit{ColorPointClass}\ \mathit{with} \\ &\quad (\mathit{self}:\mathit{MyType} \leq_{\mathit{meth}} \mathit{ColorPointType},\ \mathit{super}) \\ &\quad \{\mathit{eq} = \mathit{fun}(p{:}\mathit{MyType})\ \mathit{super.eq}(p) \\ &\quad\quad \& \ ((\mathit{self} \Leftarrow c) = (p \Leftarrow c))\ \}. \end{aligned}$$

Notice that the updated method *eq* in *NuColorPtClass* calls the inherited *eq* from *ColorPointClass* (using the keyword *super*) and then checks the "*c*" components for equality. All other methods from *ColorPointClass* are inherited unchanged. Note as well that both *ColorPointClass* and *NuColorPtClass* generate objects of type *ColorPointType*.

The denotational semantics of TOOPLE is written in terms of an extension of $F_{\leq}$. For example, if ρ is an assigment of variables to elements of a model $\mathcal{A}$ of $F_{\leq}$, the semantics of classes are as given in Figure 2.

While this is quite complex, the main point to note is that the meaning is defined in terms of a function whose parameter ranges over types, ξ, such that $\xi \leq_{\mathcal{A}} [\![\tau]\!]\rho[\xi/\mathit{MyType}]$. This is a term of an extension of $F_{\leq}$ (this term is in an extension because the bound on ξ is an expression involving ξ, something not allowed in $F_{\leq}$). Moreover, the semantics of class types involve second order bounded quantification over types.

Thus, while the syntax of TOOPLE appears to have little to do with $F_{\leq}$, the denotational semantics depends heavily upon it. This raised concerns about the relevance of Pierce's undecidability result to TOOPLE. The question is whether the dependence is great enough for type-checking of TOOPLE to be undecidable.

3 Subtype Checking

In this section we show that for a restricted type constraint system, C, and types, σ and τ, $C \vdash \sigma \leq \tau$ is decidable. Our proof has two main steps. First, we introduce a new subtype system that differs from the original in that it has a restricted form of the transitivity rule. We prove that the new system is equivalent to the original one in the sense that a subtyping judgement is derivable in the old system if and only if it is derivable in the new one. Once we specify the application of the (*SRefl*) axiom in the new system, we obtain a canonical-form proof tree. We then specify a deterministic strategy for applying the new rules and prove that the strategy constitutes an algorithm for subtype checking — if a subtyping judgement is derivable, it can be derived using this strategy and the strategy halts on all inputs. This is shown by defining a decreasing metric on the size of types.

The proof of decidability outlined above follows the general approach presented in [CG92] for subtype checking in $F_{\leq}$. The key difference is that their strategy gives only a *semi*-algorithm: for certain non-theorems their semi-algorithm enters an infinite loop.

3.1 A Canonical-Form Subtype System

We first show that any judgement derivable in the subtype system of the Appendix is also derivable in a system in which the (*STrans*) rule is replaced by the following specialization of it:

$$(STrans') \qquad \frac{C \vdash t \leq \sigma, C \vdash \sigma \leq \tau}{C \vdash t \leq \tau}$$

where t is a type variable or constant.

Let $\vdash_T$ denote provability in the restricted system. The connection between the two systems is given by the following lemma.

Lemma 3.1 *For restricted type constraint system, C, and type expressions, σ and τ, $C \vdash \sigma \leq \tau \Leftrightarrow C \vdash_T \sigma \leq \tau$.*

The proof of Lemma 3.1 requires the following lemma which guarantees that comparable types have the same shape.

Lemma 3.2 *Let C be a restricted type constraint system, and let σ and τ be type expressions. If $C \vdash \sigma \leq \tau$, then σ and τ are structurally similar at the top level.*

Proof. (Of Lemma 3.1.)

$$[\![C, E \vdash class(self : MyType)e: ClassType(MyType)\tau]\!]\rho = \lambda\xi \leq_{\mathcal{A}} [\![\tau]\!]\rho[\xi/MyType].\lambda o \in \mathcal{A}^{\xi}.$$
$$[\![C;\ MyType \leq_{meth} ObjectType(MyType)\tau, E \cup \{self : MyType\} \vdash e: \tau]\!]\rho[\xi/MyType, o/self\ ]$$

Figure 2: Denotational semantics of classes

$\Leftarrow$ Trivial.

$\Rightarrow$ We show by cases on type expressions that any fragment of a proof tree with (*STrans*) at its root can be rewritten so that the (*STrans*) rule has been moved leafward through ($S\rightarrow$), (*SRec*) and (*SObj*) nodes. In light of [Ghe92], it is important that the rewrite rules preserve the leaves, the root and the height of the proof tree fragment.

We first simplify proof trees in which (*STrans*) appears at the root and where either antecedent is (*SRefl*) by eliminating the (*STrans*) and the (*SRefl*) nodes. By Lemma 3.2 and by induction on the height of proof trees, a proof tree with (*STrans*) at the root, a structured rule in one antecedent and (*STrans*) in the other can be rewritten so that the same structured rule appears in both antecedents.

The rewrite rules for proofs in which both antecedents are either ($S\rightarrow$) and (*SRec*) are straightforward and omitted here. The rewrite for (*SObj*) requires the following technical lemma.

Lemma 3.3 *If* $C;\ t \leq t_1$ *is a restricted type constraint system, then if*

$$C;\ t \leq t_1 \vdash \tau[t/MyType] \leq \tau_1[t_1/MyType]$$

is provable, then so is

$$C \vdash \tau[t_1/MyType] \leq \tau_1[t_1/MyType].$$

Moreover, the height of the proof tree of the second type assignment is no greater than that of the first.

In Figure 3, we show the transformation for (*SObj*) nodes. Note that the last transformation in the Figure involves inserting a direct proof (guaranteed to exist by the above Lemma) for $C;\ t_1 \leq t_2 \vdash \tau[t_1/MyType] \leq \tau_1[t_1/MyType]$ in the upper left-most part of the proof tree in place of one which used the Lemma and weakening. This new proof has height no greater than the original.

Since C contains only simple type constraints, and since we have now shown that (*STrans*) can be moved leafward in a subtyping rule through any structured rule, it follows that all (*STrans*) nodes in the resulting tree are of the form given in rule (*STrans'*) at the beginning of this section. ∎

The only remaining degree of freedom is the application of the (*SRefl*) axiom for inclusions of the form $\sigma \leq \sigma$. The (*SRefl*) axiom is obviously required for proving such inclusions between type constants and variables. It is also required when $\sigma = ObjectType(MyType)\tau$ and *MyType* occurs in a contravariant position in τ. We restrict applications of (*SRefl*) to these two cases. Although the (*SRefl*) axiom is also applicable when σ is a record or function type, the following lemma guarantees that a (different) proof can always be obtained by first destructuring with the (*SRec*) or ($S\rightarrow$) rules, respectively.

Lemma 3.4 *For restricted type constraint system* C *and type expressions* σ *and* τ*, if* $C \vdash \sigma \leq \tau$ *then there exists a proof tree in which the* (SRefl) *axiom is never applied to a record or function type.*

3.2 An Algorithm for the Canonical-Form System

We now present a deterministic algorithm for tracing canonical-form proof trees.

Algorithm $\mathbf{S}(C, \sigma, \tau)$: return *true* if $C \vdash_T \sigma \leq \tau$ and *false* otherwise.

1. If $\sigma = t$, where $t \in \mathcal{V} \cup \mathcal{C}$, then
   (a) If $\tau = t$ then the proof is completed using (*SRefl*). Return *true*.
   (b) Otherwise, if $t \leq \tau' \in C$, then use (*STrans'*). Return $\mathbf{S}(C, \tau', \tau)$.
   (c) Otherwise, if $t \leq \tau' \notin C$ then return *false*.
2. If $\sigma = \sigma_1 \rightarrow \sigma_2$, $\tau = \tau_1 \rightarrow \tau_2$, then use ($S\rightarrow$). Return $\mathbf{S}(C, \tau_1, \sigma_1)$ **and** $\mathbf{S}(C, \sigma_2, \tau_2)$.
3. If $\sigma = \{m_1: \sigma_1; \ldots; m_n: \sigma_n\}$, $\tau = \{m_1: \tau_1; \ldots; m_k: \tau_k\}$, then use (*SRec*). Return $k \leq n$ **and** $\mathbf{S}(C, \sigma_1, \tau_1)$ **and** ... **and** $\mathbf{S}(C, \sigma_k, \tau_k)$.
4. If $\sigma = ObjectType(MyType)\sigma'$, then

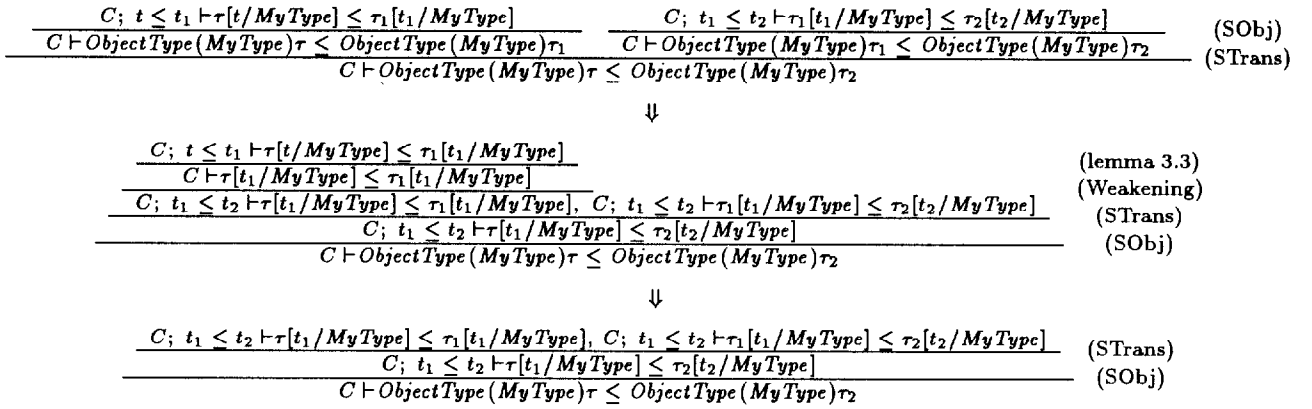

Figure 3: Transformation for $(SObj)$ Nodes

(a) If $\tau = ObjectType(MyType)\sigma'$ then the proof is completed using $(SRefl)$. Return *true*.

(b) Otherwise, if $\tau = ObjectType(MyType)\tau'$ then use $(SObj)$. Return $\mathbf{S}(C;\ s \leq t, \sigma'[s/MyType], \tau'[t/MyType])$.

Lemma 3.5 *If C is a restricted type constraint system and σ and τ are type expressions then if $C \vdash_T \sigma \leq \tau$ then $\mathbf{S}(C, \sigma, \tau) =$ true and if $C \not\vdash_T \sigma \leq \tau$ then $\mathbf{S}(C, \sigma, \tau) =$ false.*

Proof. (Sketch) The restricted type constraint system condition ensures that C has a forest ordering, and the extension to C in rule $(SObj)$ preserves this order. Therefore, there are no infinite sequences of $(STrans')$'s. $(SObj)$, $(SRec)$ and $(S\rightarrow)$ decrease the number of type constructors on recursion. ∎

Finally, the main theorem:

Theorem 3.6 *Let C be a restricted type constraint system and let σ and τ be type expressions. Then it is decidable whether $C \vdash \sigma \leq \tau$.*

Proof. Immediate from Lemmas 3.1, 3.4 and 3.5. ∎

3.3 Complexity of the Subtyping Algorithm

Under realistic assumptions on the cost of primitive operations, algorithm $\mathbf{S}$ is time $O(n^2)$ in the maximum of the number of type constructors in σ and τ or the length of the longest chain in C, assuming C starts with only subtype relations between constants. The worst case comes from trying $(SRefl)$ many times on nested object types.

The algorithm can be improved by a prepass over σ, τ marking which uses of *MyType* are covariant relative to their definitions, and which are contravariant; thus fixing the choice of $(SObj)$, $(SRefl)$ in advance. The prepass then becomes the asymptotically greatest cost requiring either $E(O(n))$ using hash tables, or $O(n \log n)$ using balanced trees.

4 Type Checking and Minimum Types in TOOPLE

Definition 4.1 *We say e is typable with respect to C, E iff there is a type τ such that $C, E \vdash e{:}\tau$. We say that τ is the minimum type for e with respect to C, E iff $C, E \vdash e{:}\tau$ and for all τ', if $C, E \vdash e{:}\tau'$, then $C \vdash \tau \leq \tau'$.*

In this section we show that every pre-term of TOOPLE which is typable has a minimum type, and that this minimum type is computable. The derivation of the algorithm to compute the minimum type, and the proof of its correctness were complicated by a number of issues that have been mentioned earlier. It was necessary to add annotations to the class terms of the original TOOPL language to ensure that minimum types existed, and it was necessary to show that it was possible to determine if two types had a least upper bound and to find that bound. Once these issues have been addressed, most of the argument supporting the

algorithm to find minimum types is fairly straightforward, with the possible exception of the handing of the message passing operation.

In the remainder of this section, we briefly discuss each of the issues mentioned above, state the theorem that establishes the decidability of typing in TOOPLE, and outline its proof.

4.1 Minimum Types for Classes

In TOOPL, class terms did not include any constraint on the type produced. Thus, one would write $class(self : MyType)e$ in TOOPL rather than $class(self : MyType \leq_{meth} ObjectType(MyType)\tau)$ e. Without this extra information, a class term such as $class(self : MyType)\{m = self \Leftarrow m\}$ would not have a minimum type. Possible types for this term include all types of the form $ObjectType(MyType)\{m{:}\,\tau\}$ for any type τ. There is no smallest type of this form.

To avoid this difficulty in TOOPLE, we annotate class terms with type information. From the typing rules in Figures 4 and 5 of the Appendix, it is easy to see that the only possible type for a term of the form $class(self : MyType \leq_{meth} ObjectType(MyType)\tau)e$ is $ClassType(MyType)\tau$. Similar type information in *update* and *extend* terms ensures that every typable class term has a unique type.

4.2 Least Upper Bounds of Type Pairs

In order to find the minimum type of a conditional expression, one must find the least upper bound (lub) of the minimum types of the *then* and *else* branches of the expression. For structured types, the obvious way to look for least upper bounds is to recursively look for bounds on the subexpressions associated with the range and domain of the function space, the corresponding components of record types or the bodies of object types. In particular, one can prove:

Lemma 4.2 *Given types, σ and τ, and a restricted type constraint system, C, $lub(\sigma, \tau, C)$ exists only if σ and τ are structurally similar at the top level. Furthermore, the lub is structurally similar at the top level to σ and τ.*

Due to the contravariant nature of the subtyping rule for functions, such a recursive algorithm to find lubs requires a corresponding algorithm to find greatest lower bounds (glb's).

To make this recursion work in the case of object types with local bound names, we need a notion of lower and upper bounds relative to sets of local names. To see why, consider the problem of finding a lower bound for the types:

$$\sigma = ObjectType(MyType)\{x{:}\,integer;\ y{:}\,MyType \rightarrow MyType\}$$

and

$$\tau = ObjectType(MyType)\{y{:}\,MyType \rightarrow MyType\}.$$

It is clear that any lower bound for such types must contain both an "x" and a "y" component. The type of the "x" component would be "integer" and it seems clear that the type of the "y" component should be $MyType \rightarrow MyType$. That is, σ appears to be a lower bound for these two types.

However, to show that $\vdash \sigma \leq \tau$ using the axioms and rules for subtyping, one must show that

$$s \leq t \vdash \{x{:}\,integer; y{:}\,s \rightarrow s\} \leq \{y{:}\,t \rightarrow t\}$$

which is impossible due to the contravariant rule for subtyping of function types. In fact, it is easy to see that τ has no proper subtype in which y has type $MyType \rightarrow MyType$.

The problem is that when we look for lub's or glb's of subtypes, we must ensure that it will be provable that the types we select are bounds using the limited assumptions the subtyping axioms and rules will allow us to make about local names of object types. To address this problem, we must define a notion of a type that provably bounds two other types even when only limited assumptions are made about the relationships between free variables appearing in the two types.

Definition 4.3 *For types, σ, τ and γ, a restricted type constraint system, C, and two sets of type variables, $L = \{L_1, L_2, \ldots, L_n\}$ and $U = \{U_1, U_2, \ldots, U_k\}$, we will say that γ is* **a monotonic upper bound for σ and τ relative to C, L and U** *if, given a set $\{L'_1, \ldots, L'_n, U'_1, \ldots U'_k, L''_1, \ldots, L''_n, U''_1, \ldots, U''_k\}$ of variables distinct from those appearing in σ, τ, γ, C, L or U, the following conditions hold:*

1. $C' \vdash \sigma[L_1'/L_1, \ldots, U_k'/U_k] \leq \gamma[L_1''/L_1, \ldots, U_k''/U_k]$

2. $C' \vdash \tau[L_1'/L_1, \ldots, U_k'/U_k] \leq \gamma[L_1''/L_1, \ldots, U_k''/U_k]$

where $C' = C$; $L_1' \leq L_1''; \ldots; L_n' \leq L_n''; U_1'' \leq U_1'; \ldots; U_k'' \leq U_k'$.

We define the notion of a least monotonic upper bound (lmub) in the obvious way. Note that in the case that the sets U and L are empty, which is how we begin our recursive algorithm, the least monotonic upper bound and least upper bound are identical.

For each kind of structured type, we can show a result similar to the following, which leads to an algorithm for computing lmub's (and gmlb's).

Lemma 4.4
Given types, $\sigma = ObjectType(MyType)\sigma_M$ *and* $\tau = ObjectType(MyType)\tau_M$, *a restriced type constraint system,* C, *and two sets of type variables,* $L = \{L_1, L_2, \ldots L_n\}$ *and* $U = \{U_1, U_2, \ldots, U_k\}$, *then* $lmub(\sigma, \tau, C, L, U)$ *exists if and only if* $\gamma_M =_{def} lmub(\sigma_M, \tau_M, C, L \cup \{MyType\}, U)$ *exists and*

$$lmub(\sigma, \tau, C, L, U) = ObjectType(MyType)\gamma_M$$

In particular we get the following important result.

Lemma 4.5 *Let* σ, τ *and* γ *be types,* C *a restricted type constraint system, and let* L *and* U *be disjoint sets of type variables.*

1. *There is an algorithm which determines if* $lmub(\sigma, \tau, C, L, U)$ *exists, and if so, returns that type.*

2. *If* γ *is a monotonic upper bound for* σ *and* τ *relative to* C, L *and* U, *then* $lmub(\sigma, \tau, C, L, U)$ *exists.*

Similar results hold for monotonic lower bounds.

4.3 The Existence of Minimum Types

The key to the proof of the existence of minimum types is a deterministic set of rules for deriving minimum types, marked with $\vdash_M$. These rules can be found in Figure 6 of the Appendix. Note that there are now two distinct type-checking rules for each of function application, record component extraction, and message passing. The extra cases result when the minimum type of a term is given by a type variable, yet, by subtyping (or $\leq_{meth}$), it is known that the type must represent either a functional or object type.[1]

The relation, $t \ll \tau$, defined in the Appendix, helps determine the smallest type expression, τ, which is not a type variable and is greater than t. It turns out to be useful in determining minimum types of terms. Note that the second message passing rule, in which the type of o is a type variable rather than a *ObjectType* type expression, actually arises frequently in practice when the user sends a message to *self*, since *self* has type *MyType*.

The following theorem provides the basis for our algorithm to find minimum types.

Theorem 4.6 *Let* C *be a restricted type constraint system,* E *a syntactic type assignment,* e *a pre-term of TOOPLE, and* τ *a type.*

1. *If* $C, E \vdash_M e{:}\tau$ *then* $C, E \vdash e{:}\tau$.

2. *If* e *is typable with respect to* C, E, *then there is a* τ *such that* $C, E \vdash_M e{:}\tau$ *and* τ *is the minimum type for* e *with respect to* C, E. τ *is unique up to renaming of bound variables.*

Proof. The proof of (1) is easy. The proof of (2) is by induction on the size of e. The most interesting cases are for conditionals, classes, and message sending. The argument for conditionals follows easily from Lemma 4.5, part 2. The proof for classes is trivial since the only possible type for $class(self : MyType \leq_{meth} ObjectType(MyType)\tau)e$ is $ClassType(MyType)\tau$. The proof for message sending is complex and divides into two cases depending on whether the minimum type of the receiving object is an object type or a type variable (*e.g.*, *MyType*).

Suppose $C, E \vdash o \Leftarrow m_i{:}\rho$. By induction we may suppose that $C, E \vdash_M o{:}\rho'$. There are two possibilities to consider. The first is that ρ' is an

[1] While the extra cases for function application and record extraction do not arise with our restricted type constraints, they do for other reasonable restrictions on type constraints, so we include those rules here.

object type and the second is that ρ' is some type variable t.

Case 1: Suppose that $C, E \vdash_M o$: $ObjectType(MyType)\{m_1: \tau_1; \ldots; m_n: \tau_n\}$. Then by the (MMsg) rule, $C, E \vdash_M o \Leftarrow m_i$: $\tau_i[ObjectType(MyType)\{m_1: \tau_1; \ldots; m_n: \tau_n\}/MyType]$. We claim that the type $\tau_i[ObjectType(MyType)\{m_1: \tau_1; \ldots; m_n: \tau_n\}/MyType]$ is the minimum for $o \Leftarrow m_i$ with respect to C, E.

Suppose that $C, E \vdash o \Leftarrow m_i: \tau'$ for some τ'. Without loss of generality, we may assume that the last step of the proof of that typing is (Msg). Thus $\tau' = \tau_i'[\gamma/MyType]$, where $C, E \vdash o: \gamma$ and $C \vdash \gamma \leq_{meth} ObjectType(MyType)\{m_i: \tau_i'\}$. Since $ObjectType(MyType)\{m_1: \tau_1; \ldots; m_n: \tau_n\}$ is minimum for o,

$$C \vdash ObjectType(MyType)\{m_1: \tau_1; \ldots; m_n: \tau_n\} \leq \gamma.$$

Inspection of subtyping and inheritance rules indicates that γ must be of the form $ObjectType(MyType)\{\ldots; m_i: \tau_i''; \ldots\}$ (by the subtyping rules), and $C \vdash \tau_i'' \leq \tau_i'$ (by the $\leq_{meth}$ rules). In particular, $C \cup \{s \leq t\} \vdash \{m_1: \tau_1; \ldots m_i: \tau_i; \ldots m_n: \tau_n\}[s/MyType] \leq \{\ldots; m_i: \tau_i''; \ldots\}[t/MyType]$ (by the subtyping rules). The latter implies that $C \cup \{s \leq t\} \vdash \tau_i[s/MyType] \leq \tau_i''[t/MyType]$. Letting $t = \gamma$ and $s = ObjectType(MyType)\{m_1: \tau_1; \ldots; m_n: \tau_n\}$, it follows that

$$C \vdash \tau_i[ObjectType(MyType)\{m_1: \tau_1; \ldots; m_n: \tau_n\}/ MyType] \leq \tau_i''[\gamma/MyType] \leq \tau_i'[\gamma/MyType],$$

confirming that

$$\tau_i[ObjectType(MyType)\{m_1: \tau_1; \ldots; m_n: \tau_n\}/MyType]$$

is minimum.

Case 2: Suppose that $C, E \vdash_M o: t$, where t is a type variable. Since $o \Leftarrow m_i$ is typable, there is a γ such that $C, E \vdash o: \gamma$ and $C \vdash \gamma \leq_{meth} ObjectType(MyType)\{m_i: \tau_i\}$. Since t is minimum, $C \vdash t \leq \gamma$.

Because

$$C \vdash \gamma \leq_{meth} ObjectType(MyType)\{m_i: \tau_i\},$$

γ must be a type variable or of the form $ObjectType(MyType)\sigma$. However, since C is restricted, it cannot be the case that $C \vdash t \leq ObjectType(MyType)\sigma$. Thus γ must be a type variable.

An examination of the $\leq_{meth}$ rules shows that if $C \vdash \gamma \leq_{meth} ObjectType(MyType)\{m_i: \tau_i\}$, for γ a type variable, then $\gamma \leq_{meth} ObjectType(MyType)\{\ldots; m_i: \tau_i'; \ldots\} \in C$ for some object type such that $C \vdash \tau_i' \leq \tau_i$. Again because C is restricted, $C \vdash t \leq \gamma$ only if $t = \gamma$. Thus $t \leq_{meth} ObjectType(MyType)\{\ldots; m_i: \tau_i'; \ldots\} \in C$. Thus, $C, E \vdash_M o \Leftarrow m_i: \tau_i'[t/MyType]$. Moreover, $C \vdash \tau_i'[t/MyType] \leq \tau_i[t/MyType]$. Using similar ideas, one can show that $\tau_i'[t/MyType]$ is the minimum type of $o \Leftarrow m_i$ with respect to C, E. ∎

We can now write down the algorithm for minimum typing.

Algorithm M(C, E, e): return the minimum type for e, with respect to C, E, if it exists, and *false* otherwise.

(Sketch) Generally, the conclusion of only one of the minimum typing rules will match the shape of e. (In case of function application or message sending, determining the minimum type of the function or receiving object uniquely determines the appropriate rule.) For each hypothesis of that rule which is a type assignment for a subterm of e, call this algorithm recursively to determine the minimum type (if any) of that subterm. If the hypothesis includes a subtype assertion, call algorithm S from the previous section. If the hypothesis involves computation of a lub of two types, use the algorithm sketched at the end of Section 4.2. The only other hypotheses involve looking up items in C. If any of these fail, the entire algorithm fails. Otherwise use the types returned from the hypotheses to construct the appropriate type for e.

The correctness of the algorithm follows from the previous theorem.

Corollary 4.7 *There is an algorithm which, given C, E, e, and τ, determines if e is typable with respect to C, E, and if so, whether $C, E \vdash e: \tau$.*

Proof. The algorithm proceeds by using Algorithm M to compute the minimum type of e. If there is no such type, then e is not typable. If e does have a minimum type, τ', then call Algorithm S with C, E, τ', and τ. If it returns true then $C, E \vdash e: \tau$ by Theorem 4.6, part 1, and subsump-

tion. If it returns false then e cannot possibly have type τ since τ' was the minimum type of e. ∎

Unfortunately, the algorithm given is not polynomial in the size of the inputs. The problem is that the size of the type of a term is not bounded by a polynomial on the size of the term. It is easy to write a sequence of terms, $\{o_n\}$, such that for each $n < \omega$,

$$\begin{aligned}\vdash o_n\colon & \mathit{ObjectType}(\mathit{MyType}_1)\\ & \{m_1\colon \mathit{ObjectType}(\mathit{MyType}_2)\\ & \quad \{m_2\colon \mathit{ObjectType}\ (\mathit{MyType}_3)\\ & \qquad \ldots\\ & \qquad \{m_n\colon \mathit{MyType}_1 \to \mathit{MyType}_2 \to\\ & \qquad\quad \ldots \to \mathit{MyType}_n\}\}\ldots\}\end{aligned}$$

It is then easy to show that the sizes of the types of the terms,

$$o_n \Leftarrow m_1 \Leftarrow m_2 \Leftarrow \ldots \Leftarrow m_n$$

grow exponentially in n, whereas the sizes of the terms themselves are proportional to n^2.

As a result, algorithm M may involve calls to the subtyping algorithm on types whose size is exponential in the size of the term. On the other hand, it is certainly not common to define terms whose types involve nested (and dependent) object type definitions. Thus while the worst case behavior of the algorithm is not good, we expect it to perform in acceptably small polynomial time in practice.

5 Adding Instance Variables to TOOPLE

The extension of TOOPLE to include hidden instance variables is described in [Bru93a]. The key difference between methods and instance variables is that methods are frozen when an object is created, while the instance variables of an object may be updated. Values of instance variables are specified in class definitions, providing initial values to be used when new objects are created from classes (using the *new* operator). However it is possible to make a new copy of an object with a different value for an instance variable using the "*gets*" expression.

In TOOPL we provide different notation for accessing instance variables than for sending messages. We write $q.x$ to access the instance variable x of q and $p \Leftarrow getx$ to send the message $getx$ to p. We update an instance variable x of object p by writing $p\ gets\ \{x = e\}$. The value of this expression is a new object identical to p but with the value of e replacing the old value of x.

We do not wish to have instance variables visible outside of an object. Thus, we will have two different views of an object: the view from inside the object in which all instance variables are visible, and that from without, in which all instance variables are hidden. We will continue to refer to the type of an object from the outside using *MyType*, but we will now refer to the type from the inside using *SelfType*. Inside a method, the type of *self* will now be *SelfType*. Often we will need to "close up" an object to hide the instance variables from the outside world. The function "*close*" with type $\mathit{SelfType} \to \mathit{MyType}$ will perform this action. The following example of a movable point in the full TOOPLE should get across the basic idea.

Let $\mathit{PtInst} = \{x, y\colon \mathit{Int}\}$ and

$$\begin{aligned}\mathit{PtMeth} = \{&mv\colon \mathit{Int} \to \mathit{Int} \to \mathit{MyType};\\ &getx, gety\colon \mathit{Int}; eq\colon \mathit{MyType} \to \mathit{Bool}\})\end{aligned}$$

The following class has instance variables x and y which are initialized to 0.

$$\begin{aligned}&\mathit{PointClass} = \mathit{class}\\ &\quad (\ \mathit{self} : \mathit{SelfType} \le (\mathit{PtInst}, \mathit{PtMeth}),\\ &\quad\ \ \mathit{close}\colon \mathit{SelfType} \to \mathit{MyType})\\ &\quad (\{x = 0,\ y = 0\},\\ &\quad\ \{mv(dx, dy\colon \mathit{Int}) = \mathit{close}(\mathit{self}\ gets\\ &\qquad\qquad \{x = \mathit{self}.x + dx, y = \mathit{self}.y + dy\}),\\ &\quad\ \ getx = \mathit{self}.x,\\ &\quad\ \ gety = \mathit{self}.y,\\ &\quad\ \ eq = \mathit{fun}(p\colon \mathit{MyType})\ (\mathit{self}.x =\\ &\qquad (p \Leftarrow getx))\ \&\ (\mathit{self}.y = (p \Leftarrow gety))\})\end{aligned}$$

Notice that there is no way to directly access the instance variables of the parameter p of eq. The type of this class is $\mathit{PointClassType} = \mathit{ClassType}(\mathit{MyType})(\mathit{PtInst}, \mathit{PtMeth})$.

While the instance variables are visible in the type of *PointClass*, they are not visible in the corresponding object types. If $\mathit{MyPoint} = new\ \mathit{PointClass}$ then the type of *MyPoint* is:

$$\mathit{PointType} = \mathit{ObjectType}(\mathit{MyType})\mathit{PtMeth}.$$

We have extended our algorithm for determining

subtypes and minimum types in this more interesting language.

5.1 Extending Subtype Checking for Instance Variables

In this section we sketch how algorithm S can be extended to include instance variables.

The definition of same shape is loosened so that s and τ have the same shape if $s \ll \sigma$ and σ and τ have the same top level constructor.

The type expressions of the full language include pairs (δ, γ) where δ is a record of instance variable types and γ is a record of method types. As with the other structured types, the (*STrans*) rule can be moved through the components of a pair. However, a complication arises because the new (*Class*), (*Update*) and (*Extend*) rules introduce non-trivial type constraints of the form $SelfType \leq (\delta, \gamma)$ into the type constraint system.

Because these rules are constrained so that *SelfType* is not included in either δ or γ, any constraint $t \leq \tau'$ in C can be discharged when used: the subgoals are provable in C if and only if they are provable in the smaller constraint system $C - \{t \leq \tau'\}$.

In order to prove termination of the extended algorithm on $C \vdash \sigma \leq \tau$, we take as our metric the number of subtype constraints in C and the combined sizes of σ and τ. It is easy to see that the metric is strictly decreasing: any use of the (*Trans'*) rule reduces the size of C and the remaining structured rules reduce the size of σ and τ.

The $O(n^2)$ complexity result continues to hold since the case of (*STrans'*) on a (*Sobj*) rule is identical, and because the side condition on the (*Class*) rule ensures that in the new case of $C \vdash SelfType \leq (\delta, \gamma),\ C \vdash (\delta, \gamma) \leq (\delta', \gamma') \Rightarrow C \vdash SelfType \leq (\delta', \gamma')$, that there are no $SelfType_i$ variables in either (δ, γ) or (δ', γ').

5.2 Extending the Minimal Typing

The cases for type-checking terms which set or evaluate instance variables inside methods involve extra complications similar to those for function application and message sending. One must distinguish between cases in which the (internal) type of the object is given as an explicit pair of types (one type for the record of instance variables, the other for the record of methods) and the case in which the type of the object is a type variable. This latter case is actually the most common, since we typically have access only to the instance variables of *self* (or objects representing updated versions of *self*), whose type is now *SelfType*.

By adding relatively minor and natural restrictions on τ such that $(t \leq \tau) \in C$ (essentially, τ cannot represent an external object type), we can still show all terms have minimum types. The algorithm for type-checking is as before.

6 Comparison with previous work

As noted earlier, Curient and Ghelli [CG92] sketched out a plan for proving type checking is decidable in $F_{\leq}$ by providing algorithms to check subtyping and for computing minimum types. They were only able to provide a semi-algorithm for checking subtypes, however. Later, the type-checking problem for $F_{\leq}$ was shown to be undecidable by Pierce ([Pie92]), who showed that the problem of determining whether one type was a subtype of another type was undecidable in $F_{\leq}$.

Amadio and Cardelli ([AC90]) described an algorithm for determining subtyping relations in a language with subtyping and recursive types, but no polymorphic types. A simpler and more efficient algorithm to solve this problem using finite automata was later given in [KPS93].

While the denotational meaning of object types in TOOPLE can be given recursively, and the subtyping rules for object types are based on the rules in [AC90], the subtyping rules for TOOPLE are slightly weaker than for their language. We do not directly support recursively defined types in the language and we do not allow object definitions to be unwound to be recursively defined records. Unfortunately, we see no way of adapting the clever algorithm of [KPS93] for determining subtypes to our situation. The algorithm for subtyping in TOOPLE is fairly straightforward (though its proof of correctness is complex). On the other hand, the computation of minimum types is more complex than originally anticipated, with conditional expressions raising unexpected difficul-

ties, and with the size of minimal types exponential in the size of the terms in the worst case.

7 Summary

In this paper we have described a type-checking algorithm for the language, TOOPLE, a functional object-oriented language whose semantics is based on $F_{\leq}$. While the algorithm described in this extended abstract does not include instance variables, a similar algorithm can be described for the full language.

This paper is one of a series investigating theoretical and computational properties of TOOPLE. [Bru93a] and [Bru93b] presented type-checking axioms and rules, a denotational semantics for TOOPLE, and showed that the type system was safe. [BCK93] presented a natural (operational) semantics, proved a subject reduction theorem for the language (giving an alternative proof of type safety), and proved the relative consistency of the denotational and operational semantics for TOOPLE.

Our goal in this paper was to show that TOOPLE has good practical as well as theoretical properties. We believe this provides rather convincing evidence that one can indeed use $F_{\leq}$ as a foundation for the study of object-oriented programming languages. In particular, the design and implementation of TOOPLE shows that one can design a type-safe statically-typed programming language which captures the most important features of object-oriented languages, including classes, objects, methods, hidden instance variables, subtypes, and inheritance, while retaining important practical features (*e.g.*, the decidability of type checking). While the type-checking algorithm for TOOPLE is not polynomial, the examples generating this behavior are very unnatural and are unlikely to arise in practice.

We have implemented a fully functional interpreter for TOOPLE (with instance variables) in ML. The interpreter parses expressions, applies a type checking algorithm which is based on the minimum typing rules for the language, and then evaluates the expression by implementing the natural semantics given in [BCK93]. Further work is continuing on extending the language to include imperative features and to develop formal techniques for verifying programs in TOOPLE, with special emphasis on avoiding the need for re-verifying inherited methods in subclasses.

While TOOPLE is certainly not a full-featured language at this point, we believe that it can serve as the basis for the core of a practical, well-behaved, object-oriented language.

References

[AC90] Roberto Amadio and Luca Cardelli. Subtyping recursive types. Technical Report 62, Digital Systems Research Center, 1990.

[BCK93] K. Bruce, J. Crabtree, and G. Kanapathy. An operational semantics for TOOPLE: A statically-typed object-oriented programming language. To appear in Proceedings of MFPS IX, 1993.

[BL90] K. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87(1/2):196–240, 1990.

[BM92] Kim B. Bruce and J. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 316–327, 1992.

[Bru92] K. Bruce. The equivalence of two semantic definitions of inheritance in object-oriented languages. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics*, pages 102–124. LNCS 598, Springer-Verlag, 1992.

[Bru93a] K. Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics.

Technical Report CS-92-01, revised, Williams College, 1993. To appear in Journal of Functional Programming.

[Bru93b] K. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 285–298, 1993.

[BTCGS91] V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance and implicit coercion. *Information and Computation*, 93(1):172–221, 1991.

[Car89] L. Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989. Presented at IFIP Advanced Seminar on Formal Descriptions of Programming Concepts.

[Car92] Luca Cardelli. Typed foundations of object-oriented programming, 1992. Tutorial given at POPL '92.

[CCH+89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Functional Prog. and Computer Architecture*, pages 273–280, 1989.

[CCHO89] P. Canning, W. Cook, W. Hill, and W. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 457–467, 1989.

[CG92] P.L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 125–135, January 1990.

[CL91] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, 1991.

[CM90] L. Cardelli and J.C. Mitchell. Operations on records. In *Math. Foundations of Prog. Lang. Semantics*, pages 22–52. Springer LNCS 442, 1990.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[Ghe90] G. Ghelli. *Proof Theoretic Studies about a minimal type system integrating inclusion and parametric polymorphism.* PhD thesis, Universita di Pisa, 1990.

[Ghe92] G. Ghelli. Divergence of $F_{\leq}$ type-checking, 1992. unpublished manuscript.

[KPS93] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *20th ACM Symp. Principles of Programming Languages*, 1993.

[Mit90] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 109–124, January 1990.

[Pie92] Benjamin C. Pierce. Bounded quantification is undecidable. In *Proc 19th ACM Symp. Principles of Programming Languages*, pages 305–315, 1992.

[PT92a] Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. Technical Report ECS-LFCS-92-226, University of Edinburgh, 1992.

[PT92b] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. Technical report, University of Edinburgh, 1992.

A The Formal Definition of TOOPLE

As explained in the body of the paper, $\leq$ represents the subtype relation between types, while $\leq_{meth}$ is an ordering relating types of objects whose classes could have been defined using inheritance.

Definition A.1 *Relations of the form $\sigma \leq \tau$ and $\sigma \leq_{meth} \tau$, where σ and τ are type expressions, are said to be* type constraints. *If, moreover, t is a type variable or constant then we say $t \leq \tau$ and $t \leq_{meth} \tau$ are* simple type constraints. *If for some τ, $t \leq \tau$ or $t \leq_{meth} \tau$ are included in a set C of simple type constraints, then we say t is* declared *in C. A* restricted type constraint system *is defined as follows:*

1. *The empty sequence, ϵ, is a restricted type constraint system.*
2. *If C is a type constraint system and t and u are distinct type variables or constants such that t does not appear in C and there is no constraint of the form $u \leq_{meth} \tau$ in C, then $C;\ t \leq u$ is a restricted type constraint system.*
3. *If C is a restricted type constraint system, τ is of the form $ObjectType(MyType)\sigma$, and $t \leq_{meth} \tau$ is a simple type constraint such that t does not appear in C or τ, then $C;\ t \leq_{meth} \tau$ is a restricted type constraint system.*

We define type constraint derivations of the form $C \vdash \sigma \leq \tau$ and $C \vdash \sigma \leq_{meth} \tau$, for C a restricted type constraint system, and σ, τ type expressions, via the sets of axioms and rules given below. Note that $\tau[\sigma/t]$ denotes the expression obtained by replacing all free occurrences of variable t in τ by σ.

The following are the axioms and rules for subtypes.

$$(SRefl) \qquad C \vdash \tau \leq \tau,$$

$$(SVar) \qquad C;\ t \leq \tau \vdash t \leq \tau,$$

$$(STrans) \qquad \frac{C \vdash \gamma \leq \sigma,\ C \vdash \sigma \leq \tau}{C \vdash \gamma \leq \tau},$$

$$(S \rightarrow) \qquad \frac{C \vdash \sigma' \leq \sigma,\ C \vdash \tau \leq \tau'}{C \vdash \sigma \rightarrow \tau \leq \sigma' \rightarrow \tau'},$$

$$(SRec) \quad \frac{C \vdash \sigma_j \leq \tau_j, \quad \text{for } 1 \leq j \leq k \leq n}{C \vdash \{m_1{:}\,\sigma_1 \ldots; m_k{:}\,\tau_k; \ldots; m_n{:}\,\sigma_n\} \leq \{m_1{:}\,\tau_1; \ldots; m_k{:}\,\tau_k\}},$$

$$(SObj) \quad \frac{C;\ s \leq t \vdash \tau[s/MyType] \leq \tau'[t/MyType]}{C \vdash ObjectType(MyType)\tau \leq ObjectType(MyType)\tau'}.$$

In the $SObj$ rule, neither s nor t may occur free in C, τ, or τ'.

Definition A.2 *(from [CG92]) We write $C \vdash t \ll \tau$, if t is a type variable, and $C \vdash t \leq \tau$ is provable using only (SVar) and (STrans).*

This ordering is useful in determining the minimum type of a term. The axioms and rules for $\leq_{meth}$ are given below:

$$(MeVar) \qquad C;\ t \leq_{meth} \tau \vdash t \leq_{meth} \tau,$$

$$(MeRefl) \qquad C \vdash ObjectType(MyType)\tau \leq_{meth} ObjectType(MyType)\tau,$$

$$(MeTrans) \quad \frac{C \vdash \gamma \leq_{meth} ObjectType(MyType)\tau,\ C \vdash \tau \leq \tau'}{C \vdash \gamma \leq_{meth} ObjectType(MyType)\tau'},$$

Definition A.3 *A syntactic type assignment, E, is a finite set of the form:*

$$E = \{x_1{:}\,\tau_1, \ldots, x_n{:}\,\tau_n\}$$

with no variable x_i appearing more than once in E.

The type assignment axioms and rules for TOOPLE are given in Figures 4 and 5.

As discussed in the body of the report, there is another set of axioms and rules which can be used to derive the minimum types for terms of TOOPLE. They are given in Figures 6 and 7.

$$(Var) \qquad C, E \vdash x{:}\tau \text{ if } E(x) = \tau$$

$$(Cond) \qquad \frac{C, E \vdash B{:}Bool, \quad C, E \vdash M{:}\tau, \quad C, E \vdash N{:}\tau}{C, E \vdash if\ B\ then\ M\ else\ N{:}\tau}$$

$$(Abs) \qquad \frac{C, E \cup \{v{:}\sigma\} \vdash M{:}\tau}{C, E \vdash \lambda v{:}\sigma.M{:}\sigma \to \tau}$$

$$(Appl) \qquad \frac{C, E \vdash M{:}\sigma \to \tau,\ N{:}\sigma}{C, E \vdash M\ N{:}\tau}$$

$$(Eq?) \qquad \frac{C, E \vdash M{:}Num, \quad C, E \vdash N{:}Num}{C, E \vdash M = N{:}Bool}$$

$$(Rec) \qquad \frac{C, E \vdash e_i{:}\tau_i \text{ for all } 1 \le i \le n}{C, E \vdash \{m_1 = e_1, \ldots, m_n = e_n\}{:}\{m_1{:}\tau_1; \ldots; m_n{:}\tau_n\}}$$

$$(Proj) \qquad \frac{C, E \vdash e{:}\{m_1{:}\tau_1; \ldots; m_n{:}\tau_n\}}{C, E \vdash e.m_i{:}\tau_i \text{ for all } 1 \le i \le n}$$

$$(Class) \qquad \frac{C;\ MyType \le_{meth} ObjectType(MyType)\tau, E \cup \{self : MyType\} \vdash e{:}\tau}{C, E \vdash class(self : MyType \le_{meth} ObjectType(MyType)\tau)e{:}\ ClassType(MyType)\tau}$$

$$(New) \qquad \frac{C, E \vdash c{:}\ ClassType(MyType)\tau}{C, E \vdash new\ c{:}\ Tobj(MyType)\tau}$$

$$(Msg) \qquad \frac{C \vdash \gamma \le_{meth} ObjectType(MyType)\{m{:}\tau\}, \qquad C, E \vdash o{:}\gamma}{C, E \vdash o \Leftarrow m{:}\tau[\gamma/MyType]}$$

$$(Update) \qquad \frac{\begin{array}{c} C, E \vdash c{:}\ ClassType(MyType)\{m_1{:}\tau_1; \ldots; m_n{:}\tau_n\}, \qquad C \vdash \tau_1' \le \tau_1, \\ C;\ MyType \le_{meth} ObjectType(MyType)\{m_1{:}\tau_1'; m_2{:}\tau_2; \ldots; m_n{:}\tau_n\}, \\ E \cup \{self : MyType, super{:}\{m_1{:}\tau_1; \ldots; m_n{:}\tau_n\}\} \vdash e_1'{:}\tau_1'', \\ C;\ MyType \le_{meth} ObjectType(MyType)\{m_1{:}\tau_1'; m_2{:}\tau_2; \ldots; m_n{:}\tau_n\} \vdash \tau_1'' \le \tau_1' \end{array}}{\begin{array}{r} C, E \vdash update\ c\ by\ (self : MyType \le_{meth} ObjectType\{m_1{:}\tau_1'; \ldots; m_n{:}\tau_n\}, super) \\ \{m_1 = e_1'\}{:}\ ClassType(MyType)\{m_1{:}\tau_1'; m_2{:}\tau_2; \ldots; m_n{:}\tau_n\} \end{array}}$$

Figure 4: Type Assignment Axioms and Rules

$$(Extend)\quad \frac{\begin{array}{c} C, E \vdash c\colon ClassType(MyType)\{m_1\colon\tau_1; \ldots; m_n\colon\tau_n\}, \\ C;\ MyType \leq_{meth} ObjectType(MyType)\{m_1\colon\tau_1; \ldots; m_n\colon\tau_n; m_{n+1}\colon\tau_{n+1}\}, \\ E \cup \{self : MyType, super\colon \{m_1\colon\tau_1; \ldots; m_n\colon\tau_n\}\} \vdash e_{n+1}\colon\tau'_{n+1}, \\ C;\ MyType \leq_{meth} ObjectType(MyType)\{m_1\colon\tau_1; \ldots; m_{n+1}\colon\tau_{n+1}\} \vdash \tau'_{n+1} \leq \tau_{n+1} \end{array}}{\begin{array}{l} C, E \vdash extend\ c\ with\ (self : MyType \leq_{meth} ObjectType\{m_1\colon\tau_1; \ldots; m_n + 1\colon\tau_n + 1\}, \\ super)\{m_{n+1} = e_{n+1}\}\colon ClassType(MyType)\{m_1\colon\tau_1; \ldots; m_n\colon\tau_n; m_{n+1}\colon\tau_{n+1}\} \end{array}}$$

$$(Subsum)\quad \frac{C \vdash \sigma \leq \tau,\quad C, E \vdash e\colon\sigma}{C, E \vdash e\colon\tau}$$

Figure 5: Type Assignment Axioms and Rules (continued)

$$(MVar)\quad C, E \vdash_M x\colon\tau \text{ if } E(x) = \tau$$

$$(MCond)\quad \frac{C, E \vdash_M B\colon\rho,\quad C \vdash \rho \leq Bool,\quad C, E \vdash_M M\colon\tau',\quad C, E \vdash_M N\colon\tau''}{C, E \vdash_M if\ B\ then\ M\ else\ N\colon lub(\tau', \tau'')} \text{ if } lub(\tau', \tau'') \text{ exists}$$

$$(MAbs)\quad \frac{C, E \cup \{v\colon\sigma\} \vdash_M M\colon\tau}{C, E \vdash_M \lambda v\colon\sigma.M\colon\sigma \to \tau}$$

$$(MAppl)\quad \frac{C, E \vdash_M M\colon\sigma \to \tau,\quad C, E \vdash_M N\colon\sigma',\quad C \vdash \sigma' \leq \sigma}{C, E \vdash_M M\ N\colon\tau}$$

$$(MAppl')\quad \frac{C, E \vdash_M M\colon t,\quad C, E \vdash_M N\colon\sigma',\quad C \vdash t \ll \sigma \to \tau,\quad C \vdash \sigma' \leq \sigma}{C, E \vdash_M M\ N\colon\tau}$$

$$(MEq?)\quad \frac{C, E \vdash_M M\colon\tau,\quad C, E \vdash_M N\colon\tau',\quad C \vdash \tau \leq Num,\quad C \vdash \tau' \leq Num}{C, E \vdash_M M = N\colon Bool}$$

$$(MRec)\quad \frac{C, E \vdash_M e_i\colon\tau_i \text{ for all } 1 \leq i \leq n}{C, E \vdash_M \{m_1 = e_1, \ldots, m_n = e_n\}\colon\{m_1\colon\tau_1; \ldots; m_n\colon\tau_n\}}$$

$$(MProj)\quad \frac{C, E \vdash_M e\colon\{m_1\colon\tau_1; \ldots; m_n\colon\tau_n\}}{C, E \vdash_M e.m_i\colon\tau_i} \text{ for } i \text{ s.t. } 1 \leq i \leq n$$

Figure 6: Minimum Type Assignment Axioms and Rules

$$(MProj') \qquad \frac{C, E \vdash_M e{:}\, t, \quad C \vdash t \ll \{m_1{:}\, \tau_1; \ldots; m_n{:}\, \tau_n\}}{C, E \vdash_M e.m_i{:}\, \tau_i} \text{ for } i \text{ s.t. } 1 \leq i \leq n$$

$$(MClass) \qquad \frac{\begin{array}{c} C;\ MyType \leq_{meth} ObjectType(MyType)\tau, E \cup \{self : MyType\} \vdash_M e{:}\, \tau', \\ C;\ MyType \leq_{meth} ObjectType(MyType)\tau \vdash \tau' \leq \tau \end{array}}{C, E \vdash_M class(self : MyType \leq_{meth} ObjectType(MyType)\tau)e{:}\ ClassType(MyType)\tau}$$

$$(MNew) \qquad \frac{C, E \vdash_M c{:}\ ClassType(MyType)\tau}{C, E \vdash_M new\ c{:}\ ObjectType(MyType)\tau}$$

$$(MMsg) \qquad \frac{C, E \vdash_M o{:}\ ObjectType(MyType)\{m_1{:}\, \tau_1; \ldots; m_n{:}\, \tau_n\}}{C, E \vdash_M o \Leftarrow m_i{:}\, \tau_i[ObjectType(MyType)\{m_1{:}\, \tau_1; \ldots; m_n{:}\, \tau_n\}/MyType]}$$

$$(MMsg') \qquad \frac{C, E \vdash_M o{:}\, t, \quad t \leq_{meth} ObjectType(MyType)\{\ldots; m_i{:}\, \tau_i; \ldots\} \in C}{C, E \vdash_M o \Leftarrow m_i{:}\, \tau_i[t/MyType]}$$

$$(MUpdate) \qquad \frac{\begin{array}{c} C, E \vdash_M c{:}\ ClassType(MyType)\{m_1{:}\, \tau_1; \ldots; m_n{:}\, \tau_n\}, \qquad C \vdash \tau_1' \leq \tau_1, \\ C;\ MyType \leq_{meth} ObjectType(MyType)\{m_1{:}\, \tau_1'; m_2{:}\, \tau_2; \ldots; m_n{:}\, \tau_n\}, \\ E \cup \{self : MyType, super{:}\, \{m_1{:}\, \tau_1; \ldots; m_n{:}\, \tau_n\}\} \vdash_M e_1'{:}\, \tau_1'', \\ C;\ MyType \leq_{meth} ObjectType(MyType)\{m_1{:}\, \tau_1'; m_2{:}\, \tau_2; \ldots; m_n{:}\, \tau_n\} \vdash \tau_1'' \leq \tau_1' \end{array}}{\begin{array}{l} C, E \vdash_M update\ c\ by\ (self : MyType \leq_{meth} ObjectType(MyType)\{m_1{:}\, \tau_1'; \ldots; m_n{:}\, \tau_n\}, \\ \qquad super)\{m_1 = e_1'\}{:}\ ClassType(MyType)\{m_1{:}\, \tau_1'; m_2{:}\, \tau_2; \ldots; m_n{:}\, \tau_n\} \end{array}}$$

$$(MExtend) \qquad \frac{\begin{array}{c} C, E \vdash_M c{:}\ ClassType(MyType)\{m_1{:}\, \tau_1; \ldots; m_n{:}\, \tau_n\}, \\ C;\ MyType \leq_{meth} ObjectType(MyType)\{m_1{:}\, \tau_1; \ldots; m_n{:}\, \tau_n; m_{n+1}{:}\, \tau_{n+1}\}, \\ E \cup \{self : MyType, super{:}\, \{m_1{:}\, \tau_1; \ldots; m_n{:}\, \tau_n\}\} \vdash_M e_{n+1}{:}\, \tau_{n+1}', \\ C;\ MyType \leq_{meth} ObjectType(MyType)\{m_1{:}\, \tau_1; \ldots; m_{n+1}{:}\, \tau_{n+1}\} \vdash \tau_{n+1}' \leq \tau_{n+1} \end{array}}{\begin{array}{l} C, E \vdash_M extend\ c\ with \\ \qquad (self : MyType \leq_{meth} ObjectType(MyType)\{m_1{:}\, \tau_1; \ldots; m_{n+1}{:}\, \tau_{n+1}\}, super) \\ \qquad \{m_{n+1} = e_{n+1}\}{:}\ ClassType(MyType)\{m_1{:}\, \tau_1; \ldots; m_n{:}\, \tau_n; m_{n+1}{:}\, \tau_{n+1}\} \end{array}}$$

Figure 7: Minimum Type Assignment Axioms and Rules (continued)