J. Functional Programming **1** (1): 1–000, January 1993 © 1993 Cambridge University Press

A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics†

Kim B. Bruce
Williams College
Williamstown, MA, USA 01267
kim@cs.williams.edu

Abstract

In order to illuminate the fundamental concepts involved in object-oriented programming languages, we describe the design of TOOPL, a paradigmatic, statically-typed, functional, object-oriented programming language which supports classes, objects, methods, hidden instance variables, subtypes, and inheritance.

It has proven to be quite difficult to design such a language which has a secure type system. A particular problem with statically type checking object-oriented languages is designing type-checking rules which ensure that methods provided in a superclass will continue to be type correct when inherited in a subclass. The type-checking rules for TOOPL have this feature, enabling library suppliers to provide only the interfaces of classes with actual executable code, while still allowing users to safely create subclasses. In order to achieve greater expressibility while retaining type-safety, we choose to separate the inheritance and subtyping hierarchy in the language.

The design of TOOPL has been guided by an analysis of the semantics of the language, which is given in terms of a model of the F-bounded second-order lambda calculus with fixed points at both the element and type level. This semantics supports the language design by providing a means to prove that the type-checking rules are sound, thus guaranteeing that the language is type-safe.

While the semantics of our language is rather complex, involving fixed points at both the element and type level, we believe that this reflects the inherent complexity of the basic features of object-oriented programming languages. Particularly complex features include the implicit recursion inherent in the use of the keyword, *self*, to refer to the current object, and its corresponding type, *MyType*. The notions of subclass and inheritance introduce the greatest semantic complexities, whereas the notion of subtype is more straightforward to deal with. Our semantic investigations lead us to recommend caution in the use of inheritance, since small changes to method definitions in subclasses can result in major changes to the meanings of the other methods of the class.

† An extended abstract of this paper appeared in the Proceedings of the 20th ACM Symposium on the Principles of Programming Languages.

1 Introduction

For the last several years, object-oriented programming languages and the object-oriented style of programming have been "hot" topics in computer science, both at the theoretical and applied levels. While theoreticians have struggled to understand clearly the concepts of object-oriented languages, programmers have rushed ahead to use object-oriented programming languages because of their perceived advantages. For example, several well-known computer companies (both hardware and software) are in the midst of or have recently completed rewriting major system software (including operating systems) and applications in object-oriented languages. Unfortunately, several of these object-oriented languages have significant type-insecurities, compromising the construction of reliable, type-safe computer programs.

This paper presents the design of a statically-typed object-oriented programming language, TOOPL (for Typed Object-Oriented Programming Language), which includes the most important notions commonly associated with object-oriented programming languages. The features supported by this language include:

- static typing,
- classes,
- objects,
- methods,
- dynamic method-lookup,
- hidden instance variables,
- subtypes, and
- inheritance.

Moreover we provide keywords which allow the programmer to reference the object executing a method, *self*, its type, *MyType*, and the record of methods of its superclass, *super*.

One of the most important features of our language is that we can prove that the type system is safe. In particular, no program which type checks will ever fail with an error due to sending a message to an object that it does not understand.

One of the major selling points of object-oriented languages is the support for reusability. Because of the relative ease of modifying classes by creating subclasses, it is expected that libraries of classes will become even more important than the provision of libraries in more traditional imperative languages such as FORTRAN, Modula-2, Ada, and C.

Because library vendors often wish to retain a competitive advantage, many will prefer to provide libraries in the form of compiled code only, providing sufficient interface information (e.g., types and specifications of routines) to allow the user to make effective use of the library components. Since the original source code for the library will not be available, it is important that the user have sufficient information available to predict the behavior of subclasses derived from library classes.

What is necessary to assure the user that these classes and the subclasses derived from them behave properly? Clearly, objects created from the library classes should always meet their specifications. In particular, no run-time type errors should occur as long as messages and their associated parameters meet the given specifications. Moreover, the specifications of subclasses definable via inheritance from a library class should be derivable from the specifications of the superclass and the new code provided to implement the subclass. Another way of putting this is that any compile-time or run-time errors which arise from a subclass should be explainable from the library class specification and subclass code. In this paper, we solve a version of this problem where the specifications are restricted to type information.

While it is possible to write down a variety of type-checking rules for features of object-oriented programming languages, how can the user be assured that these type checking rules are sound? That is, if the type checker indicates that a term has type τ, how can the user be assured that the value produced by the evaluation of that term will belong to the set of elements of type τ? More specifically, how can the user be assured that type-incorrect computations will never take place; for example, that an object will never be sent a message that it doesn't understand?

The approach taken here is to provide a careful denotational semantics for the constructs of our language. In particular, the meaning of all terms of the language will be interpreted in well-understood mathematical models. This will allow us to prove that the meaning of a term belongs to the set of elements of the type assigned to it, and, in particular, an object will not be sent any messages that it doesn't understand.

While some of the constructs of object-oriented programming languages (*e.g.*, subtypes) can be interpreted in relatively straightforward ways, others, subclasses and inheritance in particular, seem to require much more complex semantics. As a result, most theoretical studies of object-oriented language features have tended to look at only isolated aspects of the languages; for example, only subtyping, or inheritance without subtyping.

The language TOOPL presented in this paper includes all of the most important features of object-oriented languages. Most of TOOPL's features should be familiar to those acquainted with object-oriented languages. One major difference from most current object-oriented languages is that we have separated the subtyping and inheritance hierarchies. This was necessary in order to provide a more expressive language, while retaining the advantages of safe type checking.

While we have succeeded in presenting relatively straightforward type-checking rules, the semantics of our language are far from straightforward. As a result, we have endeavored to keep the rest of the language as simple as possible by leaving out of the formal language definition many of the features commonly found in programming languages. Obviously, any usable programming language must contain such features (and, indeed, we use them in examples), but they can be added easily, with little or no complication to the language.

The most significant restriction in TOOPL is that it is a purely functional language. Even so, this should not be seen as a severe restriction, since we do provide for updatable instance variables (the meaning of updating an instance variable in an object will be to create a new copy of the object, with new values stored in the instance variable). More recently, with Robert van Gent (van Gent, 1993; Bruce and van Gent, 1993), we have designed an imperative version of this language which has a similar type system, and which can also be proved to be type-safe.

The language described here is first-order, providing no direct support for parametric polymorphic functions or higher-order types in the language itself. However, the semantics is given in terms of a model for the bounded, higher-order, polymorphic lambda calculus. This reflects the fact that object-oriented languages support what is sometimes called subtype polymorphism.

An alternative to providing the semantics in terms of such a model is simply to provide a translation into a variant of bounded higher-order lambda calculus, $F^{\omega}_{\leq}$, and then interpret this language in any appropriate model. This really corresponds to compiling the program in the *source* language into a *target* language consisting of the bounded higher-order lambda calculus, and then running – or more correctly, interpreting – this program. We prefer working directly with a model since it allows us to avoid some of the difficulties involving the creation of appropriate syntax for a core language which can encode the features of object-oriented languages. (See (Cardelli, 1992b; Cardelli, 1992a; Cardelli and Mitchell, 1990; Pierce and Turner, 1993) for various attempts in this direction.) We have bypassed that step here by going directly to an appropriate model for the language. Not coincidentally, though, the model we use was originally designed for the bounded higher-order lambda calculus.

It is an important open problem to find a core language based on a higher-order typed lambda calculus which allows one to express all of the features (and programs) which are expressible in TOOPL. Nevertheless, there are several advantages to our approach of bypassing this problem. One is that our semantics provides a relatively simple example of what such a core language must encompass. In fact, we hope that this semantically-based work will provide hints as to the appropriate syntax for an extension of the lambda calculus which can express all features of object-oriented languages.

Another possible advantage of our approach is that the language $F_{\leq}$, which has been used as the starting point for efforts at finding a core language, appears to be simultaneously too weak and too strong to serve as a basis for object-oriented programming languages. It is too weak in that it must be extended with recursively-defined elements, recursive types, either F-bounded quantification (Canning *et al.*, 1989) or higher-order functions from types to types, and some sort of record extension operator in order to express many of the features of object-oriented languages. An important place where the limitations of $F_{\leq}$ show up is in its inability to express polymorphic record updates in the presence of subtyping (see the discussion in Section 6 of (Bruce and Longo, 1990)). This capability seems necessary in order to express updating of instance variables in the presence of inheritance. (See (Pierce and Turner, 1993) for one approach to extending $F_{\leq}$ in order to express updates.)

A recent result of Pierce (1992) indicates that $F_{\leq}$ may be viewed as too strong (at least in its full strength) to use as a foundation for object-oriented languages. He has shown that type checking in $F_{\leq}$ is undecidable. While more recent results (Castagna and Pierce, 1994) show that it is possible to provide a rather natural (and minimal) restriction to this language whose type-checking problem is decidable, it is not yet known whether the extensions needed here cause more problems. Thus it seemed wise to avoid the problematic portions of the language while adding new parts which will increase the expressibility in order to obtain a language whose type checking problem is decidable. Nevertheless, it seemed wise at this point to take a somewhat more conservative approach. Moreover, we perform type checking directly in TOOPL, not in higher-order terms into which it might be translated. In fact, with others (see (Bruce *et al.*, 1993)), we have recently shown that if we modify TOOPL by insisting that the programmer provide a small amount of extra type information for class expressions, the type checking problem for TOOPL is decidable. We have decided not to include that extra type information here since it would only clutter our presentation.

We have chosen not to include parametric polymorphism in the core language in order to make clear that the complexity in both type checking and semantics that arises in our language comes solely from the first-order features of the language. Moreover, as discussed in Section 7, the addition of polymorphism to the language presented can be made simply and naturally. This arises from the fact that the language is interpreted in a model supporting polymorphism.

I undertook to write this paper at this time because of a conviction that the theoretical programming languages community had finally developed enough tools to fully understand and explain the most important basic notions in object-oriented programming languages. A large number of researchers have made important contributions to a number of different aspects of object-oriented languages, all of which have combined to make this paper possible. Their contributions are acknowledged in Section 9.

It may be useful to explain the process by which the language contained in this paper came to be designed. My earlier work on describing the semantics of programming languages which included only a few of the basic concepts of object-oriented languages (Bruce and Longo, 1990; Bruce, 1992) convinced me that designing a language combining all of the common features of OOL's was likely to be quite difficult and error-prone. Thus I started with the denotational model of bounded higher-order lambda calculus given in (Bruce and Mitchell, 1992), and began to integrate the semantics of subtyping given in (Bruce and Longo, 1990) with the semantics of inheritance, as explained in (Bruce, 1992) (which itself was based on the work of (Mitchell, 1990a) and (Cook *et al.*, 1990)). While the language described in this paper does not appear to be closely related to the bounded higher-order lambda calculus, the semantics are given with respect to a model of this language.

The type checking rules and semantics were checked (and corrected several times) with respect to the denotational model. Only after the language design stabilized and the appropriate soundness theorem was proved, did I go on to add such features as references to superclasses, which turned out to be easy, and instance variables, which turned out to be more difficult. The existence of a mathematical model for the semantics of the language made it relatively straightforward, though somewhat tedious, to check the correctness of the type checking rules and formal semantics. While language designers who do not use formal tools of this nature may be surprised later by counter-examples to type safety, in this circumstance we actually prove the type safety of the language.

In section 4.1.3 we discuss in some detail the problems with type-checking object-oriented languages, and some of the compromises made by existing languages. One of the biggest problems with existing languages is the identification of the subtype and subclass hierarchies. This forces language designers to choose between type-safety and expressibility in statically-typed languages. Languages like C++, Beta, and Eiffel choose to compromise the safety of their type systems in order to achieve greater expressibility, while others, such as Trellis / Owl and Sather sacrifice expressibility in order to achieve type-safety. (See 4.1.3 for references for these languages.)

In TOOPL we have retained both expressibility and type-safety by separating the subtype and subclass hierarchies. Moreover, this separation leads to opportunities for even greater flexibility in defining and using classes and objects.

I have attempted to organize this paper to be more understandable to the reader by introducing features gradually. We begin with a very brief overview of the main features of object-oriented programming languages in Section 2, and then begin to describe a succession of increasingly complicated languages. We begin in Section 3 with a rudimentary object-oriented language (ROOPL) which includes classes, objects, methods, and dynamic method lookup, but no subtyping, subclasses, or instance variables. In Section 4 we add subtyping and subclasses to our language to get our next approximation, SOOPL (for Simple Object-Oriented Programming Language), of our desired language. Finally in Section 5, we introduce the full language TOOPL with instance variables. In each of these sections we begin with an informal discussion of the new language features, followed by a presentation of the formal syntax and type-checking rules for the language. We conclude each section with a description of the denotational semantics of the language and a proof of the soundness of type checking with respect to the denotational semantics. While the formal denotational semantics are the most difficult part of each section, it is hoped that the accompanying informal discussion of semantics will help the reader through these portions of the paper.

In Section 6 we present several sample programs illustrating the use of various features of the language. The reader may wish to peek ahead to this section in order to see some more motivating examples early on. In Section 7 we discuss possible additions to the language, such as polymorphism. In Section 8, we summarize our contributions and raise some concerns about what seems to be the inherent complexity of some of the basic features of object-oriented programming language. In particular, we offer strong cautions on the excessive reliance on inheritance in object-oriented languages. Finally, in Section 9 we discuss the relevance of other research in object-oriented languages to this paper.

Though we provide definitions below, we presume that the reader has some familiarity with the concepts of object-oriented programming languages. General sources of information on object-oriented languages include (Meyer, 1988; Cox, 1986; Goldberg and Robson, 1983). General background on type-theoretic issues for object-oriented languages can be found in (Danforth and Tomlinson, 1988; Cardelli, 1988a; Cook *et al.*, 1990).

We also assume that the reader is familiar with lambda notation and the standard notations for representing type checking rules and denotational semantics. See (Gunter, 1992) or (Mitchell, 1990b), for instance. It is also helpful for the more technical sections on semantics to be familiar with the second-order (polymorphic) bounded lambda calculus and the extension to so-called "F-bounded" polymorphism. The former language was first introduced in (Cardelli and Wegner, 1985), while the latter was introduced in (Canning *et al.*, 1989). However, the non-technical sections of this paper should be accessible to those without this background.

2 Terminology for object-oriented features

The language described in this paper offers full support for object-oriented features including objects, classes, methods, instance variables, dynamic method invocation, subclasses, and subtypes. Moreover, we provide mechanisms to allow the programmer to refer to the current object, its type, and the record of methods of its superclass. These concepts are described briefly below. Later sections of the paper go into more detail.

Objects consist 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. (Some restrictions on the modification of the types of methods and instance variables in subclasses are necessary in order to preserve type safety).

All terms of the language, including both classes and objects, have associated types. In the language described in this paper, instance variables are not visible outside of objects. The instance variables in classes are visible, however, since subclasses often need this information. Since types describe the public interfaces of terms, object types do not mention the types of instance variables, while class types do.

We say type T is a *subtype* of U if a value of type T can be used in any context in which a value of type U is expected. Note that subtyping depends only on the types of values, while inheritance depends upon their implementations. We give examples later that show that if one class inherits from another, the type of the objects generated by the derived class is not necessarily a subtype of the type of the objects generated by the original class.

Virtually all object-oriented languages provide programmers with a mechanism for sending a message to an object from inside one of its own methods. We will use a bound variable, usually written as *self*, as a name for the current object. Since our language is statically typed, it will be necessary to assign a type to all occurrences of *self*. While one could simply give *self* the same type as the object being defined, complications involving subclasses lead us to find a different solution. The problem is that when a method whose body involves *self* is inherited in a subclass, *self* will now refer to objects generated from the subclass rather than the original class. Because the meaning of *self* will change in subclasses, its type will change as well. Thus we will use a bound variable, usually written as *MyType*, as the type of *self*.

Finally, when new definitions are given to methods in a subclass, it is useful to be able to refer to the methods of the superclass. For instance, one often wishes to apply the method body from the superclass and then perform a few more operations before returning from the redefined method. We provide a bound variable, usually written as *super*, to refer to the record of methods of the superclass.

We will see later that the inclusion of hidden instance variables will add several complications to our language. Rather than describing these here, we will put the description off until later.

3 ROOPL: A Rudimentary Object Oriented Programming Language

In this section, we discuss a very simple initial approximation of our object-oriented language, called ROOPL (for Rudimentary Object-Oriented Programming Language). This first approximation includes *objects*, *methods*, and *classes*, but important features like subclasses, inheritance, and subtypes will be postponed to Section 4, and the important concept of instance variable will be deferred until Section 5. Our hope is that an introduction via this greatly simplified language will make it easier to understand the more complex versions which will be introduced later.

3.1 Classes, objects, methods, and dynamic method lookup

In this subsection we provide an informal description of the important concepts of classes, objects, and methods as they are used in ROOPL. The following subsection provides a formal definition of the language.

Classes provide templates for the creation of objects. A typical class is written in ROOPL as

$$class\ (self\colon MyType)\{m_1 = e_1, \ldots, m_n = e_n\},$$

where the m_i are the names of its methods and the e_i are terms representing the bodies of the methods. (We explain the role of "*self*: *MyType*" in the notation later.)

Classes in ROOPL are *not* types, but instead represent values. While this is not common in object-oriented programming languages, it is a natural decision since classes describe values rather than interfaces. In particular, classes provide the bodies of methods (and, in the full language, will eventually provide initial values for instance variables). As with all other terms, classes have types, which specify their public interfaces; in particular, the types of their methods. In the above example, if each e_i has type τ_i then the type of the above class is

$$ClassType\ (MyType)\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}.$$

If c is a class, then

$$new\, c$$

denotes a new object which responds to messages by evaluating the corresponding method body in the definition of the class c. If the term c has type of the form $ClassType(MyType)\{m_1{:}\,\tau_1; \ldots; m_n{:}\,\tau_n\}$, then the type of the object created by $new\, c$ will be written as

$$ObjectType(MyType)\{m_1{:}\,\tau_1; \ldots; m_n{:}\,\tau_n\}.$$

The essential difference between classes and objects is that objects can be the targets of messages, while classes can only be instantiated to objects (through the *new* command) or modified via inheritance as described later.

If o is an object which has a method with name m, then we write

$$o \Leftarrow m$$

for the results of evaluating the method m on o (some languages use the notation $o.m$ instead). This is often described as "sending message m to object o." The bound variable, *self*, which occurs in the heading of a class definition can be used in the body of a method to stand for the object to which the corresponding message has been sent. C++ uses the notation *this*, while Eiffel uses *Current* to refer to this object. For example if, in the body of a method m, one wishes to send the message m' to the current object, one writes $self \Leftarrow m'$. The bound type variable, *MyType*, which occurs in class definitions and in *ClassType* and *ObjectType* definitions stands for the type of the current object (*i.e.*, the type of the corresponding *self*). Eiffel would use the notation *like Current* to refer to the type of the current object.

The following simple example of a class should help clarify these ideas. Let

$$\begin{aligned}
PointClass = class\ \ (\ & self{:}\, MyType)\\
\{\ & x = 47,\\
& y = 13,\\
& eq = fun(p{:}\, MyType)([(p \Leftarrow x) = (self \Leftarrow x)]\ \&\\
& \qquad\qquad\qquad [(p \Leftarrow y) = (self \Leftarrow y)])\},
\end{aligned}$$

where a term of the form $fun(x{:}\,\tau)e$ represents a function with body e which takes a parameter x of type τ.

The type of $PointClass$ is

$$PointClassType = ClassType(MyType)\{x{:}\, Num; y{:}\, Num; eq{:}\, MyType \rightarrow Bool\}$$

where a type of the form $A \rightarrow B$ represents the type consisting of functions from A to B.

Let

$$MyPoint = new\ PointClass.$$

Then $MyPoint$ is an object with type

$$PointType = ObjectType(MyType)\{x{:}\, Num; y{:}\, Num; eq{:}\, MyType \rightarrow Bool\}.$$

In particular, $MyPoint \Leftarrow eq$ is a function with type $PointType \rightarrow Bool$. That is, it takes an argument whose type is the same as that of $MyPoint$ (namely $PointType$), returning either true or false depending on whether its x and y components are equal to those of MyPoint. Note that when a message is sent to an object, each occurrence of the bound variable $MyType$ in the type of the method is replaced by the actual type of the object to which the message is sent.

At this point the reader may wonder why we didn't just write $PointType$ in place of $MyType$ in the type of eq in $PointType$. One reason is that we would like to give the type of eq without having to give a name to the entire object type. However, a more important reason is that we will be modifying classes by inheritance. When we do so, the meaning of $MyType$ will depend on whatever modifications we have made to the type. It will turn out to be best to think of $ObjectType(MyType)$ as a kind of fixed point operator over $MyType$. Thus $MyType$ (along with *self*) will be treated as a bound variable in class and type definitions.

As a consequence of our definitions, there may be many classes with the same class type, and objects of a particular object type may have been generated by several different classes (though all of the generating classes must have had the same class type). Because most object-oriented programming languages identify classes with types, it is not possible to have objects of the same type generated by distinct classes, even though this is very natural. Later, for instance, when we introduce instance variables, we will see that we can have objects of type *Point* which are generated by classes using both Cartesian and polar representations of points.

Suppose classes c and c' both generate objects of the same type τ. If an object o of type τ is sent the message m, how do we determine whether the method m from c or from c' is invoked? The answer to this question is at the heart of the object-oriented style of programming. The key is that each object keeps track of its own methods. That is, if o was generated from class c, then $o \Leftarrow m$ will result in the execution of the body of the method m in c, while if o was generated from c', it will execute the body of m from c'. The introduction of subtypes in Section 4 makes this *dynamic method lookup* (sometimes called *dynamic method invocation*) even more powerful.

Because we have not yet introduced instance variables, all objects generated from the same class are identical. Of course we may define parameterized classes for added flexibility. For example, $C\colon Num \rightarrow Num \rightarrow PointClassType$ could be defined so that $new\ (C(a)(b))$ results in an object whose x and y components are a and b. In Section 5 we will add updatable instance variables to provide a more realistic language.

We start with this very simple language because it allows us to present a fairly straightforward set of type checking rules and semantics in the next section in order to get the reader used to our notation. Both type checking rules and semantics become gradually more complex as we add subtyping, subclasses, and instance variables.

3.2 Syntax and type checking for ROOPL

In this section we provide a formal definition of the language ROOPL introduced in the previous section. The definition includes the syntax of types and terms, and the formal type checking rules. In the next section we present the formal semantics of ROOPL. The definitions in these sections should be understood as preliminary, as we will replace them by more complex type checking rules and semantics when we introduce more features into the language.

3.2.1 Type and term expressions

We begin with the formal definition of the types of our language.

Definition 3.1

Let $\mathcal{V}^{Tp}$ be an infinite collection of type variables, $\mathcal{L}$ be an infinite collection of labels, and $\mathcal{C}^{Tp}$ be a collection of type constants which includes at least the type constants *Bool* and Num. The type expressions with respect to $\mathcal{V}^{Tp}$ and $\mathcal{C}^{Tp}$ are defined by the following production:

$$\begin{aligned}\tau ::= \; & c \mid t \mid \tau \rightarrow \tau' \mid \{m_1 \colon \tau_1; \ldots; m_n \colon \tau_n\} \mid \\ & ObjectType(MyType)\tau \mid ClassType(MyType)\tau.\end{aligned}$$

In the above grammar, $c \in \mathcal{C}^{Tp}$ and $t \in \mathcal{V}^{Tp}$. Moreover, the τ's appearing in ObjectType and ClassType definitions must be record types (*i.e.*, of the form $\{m_1 \colon \tau_1; \ldots; m_n \colon \tau_n\}$).

As mentioned earlier, types of the form $\sigma \rightarrow \tau$ denote functions from σ to τ. Also, as described earlier, the bound variable *MyType* occurring in the method types of object and class types stands for the type of the object which is executing the method. We will consider two type expressions to be the same if they are the same up to renaming of bound variables and reordering of record components.

The terms of ROOPL include the usual programming language constructs as well as the object-oriented features that were described in the previous section. The following context-free grammar defines the pre-terms of the language. A pre-term will be considered a term only if it can be type checked with respect to a given assignment of types to free variables.

Definition 3.2

The pre-terms of ROOPL are given by the following production:

$$\begin{aligned}M ::= \; & x \mid \mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ N \mid \mathit{fun}(v \colon \sigma)M \mid M\ N \mid M = N \mid R.m_i \mid \\ & \{m_1 = M_1, \ldots, m_n = M_n\} \mid \mathit{class}(\mathit{self} \colon MyType)R \mid \mathit{new}\ c \mid o \Leftarrow m.\end{aligned}$$

In the above grammar, B, M, N, R, M_i, c, and o all represent pre-terms. M, N, and the M_i are intended to suggest general pre-terms, B a Boolean expression, R a record, c a class, and o an object. The m_i are labels, while σ is a type.

3.2.2 Type checking rules

The type-checking rules of the language are given with respect to a kind of environment which indicates the types of free variables which may occur in terms.

Definition 3.3

A syntactic type assignment, E, is a finite set of the form $\{x_1 \colon \tau_1, ..., x_n \colon \tau_n\}$, where each τ_i is a type expression, and in which no variable x_i appears more than once in E. We write $E(x) = \tau$ if $x \colon \tau \in E$.

A formula of the form $E \vdash M \colon \sigma$, for E a syntactic type assignment, M a term of our language, and σ a type, is said to be a type assignment. Its intuitive meaning is that M has type σ under the type assignment, E. The axioms and rules for deriving type assignments are given in Figure 1. A rule with name *Rname* will be written in the form

$$Rname \qquad \frac{A\ldots}{B\ldots}$$

where $A\ldots$ is the hypothesis and $B\ldots$ is the conclusion of the rule. We say that M is a term of ROOPL with respect to syntactic type assignment E if there exists a type τ such that $E \vdash M \colon \tau$.

Most of the type checking rules for ROOPL should be familiar to those who have worked with the typed lambda calculus. For example, the rule *Abs* indicates that in order to show a term of the form $fun(v \colon \sigma)M$ has type $\sigma \to \tau$, it is sufficient to show that the body M has type τ under the extra assumption that the formal parameter v has type σ. We describe the intuition behind the type-checking rules for object-oriented terms here.

The rule *Object* formalizes the fact indicated earlier that when *new* is applied to a class, c, of type $ClassType(MyType)\tau$, it results in an object with type $ObjectType(MyType)\tau$.

Because *MyType* represents the type of *self*, a class whose intended type is $ClassType(MyType)\tau$ should be type checked under the assumption that *self* has type $ObjectType(MyType)\tau$, the type of objects generated from that class. This is exactly the content of rule *Class*. Note that the notation $M[u/t]$, which occurs in this typing rule, denotes the term obtained from M by replacing all free occurrences of t in M by u.

Finally the rule *Message* just says that the type of a message-passing expression is the type of the method, with *MyType* replaced by the type of the object. This corresponds to the intuition that the meaning of *MyType* is to be the type of the receiving object.

We present these relatively straight-forward rules here in order to make it easier for the reader to understand the somewhat more complex rules that will be necessary when we add inheritance to the language.

Var $$E \cup \{x\colon \tau\} \vdash x\colon \tau$$

Cond $$\frac{E \vdash B\colon Bool,\ E \vdash M\colon \tau,\quad E \vdash N\colon \tau}{E \vdash \mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ N\colon \tau}$$

Abs $$\frac{E \cup \{v\colon \sigma\} \vdash M\colon \tau}{E \vdash fun(v\colon \sigma)M\colon \sigma \to \tau}$$

Appl $$\frac{E \vdash M\colon \sigma \to \tau,\quad E \vdash N\colon \sigma}{E \vdash M\ N\colon \tau}$$

Eq $$\frac{E \vdash M\colon Num,\quad E \vdash N\colon Num}{E \vdash M = N\colon Bool}$$

Record $$\frac{E \vdash M_i\colon \tau_i \ \text{ for all } 1 \leq i \leq n}{E \vdash \{m_1 = M_1, \ldots, m_n = M_n\}\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}}$$

Proj $$\frac{E \vdash R\colon \{m_1\colon \tau_1; \ldots m_n\colon \tau_n\}}{E \vdash R.m_i\colon \tau_i \ \text{ for all } 1 \leq i \leq n}$$

Class $$\frac{E \cup \{\mathit{self}\colon \mathit{ObjectType}(\mathit{MyType})\tau\} \vdash (R\colon \tau)[\mathit{ObjectType}(\mathit{MyType})\tau / \mathit{MyType}]}{E \vdash \mathit{class}(\mathit{self}\colon \mathit{MyType})R\colon \mathit{ClassType}(\mathit{MyType})\tau}$$

Object $$\frac{E \vdash c\colon \mathit{ClassType}(\mathit{MyType})\tau}{E \vdash \mathit{new}\ c\colon \mathit{ObjectType}(\mathit{MyType})\tau}$$

Message $$\frac{E \vdash o\colon \mathit{ObjectType}(\mathit{MyType})\{m_1\colon \tau_1; \ldots m_n\colon \tau_n\}}{E \vdash o \Leftarrow m_i\colon \tau_i[\mathit{ObjectType}(\mathit{MyType})\{m_1\colon \tau_1; \ldots m_n\colon \tau_n\}/\mathit{MyType}]}$$

Fig. 1. Type Assignment Axioms and Rules for ROOPL

3.3 Semantics

While we present the semantics of ROOPL using a formal denotational model, we will also discuss informally the meanings of terms. The reader who is not familiar with formal denotational semantics may skim the formal definitions and go on to the intuitive discussions of the meanings of terms. However, we urge readers to make the effort to understand this section as it will provide a deeper understanding of the constructs introduced so far.

The semantics of terms of ROOPL are given with respect to a model, $\mathcal{A}$, of the F-bounded second-order polymorphic lambda calculus with recursive types and elements. This model contains denotations of functions which take types as parameters as well as recursively-defined types and terms. The need to support these features will be seen in the formal description of the semantics of terms of ROOPL. The reason for including models of "F-boundedness" will not be apparent until after we have introduced subtyping.

$[\![c]\!]\rho = \mathcal{A}_c$ for $c \in \mathcal{C}^{Tp}$.

$[\![t]\!]\rho = \rho(t)$ for $t \in \mathcal{V}^{Tp}$.

$[\![A \rightarrow B]\!]\rho = [\![A]\!]\rho \rightarrow [\![B]\!]\rho$.

$[\![\{m_1 : \tau_1, \ldots, m_n : \tau_n\}]\!]\rho = \prod_{m_i \in \{m_1,\ldots,m_n\}} [\![\tau_i]\!]\rho$.

$[\![ObjectType(MyType)\tau]\!]\rho = FIX(\lambda\xi.[\![\tau]\!]\rho[\xi/MyType])$.

$[\![ClassType(MyType)\tau]\!]\rho = \xi \rightarrow [\![\tau]\!]\rho[\xi/MyType]$,
where $\xi = [\![ObjectType(MyType)\tau]\!]\rho$.

Fig. 2. The Semantics of Type Expressions in ROOPL.

It can be shown that the existence of such a model is sufficient to show that all typed expressions of ROOPL have interpretations in $\mathcal{A}$. If ξ is the interpretation or meaning of a type expression, σ, then $\mathcal{A}^\xi$ denotes the set of elements of that type. If $c \in \mathcal{C}^{Tp}$ then $\mathcal{A}_c$ denotes the interpretation of c in $\mathcal{A}$. For simplicity, we will assume that $\mathcal{A}^{\sigma \rightarrow \tau} = \mathcal{A}^\sigma \rightarrow \mathcal{A}^\tau$ and that $\mathcal{A}^{Bool} = \{true, false, \perp\}$. It is straightforward to replace this by the more complex (but more flexible) definition given in (Bruce *et al.*, 1990) and (Bruce and Longo, 1990) in case these two assumptions do not hold.

In the following, $\prod_{i \in S} T_i$ denotes the set of all functions, f, with domain, S, such that for all $i \in S$, $f(i) \in T_i$.

The meanings of types are given with respect to a type environment, ρ, which maps type variables to types in the model, $\mathcal{A}$. We write $[\![\tau]\!]\rho$ for the meaning of τ with respect to environment ρ. The expression $\rho[\xi/t]$ denotes the environment which is identical to ρ except that $(\rho[\xi/t])(t) = \xi$. The definition of the semantics of types is given in Figure 2.

The meanings of types are sometimes better understood in conjunction with the meanings of terms of that type. Nevertheless, we discuss the meanings of types first, with the interpretations of terms following. The reader should feel free to go back and forth between terms and their corresponding types in order to gain the most complete understanding of their meanings.

The meanings of most non-object-oriented types should be familiar. Records are interpreted as functions with domain the set of labels for the record fields. These functions should take each label of the record to an element of the meaning of the associated type for that label.

Object types are defined as fixed points. Recall that if $F: T \rightarrow T$, then FIX(F) is an element t_F such that $F(t_F) = t_F$. By the semantic definition given above, $ObjectType(MyType)\tau$ is interpreted as a recursively-defined record type, τ, in which every occurrence of *MyType* is replaced by $ObjectType(MyType)\tau$. Another way of writing this is

$$[\![ObjectType(MyType)\tau]\!]\rho = [\![\tau]\!]\rho[[\![ObjectType(MyType)\tau]\!]\rho/MyType].$$

This corresponds to our intuition that *MyType* stands for the object type itself.

Class types are the most difficult to make sense of at this point, though the meaning should come clearer when we discuss the interpretation of class terms. Briefly, classes are interpreted as functions which take a parameter which will provide the interpretation for *self* and return the record of corresponding methods. Thus the meaning of type $ClassType(MyType)\tau$ will be a function type with domain $[\![ObjectType(MyType)\tau]\!]\rho$ and range $[\![\tau]\!]\rho[[\![ObjectType(MyType)\tau]\!]\rho/MyType]$.

Of course, by above, we can rewrite the range as $[\![ObjectType(MyType)\tau]\!]\rho$. We write the range in the longer, more awkward form because our more complex definition when we add inheritance is most easily seen as a modification of this form. Understanding the meaning of classes as functions with both domain and range $[\![ObjectType(MyType)\tau]\!]\rho$ is also useful, since objects will be formed as fixed points of classes, and we need the domain and range to be the same in order to have fixed points.

The definition of the meaning of terms is given with respect to semantic environments which assign types to type variables as well as values to term variables.

Definition 3.4
Let ρ be a semantic environment mapping type variables to types and regular variables to elements of $\mathcal{A}$. We say ρ is *consistent* with respect to E, if for all x in the domain of E, if $E(x) = \sigma$ then $\rho(x) \in [\![\sigma]\!]\rho$.

Each clause of the definition of the semantics of terms in Figure 3 corresponds to a type assignment axiom or rule from the previous subsection. We write $[\![E \vdash e\colon \tau]\!]\rho$ for the meaning of e with respect to environment ρ.

The meanings of most terms should be familiar to those with previous exposure to denotational semantics. We note again that the meanings of records are functions and that field extraction corresponds to applying the function representing the meaning of the record to the appropriate label.

As suggested above in the definition of the meaning of class types, the meaning of a class term is a function which takes a parameter of its corresponding object type. This parameter is used as the meaning of *self* in the interpretation of the methods of the class.

The meaning of *new c* is obtained by taking a fixed point of this function. Another way of putting this is that if $o = [\![E \vdash new\ c\colon ObjectType(MyType)\tau]\!]\rho$, then $o = ([\![E \vdash c\colon ClassType\ (MyType)\tau]\!]\rho)(o)$. To understand this in more detail, suppose $c =_{def} class(self\colon MyType)e\colon ClassType(MyType)\tau$. By the semantics of classes,

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

where $\xi = [\![ObjectType(MyType)\tau]\!]\rho$. Thus

$$\begin{array}{lll} o & =_{def} & [\![E \vdash new\ c\colon ObjectType(MyType)\tau]\!]\rho \\ & = & ([\![E \vdash c\colon ClassType\ (MyType)\tau]\!]\rho)(o) \\ & = & [\![E \cup \{self\colon MyType\} \vdash e\colon \tau]\!]\rho[\xi/MyType, o/self] \end{array}$$

Thus the meaning of *new c* is the meaning of its record of methods where *MyType* is interpreted as the corresponding object type, and *self* is interpreted as the object as a whole. This is exactly the intuition that we were attempting to model.

$[\![E \vdash x\colon \tau]\!]\rho = \rho(x).$

$$[\![E \vdash \mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ N\colon \tau]\!]\rho = \begin{cases} [\![E \vdash M\colon \tau]\!]\rho, & \text{if } [\![E \vdash B\colon \mathit{Bool}]\!]\rho, \\ [\![E \vdash N\colon \tau]\!]\rho, & \text{if not } [\![E \vdash B\colon \mathit{Bool}]\!]\rho, \\ \bot, & \text{otherwise.} \end{cases}$$

$[\![E \vdash \mathit{fun}(v\colon \sigma)\ M\colon \sigma \to \tau]\!]\rho = \lambda d \in \mathcal{A}^{[\![\sigma]\!]\rho}.[\![E \vdash M\colon \tau]\!]\rho[d/v].$

$[\![E \vdash MN\colon \tau]\!]\rho = ([\![E \vdash M\colon \sigma \to \tau]\!]\rho)([\![E \vdash N\colon \sigma]\!]\rho).$

$$[\![E \vdash M = N\colon \mathit{Bool}]\!]\rho = \begin{cases} \bot, & \text{if } [\![E \vdash M\colon \tau]\!]\rho = \bot \text{ or } [\![E \vdash N\colon \tau]\!]\rho = \bot, \\ \mathit{true}, & \text{if } [\![E \vdash M\colon \tau]\!]\rho = [\![E \vdash N\colon \tau]\!]\rho \neq \bot, \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$[\![E \vdash \{m_1 = M_1, \ldots, m_n = M_n\}\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}]\!]\rho = f,$

where $\mathit{dom}(f) = \{m_1, \ldots, m_n\}$ and for $1 \leq i \leq n, f(m_i) = [\![E \vdash M_i\colon \tau_i]\!]\rho.$

$[\![E \vdash R.m_i\colon \tau_i]\!]\rho = ([\![E \vdash R\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}]\!]\rho)\ (m_i).$

$[\![E \vdash \mathit{class}(\mathit{self}\colon \mathit{MyType})e\colon \mathit{ClassType}(\mathit{MyType})\tau]\!]\rho =$
$\lambda o \in \mathcal{A}^{\xi}.[\![E \cup \{\mathit{self}\colon \mathit{MyType}\} \vdash e\colon \tau]\!]\rho[\xi/\mathit{MyType}, o/\mathit{self}],$
where $\xi = [\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho.$

$[\![E \vdash \mathit{new}\ c\colon \mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho = \mathit{Fix}([\![E \vdash c\colon \mathit{ClassType}\ (\mathit{MyType})\tau]\!]\rho).$

$[\![E \vdash o \Leftarrow m\colon \tau[\gamma/\mathit{MyType}]]\!]\rho = ([\![E \vdash o\colon \gamma]\!]\rho)\ (m).$

Fig. 3. The Semantics of terms in ROOPL.

In this simple language, the meaning of message passing is just field extraction from the recursively-defined record corresponding to the meaning of the object. The addition of inheritance and subtypes will not have much effect on the meaning of message passing, but the inclusion of instance variables will make message passing significantly more complex.

It is relatively straightforward to show by induction that if $E \vdash e\colon \tau$ and ρ is consistent with E then $[\![E \vdash e\colon \tau]\!]\rho \in \mathcal{A}^{[\![\tau]\!]\rho}$. A similar theorem will be proved in the next section about the language SOOPL.

4 SOOPL: Adding subtypes and inheritance to ROOPL

In this section we introduce the more complex notions of subtype, subclass and inheritance, which will be added to our language. We draw careful distinction between the notions of subclass and subtype (as suggested in (Cook *et al.*, 1990)). While this distinction may at first seem to add extra complexity to the language, the confusion between these concepts in most existing object-oriented languages has resulted either in unnecessary restrictions on expressibility or to type insecurities in the languages. The notion of subtype has only to do with the interface (type) of objects, while that of subclass has to do with the implementation of their defining classes.

4.1 An informal introduction to subtypes and inheritance

Before providing a formal definition for our next language, SOOPL, and its semantics, we provide an intuitive description of subtyping and inheritance. We also provide a brief description of the difficulties in type checking which arise from the addition of these features.

4.1.1 Subtypes

The intuition behind subtypes is that one type is a subtype of another if the first is a specialization or refinement of the second. Operationally, this means that any object of the first type can be used in any context which expects an object of the second type. In terms of type-checking, this implies that any value of the first type can be treated as a value of the second type.

This is made explicit, for example, in (Bruce and Longo, 1990) and (Breazu-Tannen *et al.*, 1991) by requiring the existence of a well-behaved "coercion" function which takes values of the first type to the second. Thus we will treat the relation "σ is a subtype of τ" (abbreviated as $\sigma \leq \tau$) as asserting the existence of a well-behaved coercion function from elements of type σ to elements of type τ. It will not be necessary to define what a "well-behaved coercion function" is here, as we will simply use this notion to argue for the intuitive correctness of our subtyping rules. See (Bruce and Longo, 1990) or (Breazu-Tannen *et al.*, 1991) for details on these coercion functions.

The subtype relation between types is reflexive and transitive. If σ is a subtype of τ, we say that τ is a supertype of σ. A language may have non-trivial subtype relations defined between base types. For instance, Integer may be a subtype of Float.

For record and function types, rules for determining subtypes are relatively straightforward. For objects, the rules are more complex due to the implicit mutual recursion in the definition of objects which arises through the use of *self.*

Record types present the most interesting examples of subtypes. A record type can be a specialization of another record type in two ways. One way is if the fields of the first are a superset of those of the second. For example, suppose

$$PersonType = \{Name\colon string;\, Address\colon string\}$$

and

$$StudentType = \{Name\colon string;\, Address\colon string;\, YearOfGrad\colon integer\}.$$

Clearly any record of type *StudentType* can be understood as a subtype of *PersonType* by simply ignoring the fact that it has a field, *YearOfGrad.* More formally, a coercion function between the two types takes a value of type *StudentType* and simply returns the record consisting of the *Name* and *Address* fields of the original value. Thus $StudentType \leq PersonType$.

One record type can also be a subtype of another record type if the types of one or more of the second type's components are replaced by subtypes in the first record type. For instance if

$$FriendType = \{Who\colon PersonType; HowLong\colon integer\}$$

and

$$StudentFriendType = \{Who\colon StudentType; HowLong\colon integer\},$$

we can define a coercion function from *StudentFriendType* to *FriendType* which coerces the *Who* component from type *StudentType* to type *PersonType*, leaving the *HowLong* component unchanged. Thus $StudentFriendType \leq FriendType$.

These two mechanisms for constructing subtypes of records can be combined into one rule which can be found in Section 4.2.1.

The subtyping rules for function spaces are a bit more complex. If σ and τ are types, let $\sigma \to \tau$ denote the type of functions from type σ to type τ. If $\tau \leq \tau'$, then $\sigma \to \tau \leq \sigma \to \tau'$ since any function of type $\sigma \to \tau$ can be coerced to be a function of type $\sigma \to \tau'$ by applying the function to a value of type σ and then coercing the result (which is of type τ) to a value of type τ'. More explicitly, if $C_{\tau,\tau'}\colon \tau \to \tau'$ is a coercion function from τ to τ', define $C_{\sigma\to\tau,\sigma\to\tau'}(f) = C_{\tau,\tau'} \circ f$.

Modifying the domain of a function is a bit trickier. If $\sigma' \leq \sigma$ then $\sigma \to \tau \leq \sigma' \to \tau$, the reverse of what might be expected. This can be seen as follows. Suppose f is a function of type $\sigma \to \tau$. Then f can be coerced to be a function from σ' to τ as follows. To apply f to a value of type σ', first coerce the value of type σ' to a value of type σ, and then apply f. Using the same notation as above, if $C_{\sigma',\sigma}\colon \sigma' \to \sigma$ is a coercion function from σ' to σ, define $C_{\sigma\to\tau,\sigma'\to\tau}(f) = f \circ C_{\sigma',\sigma}$.

Adopting terminology from category theory, we say that subtyping for function spaces is *covariant* in the range (*i.e.*, increasing the range type results in increasing the resulting function space type), but *contravariant* in the domain (*i.e.*, increasing the domain type results in decreasing the resulting function space type).

In many ways, an object is like a record. For instance, the methods of an object can be accessed in a way similar to the fields of a record. Thus we might expect that $ObjectType(MyType)\sigma \leq ObjectType(MyType)\tau$ if $\sigma \leq \tau$. Unfortunately, however, this straightforward rule does not hold for object types. The reason that this rule fails has to do with the recursion implicit in the definitions of *self* and its type, *MyType*.

We use the point example from Section 3.1 to illustrate the problem. Let

$$PointType = ObjectType(MyType)\{x\colon Num; y\colon Num; eq\colon MyType \to Bool\}.$$

and

$$\begin{aligned} ColorPointType = {} & ObjectType(MyType) \\ & \{x\colon Num; y\colon Num; color\colon ColorType; eq\colon MyType \to Bool\}. \end{aligned}$$

The only difference between the two types is that *ColorPointType* has an extra *color* method not found in *PointType*. The types of the shared x, y, and *eq* methods are exactly the same. If these were records (*i.e.*, if the *ObjectType*(*MyType*) were removed from the beginning of the type definition) they would certainly be subtypes. However, we can show easily that *ColorPointType* is not a subtype of

PointType by writing a piece of code that works correctly for points but not color points.

The function, *PointFunc*, is defined as:

$$PointFunc = fun(p\colon PointType)\ (p \Leftarrow eq(new\ PointClass)).$$

PointFunc is a function of type $PointType \rightarrow Bool$ which determines whether its parameter is equal to a new point object created from *PointClass*. However, if we try to use a parameter of type *ColorPointType* in a call of *PointFunc* we get a type error.

The problem is that if *cp* is an object expression of type *ColorPointType*, the expression $cp \Leftarrow eq$ has type $ColorPointType \rightarrow Bool$. Yet in the body of *PointFunc*, $cp \Leftarrow eq$ is applied to *new PointClass* which has type *PointType*, rather than *ColorPointType*. This type error corresponds to problems that one might expect in the body of the *eq* method of *ColorPointType*. For instance, if cp' is the formal parameter of the method *eq* in a color point, then a test of the form $(self \Leftarrow color) = (cp' \Leftarrow color)$ is likely to occur in the body of that method. Needless to say, if the actual parameter corresponding to cp' is only a point then $cp' \Leftarrow color$ will be undefined.

The source of the problem is the use of *MyType* in the definition of the type of objects. Let p be an object of type *PointType* and *cp* be an object of type *ColorPointType*. Even though the *eq* methods of *PointType* and *ColorPointType* are both $MyType \rightarrow Bool$, the types of $p \Leftarrow eq$ and $cp \Leftarrow eq$ are, respectively, $PointType \rightarrow Bool$ and $ColorPointType \rightarrow Bool$. We note that while this somewhat counterintuitive behavior is annoying in the case of subtyping, it is essential to the use of inheritance, as we shall see in a later section.

How, then, can we show that one object type is a subtype of another? The correct solution to this problem is reflected in the recursive nature of *MyType*. Suppose $\tau(MyType)$ and $\tau'(MyType)$ are record types, where we have indicated with our notation that each of τ and τ' involve the type variable *MyType*. Let $\sigma = ObjectType(MyType)\tau(MyType)$ and $\sigma' = ObjectType(MyType)\tau'(MyType)$. Then $\sigma \leq \sigma'$ if we can show $\tau(MyType) \leq \tau'(MyType')$ under the assumption that $MyType \leq MyType'$. This is similar to the way that one shows that two mutually recursive functions satisfy their specifications. That is, show the first is correct under the assumption that the second is correct and vice-versa. The solution presented here is based on the criteria presented in (Amadio and Cardelli, 1990) for determining when two recursively defined types are subtypes.

As indicated above, the place where our example with points breaks down is with the *eq* method. If $MyType \leq MyType'$ then $MyType \rightarrow Bool$ need not be a subtype of $MyType' \rightarrow Bool$. (In fact, the reverse is true: $MyType' \rightarrow Bool \leq MyType \rightarrow Bool$, since *MyType* occurs in a contravariant position in the type of *eq*.) As a result, if all we know is that $MyType \leq MyType'$, we cannot show that $\{x\colon Num; y\colon Num; color\colon ColorType; eq\colon MyType \rightarrow Bool\}$ is a subtype of the type $\{x\colon Num; y\colon Num; eq\colon MyType' \rightarrow Bool\}$. Thus *ColorPointType* cannot be shown to be a subtype of *PointType*. Note that if the *eq* method in each of *PointType* and *ColorPointType* were dropped or replaced by a method such as $mv\colon Num \times Num \rightarrow$

MyType, in which *MyType* occurs only in a covariant position, then it would follow that $ColorPointType \leq PointType$.

A formal statement of the subtyping rules, including that for object types, will be given in Section 4.2.1.

The difficulty in finding a correct subtyping rule for object types has resulted in the creation of statically-typed object-oriented languages which are not type safe. We discuss problems with type-checking rules in some of these languages in more detail after the next section.

4.1.2 Inheritance and subclasses

While the definition of subtype depends on the types or interfaces of objects (and hence the types of their associated methods), inheritance is based on the implementations of methods, as specified in the bodies of their generating classes. In particular, inheritance is used to construct a class by making modifications to a previously defined class. Subclasses of a class are constructed by adding new methods or modifying old methods while *inheriting* all other methods from the first class (superclass) which are not modified. (We remind the reader that we are ignoring instance variables for now.) In C++, subclasses are called derived types.

Classes can be modified in two ways. First, a class may be updated by modifying the value of a method. If c is a class with method m_i, we write

$$\mathit{update}\ c\ \mathit{by}\ (\mathit{self}\colon \mathit{MyType})\{m_i = M'_i\}$$

for the new class whose m_i component is replaced by M'_i, but all of whose other methods are as given in c. The type of M'_i must be a subtype of the type of the body of m_i in c.

For example, using *PointClass* defined in section 3.1, we can define

$$\mathit{DifferentPointClass} = \mathit{update}\ \mathit{PointClass}\ \mathit{by}\ (\mathit{self}\colon \mathit{MyType})\{x = 17\}$$

Objects generated from this new class will be of the same type as those generated from $PointClass$, but will have an x value of 17 rather than 47. We write $\mathit{update}\ c\ \mathit{by}\ (\mathit{self}\colon \mathit{MyType})\{m'_1 = M'_1, \ldots, m'_k = M'_k\}$ as an abbreviation for the obvious series of updates of single methods. Because of the advantages to allowing several methods to be updated at once (*e.g.*, in updating mutually-recursive methods), a more practical version of this language would allow the simultaneous updating of multiple methods. We have chosen to modify just one at a time here for notational convenience.

Second, a class may be extended by adding a new method. If c is a class which does not contain a method m' then we write

$$\mathit{extend}\ c\ \mathit{with}\ (\mathit{self}\colon \mathit{MyType})\{m' = M'\}$$

for the new class obtained from c which contains all of the methods of c plus a new method m' with body M'. Again, we can write

$$\mathit{extend}\ c\ \mathit{with}\ (\mathit{self}\colon \mathit{MyType})\{m'_1 = M'_1, \ldots, m'_n = M'_n\}$$

as an abbreviation for the obvious series of extensions formed by adding methods one at a time. As before the restriction on adding only one method at a time would be dropped in a more practical version of this language.

As an example of a class extension, we define

$$ColorPointClass = extend\ PointClass\ with\ (self\colon MyType)\{color = red\}.$$

where *PointClass* was defined in section 3.1.

We say that a class c' is an *immediate subclass* of a class c if c' is of the form *update* c *by* $(self\colon MyType)R$ or *extend* c *with* $(self\colon MyType)R$. A class c' is a *subclass* of c if $c' = c$, c' is an immediate subclass of c, or c' is a subclass of an immediate subclass of c. (In other words, the "subclass" relation is the reflexive and transitive closure of the "immediate subclass" relation.) If c' is a subclass of c then we say that c is a *superclass* of c' (similarly for the definition of *immediate superclass*).

In order to type check classes formed by updating or extending classes, it will be useful to define a relation between types which will be satisfied by the types of objects generated by classes formed by updating or extending a previously given class. If $ObjectType(MyType)\sigma$ and $ObjectType(MyType)\tau$ are both object types then we write

$$ObjectType(MyType)\sigma \leq_{meth} ObjectType(MyType)\tau \text{ iff } \sigma \leq \tau.$$

That is, $T_1 \leq_{meth} T_2$ iff T_1 can be obtained from T_2 by adding new methods or by replacing the types of methods in T_2 by subtypes. Thus the intuitively plausible subtyping rule which failed for object types has been resurrected here to generate the $\leq_{meth}$ relation between types. In particular, note that if c' is a subclass of c then their corresponding object types are in the $\leq_{meth}$ relation.

We are now ready to reexamine our counter-example to subtyping in the last section. Recall that in that section we defined:

$$PointType = ObjectType(MyType)\{x\colon Num; y\colon Num; eq\colon MyType \rightarrow Bool\}.$$

and

$$\begin{aligned} ColorPointType = {} & ObjectType(MyType) \\ & \{x\colon Num; y\colon Num; color\colon ColorType; eq\colon MyType \rightarrow Bool\}. \end{aligned}$$

Notice that *ColorPointType* is the type of objects generated by the class *ColorPointClass* defined above. We saw in the last section that *ColorPointType* is not a subtype of *PointType*. Clearly, however, $ColorPointType \leq_{meth} PointType$.

This example should help clarify the difference between subtypes and inheritance. Subtyping has only to do with the interfaces of objects, and reflects the usability of objects of one type in a context which expects the other. Inheritance, on the other hand, is related to the modification of a class to either update old methods or add new ones. All methods from the superclass which are not mentioned in the update or extend term are *inherited* from the superclass, and may be used as though they were defined directly in the subclass.

Our *ColorPointClass* example shows that the creation of a subclass need not result in the corresponding object types being subtypes. The ordering $\leq_{meth}$ is related to subtyping, but is only concerned with the subtyping relation on the types of methods in object types. The example above results in object types σ and τ such that $\sigma \leq_{meth} \tau$ but such that $\sigma \leq \tau$ *fails.*

4.1.3 Difficulties in type checking object-oriented languages

As motivation for the discussion of the type-checking rules and semantics for the language with subtypes, subclasses, and inheritance which will be given beginning in Section 4.2.1, we next discuss problems which arise in attempting to design type-safe statically-typed object-oriented programming languages. We begin with a discussion of problems with some existing statically-typed object-oriented programming languages.

The advantages of statically-typed languages are well-known. They include earlier detection of errors and the provision of better information to allow compiler optimizations. However, early object-oriented languages either had gaping type holes (Simula-67), or provided only very weak dynamic typing (Smalltalk). Some languages, such as C++ (Stroustrop, 1986), make no pretense to being type safe. The designers of the language Beta originally seemed to be interested in designing a type-safe language, but have recently argued that they don't care if the language is completely type safe (Madsen *et al.*, 1990). In fact, we argue below that fundamental decisions (especially the decision to identify subclasses and subtypes) made by some of the language designers would make it extremely difficult to create a language as expressive as desired, and still have a safe type system.

To make this discussion more concrete, we discuss three of the more successful attempts to define statically-typed object-oriented programming languages, Eiffel (Meyer, 1988; Meyer, 1992), Sather (Omohundro, 1991), which is a variant of Eiffel, and Trellis/Owl (Schaffert *et al.*, 1986). We note that each of these languages, as well as those languages mentioned earlier, identify classes with types. As a result, in the discussion below of Eiffel, we make the same identification.

Eiffel originally had rather serious holes in the typing system, but these are repaired, though in a way that we consider relatively unsatisfactory, in Eiffel 3.0. The most important problem is that Eiffel uses the "covariant" rule for parameters in redefining methods. That is, if c is a class with method m, one is allowed to redefine m in a subclass, c', of c in such a way that the classes of the parameters of m in c' are subclasses of the classes of the original parameters of m in c. Unfortunately, many examples exist which show this rule is not type-safe. For example, we can rewrite $PointClass$ and $ColorPointClass$ so that the parameter of the eq method is explicitly $PointType$ and $ColorPointType$, respectively, in the two classes. (Of course since Eiffel identifies classes and types, the types would really be declared as $PointClass$ and $ColorPointClass$.) Our example showing the failure of subtyping in subclasses breaks the type system of Eiffel 2.0. In fact the "contravariant" rule for subtyping functions as given in section 4.1.1 is the type-safe one for parameters. That is, the types of the parameters of m in c' should be supertypes (or superclasses in Eiffel) of the types (classes) of the original parameters in c.

To compensate for this deficiency (and others) in the type system, in version 3.0 of Eiffel an extra level of type checking is performed at link time. This system-wide type checking essentially performs a data flow analysis of the program in order to ensure that only type-safe message sends are performed (see (Meyer, 1992) for more details). We consider this solution unsatisfactory since the addition of a new class to a system may result in link-time type errors which would not be predictable from looking at the interface of the previous classes or the code of the new class. In particular this system-wide type checking will not be feasible if library vendors only distribute the interfaces of supplied classes along with the object code. The source code or equivalent high-level information will be necessary in order to run this type check.

In spite of these difficulties, Eiffel retains the "covariant" rule for parameters because of its usefulness. As we have seen in our example of $PointClass$ and $ColorPointClass$ in the previous section, it is very convenient to extend a class with an *eq* method by adding new attributes. Suppose the definition of method *eq* in class c determines if another object of class c is equal to the current object. The type of *eq* is clearly $c \rightarrow Bool$. If a subclass c' of c is defined, it is natural to want to redefine *eq* so that it now has type $c' \rightarrow Bool$. The covariant typing rule for parameters would allow this redefinition while the contravariant rule would disallow it. We accomodate this example in our language by giving the parameter type *MyType*. Our *MyType* corresponds to the use of "like Current" in Eiffel. This example helps explain why we felt it was extremely important to provide for the type of *self* in TOOPL.

The language Sather is derived from Eiffel. One of the most important differences from Eiffel is that it adopts the contravariant rule for parameters, fixing this type insecurity. Trellis/Owl was designed independently from Eiffel, but also essentially adopts the contravariant rule. (Technically, in Trellis/Owl one may not define a subclass unless it is also a subtype.) As a result one could not define c' as described above by inheritance in either Sather or Trellis/Owl. (Of course one could still define a class with the same behavior as c' by just copying all necessary methods from c in the definition of c'.) Thus, in practical terms, the adoption of the contravariant parameter typing rule in these languages implies a loss of expressibility relative to Eiffel.

We have extricated ourselves from this type-safety versus expressiveness tradeoff in our language design by adopting the separation of classes and types suggested in (Cook *et al.*, 1990). This allows us to write down subclasses like $ColorPointClass$ of $PointClass$ without breaking the type system. The trade-off in this case is that the corresponding object types are not subtypes. Notice, however, that we have not chosen to adopt the covariant subtyping rule for function types. We are able to write down subclasses like $ColorPointClass$ because our definition of $\leq_{meth}$ allows us to check for type-safety without considering whether the resulting object type (which results from a fixed point) is itself a subtype.

Moreover we gain extra expressibility by our use of *MyType* in specifying the types of methods. In particular, by the definition of $\leq_{meth}$, the fact that the meaning of *MyType* changes in defining subclasses may be disregarded when inheriting or modifying methods. This flexible use of *MyType* results in a much more expressive language, allowing us to define many more subclasses than would otherwise be possible.

Unfortunately, there are more things that we need to consider in order to ensure that we have a type-safe language. Just as we needed to be careful in defining the subtype relation on objects, we must be careful in type checking methods of a class, since these methods must remain type-correct when updating or extending the class in order to create other classes. That is, if a class, c, is type correct, and we define c' by updating or extending c, we want the inherited methods to remain type correct.

There are two alternatives if we want a type-safe language. We can either type check all inherited methods again when we inherit them, or we can attempt to type check the methods of a class in such a way that they will remain type correct under all possible legal updates or extensions. Since we have already committed ourselves to supporting a language in which the source code for library classes (and hence for their methods) may not be available, we design our language to support the second alternative. (There is a third alternative, which is adopted by most object-oriented languages, which is simply to ignore the issue and hope the user will not write type-unsafe programs.)

The key to designing type checking rules which guarantee type correctness in all possible subclasses is the choice as to which assumptions are made about the bound type variable, *MyType*, in type checking a term of the form $class(self\colon MyType)R$. Suppose we wish to show $class(self\colon MyType)R$ has type $ObjectType(MyType)\tau$. Before we introduced subtyping and inheritance, we assumed that $MyType = ObjectType(MyType)\tau$, for τ the type of R, in type-checking $class(self\colon MyType)R$ (see the type-checking rules in Section 3.2). However, in the presence of inheritance we may end up with methods that type check for the original class, but fail to type check when inherited in subclasses. Recall that one of our goals is to type check methods only once, when they are originally defined, so that we do not have to type check them again in every class in which they are inherited.

The following example illustrates the problems that may occur. Let

$$XType = ObjectType(MyType)\{x\colon Num; eq\colon MyType \rightarrow Bool; test\colon Bool\}$$

and let xob be any fixed expression of type $XType$. Then define

$$\begin{aligned} XClass = class(self\colon MyType)\{\;& x = 0;\\ & eq = fun(p\colon MyType)\ ((self \Leftarrow x) = (p \Leftarrow x));\\ & test = \ldots(self \Leftarrow eq)(xob)\ldots\} \end{aligned}$$

where we indicate only that the body of $test$ includes the subexpression $(self \Leftarrow eq)(xob)$. If we assume that $MyType = XType$ then, because $self$ will have type $XType$ (and presuming the body of $test$ type checks to have type $Bool$), $XClass$ will type check with type

$$ClassType(MyType)\{x\colon Num; eq\colon MyType \rightarrow Bool; test\colon Bool\}.$$

Therefore as expected, objects generated from it would have type $XType$, and, in fact, we encounter no problems with the use of objects generated from $XClass$.

However, suppose we modify $XClass$ to create:

$$TempColoredXClass = extend\ XClass\ with(self{:}\, MyType)\{color = red\},$$

and then update the eq method in

$$\begin{aligned}ColoredXClass = update\ PartColoredXClass\ by(self{:}\, MyType)&\\ \{eq = fun(p{:}\, MyType)\ ((self \Leftarrow x) = (p \Leftarrow x))&\\ \&\ ((self \Leftarrow color) = (p \Leftarrow color))\}&,\end{aligned}$$

which generates objects of type

$$\begin{aligned}ColorXType = ObjectType(MyType)\{x{:}\, Num; eq{:}\, MyType \rightarrow Bool;&\\ test{:}\, Bool; color{:}\, ColorType\}&.\end{aligned}$$

Then the $test$ method which is inherited from $XClass$ is no longer type-correct in $ColoredXClass$ (actually, it was already type-incorrect in $TempColoredXClass$). The problem lies with the subterm, "$(self \Leftarrow eq)(xob)$." This was correctly typed in the original definition, but in $ColorXClass$, $self$ has type $MyType$, which will now correspond to $ColorXType$, and thus the type of $self \Leftarrow eq$ is $ColorXType \rightarrow Bool$ rather than $XType \rightarrow Bool$. Since xob has type $XType$ and not type $ColorXType$, $(self \Leftarrow eq)(xob)$ is not well-typed. Moreover, if we try to evaluate the term, we see that it will crash when it attempts to examine the $color$ field of xob.

One solution to this problem is to explicitly check that all inherited methods are still correctly typed each time a class is modified. This, however, would require that all classes defined by inheritance have access to the actual code for all methods of the classes from which it inherits. (Note that one must not only repeat the type checking for all methods defined in the immediate superclass, but all the classes it inherits from, etc.) Aside from being time-consuming, this violates our normal expectations of the kind of information necessary in order to use external (and perhaps separately compiled) segments of code in a program.

We avoid these problems by adopting a more conservative type-checking regimen. Again, the key is our assumption about the relation of $MyType$ to the type of objects generated by the class when type checking methods of that class. We have seen that in type checking a term of the form $class(self{:}\, MyType)R$, the assumption that $MyType = ObjectType(MyType)\tau$, where τ is the type of R, leads to possibly incorrect typing when methods are inherited. On the other extreme, if we make no assumptions at all about $MyType$ then we will be unable to type check any term of the form $self \Leftarrow m$, since we will know nothing about $MyType$, the type of $self$.

The position taken here is between these extremes. We assume just enough information about $MyType$ to ensure that methods remain well-typed in all possible updates and extensions of $MyType$. If $ObjectType(MyType)\tau'$ is the type of objects generated from a subclass, c', of a class, c, and $ObjectType(MyType)\tau$ is the type of objects generated from c, then $ObjectType(MyType)\tau' \leq_{meth} ObjectType(MyType)\tau$. As a result, we type check the methods of the class c under the assumption that $MyType \leq_{meth} ObjectType(MyType)\tau$. Under these conditions, the method $test$ of the $XClass$ example fails to type check, since the parameter of $self \Leftarrow eq$ should be of type *MyType*, not *XType*. The reader should note the importance of having a bound variable like *MyType* in order to carry out this type checking.

This conservative assumption on the relationship of *MyType* to the class being defined may seem to be rather severe, but it appears necessary if we wish to guarantee that inherited methods remain well-typed.

The rule adopted here is a generalization of Mitchell's assumption in (Mitchell, 1990a) that methods should type-check assuming only that *MyType* is some *extension* of the intended object type. Since his paper did not deal with subtypes, it was not possible to replace the type of a method by a subtype there.

On the other hand, notice that, unlike C++ and Object Pascal, we do allow the types of methods to be changed in subclasses, as long as the new type is a subtype of the original type of the method. It is easy to create examples in which an inherited method, m, breaks when the type of another method, m', called in the body of m, has its type changed in an undisciplined way. For example, if m' used to return a number, but is redefined to return a string, other methods of the superclass which depended on m' to return a number will now break when inherited in the subclass. If the new type of m' is a subtype of the original type, however, then it can be used in any context which expected a value of the original type. Hence inherited methods will continue to type check properly.

4.1.4 Referring to superclass methods

In the definition of a subclass in object-oriented programming, it is quite common to refer to methods of the immediate superclass. For instance, if method m is being redefined in a subclass, it may be desirable to first perform some preliminary actions, next execute the method body as defined in the superclass, and then perform some final actions. This sort of computation is typically supported by making available the methods of the immediate superclass via a keyword *super*.

We will provide that access in our language via a bound variable (similar in usage to *self*), which will usually be designated as *super*. It is added to the heading of subclass definitions and denotes a record containing all of the method definitions of the immediate superclass. Thus if c has type $ClassType(MyType)\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}$ and c' is obtained by updating method m_i, for some $1 \leq i \leq n$, then we write:

$$c' = update\ c\ by\ (self\colon MyType; super)\{m_i = M_i'\}$$

where M_i' may contain occurrences of both *self* and *super*, and where *super* has type $\{m_1\colon \tau_1; \ldots, m_n\colon \tau_n\}$. Note that because *super* has a record type rather than an object type, its methods are invoked by $super.m$ rather than $super \Leftarrow m$. The variable *super* can also be used in subclasses formed by extension, though it may not be as useful.

A good example of the use of *super* is in defining a subclass of *ColorPointClass*. Recall that *ColorPointClass* was defined by simply adding a new *color* field. However, it also makes sense to change the definition of the *eq* method so that it also ensures that the color fields are equal. Therefore, we define,

$$\begin{array}{l} \mathit{BetterColorPointClass} = \mathit{update}\ \ \mathit{ColorPointClass}\ \ \mathit{by}\ (\mathit{self}\colon \mathit{MyType}; \mathit{super}) \\ \qquad\qquad \{\mathit{eq} = \mathit{fun}(p\colon \mathit{MyType})\ \ \mathit{super.eq}(p)\ \ \&\ \ (p \Leftarrow \mathit{color}) = (\mathit{self} \Leftarrow \mathit{color})\} \end{array}$$

Thus we see that we can combine the effect of the old method with additional actions in order to define the new operation.

4.2 Syntax and type-checking rules for SOOPL

Now that we have completed our informal discussion of subtyping, inheritance, and type checking, we are ready to present the formal syntax and type-checking rules for SOOPL. We begin with a formal definition of our two orderings on object types.

4.2.1 Rules for $\leq$ and $\leq_{meth}$ for SOOPL

In this section we provide formal axioms and rules for the orderings $\leq$ and $\leq_{meth}$ on type expressions. The syntax for type expressions for SOOPL is exactly the same as for ROOPL.

As explained in Sections 4.1.1 and 4.1.2, $\leq$ represents the subtype relation between types, while $\leq_{meth}$ is a pointwise ordering relating types of objects in which the class of the first object could have been generated by inheritance from the class of the second.

Definition 4.1
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 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 sequence, C, of simple type constraints, then we say t is *declared* in C. A *type constraint system* is defined as follows:

1. The empty sequence, ϵ, is a type constraint system.
2. If C is a type constraint system and $t \leq \tau$ is a simple type constraint such that t does not appear in C or τ, then $C;\ t \leq \tau$ is a type constraint system.
3. If C is a type constraint system, τ is of the form $\mathit{ObjectType}(\mathit{MyType})\sigma$, 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 type constraint system.

For example, if t is a type variable, then

$$\begin{array}{l} C = \epsilon;\ t \leq\ \{x\colon Num; add\colon Num \rightarrow Num\}; \\ \qquad\qquad\qquad \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\{mv\colon \mathit{MyType} \rightarrow t; z\colon Num\} \end{array}$$

is a type constraint system. (We will usually delete the leading ϵ.)

Notice that in the above example, the *MyType* which occurs to the left of the last $\leq$ is different from the two occurrences to the right of the $\leq$. They differ because the second occurrence of *MyType* represents a new bound variable which includes the last *MyType* within its scope. We will take advantage of these distinctions to simplify some of the statements of type-checking rules later in this section.

Refl($\leq$) $$C \vdash \tau \leq \tau$$

Var($\leq$) $$C;\; t \leq \tau; C' \vdash t \leq \tau$$

Trans($\leq$) $$\frac{C \vdash \gamma \leq \sigma,\;\; C \vdash \sigma \leq \tau}{C \vdash \gamma \leq \tau}$$

Func($\leq$) $$\frac{C \vdash \sigma' \leq \sigma,\;\; C \vdash \tau \leq \tau'}{C \vdash \sigma \rightarrow \tau \leq \sigma' \rightarrow \tau'}$$

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

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

In the *Obj($\leq$)* rule, neither s nor t may occur free in C, τ, or τ'.

Fig. 4. Subtype Axioms and Rules for SOOPL.

We define *subtyping* or *type constraint* derivations of the form $C \vdash \sigma \leq \tau$, for C a type constraint system, and σ, τ type expressions, via the axioms and rules given in Figure 4.

As stated earlier, in record, *ObjectType*, and *ClassType* types, the ordering of the component (or method) names, the m_i's, are not important. For instance, $\tau' \leq \tau$, for record types τ and τ', if all of the labels of τ' are included in those in those of τ, and the types of corresponding labels are in the subtype relation.

The subtyping rules presented here reflect our discussion in Section 4.1.1. In particular, subtyping for function spaces, *Func($\leq$)*, is covariant in the range and contravariant in the domain. Rule *Rec($\leq$)* states that subtypes of records can be formed either by adding extra fields to the record or replacing types of fields by subtypes. As discussed in Section 4.1.1, the obvious rule for subtyping objects fails, and we are forced to use the more complex rule *Obj($\leq$)*, though the more intuitive rule turns out to be valid for $\leq_{meth}$ below.

Only two axioms and one rule are applicable for the case of the ordering $\leq_{meth}$. They are given in Figure 5. The two axioms are rather trivial. The *Trans($\leq_{meth}$)* rule, which provides a weak form of transitivity, provides the connection between the subtyping and inheritance relations. The following rule, which can be derived from *Trans($\leq_{meth}$)* and *Refl($\leq_{meth}$)*, illustrates this connection.

$$\frac{C \vdash \tau \leq \tau'}{C \vdash \mathit{ObjectType}(\mathit{MyType})\tau \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau'}\;.$$

Essentially it says that if one (record) type is a subtype of another (where either or both may involve a type variable "*MyType*") then a class whose record of method

$Var(\leq_{meth})$ $\quad C;\ t \leq_{meth} \tau; C' \vdash t \leq_{meth} \tau$

$Refl(\leq_{meth})$ $\quad C \vdash ObjectType(MyType)\tau \leq_{meth} ObjectType(MyType)\tau$

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

In the *Trans*($\leq_{meth}$) rule, *MyType* may not occur free in C.

Fig. 5. Inheritance Relation Axioms and Rules for SOOPL.

types matches that of the subtype could have inherited from a class whose method types matches that of the supertype. Again, this is the naive rule that we rejected for proving objects are subtypes. Instead, this rule reflects the restrictions on constructing subclasses by extension or modification of methods of the superclass. An easy induction on the length of proofs shows that if $C \vdash \sigma \leq_{meth} \sigma'$ then σ' is of the form $ObjectType(MyType)\tau$ for some τ.

4.2.2 Type-checking rules for SOOPL

There are only two new pre-terms introduced when expanding from ROOPL to SOOPL. They are given by the following grammar:

$$\begin{aligned} M ::= \;& update\ c\ by\ (self\colon MyType; super)\{m_1 = M_1'\} \mid \\ & extend\ c\ with\ (self\colon MyType; super)\{m_{n+1} = M_{n+1}\}. \end{aligned}$$

As before, c and M_i represent pre-terms in the grammar.

Also as before, terms of the language are those which can be assigned types. The type assignment derivations in SOOPL depend on both a syntactic type assignment, E, and a type constraint system, C. We only include the type-checking rules for object-oriented terms here. The other rules and axioms differ only in the inclusion of C to the left of the $\vdash$. The type assignment axioms and rules are presented in Figure 6.

As indicated earlier, the variables *self*, *MyType*, and *super* are considered bound in the *class* and *update* expressions. Terms which agree up to bound variables are considered to be identical. The terms representing subclass formation are simplified to allow only the addition or replacement of one method at a time. In fact, it is straightforward to write more general rules allowing the change of any number of methods at a time (and in any order). We have not done that here in order to keep the rules as simple as possible.

Aside from the rule, *Object*, the type-checking rules for object-oriented terms are now somewhat more complicated than they were for ROOPL.

As indicated in Section 4.1.3, in order to ensure that methods continue to type check in subclasses, we type check methods in rule *Class* only with the assumption that $MyType \leq_{meth} ObjectType(MyType)\tau$.

The primary reason that the rule for message sending, *Message*, is more complex

$$\textit{Class}\quad \frac{C;\ \textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\tau, E \cup \{\textit{self}: \textit{MyType}\} \vdash R: \tau}{C, E \vdash \textit{class}(\textit{self}: \textit{MyType})R: \textit{ClassType}(\textit{MyType})\tau}$$

$$\textit{Object}\quad \frac{C, E \vdash c: \textit{ClassType}(\textit{MyType})\tau}{C, E \vdash \textit{new}\ c: \textit{ObjectType}(\textit{MyType})\tau}$$

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

$$\textit{Update}\quad \frac{\begin{array}{c} C, E \vdash c: \textit{ClassType}(\textit{MyType})\{m_1: \tau_1; \ldots; m_n: \tau_n\}, \qquad C \vdash \tau_1' \leq \tau_1, \\ C;\ \textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\{m_1: \tau_1'; m_2: \tau_2; \ldots; m_n: \tau_n\}, \\ E \cup \{\textit{self}: \textit{MyType}, \textit{super}: \{m_1: \tau_1; \ldots; m_n: \tau_n\}\} \vdash M_1': \tau_1' \end{array}}{\begin{array}{l} C, E \vdash \textit{update}\ c\ \textit{by}\ (\textit{self}: \textit{MyType}; \textit{super})\{m_1 = M_1'\}: \\ \qquad\qquad \textit{ClassType}(\textit{MyType})\{m_1: \tau_1'; m_2: \tau_2; \ldots; m_n: \tau_n\} \end{array}}$$

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

$$\textit{Subsump}\quad \frac{C \vdash \tau \leq \sigma, \quad C, E \vdash M: \tau}{C, E \vdash M: \sigma}$$

Fig. 6. Type Assignment Axioms and Rules for SOOPL.

than before is to handle the important special case of sending a message to *self* in the body of a method. If the method appears in the body of a class c with type $\textit{ObjectType}(\textit{MyType})\{m_1: \tau_1; \ldots; m_n: \tau_n\}$, then, as can be seen by examining the rule, *Class*, the only thing that we may assume about the type of *self*, *MyType*, is that $\textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\{m_1: \tau_1; \ldots; m_n: \tau_n\}$. Because we do not know the exact object type *MyType* represents (since the method may be inherited in any subclass), our typing rule must rely only on knowledge of an upper bound in the $\leq_{meth}$ ordering. It is easy to see using rules *Trans(*$\leq_{meth}$*)* and *Rec(*$\leq$*)* that it is sufficient to use as upper bound an object type with only one method.

While the rules for subclasses look complex, they are rather easily justified. We discuss the rule *Extend* before *Update* since it is slightly less complicated. To type check an extension of a class, c, with type $\textit{ClassType}(\textit{MyType})\{m_1: \tau_1; \ldots; m_n: \tau_n\}$, it is sufficient to type check the body of the new method under the assumption that *MyType* will be $\leq_{meth}$ the expected type of the extended class. Because *super* may occur in the type of the body of the new method, one may also assume that *super* has the type of the record of methods of the superclass, c. It is not necessary to type check the inherited methods since they were originally checked under assumptions which are weaker than are used here.

The rule *Update* is only slightly more complex. Since we are replacing the body of a method in the update, we need, for reasons discussed in Section 4.1.3, to have the type of the new method be a subtype of the type of the method body being replaced. Aside from that it is very similar to *Extend.*

Finally the subsumption rule is a formal statement of the intuition behind our definition of subtype. That is, if $\tau \leq \sigma$ and M has type τ, then it can be used as if it had type σ.

4.3 Formal semantics for SOOPL

In this section we modify the semantics given for ROOPL in order to handle the added complexity of subtyping and inheritance. In order to give a more compact presentation of the semantics we do not repeat the semantic definitions for those constructs which remain essentially unchanged from ROOPL. A complete semantics for the language which includes instance variables can be found in Section 5.3. Here we provide a formal denotational semantics as well as an intuitive explanation of the meanings of the terms in the presence of subtyping and inheritance.

4.3.1 Semantics of types

The semantics of type expressions remain the same as given for ROOPL, with the exception of class types. Before providing the revised definition of class types, we take a brief detour in order to gain an understanding of the semantics of $\leq$ and $\leq_{meth}$.

As in section 3.3, the semantics of SOOPL is given with respect to a model, $\mathcal{A}$, of the F-bounded second-order polymorphic lambda calculus with recursive types and elements. While we did not bother to remark upon this in section 3.3, this model provides an interpretation, $\leq_{\mathcal{A}}$, for $\leq$ which satisfies all of the axioms and rules for subtypes given in the previous section.

Moreover, in each such model there exists a well-behaved coercion function, **convert**, such that, if $\xi \leq_{\mathcal{A}} \xi'$ are interpretations of types, then $\mathbf{convert}[\xi'][\xi]: \xi \rightarrow \xi'$. This coercion function was described informally in section 4.1.1, and may be understood as a sort of homomorphism that preserves behavior. A set of axioms and rules governing the behavior of **convert** can be found in (Bruce and Longo, 1990).

The subsumption rule in the previous section stated that if $C, E \vdash M{:}\,\tau$ and $C \vdash \tau \leq \sigma$, then $C, E \vdash M{:}\,\sigma$. It will be possible to show that under these circumstances, for all consistent environments, ρ,

$$[\![C, E \vdash M{:}\,\sigma]\!]\rho = \mathbf{convert}[[\![\sigma]\!]\rho][[\![\tau]\!]\rho]([\![C, E \vdash M{:}\,\tau]\!]\rho).$$

The meaning of the $\leq_{meth}$ relation on types requires more care. Recall from the end of section 4.2.1 that if $C \vdash \sigma \leq_{meth} \sigma'$, then σ' must be of the form $\mathit{ObjectType}(\mathit{MyType})\tau$. From the definitions given below, it will follow that $\sigma \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau$ is true in $\mathcal{A}$ if and only if $[\![\sigma]\!]\rho \leq_{\mathcal{A}} [\![\tau]\!]\rho[[\![\sigma]\!]\rho/\mathit{MyType}]$.

The intuition behind this is as follows. If $C \vdash \mathit{ObjectType}(\mathit{MyType})\tau' \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau$, then the corresponding records of methods are subtypes. That is, $C \vdash \tau' \leq \tau$. It follows that if $\xi' = [\![\mathit{ObjectType}(\mathit{MyType})\tau']\!]\rho$, then $\xi' =$

$[\![\tau']\!]\rho[\xi'/MyType] \leq_{\mathcal{A}} [\![\tau]\!]\rho[\xi'/MyType]$. Hence $\xi' \leq_{\mathcal{A}} [\![\tau]\!]\rho[\xi'/MyType]$ as suggested above.

We begin with the formal definition of when a type environment is consistent with a type constraint system. Notice that the clause for $\leq_{meth}$ corresponds to the intuition given above.

Definition 4.2
The following inductive definition determines when a type environment ρ is *consistent* with a type constraint system C:

1. If C is empty then ρ is consistent with C.
2. Suppose $C;\ t \leq \sigma$ is a type constraint system. If ρ is consistent with C and $\rho(t) \leq_{\mathcal{A}} [\![\sigma]\!]\rho$ then ρ is *consistent* with $C;\ t \leq \sigma$.
3. Suppose $C;\ t \leq_{meth} ObjectType(MyType)\tau$ is a type constraint system. If ρ is consistent with C and $\rho(t) \leq_{\mathcal{A}} [\![\tau]\!]\rho[\rho(t)/MyType])$ then ρ is *consistent* with $C;\ t \leq_{meth} ObjectType(MyType)\tau$.

The lemma below states that the intuitive understanding of $\leq$ and $\leq_{meth}$ is preserved by the axioms and rules for $\leq$ and $\leq_{meth}$.

Lemma 4.3
Let the type environment ρ be consistent with C.

1. If $C \vdash \sigma \leq \tau$ then $[\![\sigma]\!]\rho \leq_{\mathcal{A}} [\![\tau]\!]\rho$.
2. If $C \vdash \gamma \leq_{meth} ObjectType(MyType)\tau$ then $[\![\gamma]\!]\rho \leq_{\mathcal{A}} [\![\tau]\!]\rho[[\![\gamma]\!]\rho/MyType]$.

Proof
The proof proceeds by an induction on the length of derivations. For part 1, the base clauses of the induction follow from the reflexivity of $\leq_{\mathcal{A}}$ and the definition of consistency of an environment given above. The only difficult part of the induction is that corresponding to the subtyping rule for object types. Since object types denote fixed points, the desired result follows from a theorem of (Amadio and Cardelli, 1990), which states that our subtyping rule for recursive types is sound.

For part 2, the soundness of the (*Var(*$\leq$*)*) axiom follows from the definition of consistency of an environment given above. The (*Refl(*$\leq$*)*) axiom follows from the fact that if $\xi = [\![ObjectType(MyType)\tau]\!]\rho$ then $\xi = [\![\tau]\!]\rho[\xi/MyType]$. We provide details only for the (*Trans(*$\leq$*)*) rule here.

Suppose $C \vdash \gamma \leq_{meth} ObjectType(MyType)\tau$ and $C \vdash \tau \leq \tau'$. It follows that $C \vdash \gamma \leq_{meth} ObjectType(MyType)\tau'$ by the (*Trans(*$\leq$*)*) rule. We must show that $[\![\gamma]\!]\rho \leq_{\mathcal{A}} [\![\tau']\!]\rho[[\![\gamma]\!]\rho/MyType]$. But

$$[\![\gamma]\!]\rho \leq_{\mathcal{A}} [\![\tau]\!]\rho[[\![\gamma]\!]\rho/MyType]$$

by induction, and, by part 1,

$$[\![\tau]\!]\rho[[\![\gamma]\!]\rho/MyType] \leq_{\mathcal{A}} [\![\tau']\!]\rho[[\![\gamma]\!]\rho/MyType].$$

Therefore,

$$[\![\gamma]\!]\rho \leq_{\mathcal{A}} [\![\tau]\!]\rho[[\![\gamma]\!]\rho/MyType] \leq_{\mathcal{A}} [\![\tau']\!]\rho[[\![\gamma]\!]\rho/MyType],$$

and we are done. $\square$

We are now finally ready to return to the definition of class types. The types of classes are more complex than those of objects, because a class must encode sufficient information to provide the meaning of methods in objects generated from any subclass as well as the original class itself.

The meaning of a class term must include interpretations for both *self* and *MyType*. In particular the type *MyType* may appear in the type of a method of a class. When a method is inherited in a subclass, the meaning of *MyType* will be the type of objects generated by that subclass. Because the meaning of *MyType* may change, the meaning of a class will take as a parameter an argument which indicates the meaning of *MyType*. The only object types which will be allowed for the actual parameter are those that might be generated from subclasses of the original class. If the type of the original class is $\mathit{ClassType}(\mathit{MyType})\tau$, then the possible types for objects generated by the class or any of its subclasses are those γ such that $\gamma \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau$. By the Lemma above, this is the same as the set of those γ such that $[\![\gamma]\!]\rho \leq_{\mathcal{A}} [\![\tau]\!]\rho[[\![\gamma]\!]\rho/\mathit{MyType}]$.

Now, in order to interpret the body of methods, we will also need to provide the meaning of the other bound variable in class definitions, *self*. The meaning of *self* must be an element of the meaning of *MyType*. Thus to interpret the method bodies of classes, we must provide an interpretation for *MyType*, and an interpretation for *self* which is an element of the interpretation of *MyType*. Thus the meaning of class types is as follows:

$$[\![\mathit{ClassType}(\mathit{MyType})\tau]\!]\rho = \prod_{\xi \leq_{\mathcal{A}} [\![\tau]\!]\rho[\xi/\mathit{MyType}]} (\xi \rightarrow [\![\tau]\!]\rho[\xi/\mathit{MyType}])$$

That is, the meaning of a class is a function which takes a ξ such that $\xi \leq_{\mathcal{A}} [\![\tau]\!]\rho[\xi/\mathit{MyType}]$ (*i.e.*, a type which might be generated by a subclass of a class of type $\mathit{ClassType}(\mathit{MyType})\tau$) as the interpretation of *MyType*, and an element of ξ as the interpretation of *self*, and returns a value of type τ (in which *MyType* is interpreted as ξ). The value of type τ returned is the meaning of the record of methods of the class.

The product over ξ such that $\xi \leq_{\mathcal{A}} [\![\tau]\!]\rho[\xi/\mathit{MyType}]$ is an example of a type defined by "F-bounded quantification", since the upper bound of ξ is a term involving ξ. This technique is used in (Cook *et al.*, 1990) and (Bruce, 1992), and was originally proposed in (Canning *et al.*, 1989).

An alternative, due to Cardelli and Mitchell, and used for example in (Mitchell, 1990a) and (Pierce and Turner, 1993), is to replace the parameter, ξ, ranging over types, by a parameter whose possible values are functions from types to types, and whose fixed points represent the types of objects generated from subclasses. (That is, these fixed points replace the ξ which appear in our definition.) An earlier version of this paper used that approach, but it has been our experience that this approach is harder to understand than F-bounded quantification. Interestingly, Abadi (Abadi, 1992) has presented a semantic argument which shows that anything which can be expressed with one formalism can also be expressed (with the help of set-theoretic operations) in the other.

$$\begin{aligned}
&\llbracket C, E \vdash \mathit{class}(\mathit{self}\colon \mathit{MyType}) R\colon \mathit{ClassType}(\mathit{MyType})\tau \rrbracket\rho = \\
&\quad \lambda\xi \leq_{\mathcal{A}} \llbracket\tau\rrbracket\rho[\xi/\mathit{MyType}].\lambda o \in \mathcal{A}^{\xi}.\llbracket C;\ \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau, \\
&\qquad\qquad E \cup \{\mathit{self}\colon \mathit{MyType}\} \vdash R\colon \tau\rrbracket\rho[\xi/\mathit{MyType}, o/\mathit{self}].
\end{aligned}$$

$$\begin{aligned}
&\llbracket C, E \vdash \mathit{new}\ c\colon \mathit{ObjectType}(\mathit{MyType})\tau\rrbracket\rho = \\
&\qquad \mathit{Fix}((\llbracket C, E \vdash c\colon \mathit{ClassType}\ (\mathit{MyType})\tau\rrbracket\rho)(\llbracket \mathit{ObjectType}(\mathit{MyType})\tau\rrbracket\rho)).
\end{aligned}$$

$$\begin{aligned}
&\llbracket C, E \vdash o \Leftarrow m\colon \tau[\gamma/\mathit{MyType}]\rrbracket\rho = o'(m), \\
&\quad \text{where } o' = (\mathbf{convert}[\llbracket\{m\colon\tau\}\rrbracket\rho[\llbracket\gamma\rrbracket\rho/\mathit{MyType}]][\llbracket\gamma\rrbracket\rho])(\llbracket C, E \vdash o\colon \gamma\rrbracket\rho).
\end{aligned}$$

$$\begin{aligned}
&\llbracket C, E \vdash \mathit{update}\ c\ \mathit{by}\ (\mathit{self}\colon \mathit{MyType}; \mathit{super})\{m_1 = M_1'\}\colon \\
&\qquad\qquad \mathit{ClassType}(\mathit{MyType})\{m_1\colon \tau_1'; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\}\rrbracket\rho = \\
&\quad \lambda\xi \leq_{\mathcal{A}} \llbracket\{m_1\colon \tau_1'; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\}\rrbracket\rho[\xi/\mathit{MyType}].\lambda o \in \mathcal{A}^{\xi}.f, \\
&\quad \text{where } \mathit{sup} = (\llbracket C, E \vdash c\colon \mathit{ClassType}(\mathit{MyType})\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}\rrbracket\rho)(\xi)(o), \\
&\qquad \mathit{dom}(f) = \{m_1, \ldots, m_n\}, \\
&\qquad f(m_1) = \llbracket C;\ \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\{m_1\colon \tau_1'; \ldots; m_n\colon \tau_n\}, \\
&\qquad\qquad E \cup \{\mathit{self}\colon \mathit{MyType}, \mathit{super}\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}\} \\
&\qquad\qquad\qquad \vdash M_1'\colon \tau_1'\}\rrbracket\rho[\xi/\mathit{MyType}, o/\mathit{self}, \mathit{sup}/\mathit{super}], \\
&\quad \text{and for } 2 \leq j \leq n, \\
&\qquad f(m_j) = \mathit{sup}\ (m_j).
\end{aligned}$$

$$\begin{aligned}
&\llbracket C, E \vdash \mathit{extend}\ c\ \mathit{with}\ (\mathit{self}\colon \mathit{MyType}; \mathit{super})\{m_{n+1} = M_{n+1}\}\colon \\
&\qquad\qquad \mathit{ClassType}(\mathit{MyType})\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n; m_{n+1}\colon \tau_{n+1}\}\rrbracket\rho = \\
&\quad \lambda\xi \leq_{\mathcal{A}} \llbracket\{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\}\rrbracket\rho[\xi/\mathit{MyType}].\lambda o \in \mathcal{A}^{\xi}.f, \\
&\quad \text{where } \mathit{sup} = (\llbracket C, E \vdash c\colon \mathit{ClassType}(\mathit{MyType})\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}\rrbracket\rho)(\xi)(o), \\
&\qquad \mathit{dom}(f) = \{m_1, \ldots, m_{n+1}\}, \\
&\qquad f(m_{n+1}) = \llbracket C; \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\}, \\
&\qquad\qquad E \cup \{\mathit{self}\colon \mathit{MyType}, \mathit{super}\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}\} \\
&\qquad\qquad\qquad \vdash M_{n+1}\colon \tau_{n+1}\}\rrbracket\rho[\xi/\mathit{MyType}, o/\mathit{self}, \mathit{sup}/\mathit{super}], \\
&\quad \text{and for } 1 \leq j \leq n \\
&\qquad f(m_j) = \mathit{sup}\ (m_j).
\end{aligned}$$

$$\llbracket C, E \vdash M\colon \sigma\rrbracket\rho = \mathbf{convert}[\llbracket\sigma\rrbracket\rho][\llbracket\tau\rrbracket\rho](\llbracket C, E \vdash M\colon \tau\rrbracket\rho), \text{ if } C \vdash \tau \leq \sigma \text{ and } C, E \vdash M\colon \tau.$$

Fig. 7. Semantics of Terms in SOOPL.

4.3.2 Semantics of terms

As with ROOPL, ordinary terms of SOOPL will be interpreted in the context of an environment, ρ, which specifies the meaning of type and element variables. As in the previous section on type-checking SOOPL, we only include here the semantics for object-oriented terms. The semantics of the other terms remain the same as before. We also provide the intuition behind each of the new definitions.

In defining the meaning of terms we assume that the semantic environment is consistent with the sets of assumptions, C and E. (Recall the definition of an environment being consistent with E given in Definition 3.4.) The semantics of terms is given in Figure 7. Note that each clause of the semantic definition corresponds to a type assignment axiom or rule from section 4.2.2. The function **convert** which appears in the definitions is the one described in section 4.3.1. A discussion of why the semantics is well-defined can be found in Section 4.4.

We now briefly give an informal explanation of the formal semantics presented in Figure 7. In the following, we simplify our notation and write $[\![M]\!]\rho$ rather than the more complete and correct $[\![C, E \vdash M\colon \tau]\!]\rho$.

Class terms: Recall that

$$[\![\mathit{ClassType}(\mathit{MyType})\tau]\!]\rho = \prod_{\xi \leq_{\mathcal{A}} [\![\tau]\!]\rho[\xi/\mathit{MyType}]} (\xi \rightarrow [\![\tau]\!]\rho[\xi/\mathit{MyType}]).$$

Thus the meaning of $\mathit{class}(\mathit{self}\colon \mathit{MyType})R$ will be a function which takes two parameters. The first is a type ξ such that $\xi \leq_{\mathcal{A}} [\![\tau]\!]\rho[\xi/\mathit{MyType}]$, which will be the interpretation of *MyType*. (Recall that such a ξ is an object type which might be generated by a subclass of $\mathit{class}(\mathit{self}\colon \mathit{MyType})R$.) The second parameter is an element, o, from $\mathcal{A}^{\xi}$ which will be the interpretation of *self* in R. Given ξ and o from the appropriate domains,

$$([\![\mathit{class}(\mathit{self}\colon \mathit{MyType})R]\!]\rho)\ (\xi)(o) = [\![R]\!]\rho[\xi/\mathit{MyType}, o/\mathit{self}],$$

the meaning of the record of methods, R, with the appropriate interpretations for *MyType* and *self*.

Object constructors: In ROOPL, the meaning of *new* c was simply the fixed point of the meaning of c. However, in SOOPL, a class also takes a type parameter which corresponds to the meaning of *MyType*. If $C, E \vdash c\colon \mathit{ClassType}(\mathit{MyType})\tau$, then $C, E \vdash \mathit{new}\ c\colon \mathit{ObjectType}(\mathit{MyType})\tau$. We obtain the meaning of *new* c as follows. First compute $([\![c]\!]\rho)([\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho)$, which is a function from $\mathcal{A}^{[\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho}$ to $\mathcal{A}^{[\![\tau]\!]\rho[[\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho/\mathit{MyType}]}$. However, because

$$[\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho = [\![\tau]\!]\rho[[\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho/\mathit{MyType}],$$

this is just a function from $\mathcal{A}^{[\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho}$ to itself. Taking a fixed point of this function results in an element of $\mathcal{A}^{[\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho}$.

While the meaning of *new* c is an element of the right type, it may not be at all clear that the resulting value has the desired behavior. The following example illustrates that our definition assigns the correct meaning to objects. Let $c = \mathit{class}(\mathit{MyType})R\colon \mathit{ClassType}(\mathit{MyType})\tau$, where R is the record of methods of c. We show that $[\![\mathit{new}\ c]\!]\rho$ has the desired behavior. Recall that for $\xi \leq_{\mathcal{A}} [\![\tau]\!]\rho[\xi/\mathit{MyType}]$ and o an element of ξ,

$$([\![c]\!]\rho)\ (\xi)(o) = [\![R]\!]\rho[\xi/\mathit{MyType}, o/\mathit{self}].$$

Since $[\![\mathit{new}\ c]\!]\rho = \mathit{Fix}(([\![c]\!]\rho)\ ([\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho))$, it follows that for $o' =_{def} [\![\mathit{new}\ c]\!]\rho$,

$$\begin{aligned} o' &= ([\![c]\!]\rho)\ ([\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho)\ (o') \\ &= [\![R]\!]\rho[[\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho/\mathit{MyType}, o'/\mathit{self}]. \end{aligned}$$

Thus the meaning of *new* c is the meaning of the record of methods, R, in which *MyType* is interpreted as $[\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho$ and *self* is interpreted as the object as a whole. This is exactly the desired meaning of *new* c.

Message sending: The meaning of a term $o \Leftarrow m_i$ in SOOPL is more complex in SOOPL than in ROOPL, because the type-checking rule for methods, (*Message*), does not necessarily provide us with a type of the form $\mathit{ObjectType}(\mathit{MyType})\tau$ for the object receiving the message. If it does happen that the receiving object has a statically-determined type of that form, then the term has the same semantics as in ROOPL. However, as discussed in Section 4.2.2, we may only know that the receiving object's type is $\leq_{meth}$ to a given object type.

Recall that $\gamma \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\{m_i\colon \tau_i\}$ is true in the model if and only if

$$[\![\gamma]\!]\rho \leq_{\mathcal{A}} [\![\{m_i\colon \tau_i\}]\!]\rho[[\![\gamma]\!]\rho/\mathit{MyType}]$$

The type-checking rules state that $o \Leftarrow m_i$ can be assigned the type $\tau_i[\gamma/\mathit{MyType}]$ in this circumstance. In order to accomplish this semantically, we interpret $o \Leftarrow m_i$ by first coercing the meaning of o from an element of type $[\![\gamma]\!]\rho$ to an element of type $[\![\{m_i\colon \tau_i\}]\!]\rho[[\![\gamma]\!]\rho/\mathit{MyType}]$ (using **convert**), and then applying that element to m_i.

Subclass constructions: The only terms left to discuss are those which involve the construction of subclasses. Because the meanings of the *update* and *extend* constructs are similar, we discuss only the *update* construct in this section. The meaning of *extend* is similar, but slightly simpler.

Recall that the meaning of c with type $\mathit{ClassType}(\mathit{MyType})\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}$ is a function which takes two parameters, the first ranging over ξ such that $\xi \leq_{\mathcal{A}} [\![\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}]\!]\rho[\xi/\mathit{MyType}]$, and the second ranging over elements of ξ, returning a value from type $[\![\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}]\!]\rho[\xi/\mathit{MyType}]$. Therefore for ξ and o chosen properly, $([\![c]\!]\rho)(\xi)(o)$ is a function which takes an m_i in $\{m_1, \ldots, m_n\}$ and returns an element of $[\![\tau_i]\!]\rho[\xi/\mathit{MyType}]$.

The meaning of a term, c', of the form *update* c *by* $(\mathit{self}\colon \mathit{MyType}; \mathit{super})\{m_1 = M_1'\}$, with type $\mathit{ClassType}(\mathit{MyType})\{m_1\colon \tau_1'; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\}$, is given in terms of the meaning of c. Since $\tau_1' \leq \tau_1$, the type of m_1 in c, it follows that for all ξ,

$$[\![\{m_1\colon \tau_1'; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\}]\!]\rho[\xi/\mathit{MyType}] \leq_{\mathcal{A}} [\![\{m_1\colon \tau_1; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\}]\!]\rho[\xi/\mathit{MyType}].$$

As with our earlier definition of the meaning of classes, $[\![c']\!]\rho$ will be a function which takes two parameters. The first parameter ranges over types ξ such that $\xi \leq_{\mathcal{A}} [\![\{m_1\colon \tau_1'; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\}]\!]\rho[\xi/\mathit{MyType}]$, while the second ranges over elements, o, of ξ. Since $f = ([\![c']\!]\rho)(\xi)(o)$ is a record, it can be applied to a method name m_i, returning the value of that method.

If the method m_i is one of those unchanged by the *update* term, then it should return the same value as c did. That is, for $i \geq 2$, define

$$f(m_i) = ([\![c]\!]\rho)(\xi)(o)(m_i).$$

(This is well-defined since if $\xi \leq_{\mathcal{A}} [\![\{m_1\colon \tau_1'; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\}]\!]\rho[\xi/\mathit{MyType}]$, then, by transitivity, $\xi \leq_{\mathcal{A}} [\![\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}]\!]\rho[\xi/\mathit{MyType}]$.) On the other hand,

$$f(m_1) = [\![M_1']\!]\rho[\xi/\mathit{MyType}, o/\mathit{self}].$$

Thus we see how the meaning of a subclass depends on the meaning of the superclass. A similarly derived semantics can be given in the case of subclasses formed by adding new methods.

There is still one detail that we have not yet taken care of, and that is the use of the bound variable, *super*, which provides access to the methods of the immediate superclass. In the case of the subclasses defined above by updating methods, this would make it possible to refer to methods from the superclass in the new definition of a method. In the example where method m_1 was updated, it would allow the new body of m_1 to refer to the old value of m_1 (or any other method, for that matter).

Not unexpectedly, we proceed exactly as we did earlier in order to get access to the unmodified methods of the superclass. That is, we simply assign *super* the meaning $([\![c]\!]\rho)(\xi)(o)$. This results in a record containing all of the methods of c. Note that the meaning of *self* in the methods of *super* is o, the same as in the regular methods of the updated class.

Let Ξ' be the meaning of *update c by* $(self\colon MyType;\ super)\{m_1 = M_1'\}$. Then the final definition of $f = \Xi'(\xi)(o)$ satisfies:

$$f(m_i) = ([\![c]\!]\rho)\ (\xi)(o)(m_i),\ \text{for } i \geq 2, and$$

$$f(m_1) = [\![M_1']\!]\rho[\xi/MyType, o/self, ([\![c]\!]\rho)(\xi)(o)/super].$$

Since the ordering of methods does not matter, it is not necessary to provide separate semantics for updating other methods.

4.3.3 Warning on inheritance as textual substitution

This is a good place for us to comment on a problem with one of the intuitive ways of understanding inheritance. Inheritance is sometimes explained as being equivalent to textually substituting a method body from the superclass to the subclass. This intuition is helpful for many purposes, but breaks down decisively when the keyword *super* is used. A concrete example should make this clear.

In section 4.1.4, the class, *BetterColorPointClass*, was defined as a subclass of *ColorPointClass* while changing the meaning of the method, *eq*. The definition of *eq* given was:

$$eq = fun(p\colon MyType)\ super.eq(p)\ \&\ (p \Leftarrow color) = (self \Leftarrow color)$$

If a new subclass of *BetterColorPointClass* is formed by, say, adding a new method, then the definition of *eq* in *BetterColorPointClass* will be inherited by the subclass. Of course, this inherited method should do the same thing that it did in the superclass, that is, compare x, y, and *color* values. However, if we textually substitute the body of *eq* given in *BetterColorPointClass* into the subclass, then we see that it first does the *eq* operation found in the superclass, which is now *BetterColorPointClass*, making the comparison of x, y, and *color* fields. It then once more compares *color* fields. In this instance, checking the *color* field twice makes no difference, but one can easily imagine other situations, in which totals are being computed, for instance, in which this strategy would return a different value than what is expected. Thus, in the presence of *super*, one should not expect inheritance to work as simple textual substitution of method bodies into subclasses.

Of course, the so-called "call-by-textual-substitution" has a host of other problems in regular programming languages, because of problems with inserting text into scopes in which variables might be captured. This is a common problem with macro-expansion languages.

4.4 Soundness of typing

Our goal in this section is to show that our semantic definitions are consistent with our typing rules. The recursive definition of the meaning of terms given in Figure 7 only makes sense if we can be assured that the meaning of each term is an element of the meaning of its type. The following lemma shows that this is indeed true.

Theorem 4.4
(Soundness of semantics with respect to the typing rules) Let $\mathcal{A}$ be a model for the F-bounded second-order lambda calculus with fixed points at both the type and term levels. For all typing derivations, $C, E \vdash M{:}\,\sigma$, if ρ is consistent with C, E, then $[\![C, E \vdash M{:}\,\sigma]\!]\rho \in \mathcal{A}^{[\![\sigma]\!]\rho}$.

Proof
The proof is by induction on the length of derivations. The only interesting cases are those involving the object-oriented terms.

Suppose

$$C, E \vdash \mathit{class}(\mathit{self}{:}\, \mathit{MyType})R{:}\; \mathit{ClassType}(\mathit{MyType})\tau$$

follows from

$$C;\; \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau, E \cup \{\mathit{self}{:}\, \mathit{MyType}\} \vdash R{:}\, \tau.$$

Then

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

By induction, $\rho[\xi/\mathit{MyType}, o/\mathit{self}]$ is consistent with

$$C;\; \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau, E \cup \{\mathit{self}{:}\, \mathit{MyType}\}.$$

Then, again by induction,

$$\begin{aligned}[\![C;\; \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau, \\ E \cup \{\mathit{self}{:}\, \mathit{MyType}\} \vdash R{:}\, \tau]\!]\rho[\xi/\mathit{MyType}, o/\mathit{self}] \in \mathcal{A}^{[\![\tau]\!]\rho[\xi/\mathit{MyType}]}.\end{aligned}$$

It follows easily that

$$[\![C, E \vdash \mathit{class}(\mathit{self}{:}\, \mathit{MyType})R{:}\; \mathit{ClassType}(\mathit{MyType})\tau]\!]\rho \in \mathcal{A}^{[\![\mathit{ClassType}(\mathit{MyType})\tau]\!]\rho}.$$

The case for $M = \mathit{new}\ c$ follows from the fact that the meaning of an object type is a fixed point. Recall that

$$[\![C, E \vdash \textit{new } c\colon \textit{ObjectType}(\textit{MyType})\tau]\!]\rho =$$
$$\textit{Fix}(([\![C, E \vdash c\colon \textit{ClassType } (\textit{MyType})\tau]\!]\rho)([\![\textit{ObjectType}(\textit{MyType})\tau]\!]\rho)).$$

By induction, $([\![C, E \vdash c\colon \textit{ClassType } (\textit{MyType})\tau]\!]\rho)([\![\textit{ObjectType}(\textit{MyType})\tau]\!]\rho) \in \mathcal{A}^{\xi \rightarrow [\![\tau]\!]\rho[\xi/\textit{MyType}]}$ for $\xi = [\![\textit{ObjectType}(\textit{MyType})\tau]\!]\rho$. However, by the definition of $[\![\textit{ObjectType}(\textit{MyType})\tau]\!]\rho$ as a fixed point, $\xi = [\![\tau]\!]\rho[\xi/\textit{MyType}]$. Thus,

$$([\![C, E \vdash c\colon \textit{ClassType } (\textit{MyType})\tau]\!]\rho)([\![\textit{ObjectType}(\textit{MyType})\tau]\!]\rho) \in \mathcal{A}^{\xi \rightarrow \xi}.$$

As a result,

$$\textit{Fix}(([\![C, E \vdash c\colon \textit{ClassType } (\textit{MyType})\tau]\!]\rho)([\![\textit{ObjectType}(\textit{MyType})\tau]\!]\rho)) \in \mathcal{A}^{\xi},$$

and hence,

$$[\![C, E \vdash \textit{new } c\colon \textit{ObjectType}(\textit{MyType})\tau]\!]\rho \in \mathcal{A}^{[\![\textit{ObjectType}(\textit{MyType})\tau]\!]\rho}.$$

The cases for terms of the form *update c by* $(\textit{self}\colon \textit{MyType}; \textit{super})\{m_1 = M_1'\}$ and *extend c with* $(\textit{self}\colon \textit{MyType}; \textit{super})\{m_{n+1} = M_{n+1}\}$ depend on Lemma 4.3 (using the fact that if $\xi \leq_{\mathcal{A}} [\![\{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\}]\!]\rho[\xi/\textit{MyType}]$, then $\xi \leq_{\mathcal{A}} [\![\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}]\!]\rho[\xi/\textit{MyType}]$).

The only other case of interest is that for message-passing. Suppose $C, E \vdash o \Leftarrow m\colon \tau[\gamma/\textit{MyType}]$ follows from $C \vdash \gamma \leq_{meth} \textit{ObjectType}(\textit{MyType})\{m\colon \tau\}$ and $C, E \vdash o\colon \gamma$. Then

$$[\![C, E \vdash o \Leftarrow m\colon \tau[\gamma/\textit{MyType}]]\!]\rho = o'(m)$$

where

$$o' = (\mathbf{convert}[[\![\{m\colon \tau\}]\!]\rho[[\![\gamma]\!]\rho/\textit{MyType}][[\![\gamma]\!]\rho])\ ([\![C, E \vdash o\colon \gamma]\!]\rho).$$

By Lemma 4.3, $[\![\gamma]\!]\rho \leq_{\mathcal{A}} [\![\{m\colon \tau\}]\!]\rho[[\![\gamma]\!]\rho/\textit{MyType}]$, so the coercion is well-defined. By induction, $o' \in \mathcal{A}^{[\![\{m:\tau\}]\!]\rho[[\![\gamma]\!]\rho/\textit{MyType}]}$. As a result, $o'(m) \in \mathcal{A}^{[\![\tau]\!]\rho[[\![\gamma]\!]\rho/\textit{MyType}]} = \mathcal{A}^{[\![\tau[\gamma/\textit{MyType}]]\!]\rho}$. $\square$

The above theorem guarantees, for example, that if a term of the form $o \Leftarrow m$ type checks, and $[\![o]\!]\rho$ converges (i.e., $\neq \bot_{\mathcal{A}}$), then it will contain a field corresponding to m. Thus if o has a non-trivial value at run-time, it will be able to respond to the message m.

An alternate way of producing such a type-soundness theorem is to provide an operational semantics for the language, and show that types are preserved under computation. That is, if a term M "reduces to" (*i.e.*, computes) an irreducible value, v, then v can be assigned the same type as M. This is sometimes known as a subject-reduction theorem.

In (Bruce *et al.*, 1993) we provide an operational semantics for SOOPL, show this operational semantics is consistent with the denotational semantics given here, and prove such a subject-reduction theorem. This provides an alternative proof to Theorem 4.4 for the safety of the typing system.

We remind the reader that the definition of the semantics of a term is given by induction on the type derivation of a term. This is necessary, since by the rule (*Subsump*), a term can have many types. Nevertheless, this opens up the possibility that a given term can have two different derivations of the same type. How do we know then that the resulting meanings are the same?

One can deal with this problem either semantically or syntactically. In PER models of the bounded second-order lambda calculus (see (Bruce and Longo, 1990)), the meanings of terms may be obtained from the meanings of an underlying untyped model. In this way, one may show that the meanings of terms depend on only the "erasure" of the term (the term obtained by erasing all type information) and its type. The model given in (Bruce and Mitchell, 1992), which has been cited earlier as a model containing all necessary fixed points for our semantics, is a PER model. Thus, once a term and one of its types has been given, we can determine its meaning independently of the type derivation.

Alternatively, one may show the uniqueness of the meanings of types by showing that, given a fixed C, E, every typable term has a minimum type, τ_0, and that the meaning of a term at any type, $\tau \geq \tau_0$, may be obtained by coercing the meaning at τ_0 up to τ. Such a proof for the bounded second-order lambda calculus was given by (Curien and Ghelli, 1992) and is used in (Bruce and Longo, 1990). (An earlier, somewhat different syntactic proof appeared in (Breazu-Tannen *et al.*, 1991)). In (Bruce *et al.*, 1993), we show that in a slight variant of SOOPL, every typable term has a minimum type. This can then be used to show the meanings of types given above depend only on the term and its intended type, not on the particular derivation.

4.5 Why is SOOPL more complex?

The type-checking rules and semantics of our language have certainly greatly increased in complexity with the addition of subtypes and subclasses. How does this complexity arise?

The addition of inheritance forced us to both type check and specify the meaning of classes so that methods are well-typed and defined for all possible subclasses. This required us to type check methods in classes under weaker assumptions on the type of *self*. While the the type-checking rules and semantics for *update* and *extend* terms look complex, they follow rather naturally from those of the class they are modifying. It is important to note that in order to type check subclasses of a class, c, it is only necessary to know the type of c. It is not necessary to repeatedly type check inherited methods in subclasses.

The semantics of message passing is more complex than it was for ROOPL, since we need not know the type (at least as a concrete object type) of the object to which the message is being sent. Since this case occurs when messages are sent to *self* as part of the body of a method, a common occurrence in object-oriented programs, we clearly must be able to handle this situation. Thus, what might seem to be a rather useless generalization of message-passing – allowing a message to be sent to an object whose type is only known to be in the $\leq_{meth}$ relation to a known object type – is in fact critical to type checking methods which involve the use of *self*.

In the next section we generalize our language to include the use of updatable instance variables.

5 TOOPL: Adding instance variables

In this section we describe the full language, TOOPL, which includes instance variables and the operation *gets* which can be used to update instance variables. The primary difference between instance variables and methods is that methods can only be changed in classes, either by defining a new class or by updating a method in a subclass. Once an object is created, its methods are frozen. On the other hand, instance variables can be updated at any point during the lifetime of an object.

The language SOOPL that we have been discussing to this point has been quite restricted because of the absence of instance variables. For instance, we could create a point by applying *new* to a point class, but the only way we could move a point is by creating a brand new point by applying *new* to a class with different initial values for its instance variables.

In imperative object-oriented languages, sending a message to an object can result in updating the instance variables in place. Because our language is functional, we will update instance variables with a copy semantics. That is, we will create an object which looks the same as the original except that one of its instance variables now has a different value.

We illustrate the use of instance variables with an example. Let

$$\begin{aligned} \mathit{SlidingPtClass}(a,b) = {}& \mathit{class}(\mathit{self}\colon \mathit{MyType})(\{x = a,\ \ y = b\},\\ & \{\mathit{slidex} = \mathit{fun}(dx\colon \mathit{Num})\ \mathit{self}\ \mathit{gets}\ \{x = \mathit{self}.x + dx\}\}), \end{aligned}$$

This class has instance variables x and y which are initialized (on object creation) as a and b, respectively. The method *slidex* takes a parameter, dx, returning an object which is identical to the receiving object except that the value of the x instance variable is incremented by dx. If a and b are of type *Num*, *SlidingPtClass(a,b)* has type:

$$\mathit{ClassType}(\mathit{MyType})(\{x\colon \mathit{Num};\, y\colon \mathit{Num}\}, \{\mathit{slidex}\colon \mathit{Num} \rightarrow \mathit{MyType}\}).$$

Now suppose we define

$$\begin{aligned} \mathit{ColorSlidingPtClass}(a,b) = {}& \mathit{extend}\ \ \mathit{SlidingPtClass}(a,b)\\ & \mathit{with}\ (\mathit{self}\colon \mathit{MyType};\, \mathit{super})(\{\mathit{color} = \mathit{red}\}, \{\mathit{getcolor} = \mathit{self}.\mathit{color}\}). \end{aligned}$$

Thus *ColorSlidingPtClass(a,b)* also has an instance variable, *color*, and a method, *getcolor*, which returns the current value of *color*.

What happens when we send a *slidex* message to an object, *cp*, which is a color sliding point? Clearly the result should be another color sliding point.

If we tried to simulate this in SOOPL, however, we would not be able to achieve this functionality with inheritance. The only way of creating a new object with updated instance variables would be to apply *new* to *SlidingPtClass*(a', b') for some a', b', thus returning a sliding point. If we now create a subclass for color sliding point, though, the inherited *slidex* routine will still return a sliding point rather than a color sliding point. While it is possible to attempt to achieve the same effect with a *MyClass* construct similar to our *MyType* (see Section 7 for a discussion of this construct), providing updatable instance variables is a simpler solution to this problem.

5.1 Complications due to instance variables

The fact that instance variables are updatable (*i.e.*, are *acceptors* as well as *evaluators*, see (Reynolds, 1980)) will lead to restrictions on changing the types of instance variables in subclasses. Suppose the expression *self gets* $\{x = a\}$, where $a\colon \tau$, appears in the body of a method m of class c. If the instance variable x declared in c has type σ, then we must have $\tau \leq \sigma$ for the update to be legal, since the value denoted by a must be interpretable as a value of type σ in order for the assignment to be legal. Now suppose we attempt to define a subclass c' of c where the type of instance variable x is σ' for some $\sigma' < \tau$ (*i.e.*, $\sigma' \leq \tau$ and $\sigma' \neq \tau$). Then the update to x in the inherited method m will no longer be legal since a cannot be given type σ'.

The formal rule is that expressions which are evaluators (*i.e.*, they return values) can only be replaced with expressions whose types are subtypes of the type of the original, while expressions which are acceptors (*i.e.*, they receive values) can only be replaced by expressions whose types are supertypes of the type of the original. Since instance variables can appear in different contexts as evaluators and as acceptors, we may not change the type of instance variables when forming subclasses.

If instance variables were visible in objects, we would have the same restrictions on changing the types of instance variables for subtypes of object types as well. However, for a variety of reasons (including this restriction on subtyping), we choose to make the instance variables of objects invisible from outside of the object. Some reasons for this have to do with the advantages of information hiding. The user should not be provided with any unnecessary information on the implementation of objects of the language. This is standard practice in languages supporting abstract data types (ADT's), and should be standard practice in languages supporting objects and classes for similar reasons. (We note that Smalltalk and Eiffel adopt similar restrictions.)

However, there will be extra benefits to this decision in TOOPL. Because the interfaces (types) of objects do not mention instance variables, two objects can have the same type if their methods have the same types, even if they have different sets of instance variables. For example two objects can be of the same point type, even if one is represented using Cartesian coordinates (*e.g.*, has x and y as instance variables) and the other is represented with polar coordinates (*e.g.*, has r and θ as instance variables). While the bodies of all of the methods will be implemented differently, the two kinds of objects can be used interchangeably. Of course, in practice we would only want to use these objects interchangeably if the semantics of their methods were the same (with respect to a suitably abstract representation of points).

Similarly, the determination as to whether two object types are subtypes is independent of their instance variables. Thus the type of a color point object which is represented in Cartesian coordinates may be a subtype of the type of a point object which is represented in polar coordinates.

For those who wish to expose the instance variables, we note that it is easy to make an instance variable visible as either an evaluator or acceptor by providing appropriate methods. If $x{:}\,\sigma$ is an instance variable, we may provide methods, $getx{:}\,\sigma$ and $setx{:}\,\sigma \rightarrow MyType$, which function respectively as evaluators and acceptors for x. If only the first is provided, then σ can be replaced by a subtype in forming an object type which is a subtype of the original object type, while if only the second is provided, then σ can be replaced by a supertype. As expected, if both are provided, then σ cannot be uniformly replaced by either a subtype or supertype.

The addition of instance variables also adds some complications to the semantics of TOOPL as well as to the type-checking rules. We put off the discussion of these complexities until after the discussion of syntax and type checking below.

5.2 Syntax and Type-Checking Rules for TOOPL

In this section we present the syntax, subtyping, inheritance, and type-checking rules for TOOPL with instance variables. The following section presents the revised semantics. In this extension to our language, the types of classes will depend on the types of both instance variables and methods, but the types of objects will suppress all mention of instance variables. As discussed above, this gives us the added advantage that an object whose type is a subtype of another may have a completely different set of hidden instance variables.

The types for our new language follow:

Definition 5.1
Let $\mathcal{V}^{Tp}$ be an infinite collection of type variables, $\mathcal{L}$ be an infinite collection of labels, and $\mathcal{C}^{Tp}$ be a collection of type constants which includes at least the type constants Bool and Num. The type expressions with respect to $\mathcal{V}^{Tp}$ and $\mathcal{C}^{Tp}$ are defined by the following production:

$$\begin{aligned}\tau ::= \; & c \mid t \mid \tau \rightarrow \tau' \mid \{m_1{:}\,\tau_1; \ldots; m_n{:}\,\tau_n\} \mid ObjectType(MyType)\tau \mid \\ & (\sigma, \tau) \mid ClassType(MyType)(\sigma, \tau).\end{aligned}$$

In the above grammar, $c \in \mathcal{C}^{Tp}$ and $t \in \mathcal{V}^{Tp}$. Moreover, the σ's and τ's appearing in the last three terms of the production must be record types (*i.e.*, of the form $\{m_1{:}\,\tau_1; \ldots; m_n{:}\,\tau_n\}$).

The new types all arise in the last two clauses above. In those clauses the type expression σ is intended to stand for a record of instance variables, while τ is the type of the record of methods. We explain the intended meaning of these new types below.

Because the interfaces (types) of objects do not mention their instance variables, objects will actually have two different types: an *internal* type which includes the types of instance variables, and an *external* type which does not. Types of the form

$Refl(\leq)$ $$C \vdash \tau \leq \tau$$

$Var(\leq)$ $$C;\ t \leq \tau; C' \vdash t \leq \tau$$

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

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

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

$Pair(\leq)$ $$\frac{\sigma\ \mathbf{ext}\ \sigma',\ C \vdash \tau \leq \tau'}{C \vdash (\sigma, \tau) \leq (\sigma', \tau')}$$

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

where neither s nor t may occur free in C, τ, or τ'.

Fig. 8. Subtype Axioms and Rules for TOOPL.

(σ, τ) will describe internal types of objects, and are used in type-checking methods in order to ensure that all accesses to instance variables are type-safe. Types of this form will normally not appear in programs written in the language, but are a useful abstraction making it easier to express type-checking rules and the semantics of the language. Types of the form $\mathit{ObjectType}(\mathit{MyType})\tau$, which do not mention the types of instance variables, will continue to describe the external types of objects. $\mathit{ClassType}(\mathit{MyType})(\sigma, \tau)$ will denote the type of classes with records of instance variables of type σ and of methods of type τ.

There is very little difference between the old and new subtype and $\leq_{meth}$ axioms and rules in this extended language. However we must introduce a new relation, **ext** (read *extends*), between records of instance variables. The intended meaning of σ' **ext** σ is that σ' contains all of the fields of σ, and perhaps more. Moreover, all fields of σ have exactly the same types in σ'. This relation is more restrictive than subtyping since it does not allow fields of a record to be replaced by subtypes. Summarizing, for all $1 \leq k \leq n$ and all sets, $\{\sigma_i\}_{i \leq n}$, of types,

$$\{v_1\colon \sigma_1; \ldots; v_k\colon \sigma_k; \ldots; v_n\colon \sigma_n\}\ \mathbf{ext}\ \{v_1\colon \sigma_1; \ldots; v_k\colon \sigma_k\}.$$

Notice that this rule is independent of any type constraint system, C.

The complete collection of subtyping rules for TOOPL is given in Figure 8. Notice that the only change is the addition of the rule (*Pair(≤)*) for our new pair types.

$Var(\leq_{meth})$ $\qquad C;\ t \leq_{meth} \tau; C' \vdash t \leq_{meth} \tau$

$Refl(\leq_{meth})$ $\qquad C \vdash \mathit{ObjectType}(\mathit{MyType})\tau \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau$

$Trans(\leq_{meth})$ $\qquad \dfrac{C \vdash \gamma \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau, C \vdash \tau \leq \tau'}{C \vdash \gamma \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau'}$

where *MyType* does not occur free in C.

Fig. 9. Inheritance Relation Axioms and Rules for TOOPL.

In particular, the rule for object types is unchanged, because instance variables do not appear in the types of objects.

The two axioms and rule for the ordering $\leq_{meth}$ are unchanged. For completeness, however, we repeat them in Figure 9.

The type of an object will look different from the outside and the inside because TOOPL supports hidden instance variables. We indicated earlier that internal types are of the form (σ, τ). In the body of methods, *self* will need access to its own instance variables. Thus we will need a mechanism to assign an internal type to *self*. However, as with *MyType*, if a method is inherited in a subclass, the internal type of *self* will change. As a result we will need to assign *self* an internal type whose value may change in subclasses. To accomplish this we will provide *self* with bound variables to represent both an internal type, usually written *SelfType*, and an external type, usually written *MyType*, as before. In order to convert between these two we will also need to introduce a new bound term variable, $\mathit{close}: \mathit{SelfType} \rightarrow \mathit{MyType}$, representing a function which converts an object from its private type, *SelfType*, to its public type, *MyType*. It is not possible to go the other direction, since *close* represents information hiding.

Because *SelfType* is a private or secret type, it is not allowed to appear in the type of any method. As a result, we did not need to list it in the syntax of object types. We will list it in the typings of *self* and *close* in the header of class definitions.

Definition 5.2
The pre-terms of TOOPL are given by the following context-free grammar:

$$\begin{aligned}
M ::= \;& x \mid \mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ N \mid \mathit{fun}(v{:}\sigma)M \mid M\ N \mid M = N \mid \\
& \{m_1 = M_1, \ldots, m_n = M_n\} \mid e.m_i \mid \\
& \mathit{class}(\mathit{self}{:}\ \mathit{SelfType}; \mathit{close}{:}\ \mathit{SelfType} \rightarrow \mathit{MyType})(IV, MR) \mid \\
& \mathit{new}\ c \mid o.v_i \mid o\ \mathit{gets}\ \{v_i = a_i\} \mid o \Leftarrow m \mid \\
& \mathit{update}\ c\ \mathit{by}\ (\mathit{self}{:}\ \mathit{SelfType}; \mathit{close}{:}\ \mathit{SelfType} \rightarrow \mathit{MyType}; \mathit{super}) \\
& \qquad\qquad\qquad\qquad (\{v_1 = I_1'\}, \{m_1 = M_1'\}) \mid \\
& \mathit{extend}\ c\ \mathit{with}\ (\mathit{self}{:}\ \mathit{SelfType}; \mathit{close}{:}\ \mathit{SelfType} \rightarrow \mathit{MyType}; \mathit{super}) \\
& \qquad\qquad\qquad\qquad (\{v_{m+1} = I_{m+1}\}, \{m_{n+1} = M_{n+1}\}).
\end{aligned}$$

The new terms, $o.v_i$ and $o\ \mathit{gets}\ \{v_i = I_i\}$, provide access and updates to instance variables. The *class*, *update*, and *extend* rules have minor changes to reflect the addition of instance variables. Note that *self* now has type *SelfType* rather than

MyType, and application of the function *close* is required in order to convert an object of type *SelfType* to type *MyType*. An important restriction not expressible in the context-free grammar is that *self*, *SelfType*, and *close* may not occur in the expressions providing initial values for instance variables in *class*, *update*, and *extend* terms. This restriction allows us to simplify the semantics of instance variables.

Of course the restriction that each subclass extension or update must add or modify exactly one method and one instance variable is wholly unreasonable. We adopt that restriction here simply to make it easier to write down the type-checking rules and semantics. The actual language allows arbitrary numbers of methods and instance variables to be mentioned in extend or update terms.

The axioms and rules for deriving type assignments are given in Figures 10 and 11. As usual we say that a pre-term, M, is a term of TOOPL with respect to a given type constraint system, C, and syntactic type assignment E, if there exists a type τ such that $C, E \vdash M : \tau$.

The main changes in type checking in this section are reflected in the type-checking rules for the new terms $o.v_i$ and $o \; gets \; \{v_i = a_i\}$, which provide access to and updates for instance variables. Instance variables are only accessible for elements whose types are subtypes of types of the form (σ, τ). These rules will tend to be used most often where o is *self*. The class formation rules have minor changes to reflect the addition of the type variable, *SelfType*, and the term, *close*, which is used to convert terms to the form in which instance variables are hidden. The *update* and *extend* rules have minor changes to reflect the possibility of changing the value of or adding new instance variables. Again, recall that the type of an instance variable may not be changed in a subclass. However, one object type may be a subtype of another object type regardless of the instance variables that may be hidden within the objects themselves.

5.3 Semantics for TOOPL

In this section we provide new semantic definitions which reflect the modification of our language to support hidden instance variables. The changes to the semantics are relatively extensive since we must ensure both that the initial values of instance variables are not captured when objects are created, and that instance variables are not accessible outside of the object. We will introduce mechanisms to solve each of these problems separately and then combine them to provide the formal semantics for our language.

5.3.1 The semantics of methods in objects with instance variables

The addition of instance variables adds considerable complexity to the semantics of terms of TOOPL. The problem is the occurrence of references to instance variables in methods. If $self.x$ occurs in the body of a method, m, in a class, c, then $self.x$ should only be evaluated when the message m is actually sent to an object generated from c, *not* when the object is created by applying *new* to c.

Recall that the meanings of methods are provided when an object is constructed

Var
$$E \cup \{x\colon \tau\} \vdash x\colon \tau$$

Cond
$$\frac{E \vdash B\colon Bool,\ E \vdash M\colon \tau,\quad E \vdash N\colon \tau}{E \vdash if\ B\ then\ M\ else\ N\colon \tau}$$

Abs
$$\frac{E \cup \{v\colon \sigma\} \vdash M\colon \tau}{E \vdash fun(v\colon \sigma)M\colon \sigma \rightarrow \tau}$$

Appl
$$\frac{E \vdash M\colon \sigma \rightarrow \tau,\ N\colon \sigma}{E \vdash M\ N\colon \tau}$$

Eq
$$\frac{C, E \vdash M\colon Num,\quad C, E \vdash N\colon Num}{C, E \vdash M = N\colon Bool}$$

Record
$$\frac{E \vdash M_i\colon \tau_i \ \text{ for all } 1 \leq i \leq n}{E \vdash \{m_1 = M_1, \ldots, m_n = M_n\}\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}}$$

Proj
$$\frac{E \vdash R\colon \{m_1\colon \tau_1; \ldots m_n\colon \tau_n\}}{E \vdash R.m_i\colon \tau_i \ \text{ for all } 1 \leq i \leq n}$$

Class
$$\frac{\begin{array}{r} C;\ MyType \leq_{meth} ObjectType(MyType)\tau, E \vdash I\colon \sigma \\ C;\ MyType \leq_{meth} ObjectType(MyType)\tau;\ SelfType \leq (\sigma, \tau), \\ E \cup \{self\colon SelfType, close\colon SelfType \rightarrow MyType\} \vdash M\colon \tau \end{array}}{\begin{array}{r} C, E \vdash class(self\colon SelfType; close\colon SelfType \rightarrow MyType)(I, M)\colon \\ ClassType(MyType)(\sigma, \tau) \end{array}}$$

where *self* does not occur free in I, and *SelfType* does not occur free in σ or τ.

Object
$$\frac{C, E \vdash c\colon ClassType(MyType)(\sigma, \tau)}{C, E \vdash new\ c\colon ObjectType(MyType)\tau}$$

SetInst
$$\frac{C \vdash \gamma \leq (\{v_1\colon \sigma_1; \ldots; v_n\colon \sigma_n\}, \tau),\ C, E \vdash o\colon \gamma,\quad C, E \vdash a_i\colon \sigma_i}{C, E \vdash o\ gets\ \{v_i = a_i\}\colon \gamma}$$

Fig. 10. Type Assignment Axioms and Rules for TOOPL

from a class by taking a fixed point (which ensures that the meaning of *self* is the entire object). When we apply the *gets* operation to an object, an instance variable, and a value, we do not wish to take a new fixed point. Rather we just create another object with the same methods and the same values for its instance variables as before, except that the value of the appropriate instance variable is changed.

In order for this to work, we must ensure that the values of the instance variables are not captured in the fixed point when objects are constructed from classes. If they are captured, then they will essentially be compiled into the semantics of the methods, and changes to the values of instance variables through *gets* expressions will not be reflected in the methods of the resulting objects.

This is one place where it would be somewhat easier to work with an imperative

$$\textit{GetInst}\qquad \frac{C \vdash \gamma \leq (\{v_1{:}\,\sigma_1;\ldots;v_n{:}\,\sigma_n\},\tau),\ \ C,E \vdash o{:}\,\gamma}{C,E \vdash o.v_i{:}\,\sigma_i}$$

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

$$\textit{Update}\quad \frac{\begin{array}{r} C,E \vdash c{:}\,\textit{ClassType}(\textit{MyType})(\{v_1{:}\,\sigma_1;\ldots;v_m{:}\,\sigma_m\},\{m_1{:}\,\tau_1;\ldots;m_n{:}\,\tau_n\}),\\ C \vdash \tau_1' \leq \tau_1,\\ C;\textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\{m_1{:}\,\tau_1';m_2{:}\,\tau_2;\ldots;m_n{:}\,\tau_n\},\\ E \vdash I_1'{:}\,\sigma_1,\\ C;\ \textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\{m_1{:}\,\tau_1';m_2{:}\,\tau_2;\ldots;m_n{:}\,\tau_n\};\\ \textit{SelfType} \leq (\{v_1{:}\,\sigma_1;\ldots;v_m{:}\,\sigma_m\},\{m_1{:}\,\tau_1';m_2{:}\,\tau_2;\ldots;m_n{:}\,\tau_n\}),\\ E \cup \{\textit{self}{:}\,\textit{SelfType},\textit{close}{:}\,\textit{SelfType} \rightarrow \textit{MyType},\\ \textit{super}{:}\,\{m_1{:}\,\tau_1;\ldots;m_n{:}\,\tau_n\}\} \vdash M_1'{:}\,\tau_1'\end{array}}{\begin{array}{r} C,E \vdash \textit{update } c \textit{ by } (\textit{self}{:}\,\textit{SelfType};\textit{close}{:}\,\textit{SelfType} \rightarrow \textit{MyType};\textit{super})\\ (\{v_1 = I_1'\},\{m_1 = M_1'\}){:}\\ \textit{ClassType}(\textit{MyType})(\{v_1{:}\,\sigma_1;\ldots;v_m{:}\,\sigma_m\},\{m_1{:}\,\tau_1';m_2{:}\,\tau_2;\ldots;m_n{:}\,\tau_n\})\end{array}}$$

where *self* does not occur free in I, and *SelfType* does not occur free in σ or τ.

$$\textit{Extend}\quad \frac{\begin{array}{r} C,E \vdash c{:}\,\textit{ClassType}(\textit{MyType})(\{v_1{:}\,\sigma_1;\ldots;v_m{:}\,\sigma_m\},\{m_1{:}\,\tau_1;\ldots;m_n{:}\,\tau_n\}),\\ C;\ \textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\{m_1{:}\,\tau_1;\ldots;m_{n+1}{:}\,\tau_{n+1}\},\\ E \vdash I_{m+1}{:}\,\sigma_{m+1},\\ C;\ \textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\{m_1{:}\,\tau_1;\ldots;m_n{:}\,\tau_n;m_{n+1}{:}\,\tau_{n+1}\};\\ \textit{SelfType} \leq (\{v_1{:}\,\sigma_1;\ldots;v_{m+1}{:}\,\sigma_{m+1}\},\{m_1{:}\,\tau_1;\ldots;m_{n+1}{:}\,\tau_{n+1}\}),\\ E \cup \{\textit{self}{:}\,\textit{SelfType},\textit{close}{:}\,\textit{SelfType} \rightarrow \textit{MyType},\\ \textit{super}{:}\,\{m_1{:}\,\tau_1;\ldots;m_n{:}\,\tau_n\}\} \vdash M_{n+1}{:}\,\tau_{n+1}\end{array}}{\begin{array}{r} C,E \vdash \textit{extend } c \textit{ with } (\textit{self}{:}\,\textit{SelfType};\textit{close}{:}\,\textit{SelfType} \rightarrow \textit{MyType};\textit{super})\\ (\{v_{m+1} = I_{m+1}\},\{m_{n+1} = M_{n+1}\}){:}\\ \textit{ClassType}(\textit{MyType})\qquad\qquad\qquad\qquad\\ (\{v_1{:}\,\sigma_1;\ldots;v_{m+1}{:}\,\sigma_{m+1}\},\{m_1{:}\,\tau_1;\ldots;m_n{:}\,\tau_n;m_{n+1}{:}\,\tau_{n+1}\}\end{array}}$$

where *self* does not occur free in I, and *SelfType* does not occur free in σ or τ.

$$\textit{Subsump}\qquad \frac{C \vdash \sigma \leq \tau,\quad C,E \vdash M{:}\,\sigma}{C,E \vdash M{:}\,\tau}$$

Fig. 11. Type Assignment Axioms and Rules for TOOPL (Continued)

language, since we could simply capture the locations of the instance variables in the fixed point, rather than their values. However, we can simulate this procedure as follows.

We will interpret objects as pairs, one piece of which is the record of current values of instance variables, while the other represents the methods. A *gets* expression creates a new pair which is identical to the original, except that the appropriate component of the record of values of instance variables is updated. The value of an instance variable is obtained simply by extracting that component from the record of values of instance variables.

Sending messages, however, will be somewhat more complex. The second component of the object will *not* just be a record of values of methods. It will be a function which takes as an argument a record of values of instance variables, returning the record of methods in which the instance variables of *self* are interpreted by the values given in the parameter. Thus, if the record of instance variables has value 3 for component x, then the value of $\mathit{self}.x$ in the body of the methods will be 3. If the record of instance variables has type σ and the record of methods has type τ, the corresponding object will be a pair with type

$$\sigma \times (\sigma \rightarrow \tau).$$

When we send a message to an object, we apply the second component of the object to its first component, resulting in a record of methods of type τ in which the appropriate values for the instance variables have been inserted. The appropriate method is then extracted from this record. For example, if $\langle v, f\rangle$ is the meaning of object expression o, and m is the name of a method of o, the meaning of $o \Leftarrow m$ is obtained by extracting the m component of $f(v)$. This process of applying the second component to the record of current values of instance variables is similar to looking up the current values of instance variables in specified locations.

5.3.2 Representing hidden instance variables

We wish our semantics to reflect the fact that the values of instance variables of an object are not accessible outside of the methods of that object. In particular we wish to treat objects with the same method interface as elements of the same type, regardless of the structure of their instance variables. In terms of the representation introduced above for objects, we wish to find a way to treat types of the form $\sigma \times (\sigma \rightarrow \tau)$ and $\sigma' \times (\sigma' \rightarrow \tau)$ as being the same. In other words, we only care that their types are of the form $t \times (t \rightarrow \tau)$ for some t.

The existential types introduced by Mitchell and Plotkin (Mitchell and Plotkin, 1988) provide exactly the mechanism necessary to accomplish this. Existential types were originally introduced to hide the representation of a data type. (Mitchell and Plotkin, 1988) describes a very general mechanism for existential types which allows one to hide both the data type and the implementation of operations on that data type. As we have no need for hiding operations here, we describe a simpler version.

If τ is a type expression, then any element with type of the form $\tau[\sigma/t]$ can be "packed" into an element of type $\exists t.\tau$. Once packed, the element can no longer be used in a way which uses the fact that t is actually σ. One may "open" an element of an existential type in order to use it, but one is not able to obtain access to the representation of t through this operation. That is, one is able to apply only operations that do not depend on knowing the representation of t.

The following is a description of existential types as used in the higher-order lambda calculus. If t is a type variable and τ is a type then $\exists t.\tau$ is a type. Associated with existential types are two new term constructors, *pack* and *open* with the following typing rules. We presume that E is a syntactic type assignment for the higher-order lambda calculus. (Note that the typing rules given here are for the higher order lambda calculus, not the object-oriented programming language introduced in this paper.)

$$E \vdash pack\ t = \sigma\ in\ \tau \colon \tau[\sigma/t] \rightarrow \exists t.\tau$$

$$\frac{E \vdash M \colon \exists t.\tau, \quad E \cup \{x \colon \tau\} \vdash N \colon \rho}{E \vdash open\ M\ as\ x\ in\ N \colon \rho} \text{ if } t \text{ is not free in } \rho \text{ or } E.$$

These rules correspond exactly to the intuitive description of *pack* and *open* given above. In particular, the last rule states that if N is a term involving the free variable x of type τ, and M is a term of type $\exists t.\tau$, then it is possible to "open" M as a term of type τ and use it in place of x in evaluating N. Note that the restrictions on the construction of *open* and *pack* terms ensure that the representation of the packed term is not exposed when it is opened.

We apply this technique to the representation of objects suggested in the previous subsection. Suppose M is of type $\sigma \times (\sigma \rightarrow \tau)$, where t is not free in τ. Then

$$o = (pack\ t = \sigma\ in\ t \times (t \rightarrow \tau))M$$

has type $\exists t.t \times (t \rightarrow \tau)$. In order to actually use o in a computation, we will need to "open" it first. Once opened, we can apply the second component to the first component since this operation does not depend on the knowledge of what t really is. The result of sending the message m to o is given by

$$(open\ o\ as\ ob\ in\ (ob)_2(ob)_1)(m),$$

where the operations $(\cdots)_1$ and $(\cdots)_2$ extract the first and second components, respectively, from an ordered pair. That is, first we "open" o, giving it the name ob with type $t \times (t \rightarrow \tau)$. We can now apply $(ob)_2$ to $(ob)_1$, giving a result of type τ, which is the record of methods in which *self* denotes the current values of o's instance variables. Finally the method m is extracted from that record.

In the following sections we will use existential types to represent the meanings of object types of our language. As suggested earlier, *SelfType* will denote the internal type of an object, while *MyType* will denote the type once it has been "closed" or "packed".

5.3.3 The meaning of types in TOOPL

As usual, our semantics are given with respect to a model, $\mathcal{A}$, of the F-bounded second-order polymorphic lambda calculus with recursive types and elements. The meaning of $\leq$ continues to be given by the ordering $\leq_{\mathcal{A}}$ of the model. We do have to provide the semantics for the relation **ext** introduced in section 5.2.

Since **ext** simply represents extending a record with extra fields, it can be interpreted very simply by the relation, *ext* , defined as follows.

Definition 5.3
Let π and π' be record types. Then $\pi\ ext\ \pi'$ if and only if $\pi \mid Dom(\pi') = \pi$, where $Dom(\pi)$ is the set of labels of the record type π, and $\pi \mid Dom(\pi')$ is the restriction of π to those labels which occur in π'.

$[\![c]\!]\rho = \mathcal{A}_c$ for $c \in \mathcal{C}^{Tp}$.

$[\![t]\!]\rho = \rho(t)$ for $t \in \mathcal{V}^{Tp}$.

$[\![\sigma \rightarrow \tau]\!]\rho = [\![\sigma]\!]\rho \rightarrow [\![\tau]\!]\rho$.

$[\![\{m_1\colon \tau_1, \ldots, m_n\colon \tau_n\}]\!]\rho = \prod_{m_i \in \{m_1,\ldots,m_n\}} [\![\tau_i]\!]\rho$.

$[\![(\sigma, \tau)]\!]\rho = [\![\sigma]\!]\rho \times ([\![\sigma]\!]\rho \rightarrow [\![\tau]\!]\rho)$

$$[\![\mathit{ClassType}(\mathit{MyType})(\sigma,\tau)]\!]\rho = \prod_{\mu \leq_A [\![\tau]\!]\rho'(\mu)} \prod_{\nu\ \mathit{ext}\ [\![\sigma]\!]\rho'(\mu)} ([\![\sigma]\!]\rho'(\mu) \times (\xi^S(\nu,\mu) \rightarrow [\![\tau]\!]\rho'(\mu)))$$

where $\xi^S(\nu,\mu) = \nu \times (\nu \rightarrow \mu)$, $\xi^P(\mu) = \exists v.(v \times (v \rightarrow \mu))$, and $\rho'(\mu) = \rho[\xi^P(\mu)/\mathit{MyType}]$.

$[\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho = \mathit{FIX}(\lambda\xi^P.\exists v.[\![(t,\tau)]\!]\rho[\xi^P/\mathit{MyType}, v/t])$

Fig. 12. The Semantics of Type Expressions in TOOPL.

Thus $\pi\ \mathit{ext}\ \pi'$ if and only if all of the labels of π' are included in π, while those that they share have exactly the same type. This is the semantic equivalent to our syntactic restriction on subtypes with instance variables.

The definition of the semantics of types is given in Figure 12. For completeness we include the definitions of non-object-oriented types given earlier.

Recall that the type (σ, τ) represents the internal type of an object with record of instance variables with type σ and record of methods of type τ. As discussed above, the meaning of these types are given as a pair, where the first element of the pair is the record of values of instance variables and the second element is a function from a record of values of instance variables to the record of methods.

The object type, $\mathit{ObjectType}(\mathit{MyType})\tau$, represents the external type of objects whose internal types are of the form (σ, τ). Since the type of the record of instance variables is hidden outside of the object, the type σ is "quantified out" via an existential type. Since we must also ensure that *MyType* stands for the object's type, we must also take a fixed point. Unwinding the fixed point, the semantics of object types can be rewritten as:

$$\begin{aligned}
&[\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho \\
&\qquad = \exists v.([\![(t,\tau)]\!]\rho[[\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho/\mathit{MyType}, v/t]) \\
&\qquad = \exists v.(v \times (v \rightarrow [\![\tau]\!]\rho[[\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho/\mathit{MyType}])).
\end{aligned}$$

Thus an object type is represented by an existential type in which the type of the record of instance variables has been quantified out and *MyType* is interpreted as the entire type. When we define the meaning of objects, we will see that they are formed by "packing" elements of types of the form (σ, τ).

The meaning of $[\![\mathit{ClassType}(\mathit{MyType})(\sigma,\tau)]\!]\rho$ is a direct generalization of that given in Section 4.3.1 for SOOPL. The types $\mu \leq [\![\tau]\!]\rho'(\mu)$ and $\nu\ \mathit{ext}\ [\![\sigma]\!]\rho'(\mu)$ determine the types of the records of methods and instance variables in an object type obtainable as a subclass of a class of type $\mathit{ClassType}(\mathit{MyType})(\sigma,\tau)$. In such a subclass, the meaning of *MyType* will be $\xi^P(\mu) = \exists v.(v \times (v \rightarrow \mu))$, while the meaning of *SelfType* will be $\nu \times (\nu \rightarrow \mu)$. The class will have two components: a record of type $[\![\sigma]\!]\rho'(\mu)$ containing initial values of instance variables, and a function which will take an element of type *SelfType* and return the record of methods with that element of *SelfType* plugged in as the meaning of *self*. Notice that both σ and τ (the types of the records of instance variables and methods) are interpreted in an environment, $\rho'(\mu)$, in which in which *MyType* is interpreted as $\xi^P(\mu)$.

We next define the notion of consistency of a pair C, E with an environment, ρ. In order to do this, we must provide a meaning for $\leq_{\mathcal{A}}$ when relating the interpretations of internal and external object types, (σ, τ) and *ObjectType*(*MyType*)τ. Neither pairs nor existential types appeared in standard models of the second-order lambda calculus. Nonetheless, we wish to have models which interpret these constructs and such that the interpretation, $\leq_{\mathcal{A}}$, of $\leq$ satisfies our axioms and rules.

We do not have space here to describe in detail the construction of such models, but we can indicate the behavior of the coercion function **convert** on elements of the appropriate types. Since **convert**$[\tau'][\tau]$ is defined only for $\tau \leq_{\mathcal{A}} \tau'$ we shall be satisfied here with defining the appropriate **convert** function for these types.

A complicating factor in satisfying the new subtyping rule for terms of the form (σ, τ) is that $[\![\sigma]\!]\rho$ occurs in both covariant and contravariant positions in the above expansion of $[\![(\sigma, \tau)]\!]\rho$. This is unfortunate since, in general, one cannot find a coercion function if a variable which occurs in a contravariant position is replaced by a subtype. In this case, however, since both occurrences of σ must change in tandem, and only by extensions rather than general subtypes, it is possible to define such a function.

Suppose σ **ext** σ' and $C \vdash \tau \leq \tau'$. Then for all $d \in \mathcal{A}^{[\![\sigma]\!]\rho}$ and $g \in \mathcal{A}^{[\![\sigma]\!]\rho} \rightarrow \mathcal{A}^{[\![\tau]\!]\rho}$ (*i.e.*, for all $\langle d, g\rangle \in \mathcal{A}^{[\![(\sigma,\tau)]\!]\rho}$), define **convert**$[[\![(\sigma', \tau')]\!]\rho][[\![(\sigma, \tau)]\!]\rho]\langle d, g\rangle = \langle d', g'\rangle$, where $d' = d \mid Dom(\mathcal{A}^{[\![\sigma']\!]\rho})$ and g' is defined as follows. For all $e \in \mathcal{A}^{[\![\sigma']\!]\rho}$, define $g'(e) =$ **convert**$[[\![\tau']\!]\rho][[\![\tau]\!]\rho](g(e'))$ where $e' \in \mathcal{A}^{[\![\sigma]\!]\rho}$ is defined so that for all labels $l \in Dom(\mathcal{A}^{[\![\sigma']\!]\rho}), e'(l) = e(l)$, and $e'(l) = d(l)$, otherwise. In other words, g' "compiles in" the components of d which were discarded when d' was extracted from d. Thus if $C \vdash \sigma$ **ext** σ' and $C \vdash \tau \leq \tau'$ then $[\![(\sigma, \tau)]\!]\rho \leq_{\mathcal{A}} [\![(\sigma', \tau')]\!]\rho$.

The coercion function for existential types is much simpler. Suppose $C \vdash \tau \leq \tau'$ and $C, E \vdash M\colon \exists t.\tau$. Then

$$\mathbf{convert}[\exists t.\tau'][\exists t.\tau](M) = \\ \qquad open\ M\ as\ M_{open}\ in\ ((pack\ t = t\ in\ \tau')(\mathbf{convert}[\tau'][\tau](M_{open})))$$

That is, we open up M as an element of type τ, convert it to type τ', and then pack it up again. Thus if $C \vdash \tau \leq \tau'$ then $[\![\exists t.\tau]\!]\rho \leq_{\mathcal{A}} [\![\exists t.\tau']\!]\rho$

With $\leq_{\mathcal{A}}$ defined on the interpretations of internal and external views of object types, we are ready to proceed with our new definition of an environment being consistent with a type constraint system.

Definition 5.4
The following inductive definition determines when a type environment ρ is *consistent* with a type constraint system C:

1. If C is empty then ρ is consistent with C.

2. Suppose $C;\ t \leq \sigma$ is a type constraint system. If ρ is consistent with C and $\rho(t) \leq_{\mathcal{A}} [\![\sigma]\!]\rho$ then ρ is *consistent* with $C;\ t \leq \sigma$.
3. Suppose $C;\ t \leq_{meth}\ \mathit{ObjectType}(\mathit{MyType})\tau$ is a type constraint system. If ρ is consistent with C and $\rho(t) \leq_{\mathcal{A}} \exists v.(v \times (v \rightarrow [\![\tau]\!]\rho[\rho(t)/\mathit{MyType}]))$ then ρ is *consistent* with $C;\ t \leq_{meth}\ \mathit{ObjectType}(\mathit{MyType})\tau$.

The definition of the consistency of an environment, ρ, with E is as before. A proof by induction on the length of derivations similar to that given for SOOPL shows that the type constraint rules preserve the meanings of the orderings.

Lemma 5.5
Suppose the type environment ρ is consistent with C.

1. If $C \vdash \sigma \leq \tau$ then $[\![\sigma]\!]\rho \leq_{\mathcal{A}} [\![\tau]\!]\rho$.
2. If $C \vdash \gamma \leq_{meth}\ \mathit{ObjectType}(\mathit{MyType})\tau$ then
$$[\![\gamma]\!]\rho \leq_{\mathcal{A}} \exists v.(v \times (v \rightarrow [\![\tau]\!]\rho[[\![\gamma]\!]\rho/\mathit{MyType}]))$$

Proof
For part 1, we only need concern ourselves with the rules involving object types: those of the form (σ, τ) for the internal view of the object, and those of the form $\mathit{ObjectType}(\mathit{MyType})\tau$ for the external view. But the coercion functions for each of these kinds of types were presented before the definition of consistent type environments. It is easy to see that for each of these rules, if the coercions promised in the hypotheses hold, then the appropriate coercions exist for the conclusions.

The proof of part 2 proceeds essentially as for SOOPL. The added complications due to the use of existential types for object types are taken care of since the operation of appending existential quantifiers to types preserves the subtyping relation. $\square$

5.3.4 The meaning of terms in TOOPL

We specify the meaning of terms of TOOPL in Figures 13 and 14. For completeness, we include the semantics of regular as well as object-oriented terms. The definition of the semantics with the addition of the instance variables is extremely complex. We provide the intuition behind these definitions below. We urge the reader to review the corresponding definitions for SOOPL in section 4.3.2 and compare them to the definitions given here.

Remark 5.6
The reader may find it helpful to skip this and the following section on the first reading of this paper. When reading this section we urge the reader to have pencil, paper, and a large quantity of caffeinated beverages at hand for help in working through this very complex material.

Class terms: As usual, the meaning of a class,

$$c = \mathit{class}(\mathit{self}\colon \mathit{SelfType};\ \mathit{close}\colon \mathit{SelfType} \rightarrow \mathit{MyType})(I, M)\colon \mathit{ClassType}(\mathit{MyType})(\sigma, \tau)$$

$[\![C, E \vdash x\colon \tau]\!]\rho = \rho(x).$

$$[\![C, E \vdash \mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ N\colon \tau]\!]\rho = \begin{cases} [\![C, E \vdash M\colon \tau]\!]\rho, & \text{if } [\![C, E \vdash B\colon \mathit{Bool}]\!]\rho, \\ [\![C, E \vdash N\colon \tau]\!]\rho, & \text{if not } [\![C, E \vdash B\colon \mathit{Bool}]\!]\rho, \\ \bot, & \text{otherwise.} \end{cases}$$

$[\![C, E \vdash \mathit{fun}(v\colon \sigma)\ M\colon \sigma \to \tau]\!]\rho = \lambda d \in \mathcal{A}^{[\![\sigma]\!]\rho}.[\![C, E \vdash M\colon \tau]\!]\rho[d/v].$

$[\![C, E \vdash MN\colon \tau]\!]\rho = ([\![C, E \vdash M\colon \sigma \to \tau]\!]\rho)([\![C, E \vdash N\colon \sigma]\!]\rho).$

$$[\![C, E \vdash M = N\colon \mathit{Bool}]\!]\rho = \begin{cases} \bot, & \text{if } [\![C, E \vdash M\colon \tau]\!]\rho = \bot \text{ or } [\![C, E \vdash N\colon \tau]\!]\rho = \bot, \\ \mathit{true}, & \text{if } [\![C, E \vdash M\colon \tau]\!]\rho = [\![C, E \vdash N\colon \tau]\!]\rho \neq \bot, \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$[\![C, E \vdash \{m_1 = M_1, \ldots, m_n = M_n\}\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}]\!]\rho = f,$

where $\mathit{dom}(f) = \{m_1, \ldots, m_n\}$ and for $1 \leq i \leq n, f(m_i) = [\![C, E \vdash M_i\colon \tau_i]\!]\rho.$

$[\![C, E \vdash R.m_i\colon \tau_i]\!]\rho = ([\![C, E \vdash R\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}]\!]\rho)\ (m_i).$

$$\begin{aligned} &[\![C, E \vdash \mathit{class}(\mathit{self}\colon \mathit{SelfType}; \mathit{close}\colon \mathit{SelfType} \to \mathit{MyType})(I, M)\colon \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathit{ClassType}(\mathit{MyType})(\sigma, \tau)]\!]\rho = \\ &\qquad \lambda\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'(\mu).\lambda\nu\ \mathit{ext}\ [\![\sigma]\!]\rho'(\mu). \\ &\qquad\qquad \langle [\![C', E \vdash I\colon \sigma]\!]\rho'(\mu), \lambda o \in \mathcal{A}^{\xi^S(\nu,\mu)}.[\![C'', E'' \vdash M\colon \tau]\!]\rho''(\mu, \nu, o)\rangle, \end{aligned}$$

where $C' = C;\ \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau,$
$C'' = C';\ \mathit{SelfType} \leq (\sigma, \tau),$
$E'' = E \cup \{\mathit{self}\colon \mathit{SelfType}, \mathit{close}\colon \mathit{SelfType} \to \mathit{MyType}\},$
$\xi^P(\mu) = \exists v.(v \times (v \to \mu)),$
$\xi^S(\nu, \mu) = \nu \times (\nu \to \mu),$
$\mathit{cl}(\nu, \mu) = \lambda o \in A^{\xi^S(\nu,\mu)}.\mathit{pack}[v = \nu\ \mathit{in}\ v \times (v \to \mu)]\ o.$
$\rho'(\mu) = \rho[\xi^P(\mu)/\mathit{MyType}]$
$\rho''(\mu, \nu, o) = \rho'(\mu)[\xi^S(\nu, \mu)/\mathit{SelfType}, o/\mathit{self}, \mathit{cl}(\nu, \mu)/\mathit{close}]$

$[\![C, E \vdash \mathit{new}\ c\colon \mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho = \mathit{pack}[v = \nu\ \mathit{in}\ v \times (v \to \mu)]\ \langle \mathit{inst}, \mathit{methfun}\rangle,$

where $\xi^P = [\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho,$
$\rho' = \rho[\xi^P/\mathit{MyType}],$
$\nu = [\![\sigma]\!]\rho',$
$\mu = [\![\tau]\!]\rho',$
$\langle \mathit{inst}, \mathit{meth}_{approx}\rangle = ([\![C, E \vdash c\colon \mathit{ClassType}(\mathit{MyType})(\sigma, \tau)]\!]\rho)\ (\mu)\ (\nu),$
$\mathit{methfun} = \mathit{Fix}(\lambda f \in \mathcal{A}^{\nu \to \mu}.\lambda w \in \mathcal{A}^{\nu}.\mathit{meth}_{approx}(\langle w, f\rangle)).$

$[\![C, E \vdash o \Leftarrow m\colon \tau[\gamma/\mathit{MyType}]]\!]\rho = \mathit{open}\ o'\ \mathit{as}\ \mathit{ob}\ \mathit{in}\ (\mathit{ob})_2(\mathit{ob})_1\ (m),$

where $o' = \mathbf{convert}[\exists v.(v \times (v \to [\![\{m\colon \tau\}]\!]\rho[[\![\gamma]\!]\rho/\mathit{MyType}]))][[\![\gamma]\!]\rho][\![C, E \vdash o\colon \gamma]\!]\rho.$

$[\![C, E \vdash o\ \mathit{gets}\ \{v_i = I_i\}\colon \gamma]\!]\rho = \langle (\mathit{ob})_1[[\![C, E \vdash I_i\colon \sigma_i]\!]\rho/v_i], (\mathit{ob})_2\rangle,$

where $\mathit{ob} = [\![C, E \vdash o\colon \gamma]\!]\rho.$

$[\![C, E \vdash o.v_i\colon \sigma_i]\!]\rho = ([\![C, E \vdash o\colon \gamma]\!]\rho)_1\ (v_i).$

Fig. 13. Semantics of Terms in TOOPL.

$$[\![C, E \vdash \mathit{update}\ c\ \mathit{by}\ (\mathit{self}\colon \mathit{SelfType}; \mathit{close}\colon \mathit{SelfType} \to \mathit{MyType}; \mathit{super})\ (\{v_1 = I'_1\}, \{m_1 = M'_1\})\colon$$
$$\mathit{ClassType}(\mathit{MyType})(\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}, \{m_1\colon \tau'_1; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\})]\!]\rho =$$
$$\lambda\mu \leq_{\mathcal{A}} [\![\{m_1\colon \tau'_1; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\}]\!]\rho'(\mu).$$
$$\lambda\nu\ \mathit{ext}\ [\![\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}]\!]\rho'(\mu).\langle \mathit{inst}, \lambda o \in \mathcal{A}^{\xi^S(\nu,\mu)}.\mathit{meth}\rangle,$$

where $C' = C;\ \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\{m_1\colon \tau'_1; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\}$,
$C'' = C';\ \mathit{SelfType} \leq (\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}, \{m_1\colon \tau'_1; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\})$,
$E'' = E \cup \{\mathit{self}\colon \mathit{SelfType}, \mathit{close}\colon \mathit{SelfType} \to \mathit{MyType}, \mathit{super}\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}\}$,
$\xi^P(\mu) = \exists v.(v \times (v \to \mu))$,
$\xi^S(\nu, \mu) = \nu \times (\nu \to \mu)$,
$\rho'(\mu) = \rho[\xi^P(\mu)/\mathit{MyType}]$,
$\mathit{cl}(\nu, \mu) = \lambda o \in A^{\xi^S(\nu,\mu)}.\mathit{pack}[v = \nu\ \mathit{in}\ v \times (v \to \mu)]\ o$,
$\mathit{sup} = ([\![C, E \vdash c\colon \mathit{ClassType}(\mathit{MyType})(\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}, \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\})]\!]\rho)(\mu)\ (\nu)$,
$\mathit{supmeth} = (\mathit{sup})_2\ (o)$,
$\mathit{supvar} = (\mathit{sup})_1$,
$\mathit{inst} = \mathit{supvar}[[\![C', E \vdash I'_1\colon \sigma_1]\!]\rho'(\mu)/v_1]$,
$\mathit{dom}(\mathit{meth}) = \{m_1, \ldots, m_n\}$,
$\mathit{meth}(m_1) = [\![C'', E'' \vdash M'_1\colon \tau'_1]\!]\rho'(\mu)[\xi^S(\nu, \mu)/\mathit{SelfType}, o/\mathit{self}, \mathit{cl}(\nu, \mu)/\mathit{close}, \mathit{supmeth}/\mathit{super}]$,
and for $2 \leq j \leq n$,
$\mathit{meth}(m_j) = \mathit{supmeth}(m_j)$.

$$[\![C, E \vdash \mathit{extend}\ c\ \mathit{with}\ (\mathit{self}\colon \mathit{SelfType}; \mathit{close}\colon \mathit{SelfType} \to \mathit{MyType}; \mathit{super})\ (\{v_{m+1} = I_{m+1}\}, \{m_{n+1} = M_{n+1}\})\colon$$
$$\mathit{ClassType}(\mathit{MyType})(\{v_1\colon \sigma_1; \ldots; v_{m+1}\colon \sigma_{m+1}\}, \{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\})]\!]\rho =$$
$$\lambda\mu \leq_{\mathcal{A}} [\![\{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\}]\!]\rho'(\mu).$$
$$\lambda\nu\ \mathit{ext}\ [\![\{v_1\colon \sigma_1; \ldots; v_{m+1}\colon \sigma_{m+1}\}]\!]\rho'(\mu).\langle \mathit{inst}, \lambda o \in \mathcal{A}^{\xi^S(\nu,\mu)}.\mathit{meth}\rangle,$$

where $C' = C;\ \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\}$,
$C'' = C';\ \mathit{SelfType} \leq (\{v_1\colon \sigma_1; \ldots; v_{m+1}\colon \sigma_{m+1}\}, \{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\})$,
$E'' = E \cup \{\mathit{self}\colon \mathit{SelfType}, \mathit{close}\colon \mathit{SelfType} \to \mathit{MyType}, \mathit{super}\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\})\}$,
$\xi^P(\mu) = \exists v.(v \times (v \to \mu))$,
$\xi^S(\nu, \mu) = \nu \times (\nu \to \mu)$,
$\rho'(\mu) = \rho[\xi^P(\mu)/\mathit{MyType}]$,
$\mathit{cl}(\nu, \mu) = \lambda o \in A^{\xi^S(\nu,\mu)}.\mathit{pack}[v = \nu\ \mathit{in}\ v \times (v \to \mu)]\ o$,
$\mathit{sup} = ([\![C, E \vdash c\colon \mathit{ClassType}(\mathit{MyType})(\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}, \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\})]\!]\rho)(\mu)\ (\nu)$,
$\mathit{supmeth} = (\mathit{sup})_2\ (o)$,
$\mathit{supvar} = (\mathit{sup})_1$,
$\mathit{inst} = \mathit{supvar}[[\![C', E \vdash I_{m+1}]\!]\rho'(\mu)/v_{m+1}]$,
$\mathit{dom}(\mathit{meth}) = \{m_1, \ldots, m_{n+1}\}$,
$\mathit{meth}(m_{n+1}) = [\![C'', E'' \vdash M_{n+1}\colon \tau_{n+1}]\!]\rho'(\mu)[\xi^S(\nu, \mu)/\mathit{SelfType}, o/\mathit{self}, \mathit{cl}(\nu, \mu)/\mathit{close}, \mathit{supmeth}/\mathit{super}]$,
and for $1 \leq j \leq n$,
$\mathit{meth}(m_j) = \mathit{supmeth}(m_j)$.

$[\![C, E \vdash M\colon \sigma]\!]\rho = \mathbf{convert}[[\![\sigma]\!]\rho][[\![\tau]\!]\rho]([\![C, E \vdash M\colon \tau]\!]\rho)$, if $C \vdash \tau \leq \sigma$ and $C, E \vdash M\colon \tau$.

Fig. 14. Semantics of Terms in TOOPL (continued).

must provide the meaning of methods for all possible subclasses. Let

$$\xi_0 =_{def} [\![ObjectType(MyType)\tau]\!]\rho = \exists v.(v \times (v \rightarrow [\![\tau]\!]\rho[\xi_0/MyType]))$$

In order for $ClassType(MyType)(\sigma', \tau')$ to be the type of a subclass, c', of c, we must have σ' **ext** σ and $C \vdash \tau' \leq \tau$. In particular, we must have

$$C \vdash ObjectType(MyType)\tau' \leq_{meth} ObjectType(MyType)\tau. \qquad (1)$$

Let

$$\xi^P =_{def} [\![ObjectType(MyType)\tau']\!]\rho = \exists v.(v \times (v \rightarrow [\![\tau']\!]\rho[\xi^P/MyType])))). \qquad (2)$$

By Lemma 5.5, the inequality (1) implies that

$$\xi^P \leq_{\mathcal{A}} \exists v.(v \times (v \rightarrow [\![\tau]\!]\rho[\xi^P/MyType])). \qquad (3)$$

Expanding ξ^P using (2), we see that the only way inequality (3) can hold is if

$$\mu =_{def} [\![\tau']\!]\rho[\xi^P/MyType] \leq_{\mathcal{A}} [\![\tau]\!]\rho[\xi^P/MyType]. \qquad (4)$$

It follows that if ξ^P is the meaning of an object type generated by a subclass of $ClassType(MyType)(\sigma, \tau)$, then inequality (4) must hold. Thus if we write $\rho'(\mu)$ for $\rho[\xi^P/MyType]$, then we must have $\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'(\mu)$.

What about the meaning of the type of the record of instance variables, σ', of the subclass? Since σ' may involve the variable, *MyType*, we interpret σ' in an environment, $\rho'(\mu)$, in which *MyType* is interpreted as ξ^P. Since we can only extend the record of instance variables, it follows that σ' **ext** σ. If ν is the meaning of the type of the record of instance variables of c' then

$$\nu =_{def} [\![\sigma']\!]\rho'(\mu) \ ext \ [\![\sigma]\!]\rho'(\mu). \qquad (5)$$

Thus if ξ^P is the meaning of an object type arising from a subclass of a class of type $ClassType(MyType)(\sigma, \tau)$, the inequalities (4) and (5) must hold. If μ and ν are defined as above, then the meaning of *MyType* is $\xi^P = \exists v.(v \times (v \rightarrow \mu))$, while the meaning of *SelfType* is $\xi^S = [\![(\sigma', \tau')]\!]\rho[\xi^P/MyType] = \nu \times (\nu \rightarrow \mu)$. The meaning of a class term is a function which takes a μ representing the possible meaning of the type of the record of methods in a subclass, a ν representing the possible meaning of the type of the record of instance variables, and an o which is the interpretation of *self*. Then

$$([\![class(self\colon SelfType; close\colon SelfType \rightarrow MyType)(I, M)]\!]\rho)\ (\mu)\ (\nu) = \\ \langle [\![I]\!]\rho'(\mu), \lambda\ o \in \mathcal{A}^{\xi^S}.[\![M]\!]\rho''(\mu, \nu, o)\rangle$$

where $\rho''(\mu, \nu, o) = \rho'(\mu)[\xi^S/SelfType, o/self, cl(\nu, \mu)/close]$. The term $cl(\nu, \mu)$ is the obvious function which takes an element of the internal type, *SelfType*, and packs it into an element of the external type, *MyType*.

Object constructors: In order to interpret *new c*, we must supply the interpretation of c with the appropriate μ and ν representing the interpretations of the types for methods and instance variables, take a fixed point to get *self* to refer to the object as a whole, and then "pack" it in order to hide the record of instance variables.

As mentioned earlier, though, we must be sure not to "compile in" the initial values of the instance variables in the methods when we take the fixed point. Let $\xi^P =_{def} [\![ObjectType(MyType)\tau]\!]\rho = \exists v.(v \times (v \rightarrow [\![\tau]\!]\rho[\xi^P/MyType]))$. The meaning of *new c* will be an element of this type. In order to compute the meaning of *new c*, let $\mu = [\![\tau]\!]\rho[\xi^P/MyType]$ and $\nu = [\![\sigma]\!]\rho[\xi^P/MyType]$, where the type of c is $ClassType(MyType)(\sigma, \tau)$. Note that $[\![c]\!]\rho$ can be legally applied to μ and ν since they satisfy the appropriate constraints.

Let $\langle inst, meth_{approx}\rangle = [\![c]\!]\rho\ (\mu)\ (\nu)$. That is, *inst* is the meaning of the record of initial values of instance variables from c, while $meth_{approx}$ is a function which takes an object, o', whose type is the internal type of the objects generated by the class, and returns the meaning of the record of methods, in which *self* is interpreted as o'. If o is the meaning of *new c*, then it is clear that its first component should just be the record of values of instance variables, *inst*.

Now, the second component of the object is, in fact, supposed to be a function which takes a record of values of instance variables of type ν, and returns the meaning of the record of methods where *self* is interpreted by the object as a whole. Unfortunately, however, $meth_{approx}$ takes an argument of type $\xi^S = \nu \times (\nu \rightarrow \mu)$, rather than just ν. Thus we define,

$$methfun = Fix(\lambda f \in \mathcal{A}^{\nu \rightarrow \mu}.\lambda w \in \mathcal{A}^{\nu}.meth_{approx}(\langle w, f\rangle)).$$

Rewriting by unwinding the fixed point, we get

$$methfun = \lambda w \in \mathcal{A}^{\nu}.meth_{approx}(\langle w, methfun\rangle)),$$

which is an element of type $\nu \rightarrow \mu$. Thus, if i is a record of values of instance variables, then

$$methfun(i) = meth_{approx}(\langle i, methfun\rangle)$$

But this latter term is simply the meaning of the record of methods of the class where *self* is interpreted as $\langle i, methfun\rangle$, *i.e.*, an object with instance variable values given by i, and methods interpreted the same way as objects generated directly from the class. This, of course, is exactly what we wanted! Finally, we must "pack" the pair $\langle inst, methfun\rangle$ in order to hide the instance variables.

Summarizing, the meaning of *new c* is a pair in which:

1. The first element is the record of values of instance variables given in c.
2. The second is a function, which, when applied to a record of values for the instance variables, gives the meaning of the record of methods in which *self* is interpreted to have the instance variables provided, and has the same methods as before.
3. This pair is "packed" in order to hide the first component, the record of instance variables and its type.

Message sending: Given the definition of *new c*, message sending is now relatively straightforward. If o has type $ObjectType(MyType)\{m{:}\,\tau\}$ and $o' = [\![o]\!]\rho$ then

$$[\![o \Leftarrow m]\!]\rho = open\ o'\ as\ ob\ in\ (ob)_2(ob)_1(m).$$

As we saw above, the second component of the object must be applied to the current values of the instance variables in order to obtain the record of values of methods. Since o' has the instance variables hidden, it must first be opened as ob before the second component can be applied to the first. Note that this application will always be well-defined no matter what the type of the instance variables.

We may not know the exact type of the receiving object (particularly if this object is *self*), only that the type, γ, of o satisfies $\gamma \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\{m\colon \tau\}$. Thus, as with SOOPL, we may need to coerce the meaning of the object to type $\mathit{ObjectType}(\mathit{MyType})\{m\colon \tau\}$ before opening and applying the second component to the first.

Updating instance variables: The semantics of terms of the form $o\ \mathit{gets}\ \{v_i = I_i\}$ and $o.v_i$ are relatively straightforward in either updating or extracting values from the first component of the object (which represents the record of values of instance variables). Note that o must have a type which corresponds to the internal type of an object in order for these operations to be applicable. That is, an object can only access or manipulate its own instance variables.

Summing up, the key difference in the semantics of terms in TOOPL with instance variables is that objects have two types, an internal type (labeled *SelfType*) which is of the form $\sigma \times (\sigma \rightarrow \tau)$, and an external type of the form $\exists t.t \times (t \rightarrow \tau)$, in which the types of instance variables have been hidden. Thus the interpretation of $\mathit{new}\ c$ is given as a recursively-defined pair in which the values of instance variables have been "packed" in an existential type. Operations on instance variables of an object can only be done when the object's type is an internal object type. When a message is sent to an object, the object is "opened" so that the second component can be applied to the first in order to provide the current values of instance variables to the methods.

5.4 Soundness of typing

We again wish to show that our semantic definitions are consistent with our typing rules in order to show that the definitions of the meaning of terms given in Figures 13 and 14 make sense. The proof of this theorem is very similar to the one for SOOPL in Section 4.4.

Theorem 5.7

(Soundness of semantics with respect to the typing rules for TOOPL with hidden instance variables) Let $\mathcal{A}$ be a model for the F-bounded second-order lambda calculus with fixed points at both the type and term levels. For all typing derivations, $C, E \vdash M\colon\sigma$, if ρ is consistent with $C,\ E$, then $[\![C, E \vdash M\colon\sigma]\!]\rho \in \mathcal{A}^{[\![\sigma]\!]\rho}$.

Proof

The proof is by induction on the length of derivations. The only interesting cases are those involving the object-oriented terms.

$\underline{class(self\colon SelfType;\, close\colon SelfType \to MyType)(I, M)\colon}$ Suppose

$$C, E \vdash class(self\colon SelfType;\, close\colon SelfType \to MyType)(a, e)\colon \\ ClassType(MyType)(\sigma, \tau)$$

follows from $C', E \vdash I\colon \sigma$ and $C'', E'' \vdash M\colon \tau$, for C', C'', and E'' as defined in the semantics of class terms in Figure 13. Then

$$[\![C, E \vdash class(self\colon SelfType;\, close\colon SelfType \to MyType)(I, M)\colon \\ ClassType(MyType)(\sigma, \tau)]\!]\rho = \\ \lambda\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'(\mu).\lambda\nu\ ext\ [\![\sigma]\!]\rho'(\mu). \\ \langle [\![C', E \vdash I\colon \sigma]\!]\rho'(\mu), \lambda o \in \mathcal{A}^{\xi^S(\nu,\mu)}.[\![C'', E'' \vdash M\colon \tau]\!]\rho''(\mu, \nu, o)\rangle,$$

for $\xi^S(\nu, \mu)$, $\xi^P(\mu)$, $\rho'(\mu)$, and $\rho''(\mu)$ also as defined as in Figure 13.

We claim $\rho'(\mu) =_{def} \rho[\xi^P(\mu)/MyType]$ is consistent with C' for all $\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'(\mu)$. By assumption, ρ is consistent with C, so we need only show that $\xi^P(\mu) \leq_{\mathcal{A}} \exists v.(v \times (v \to [\![\tau]\!]\rho'(\mu)))$. But this follows from the fact that $\xi^P(\mu) = \exists v.(v \times (v \to \mu))$ where $\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'(\mu)$.

We also claim that $\rho''(\mu, \nu, o) =_{def} \rho'(\mu)[\xi^S(\nu, \mu)/SelfType, o/self, cl(\nu, \mu)/close]$ is consistent with C'', E''. It suffices to show that $\xi^S(\nu, \mu) \leq_{\mathcal{A}} [\![(\sigma, \tau)]\!]\rho''(\mu, \nu, o) = [\![(\sigma, \tau)]\!]\rho'(\mu)$ (since *SelfType* does not occur in (σ, τ)). But $\xi^S(\nu, \mu) = \nu \times (\nu \to \mu)$, and $[\![(\sigma, \tau)]\!]\rho'(\mu) = [\![\sigma]\!]\rho'(\mu) \times ([\![\sigma]\!]\rho'(\mu) \to [\![\tau]\!]\rho'(\mu))$, for $\nu\ ext\ [\![\sigma]\!]\rho'(\mu)$ and $\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'(\mu)$. As discussed in Section 5.3.3, this implies that $\xi^S(\nu, \mu) \leq_{\mathcal{A}} [\![(\sigma, \tau)]\!]\rho''$. It is easy to show that $\rho''(\mu, \nu, o)$ is consistent with E''. The rest of the verification that the meaning of a class term is in the appropriate class type follows easily from the definitions of semantics of class terms and types.

$\underline{new\ c\colon}$ Suppose $C, E \vdash new\ c\colon ObjectType(MyType)\tau$ follows from the hypothesis $C, E \vdash c\colon ClassType(MyType)(\sigma, \tau)$. Recall that

$$[\![C, E \vdash new\ c\colon ObjectType(MyType)\tau]\!]\rho = \\ pack[v = \nu\ in\ v \times (v \to \mu)]\ \langle inst, methfun\rangle$$

for ν, μ, $inst$, $meth_{approx}$, $methfun$, and ξ^S defined in the semantics of $new\ c$ in Figure 13. It is easy to check that $inst \in \mathcal{A}^{[\![\sigma]\!]\rho'(\mu)}$, $meth_{approx} \in \mathcal{A}^{\xi^S \to \mu}$, and hence that $methfun \in \mathcal{A}^{\nu \to \mu}$. Thus,

$$pack[v = \nu\ in\ v \times (v \to \mu)]\ \langle inst, methfun\rangle \in \mathcal{A}^{\exists v.v \times (v \to \mu)}.$$

But, $\mathcal{A}^{\exists v.v \times (v \to \mu)} = \mathcal{A}^{[\![ObjectType(MyType)\tau]\!]\rho}$, so we are done.

$\underline{o \Leftarrow m\colon}$ It is easy to see that $[\![o']\!]\rho \in \mathcal{A}^{\exists v.(v \times (v \to [\![\{m\colon\tau\}]\!]\rho[[\![\gamma]\!]\rho/MyType]))}$ for o' as given in Figure 13 for the definition of

$$[\![C, E \vdash o \Leftarrow m\colon \tau[\gamma/MyType]]\!]\rho = open\ o'\ as\ ob\ in\ (ob)_2(ob)_1\ (m).$$

By the definition of the *open* construct, *ob* is in $\mathcal{A}^{\nu \times (\nu \to [\![\{m\colon\tau\}]\!]\rho[[\![\gamma]\!]\rho/MyType])}$ for some (unknown) type ν. Thus $(ob)_2(ob)_1\ (m) \in \mathcal{A}^{[\![\tau]\!]\rho[[\![\gamma]\!]\rho/MyType]}$ as desired.

The cases for extracting and updating instance variables are straightforward. Those for updating and extending classes are tedious, but similar to that for classes, and are omitted here. $\Box$

We now find ourselves with a much more expressive language. While we cannot change the types of instance variables in subclasses, the subtype relation is totally independent of instance variables. This is yet another reason to separate the subtype and subclass hierarchies in object-oriented languages. In the next section we provide several examples of classes and computations on objects in order to illustrate the power of the language.

6 Sample programs in TOOPL

In this section we illustrate the use of our language through several sample programs. All sample programs will be written in TOOPL, the full language with hidden instance variables.

The language as presented so far used an abstract syntax that was convenient for writing type-checking rules and writing semantic definitions. Here we modify the syntax somewhat in order to write programs in a style which is easier to read. It is straightforward to rewrite these sample programs in the formal language presented earlier.

We make the following simplifications to syntax:

1. We allow multiple simultaneous extensions or updates to classes. Since a given construct may be a combination of updates and extensions, we shall use the keyword *modify* in place of *update* and *extend*. We indicate those methods and instance variables which are being changed by listing them in a "redefining" clause as in Eiffel (Meyer, 1988).
2. We write $f(x{:}\,T_1, y{:}\,T_2) = e$ to abbreviate $f = fun(x{:}\,T_1)\ fun(y{:}\,T_2)\ e$.
3. We omit the list of bound variables, *self*, *super*, *close*, *SelfType*, and *MyType* in term and type definitions. This will not cause problems because we will not nest class definitions.
4. To break up the different parts of the definitions, we will use the reserved words **var** and **methods** as headers for the list instance variables and methods of a class. We use labeled **end**'s to terminate compound definitions of terms and types.
5. We allow identifiers to stand for terms and types in other expressions.

These changes should result in more readable programs and suggest the form of a real language built from a core of TOOPL.

Our first example is of a point class. This example differs from that given in Section 4 in that it uses hidden instance variables.

```
PointClass =
  class
    var
      x = 0,
      y = 0
  methods
      mv(dx,dy:Num) = close(self gets
                               {x = self.x + dx, y = self.y + dy}),
      getx = self.x,
      gety = self.y
  end class.
```

Note the use of `close` in the definition of the `mv` method of `PointClass`. This is required so that the result type of `mv` involves only `MyType` and not `SelfType`.

```
PointClassType =
  classType
    var
      x,y:Num
    methods
      mv:Num -> Num -> MyType;
      getx,gety:Num
  end classType.

PointType =
  objectType
    methods
      mv:Num -> Num -> MyType;
      getx,gety:Num
  end objectType.
```

`PointClass` has the type `PointClassType`, and objects generated via `new` from `PointClass` have the type `PointType`. The fact that instance variables are hidden is reflected in the fact that the types of instance variables do not appear in `PointType`. We leave it as an exercise to the reader to design a class `PolarPtClass` whose instance variables are polar coordinates, `r`, θ, yet objects generated from `PolarPtClass` have type `PointType` above.

Now we can generate some subclasses. We begin by adding an `eq` method to `PointClass`.

```
EqPtClass =
  modify PointClass by
    methods
      eq(p:MyType) = (self.x = (p <= getx)) & (self.y = (p <= gety))
  end class.

EqPtClassType =
  classType
    var
      x,y:Num
    methods
      mv:Num -> Num -> MyType;
      getx,gety:Num;
      eq:MyType -> Bool
  end classType.

EqPtType =
  objectType
    methods
      mv:Num -> Num -> MyType;
      getx,gety:Num;
      eq:MyType -> Bool
  end objectType.
```

Alternatively we can extend `PointClass` by simply adding a new instance variable, `color`.

```
ColorPtClass =
  modify PointClass by
    var
      c = Red
    methods
      getcolor = self.c
  end class.

ColorPtClassType =
  classType
    var
      x,y:Num;
      c:Color
    methods
      mv:Num -> Num -> MyType;
      getcolor:Color;
      getx,gety:Num
  end classType.

ColorPtType =
  objectType
    methods
      mv:Num -> Num -> MyType;
      getcolor:Color;
      getx,gety:Num
  end objectType.
```

We could also obtain a `ColorPolarPtClass` by simply replacing `PointClass` by `PolarPtClass` in the definition of `ColorPtClass`. The type of this new class would also be `ColorPtClassType`.

Finally we can combine these two extensions into a color point class which contains an `eq` method. We could obtain this by modifying either `ColorPtClass` or

`EqPtClass`, since our language does not support multiple inheritance. Notice the use of `super` in the definition of `ColorEqPtClass`.

```
ColorEqPtClass =
  modify EqPtClass
    redefining eq by
    var
      c = Red
    methods
      getcolor = self.c,
      eq(p:MyType) = super.eq(p) & (self.c = (p <= getcolor))
  end class.
```

```
ColorEqPtClassType =
  classType
    var
      x,y:Num;
      c:Color
    methods
      mv:Num -> Num -> MyType;
      getcolor:Color;
      getx,gety:Num;
      eq:MyType -> Bool
  end classType.
```

```
ColorEqPtType =
  objectType
    methods
      mv:Num -> Num -> MyType;
      getcolor:Color;
      getx,gety:Num;
      eq:MyType -> Bool
  end objectType.
```

Now we can examine the relations between the object types defined above. Everything works as expected with the $\leq_{meth}$ relation. That is:

$$\texttt{ColorEqPtType} \leq_{meth} \texttt{EqPtType},$$

$$\texttt{ColorEqPtType} \leq_{meth} \texttt{ColorPtType},$$

$$\texttt{ColorEqPtType} \leq_{meth} \texttt{PointType},$$

$$\texttt{EqPtType} \leq_{meth} \texttt{PointType}, \text{ and}$$

$$\texttt{ColorPtType} \leq_{meth} \texttt{PointType}.$$

However, the subtype relation is more restricted because of the contravariant occurrence of `MyType` in the type of `eq`. Thus, types involving `eq` will not be supertypes of other types.

$$\texttt{ColorEqPtType} \leq \texttt{ColorPtType},$$
$$\texttt{ColorEqPtType} \leq \texttt{PointType},$$
$$\texttt{EqPtType} \leq \texttt{PointType}, \text{ and}$$
$$\texttt{ColorPtType} \leq \texttt{PointType}$$

The only loss is that `ColorEqPtType` is *not* a subtype of `EqPtType`.

Notice that we can also define parameterized classes. For example,

```
PointClassGenerator(a,b:Num) =
  class
    var
      x = a,
      y = b
    methods
      mv(dx,dy:Num) = close(self gets
                                {x = self.x + dx, y = self.y + dy}),
      getx = self.x,
      gety = self.y
  end class.
```

As another example, we could write a function `AddColorToClass` which takes a class of type `PointClassType` and returns a class of type `ColorPointClassType`. This function would have a body which is an easy variant of `ColorPtClass` given earlier.

We now present a few examples of simple functions which use these classes. Our first example is a function which takes two points and returns the midpoint of the line between them. It always returns an object of type `PointType`. For readability we introduce "`let`" expressions into our language. That is, we write "`let x:T = a in e`" as an abbreviation for `(fun (x:T) e) (a)`.

```
MidPt(p1,p2:PointType)=
  let newx:Num = ((p1 <= getx) + (p2 <= getx))/2,
    newy:Num = ((p1 <= gety) + (p2 <= gety))/2
  in
    new (PointClassGenerator(newx,newy))
  end function.
```

Note that we can apply this function to elements whose types are subtypes of `PointType`, though the answer will always be of type `PointType`.

If we wish to return an answer of the same type as, for example, the first parameter, we will have to extend our language to support bounded polymorphism. To support this, we will allow functions to take a type parameter, which may be bounded above by another type. For example, suppose the type `UpdatePtType` is a subtype of `PointType` with a method `setxy:Num -> Num -> MyType` which updates the x and y instance variables to the values of the two parameters. Then we can rewrite the above example as:

```
PolyMidPt(T ≤ UpdatePtType; p1:T; p2:PointType) =
  let newx:Num = ((p1 <= getx) + (p2 <= getx))/2,
    newy:Num = ((p1 <= gety) + (p2 <= gety))/2
  in
    (p1 <= setxy)(newx)(newy)
  end function.
```

We can write the type of `PolyMidPt` as `Forall (T ≤ UpdatePtType) (T -> PointType -> T)`, where `Forall (T ≤ UpdatePtType)` indicates that the first argument of the function is a subtype of `UpdatePtType`.

Of course, if `PolyMidPt` were written as a method of `PointClass`, we could have written it without the polymorphism (and without the use of `setxy`). We illustrate this with the definition of `MidPointClass`.

```
MidPointClass =
  modify PointClass by
    methods
      MidPt(p:PointType) = close (self gets
                                        {x =((self.x) + (p <= getx))/2,
                                        y = ((self.y) + (p <= gety))/2})
  end class.

MidPointClassType =
  classType
    var
      x,y:Num
    methods
      mv:Num -> Num -> MyType;
      getx,gety:Num;
      MidPt:PointType -> MyType
  end classType.
```

Of course, it would be easy to change this so that `MidPt` has type `MyType -> MyType`, however it would likely be more flexible to leave the formal parameter as `PointType` since this will allow the use of a parameter with any subtype of `PointType`, even when `MidPt` is inherited in subclasses of `MidPointClass`.

7 Possible extensions

In this section we discuss a few possible extensions to our language. These include recursion, (bounded) polymorphism, information hiding, a "MyClass" construct, and multiple inheritance. We also make a few brief remarks on "deferred classes".

Finally we discuss some essential additions necessary in order to obtain a language in which one can actually write useful programs.

Several powerful features were omitted from the language described here, even though they could be easily have been added. This was necessary in order to obtain a language that was as simple as possible, yet included all of the key features normally found in object-oriented languages.

The two most obvious features to add are recursion and polymorphism. The reason for this is that the semantics of the language is given in terms of a model of the F-bounded higher-order lambda calculus with contains elements corresponding to recursive terms, types, and type constructors. (We only used the second-order parts of the model above, but the model in (Bruce and Mitchell, 1992) includes higher-order elements as well.) As a result, we can add both recursion and (higher-order) F-bounded polymorphism with few semantic problems.

Interestingly, the language presented already supports an indirect form of recursion in objects because of the (implicitly) recursive meaning of *self.* In essence, all methods of a given class are defined by mutual recursion. The following simple example should illustrate the point. Define

$$\begin{aligned} &FactClass = class(self\colon MyType; close\colon SelfType \rightarrow MyType) \\ &\qquad (\{\}, \{fact = fun(n\colon int)\ if\ n = 0\ then\ 1\ else\ n * ((self \Leftarrow fact)(n \perp 1))\}. \end{aligned}$$

If $factobj = new\ FactClass$, then $f(n) = (factobj \Leftarrow fact)(n)$ is the factorial function. Mutually recursive functions can be defined as members of the same class. Thus, while explicit recursive function definitions could be added to TOOPL, they are already expressible via mutually recursive methods in classes and objects. Explicit recursion can be added either through explicit fixed point operators or some other syntax (for instance, a `letrec` operator). We prefer not to explicitly add recursively-defined types because the subtyping rules are more complex. We note that object types are effectively recursively defined because of the use of *MyType.*

The PolyMidPt procedure in the previous section provides a good example of the syntax for an extension to the language involving bounded polymorphism. In the more formal syntax given earlier in the paper we would write:

$$\frac{C; t \leq \sigma, E \vdash M\colon \tau,\ \text{for t not free in C or E}}{\text{C, E} \vdash \text{fun(t} \leq \sigma\text{) M: Forall (t} \leq \sigma\text{)}\ \tau}$$

$$\frac{C, E \vdash M\colon Forall\ (t \leq \sigma)\ \tau, \quad C \vdash \sigma' \leq \sigma}{C, E \vdash M[\sigma']\colon \tau[\sigma'/t]}$$

Interestingly, research in progress suggests that bounded polymorphism using $\leq_{meth}$ rather than $\leq$ may be even more valuable than ordinary bounded polymorphism.

We could also add unbounded polymorphism to the language without problem. The semantics of this extension is analogous to that for ordinary function abstraction and application and contains no surprises (after all, our underlying model supports exactly these features). More details can be found in (Bruce and Longo, 1990).

In Section 5 we discussed how to implement a language design that hid instance variables outside of an object, and, conversely, how one could make these hidden instance variables accessible by adding methods to access and update their values. One might also want to hide certain methods from external access. For instance, it is quite common in abstract data types to have internal functions which depend heavily on the implementation, and are not accessible except to the routines of the ADT. Selected methods can be hidden using the same techniques as used above for hidden instance variables.

Long inheritance chains in object-oriented languages often result in methods being inherited which are irrelevant or confusing to users of a class. Some object-oriented languages provide two different interfaces to the outside world: one to the users of the objects generated from the class, and another to inheriting classes. This could be added to the language presented here in the same way that hidden instance variables were provided. Methods could be hidden from users by "packing" them during the "new" operation. (See the semantics of "new" in section 5.3.4.) Selected methods and instance variables could also be hidden from subclasses by "packing" them in the semantics of the class definition.

We are less happy with our modeling of *super* than with other parts of our language. The bound variable, *super*, denotes the record of methods from the superclass. (We could also include a variable *supervar* which denotes the record of initial values of instance variables from the superclass, but this seemed less useful and so was omitted.) It would be more natural to represent this as an object which included the values of the instance variables of the current object and the methods of the superclass, but we found it difficult to formulate this properly. Moreover, we have recently concluded that this may not be the best way to obtain access to methods of the superclass.

A preferable solution might be to adopt the mechanism used in Eiffel, which is to rename inherited methods which are overridden, and then add these renamed methods to those of the subclass. Thus, for instance, if method m is defined in class c, then in defining a subclass c' which overrides the inherited m, we can rename the original m as *oldm*. The method *oldm* would then be available inside c. One could then send message *oldm* to *self* or any other expression of type *MyType*.

The only drawback to this solution is that one would usually not like to export the renamed method outside of the class. If we had added selective export to the language presented here, we would have provided this mechanism to handle calls to methods of the superclass. Because of the added complexity, we did not add this feature here, leaving us with this less satisfactory way of handling references to methods of the superclass. However, we expect eventually to implement this other way of handling *super*.

(Cook *et al.*, 1990) introduced the idea of a "*MyClass*" keyword which is analogous to our "*MyType*". "*MyClass*" would be a bound variable which would stand for the class which constructed the object. That paper gave several examples showing the usefulness of this construct. Unfortunately, close examination of the construct defined there shows that it has only part of the functionality that might be expected for a class. That is, it could be used to generate new objects, but it could not be used to create subclasses from that class.

Unfortunately, it appears that such a flexible construct for constructing subclasses would be extremely difficult to type. When "*MyClass*" occurs in the body of a method, the system may only assume that the objects generated from "*MyClass*" contain at least the methods of the original defining class, and that the types of these methods are subtypes of those given in the class. In particular, it is not known whether a method name not occurring in the original class occurs in the object which is then executing the method. Thus one can not tell in an *update* expression whether the new method is already defined in the class. Even more serious, if the new method name does occur in the current class, its type is not known, so it is impossible to tell if the update is legal or not.

The more complex record calculus presented in (Cardelli and Mitchell, 1990) may help solve this problem, as it provides both positive and negative information on the presence of particular methods. It was decided not to move to such a system here since the added complexities seem to outweigh the benefits. The use of "*MyType*" to denote the type of the current object seems to provide more (though different) benefits than "*MyClass*".

There are several ways to enhance the language by allowing more flexibility in using methods from either multiple parents or more remote ancestors. For example, a language supporting multiple inheritance allows a subclass to be defined with multiple parents.

The main difficulty with multiple inheritance seems to be determining which method to inherit if a method name occurs in more than one parent class. We see this as mainly a design question, rather than a technical one. For instance, it would be relatively straightforward to write the semantics of subclasses with multiple inheritance if the default were always to inherit from the first (or, alternatively, last) class listed in which the method occurs. Our preference would be to require that if a method name occurs in more than one parent then the user must specify explicitly from which parent to inherit the method. Again, it is relatively straightforward to write the semantics for such a form of multiple inheritance.

One could also allow a method body to refer to a method in any given ancestor by specifying the name of that ancestor. For example if the class `PointClass` occurred as an ancestor (even as one of several parents) of the current class, one could write `PointClass.m` (or similar notation) to indicate that method `m` from class `PointClass` is the one which should be invoked. We expect that such an addition to the language would be relatively straightforward as well.

Many object-oriented programming languages support "deferred" classes. These are classes which include declarations for instance variables and names for methods, but do not include the bodies for the methods. A programmer can define multiple subclasses of the deferred class whose implementations of the methods are unrelated. The advantages of deferred classes arise when the deferred class name is used to specify the type of a parameter to a function. Any subclasses of the deferred class may then be passed in as the actual parameters.

In our language we have separated the role of interface and implementation. Since a deferred class really specifies only the interface of the class, its role is provided for in our language by types. One may easily define an object type, yet not provide any classes to generate objects of that type. If a parameter is declared to have that type, objects of any subtype may be used as the actual parameters.

The language specified in this paper cannot handle easily what might be called "semi-deferred classes." These are classes in which some method bodies are specified while others are only given by their interfaces. It might be possible to model such classes by defining parameterized classes, which take the implementations of the "missing" methods as parameters. However, we have not investigated this in any detail.

The language described here allows the user to have instance variables with type *MyType*, but we have not provided any mechanism here for the user to define values of this type for instance variables. (Recall we do not allow the use of *self* in specifying the value of instance variables. At the cost of complicating the semantics of classes and objects, we could have allowed *self* to occur in expressions defining initial values of instance variables, but they were not particularly useful, and were omitted.) An example of a class definition which needs these facilities would be a binary tree class. The natural definition of this class would include instance variables representing the left and right subtrees, and these would most naturally have type *MyType*. In order to represent finite trees we need to be able to specify initial values for the instance variables which indicate the absence of subtrees.

Of course, what is really desired is to initialize these instance variables with an initial default value (corresponding to the use of *nil* in languages using pointers). The two major alternatives are:

1. Introduce a new object type *NIL* whose only element is *nil*, such that *NIL* is a subtype of every other object type.
2. Introduce a new type *UNIT* whose only element is *unit* and replace instance variable types of the form *MyType* by the union of *UNIT* and *MyType*.

Each of these solutions has some problems. The first requires that *NIL* essentially have an infinite type, since it must respond to any possible message. Moreover its response to any message should be an error value. This is not desirable in a strongly typed language.

The second solution has the advantage that a typecase statement could be provided which would not allow type-unsafe access. The corresponding disadvantage is that accesses to instance variables must be guarded by typecase statements and that disjoint unions must be added to the language (though these would probably be required in a real language anyway). Moreover many operations that used to return the original object type must now return the union type. It may be possible to disguise some of this complexity with cleverly chosen notations, but it will still not be completely straightforward.

Rather than choosing either of these alternatives here, we left this issue open for further investigation. (The alternatives described here were adopted from suggestions in (Black, 1992).) Any reasonable solution will also likely involve the introduction of a mechanism such as exceptions to handle errors.

Our purpose in this paper was to set out a language which included the main concepts of object-oriented languages as commonly understood in the programming language community. While our language has several features not commonly found in object-oriented languages (*e.g.*, separate subtype and subclass hierarchies, the bound variable "*MyType*" for the type of an object, etc.), they were virtually required by the demands of a type-safe language.

Essentially all non-OO-specific features were dropped from the language for simplicity. It should be clear to the reader that the common programming language features which were omitted from this treatment (for example, numeric operators) can be added easily in order to provide the features necessary for a truly usable language. As noted above, more complex features such as recursion and bounded polymorphism can also be added without great difficulty. Of course, the surface syntax of the language needs to be reworked in order to make it easier for programmers to read and write programs in the language. The examples in the previous section come closer to that goal than the formal syntax given earlier in the paper. We have implemented a variant of TOOPL with a more friendly syntax. Work is continuing on simplifying the concrete syntax of the language.

8 Summary

This paper has described a paradigmatic, functional statically-typed object-oriented language and its semantics. The language presented includes constructs for classes, objects, methods, (hidden) instance variables, subtypes, and inheritance. Moreover the language constructs for defining methods in classes include bound variables representing "*self*", "*SelfType*" and "*MyType*" (the internal and external types of "*self*"), and "*super*" (for references to methods of the superclass). The language with hidden instance variables includes both internal and external names for the type of "*self*" as well as an operator to "*close*" an object by making the instance variables inaccessible.

The most innovative features of the language presented here are its (provable) type-safety, the separation of subtype and inheritance hierarchies, the support for multiple implementations for objects of the same type, the inclusion in the language of names for the internal and external types of an object, the provision for type-safe use of updatable instance variables, and mechanisms for supporting the hiding of instance variables.

Static type-checking rules were given for the language and the type-checking rules were proved sound with respect to the semantics. The existence of models for the F-bounded higher-order lambda calculus with fixed points for elements, types, and type constructors made it possible to provide such a proof. We feel strongly that any language which is claimed to be type safe should include such a proof of the soundness of its type-checking rules with respect to a model of its semantics.

The advantages of static type checking are well-known. These range from guaranteeing that the programmer will not see error messages of the form "message not understood" at run-time to providing the compiler with extra information which can be used to optimize the compiled code. In particular, run-time type checking can be eliminated, providing the possibility for significant savings during program execution.

A major advantage of the type-checking rules presented here is that once the methods of a class have been type checked, they do not need to be type checked again when they are inherited in subclasses. This makes it possible to distribute a library of classes as a collection of executable code and types, while keeping the source code private.

The statically-typed language presented here supports the full variety of subclasses commonly found in object-oriented programs. While this freedom in forming subclasses forced us to separate the subtype and subclass hierarchies, we find the separation of the hierarchies more attractive than the other alternatives. Because the notions of subtype and inheritance are distinct (relating to interface versus implementation), it makes more sense to create separate language mechanisms to support them, rather than to artificially identify them. Moreover, the type-checking rules provided with this language (in particular the uses of the subtype and $\leq_{meth}$ relations) make clear where the subtyping and inheritance hierarchies are each being used.

An important consequence of the separation of the subtyping and subclass hierarchies is that there may be objects of the same type which have entirely different implementations. For example, because of the hidden instance variables, a program may contain some points which are represented in Cartesian coordinates, and others which are represented in polar coordinates. As long as their interfaces are the same, they will share the same type. Similarly, there need be no connection between the implementations of objects whose types are in the subtype relation. In short, subtypes need not come from subclasses, and subclasses need not result in subtypes.

The ability to refer to the type of an object within the bodies of its methods is very important in a strongly-typed object-oriented language. For instance, in Eiffel one can refer to the type of the current object with the phrase "like Current." The inclusion in our language of a bound variable (typically designated as "*MyType*" in our presentation) to stand for the type of the current object provides the opportunity for relatively fine-grained type checking.

The addition of instance variables to the language required great care in defining subtyping rules. Because of the existence of an operator to update instance variables (actually the operator creates a new copy of the object with a different value for the instance variable), we could not allow the types of instance variables to be changed in subclasses. As noted in (Bruce and Longo, 1990) and (Cardelli and Mitchell, 1990), an update operator is not definable in the standard presentation of bounded second-order lambda calculus. Our restrictions on subclasses, while adding some complexity to the type-checking rules, allow us to overcome this limitation.

The addition of hidden instance variables to the language required the provision of two different types for objects: a type variable (typically written as "*SelfType*" in this paper) representing the type of the object as viewed from the inside (*i.e.*, with its instance variables), and another type variable (typically written as "*MyType*") for the type of the object from the outside (without its instance variables). An operator "*close*" was provided which converts an object from type "*SelfType*" to "*MyType*" by hiding the instance variables. Existential types, as presented in

(Mitchell and Plotkin, 1988), were used to provide the semantics of this mechanism.

While minor variations of the semantic definitions presented here are possible, the semantics given here faithfully represent the intended meanings of the basic constructs of object-oriented languages. It is hoped that the careful semantic definitions given here will increase the reader's understanding of these notions, especially complex notions like inheritance.

While the language supports all of the fundamental object-oriented constructs, it has been kept as simple as possible. For instance, the language does not support recursion or polymorphism. However, the semantics of the language are given in a model of the higher-order F-bounded polymorphic lambda calculus. As noted in Section 7, this makes it relatively easy to add such advanced constructs to the language.

Despite the correctness of the definitions given here, the complexity of the semantics is troubling. This complexity arises in several ways. First of all, the presence of "*self*" requires that objects be defined recursively. Similarly, the presence of "*MyType*" as the type of "*self*" leads us to define object types as the solutions of recursive equations. Thus fixed points are required in this presentation at both the element and type level, even when no explicit recursion occurs in the method bodies of the class generating the object (of course any reference to "*self*" or "*MyType*" in a method body represents an implicit recursive reference). While fixed points certainly raise the level of difficulty in understanding these constructs, simple fixed points are probably well within the capability of most programmers to understand.

The construct which seems most complex, and hence most troubling, is that of inheritance. In a language with inheritance, a class must specify (at least implicitly) the meaning of all of its methods in all possible subclasses. From a theoretical point of view it is most useful to think of classes as parameterized generators for fixed points. One consequence of this is that the meaning of an inherited method, m, in a subclass may be quite different from that in the superclass when another method, m', on which it depends is redefined. This happens because when the fixed point is taken of the subclass, each occurrence of m' in the body of m is replaced by the new definition of m'. This may have a major impact on the meaning of the inherited m.

As a result, a programmer creating a subclass may not blindly override one method and expect it to have no impact on the others. In particular, a programmer expecting to modify classes contained in a library may require more information about the implementation of the library class than might otherwise be expected.

As a result of this complexity, I believe that the use of inheritance should be limited as much as possible. Inheritance has clear advantages in prototyping systems. However there appear to be real dangers of excessive complexity when long subclass chains are left in large systems. Others (see (Liskov, 1988; Snyder, 1986), for example) have cautioned about the dangers of loss of encapsulation in inheritance. A related problem is that minor changes in a superclass can invalidate method definitions lower in the subclass hierarchy.

In contrast, the addition of subtyping to the language represents an important addition of capability with only a small increase in complexity. Moreover, while the packing and unpacking operations which occur in the semantics of hidden instance variables look a bit complex, they are relatively straightforward in their semantic import.

9 Comparison with previous work

There has been a great deal of interest over the last several years in the design of object-oriented programming languages. There are now several object-oriented languages which are being used to produce commercial software. These include Simula (Birtwistle *et al.*, 1973), Smalltalk (Goldberg and Robson, 1983), Eiffel (Meyer, 1988), and Trellis/Owl (Schaffert *et al.*, 1986), as well as extensions to older programming languages such as C++ (Stroustrop, 1986), Oberon (Wirth, 1988), Modula 3 (Cardelli *et al.*, 1988), Object Pascal (Tesler, 1985), and various extensions of LISP ((Moon, 1986), for example).

There has also been considerable interest in the theoretical community in designing object-oriented languages with clean semantics. The major influences on the design work described in this paper have come from Cook and his collaborators in the (late and lamented) ABEL project at HP Labs (Canning *et al.*, 1989; Cook, 1989b; Canning *et al.*, 1989; Cook *et al.*, 1990) and the work of Luca Cardelli, John Mitchell and their collaborators (Cardelli, 1988a; Cardelli and Wegner, 1985; Cardelli, 1988b; Cardelli and Longo, 1991; Cardelli and Mitchell, 1990). Readers familiar with the work of the ABEL project will recognize its influence on the language presented here. The influence of Cardelli and Mitchell may not be as directly obvious, but has played a strong role in the evolution of the language design. The work of Peter Wegner (Wegner, 1990) was helpful in clarifying the differences between various approaches to object-oriented programming languages. In spite of its known problems with type safety, the language Eiffel, designed by Bertrand Meyer, has also had a great influence on my thinking about object-oriented programming languages.

The definition of the semantics of inheritance in a typed language comes from (Cook *et al.*, 1990), which in turn follows from the work on the semantics of inheritance in untyped languages in (Cook, 1989a; Cook and Palsberg, 1989; Reddy, 1988; Kamin, 1988). Mitchell (1990a) proposed an alternative, more operational, semantics for typed object-oriented languages, which was shown to be equivalent to that of (Cook *et al.*, 1990) in (Bruce, 1992). (Cardelli and Wegner, 1985), following earlier work in (Mitchell, 1988) (originally published as (Mitchell, 1984)), introduced the notion of bounded quantification in higher-order languages, which was extended in (Canning *et al.*, 1989) to handle the more general "F-bounded" quantification used here in the semantics of class definitions. (Bruce and Longo, 1990) (an early version of which appeared as (Bruce and Longo, 1988)) provided the first careful formal semantics of subtyping (using PER models) in this language. Later work by (Amadio, 1991) and (Cardone, 1989) provided models supporting subtypes and recursively defined elements and types. (Abadi and Plotkin, 1990) provided a more category-theoretic construction which also resulted in types which are CPO's. Recently, (Bruce and Mitchell, 1992) strengthened these constructions to ones which provide the existence of higher-order recursive type constructors as well as supporting higher-order F-bounded quantification. These last models are rich enough to support all constructs described in this paper. Other important early contributions to the semantics of object-oriented languages (especially with regard to the notion of "coherence" of interpretations) appeared in (Breazu-Tannen *et al.*, 1989; Breazu-Tannen *et al.*, 1991; Curien and Ghelli, 1992).

(America *et al.*, 1986; America *et al.*, 1989) present a language design and both operational and denotational semantics for a parallel untyped object-oriented language. (America, 1987; America and van der Linden, 1990) respectively discuss and present an extension to a typed language which includes both subtyping and inheritance. This work, like that described in this paper, separates the subclass and subtype hierarchies and also introduces keywords, "*self*" and "*MyType*," like those described here. No denotational semantics or soundness results are given there.

The language Emerald (Black *et al.*, 1986) is an object-oriented language designed for distributed systems which has types and subtypes, but no classes or inheritance. It includes type parameters and introduces a notion similar to F-bounded quantification to check the applicability of operations to types. (Black and Hutchinson, 1991) includes a careful discussion of its type system, including proofs that the dynamic type of an object always "conforms to" its static type, thereby ensuring the absence of "message not understood" error messages.

(Hense, 1991; Hense, 1990) present both a language design and semantics for an untyped object-oriented language. The semantics of inheritance in this language is based on (Cook, 1989a). Hense provides language constructs and semantics for explicit wrappers (Hense, 1991) and instance variables (Hense, 1990).

The paper (Palsberg and Schwartzback, 1991) presents a theory of statically typed object-oriented languages in which subclasses preserve subtypes. Their system requires that types be finite sets of classes. While this seems to work satisfactorily within an individual program, the authors note the problems which occur when considering the problem of separate compilation. In that circumstance, one must consider a class as having a potentially infinite number of subclasses (and thus the type of the class must be composed of all of those subclasses). They point out that their system can result in statically incorrect subclasses in this situation. There seems to be no work-around to this problem with their system.

(Palsberg and Schwartzback, 1991) and (Palsberg and Schwartzback, 1990) introduce the notion of substitution as a mechanism orthogonal to inheritance which, when combined with inheritance, provides a more general notion of subclassing. I have not studied this notion in detail, but suspect that the mechanism of class substitution can be viewed as an implicit form of bounded quantification in the original class and as a corresponding implicit type application in the actual class substitution. While likely definable in terms of bounded quantification, class substitution has the advantage of not requiring the author of a class to specify all locations in which one might later like to make a substitution. This mechanism may have advantages similar to the implicit polymorphism in languages like ML, Miranda†, and Haskell.

The research closest in spirit to this paper is that of (Cardelli, 1992b). In fact much of the research contained in this paper took place during a visit to DEC's Systems Research Center in which Cardelli and I had many substantive discussions on these topics. Cardelli's goal was slightly different than that of this paper. He wished to find a minimal language which was sufficient for expressing the key concepts of object-oriented programming language. Thus he provides a translation of a language similar to that given here into a simpler language with constructs for subtyping, bounded quantification, and powerful record operations. Nevertheless his work could be recast as an alternative solution to the one given in this paper (and vice versa). While there are technical differences in the solutions given here and in Cardelli's work, the end results are similar: a semantics (or translation) of a typed object-oriented language into a typed model (respectively, language) which provides an explanation of the very complex notions of object-oriented languages.

A recent series of papers by Pierce and his coworkers (Pierce and Turner, 1993; Pierce and Turner, 1992a; Pierce and Hoffman, 1992) represent an alternative approach to that of Cardelli and this paper. Their goal is to model object-oriented languages using second-order bounded lambda calculus, but using existential types to replace the use of F-bounded lambda calculus (as used in this paper and the work of Cook *et al.*) or higher-order fixed points (as used in the work of Cardelli and Mitchell). While their language is somewhat less expressive (they have difficulty modeling our `Point` with `eq` method), their approach allows them to avoid some of the complexity which arises through the proliferation of fixed points in determining the meaning of object-oriented constructs. Another more recent paper (Pierce and Turner, 1992b) provides a general technique for modeling binary operations. This can be applied either to the approach of (Pierce and Turner, 1993) or to that given here.

Recent work by Martin Abadi (1994) has resulted in a somewhat different style of semantics for the object-oriented features of Modula-3. As Modula-3 is based on a delegation style of inheritance rather than the class-based approach of the languages cited here, it is not clear whether this work can be extended to class-based inheritance. However this simpler (though more restrictive) style of inheritance via delegation allows for a much less complex semantic definition.

(Ghelli, 1991; Castagna *et al.*, 1991) present an alternative approach to obtain many of the advantages of object-oriented languages with a different set of language constructs. These constructs eliminate many of the fixed points which add to the complexity here. Their approach is to organize the program around overloaded functions. Thus rather than looking up a method inside a particular object, one may imagine going directly to an overloaded function name and then determining which version of the function is to be used based on the type of the first argument of the function. The inclusion of subtyping in the language requires one to be quite careful

† Miranda is a trademark of Research Software Limited.

about the collection of functions which share the same name. Restrictions on such collections ensure that no conflicts between definitions occur at run-time. These functions which branch on the types of more than one argument are sometimes terms multi-methods, and are used in some object-oriented extensions of LISP.

While this approach is quite different from that normally taken in object-oriented languages, it may allow users to avoid some of the complexity of the semantics of inheritance given here.

10 Future research and other work in TOOPL

We have not discussed an algorithm for type checking in this paper. The recent paper (Pierce, 1992) showed that the type-checking problem for $F_{\leq}$ is undecidable. Since the semantics of TOOPL is based on models of an extension of $F_{\leq}$, one might suspect that the type-checking problem for this language might be undecidable as well.

The undecidability of type checking $F_{\leq}$ results from the undecidability of determining if one type is a subtype of another. In contrast, (Dimock and Muller, 1992) have recently shown that the subtype problem is decidable for TOOPL. Kozen *et al.* (1993), have shown that subtyping for a similar language which also adds a rule to "unwind" recursively defined types has an $O(n^2)$ algorithm. While their paper does not discuss record types, an addition to handle this seems relatively straightforward. Unfortunately, we do not see a way of adapting their algorithm to the subtyping problem for TOOPL.

Building on the work of Dimock and Muller, Bruce, Crabtree, Murtagh, and van Gent have created a simple variant of TOOPL, called TOOPLE (for TOOPL with Extra information). The only differences in syntax between TOOPL and TOOPLE are that *class*, *extend*, and *update* terms must be labeled with their intended types. Then, applying techniques suggested in (Curien and Ghelli, 1992), it can be shown (under very natural restrictions on C) that every term which is typable with respect to C, E has a minimum type. From this an algorithm can be derived which solves the type-checking problem for TOOPLE. Thus this language has useful practical as well as theoretical properties. A description of the subtyping and type-checking algorithms can be found in (Bruce *et al.*, 1993).

A natural (operational) semantics for TOOPLE is given in (Bruce *et al.*, 1993). This semantics is somewhat easier to understand since it does not rely on the complex fixed points used here. That paper shows that the natural and denotational semantics are consistent and also provides a proof of the subject reduction theorem for TOOPLE, providing another proof of the soundness of the type-checking system.

The operational semantics of TOOPL also implicitly suggests a relatively simple extension of the type lambda calculus which may be useful in providing a theoretical foundation for investigations of object-oriented languages.

One possible complaint about TOOPL is that it is a functional language, and object-oriented languages are primarily imperative because of the instance variables. While we have modeled updatable instance variables here with a copy semantics, it would clearly be interesting to extend our language design principles to an imperative language. Recent work with Robert van Gent, (van Gent, 1993; Bruce and van Gent, 1993), has resulted in such a language, TOIL. As with TOOPL, we can prove that TOIL is type-safe. A prototype implementation exists, and work is progressing on adding polymorphism to the language.

Another important topic to examine with regard to the work presented in this paper is the development of axioms and rules for proving the correctness of programs in the language presented here. These axioms and rules should be shown to be sound with respect to the denotational semantics given here. There is hope that these axioms and rules may be substantially simpler than the denotational semantics. In particular, restrictions on the redefinition of methods in subclasses similar to those specified in Eiffel (Meyer, 1988) (*i.e.*, preconditions must be weaker and postconditions stronger in overridden methods in subclasses) may result in a language which is more restrictive, but whose semantics can be more easily comprehended. A (relative) completeness theorem for such a system would be desirable, though not essential for practical applications. I am currently examining such a system with my students.

Acknowledgements

This research was partially supported by NSF grants CCR - 9105316 and CCR - 9121778, and, at Stanford, the Powell Foundation.

Special thanks go to the Computer Science Department at Stanford University, DEC's System Research Center, and the Départment de Mathématiques et d'Informatique at the Ecole Normale Supérieure in Paris for their generous support and hospitality during my sabbatical leave in 1991, when much of this work was being done. The work described in this paper benefited greatly from numerous discussions with Luca Cardelli. The work with John Mitchell on constructing models for the higher-order bounded lambda calculus provided the semantical underpinnings of this work. Discussions with William Cook, other members of the former ABEL team at HP Labs, Peter Wegner, and Andrew Black helped clarify many of the complexities of object-oriented languages. The Harvard University Object-Oriented Languages Seminar from 1992-93, Benli Pierce, and Jon Riecke provided very useful comments on earlier drafts of this paper. Special thanks to the Williams TOOPL research group (Tom Murtagh, Jon Crabtree, Gerald Kanapathy, and Robert van Gent) for helping work out extensions and refinements to the language design.

My long-standing collaboration with Giuseppe Longo has provided many insights into the nature of subtyping and the more general aspects of the semantics of programming languages. His enthusiasm, insights, and willingness to explore alternative approaches have helped make research an enjoyable and rewarding activity.

References

M Abadi. Doing without F-bounded quantification. Message to Types electronic mail list, February, 1992.

Martin Abadi. Baby modula-3 and a theory of objects. *Journal of functional programming*, to appear, 1994.

M Abadi and G.D. Plotkin. A PER model of polymorphism and recursive types. In *Proc. Symposium on Logic in Computer Science*, pages 355–365. IEEE, 1990.

R.M. Amadio. Recursion over realizability structures. *Information and Computation*, 91(1):55–86, 1991.

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

Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editor, *ECOOP '87*, pages 234–242. Springer-Verlag, LNCS 276, 1987.

Pierre America, J. de Bakker, J. Kok, and J. Rutten. Operational semantics of a parallel object-oriented language. In *Proc 13th ACM Symp. Principles of Programming Languages*, pages 194–208, 1986.

Pierre America, J. de Bakker, J. Kok, and J. Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83(2):152–205, 1989.

Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *OOPSLA-ECOOP '90 Proceedings*, pages 161–168. ACM SIGPLAN Notices,25(10), October 1990.

G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA Begin.* Aurbach, 1973.

Andrew Black. Private communication, 1992.

A. Black and N. Hutchinson. Typechecking polymorphism in Emerald. Technical Report CRL 91/1 (Revised), DEC Cambridge Research Lab, 1991.

A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 78–86, October 1986.

V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Fourth IEEE Symp. Logic in Computer Science*, pages 112–129, 1989.

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

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.

K. Bruce, J. Crabtree, A. Dimock, R. Muller, T. Murtagh, and R. van Gent. Safe and decidable type checking in an object-oriented language. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 29–46, 1993.

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.

K. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. In *Third IEEE Symp. Logic in Computer Science*, pages 38–51, 1988.

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

K. B. Bruce, A. R. Meyer, and J. C. Mitchell. The semantics of second-order lambda calculus. *Information and Computation*, 85(1):76–134, 1990. Reprinted in *Logical Foundations of Functional Programming,* ed. G. Huet, Addison-Wesley (1990) 213–273.

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.

Kim B. Bruce and Robert van Gent. TOIL: A new type-safe object-oriented imperative language. to appear, 1993.

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.

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.

L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Special issue devoted to *Symp. on Semantics of Data Types,* Sophia-Antipolis (France), 1984.

L. Cardelli. Structural subtyping and the notion of powertype. In *Proc 15th ACM Symp. Principles of Programming Languages*, pages 70–79, 1988.

Luca Cardelli. Extensible records in a pure calculus of subtyping. Technical Report 81, DEC Systems Research Center, 1992.

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

L. Cardelli, J. Donahue, L. Galssman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report SRC-31, DEC systems Research Center, 1988.

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

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

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

F. Cardone. Relational semantics for recursive types and bounded quantification. In *ICALP*, pages 164–178. Springer-Verlag LNCS 372, 1989.

Giuseppe Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. Technical report, Ecole Normale Superieure, 1991. To appear.

G. Castagna and Benjamin Pierce. Decidable bounded quantification. In *Proc. ACM Symp. on the Principles of Programming Languages*, 1994.

W.R. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

W.R. Cook. A proposal for making Eiffel type-safe. In *European Conf. on Object-Oriented Programming*, pages 57–72, 1989.

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.

W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 433–444, 1989.

Brad Cox. *Object-oriented programming; an evolutionary appoach.* Addison-Wesley, 1986.

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.

S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, 1988.

Allyn Dimock and Robert Muller. Private communication, 1992.

G. Ghelli. A static type system for message passing. In *OOPSLA '91 Proceedings*, pages 129–145. ACM SIGPLAN Notices,26(11), November 1991.

A. Goldberg and D. Robson. *Smalltalk–80: The language and its implementation.* Addison Wesley, 1983.

Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.

Andreas V. Hense. Denotational semantics of an object oriented programming language with explicit wrappers. Technical Report A 11/90, Universitat des Saarlandes, 1990.

Andreas V. Hense. Wrapper semantics of an object oriented programming language with state. In *TACS '91 proceedings*, pages 548–568. Springer-Verlag, 1991.

S. Kamin. Inheritance in Smalltalk-80: a denotational definition. In *ACM Symp. Principles of Programming Languages*, pages 80–87, 1988.

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

Barbara Liskov. Data abstraction and hierarchy. In *OOPSLA '87 Addendum to the Proceedings*, pages 17–34. ACM SIGPLAN Notices,23(5), May 1988.

O. Madsen, B. Magnusson, and B. Moller-Pedersen. Strong typing of object-oriented languages revisited. In *OOPSLA-ECOOP '90 Proceedings*, pages 140–150. ACM SIGPLAN Notices,25(10), October 1990.

B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

B. Meyer. *Eiffel: the language.* Prentice-Hall, 1992.

J.C. Mitchell. Type inference and type containment. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France), Springer LNCS 173*, pages 257–278, June 1984.

J.C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):211–249, 1988. Reprinted in *Logical Foundations of Functional Programming,* ed. G. Huet, Addison-Wesley (1990) 153–194.

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.

J.C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 365–458. North-Holland, 1990.

J.C. Mitchell and G.D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages,* 1985.

D. Moon. Object-oriented programming with flavors. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 1–9, 1986.

Stephen M. Omohundro. The Sather language. Technical report, International Computer Science Institute, 1991.

J. Palsberg and M. Schwartzback. Type substitution for object-oriented programming. In *OOPSLA-ECOOP '90 Proceedings*, pages 151–160. ACM SIGPLAN Notices,25(10), October 1990.

J. Palsberg and M. Schwartzback. Static typing for object-oriented programming. Technical report, Aarhus University Computer Science Department, 1991.

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

Benjamin C. Pierce and Martin Hoffman. An abstract view of objects and subtyping (preliminary report). Technical Report ECS-LFCS-92-226, University of Edinburgh, 1992.

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

Benjamin C. Pierce and David N. Turner. Statically typed multi-methods via partially abstract types. Technical Report to appear, University of Edinburgh, 1992.

Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. In *Proc 20th ACM Symp. Principles of Programming Languages*, pages 299–312, 1993.

U.S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 289–297, July 1988.

J.C. Reynolds. Using category theory to design implicit conversions and generic operators. In N.D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 211–2580. Springer-Verlag Lecture Notes in Computer Science, Vol. 94, 1980.

C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *OOPSLA '86 Proceedings*, pages 9–16. ACM SIGPLAN Notices,21(11), November 1986.

A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. 1st ACM Symp. on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–46, October 1986.

B. Stroustrop. *The* C^{++} *Programming Language*. Addison-Wesley, 1986.

L. Tesler. Object Pascal report. Technical Report 1, Apple Computer, 1985.

Robert van Gent. *TOIL: An imperative type-safe object-oriented language.* Williams College Senior Honors Thesis, 1993.

Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.

N. Wirth. The programming language Oberon. *Software - Practice and Experience*, 18:671–690, 1988.