Methods as Assertions

John Lamping[1], Martín Abadi[2]

[1] Xerox PARC
3333 Coyote Hill Road, Palo Alto CA 94304, USA
lamping@parc.xerox.com
[2] Digital Equipment Corporation Systems Research Center
130 Lytton Avenue, Palo Alto, CA 94301, USA
ma@src.dec.com

Abstract. A method definition can be viewed as a logical assertion. Whenever we declare a method as the implementation of an operation, we assert that if the operation is invoked on objects of the appropriate types then the method body will satisfy the specification of the operation. This view of methods as assertions is simple but general. Among its applications are: methods defined on interfaces as well as on classes; an elementary type system for objects that handles multi-methods; and a mechanism for method dispatch based on the desired output type as well as on the types of arguments. Further, these applications are compatible with traditional execution models and implementation techniques. Logical reasoning about methods plays a role at compile time, then gets out of the way.

1 Introduction

An object is commonly characterized as a collection of data together with associated procedures, called methods. Each method implements an operation on the object; an operation may have other implementations for other objects. (Operations are called “messages” in Smalltalk [12] and “member functions” in C++ [17].) When an operation is invoked on an object, the corresponding method is executed. The method is found by using the name of the operation as an index to select among the object's methods. This “record view” of object oriented programming describes both the common implementation techniques of many languages, such as C++, and most efforts to explain objects in terms of formal systems (e.g., [5, 6, 7, 15]).

An alternative view takes methods as belonging to operations, rather than to objects. An operation comes with a collection of methods, each of which implements the operation for different classes of argument. Multiple dispatch is supported rather naturally: the classes of several arguments can be used as indices to select among the operation's methods. This “operation view” meshes with the implementation techniques of programming languages like CLOS [3] and Cecil [9], and with the account of object oriented programming in terms of the $\lambda\&$-calculus [8].

Each view takes either operations or classes as primitive, and the other as defined in terms of the primitive concept. Dispatch on the primitive concept yields an efficient execution model. The first view takes operations as primitive: they are just selectors with no further structure. Classes are then defined in terms of how they handle operations. The second view takes classes as primitive. (However, classes also have a subclass structure.) Operations are then defined in terms of how they handle different classes of arguments.

In this paper we investigate a higher-level view according to which neither operations nor classes are primitive, but rather are jointly described by method declarations. Method declarations are assertions about the connections between operations and objects:

> A method declaration is an assertion that if a particular operation is invoked in certain circumstances on objects having certain properties then a particular procedure will correctly implement the operation.

The assertional view provides a framework for extending object oriented programming. It allows the range of factors that can potentially affect the applicability of a method to include not only the classes of the objects given as arguments, but also which operations are supported by those classes, additional properties declared when the objects were created, dynamic properties of the objects, and others. While some of these extensions may require new implementation techniques, others do not. This paper explores some applications of the assertional view that are compatible with standard implementation techniques.

Section 2 explores how the assertional view leads to a clear distinction between a notion of class (a set of objects with common properties) and a notion of interface (a set of operations supported by a class or classes). The assertional view allows methods to be defined not only on classes but also on interfaces. Thus, the choice of method for an operation on an object can depend on whether the object supports other operations.

Section 3 shows how method applicability can be treated with simple logical reasoning. This reasoning is independent of the details of a particular model of computation, or of the language in which method bodies are written. In particular, the reasoning is compatible with both functional and imperative models, and with both dynamic and static typing. Further, all the reasoning can be done at compile time; therefore, traditional execution techniques still apply.

Section 4 describes one way to embed reasoning about method applicability in a static type system. We obtain a strongly typed language where typechecking ensures that there will be an applicable method for every operation invocation. The type system is quite elementary, yet suitable for supporting object oriented programming, including multi-methods.

Section 5 presents an evaluation model. One interesting feature of this model is that operations can be relations, with several possible outputs of different types for the same inputs. The evaluation model allows method selection based not only on the types of the inputs but also on the desired type of the output. While the assertional view does not require this feature, we show how it supports it.

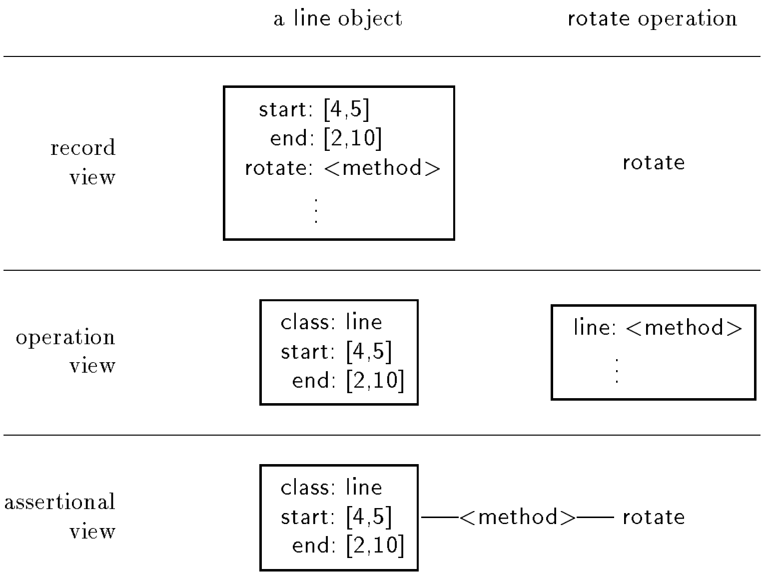

Fig. 1. Three views of a line object and the rotate operation

2 Methods as Assertions

Before giving a more formal explanation of the assertional view, we motivate it and compare it with the record view, with the operation view, and with several proposals to extend object oriented programming.

2.1 Three Views, by Example

We first discuss the record view, the operation view, and the assertional view in the context of an example, an imaginary graphics application. This application deals with two kinds of objects: graphic elements, like points and lines; and rendering devices, like displays and printers. The application should have operations on graphic elements, like translation and rotation; operations on rendering devices, like resetting and querying the image size; and the crucial operation of rendering a graphic element on a device, which involves both kinds of objects.

All three views deal roughly equivalently with the simple methods that are associated with a single class of objects. Figure 1 illustrates this, with the example of a method for rotating a line. In the record view, rotate is a primitive selector, and the method is included in line objects as the rotate component. In the operation view, the class line acts as a primitive selector, and the method is included in the rotate operation, specialized to line. In the assertional view, the method is not a component of an object or of an operation, but stands on its own, saying that the method body implements the rotate operation on any line.

The fact that the rotate operation is handled on line becomes a visible property of both line and the rotate operation.

A well-known limitation of the record view is the poor handling of multi-methods which, unlike the rotate method, are not associated with a single class. For example, since a method for rendering a line on a particular display is specialized simultaneously to line and to the display type, there is no natural way to present it in the record view: the method belongs neither to a line object nor to a display object, but rather to their conjunction. Both the operation view and the assertional view deal adequately with multi-methods.

Finally, the assertional view seems preferable to either of the other views in the treatment of methods that are most naturally defined on interfaces rather than on classes. Consider a method that implements rotation by two successive reflections. It is correct on any graphic element that supports the reflect operation, and it could be a useful default method to supply in a library, since a new kind of graphic element would automatically be able to handle rotations if it could handle reflections. Applicability of the method to an object depends not on any particular class or classes, but on what other operations the object supports. The method is naturally defined on an interface rather than on a class. This natural definition can be captured with the assertional view, because the interface supported by a class is visible in the assertional view and so can be used to determine method applicability. This does not fit well within either of the other views, because the interface supported by a class is a relation between the class and some operations; in the other views, such a relation is not visible to the machinery that determines method applicability.

While we have been discussing the expressiveness of the three views, we should also note that all three can support encapsulation boundaries. In the assertional view, encapsulation can be enforced by restricting the capability to make assertions about certain kinds of objects or operations. For example, the analogue of requiring all method definitions accessing protected operations of the line class to be lexically inside the class definition is to restrict assertions that reference those operations to a limited lexical scope, which will be, in effect, the class definition. A wide range of encapsulation schemes can be obtained through scoping rules of this sort.

2.2 Work-arounds

Of course, programmers have been successfully writing programs without the benefit of support for the assertional view. Looking at common work-arounds can give a better sense of what expressive power comes from the assertional view. We describe two treatments of the definition of rotation from reflection.

One work-around consists in defining an "abstract class." The abstract class documents the desired interface, so the rotate method can be defined on the abstract class. Every class that supports the interface is declared to inherit from the abstract class, and thus gets the rotate method. This work-around implies that, for each class that supports reflect, a programmer must explicitly declare that it supports the interface and should thus inherit the rotate method. This can be a problem for modularity—the ability to maximize independence between different parts of a program—since a class writer must be aware of all relevant interfaces already defined and a method writer must be aware of all relevant classes already defined. In fact, knowing the relevant classes probably does not even help the method writer, since most languages disallow adding parents to an already declared class. If there is code that creates objects of that class, there is no way to get the objects to support the new method.

A different work-around consists in defining `rotate` as a procedure, rather than as an operation. Type information can encode the interface for the objects that the procedure accepts as arguments. This work-around sacrifices the advantages of object oriented programming: it becomes impossible to write new `rotate` methods to handle additional cases, or to provide special handling of special cases.

Both the record view and the operation view present a dichotomy between operations, which support dispatch but can dispatch only on the basis of primitive properties, and external procedures, which do not allow dispatch but whose types can specify an interface for their arguments. The assertional view can be seen as a unification that eliminates the dichotomy.

2.3 The Assertional View in Context

In procedural programming, there is a rigid link between operation invocation and operation execution, because there is exactly one procedure to carry out each operation. Traditional object oriented programming makes that link more flexible by allowing for several different methods that can implement an operation, each applicable to different classes of arguments. Several proposed extensions to object oriented programming allow additional features to affect method applicability. For example, predicate classes [10] allow method applicability to depend on dynamic properties of objects; in the DROL language [18], method applicability may depend on how much time remains to complete a task. Bobrow et al. [4] sketch a number of such extensions, including the idea of letting method applicability depend on interfaces.

The assertional view does not limit a priori what can affect method applicability, and hence it can encompass all these kinds of extensions. However, we focus on a restricted, well-behaved use of the assertional view: exploiting the visibility of method declarations to let method applicability depend on interfaces. This use of the assertional view does not require any change to run-time facilities because the necessary reasoning about interfaces can be carried out at compile time. In fact, a pre-processor could examine required interfaces, determine what methods are applicable for various classes, and add explicit method declarations to those classes.

In contrast, various logic programming approaches to objects [2, 13, 16] have focused on using logical inference at the basis of program execution. In contrast, also, Agrawal et al. [1] discuss reasoning about method applicability but do not use the reasoning for deciding applicability.

The idea of treating method declarations as assertions can be seen as an instance, in the domain of objects, of the open-semantics ideas proposed by Dixon [11], who treats procedure declarations as assertions. This approach recognizes that the programmer has an intended meaning for each operation. Thus, a method declaration is an assertion about an operation, rather than a definition of the operation.

3 Method Applicability

Next we explore how one expresses that there is a method to implement an operation on certain objects. Such a statement is a free-standing assertion in our approach. Therefore, it is possible to reason about the availability and applicability of a method independently of any particular language for writing the method body, and of any execution model for running it. Thus, our explanation is relevant to both dynamically and statically typed languages. Since method applicability depends only on classes, not on particular objects, the necessary reasoning about applicability can take place at compile time, whether the language is dynamically or statically typed.

3.1 Basic Notations and Rules

In what follows, we take a class to be just a set of objects, typically a set of objects with something in common. We allow these objects to include implementation information; two objects with identical public behavior but different implementation could belong to different classes. We use the terms class and type interchangeably, and use the subtype relation, $x \leq y$, to mean that every object in the set x is in the set y.

We use a single name space for operations. In an actual language, the same name might denote different operations in different scopes, and assertions should apply to the operations, not to the name. For simplicity, we omit this level of indirection. We assume that to each name corresponds a single operation, with a single specification.

We proceed fairly formally, but formal sophistication should not be a prerequisite to understanding this section. We embody the assertional view in logic, using logical constants and logical variables for classes. We introduce a relation OK to express that some primitive or method implements an operation for certain arguments. More precisely, we write:

$$\mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \rightarrow y)$$

to mean that there is a primitive or method to handle the operation F on arguments of types $x_1, \ldots, x_n$, returning an answer of type y or diverging. For example, $\mathrm{OK}(\mathsf{plus}\colon \mathsf{int} \times \mathsf{int} \rightarrow \mathsf{int})$ indicates that there is a plus primitive or method that takes two integers and either returns an integer or diverges.

The inclusion of the output type y in the form $\mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \rightarrow y)$ is intended to facilitate static typechecking. For a dynamically typed language, it would be reasonable to omit y and to consider only the input types. Everything in this section still applies to that case, since y could always be taken to be the universal type.

Our description of $\mathrm{OK}(\mathsf{F}{:}\, x_1\times\cdots\times x_n \to y)$ is deliberately open about whether F should be a function. In fact, we may allow F to be a relation, with several possible outputs of different types. In this case, $\mathrm{OK}(\mathsf{F}{:}\, x_1\times\cdots\times x_n \to y)$ means that there is some way of handling F that returns an answer of type y or diverges. Allowing a relation is particularly appealing in the context of a statically typed language; we explore this option below.

While now we have notation for talking about whether an operation is handled, we still lack notation for saying that a particular method or primitive is the one to use for an operation. We write:

$$\mathrm{prim}(\mathsf{F}{:}\, x_1\times\cdots\times x_n \to y;\ f)$$

to mean that the primitive function f implements the operation F on arguments of types $x_1,\ldots, x_n$, with a result of type y. Similarly, we write:

$$\mathrm{method}(\mathsf{F}{:}\, x_1\times\cdots\times x_n \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$$

to mean that the code T can implement the operation F on arguments $\mathsf{v}_1,\ldots,\mathsf{v}_n$ of types $x_1,\ldots, x_n$, with a result of type y.

We write $A \longrightarrow B$ to mean that A implies B; as we explain below, we use intuitionistic implication. The following axioms say that primitive functions and defined methods handle operations:

$$\mathrm{prim}(\mathsf{F}{:}\, x_1\times\cdots\times x_n \to y;\ f) \longrightarrow \mathrm{OK}(\mathsf{F}{:}\, x_1\times\cdots\times x_n \to y)$$

$$\mathrm{method}(\mathsf{F}{:}\, x_1\times\cdots\times x_n \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T}) \longrightarrow \mathrm{OK}(\mathsf{F}{:}\, x_1\times\cdots\times x_n \to y)$$

In addition, we adopt the evident axioms for subtyping:

$$\mathrm{prim}(\mathsf{F}{:}\, x_1\times\cdots\times x_n \to y;\ f) \wedge \textstyle\bigwedge_{1\le i\le n}(x'_i \le x_i) \wedge (y \le y') \\ \longrightarrow \mathrm{prim}(\mathsf{F}{:}\, x'_1\times\cdots\times x'_n \to y';\ f)$$

$$\mathrm{method}(\mathsf{F}{:}\, x_1\times\cdots\times x_n \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T}) \wedge \textstyle\bigwedge_{1\le i\le n}(x'_i \le x_i) \wedge (y \le y') \\ \longrightarrow \mathrm{method}(\mathsf{F}{:}\, x'_1\times\cdots\times x'_n \to y';\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$$

$$\mathrm{OK}(\mathsf{F}{:}\, x_1\times\cdots\times x_n \to y) \wedge \textstyle\bigwedge_{1\le i\le n}(x'_i \le x_i) \wedge (y \le y') \\ \longrightarrow \mathrm{OK}(\mathsf{F}{:}\, x'_1\times\cdots\times x'_n \to y')$$

3.2 Interfaces

It is common for a method that implements an operation F to call other operations. In this case, F is handled only if the other operations are handled. We express this dependence as part of a condition for F to be handled, using an implication. For example, to express that there is a method for rotate if there is a method for $\mathsf{reflect}$, we write:

$$\forall x.\ \mathrm{OK}(\mathsf{reflect}{:}\, x \to x) \longrightarrow \mathrm{OK}(\mathsf{rotate}{:}\, x \to x)$$

(omitting some arguments for simplicity). This formula illustrates how to express that there is a method for an interface, rather than for a class. It states that any class that supports reflect also supports rotate. (Recall that we assume that a name scoping mechanism has ensured that reflect and rotate refer only to the intended graphic operations.) The condition $\mathrm{OK}(\mathsf{reflect}\colon x \to x)$ serves as a description of the required interface. The universal quantification then makes the declaration apply to any class that supports the interface. The characterization of a type by its properties is one of the main uses of bounded quantification in other systems.

Recall that we distinguish between types (sets of objects) and interfaces (collections of operations that one or more types support). In this example, the interface also corresponds to a type—there is a set of all objects for which reflect is handled. In general, however, interfaces do not coincide with types. Rather, they are predicates on types (possibly involving several types at once). Suppose, for example, that the max operation is defined on any class that supports a greater comparison operation:

$$\forall x.\ \mathrm{OK}(\mathsf{greater}\colon x \times x \to \mathsf{bool}) \longrightarrow \mathrm{OK}(\mathsf{max}\colon x \times x \to x)$$

It may be possible to compare elements of a type apple and also to compare elements of another type orange, but not to compare an element of apple with an element of orange. So while each of apple and orange supports greater, and hence max, their union does not. No single set corresponds to the interface $\mathrm{OK}(\mathsf{greater}\colon x \times x \to \mathsf{bool})$.

3.3 Recursion

Recursive methods call for an additional refinement. Suppose, for example, that there are two kinds of graphic elements, atomic elements and composite elements; the components of a composite element are themselves graphic elements. It is natural to write a method for reflect on composite elements that calls the reflect operation on each component element. An obvious description for this situation is the formula:

$$\begin{array}{l}(\mathrm{OK}(\mathsf{reflect}\colon \mathsf{composite} \to \mathsf{composite}) \wedge \mathrm{OK}(\mathsf{reflect}\colon \mathsf{atomic} \to \mathsf{atomic}))\\ \qquad \longrightarrow \mathrm{OK}(\mathsf{reflect}\colon \mathsf{composite} \to \mathsf{composite})\end{array}$$

That is, the reflect operation is handled on composite objects, provided that the component objects can, recursively, be reflected. Unfortunately, it does not follow from this formula that the reflect operation is supported for composite elements if it is supported for atomic elements. In fact, the assertion is a tautology.

If a method calls the operation that it itself is implementing, the resulting recursion may never bottom out. In one sense, the operation is not truly supported. However, in accord with most type systems for programming languages, we address only the problem of ill-formed operation invocations (for partial correctness), not the problem of infinite loops (for total correctness). In other words, we address the question of whether a program will get "no method" errors, not whether it will diverge. Therefore, we would like to find a form of assertion that would allow recursive definitions, yielding for example that the reflect operation is supported on composite elements if it is supported on atomic elements.

In general, when we try to satisfy a method's prerequisites, we assume that the operation that the method defines is available. We introduce a new connective, $A \xrightarrow{+} B$, to capture this assumption. Informally, $A \xrightarrow{+} B$ should mean "if assuming B lets us prove A, then we can conclude B."

We define $A \xrightarrow{+} B$ as an abbreviation for $(B \longrightarrow A) \longrightarrow B$ in intuitionistic logic. We use intuitionistic logic for expressing the informal definition because the informal definition is talking about proof, rather than about truth. At any rate, classical logic would not work: $(B \longrightarrow A) \longrightarrow B$ is simply equivalent to B in classical logic. Moreover, it follows from the definition that $A \xrightarrow{+} B$ has the property:

$$(B \longrightarrow A) \longrightarrow ((A \xrightarrow{+} B) \longrightarrow B)$$

which agrees with the intended meaning of $A \xrightarrow{+} B$.

Now we may revise the assertion about reflect:

$$\begin{array}{l}(\mathrm{OK}(\mathsf{reflect}\colon \mathsf{composite} \rightarrow \mathsf{composite}) \wedge \mathrm{OK}(\mathsf{reflect}\colon \mathsf{atomic} \rightarrow \mathsf{atomic})) \\ \quad \xrightarrow{+} \mathrm{OK}(\mathsf{reflect}\colon \mathsf{composite} \rightarrow \mathsf{composite})\end{array}$$

This new assertion implies

$$\mathrm{OK}(\mathsf{reflect}\colon \mathsf{atomic} \rightarrow \mathsf{atomic})) \longrightarrow \mathrm{OK}(\mathsf{reflect}\colon \mathsf{composite} \rightarrow \mathsf{composite})$$

as desired. More interestingly, the same form of assertion allows mutual recursion. Imagine two operations, foo and bar, that return integers. Imagine further that we have mutually recursive methods for them. If we described this situation with the formulas:

$$\begin{array}{c}\forall x.\ \mathrm{OK}(\mathsf{foo}\colon x \rightarrow \mathsf{int}) \longrightarrow \mathrm{OK}(\mathsf{bar}\colon x \rightarrow \mathsf{int}) \\ \text{and} \\ \forall x.\ \mathrm{OK}(\mathsf{bar}\colon x \rightarrow \mathsf{int}) \longrightarrow \mathrm{OK}(\mathsf{foo}\colon x \rightarrow \mathsf{int})\end{array}$$

we would fail to get the desired conclusion that foo and bar are handled. On the other hand, $A \xrightarrow{+} B$ and $B \xrightarrow{+} A$ imply A and B, and hence the modified formulas about foo and bar:

$$\begin{array}{c}\forall x.\ \mathrm{OK}(\mathsf{foo}\colon x \rightarrow \mathsf{int}) \xrightarrow{+} \mathrm{OK}(\mathsf{bar}\colon x \rightarrow \mathsf{int}) \\ \text{and} \\ \forall x.\ \mathrm{OK}(\mathsf{bar}\colon x \rightarrow \mathsf{int}) \xrightarrow{+} \mathrm{OK}(\mathsf{foo}\colon x \rightarrow \mathsf{int})\end{array}$$

yield:

$$\forall x.\ \mathrm{OK}(\mathsf{foo}\colon x \rightarrow \mathsf{int}) \wedge \mathrm{OK}(\mathsf{bar}\colon x \rightarrow \mathsf{int})$$

In short, the connective $\xrightarrow{+}$ captures the import of recursive methods and allows reasoning about what operations are supported in terms of what methods are available, even in the presence of recursion.

3.4 Programming

In a dynamically typed system the programmer's declarations are interpreted as assertions that describe the subclass structure for the class constants and that define methods, without the benefit or the curse of static typechecking. The logical form for these assertions could be, for example:

$$\mathsf{a} \leq \mathsf{b}$$
$$\text{and}$$
$$\forall x, y.\ (\mathrm{OK}(\mathsf{F}{:}\, x \times \mathsf{a} \to y) \wedge \mathrm{OK}(\mathsf{G}{:}\, \mathsf{b} \to x)) \xrightarrow{+} \mathrm{method}(\mathsf{H}{:}\, x \to y;\ \mathsf{v}.\mathsf{T})$$

Facts about method applicability are derived from these assertions. For example, if we have $\mathrm{OK}(\mathsf{F}{:}\, \mathsf{b}{\times}\mathsf{a} \to \mathsf{a})$ and $\mathrm{OK}(\mathsf{G}{:}\, \mathsf{b} \to \mathsf{b}))$ we obtain $\mathrm{method}(\mathsf{H}{:}\, \mathsf{b} \to \mathsf{a};\ \mathsf{v}.\mathsf{T})$, and then $\mathrm{OK}(\mathsf{H}{:}\, \mathsf{b} \to \mathsf{a})$.

Thus, the programmer always provides methods, possibly with interface assumptions, but never asserts that an operation is handled without at the same time producing a method to handle the operation. Whenever the system infers that an operation is handled, one of the programmer's methods is applicable.

Furthermore, the assertions are available at compile time. Thus, all propositional reasoning can be carried out then. In this case, all that is left for run time is the usual dispatch on run-time types to the methods that were inferred to be applicable at compile time. Of course, as the system is dynamically typed, if a method body calls an operation that the programmer did not specify as a precondition of the method, the operation may have no applicable implementation.

One remaining question is what it means when one can prove that several different methods are applicable as implementations of the same operation for the same arguments. We consider that to be an issue of language design outside the scope of this paper. A language analogous to C++, for example, would probably choose the applicable method with the most stringent pre-conditions, and give an error if there was no unique such method.

4 A Language

For the rest of this paper we consider a simple, exemplar functional language. However, all of the previous discussion applies to other execution models, and mutatis mutandi so do the following results.

4.1 Programs

As method bodies we use the terms generated by the following grammar:

$$\begin{aligned} \mathrm{T} ::= {} & \mathsf{v} \\ & \mid \mathsf{let}\ \mathsf{v} = \mathrm{T}\ \mathsf{in}\ \mathrm{T} \\ & \mid \mathsf{if}\ \mathrm{T}\ \mathsf{then}\ \mathrm{T}\ \mathsf{else}\ \mathrm{T} \\ & \mid \mathsf{F}(\mathrm{T},\ldots,\mathrm{T}) \\ & \mid \mathsf{new}(type) \end{aligned}$$

This language is a first-order term language with two object oriented constructs: operation invocation ($\mathsf{F}(\mathrm{T},\ldots,\mathrm{T})$, where F is an operation name) and object creation ($\mathsf{new}(\mathit{type})$). The new form takes only constant types; allowing type variables would raise difficult issues, but fortunately this is not necessary for most of object oriented programming as it currently occurs. The language also includes forms for variables, local bindings, and conditionals.

4.2 Examples

In order to illustrate the use of the language, we give three small examples. We assume dynamic typing as in Subsection 3.4.

A programmer can declare a square method on integers by asserting:

$$\mathrm{method}(\mathsf{square}\colon \mathsf{int} \rightarrow \mathsf{int};\ \mathsf{n.times(n,\ n)})$$

This assertion implies:

$$\mathrm{OK}(\mathsf{square}\colon \mathsf{int} \rightarrow \mathsf{int})$$

A method for square on any type supporting the times operation requires a more interesting assertion:

$$\forall x.\ \mathrm{OK}(\mathsf{times}\colon x \times x \rightarrow x) \longrightarrow \mathrm{method}(\mathsf{square}\colon x \rightarrow x;\ \mathsf{n.times(n,\ n)})$$

The conditions for a method to be correct may include more than just what is required for its body to typecheck. For example, the declaration

$$\forall x.\ (x \leq \mathsf{number}) \longrightarrow \mathrm{method}(\mathsf{square}\colon x \rightarrow x;\ \mathsf{n.times(n,\ n)})$$

means that this method for square works only on numbers.

Similarly, we may consider a partial definition for points:

$$\begin{array}{l}
\mathrm{prim}(\mathsf{get\text{-}x}\colon \mathsf{point\text{-}impl} \rightarrow \mathsf{int};\ \mathit{prim\text{-}get\text{-}x}) \\
\mathrm{prim}(\mathsf{set\text{-}x}\colon \mathsf{point\text{-}impl} \times \mathsf{int} \rightarrow \mathsf{point\text{-}impl};\ \mathit{prim\text{-}set\text{-}x}) \\
\mathrm{prim}(\mathsf{get\text{-}y}\colon \mathsf{point\text{-}impl} \rightarrow \mathsf{int};\ \mathit{prim\text{-}get\text{-}y}) \\
\mathrm{prim}(\mathsf{set\text{-}y}\colon \mathsf{point\text{-}impl} \times \mathsf{int} \rightarrow \mathsf{point\text{-}impl};\ \mathit{prim\text{-}set\text{-}y})
\end{array}$$

Here $\mathsf{point\text{-}impl}$ is a class, possibly one of several different implementations of points. These four lines are declarations saying that elements of $\mathsf{point\text{-}impl}$ have primitive accessors *prim-get-x*, *prim-get-y*, *prim-set-x*, and *prim-set-y*. Because of our functional model, *prim-set-x* and *prim-set-y* should return new objects rather than side-effecting their inputs, hence their types.

The next declaration gives a method for the add operation, which takes two points and returns a new point whose coordinates are the sums of the coordinates of the arguments:

$$\begin{array}{l}
\forall p.\ \mathrm{OK}(\mathsf{get\text{-}x}\colon p \rightarrow \mathsf{int}) \wedge \mathrm{OK}(\mathsf{set\text{-}x}\colon p \rightarrow p) \\
\quad \wedge\ \mathrm{OK}(\mathsf{get\text{-}y}\colon p \rightarrow \mathsf{int}) \wedge \mathrm{OK}(\mathsf{set\text{-}y}\colon p \rightarrow p) \\
\qquad \longrightarrow \mathrm{method}(\mathsf{add}\colon p \times p \rightarrow \mathsf{point\text{-}impl};\ \mathsf{f,\ s.\ T})
\end{array}$$

where the method body T is a straightforward sequence of operations to extract the coordinates from the inputs, add them, and place the answers in the output:

```
set-x(set-y(new(point-impl),
            plus(get-y(f), get-y(s))),
      plus(get-x(f), get-x(s)))
```

The typing is written to be neutral with respect to the choice of implementation of points. It is not written specifically on point-impl, but applies to any class with the proper interface. However, it yields a point-impl, because the method calls new(point-impl) in the body. The output type, thus, will in general differ from the input type. This is a manifestation of what Kiczales has called the "make isn't generic problem" [14].

While the result of add is not of type p, it does support the get-x, get-y, set-x, set-y, and add operations, as a result of being a point-impl, so it does satisfy the interface expected of points. In some situations, a point-impl might not be acceptable as a result of an add operation. But that does not imply that the method assertion is incorrect. Below we show how method selection can depend not only on input types but also on the desired output type, so that an alternate method could be selected in those cases where a point-impl would not be an acceptable output type.

The add method, just as the square method, applies not to any particular class but to any class that obeys the appropriate protocol. Thus, behavior can be added to an object or class after its creation. The programmer's assertions apply both to classes already defined and to classes yet to be defined.

For an example of a method defined on an interface involving several classes, we return to the imaginary graphics application. We consider a variant of the operation of rendering a graphic element, which renders the element inverted. One way to implement this operation is to render the element and then to invert the image. In our functional language, the method body could be:

```
invert(render(graphic, display))
```

and the full method declaration:

$$\begin{aligned}&\forall g, d, i.\ \mathrm{OK}(\mathsf{render}\colon g \times d \to i) \wedge \mathrm{OK}(\mathsf{invert}\colon i \to i)\\ &\qquad \longrightarrow \mathrm{method}(\mathsf{render\text{-}inverted}\colon g \times d \to i;\ \ \mathsf{graphic,\ display.\ T})\end{aligned}$$

where T is the body above. The method declaration applies to an interface that pertains to triples of types, as suggested by the quantification. This is necessary because the method depends on the multi-method render. Thus, there is no way to express the interface as a predicate on a single type but only on a combination of three types. This straightforward handling of interfaces over more than one type is a natural feature of the logical approach to method applicability.

4.3 Typing

In a statically typed system, reasoning about method applicability can be used to typecheck the bodies of methods, that is, to check that a method will never call an operation that is not handled. In order to check that methods call only supported operations, we introduce the notation:

$$\mathsf{T}\colon x$$

which is intended to mean that all operations called in T are supported and that a result of type x can be returned, unless the computation diverges. Here x is a type, not an interface. The context of type inference will typically provide the information about the interfaces that x supports.

While in a dynamically typed language the programmer would make method assertions, a typecheck is interposed in a statically typed language. The programmer asserts only that certain code is candidate to being a method; it gets promoted to a method if it typechecks. We introduce another notation to express this assertion:

$$\mathrm{cand}(\mathsf{F}\colon x_1\times\cdots\times x_n \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$$

It means that the body T implements F for the specified types, provided it typechecks. The operations invoked in T do not need to be explicitly listed as hypothesis for this assertion, since they can be inferred. Thus, the programmer may simply assert $\mathrm{cand}(\mathsf{F}\colon x_1\times\cdots\times x_n \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$.

In order to obtain a tidy, tractable type system, we abandon the free use of intuitionistic logic. In its place, we use a type system that encodes enough of intuitionistic logic to express the kinds of examples presented earlier. We rely on a simple set of rules for proving judgments of the form $\Gamma \vdash A$. This judgment means that Γ implies A. Here Γ is a list of formulas, each of one of the forms $\mathsf{T}\colon x$, $x \leq y$, $\mathrm{prim}(\mathsf{F}\colon x_1\times\cdots\times x_n \to y;\ f)$, $\mathrm{cand}(\mathsf{F}\colon x_1\times\cdots\times x_n \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$, or $\mathrm{OK}(\mathsf{F}\colon x_1\times\cdots\times x_n \to y)$; and A is a single formula, of one of those forms or of the form $\mathrm{method}(\mathsf{F}\colon x_1\times\cdots\times x_n \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$.

We assume that each set of assumptions (on the left of $\vdash$) is well formed, in the sense that no term variable is given a type more than once in it. Correspondingly, we interpret terms up to renaming of bound variables.

We allow no propositional connectives in our formulas. Judgments of the form $\Gamma \vdash A$ suffice to capture the necessary conjunctions and implications. For simplicity, we do not allow quantifiers either. As the examples presented earlier illustrate, much can be achieved with quantifiers only at the outermost level of formulas; we assume, from now on, that all these quantifiers have been instantiated "by hand."

The type rules are:

$$\Gamma, A, \Gamma' \vdash A$$

$$\frac{\Gamma \vdash \mathsf{S}\colon x \qquad \Gamma, \mathsf{v}\colon x \vdash \mathsf{T}\colon y}{\Gamma \vdash \mathsf{let\ v = S\ in\ T}\colon y}$$

$$\frac{\Gamma \vdash \mathsf{S}\colon \mathsf{bool} \qquad \Gamma \vdash \mathsf{T}_1\colon x \qquad \Gamma \vdash \mathsf{T}_2\colon x}{\Gamma \vdash \mathsf{if}\ \mathsf{S}\ \mathsf{then}\ \mathsf{T}_1\ \mathsf{else}\ \mathsf{T}_2\colon x}$$

$$\frac{\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y) \qquad \Gamma \vdash \mathsf{T}_1\colon x_1 \quad \ldots \quad \Gamma \vdash \mathsf{T}_n\colon x_n}{\Gamma \vdash \mathsf{F}(\mathsf{T}_1,\ldots,\mathsf{T}_n)\colon y}$$

$$\Gamma \vdash \mathsf{new}(x)\colon x$$

$$\frac{\Gamma \vdash \mathrm{prim}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y;\ f)}{\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y)}$$

$$\frac{\Gamma \vdash \mathrm{method}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})}{\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{cand}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T}) \\ \Gamma, \mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y), \mathsf{v}_1\colon x_1, \ldots, \mathsf{v}_n\colon x_n \vdash \mathsf{T}\colon y\end{array}}{\Gamma \vdash \mathrm{method}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})}$$

The first rule simply says that A implies A; as a special case, we have a type rule for variables: $\mathsf{v}\colon x$ implies $\mathsf{v}\colon x$. The next two rules are the obvious ones for local bindings and conditionals. The fourth rule is the rule for operation invocation; it says that if the operation is handled for the argument types then the invocation is valid and its result has the result type of the operation for those types of arguments. The first hypothesis of this rule is itself checked by reasoning about method applicability. The fifth rule deals with the creation form; it simply says that $\mathsf{new}(x)$ has type x. The sixth and seventh rules correspond to axioms of Subsection 3.1. The final rule formalizes the semantics of $\mathrm{cand}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$. This rule allows methods that call themselves recursively, as it lets us assume that the operation F being defined is handled when we typecheck the method body T.

Combining the last two rules gives the derived rule:

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{cand}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T}) \\ \Gamma, \mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y), \mathsf{v}_1\colon x_1, \ldots, \mathsf{v}_n\colon x_n \vdash \mathsf{T}\colon y\end{array}}{\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y)}$$

which says that if there is a candidate for an operation, then the operation is handled provided the candidate typechecks under the assumption that the operation is handled. Thus, $\mathrm{cand}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$ implies "$\mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T}$ typechecks $\xrightarrow{+}$ $\mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y)$."

In addition, we have the expected rules for subtyping:

$$\Gamma \vdash x \leq x$$

$$\frac{\Gamma \vdash x \leq y \qquad \Gamma \vdash y \leq z}{\Gamma \vdash x \leq z}$$

$$\frac{\Gamma \vdash \mathsf{T}{:}\, x \qquad \Gamma \vdash x \leq y}{\Gamma \vdash \mathsf{T}{:}\, y}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathrm{prim}(\mathsf{F}{:}\, x_1 \times \cdots \times x_n \rightarrow y;\ f) \\ \Gamma \vdash x_i' \leq x_i \ \text{ for all } i \in \{1..n\} \qquad \Gamma \vdash y \leq y'\end{array}}{\Gamma \vdash \mathrm{prim}(\mathsf{F}{:}\, x_1' \times \cdots \times x_n' \rightarrow y';\ f)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathrm{cand}(\mathsf{F}{:}\, x_1 \times \cdots \times x_n \rightarrow y;\, \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T}) \\ \Gamma \vdash x_i' \leq x_i \ \text{ for all } i \in \{1..n\} \qquad \Gamma \vdash y \leq y'\end{array}}{\Gamma \vdash \mathrm{cand}(\mathsf{F}{:}\, x_1' \times \cdots \times x_n' \rightarrow y';\, \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathrm{method}(\mathsf{F}{:}\, x_1 \times \cdots \times x_n \rightarrow y;\, \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T}) \\ \Gamma \vdash x_i' \leq x_i \ \text{ for all } i \in \{1..n\} \qquad \Gamma \vdash y \leq y'\end{array}}{\Gamma \vdash \mathrm{method}(\mathsf{F}{:}\, x_1' \times \cdots \times x_n' \rightarrow y';\, \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathrm{OK}(\mathsf{F}{:}\, x_1 \times \cdots \times x_n \rightarrow y) \\ \Gamma \vdash x_i' \leq x_i \ \text{ for all } i \in \{1..n\} \qquad \Gamma \vdash y \leq y'\end{array}}{\Gamma \vdash \mathrm{OK}(\mathsf{F}{:}\, x_1' \times \cdots \times x_n' \rightarrow y')}$$

It is an obvious property of this system that an expression does not necessarily have a unique type, not even a principal type. For example, if Γ includes $\mathrm{OK}(\mathsf{F}{:} \rightarrow y)$ and $\mathrm{OK}(\mathsf{F}{:} \rightarrow z)$, then $\Gamma \vdash \mathsf{F}{:}\, y$ and $\Gamma \vdash \mathsf{F}{:}\, z$.

4.4 Typed Examples

With the type system, the programmer does not have to list explicitly the operations called by a method as part of the protocol that the method requires. The type system will infer these operations. Recall, for example, the operation render-inverted from Subsection 4.2. A method to implement this operation could be declared simply by writing:

$$\mathrm{cand}(\mathsf{render\text{-}inverted}{:}\, \mathsf{top} \times \mathsf{top} \rightarrow \mathsf{bottom};\ \ \mathsf{graphic,\ display.\ T})$$

where top is the most general type, bottom is the most specific type, and T is the method body invert(render(graphic, display)). In this example, typechecking suffices to guarantee the correctness of the method, provided that if invert and render are implemented then they have the expected meanings. Correspondingly, the declaration says that the method applies to any arguments for which the body typechecks. The body typechecks for any types g, d, and i on which invert and render are implemented, that is:

$$\begin{array}{l} \quad \mathrm{OK}(\mathsf{render}{:}\, g \times d \rightarrow i),\ \mathrm{OK}(\mathsf{invert}{:}\, i \rightarrow i), \\ \quad \mathrm{cand}(\mathsf{render\text{-}inverted}{:}\, \mathsf{bottom} \times \mathsf{bottom} \rightarrow \mathsf{top};\ \ \mathsf{graphic,\ display.\ T}) \\ \vdash \mathrm{method}(\mathsf{render\text{-}inverted}{:}\, g \times d \rightarrow i;\ \ \mathsf{graphic,\ display.\ T}) \end{array}$$

Thus, the effect of this method declaration is essentially the same as that of the more explicit declaration in the untyped language.

Of course, a programmer may want to express the intent that the arguments to render-inverted be graphic elements and rendering devices. This intent could be expressed explicitly in a declaration such as:

$$\mathrm{cand}(\textsf{render-inverted}: \textsf{graphic-type} \times \textsf{display-type} \rightarrow \textsf{top};\ \ \textsf{graphic, display. T})$$

Such a specific declaration is essential in the common case where the correctness of a method depends on the types of its arguments.

The programmer may also want to express the expectation that, under specified assumptions, a method body should typecheck or an operation should be handled. Our type system provides the basis for checking whether such expectations are met, but we will not introduce any syntax for expressing them.

4.5 Disjunctions

Disjunctive types can provide useful type information when a conditional statement can return objects of two different classes, and when these classes satisfy a common interface but have no common superclass satisfying the interface. This subsection discusses the motivation for disjunctive types and gives an example. It is a digression, in that we do not adopt disjunctive types in the rest of the paper; we merely want to point out the related issues.

Suppose that there are two implementation types for points, point-impl and other-point-impl, and that an operation distance is defined on all pairwise combinations of point-impl and other-point-impl:

$$\begin{array}{l}
\mathrm{OK}(\textsf{distance}: \textsf{point-impl} \times \textsf{point-impl} \rightarrow \textsf{real}) \\
\mathrm{OK}(\textsf{distance}: \textsf{point-impl} \times \textsf{other-point-impl} \rightarrow \textsf{real}) \\
\mathrm{OK}(\textsf{distance}: \textsf{other-point-impl} \times \textsf{point-impl} \rightarrow \textsf{real}) \\
\mathrm{OK}(\textsf{distance}: \textsf{other-point-impl} \times \textsf{other-point-impl} \rightarrow \textsf{real})
\end{array}$$

Consider the expression:

```
distance(if test1 then new(point-impl) else new(other-point-impl),
         if test2 then new(point-impl) else new(other-point-impl))
```

To infer that the distance operation in the expression is handled, it does not suffice to know that distance is handled for arguments of type point-impl and that it is handled for arguments of type other-point-impl. It is also necessary to know that distance is handled for one argument of type point-impl and one of type other-point-impl.

With disjunctive types, the obvious type for the statement

```
if test1 then new(point-impl) else new(other-point-impl)
```

is point-impl $\vee$ other-point-impl. We can infer this, provided a disjunction subsumes its disjuncts:

$$\Gamma \vdash x \leq x \vee y \qquad\qquad \Gamma \vdash y \leq x \vee y$$

In the opposite direction, we would need the rule:

$$\frac{\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_i \times \cdots \times x_n \rightarrow y) \qquad \Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_i' \times \cdots \times x_n \rightarrow y)}{\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_i \vee x_i' \times \cdots \times x_n \rightarrow y)}$$

This rule supports the case analysis needed for the example. In particular, notice that the rule performs a case analysis on one argument at a time. That means that all possible combinations of arguments will be considered.

A natural alternative to using disjunctive types is to talk about the properties that two types have in common, and then to introduce a type with just those properties. This approach does not work in general, because interfaces with multi-methods do not necessarily correspond to types. In particular, we could not treat the example of point-impl and other-point-impl.

5 Execution

So far, we have been implicitly assuming that there is some execution mechanism, and that it is affected by the programmer's assertions. This section formalizes such an execution mechanism.

In particular, we demonstrate how to execute an operation with several implementations which may return results of different types. The execution mechanism selects an implementation on the basis of the classes of the arguments but also of the class of result needed by the context (for the rest of the computation). While the assertional view does not require this capability of dispatching on output types, we show how it supports it.

First, we present a simple, compelling, but impractical expression evaluator. This evaluator does not rely on typechecking, and hence it is forced to guess the types of subexpressions. Roughly, given an expression $\mathsf{F}(\mathsf{T}_1,\ldots,\mathsf{T}_n)$ and a desired result type y, the evaluator may non-deterministically try any implementation of F suggested by an assertion of the form $\mathrm{prim}(\mathsf{F}\colon x_1 \times \cdots \times x_n \rightarrow y;\ f)$ or $\mathrm{cand}(\mathsf{F}\colon x_1 \times \cdots \times x_n \rightarrow y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$, and evaluates the subexpressions $\mathsf{T}_1, \ldots, \mathsf{T}_n$ guessing that they have types $x_1, \ldots, x_n$, respectively.

Then, we show how type information can be related to the behavior of this evaluator. Basically, $\mathsf{T}\colon x$ means that the evaluator will not run out of options when applied to T to get a value of type x. The evaluator may not terminate, but it never gets stuck. Similarly, $\mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \rightarrow y)$ means that the evaluator will not run out of options when applied to an invocation of F with the goal of obtaining a result of type y and with arguments of types $x_1, \ldots, x_n$.

Finally, we present a modified evaluator that uses the results of typechecking to avoid dead ends. Roughly, given an expression $\mathsf{F}(\mathsf{T}_1,\ldots,\mathsf{T}_n)$ and a desired result type y, the evaluator identifies types, $x_1, \ldots, x_n$ for the respective subexpressions $\mathsf{T}_1, \ldots, \mathsf{T}_n$ such that an implementation $\mathrm{prim}(\mathsf{F}\colon x_1 \times \cdots \times x_n \rightarrow y;\ f)$ or $\mathrm{method}(\mathsf{F}\colon x_1 \times \cdots \times x_n \rightarrow y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$ has been inferred. It then evaluates each subexpression T_i for the corresponding desired result type x_i and applies the implementation to the collected results. It never needs to explore alternative branches.

$$\Delta, \mathsf{v} \mapsto val, \Delta' \vdash \mathsf{v}: x \Rightarrow val \qquad \text{if } \Gamma \vdash \text{class}(val) \leq x$$

$$\Delta \vdash \mathsf{new(c)}: x \Rightarrow val \qquad \text{if } \Gamma \vdash \mathsf{c} \leq x \text{ and } \Gamma \vdash \text{class}(val) \leq \mathsf{c}$$

$$\frac{\Delta \vdash \mathsf{S}: x \Rightarrow val' \qquad \Delta, \mathsf{v} \mapsto val' \vdash \mathsf{T}: y \Rightarrow val}{\Delta \vdash \mathsf{let\ v = S\ in\ T}: y \Rightarrow val}$$

$$\frac{\Delta \vdash \mathsf{S}: \mathsf{bool} \Rightarrow \mathsf{true} \qquad \Delta \vdash \mathsf{T}_1: x \Rightarrow val}{\Delta \vdash \mathsf{if\ S\ then\ T_1\ else\ T_2}: x \Rightarrow val}$$

$$\frac{\Delta \vdash \mathsf{S}: \mathsf{bool} \Rightarrow \mathsf{false} \qquad \Delta \vdash \mathsf{T}_2: x \Rightarrow val'}{\Delta \vdash \mathsf{if\ S\ then\ T_1\ else\ T_2}: x \Rightarrow val}$$

$$\frac{\Delta \vdash \mathsf{T}_1: x_1 \Rightarrow val_1 \quad \cdots \quad \Delta \vdash \mathsf{T}_n: x_n \Rightarrow val_n}{\Delta \vdash \mathsf{F}(\mathsf{T}_1, \ldots, \mathsf{T}_n): y \Rightarrow f(val_1, \ldots, val_n)} \quad \text{if } \Gamma \vdash \text{prim}(\mathsf{F}: \text{class}(val_1) \times \cdots \times \text{class}(val_n) \rightarrow y;\ f)$$

$$\frac{\begin{array}{c}\Delta \vdash \mathsf{T}_1: x_1 \Rightarrow val_1 \quad \cdots \quad \Delta \vdash \mathsf{T}_n: x_n \Rightarrow val_n \\ \mathsf{v}_1 \mapsto val_1, \ldots, \mathsf{v}_n \mapsto val_n \vdash \mathsf{T}: y \Rightarrow val\end{array}}{\Delta \vdash \mathsf{F}(\mathsf{T}_1, \ldots, \mathsf{T}_n): y \Rightarrow val} \quad \text{if } \Gamma \vdash \text{cand}(\mathsf{F}: \text{class}(val_1) \times \cdots \times \text{class}(val_n) \rightarrow y;\ \mathsf{v}_1, \ldots, \mathsf{v}_n.\mathsf{T})$$

Fig. 2. Rules for the simple evaluator

5.1 A Simple Evaluator

The rules for the evaluator are shown in Figure 2. The evaluator is based on judgments of the form:

$$\Delta \vdash \mathsf{S}: x \Rightarrow val$$

Such a judgment means that the expression S reduces to *val*, with the variable bindings given in Δ and the goal type x. The evaluator returns *val* if such a judgment can be proved. It thus relies on a non-deterministic search.

The bindings in Δ have the form $\mathsf{v} \mapsto val$. Whenever we write $\Delta \vdash \mathsf{S}: x \Rightarrow val$ we implicitly assume that Δ is well formed, in the sense that it binds each variable at most once. Correspondingly, we interpret terms up to renaming of bound variables.

In addition to the bindings in Δ, which can change during the process of evaluation, the evaluator is defined relative to an environment of method applicability and typing information, Γ, which remains constant throughout an evaluation. We assume some environment Γ is given, and consists only of assertions about primitives, candidates, and subtypings. We do not explicitly display Γ in evaluation judgments, but show it whenever it is used to prove some side condition.

The evaluator does not depend on the particular choice of the universe of values. The following basic assumptions about values and their classes suffice.

We assume that there is a function which gives the class class(val) of any value val. In order for the evaluator to be correct, the primitives it calls upon must be correct. In particular, they must return values of the types they were declared to return:

$$\begin{array}{rl} \text{if } & \Gamma \vdash \text{prim}(\mathsf{F}{:}\, x_1 \times \cdots \times x_n \rightarrow y;\ f) \\ \text{and } & \Gamma \vdash \text{class}(val_1) \leq x_1, \ldots, \Gamma \vdash \text{class}(val_n) \leq x_n \\ \text{then } & \Gamma \vdash \text{class}(f(val_1, \ldots, val_n)) \leq y \end{array}$$

In order for conditionals to work as expected, true and false must be the only booleans:

$$\begin{array}{c} \text{if } \Gamma \vdash \text{class}(val) \leq \mathsf{bool} \\ \text{then } val = \mathsf{true} \text{ or } val = \mathsf{false} \end{array}$$

Finally, in order for the execution of the new form to be able to succeed, there must be a value of each class. In an actual language, new may return a fresh value each time it is called; but we do not rely on any such condition for type reasoning.

We obtain the reassuring result that, when the evaluator returns a value at all, this value is of the requested type:

Theorem 1. *If $\Delta \vdash \mathsf{T}{:}\, x \Rightarrow val$ then $\Gamma \vdash class(val) \leq x$.*

Proof sketch: By an easy induction on the derivation of $\Delta \vdash \mathsf{T}{:}\, x \Rightarrow val$. □

In addition, we obtain a result about evaluation and subsumption:

Theorem 2. *If $\Delta \vdash \mathsf{T}{:}\, x \Rightarrow val$ and $\Gamma \vdash x \leq y$ then $\Delta \vdash \mathsf{T}{:}\, y \Rightarrow val$.*

Proof sketch: By induction on the derivation of $\Delta \vdash \mathsf{T}{:}\, x \Rightarrow val$. We consider the rule instance used in the last evaluation step of $\Delta \vdash \mathsf{T}{:}\, x \Rightarrow val$, and replace it with the analogous rule instance, with y in place of x. □

In order to evaluate a term T with type x and bindings Δ, we intend to work by backward chaining on the reduction rules. On occasion, the evaluator must choose types for evaluating subterms of T. With this non-deterministic algorithm in mind, we now define what it means for the simple evaluator to get stuck, and then prove that this does not happen in the evaluation of well-typed terms.

We write:

$$\Delta \vdash \mathsf{T}{:}\, x \text{ stuck}$$

to mean that the evaluation of T with type x and bindings Δ will lead to dead ends on all its possible paths. Conversely, if $\Delta \vdash \mathsf{T}{:}\, x$ stuck is not provable, then the evaluation of T will find some path that does not dead end, even though the path may not terminate. Figure 3 gives the rules for proving the judgment $\Delta \vdash \mathsf{T}{:}\, x$ stuck. To each form of term corresponds exactly one rule. This rule codifies the condition that, no matter what types are chosen for evaluating subterms, either no reduction rule is applicable to the term or the evaluator will get stuck on a subterm.

The proof of the theorem that the simple evaluator does not get stuck in the evaluation of well-typed terms requires two lemmas about type inference.

$$\Delta \vdash \mathsf{v}\colon x \text{ stuck} \qquad \begin{array}{l}\text{if } \mathsf{v} \text{ is not bound in } \Delta\\ \text{or } \mathsf{v} \text{ is bound in } \Delta \text{ to a value}\\ \quad \text{with a class not a subtype of } x\end{array}$$

$$\Delta \vdash \mathsf{new(c)}\colon x \text{ stuck} \qquad \text{if not } \Gamma \vdash \mathsf{c} \le x$$

$$\frac{\begin{array}{l}\text{for all } x,\\ \quad \text{either } \Delta \vdash \mathsf{S}\colon x \text{ stuck}\\ \quad \text{or for some } val',\ \Delta \vdash \mathsf{S}\colon x \Rightarrow val' \text{ and } \Delta, \mathsf{v} \mapsto val' \vdash \mathsf{T}\colon y \text{ stuck}\end{array}}{\Delta \vdash \mathsf{let\ v = S\ in\ T}\colon y \text{ stuck}}$$

$$\frac{\begin{array}{l}\text{either } \Delta \vdash \mathsf{S}\colon \mathsf{bool} \text{ stuck}\\ \text{or } \Delta \vdash \mathsf{S}\colon \mathsf{bool} \Rightarrow val' \text{ and } val' \text{ is neither } \mathsf{true} \text{ nor } \mathsf{false}\\ \text{or } \Delta \vdash \mathsf{S}\colon \mathsf{bool} \Rightarrow \mathsf{true} \text{ and } \Delta \vdash \mathsf{T}_1\colon x \text{ stuck}\\ \text{or } \Delta \vdash \mathsf{S}\colon \mathsf{bool} \Rightarrow \mathsf{false} \text{ and } \Delta \vdash \mathsf{T}_2\colon x \text{ stuck}\end{array}}{\Delta \vdash \mathsf{if\ S\ then\ T_1\ else\ T_2}\colon x \text{ stuck}}$$

$$\frac{\begin{array}{l}\text{for all } x_1, \ldots, x_n,\\ \quad \text{either for some } i,\ \Delta \vdash \mathsf{T}_i\colon x_i \text{ stuck}\\ \quad \text{or for some } val_1, \ldots, val_n,\ \Delta \vdash \mathsf{T}_1\colon x_1 \Rightarrow val_1 \quad \cdots \quad \Delta \vdash \mathsf{T}_n\colon x_n \Rightarrow val_n\\ \qquad\quad \text{and there is no } f \text{ such that } \Gamma \vdash \mathrm{prim}(\mathsf{F}\colon \mathrm{class}(val_1) \times \cdots \times \mathrm{class}(val_n) \to y;\ f)\\ \qquad\quad \text{and for every } \mathsf{v}_1, \ldots, \mathsf{v}_n, \mathsf{T}\\ \qquad\qquad\quad \text{such that } \Gamma \vdash \mathrm{cand}(\mathsf{F}\colon \mathrm{class}(val_1) \times \cdots \times \mathrm{class}(val_n) \to y;\ \mathsf{v}_1, \ldots, \mathsf{v}_n.\mathsf{T})\\ \qquad\qquad\qquad \mathsf{v}_1 \mapsto val_1, \ldots, \mathsf{v}_n \mapsto val_n \vdash \mathsf{T}\colon y \text{ stuck}\end{array}}{\Delta \vdash \mathsf{F(T_1, \ldots, T_n)}\colon y \text{ stuck}}$$

Fig. 3. Rules for the simple evaluator getting stuck

Lemma 3. *If $\Gamma, \mathsf{v}_1\colon x_1, \ldots, \mathsf{v}_n\colon x_n, \Gamma' \vdash \mathsf{T}\colon y$ and $\Gamma \vdash x_1' \le x_1, \ldots, \Gamma \vdash x_n' \le x_n$, then $\Gamma, \mathsf{v}_1\colon x_1', \ldots, \mathsf{v}_n\colon x_n', \Gamma' \vdash \mathsf{T}\colon y$.*

Proof sketch: By induction on the proof of $\Gamma, \mathsf{v}_1\colon x_1, \ldots, \mathsf{v}_n\colon x_n, \Gamma' \vdash \mathsf{T}\colon y$. As environments simply get moved up the rules, the only interesting case is that for the axiom rule $\Gamma, C, \Gamma' \vdash C$, and it is trivial. □

Lemma 4. *If $\Gamma \vdash OK(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y)$ and $\Gamma, OK(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y), \Gamma' \vdash A$ then $\Gamma, \Gamma' \vdash A$.*

Proof sketch: By induction on the proof of $\Gamma, \mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y), \Gamma' \vdash A$. The only interesting case is that for the axiom rule. If the formula on the right is $\mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y)$ itself, then we obtain the desired result by weakening from $\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y)$; otherwise, we obtain it directly by the axiom rule. □

In order to state the theorem, we define a function for extracting type assignments from bindings:

$$E(\mathsf{v}_1 \mapsto val_1, \ldots, \mathsf{v}_n \mapsto val_n) \equiv \mathsf{v}_1\colon \mathrm{class}(val_1), \ldots, \mathsf{v}_n\colon \mathrm{class}(val_n)$$

The theorem is:

Theorem 5. *If $\Gamma, E(\Delta) \vdash \mathsf{T}{:}\,x$, then it is not the case that $\Delta \vdash \mathsf{T}{:}\,x$* stuck.

Proof: The argument is by induction on the structure of the proof of $\Delta \vdash \mathsf{T}{:}\,x$ stuck. We show that none of the rules of Figure 3 is applicable to a well-typed term.

- The rule for variables never applies because $\Gamma, E(\Delta) \vdash \mathsf{v}{:}\,x$, and hence v is bound in Δ, and the class of the value to which it is bound must be a subtype of x.
- The rule for the new form never applies, because $\Gamma, E(\Delta) \vdash \mathsf{new(c)}{:}\,x$ can be obtained only if $\Gamma \vdash \mathsf{c} \leq x$.
- In the case of the let form, the assumption $\Gamma, E(\Delta) \vdash \mathsf{let\ v{=}S\ in\ T}{:}\,y$ must have been inferred from $\Gamma, E(\Delta) \vdash \mathsf{S}{:}\,x$ and $\Gamma, E(\Delta), \mathsf{v}{:}\,x \vdash \mathsf{T}{:}\,y'$ for some x and some $\Gamma \vdash y' \leq y$. Since $\Gamma, E(\Delta) \vdash \mathsf{S}{:}\,x$ it cannot be that $\Delta \vdash \mathsf{S}{:}\,x$ stuck, by the induction hypothesis. Take any val' such that $\Delta \vdash \mathsf{S}{:}\,x \Rightarrow val'$. By Theorem 1, we must have $\Gamma \vdash \mathrm{class}(val') \leq x$, and by Lemma 3 we must have $\Gamma, E(\Delta), \mathsf{v}{:}\,\mathrm{class}(val') \vdash \mathsf{T}{:}\,y'$. Since $\Gamma \vdash y' \leq y$, we have by subsumption $\Gamma, E(\Delta), \mathsf{v}{:}\,\mathrm{class}(val') \vdash \mathsf{T}{:}\,y$, that is, $\Gamma, E(\Delta, \mathsf{v} \mapsto val') \vdash \mathsf{T}{:}\,y$. Therefore, by the induction hypothesis, it cannot be that $\Delta, \mathsf{v} \mapsto val' \vdash \mathsf{T}{:}\,y$ stuck.
- In the case of conditionals, the assumption $\Gamma, E(\Delta) \vdash \mathsf{if\ S\ then\ T_1\ else\ T_2}{:}\,x$ must have been inferred from $\Gamma, E(\Delta) \vdash \mathsf{S}{:}\,\mathsf{bool}$, $\Gamma, E(\Delta) \vdash \mathsf{T}_1{:}\,x'$, and $\Gamma, E(\Delta) \vdash \mathsf{T}_2{:}\,x'$ for some x' such that $\Gamma \vdash \leq x$. Since $\Gamma \vdash x' \leq x$, we have $\Gamma, E(\Delta) \vdash \mathsf{T}_1{:}\,x$ and $\Gamma, E(\Delta) \vdash \mathsf{T}_2{:}\,x$ by subsumption. Hence, by the induction hypothesis, the first, third, and fourth cases of the stuck rule do not apply. The second case does not apply because if $\Delta \vdash \mathsf{S}{:}\,\mathsf{bool} \Rightarrow val$ then $\Gamma \vdash \mathrm{class}(val) \leq \mathsf{bool}$ by Theorem 1, while by our assumption on booleans val must be either true or false.
- In the case of invocation, the assumption $\Gamma, E(\Delta) \vdash \mathsf{F}(\mathsf{T}_1,\ldots,\mathsf{T}_n){:}\,y$ implies that $\Gamma \vdash \mathrm{OK}(\mathsf{F}{:}\,x_1 \times \cdots \times x_n \to y')$ and $\Gamma, E(\Delta) \vdash \mathsf{T}_1{:}\,x_1, \ldots, \Gamma, E(\Delta) \vdash \mathsf{T}_n{:}\,x_n$ for some $x_1, \ldots, x_n, y'$ such that $\Gamma \vdash y' \leq y$. Therefore, by the induction hypothesis, we cannot have $\Delta \vdash \mathsf{T}_i{:}\,x_i$ stuck for any i.
  For the other case, assume that there are $val_1, \ldots, val_n$ such that $\Delta \vdash \mathsf{T}_1{:}\,x_1 \Rightarrow val_1, \ldots, \Delta \vdash \mathsf{T}_n{:}\,x_n \Rightarrow val_n$. By Theorem 1, $\Gamma \vdash \mathrm{class}(val_1) \leq x_1, \ldots, \Gamma \vdash \mathrm{class}(val_n) \leq x_n$.
  Analyzing the proof of $\Gamma \vdash \mathrm{OK}(\mathsf{F}{:}\,x_1 \times \cdots \times x_n \to y')$, we obtain $\Gamma \vdash \mathrm{OK}(\mathsf{F}{:}\,x''_1 \times \cdots \times x''_n \to y'')$ for some $x''_1, \ldots, x''_n, y''$, where $\Gamma \vdash \mathrm{OK}(\mathsf{F}{:}\,x''_1 \times \cdots \times x''_n \to y'')$ can be proved directly by one of the two rules of introduction for OK and where $\Gamma \vdash x_1 \leq x''_1, \ldots, \Gamma \vdash x_n \leq x''_n$, $\Gamma \vdash y'' \leq y'$. The two cases are summarized in Figure 4.
  First we consider the rule that turns a primitive assertion into an OK assertion. In this case, we have $\Gamma \vdash \mathrm{prim}(\mathsf{F}{:}\,x''_1 \times \cdots \times x''_n \to y'';\ f)$ for some f. Since $\Gamma \vdash \mathrm{class}(val_1) \leq x_1 \leq x''_1, \ldots, \Gamma \vdash \mathrm{class}(val_n) \leq x_n \leq x''_n$, and $\Gamma \vdash y'' \leq y' \leq y$, it follows that $\Gamma \vdash \mathrm{prim}(\mathsf{F}{:}\,\mathrm{class}(val_1) \times \cdots \times \mathrm{class}(val_n) \to y;\ f)$ by the rules for subtyping.

$$
\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathrm{prim}(\mathsf{F}\colon x_1''\times\cdots\times x_n'' \to y'';\ f)}{\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1''\times\cdots\times x_n'' \to y'')}}{\vdots}}{\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1\times\cdots\times x_n \to y')}
\qquad
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\Gamma \vdash \mathrm{cand}(\mathsf{F}\colon x_1'''\times\cdots\times x_n''' \to y''';\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T}) \\ \Gamma, \mathrm{OK}(\mathsf{F}\colon x_1'''\times\cdots\times x_n''' \to y'''), \mathsf{v}_1\colon x_1''', \ldots, \mathsf{v}_n\colon x_n''' \vdash \mathsf{T}\colon y'''\end{array}}{\Gamma \vdash \mathrm{method}(\mathsf{F}\colon x_1'''\times\cdots\times x_n''' \to y''';\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})}}{\vdots}}{\Gamma \vdash \mathrm{method}(\mathsf{F}\colon x_1''\times\cdots\times x_n'' \to y'';\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})}}{\dfrac{\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1''\times\cdots\times x_n'' \to y'')}{\vdots}}}{\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1\times\cdots\times x_n \to y')}
$$

Fig. 4. The possible proofs of $\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1\times\cdots\times x_n \to y')$

Next we consider the rule that turns a method assertion into an OK assertion. In this case, we have $\Gamma \vdash \mathrm{method}(\mathsf{F}\colon x_1''\times\cdots\times x_n'' \to y''; \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$ for some v_1, ..., v_n, T. In turn, we obtain $\Gamma \vdash \mathrm{method}(\mathsf{F}\colon x_1''' \times \cdots \times x_n''' \to y'''; \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$ for some $x_1''', \ldots, x_n''', y'''$, where $\Gamma \vdash \mathrm{method}(\mathsf{F}\colon x_1'''\times\cdots\times x_n''' \to y'''; \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$ can be proved directly by the rule of introduction for method, and with $\Gamma \vdash x_1'' \le x_1''', \ldots, \Gamma \vdash x_n'' \le x_n'''$, and $\Gamma \vdash y''' \le y''$. Note at once that it follows that $\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1'''\times\cdots\times x_n''' \to y''')$.
The rule of introduction for method has antecedents $\Gamma, \mathrm{OK}(\mathsf{F}\colon x_1'''\times\cdots\times x_n''' \to y'''), \mathsf{v}_1\colon x_1''', \ldots, \mathsf{v}_n\colon x_n''' \vdash \mathsf{T}\colon y'''$ and $\Gamma \vdash \mathrm{cand}(\mathsf{F}\colon x_1''' \times \cdots \times x_n''' \to y'''; \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$. Since $\Gamma \vdash \mathrm{class}(val_1) \le x_1 \le x_1'' \le x_1''', \ldots, \Gamma \vdash \mathrm{class}(val_n) \le x_n \le x_n'' \le x_n'''$, and $\Gamma \vdash y''' \le y'' \le y' \le y$, it follows that $\Gamma \vdash \mathrm{cand}(\mathsf{F}\colon \mathrm{class}(val_1)\times\cdots\times\mathrm{class}(val_n) \to y; \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$ by the rules for subtyping.
Since we have $\Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1'''\times\cdots\times x_n''' \to y''')$ and $\Gamma, \mathrm{OK}(\mathsf{F}\colon x_1'''\times\cdots\times x_n''' \to y'''), \mathsf{v}_1\colon x_1''', \ldots, \mathsf{v}_n\colon x_n''' \vdash \mathsf{T}\colon y'''$, we also have $\Gamma, \mathsf{v}_1\colon x_1''', \ldots, \mathsf{v}_n\colon x_n''' \vdash \mathsf{T}\colon y'''$ by Lemma 4. Since $\Gamma \vdash \mathrm{class}(val_1) \le x_1''', \ldots, \Gamma \vdash \mathrm{class}(val_n) \le x_n'''$, we conclude $\Gamma, \mathsf{v}_1\colon \mathrm{class}(val_1), \ldots, \mathsf{v}_n\colon \mathrm{class}(val_n) \vdash \mathsf{T}\colon y'''$ by Lemma 3. By subsumption, since $\Gamma \vdash y''' \le y$, we have $\Gamma, \mathsf{v}_1\colon \mathrm{class}(val_1), \ldots, \mathsf{v}_n\colon \mathrm{class}(val_n) \vdash \mathsf{T}\colon y$. Therefore, by the induction hypothesis, we cannot have $\mathsf{v}_1 \mapsto val_1, \ldots, \mathsf{v}_n \mapsto val_n \vdash \mathsf{T}\colon y$ stuck.
In short, for our choice of $x_1, \ldots, x_n$, we do not have $\Delta \vdash \mathsf{T}_i\colon x_i$ stuck for any i. Moreover, for any $val_1, \ldots, val_n$, if $\Delta \vdash \mathsf{T}_1\colon x_1 \Rightarrow val_1, \ldots, \Delta \vdash \mathsf{T}_n\colon x_n \Rightarrow val_n$ then either there is an f such that $\Gamma \vdash \mathrm{prim}(\mathsf{F}\colon \mathrm{class}(val_1)\times\cdots\times\mathrm{class}(val_n) \to y;\ f)$ or there are $\mathsf{v}_1, \ldots, \mathsf{v}_n, \mathsf{T}$ such that $\Gamma \vdash \mathrm{cand}(\mathsf{F}\colon x_1\times\cdots\times x_n \to y; \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$ and not $\mathsf{v}_1 \mapsto val_1, \ldots, \mathsf{v}_n \mapsto val_n \vdash \mathsf{T}\colon y$ stuck. Therefore, the rule for stuck invocations does not apply. □

5.2 A More Practical Evaluator

As the proof of Theorem 5 suggests, type information could be used to inform the evaluator about which path to try. A more practical evaluator, shown in

$$\Delta, \mathsf{v} \mapsto val, \Delta' \vdash \mathsf{v}\colon x \Leftrightarrow val \qquad \text{if } \Gamma \vdash \mathrm{class}(val) \leq x$$

$$\Delta \vdash \mathsf{new(c)}\colon x \Leftrightarrow val \qquad \text{if } \Gamma \vdash \mathsf{c} \leq x \text{ and } \Gamma \vdash \mathrm{class}(val) \leq \mathsf{c}$$

$$\frac{\Delta \vdash \mathsf{S}\colon x \Leftrightarrow val' \qquad \Delta, \mathsf{v} \mapsto val' \vdash \mathsf{T}\colon y \Leftrightarrow val}{\Delta \vdash \mathsf{let\ v = S\ in\ T}\colon y \Leftrightarrow val}$$
$$\text{if } \Gamma, E(\Delta) \vdash \mathsf{S}\colon x \text{ and } \Gamma, E(\Delta), \mathsf{v}\colon x \vdash \mathsf{T}\colon y$$

$$\frac{\Delta \vdash \mathsf{S}\colon \mathsf{bool} \Leftrightarrow \mathsf{true} \qquad \Delta \vdash \mathsf{T}_1\colon x \Leftrightarrow val}{\Delta \vdash \mathsf{if\ S\ then\ T_1\ else\ T_2}\colon x \Leftrightarrow val}$$

$$\frac{\Delta \vdash \mathsf{S}\colon \mathsf{bool} \Leftrightarrow \mathsf{false} \qquad \Delta \vdash \mathsf{T}_2\colon x \Leftrightarrow val}{\Delta \vdash \mathsf{if\ S\ then\ T_1\ else\ T_2}\colon x \Leftrightarrow val}$$

$$\frac{\Delta \vdash \mathsf{T}_1\colon x_1 \Leftrightarrow val_1 \quad \cdots \quad \Delta \vdash \mathsf{T}_n\colon x_n \Leftrightarrow val_n}{\Delta \vdash \mathsf{F}(\mathsf{T}_1,\ldots,\mathsf{T}_n)\colon y \Leftrightarrow f(val_1, \ldots, val_n)}$$
$$\text{if } \Gamma \vdash \mathrm{prim}(\mathsf{F}\colon \mathrm{class}(val_1) \times \cdots \times \mathrm{class}(val_n) \to y;\ f)$$
$$\text{and } \Gamma, E(\Delta) \vdash \mathsf{T}_1\colon x_1, \ldots, \Gamma, E(\Delta) \vdash \mathsf{T}_n\colon x_n$$
$$\text{and } \Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y)$$

$$\frac{\begin{array}{c}\Delta \vdash \mathsf{T}_1\colon x_1 \Leftrightarrow val_1 \quad \cdots \quad \Delta \vdash \mathsf{T}_n\colon x_n \Leftrightarrow val_n \\ \mathsf{v}_1 \mapsto val_1, \ldots, \mathsf{v}_n \mapsto val_n \vdash \mathsf{T}\colon y \Leftrightarrow val\end{array}}{\Delta \vdash \mathsf{F}(\mathsf{T}_1,\ldots,\mathsf{T}_n)\colon y \Leftrightarrow val}$$
$$\text{if } \Gamma \vdash \mathrm{method}(\mathsf{F}\colon \mathrm{class}(val_1) \times \cdots \times \mathrm{class}(val_n) \to y;\ \mathsf{v}_1,\ldots,\mathsf{v}_n.\mathsf{T})$$
$$\text{and } \Gamma, E(\Delta) \vdash \mathsf{T}_1\colon x_1, \ldots, \Gamma, E(\Delta) \vdash \mathsf{T}_n\colon x_n$$
$$\text{and } \Gamma \vdash \mathrm{OK}(\mathsf{F}\colon x_1 \times \cdots \times x_n \to y)$$

Fig. 5. Rules for a more practical evaluator

Figure 5, takes advantage of type information to choose only paths that are guaranteed not to dead end.

The new reduction rules differ from the old ones only in that they include some type conditions. Therefore, if the new evaluator returns an answer, the original evaluator can also return that answer. Moreover, as we show below, the additional conditions do not cause the evaluator to get stuck on terms that typecheck.

The new evaluator can use the information provided by typechecking to choose appropriate output types for evaluating subexpressions. In fact, the evaluator need try only one execution branch; it can be completely deterministic. The choice of the branch can be efficient because type conditions can be checked at compile time.

Despite the restriction in its search space, the new evaluator does not get stuck. To formalize this claim, we define a stuck predicate for the new evaluator. We write:

$$\Delta \vdash \mathsf{T}\colon x \text{ stuck}'$$

to mean that the evaluation of T with type x and bindings Δ can lead to a

$$\Delta \vdash \mathsf{v}{:}\, x \text{ stuck}' \qquad \begin{array}{l}\text{if } \mathsf{v} \text{ is not bound in } \Delta \\ \text{or } \mathsf{v} \text{ is bound in } \Delta \text{ to a value} \\ \quad \text{with a class not a subtype of } x\end{array}$$

$$\Delta \vdash \mathsf{new(c)}{:}\, x \text{ stuck}' \qquad \text{if not } \Gamma \vdash \mathsf{c} \leq x$$

$$\frac{\begin{array}{l}\text{either there is no } x \text{ such that } E(\Delta) \vdash \mathsf{S}{:}\, x \text{ and } E(\Delta), \mathsf{v}{:}\, x \vdash \mathsf{T}{:}\, y \\ \text{or for some } x \text{ such that } E(\Delta) \vdash \mathsf{S}{:}\, x \text{ and } E(\Delta), \mathsf{v}{:}\, x \vdash \mathsf{T}{:}\, y, \\ \qquad \text{either } \Delta \vdash \mathsf{S}{:}\, x \text{ stuck}' \\ \qquad \text{or for some } val', \Delta \vdash \mathsf{S}{:}\, x \Leftrightarrow val' \text{ and } \Delta, \mathsf{v} \mapsto val' \vdash \mathsf{T}{:}\, y \text{ stuck}'\end{array}}{\Delta \vdash \mathsf{let\ v = S\ in\ T}{:}\, y \text{ stuck}'}$$

$$\frac{\begin{array}{l}\text{either } \Delta \vdash \mathsf{S}{:}\, \mathsf{bool} \text{ stuck}' \\ \text{or } \Delta \vdash \mathsf{S}{:}\, \mathsf{bool} \Leftrightarrow val' \text{ and } val' \text{ is neither } \mathsf{true} \text{ nor } \mathsf{false} \\ \text{or } \Delta \vdash \mathsf{S}{:}\, \mathsf{bool} \Leftrightarrow \mathsf{true} \text{ and } \Delta \vdash \mathsf{T}_1{:}\, x \text{ stuck}' \\ \text{or } \Delta \vdash \mathsf{S}{:}\, \mathsf{bool} \Leftrightarrow \mathsf{false} \text{ and } \Delta \vdash \mathsf{T}_2{:}\, x \text{ stuck}'\end{array}}{\Delta \vdash \mathsf{if\ S\ then\ T_1\ else\ T_2}{:}\, x \text{ stuck}'}$$

$$\frac{\begin{array}{l}\text{either there are no } x_1, \ldots, x_n \text{ such that} \\ \qquad \Gamma, E(\Delta) \vdash \mathsf{T}_1{:}\, x_1, \ \ldots, \ \Gamma, E(\Delta) \vdash \mathsf{T}_n{:}\, x_n \text{ and } \Gamma \vdash \mathrm{OK}(\mathsf{F}{:}\, x_1 \times \cdots \times x_n \to y) \\ \text{or for some } x_1, \ldots, x_n \\ \quad \text{such that } \Gamma, E(\Delta) \vdash \mathsf{T}_1{:}\, x_1, \ \ldots, \ \Gamma, E(\Delta) \vdash \mathsf{T}_n{:}\, x_n \text{ and } \Gamma \vdash \mathrm{OK}(\mathsf{F}{:}\, x_1 \times \cdots \times x_n \to y), \\ \qquad \text{either for some } i, \ \Delta \vdash \mathsf{T}_i{:}\, x_i \text{ stuck}' \\ \qquad \text{or for some } val_1, \ldots, val_n, \ \Delta \vdash \mathsf{T}_1{:}\, x_1 \Leftrightarrow val_1 \quad \cdots \quad \Delta \vdash \mathsf{T}_n{:}\, x_n \Leftrightarrow val_n \\ \qquad\quad \text{and there is no } f \text{ such that } \Gamma \vdash \mathrm{prim}(\mathsf{F}{:}\, \mathrm{class}(val_1) \times \cdots \times \mathrm{class}(val_n) \to y;\ f) \\ \qquad\quad \text{and for every } \mathsf{v}_1, \ldots, \mathsf{v}_n, \mathsf{T} \\ \qquad\qquad \text{such that } \Gamma \vdash \mathrm{method}(\mathsf{F}{:}\, \mathrm{class}(val_1) \times \cdots \times \mathrm{class}(val_n) \to y;\ \mathsf{v}_1, \ldots, \mathsf{v}_n.\mathsf{T}) \\ \qquad\qquad\quad \mathsf{v}_1 \mapsto val_1, \ldots, \mathsf{v}_n \mapsto val_n \vdash \mathsf{T}{:}\, y \text{ stuck}'\end{array}}{\Delta \vdash \mathsf{F}(\mathsf{T}_1, \ldots, \mathsf{T}_n){:}\, y \text{ stuck}'}$$

Fig. 6. Rules for the more practical evaluator getting stuck

dead end on some path compatible with the type conditions. The rules for this predicate are given in Figure 6. We obtain:

Theorem 6. *If* $\Gamma, E(\Delta) \vdash \mathsf{T}{:}\, x$, *then it is not the case that* $\Delta \vdash \mathsf{T}{:}\, x$ stuck$'$.

Proof sketch: The proof is essentially the same as that of Theorem 5. The only restriction of the simple evaluator is the appearance of type conditions, which hold for the rule instances used in the proof of Theorem 5. □

Type information can thus be exploited in limiting the search space for reduction. In fact, using type information, there seems to be no obstacle to turning the evaluator into an efficient algorithm, while allowing dispatch on both input types and output types.

6 Conclusion

A method declaration can naturally be viewed as an assertion that the method body implements the operation in the appropriate circumstances. This view can lead to a simple, modular system for expressing and reasoning about object oriented programs. The system allows methods to be defined on interfaces and handles multi-methods.

This paper has only tried to motivate the treatment of methods as assertions and to show that it can be given a correct formalization. Many issues remain to be explored. Technically, it seems interesting to consider more powerful type systems and to widen the range of properties that assertions can express. Matters of language design also deserve consideration. In particular, a programming language based on the assertional view should provide convenient support for modularity, within a sound methodology.

Acknowledgments

Luca Cardelli, Mike Dixon, Georges Gonthier, and Gregor Kiczales have helped clarify these ideas. Dixon and Kiczales commented on earlier drafts of this paper. Anonymous referees made many useful suggestions on the presentation of the paper.

References

1. Agrawal, R., DeMichiel, L. G., and Lindsay, B. G. Static type checking of multi-methods. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (1991), ACM. Also published in *SIGPLAN Notices*, 16(11) (1991), pp. 113–128.
2. Andreoli, J.-M., and Pareschi, R. Linear objects: logical processes with built-in inheritance. In *Proceedings of the Seventh International Conference on Logic Programming* (1990), D. H. D. Warren and P. Szeredi, Eds., MIT Press, pp. 495–510.
3. Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. Common Lisp object system specification. *Sigplan Notices 23*, Special Issue (1988).
4. Bobrow, D. G., Gabriel, R. P., and White, J. L. CLOS in context: the shape of the design space. In *Object Oriented Programming: The CLOS Perspective*, A. Paepcke, Ed. MIT Press, 1993, pp. 29–61.
5. Canning, P., Cook, W., Hill, W., Mitchell, J., and Olthoff, W. F-bounded quantification for object-oriented programming. In *Functional Programming and Computer Architecture* (1989).
6. Cardelli, L. A semantics of multiple inheritance. *Information and Computation*, 76 (1988).
7. Cardelli, L., and Wegner, P. On understanding types, data abstraction and polymorphism. *Computing Surveys 17*, 4 (1985).
8. Castagna, G., Ghelli, G., and Longo, G. A calculus for overloaded functions with subtyping. In *ACM Conference on LISP and Functional Programming* (1992). Full paper to appear in *Information and Computation*.
9. Chambers, C. The Cecil language: Specification and rationale. Tech. Rep. 93-03-05, Department of Computer Science, University of Washington, 1993.
10. Chambers, C. Predicate classes. In *Proceedings of the European Conference on Object-Oriented Programming* (1993).
11. Dixon, M. *Embedded Computation and the Semantics of Programs.* PhD thesis, Stanford University, 1991. Also published as Xerox PARC technical report SSL-91-1.
12. Goldberg, A., and Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.
13. Hodas, J. S., and Miller, D. Representing objects in a logic programming language with scoping constructs. In *Proceedings of the Seventh International Conference on Logic Programming* (1990), D. H. D. Warren and P. Szeredi, Eds., MIT Press, pp. 511–526.
14. Kiczales, G. Traces (a cut at the "make isn't generic" problem). In *Proceedings of International Symposium on Object Technologies for Advanced Software* (1993),

S. Nishio and A. Yonezawa, Eds., JSST, Springer-Verlag, pp. 27–43. Lecture Notes in Computer Science 742.
15. Pierce, B. C., and Turner, D. Object oriented programming without recursive types. In *ACM Symposium on Principles of Programming Languages* (1993).
16. Shapiro, E., and Takeuchi, A. Object oriented programming in Concurrent Prolog. *New Generation Computing 1* (1983), 25–48.
17. Stroustrup, B. *The C++ Programming Language, Second Edition.* Addison-Wesley, 1991.
18. Takashio, K., and Tokoro, M. DROL: An object-oriented programming language for distributed real-time systems. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications* (1992), pp. 276–294.

This article was processed using the LaTeX macro package with LLNCS style