On Binary Methods

Kim Bruce[*]
Department of Computer Science, Williams College, Williamstown, Massachusetts 01267, USA.

Luca Cardelli[†]
Digital Equipment Corporation, Systems Research Center, 130 Lytton Ave, Palo Alto, CA 94301, USA.

Giuseppe Castagna[‡]
(CNRS) LIENS-DMI, École Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France.

The Hopkins Objects Group[§]
Department of Computer Science, The Johns Hopkins University, Baltimore, Maryland 21218. USA

Gary T. Leavens[¶]
229 Atanasoff Hall, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, USA.

Benjamin Pierce[||]
Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, United Kingdom.

Giving types to binary methods causes significant problems for object-oriented language designers and programmers. This paper offers a comprehensive description of the problems arising from typing binary methods, and collects and contrasts diverse views and solutions. It summarizes the current debate on the problem of binary methods for a wide audience.

1 Introduction

Binary methods have caused great difficulty for designers of strongly typed object-oriented languages and for programmers using those languages. In this paper we study the sources of these problems and compare and contrast a variety of solutions.

The authors of this paper have differing views on what the most appropriate solutions are. We have attempted here to collect together the solutions that individuals among us advocate and to present a consensus on what can be fairly stated as strengths and weaknesses of each approach. This paper grew from presentations and discussions at the 2nd Workshop on Foundations of Object-Oriented Languages, which was sponsored by NSF and ESPRIT and held in Paris in June, 1994 [19].

Let us begin by fixing some basic terminology. A *class* is the code that defines the *instance variables* and *methods* of some *objects*. The objects that conform to this definition are called *instances* of the class. (The issues that we discuss also arise in delegation-based languages; for simplicity we concentrate on classes.) In this article we use **new** as a primitive that generates an instance of a class from the class name (and some initial values for its instance variables). An *interface type*, also called an *object type* or simply a *type* contains the names of the object's methods, and the types of each method's arguments and results. Due to subtyping, an object may have multiple interface types; what we mean when we mention the "type of an object" is the least such type (i.e., the most specific such type). Similarly several classes may

[*] partially supported by NSF grant CCR-9121778 and NSF grant CCR-9424123. Internet:`kim@cs.williams.edu`

[†] Internet: `luca@src.dec.com.`

[‡] Internet: `castagna@dmi.ens.fr`

[§] Jonathan Eifrig, Scott Smith, Valery Trifonov. Contact Scott Smith. Research partially supported by NSF grant CCR-9301340 and AFOSR grant F49620-93-1-0169. Internet: `scott@cs.jhu.edu`.

[¶] partially supported by NSF grants CCR-9108654 and CCR-9593168. Internet: `leavens@cs.iastate.edu`.

[||] Internet: `benjamin.pierce@cl.cam.ac.uk` generate objects of the same interface type if the hidden implementations are distinct, but the public methods have the same type. Usually, for a class named *SomeClass*, the interface type of its objects is written *Some*; that is, we use a naming convention where dropping *Class* from the end of a class name is used to give a type name. In a few sections, classes are identified with types, as they are in languages like C++ and Eiffel. We will note this explicitly in those sections, and use the name *Some* both as the name of the class (we will not use the *Class* suffix) and as the (least) interface type of objects of that class.

Binary operations which take two arguments of the same type are quite familiar in non-object-oriented languages. Typical examples include arithmetic operations on number objects, as well as binary relations such as = and $<$, and set operations like subset and union. In object-oriented languages these operations are generally written as methods. In this case the first argument of the binary operation becomes the receiver of a corresponding "message", with the second parameter becoming the only argument. Consequently, we define a binary method of some object of type τ as a method that has an argument of the same type τ. Such a method is *binary* in the sense that it acts on two objects of the same type: the object passed as argument and the receiving object itself. In general, a binary method could also include other arguments (including other arguments of the same type); by a standard abuse of terminology we still refer to these as binary methods. We provide examples in an object-oriented style later.

A *subclass* is code that extends a class or classes (called the *superclasses* of the subclass). Subclasses *inherit* definitions of instance variables and methods from their superclasses. A subclass may also *override* the definitions of methods it would otherwise inherit by redefining them. Because a subclass inherits code for methods, it also inherits interface type information for the methods that it does not override.

The most significant problem with binary methods lies in their typing in the presence of inheritance. The source of this problem is that the type of the argument of a binary method naturally should change in parallel to changes in the type of the object produced by the subclass. The difficulty is that these type changes may result in subclasses which may no longer produce subtypes. On the other hand if inheritance is limited to always produce subtypes then useful subclasses can not be directly defined, and work-arounds must be found. A second problem is the asymmetry of a binary method: the method may have privileged access to only one of the two objects the method is invoked on. These two problems are described in more detail in Section 2.

Sections 3 and 4 concentrate on solutions to the problem of typing binary methods in the presence of inheritance. We consider the question from two sides: in Section 3, we reflect on whether it actually need be solved at all (i.e., whether binary operations might best *not* be treated as methods); in Section 4, we meet the problem head-on and review some solutions that have been proposed.

Turning to the problem of privileged access, Section 5 sketches a technique by which object-style data encapsulation can be blended with conventional ADT-style encapsulation to allow implementation of binary operations with privileged access to object representations.

Section 6 offers concluding remarks.

Although it is difficult to form a complete list of criteria used to evaluate different solutions to the binary methods problem, a partial list of general criteria could be formulated as follows.

1. How expressive is the solution? In particular, to what extent does it allow reasonable subclassing and message sends?

2. Do subclasses always produce subtypes?

3. Do binary methods have privileged access to the argument's state?

4. Is program development modular? In particular, does adding a new class ever force modification to existing code, and can module interfaces be defined?

5. Are the receiver and argument of a binary method treated symmetrically?

6. Does the solution avoid unnecessary code duplication?

These criteria are used to evaluate the different approaches in the sections that follow.

We only consider type systems that are *sound* in the sense that code that statically passes the type system cannot produce type errors at run time. Therefore we do not consider constructs that allow one to escape from the type system (by means such as a "cast" in C++) to be a "solution" to the problems posed by binary methods; such type systems cannot guarantee soundness without run-time checks. We also ignore solutions based on the **typecase** construct, since it is not general enough to avoid the problems that message passing is supposed to solve; see Section 4.2.2 for further details.

2 The Problem of Binary Methods

This section describes the problems caused by binary methods. The first subsection describes typing problems in the presence of inheritance, and the second describes problems with privileged access.

2.1 Typing Binary Methods in the Presence of Inheritance

In procedural or functional languages, the type of a binary function that takes two arguments of type τ and returns a value of type σ is written $\tau \times \tau \rightarrow \sigma$. In an object-oriented language, functions or procedures are typically replaced by methods belonging to a class corresponding to one of the arguments. Figure 1 shows a standard example of a class with

```
class PointClass
   instance variables
      xValue: real
      yValue: real
   methods
      x: real is return(xValue)
      y: real is return(yValue)
      equal(p: Point): bool is
           return( (xValue==p.x) && (yValue==p.y) )
end class
```

Figure 1: The class *PointClass*.

a binary method. In *PointClass*, the method *equal*, which tests for equality with another instance of *PointClass*, is written with a single parameter of type *Point*, the type of objects instantiated from *PointClass*. As may be seen in this example, binary operations—when regarded as methods—are asymmetric: the receiver plays a role somewhat different than the parameter. This distinction is highlighted when we define a subclass of a class with a binary method.

Figure 2 defines a subclass *ColorPointClass* of *PointClass*. In *ColorPointClass*, the type of the parameter of *equal* is changed to *ColorPoint* to match the type of the receiver, allowing two *ColorPoint* objects to be compared by the *equal* method, which overrides the behavior of *equal* for points.

```
class ColorPointClass subclass of  PointClass
   instance variables
       -- xValue and yValue are inherited
      cValue : string
   methods
       -- x and y are inherited
      c: string is return(cValue)
       -- equal is overriden
      equal(p: ColorPoint): bool is
           return( (cValue==p.c) && (xValue==p.x)
                       && (yValue==p.y) )
end class
```

Figure 2: The class *ColorPointClass*.

We generally write object types similarly to the type of the record of methods.[2] Therefore instances of *PointClass* and *ColorPointClass* have the following object types:

$$\mathit{Point} \equiv OT\ \langle\!\langle x\colon \mathit{real};\ y\colon \mathit{real};\\ \mathit{equal}\colon \mathit{Point} \to \mathit{bool}\rangle\!\rangle$$

$$\mathit{ColorPoint} \equiv OT\ \langle\!\langle x\colon \mathit{real};\ y\colon \mathit{real}; c\colon \mathit{string};\\ \mathit{equal}\colon \mathit{ColorPoint} \to \mathit{bool}\rangle\!\rangle$$

The prefix *OT* is used to distinguish object types from record types. Note that both of these definitions happen to be recursive: the type being defined appears on the right-hand side of the $\equiv$. It is not uncommon for the type being defined to appear as either an argument or result type in its methods.

Informally, a type σ is a *subtype* of τ, written $\sigma <: \tau$, if an expression of type σ can be used in any context that expects an expression of type τ (cf. [15, 16, 49]). Associated with subtyping is the principle of *subsumption* (subtype polymorphism): if $\sigma <: \tau$ and a program fragment has type σ, it also has type τ. A simple example of subtyping in object-oriented programming is that an object type is a subtype of the type with some methods removed, as any context that expects the object with fewer methods will not directly use the extra methods and thus no type errors will occur. In fact it is also possible to replace the type of any method by a subtype and still have the resulting object types in the subtype relation. Thus the general rule is *OT* $\langle\!\langle m_1\colon S_1, \ldots, m_n\colon S_n, \ldots, m_{n+k}\colon S_{n+k}\rangle\!\rangle <: OT\,\langle\!\langle m_1\colon T_1,$ $\ldots, m_n\colon T_n\rangle\!\rangle$ (with $k \geq 0$) if and only if, for each $i \in \{1..n\}$, $S_i <: T_i$.

The rule for subtyping functions states that $\sigma \to \tau <: \sigma' \to \tau'$ if and only if $\sigma' <: \sigma$ and $\tau <: \tau'$ [15]. (This is sometimes called the "contravariant rule" because it is contravariant in the left argument of $\to$.) This rule is informally justified by the following. If f is expected to have type $\sigma' \to \tau'$, but actually has type $\sigma \to \tau$, then f can be passed an argument of type σ' when (by subsumption) $\sigma' <: \sigma$; furthermore, the result of such a call will have type τ, which (by subsumption) can be considered to be of type τ'. Hence all functions of type $\sigma \to \tau$ can be used as if they had type $\sigma' \to \tau'$ without type error.

Subtype polymorphism is a useful feature of object-oriented programming: if subclasses correspond to subtypes, a subclass object can always be passed to a function or method expecting a superclass object, allowing reuse of code. Unfortunately, subclasses do not always generate subtypes; this can happen if the types of methods need to change in subclasses to require more specialized behavior from their arguments. In particular, since the *equal* method in *ColorPointClass* checks its argument for color as well as position, the argument type needs to change, as the example indicates. Because of the contravariance of the subtyping relation on the domain of *equal*, *ColorPoint* is not a subtype of *Point*. For the subtype relation to hold, the type of *equal* in *ColorPoint* would have to be a subtype of the type in *Point*. Thus *ColorPoint* $\to$ *bool* must be a subtype of *Point* $\to$ *bool*. But, by the subtyping principle for functions, this requires *Point* to be a subtype of *ColorPoint*, exactly the *opposite* of what we are after and clearly untrue.

This loss of subtyping in this case is not due to any problem with the definition of subtyping for functions; the procedure breakit of Figure 3 illustrates how allowing this subtyp-

[1] A few notes on our notation. Methods are functions or procedures whose body occurs after the keyword **is**. We write parameterless functions and procedures by omitting the parentheses. We write the type of parameterless functions as if they were variables of their return type. That is, we omit an implicit $unit \to$ before the result type. Methods are selected by dot notation; thus o.*m* denotes the method of name *m* defined for the object o. Commented text is preceded by "--".

[2] We presume that instance variables are not accessible from outside of the object.

ing would be unsound. When breakit is applied to an actual parameter of type *Point*, there is no problem. However if the actual parameter is a *ColorPoint*, a run-time error will occur when p.*equal*(nuPt) is evaluated. Since the value of p will be a *ColorPoint*, the code for *equal* in *ColorPointClass* will be executed. When nuPt is sent the message *c*, it will fail because it has no corresponding method. Thus, in a sound type system, a call of breakit with an actual parameter of type *ColorPoint* must not type check.

```
procedure breakit(p: Point)
   var
      nuPt: Point
   begin
      nuPt := new PointClass(3.2, 4.5)
      if p.equal(nuPt) then
      ...
   end
```

Figure 3: The procedure breakit.

Most statically-typed object-oriented languages require subclasses to generate subtypes, even in the presence of binary methods. One type requirement that has been used to this end is that the types of methods may not be changed upon inheritance; this is done, for example, in C++ [52] Object Pascal [53], and Modula-3 [46]. In such languages, one cannot write *ColorPointClass* as in Figure 2, with the typing discussed above.

Eiffel does allow argument types to be specialized in a subclass's methods; for example, it would allow *ColorPointClass* to be written as in Figure 2. We call such argument specialization *covariant argument specialization*, because it goes against the contravariant rule in argument positions. Eiffel in addition preserves the invariant that subclasses generate subtypes, but this means breakit would type-check and produce a run-time error when passed a *ColorPoint*. For Eiffel there is a proposal to compensate for the resulting insecurity in the type system by a link-time data-flow analysis of the program (called a system validity check), which would, if implemented, catch possible type errors [43]. But even if that were done, the "subtype" relation would have no clear meaning: Eiffel would claim *ColorPoint* to be a subtype of *Point*, but would not allow anything but a *Point* to be passed to breakit. So even though Eiffel would judge *ColorPoint* to be a "subtype" of *Point*, *ColorPoint* objects could not be used in all contexts where *Point* objects could be used.

The *Point*/*ColorPoint* example illustrates some but not all of the problems that arise in typing binary methods in the presence of inheritance. Further examples that illustrate additional problems will be presented in the sections below.

2.2 Privileged Access to Object Representations

A completely different kind of problem with binary operations on objects—whether they are methods or free-standing procedures—is that they must often be given privileged access to the instance variables of *both* of their arguments.

The equality methods of points and colored points are examples of the simpler case where this need does not arise—the necessary attributes of the argument are already publicly available through existing methods. In order to write the *equal* method for the point class, we only needed to compare the receiver's instance variables xValue and yValue to the values returned by the *x* and *y* methods of the argument p. There is no need to access p's instance variables directly. Indeed, p might not even have instance variables named xValue and yValue; there is no need to know anything at all about its internal representation. The situation is similar for the *equal* of the colored point class.

On the other hand, suppose we want to write a class definition for simple integer set objects with the following interface type:

$$\begin{aligned} IntSet \equiv OT\ \langle\!\langle & add\colon int \to unit;\\ & member\colon int \to bool;\\ & union\colon IntSet \to IntSet;\\ & superSetOf\colon IntSet \to bool\rangle\!\rangle \end{aligned}$$

We can easily choose a representation for integer sets, (say, as lists of integers) and implement the *add* and *member* methods as in Figure 4. But when we come to implementing the *union* and *superSetOf* methods, we get stuck: given the interface type we have chosen for sets, there is no way to find out what elements a given set contains.

```
class IntSetClass
   instance variables
      elts: IntList
   methods
      add(i: int): unit is elts := elts.cons(i)
      member(i: int): bool is return(elts.memq(i))
      union(s: IntSet): IntSet is ???
      superSetOf(s: IntSet): bool is ???
end class
```

Figure 4: The class *IntSetClass*, for which writing *superSetOf* and *union* is problematic.

The obvious thing to do is to extend the public interface of sets with an *enumerate* method that (for example) returns a list of the elements of the set. But suppose we want to use a more efficient internal representation for sets, storing the elements in a bit string. We would certainly expect not only the *add* and *member* methods to be efficient, but *superSetOf* and *union* as well. But, to achieve good performance, *union* needs to work directly with the bit string representations of the two sets, so the *enumerate* method has to be replaced by an *asBitString* method that returns the underlying representation. Doing so is unsatisfactory, because it makes representation details visible to users.

In order to handle binary operations like *equal* as methods, we need the *type* of a parameter of a method to be the same as the type of the receiving object; for methods like *superSetOf* and *union*, we need an additional mechanism for constraining the *implementation* of a parameter to be the same as the receiver's. Indeed, such a mechanism is required whether or not we want to consider *union* a proper method of set objects: an external procedure for computing the union of two set objects will also need to gain privileged access to the internal representations of both of its arguments.

3 Avoiding Binary Methods

Sometimes the simplest solution to a problem is to ignore it. In this section we explore the position that binary operations like *equal*, *union*, and $+$ should not be regarded as *methods* of either of their argument objects, thus sidestepping the thorny typing issues raised so far.

There are some theoretical benefits to taking this step. For example, aside from binary methods, the types of methods are always *positive*, in the sense that the object type itself appears only in result positions. In this case, the classic encoding of object types as recursive records [15, 25] may be replaced by an encoding where objects are modeled by existential types [48, 33].

It may also be argued that keeping binary operations separate from their arguments avoids conceptual confusion. Turning a symmetric operation like $+$ into a method gives one of its arguments an artificially special status, requiring programmers to think in terms of contorted locutions like "Ask the number a to add itself to b and send back the result," instead of the more straightforward "Compute the sum of a and b." (However, having said this, it is only fair to give the methodological counterargument: An important property of objects is their appearance as active entities that encapsulate both data and the code acting on that data. Removing binary methods from objects disrupts this property, requiring an additional layer of module structure to encapsulate the binary methods with their class. Section 5 suggests that when binary methods require privileged access to both object states, such additional encapsulation may be needed anyway.)

A final reason for avoiding binary methods is that they can exacerbate difficulties with *behavioral subtyping* of specifications. Behavioral subtyping is a stronger relationship than subtyping, and, in addition to guarantees about lack of type errors, makes behavioral guarantees [4, 5, 38, 37, 41, 39]. The degree of behavioral subtyping between specifications is limited if the specifications of supertypes are too strong to allow reasonable implementations of "behavioral subtypes." The problem is that if a subtype has extra information in its objects, then the methods of the supertype have to be carefully specified if they are to be weak enough to allow for behavioral subtyping. A weak enough specification will allow a subtype's binary methods to combine the extra information in the subtype objects: the receiver and the additional arguments. (With just unary methods, on the other hand, keeping or ignoring the extra information usually works, even without any forethought on the part of the specifier.) For example, the type *ColorPoint* has extra information, namely the color of the point. The specifications of the unary methods *x* and *y* simply ignore the point's color, which allows for behavioral subtyping. However, if one specifies the *equal* method for the type *Point* so that it returns true if and only if the *x* and *y* coordinates are equal, then behavioral subtypes cannot take such extra information into account.

With these arguments in mind, we consider in this section some alternatives to binary methods.

3.1 Using Functions Instead of Binary Methods

In languages that provide both objects and conventional procedural abstraction, an alternative to using binary methods is simply to make binary operations into functions. These binary functions can be defined outside of classes, and can be applied to pairs of arguments as usual.

```
function eqPoint(p1,p2: Point): bool is
    return( (p1.x == p2.x) && (p1.y == p2.y));

function eqColorPoint(cp1,cp2: ColorPoint):bool is
    return( (cp1.x == cp2.x) && (cp1.y == cp2.y)
          && (cp1.c == cp2.c))
```

where *Point* and *ColorPoint* no longer include the *equal* method.

Ordinarily, one advantage of using methods instead of functions is dynamic dispatch: each class can choose its own code to execute in response to a given message. Therefore, moving from binary methods to binary functions may seem a step backwards. The programmer must now know when to apply eqPoint and eqColorPoint, instead of relying on the objects themselves "knowing" which equality is appropriate. (To be fair, it is worth noting that it is difficult to achieve dynamic dispatch for binary methods such as *equal*, without adding additional methods to the classes *PointClass* and *ColorPointClass*, as in Section 4.3.)

The loss of dynamic dispatch when functions are used instead of binary methods is a serious problem. The problem manifests itself by causing code duplication which would not be needed if methods were used. When methods are used, it often occurs that the body of one method, m, invokes another method, n, on the receiving object itself. If n is overridden in a subclass, then invocations of m on objects generated by the subclass will correctly call the new version of n. However, if n happens to be a binary method, then replacing it with two (or more) binary functions results in a loss of dynamic dispatch. The loss of dynamic dispatch means that either the correct version of n will not be called from the inherited m in the subclass, or extra code must be written in the subclass (for method m) to call the proper version of n.

Figure 5 is an example that illustrates the problems caused by loss of dynamic dispatch. (Although we have no hard data on how common such examples are, this example is

```
class LinkClass
    instance variables
        value: integer
        next: MyType
    methods
        getValue: integer is return(value)
        getNext: MyType is return(next)
        setValue(n: integer): unit is value := n
        setNext(link: MyType): unit is next := link
        append(link: MyType): unit is
                        if next == nil then self.setNext(link) else next.append(link)
end class

class DoubleLinkClass subclass of LinkClass
    instance variables  -- value and next are inherited
        prev: MyType
    methods -- getValue, getNext, setValue, and append are inherited; setNext is overridden
        getPrev: MyType is return(prev)
        setNext(link: MyType): unit is next := link; link.setPrev(self)
        setPrev(link: MyType): unit is prev := link
end class
```

Figure 5: The classes *LinkClass* and *DoubleLinkClass*.

a combination of standard idioms.) *LinkClass* is a simple class of linked list objects, and *DoubleLinkClass* is a subclass that uses double links (a more complete implementation would include methods such as *reverse*, *map*, and *length*). The type *MyType* given to variables next and link in the example represents the type of objects of the current class. That is, it means *Link* in the class *LinkClass*, but means *DoubleLink* in the class *DoubleLinkClass* and in the instance variables and methods it inherits from *LinkClass*. *MyType* will be discussed in more detail in Section 4.1 below; also cf. [51, 12, 13, 29]. The objects now have only one interesting method, *append*, which is inherited by *DoubleLinkClass*. This method uses *setNext*, a binary method, to set the pointer next, and *setNext* is overridden in *DoubleLinkClass* to also properly maintain the prev link to the previous object.

In a hypothetical function encoding, the *setNext* method would be replaced by functions setNextLink and setNextDoubleLink that lie outside the class definition (ignoring for now questions of privileged access). However, since *append* invokes *setNext*, it would have to be re-written as two almost identical functions, one invoking setNextLink and the other invoking setNextDoubleLink, causing unnecessary code duplication. An in-place *reverse* method of no arguments is another method for which inheritance would suffer under this encoding. Thus dispatch can be statically resolved, but only at the cost of code duplication if this scheme is used.

3.2 Making Both Arguments into One Object

Even in a "purist" object-oriented language where every operation is treated as a message sent to some object, we may place binary operations outside of the objects on which they operate by turning the two argument objects into a single pair object and invoking the method on the pair. To see how this would work, imagine that the types *Point* and *ColorPoint* do not have any binary methods. For example, they could be:

$$\begin{aligned} \mathit{Point} &\equiv OT\ \langle\!\langle x\colon \mathit{real};\ y\colon \mathit{real}\rangle\!\rangle \\ \mathit{ColorPoint} &\equiv OT\ \langle\!\langle x\colon \mathit{real};\ y\colon \mathit{real};\ c\colon \mathit{string}\rangle\!\rangle \end{aligned}$$

With this definition, *ColorPoint* would be a subtype of *Point*.

Now define two new classes, *PointPairClass* and *ColorPointPairClass*, each with a method named *equal*, as shown in Figure 6. Note that the types of the objects generated by these classes are the same, $OT\ \langle\!\langle \mathit{equal}\colon \mathit{bool}\rangle\!\rangle$, since they have the same public interface.

So the former binary methods are now unary methods of these new classes. What would originally have been written as:

```
aCPt.equal(anotherCPt)
```

to compare two colored points, will now be written with these new classes as:

```
(new PointPairClass(aCPt, anotherCPt)).equal
```

If the types of aCPt and anotherCPt are both *ColorPoint*, then one might wish instead to compare them as colored points, in which case one would write:

```
(new ColorPointPairClass(aCPt, anotherCPt)).equal
```

There is a benefit in making these pair objects: it clarifies the perspective desired for the equality comparison. When one creates a *PointPairClass* object, it is clear what behavior is

```
class PointPairClass
   instance variables
      p1: Point
      p2: Point
   methods
      equal: bool is
         return( (p1.x==p2.x) && (p1.y==p2.y) )
end class

class ColorPointPairClass
   instance variables
      p1: ColorPoint
      p2: ColorPoint
   methods
      equal: bool is
         return((p1.x==p2.x) && (p1.y==p2.y)
                  && (p1.c==p2.c) )
end class
```

Figure 6: The classes *PointPairClass* and *ColorPointPairClass*.

expected from its *equal* method; this expectation is borne out even when the two points that make up the *PointPairClass* object are actually *ColorPoint* objects.

Because the classes generate objects of the same (interface) type, one can have a variable myPointPair that denotes objects generated by either *PointPairClass* or *ColorPointPairClass*. In this case a message send such as "myPointPair.*equal*" results in the invocation of the *equal* method defined in whichever class was used to generate the object. Thus, sending the *equal* message to a pair object gets the view with which the pair was created. This should be contrasted with the function call "eqPoint(aCPt, anotherCPt)," which always compares its two arguments as points. It can also be contrasted with a message-send of the form "aCPt.*equal*(anotherCPt)," which always uses the *equal* code of *ColorPointClass*.

This approach has problems similar to the function approach that was discussed previously—the *LinkClass* example of Figure 5 would require code duplication for inherited methods such as *append*.

4 Embracing Binary Methods

Having examined what happens if binary methods are avoided, in this section we consider the typing mechanisms that must come into play if one chooses *not* to avoid them.

Two important solutions have been proposed to the typing problems posed by binary methods. One solution, first proposed by the Abel project at HP labs [25], develops a method that partially solves the *Point*/*ColorPoint* problem by relaxing the requirement that subclasses generate subtypes. As they put it, "Inheritance is not subtyping." They did not, however, propose a concrete mechanism for realizing their ideas in an object-oriented language. In Section 4.1, we show one way this may be done using the concept of *matching* [12, 13].

The other important solution was presented in two papers at the 1991 OOPSLA conference [2, 31]. These papers deal with the static type-checking of languages with *multi-methods* (also called *generic functions* or *overloaded functions*). Multi-methods as in CLOS allow, as we show in Section 4.2.1, the *Point*/*ColorPoint* example to be typed preserving the subtyping of the two classes. But this is obtained at the expense of the encapsulation of the methods, since the generic functions, like the functions in Section 3.1, are separated from objects (objects encapsulate only data). In Section 4.2.2 we show how to reconcile multi-methods with objects encapsulating data and code [17, 45].

Closely related to the solutions of Section 4.2 is Ingalls' solution to the multiple dispatch problem [34]. He presented his solution in an untyped framework, but it can be adapted to a typed language, as Section 4.3 shows.

Lastly, we show in Section 4.4 how a general principle of giving more "precise" types to binary methods produces more flexible typings across a range of approaches, even in the case where binary operations are not treated as methods.

4.1 Matching

This section describes how a relation called "matching," which is weaker than subtyping, can replace subtyping in many situations [12]. In particular, we will see below that this generalization of subtyping provides us with the ability to handle binary methods smoothly.

4.1.1 Generalizing subtyping to matching

As seen in Section 2.1, languages that insist that subclasses generate subtypes often compensate for the resulting type problems by restricting the programmer's ability to change the types of parameters of inherited methods. This effectively eliminates the use of binary methods in these cases. If one feels that binary methods are important, then an obvious solution is to give up the identification of subclasses with subtypes. An important advantage of this decision, not discussed further here, is to separate the notion of interface (type) from that of implementation (class). In the remainder of this section we assume such a separation has been made, and thus that the notions of subtyping and matching (defined below) depend only on the interfaces of objects, not the classes generating them.

Most object-oriented languages provide a name for the receiver of a message (e.g., **self** or **this**), which can be used inside method bodies. Similarly, we use *MyType* as a keyword that denotes the *type* of the receiver [51]. It may be used in the definition of methods whose parameters or return types should be the same as that of the receiver. One can think of the object type in the following as simply another way of writing the type *Point* given in section 2.1, and it could also be the type of objects of a polar implementation of points.

$$Point \equiv OT\ \langle\!\langle x\colon real;\ y\colon real;\ equal\colon MyType \rightarrow bool \rangle\!\rangle$$

$\equiv$ *PolarPoint*

One advantage of *MyType* is that it makes it easier for human readers to compare types like *Point* and *PolarPoint.* A more important advantage is that it works well with inheritance of methods, because its meaning changes in the subclass. For example, when *MyType* is used in the definition of *ColorPointClass*, all occurrences of *MyType* in the methods automatically represent *ColorPoint* rather than *Point.*

$$\mathit{ColorPoint} \equiv OT \langle\!\langle\, x\colon \mathit{real};\ y\colon \mathit{real};\ c\colon \mathit{string};\\ \mathit{equal}\colon \mathit{MyType} \to \mathit{bool} \rangle\!\rangle$$

As before we can show the type *ColorPoint* is not a subtype of *Point.* However, there is a relationship between the types *ColorPoint* and *Point*, which is clearly apparent when looking at their types written using *MyType*. One can see that the only difference is the addition of a new method, *c*, to *ColorPoint.*

We say one object type *matches* another if the first has at least the methods of the second and the corresponding method types are the same, considering *MyType* in one to be "the same" as *MyType* in the other. We use $<\!\#$ to denote this relationship. In symbols,

$$OT\ \langle\!\langle m_1\colon \tau_1; \ldots; m_n\colon \tau_n \rangle\!\rangle <\!\#\ OT\ \langle\!\langle m_1\colon \tau_1; \ldots; m_k\colon \tau_k \rangle\!\rangle$$

holds iff $k \leq n$. (In fact, a more general definition is possible in which the types of corresponding methods of the first are all subtypes of the corresponding types of the second [12]. This means that the corresponding result types are subtypes—vary in a covariant way—while the corresponding parameter types are supertypes—vary in a contravariant way. However, this more general relation will not be needed here.)

Because the meaning of *MyType* changes in subclasses, the meanings of the types of methods in subclasses need *not* be the same as those of the corresponding methods in the superclass. However, type-safe rules for defining subclasses can ensure that the types of the objects from the subclass always match the types of the objects generated from the superclass. In order to obtain type safety, it is necessary to type check the methods of a class under the assumption that *MyType* only matches the type of objects being defined by the class. This ensures that these methods will continue to be type-safe when inherited in subclasses [12]. While some routines will not type check with this assumption, even though they would have passed under the stronger assumption that *MyType* is exactly the type of objects generated by this class, in our (admittedly not comprehensive) experience, very few routines fail. Matching tells you what messages can be sent to an object, and what their types will be. However, if *S* $<\!\#$ *T*, it does not allow the use of a parameter of type *S* where one of type *T* is expected. Nor does it allow assignment of expressions of type *S* to variables of type *T*.

As stated earlier, *ColorPoint* is not a subtype of *Point. ColorPoint* and *Point* provide an example of two types which match, but are not subtypes. Basically, if a class has a binary method, that is, a method with a parameter of type *MyType*, subclasses of that class that add new methods will not generate subtypes. On the other hand if a method's return type is *MyType*, this will not stand in the way of subtyping. Both of these follow easily from the subtyping rule for recursive types in [3], and the fact that *MyType* can be seen as an abbreviation for a recursive definition of types.

What if we want to use a *ColorPoint* as an actual parameter in a procedure or function that originally expected a *Point* parameter? Since the example of breakit in the introduction showed this could not always be done, another, more restrictive, construct is needed.

We can introduce a language feature to support a form of bounded polymorphism using matching. With this feature, functions can be specified to take type parameters whose values are restricted to "match" another type. Of course, unrestricted type parameters can also be provided, but in a large number of situations some sort of restriction is necessary.

As an example, suppose we wish to write a routine to sort an array whose elements are drawn from some ordered set. In an object-oriented language, the requirement that the elements be ordered can be modeled by demanding that they support (at least) *less_than* and *equal* methods. Define:

$$\mathit{Comparable} \equiv OT\ \langle\!\langle\, \mathit{less\_than}\colon \mathit{MyType} \to \mathit{bool},\\ \mathit{equal}\colon \mathit{MyType} \to \mathit{bool} \rangle\!\rangle$$

With this definition, the header of our polymorphic sort routine is as follows, where the notation "$T<\!\#$*Comparable*" means that the type parameter T must match the type *Comparable*:

procedure sort($T<\!\#$*Comparable*; a: Array of T);

And the function then has type[3]

sort : *All*$(T<\!\#$*Comparable*$)$ (Array of T) $\to$ *unit.*

If *PhoneEntry* is an object type supporting at least methods *less_than* and *equal* of type

$$\mathit{MyType} \to \mathit{bool},$$

and if pArray is an array of elements of type *PhoneEntry*, then sort(*PhoneEntry*, pArray) is a legal call of sort.

It is worth noting here that the type *Comparable* has no useful proper subtypes because of the appearance of *MyType* as the type of a parameter in its methods. Thus, if the bounds on type parameters were only expressed in terms of subtyping, it would be impossible to apply the sort routine to any interesting arguments.

The use of bounded matching is equivalent to the use of F-bounded polymorphism suggested in [14]. It is also very similar in effect to the restrictions on type parameters expressible in CLU and Ada (as well as the type classes of Haskell). For example, in Ada one would write the sort routine as:

[3] The notation *All*$(S\ <\!\#\ T)E(S)$ is the universally polymorphic type that can be instantiated to $E(S)$, for all S such that $S\ <\!\#\ T$.

```
generic
    type t is private;
       with function "<"(x,y: t) return BOOLEAN
                                     is <>;
       with function "="(x,y: t) return BOOLEAN
                                     is <>;
procedure sort (A: in out array (<>) of t) is ...
```

This is similar to the sort procedure written with bounded matching. Object-oriented languages containing similar constructs are Emerald [10], School [50], and Theta [40].

Returning to our example with *Point*, if f(p: *Point*) is a function accepting an argument of type *Point* then it can often be rewritten in the form f($T <\!\#$*Point*; p:T) so that it accepts a type parameter matching *Point* and an object of that type. If this type checks, then it will be possible to apply it to the type *ColorPoint* as well as an object of type *ColorPoint*. Of course this rewriting will not succeed in all cases—breakit being a prime example. The reason this transformation will not succeed in breakit is that the formal parameter nuPt will be of some type $T <\!\#$ *Point*, while p will always be of type *Point*. Thus we cannot guarantee that the type of the argument to *equal* in the body of breakit will be the same as the type of p, and the type check must fail.

What can actually be done with the information that one type matches another? The matching relation guarantees that certain messages may be sent to an object. If $T <\!\#$ *Comparable* then objects of type T can be sent messages *less_than* and *equal* (and their parameters must also be of type T). It turns out that for most situations this is all that is needed in order to ensure that the object is usable. The stronger information that a type is actually a subtype of another is generally not needed.

In particular, bounded matching can be viewed as an explicit, weakened (and hence more generally applicable) form of subtyping. If subsumption were necessary to type a function call, the code could be re-written so the function constrains the type parameter, like *Comparable* above, and function invocations explicitly pass the "smaller" type as argument. Simple subtyping is handled by the case where the type constraint contains no occurrences of *MyType*. The disadvantage of this encoding of subtyping is that all subtypings must be explicitly given in the program.

In general the use of bounded matching requires one to "plan ahead," by identifying the type parameter to be matched against. This was illustrated in the sort example: the type *Comparable* needs to be discovered by the programmer, and every use of sort requires that an explicit type parameter be passed. This is in contrast to subtyping, which is implicit and, as mentioned above, does not require any explicit type instantiation to be given in the program. (It is an interesting open problem to show how to automatically infer declarations that use matching.) If we were to decide to eliminate subtyping altogether in favor of matching, then all object subtypings would have to be recast as bounded matchings. Moreover, since we have, thus far, only defined matching for object types, we would not be able to capture the use of subtyping on other types without extending the definition of matching.

Another difficulty with relying only on matching is that it is not type-safe to perform an assignment to a variable of an object whose type only matches that of the variable. For example, imagine a framework for graphical user interfaces, in which one creates a main window as a subclass of some framework class, and has to store the window in some variable. In this case the type of the subclass objects has to be a subtype of the declared type of the variable in the framework. Subtyping seems to be required for this sort of cross-type assignment. While this can be worked around by using type parameters to designate the types of instance variables, it does limit flexibility in handling heterogeneous data structures, all of whose elements are subtypes of a given type.

The use of *MyType* is sufficient to write examples such as linked lists or trees, where methods for attaching a node to another or returning an adjoining node must be binary methods. The types of the instance variables of these nodes also can be written in terms of *MyType*. If the definition of singly-linked node is written using *MyType* in this way, it is easy to define a doubly-linked node as a subclass of singly-linked node. Figure 5 presents such an example. As expected, the type of a doubly-linked node is not a subtype of singly-linked node, but it does match. It is then relatively easy to write an implementation for lists which takes a type parameter which matches singly-linked node. By applying this to either the type for singly-linked node or doubly-linked node, the corresponding kind of list can be generated without code duplication. (See [13] for the details of this parameterized example.)

The use of *MyType* in class definitions makes it easier to write useful subclasses in statically typed object-oriented languages, especially when the superclasses contain binary methods. As illustrated in the sorting example above, the matching relation is very useful in defining bounded polymorphic functions. In fact, the use of these two features should provide a type-safe replacement for the (unsafe) uses of covariant argument specialization typing in languages like Eiffel or O_2 [8], while providing comparable expressiveness. The object-oriented language LOOP [29], on the other hand, has no matching relation *per se*, but has similar expressivity, achieved by circular subtype assertions $\tau <: \sigma$ where τ and σ may share free type variable X; this can be viewed as a form of operator subtyping $\tau(X) <: \sigma(X)$. The introduction of a matching relation is thus one, but not the only, solution to the problem of typing inherited binary methods.

In the next subsection we explore the mathematical aspects of the matching relation.

4.1.2 Matching and Object Types

As described in [1], matching can consistently be defined in terms of pointwise subtyping on operators from types to types. In this case an object type is used to define a function from types to types by replacing all occurrences of *MyType* by a type variable. For example, *Point*, can be used to define:

$$\textit{PointOperator} \equiv \lambda P : \text{Type}.\ \langle\!\langle\, x\!: \textit{real};\ y\!: \textit{real};\ \textit{equal}\!: P \rightarrow \textit{bool} \,\rangle\!\rangle$$

The original *Point* can be recovered by taking the fixed point of the operator[4]:

$$Point \quad \equiv \quad \mathrm{Fix}(PointOperator)$$

Details can be found in the paper cited above.

While understanding objects types as fixed points in this way is intuitively appealing, the ability to unfold recursive types does not interact well with the definition of matching as essentially a relation on these operators (rather than the fixed points themselves).

For example, look at the relationship between the following types:

$$\begin{aligned} EPoint &\equiv OT\ \langle\!\langle x\colon real;\ y\colon real; \\ &\qquad equal\colon EPoint \to bool\rangle\!\rangle \\ Point &\equiv OT\ \langle\!\langle x\colon real;\ y\colon real; \\ &\qquad equal\colon MyType \to bool\rangle\!\rangle \\ ColorPoint &\equiv OT\ \langle\!\langle x\colon real;\ y\colon real; c\colon string; \\ &\qquad equal\colon MyType \to bool\rangle\!\rangle \end{aligned}$$

Understanding object types as recursive records, the first two would seem to be the same type. However, while *ColorPoint* matches *Point* according to our definition of matching, *ColorPoint* does not match *EPoint*. Thus these two seemingly identical types must be treated as being distinct. It is worth noting that there is some justification to treating the two types as distinct, as the *equal* method of a class which generates objects of type *Point* is type checked under weaker assumptions on the parameter (i.e., it has type *MyType*, which is only assumed to match *Point*) than the corresponding method of *EPoint*, in which the parameter has type *EPoint*.

This anomaly suggests that an encoding of object types in terms of something a bit weaker than fixed points might be necessary. It is an interesting open problem to find such an encoding of object types (including *MyType*) and a corresponding semantic definition of matching.

4.2 Multi-methods

A different solution whereby binary methods can be embraced is to use multi-methods. Contrary to matching, this solution does not introduce a new relation on types, since with multi-methods one can have both type safety and subtyping relations such as *ColorPoint* $<:$ *Point*.

A *multi-method* is a collection of method bodies associated with one message name. The selection of which method body to execute depends on the classes of one or more of the parameters of the method (rather than just on the class of the receiver as in ordinary object-oriented languages).

In this survey we distinguish two different kinds of multi-methods: the ones used by the language CLOS [27], and the *encapsulated multi-methods* of [17, 45]. A unified analysis of both kinds of multi-methods is given in [17]. We now describe each kind in turn.

4.2.1 Multi-methods à la CLOS

Intuitively the idea is to consider (multi-)methods (in CLOS jargon, *generic functions*) as global functions that are dynamically bound to different method bodies according to the classes of the actual arguments. An object does not encapsulate its methods, just the data (its instance variables). There no longer exists the notion of a privileged receiver for a method (the one that encapsulates it, usually denoted by **self** or **this**) since a multi-method is applied to several arguments that equally participate in the selection of the body. In this case we talk of "multiple dispatching" languages, in antithesis to "single dispatching" ones where a privileged receiver is used. A class of objects is then characterized just by the internal variables of its instances. For example, in a typed multi-method-based language, the classes given in Figures 1 and 2 would be defined as in Figure 7.

In order to stress the difference with the formalisms presented so far we have used a different syntax. Thus a class declares only its subclassing relation and the internal representation of its instances (the **includes** keyword), while method definitions (introduced by the keyword **method**) appear outside the class declarations. In order to simplify the exposition in this section, we identify classes and types, in the sense that the name of a class (for which we no longer use the suffix *Class*) is used as the type of its instances; therefore in this section (and in this section only) `p` : *Point* will *also* mean "`p` is an instance of class *Point*." Thus, when discussing multi-methods à la CLOS, we write class names where types would otherwise appear.[5] This allows one to consider multi-methods as overloaded functions, whose actual code is dynamically selected according to the type (i.e., the class) of the arguments they are applied to.

The definitions of the methods in Figure 7 are completely disconnected from those of classes. There are two distinct definitions for *equal*, one for arguments of types *Point*×*Point* and the other for arguments of type *ColorPoint*×*ColorPoint*. We say that the message *equal* denotes a multi-method (or a generic function, or an overloaded function) formed by two *branches* (or method bodies). The type of a multi-method is the set of the types of its branches; thus *equal* has type:

$$\{Point \times Point \to bool, ColorPoint \times ColorPoint \to bool\}$$

When *equal* is applied to a pair of arguments, the system executes the branch defined for those parameters whose type "best matches" the type of the arguments. For example if *equal* is applied to two arguments in which at least one of them is of type *Point* and the other is a subtype of it, then the first definition of *equal* is executed; if both arguments have as type a subtype of *ColorPoint* then the second definition is

[4] The notation $\mathrm{Fix}(F)$ means the least fixpoint of F.

[5] If we were to distinguish between types and classes (i.e. between interfaces and implementations: cf 4.1.1) , then a new notation would be needed to specify both a class and a type parameter for multi-methods. One possibility is to use the notation of Cecil [20, 22], which does separate these concepts.

```
class Point
   includes
      xValue: real
      yValue: real
end class

class ColorPoint subclass of  Point
   includes                         -- xValue and yValue are inherited
      cValue : string
end class

method x(p: Point):real is return(p.xValue)

method y(p: Point):real is return(p.yValue)

method c(p: ColorPoint):real is return(p.cValue)

method equal(p:Point, q:Point):bool is return( (x(p)==x(q)) && (y(p)==y(q)) )

method equal(p:ColorPoint, q:ColorPoint): bool is return((c(p)==c(q)) && (x(p)==x(q)) && (y(p)==y(q)) )
```

Figure 7: *Point* and *ColorPoint* classes defined using multi-methods à la CLOS.

selected. More generally, when a multi-method of type

$$\{S_1 \to T_1, S_2 \to T_2, \ldots, S_n \to T_n\}$$

is applied to an argument of type S, the system executes the body defined for the parameter of type $S_j = \min_{i=1..n}\{S_i \mid S <: S_i\}$. This selection is performed at run-time. In this way one obtains dynamic dispatch. Note that in this paradigm binary methods are really binary, since the implicit argument given by the receiver of the message is, in this case, explicit.

In [18] it is proved that to have a sound type system it suffices that every multi-method of type $\{S_1 \to T_1, S_2 \to T_2, \ldots, S_n \to T_n\}$ satisfies the following condition.[6]

$$\forall i, j \in [1..n] \qquad \text{if } S_i <: S_j \text{ then } T_i <: T_j \tag{1}$$

(This is similar to the monotonicity condition of [49, 42].) Note that all the multi-methods defined in Figure 7 (and in particular *equal*) satisfy this condition. Therefore *ColorPoint* $<:$ *Point* does not cause type insecurities.

Intuitively, the idea underlying the multi-method approach is that binary methods may be applied to arguments of different types and that, in general, it is not possible to choose the code to execute according to the type of just one argument. To determine which method body must be executed one needs to know the types of all the arguments of the method. In single dispatching the branch selection is based only on one argument—the receiver; therefore combining subtyping and binary methods with heterogeneous arguments is not type-safe. In contrast, using a multi-method we can refine the selection by considering all the arguments. Thus it need never happen that the argument of a method has a supertype of the type of the corresponding parameter (as in the case of breakit). It is important to stress that this constitutes an approach completely different from matching, where the arguments of a binary method are statically forced to have the same type.

Note also that multi-methods allow one to specialize *equal* in a different way for each possible combination of arguments. It suffices to add the branches for the remaining cases:

```
method equal(p: Point, q: ColorPoint): bool is ...

method equal(p: ColorPoint, q: Point): bool is ...
```

As we have seen, CLOS's multi-methods induce an object-oriented style of programming that is rather different from the one of traditional single dispatching object-oriented languages. Most of the languages that use multi-methods are untyped (e.g. CLOS [27], Dylan [7], which use classes instead of types to drive the selection of multi-methods). The only strongly-typed languages in our ken that use multi-methods are Cecil [22], and Polyglot [2].

The lack of encapsulation in multi-methods is both an advantage and a drawback. The drawback is methodological: an object (or a class of objects) is no longer associated with a fixed set of methods that have privileged access to its internal representation. The usual rule is that any method with a formal parameter of a given class can access the instance variables of the actual parameter object. The advantage is that this solves the privileged access problem described in Section 2.2, because a binary method can gain privileged access to both its arguments. However, because such methods can be defined anywhere in the program, one cannot restrict direct access to instance variables to a small area of the program text. One way to fix such problems may be to add a

[6] Some further conditions are required to assure that a best matching branch always exists for the selection (see [2], [22], and [18]).

```
class ColorPointClass subclass of PointClass
    instance variables                    -- xValue and yValue are inherited
        cValue : string
    methods
        c:string is return(cValue)
        equal(p: Point):bool is return( (xValue==p.x) && (yValue==p.y) )
        equal(p: ColorPoint):bool is return( (cValue==p.c) && (xValue==p.x) && (yValue==p.y) )
end class
```

Figure 8: The class *ColorPointClass* written using encapsulated multi-methods.

separate module system to control instance variable access [22]. Instead of pursuing that idea, in the next subsection, we show how to apply the ideas of multi-methods in more traditional object-oriented languages with single dispatching and classes.

Conventional wisdom is that multiple dispatch is more expensive than single dispatch. In a single dispatch language, a single table lookup can find the best argument branch. With multiple dispatch, it may be more expensive to compute the branch of a multi-method that matches the arguments best, although various techniques have been designed to minimize the added expense [6, 23, 26]. However, in a language where the compiler can tell which argument positions need dispatching (as in CLOS), one can implement multi-method dispatch as a chain of single dispatches [36]. If this is done, then there is no extra cost for multiple dispatch in programs that do not use it; that is, in a multiple-dispatching language, programs that only use single dispatch have the same cost as in a single-dispatching language. Moreover, if a program in a single-dispatching language is written by using additional dispatching after methods are called to resolve problems caused by binary methods (as in Section 4.3), then such a program will be no faster than the equivalent multi-method program [21].

A final drawback of multi-methods à la CLOS is the difficulty of combining independently developed systems of multi-methods [24]. While other ways to solve this problem have been studied [22], the problem nearly disappears when multi-methods are combined with single dispatching, as described next.

4.2.2 Encapsulated multi-methods

To solve the encapsulation problems of multi-methods à la CLOS, we seek to emulate the Smalltalk model, where every method is the method of one object. Thus each method has a privileged receiver argument (**self**), which is the only argument whose internal state can be accessed by the method. Instead of defining multi-methods as global functions, the idea is to use them to define the bodies of some methods in a class definition [17]. In this way a multi-method is always associated to a message *m* of a class *C*. When *m* is sent to an object of class *C*, it is dispatched to the corresponding method. If this method happens to be a multi-method, then the branch is selected according to the types of the further arguments of *m*. Thus, the selection of the method is still based on the receiver, but the actual code is selected among several bodies that are encapsulated inside the object. Inside these bodies, the receiver is still denoted by the keyword **self**. Encapsulated multi-methods are to be distinguished from static overloading (as found in Ada, Haskell, C++, and other languages), because the selection of code must be made dynamically.

As an example of this technique, take the class *Point* as in Figure 1 and rewrite the class *ColorPoint* as in Figure 8. In that Figure (note we are using our original notation for classes again) there are two definitions for *equal*: the first is executed when the argument of *equal* is of type *Point*, the other when it is of type *ColorPoint*. The selection of the appropriate definition is done at run-time when the argument of *equal* has been fully evaluated and hence its run-time type is apparent. The selection is based on the type of the additional argument. In other words, we have transformed the method associated to *equal* into a multi-method, where arguments of different types are associated to different codes.

There are two differences from multi-methods à la CLOS. The first is that multi-methods are defined in particular classes, whereas in CLOS they are globally defined generic function names. This solves the encapsulation problems of CLOS multi-methods, because access to instance variables is restricted to the methods of a class, as only the receiver's instance variables can be accessed. The second difference is that dispatch is not based on actual argument classes, but rather on argument types. This is possible because no privileged access is obtained to the additional arguments. However since types are not equated with classes, the technique cannot solve the problem of privileged access to other arguments discussed in Section 2.2: several different classes might implement the same type, so from the type alone it is unclear which class implementation the method should have access to.

The type of a multi-method is the set of the types of its different codes. Thus the type of an instance of *ColorPointClass* now becomes

$$\begin{aligned} &\mathit{ColorPoint} \equiv \\ &\quad OT\ \langle\!\langle x\colon \mathit{real};\ y\colon \mathit{real};\ c\colon \mathit{string}; \\ &\qquad\quad \mathit{equal}\colon \{\mathit{Point} \to \mathit{bool}, \mathit{ColorPoint} \to \mathit{bool}\}\rangle\!\rangle \end{aligned}$$

and *ColorPoint* $<:$ *Point* holds, since, for subtyping, ordinary methods are considered as multi-methods with just one branch (their type is a singleton set) and in a type system for multi-methods (see [17]) one can deduce: $\{Point \to bool, ColorPoint \to bool\} <: \{Point \to bool\}$.

More precisely, the subtyping relation between sets of types states that one set of types is smaller than another if and only if for every type contained in the latter there exists a type in the former smaller than it. This fits the intuition that one multi-method can be replaced by another multi-method of different type when for every branch that can be selected in the former there is one branch in the latter that can replace it.

Thus, if in writing a subclass one wants the type of the instances to be a subtype of the type of the instances of the superclass, then some care in *overriding* binary methods is required. Indeed, the rule of thumb for this approach is that to override a binary method one must use an (encapsulated) multi-method with (at least) two branches: one with a parameter whose type is the type of the instances of the class being defined, the other with a parameter whose type is the type of the instances of the original superclass in which the message associated with the binary method has been first defined. Thus, when a binary method is overridden in a new class, it is not enough to specify what the new method has to do with the objects of the new class. It is also necessary to specify what it has to do when the argument is an object of a superclass. Fortunately, this does not require a large amount of extra programming. The number of branches that suffice to override a binary (or n-ary) method in a type-safe manner is independent of both the size and the depth of the inheritance hierarchy; indeed, it is always equal to two. For example, suppose that we further specialize our *Point* hierarchy by adding further dimensions:

```
class 3DPointClass subclass of PointClass
    instance variables x3Value: real
    methods ...

class 4DPointClass subclass of 3DPointClass
    instance variables x4Value: real
    methods ...
```

…and so on, up to a dimension n. The new classes form a chain in the inheritance hierarchy. If we want to override *equal*, what do we have to do in order for this to be a chain of the subtyping hierarchy too (i.e., $nDPoint <: \ldots <: 4DPoint <: 3DPoint <: Point$)? If we want to override *equal* in *nDPointClass* (thus we want that in the description of *nDPointClass* a definition of the form *equal*(p : *nDPoint*)is ... appears), then the first idea is to write for the class *nDPointClass* a multi-method with $n-1$ branches, one for each class in the chain[7]. This is possible, but for type safety a multi-method with two branches is enough: one for arguments of type *nDPoint*, which is the one we want to define, and the other for arguments of type *Point*, which will handle all the arguments of a supertype of *nDPoint*. For example, in case of $n = 4$ one could define[8]

```
class 4DPointClass subclass of 3DPointClass
    instance variables x4Value: real
    methods
      x4:real is return(x4Value)
      equal(p: Point):bool is return(p.equal(self) )
      equal(p: 4DPoint):bool is
           return((xValue==p.x) && (yValue==p.y) &&
                      (x3Value==p.x3) && (x4Value==p.x4) )
end class
```

Type safety stems from the fact that the subtyping condition is satisfied.[9]

A final remark is in order. The different branches that compose a single multi-method are not required to return the same type. For type safety it suffices to have the condition (1) as for multi-methods à la CLOS: for each pair of multi-method branches c_1, c_2 with the same name and number of arguments,[10] if the parameter types of c_1 are smaller than the corresponding parameter types of c_2, then the result type of c_1 must be smaller than the result type of c_2 [49, 18].

Note that multi-methods can be considered as a kind of **typecase** construct enhanced by two features: (a) the selection of the case to apply uses subtyping instead of type equality; (b) all the cases are not required to return the same type (they are solely required to satisfy the condition (1)). This makes multi-methods more flexible than statically defined **typecase** statements as might be found in imperative languages: without (a), a binary method whose parameter type is guarded using a **typecase** would always have to be rewritten when new subclasses are added to the program; without (b), specialization of the result type of binary methods could not be handled. The only remaining problem that multi-methods and **typecase** share is that if the method should only be defined with a parameter of *exactly* the same type as the receiver, the multi-method user will be required to add a new method body with the original parameter type whose only purpose is to raise an error message. See the conclusion for further discussion of this issue.

Some further consistency conditions are required in case of multiple inheritance [31, 45, 18, 22].

One of the main advantages of this approach is that the extra branch required to assure type safety of subtyping can

[7] Of course, if in the definition of *nDPointClass* we do not give any definition for *equal* then *nDPointClass* inherits the last (multi-)method defined for *equal* in the upper hierarchy. It is important to be clear that, in the formalization we use, a new definition of a (multi-)method completely overrides the old one (i.e. it is not possible to inherit some branches and override others: this could by obtained by adding some extra syntax.)

[8] This example is due to John Boyland.

[9] In general, if we have a hierarchy of n classes whose instances have type $S_n <: \ldots <: S_1$ and we want to define for each of them a binary method, respectively returning the type $T_n <: \ldots <: T_1$ then according to the subtyping rule for multi-methods we have the following type inequalities: $\{S_n \to T_n, S_1 \to T_{n-1}\} <: \ldots <: \{S_{i+1} \to T_{i+1}, S_1 \to T_i\} <: \{S_i \to T_i, S_1 \to T_{i-1}\} <: \ldots <: \{S_1 \to T_1\}$. This proves that two branches always suffice for binary methods. The declarations of the classes for points are a special case of this, where S_1 =*Point*, for $i \in [3..n]$ S_{i-1} =*iDPoint*, and for $i \in [1..n$-1] T_i =*bool*.

[10] Indeed multi-methods may have more than one parameter (this allows us to deal with n-ary methods), and the multi-method branches are not all required to have the same number of parameters.

```
class PointClass
        . . .
    methods
        . . .
        equal(p: Point): bool is return( p.equalPoint(self) )
        equalPoint(p: Point): bool is return( (xValue==p.x) && (yValue==p.y) )
        equalColorPoint(p: ColorPoint): bool is return( self.equalPoint(p) )

class ColorPointClass subclass of  PointClass
        . . .
    methods
        equal(p: Point): bool is return( p.equalColorPoint(self) )
          -- equalPoint is inherited
        equalColorPoint(p: ColorPoint): bool is return( (cValue==p.c) && (xValue==p.x) && (yValue==p.y) )
```

Figure 9: Ingalls' simulation of multi-methods.

be generated in an automatic way. Therefore this technique can be embedded directly in the technology of the compiler, and used to "patch" the already existing code of languages that use covariant specialization, like Eiffel and O_2. Thus, like the solution given in the next section, this solution can be directly applied to languages with covariant argument specialization without requiring any modification of the code: a recompilation of existing code will suffice (see [11]).

On the other hand this approach has some disadvantages. One disadvantage compared to multi-methods à la CLOS is that it does not solve the problem of obtaining privileged access to other arguments in a binary method. Another disadvantage of this approach is that in case of multiple inheritance additional type checking constraints are needed. The problem is that when multiple inheritance is used, the notion of a "best matching branch" to select or to inherit may be lost. Consequently, unconstrained use of multi-methods can break the modularity of programming [24], since the addition of a new class to the system might require the addition of some new code in a different class to assure the existence of the best branch (see, for example, [22]). However the problem with modularity is less critical than in the case of multi-method à la CLOS. An additional disadvantage is again the performance penalty imposed by multi-methods. One extra test and branch is required to decide which code is to be executed. The overhead to resolve uses of encapsulated multi-methods is however smaller than in the case of CLOS multi-methods since there is no special lookup needed for the privileged receiver.

There are also some less important disadvantages. The first one is that, as it depends on an *avant garde* type theory, the interactions of this theory with fairly standard features like polymorphism (both implicit and explicit) are not yet clear. (Models based on records have been more deeply studied than those based on overloading.) Also, even though there is not a blowup of the number of extra method bodies that must be written, there is at least a doubling of the number of method bodies that must be written each time a binary method is overridden. Some further negative remarks are to be found at the end of the next section.

4.3 Simulating Multi-methods in a Single-Dispatching Language

Ingalls offered a solution to what he called the problem of "multiple polymorphism" at the first OOPSLA conference [34]. His solution to the binary method problem, offered in the context of single-dispatching languages such as Smalltalk-80 [32], was to use two message dispatches, one to resolve the polymorphism of each argument.

In the example of points, colored points, and equality, the *equal* method would be coded as in Figure 9. As usual, the class *ColorPointClass* inherits the method *equalPoint* from the class *PointClass*. Now the (mutually recursive) types of the instances of *PointClass* and *ColorPointClass* are:

$$\begin{aligned} \mathit{Point} \equiv OT\ \langle\!\langle & x\colon \mathit{real};\ y\colon \mathit{real}; \\ & \mathit{equal}\colon \mathit{Point} \rightarrow \mathit{bool}; \\ & \mathit{equalPoint}\colon \mathit{Point} \rightarrow \mathit{bool}; \\ & \mathit{equalColorPoint}\colon \mathit{ColorPoint} \rightarrow \mathit{bool}\rangle\!\rangle \end{aligned}$$

$$\begin{aligned} \mathit{ColorPoint} \equiv OT\ \langle\!\langle & x\colon \mathit{real};\ y\colon \mathit{real};\ c\colon \mathit{string}; \\ & \mathit{equal}\colon \mathit{Point} \rightarrow \mathit{bool}; \\ & \mathit{equalPoint}\colon \mathit{Point} \rightarrow \mathit{bool}; \\ & \mathit{equalColorPoint}\colon \mathit{ColorPoint} \rightarrow \mathit{bool}\rangle\!\rangle \end{aligned}$$

Notice that, with this typing, *ColorPoint* is a subtype of *Point*. Also, *equal* in *ColorPoint* is a binary method, since by subsumption it can have argument type *ColorPoint* as well. This typing can be said to be more *precise* than the typing of *ColorPoint* given in the introduction; the general issue of the use of more precise typings is taken up in Section 4.4.

The solution offered by Ingalls is probably the best-known way to simulate multiple dispatch in a language with only single dispatch. With respect to true multiple-dispatch, the Ingalls simulation is more exact than the function simulation offered in Section 3.1, since it can arrange for *equal* with two arguments whose dynamic type is *ColorPoint* to always take color into account, regardless of the static types of the argument expressions. This is because of the second dynamic dispatch in the *equal* method. Such a result is not possible with the function simulation of Section 3.1: one will always be able to apply the eqPoint function to two *ColorPoint* objects and lose exact type information. This example is thus one case for which dynamic dispatch on binary methods occurs. In this respect, the simulation of multiple-dispatch with external functions is less faithful and flexible than the Ingalls simulation.

This translation can also be contrasted with encapsulated multi-methods as described in Section 4.2.2. Ingalls' translation lacks modularity in that it requires *equalColorPoint* to be added to the *PointClass* class when *ColorPointClass* is defined. With multi-methods, modularity can be preserved since the redefinition of the equal method inside *ColorPointClass* does not require any modification of the code for *PointClass*; however, this introduces an unnatural asymmetry, since the redefinition of *equal* requires one to write code for how a *ColorPoint* behaves when its *equal* method is passed a *Point*, but not vice-versa. The natural symmetry cannot be restored except by breaking the modularity of the multi-method solution.

It should be pointed out that the above argument only holds if we require (multi-)methods to be written in classes, as in Section 4.2.2. For multi-methods à la CLOS there is no problem of asymmetry, although there is still a modularity problem. However, the multi-method approach still requires one to go back and add code for types that appeared to have been completed earlier.

Ingalls' solution is surprisingly general—by overriding *equalPoint* in *ColorPointClass*, a different method can be executed for all four combinations of *Point* and *ColorPoint*. Ingalls' solution could in fact be used as one technique for implementing encapsulated multi-methods in a compiler, provided the compiler had access to all of the code at compilation time.

Finally, for large inheritance hierarchies the number of cases required by Ingalls' solution can, in principle, become quite cumbersome.

4.4 Precise Typings

It is sometimes advantageous to use more precise typings for methods. A binary method only needs its argument to have the methods that are explicitly used. Generally this is a weaker requirement than having the argument be an object of the current class, and it may allow for a "larger" (with respect to the $<:$ relation) type of the argument of this method; by the contravariant subtyping rule for functions this produces a smaller type for the method. The informal idea is thus to give methods smaller types [9, 10]. By subsumption, these types can always be lifted to "true binary" form, allowing objects of the same class to be passed as arguments to the method. Thus, specifying a smaller type of a method can only increase its usability.

Ingalls' solution in Section 4.3 in fact depends on the use of precise types, for the key to its typability is the use of *Point* for the type of the argument of *equal* in *ColorPointClass*. This gives the method a smaller type than if the argument were of type *ColorPoint*. In this section we elaborate on this technique.

By way of illustration, consider the original *Point*/*ColorPoint* example of Figures 1 and 2. Since neither *equal* method calls *equal* recursively, the types

$$
\begin{aligned}
&\mathit{Point}_{min} \equiv \\
&\quad OT\ \langle\!\langle x\colon \mathit{real};\ y\colon \mathit{real}; \\
&\qquad\quad \mathit{equal}\colon OT\ \langle\!\langle x\colon \mathit{real};\ y\colon \mathit{real}\rangle\!\rangle \to \mathit{bool}\rangle\!\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ColorPoint}_{min} \equiv \\
&\quad OT\ \langle\!\langle x\colon \mathit{real};\ y\colon \mathit{real};\ c\colon \mathit{string}; \\
&\qquad\quad \mathit{equal}\colon OT\ \langle\!\langle x\colon \mathit{real};\ y\colon \mathit{real};\ c\colon \mathit{string}\rangle\!\rangle \to \mathit{bool}\rangle\!\rangle
\end{aligned}
$$

may also be given. These types are subtypes of the types given originally. Note that the objects passed to *equal* themselves require no *equal* method be present. Since Point_{min} is a subtype of $OT\ \langle\!\langle x\colon \mathit{real};\ y\colon \mathit{real}\rangle\!\rangle$ and similarly for $\mathit{ColorPoint}_{min}$, it is easy to see that

$$
\begin{aligned}
\mathit{Point}_{min}\ <:\ OT\ \langle\!\langle & x\colon \mathit{real};\ y\colon \mathit{real}; \\
& \mathit{equal}\colon \mathit{Point}_{min} \to \mathit{bool}\rangle\!\rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathit{ColorPoint}_{min}\ <:\ OT\ \langle\!\langle & x\colon \mathit{real};\ y\colon \mathit{real};\ c\colon \mathit{string}; \\
& \mathit{equal}\colon \mathit{ColorPoint}_{min} \to \mathit{bool}\rangle\!\rangle
\end{aligned}
$$

so *equal* is indeed a binary method, and no typings are lost in this approach. In fact, something is gained over the matching interpretation described in Section 4.1: it is possible to invoke the *equal* method of a $\mathit{Point}_{(min)}$ with a $\mathit{ColorPoint}_{(min)}$ as argument. Typing this "heterogeneous" invocation is crucial for a class defining binary methods intended to be inherited without redefinition and able to take as arguments objects of any subclass [28]. In a type system based on matching, a method declared to take arguments of type *MyType* cannot, in general, accept an object of a subclass as argument; it is necessary to use bounded matching to realize this (see the discussion at the end of Section 4.1.1). Precise types here provide a simpler solution based on subtyping. Note that $\mathit{ColorPoint}_{min}$ does not match, nor is it a subtype of, Point_{min}.

As shown in Section 4.3, typing Ingalls' solution when *MyType* appears only in the types of method parameters is possible simply by using subsumption, *e.g.*, to lift a *ColorPoint* to a *Point* in cp1.*equal*(cp2), where both cp1 and cp2 are objects of type *ColorPoint*. However this technique cannot be directly applied to binary methods with result of type *MyType* (or involving *MyType*), because subsumption on the type of the argument may cause loss of interesting type information. Consider the example in Figure 10, which defines points and colored points with a binary *max* method. Objects of *MPointClass* and *ColorMPointClass* could be given the following types, which are simple modifications of the types of *Point*s in Section 4.3.

$$
\begin{aligned}
&\mathit{MPoint} \equiv \\
&\quad OT\ \langle\!\langle x\colon \mathit{real};\ y\colon \mathit{real}; \\
&\qquad\quad \mathit{max}\colon \mathit{MPoint} \to \mathit{MPoint}; \\
&\qquad\quad \mathit{maxMPoint}\colon \mathit{MPoint} \to \mathit{MPoint};
\end{aligned}
$$

```
class MPointClass
        . . .
    methods
        . . .
        max(p: MPoint): MPoint is return( p.maxMPoint(self) )
        maxMPoint(p: MPoint): MPoint is
                    if xValue **2 + yValue **2 < p.x**2+ p.y**2
                    then return(p) else return(self)
        maxColorMPoint(p: ColorMPoint): MPoint is return( self.maxMPoint(p) )

class ColorMPointClass subclass of  MPointClass
        . . .
    methods
        max(p: MPoint): MPoint is return( p.maxColorMPoint(self) )
         -- maxMPoint is inherited
        maxColorMPoint(p: ColorMPoint): ColorMPoint is
                    if  (xValue **2 + yValue **2) * brightness(cValue)
                        < (p.x**2+ p.y**2) * brightness(p.c)
                    then return(p) else return(self)
```

Figure 10: Ingalls simulation of points with a *max* method: first attempt.

$$\quad\quad maxColorMPoint\colon ColorMPoint \rightarrow MPoint\rangle\rangle$$

$$\begin{aligned}&ColorMPoint \equiv\\ &\quad OT\ \langle\langle x\colon real;\ y\colon real;\ c\colon string;\\ &\quad\quad max\colon MPoint \rightarrow MPoint;\\ &\quad\quad maxMPoint\colon MPoint \rightarrow MPoint;\\ &\quad\quad maxColorMPoint\colon ColorMPoint \rightarrow ColorMPoint\rangle\rangle\end{aligned}$$

The subtyping *ColorMPoint* $<:$ *MPoint* still holds, but note that the result of method *max* of *ColorMPointClass* is of type *MPoint*; type checking would fail if we assigned this method the type *MPoint* $\rightarrow$ *ColorMPoint*. Thus the static type of taking the *max* of two *ColorMPoint*s will have to be merely *MPoint* (unless the method *maxColorMPoint* was used explicitly). True multi-methods do not suffer from this shortcoming.

We can overcome this problem in a more expressive type system that provides for polymorphism in addition to recursive types. The idea is to make the type of *max* more precise, and in this case, polymorphic. The code for the *max* methods with their new type annotations is given in Figure 11. This modification yields types for the objects of *MPointClass* and *ColorMPointClass* described in Figure 12. If p is a *MPoint*, its *max* method can still be specialized to a binary method: p.*max*[*MPoint*] is of type *MPoint* $\rightarrow$ *MPoint*, and similarly for the *max* method of a *ColorMPoint*. The relation with the types of the "true binary" methods is more direct in an implicitly typed language, where the precise types are smaller [30, 28].

With this typing, taking the *max* of two elements of *ColorMPoint* returns a *ColorMPoint*; any other combination returns a *MPoint*, the best static type possible. Note that *ColorMPoint* is still a subtype of *MPoint* in a system with implicit unfolding of recursive types. So, this typing has all the desired properties of the typing via pure multi-methods of Section 4.2, giving more situations in which Ingalls' method may be usefully applied.

SOOP and PolyTOIL are two languages in which all of the precise typings of this section may be expressed. Precise types are complex, however, and it is difficult to imagine programmers writing them routinely. A solution to this problem is to automatically infer *minimal* types. See [28] for a type inference algorithm for the I-LOOP object-oriented language. The algorithm infers a form of F-bounded polymorphic type for classes and objects. It infers minimal types for the original *Point*/*ColorPoint* example that are very similar to the "small" types presented above. The types inferred for objects of *MPointClass* and *ColorMPointClass* are slightly more general than the form above.

To summarize some of the advantages and disadvantages of precise typing:

+ Precise types allow more flexibility in typing than matching alone. They may be expressed using bounded matching, but bounded matching requires explicit quantification and instantiation where subtyping alone may suffice.

+ Precise types are a critical component of a typed version of Ingalls' solution.

+ More precise types in module interfaces can be used to overcome some of the limitations of matching.

- The generally more complicated form of the precise types suggests that a type inference algorithm may be the only practical alternative.

- In defining a subclass, one may have to go back and modify the type annotations of the superclass (and, in general, the superclass of the superclass, etc.) in order

```
class MPointClass
   . . .
   max[X:Type](p: $OT \langle\langle maxMPoint\colon MPoint \to X;\ maxColorMPoint\colon ColorMPoint \to X\rangle\rangle$):X is
      return( p.maxMPoint(self) )
   . . .
class ColorMPointClass subclass of MPointClass
   . . .
   max[X: Type](p: $OT \langle\langle maxColorMPoint\colon ColorMPoint \to X\rangle\rangle$): X is
      return( p.maxColorMPoint(self) )
   . . .
```

Figure 11: Ingalls simulation of points with a *max* method: precise typing.

```
MPoint ≡
  $OT \langle\langle x\colon real;\ y\colon real;$
        $max\colon All(X)\ OT \langle\langle maxMPoint\colon MPoint \to X; maxColorMPoint\colon ColorMPoint \to X\rangle\rangle \to X;$
        $maxMPoint\colon MPoint \to MPoint;$
        $maxColorMPoint\colon ColorMPoint \to MPoint\rangle\rangle$

ColorMPoint ≡
  $OT \langle\langle x\colon real;\ y\colon real;\ c\colon string;$
        $max\colon All(X)\ OT \langle\langle maxColorMPoint\colon ColorMPoint \to X\rangle\rangle \to X;$
        $maxMPoint\colon MPoint \to MPoint;$
        $maxColorMPoint\colon ColorMPoint \to ColorMPoint\rangle\rangle$
```

Figure 12: Types for object of *MPointClass* and *ColorMPointClass*

to generate subtypes. This may be seen as another argument in favor of type inference, since no modifications will be required in an implicitly typed language.

5 Privileged Access to Object Representations

In Section 2.2, we saw that the problems of typing binary methods are often accompanied by difficulties in implementing binary operations without exposing object internals to public view. This section sketches a technique whereby such "overexposed objects" can be wrapped in an additional layer of abstraction, creating a limited scope in which their internal structure is visible. The technique was developed by Pierce and Turner [47] and by Katiyar, Luckham, and Mitchell [35]; we refer the reader to these papers for further details. In particular, [47] demonstrates that the mechanism shown here is compatible with inheritance (though it requires some additional machinery). These ideas give a semantic basis for some aspects of the encapsulation via friends found in C++ and the encapsulation in Cecil [20]. Returning to the example of integer set objects (and dropping the *union* method, for brevity), it is clear that the typing

$$\begin{aligned} IntSet \equiv OT \langle\langle & add\colon int \to unit; \\ & member\colon int \to bool; \\ & superSetOf\colon IntSet \to bool\rangle\rangle \end{aligned}$$

does not provide a sufficiently rich protocol to allow the *superSetOf* method to be implemented: there is no way to find out what are the elements of the other set (the one provided as argument to *superSetOf*). We have no choice but to extend the interface of set objects with a method that provides access to this information; let us call it *rep*, as a reminder that, in general, it may need to provide access to the whole internal representation of the object.

$$\begin{aligned} IntSetExposed \equiv \langle\langle & add\colon int \to unit; \\ & member\colon int \to bool; \\ & superSetOf\colon IntSetExposed \to bool \\ & rep\colon IntList\rangle\rangle \end{aligned}$$

Now we can easily implement all the methods of *IntSetExposedClass*, as shown in Figure 13.

It remains to show how to package the class *IntSetExposedClass* so that the *rep* method can only be called by other instances of the same class. For this, we generalize Mitchell and Plotkin's motto that "abstract types have existential type" [44], combining it with the idea of object interfaces as type operators from Cardelli and Wegner's *partially abstract types* [16].

The interface of the exposed integer set objects can be written as follows.

$$\begin{aligned} IntSetExposedOperator(S) \equiv \langle\langle & add\colon int \to unit; \\ & member\colon int \to bool; \\ & superSetOf\colon S \to bool \\ & rep\colon IntList\rangle\rangle \end{aligned}$$

```
class IntSetExposedClass
   instance variables
      elts: IntList
   methods
      add(i: int): unit is elts := elts.cons(i)
      member(i: int): bool is return(elts.memq(i))
      superSetOf(s: IntSet): bool is
                  return(elts.superListOf(s.rep))
      rep: IntList is return(elts)
end class
```

Figure 13: The class *IntSetExposedClass*, for which writing *superSetOf* is straightforward

```
intSetPackage =
      pack
         procedure  newIntSet() is
         var
            nuSet: Fix(IntSetExposedOperator)
         begin
            nuSet := new IntSetExposedClass();
            return(nuSet)
         end
      as
         Some(ISOp <: IntSetOperator)
               OT ⟨⟨newIntSet :  Fix(ISOp)⟩⟩
      hiding
         IntSetExposedOperator
      end
```

Figure 14: The package intSetPackage.

Similarly, the interface of ordinary integer set objects (without *rep*) can be written:

$$\begin{aligned} \mathit{IntSetOperator}(S) \quad \equiv \langle\!\langle & \mathit{add}\colon \mathit{int} \to \mathit{unit};\\ & \mathit{member}\colon \mathit{int} \to \mathit{bool};\\ & \mathit{superSetOf}\colon S \to \mathit{bool}\rangle\!\rangle \end{aligned}$$

Now comes the key point. Instead of defining $\mathit{IntSet} = \mathrm{Fix}(\mathit{IntSetOperator})$ as we did before, we build an abstract data type (ADT) and then open it to obtain *IntSet*. The implementation of the ADT uses *IntSetExposedOperator*, so that *superSetOf* makes sense, but the *rep* method is hidden from public view.

The integer set package (or module) is defined in Figure 14. To verify that its type is

$$\mathsf{intSetPackage} \;:\; \mathit{Some}(\mathit{ISOp} <: \mathit{IntSetOperator})\ \ OT\ \langle\!\langle \mathsf{newIntSet} :\ \mathrm{Fix}(\mathit{ISOp})\rangle\!\rangle$$

we need only check that when the hidden "witness type" *IntSetExposedOperator* is replaced by the abstract placeholder *ISOp* in the type of the body of the package

$$OT\ \langle\!\langle \mathsf{newIntSet} :\ \mathrm{Fix}(\mathit{IntSetExposedOperator})\rangle\!\rangle$$

we obtain the body of the abstract type:

$$OT\ \langle\!\langle \mathsf{newIntSet} :\ \mathrm{Fix}(\mathit{ISOp})\rangle\!\rangle.$$

Having built intSetPackage, we can open it to obtain the creation procedure newIntSet and the abstract interface *ISOp*, from which we define the type *IntSet*:

```
open intSetPackage
as ISOp with OT ⟨⟨newIntSet⟩⟩

type IntSet = Fix(ISOp)
```

In the remainder of the program, objects created using newIntSet have type *IntSet*. In particular, they can be sent the *superSetOf* message.

In effect, what we have accomplished is to blend object- and ADT-style abstraction mechanisms. The primary mechanism is objects: both ordinary (unary) operations like *add* and binary operations like *superSetOf* are methods of objects rather than free-standing procedures. The extra layer of packaging guarantees that elements of *IntSet* can only be created by calling newIntSet—i.e., that every element of *IntSet* is actually an instance of *IntSetExposedClass*, and hence supports the *rep* message.

6 Summary and Conclusions

Binary methods pose real problems in object-oriented programming languages. There is a typing problem because types with binary methods have few interesting subtypes, and there is a problem obtaining privileged access to additional arguments in binary methods.

We discussed the following solutions to the typing problem for binary methods.

- Avoiding binary-methods completely. We proposed several techniques for achieving similar effects.
- Using a notion of matching, which is weaker than subtyping. This allows more polymorphism in the presence of types with binary methods. However, it seems to require programmers to plan ahead more than they would using subtyping, and its flexibility is not as great as with multi-methods.
- Using multi-methods, either as a basis for object-oriented programming, or as a solution within the framework of single-dispatched languages. This gives the programmer more flexibility in programming binary methods, and consequently allows more subtyping. However, there are modularity and efficiency problems with these approaches.
- Using more precise typings for methods (including the Ingalls simulation of multi-methods). This allows more flexibility than matching. However, it seems to require type inference to be practical [28], and the resulting types may be more complicated than programmers want to see.

To solve the problem of privileged access to additional arguments, we discussed adding additional layers of abstraction. However, this by itself does not solve the typing problems of binary methods; it must be combined with one of the previous solutions.

Matching, multi-methods, and precise typings offer three different solutions to the binary methods problem. Matching insists that binary methods have the type of both arguments (the receiver and the extra argument) exactly the same, and statically enforces this property. Multi-methods allow all heterogeneous invocations of binary methods, so for instance the *equal* method of a *ColorPoint* may be passed a *Point* as argument. Precise typings lie somewhere between the two: they do not insist that binary methods have the same type for both arguments, but also do not allow all safe heterogenous invocations of binary methods.

To illustrate a weakness of multi-methods, consider the binary methods of *DoubleLinkClass* in Figure 5. Using multi-methods, there is a problem if the *setNext* method of *DoubleLink* is invoked with a *Link* node as argument: the *Link* should point back to the *DoubleLink*, but it cannot. The best solution is to define a multi-method case here to flag a run-time error. In the case of matching, the receiver and argument must be the same type and the type system safely precludes such a message send. Precise typings produce a solution between the two: a *Link* may point to a *DoubleLink*, but a *DoubleLink* may not point to a *Link*. The latter restriction, imposed by the type system, prevents the above run-time error from arising.

To illustrate a lack of expressiveness of matching, consider a graphical user interface in which an *AlertWindowClass* has been defined by inheritance from *WindowClass*. A binary method *overlap* should be able to compare plain windows with alert windows. Note also that the *overlap* method should be specialized in *AlertWindowClass* in order to take into account some priority of the alerts (thus *overlap* is a binary method). This could be programmed with multi-methods and precise typings, but not with matching.

A weakness of precise typings is illustrated by the need to use Ingalls' solution to simulate multiple dispatch. This solution is an *ad hoc* implementation of multiple dispatch. All three solutions thus have strengths and weaknesses. This suggests that the integration of different solutions into a single object-oriented language is a task worthy of study.

So, which solution is the best? None of the solutions discussed above are perfect. Some work also remains in determining if some of the solutions will scale up to full-featured languages. For practical programming languages the bottom line may be the empirical question of what sort of inconvenience the programmer is most likely to tolerate. It is our hope that further research will uncover better solutions, perhaps using some combination of the techniques discussed in this paper.

Acknowledgements

Thanks to the US National Science Foundation and ESPRIT for their support of the workshop that resulted in this paper. Thanks to the anonymous referees for comments that helped improve this paper.

References

[1] Martín Abadi and Luca Cardelli. On subtyping and matching. In *Proceedings ECOOP '95*, pages 145–167. LNCS 952, Springer-Verlag, 1995.

[2] Rakesh Agrawal, Lindga G. DeMichiel, and Bruce G. Lindsay. Static type checking of multi-methods. *ACM SIGPLAN Notices*, 26(11):113–128, November 1991. OOPSLA '91 Conference Proceedings, Andreas Paepcke (editor), October 1991, Phoenix, Arizona.

[3] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4), September 1993.

[4] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, NY, June 1987. Springer-Verlag. Lecture Notes in Computer Science, Volume 276.

[5] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, NY, 1991.

[6] Eric Amiel, Oliver Gruber, and Eric Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *OOPSLA '94 Conference Proceedings*, volume 29(10) of *SIGPLAN Notices*, pages 244–258. ACM, October 1994.

[7] Apple Computer Inc., Eastern Research and Technology. *Dylan: an object-oriented dynamic language*, April 1992.

[8] François Bancilhon, Claude Delobel, and Paris Kanellakis (eds.). *Implementing an Object-Oriented database system: The story of* O_2. Morgan Kaufmann, 1992.

[9] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. *ACM SIGPLAN Notices*, 21(11):78–86, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[10] Andrew P. Black and Norman Hutchinson. Typechecking polymorphism in Emerald. Technical Report CRL 91/1 (Revised), Digital Equipment Corporation, Cambridge Research Lab, Cambridge, Mass., July 1991.

[11] John Boyland and Giuseppe Castagna. Type-safe compiling of covariant specialization: a practical case. Technical Report CSD-95-890, University of California, Berkeley, November 1995. Currently available by anonymous ftp from `ftp.ens.fr` in file `/pub/dmi/users/castagna/o2.ps.Z`.

[12] Kim B. Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[13] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings ECOOP '95*, pages 27–51. LNCS 952, Springer-Verlag, 1995. A complete version of this paper with full proofs is available via http://www.cs.williams.edu/∼kim/.

[14] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[15] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation* 76(2/3):138–164, 1988.

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

[17] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[18] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, February 1995. A preliminary version appeared in *ACM Conference on LISP and Functional Programming*, June 1992 (pp. 182–192).

[19] Giuseppe Castagna and Gary T. Leavens. Foundations of object-oriented languages: 2nd workshop report. *SIGPLAN Notices*, 30(2):5–11, February 1995.

[20] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, New York, NY, 1992.

[21] Craig Chambers. multi-method implementation question. personal communication via e-mail, August and November 1995.

[22] Craig Chambers and Gary T. Leavens. Typechecking and modules for multi-methods. *ACM SIGPLAN Notices*, 29(10):1–15, October 1994. OOPSLA '94 Conference Proceedings, October 1994, Portland, Oregon.

[23] Weimin Chen, Volker Turau, and Wolfgang Klas. Efficient dynamic look-up strategy for multi-methods. In Mario Tokoro and Remo Pareschi, editors, *ECOOP '94, European Conference on Object-Oriented Programming, Bologna, Italy*, volume 821 of *Lecture Notes in Computer Science*, pages 408–431, New York, NY, July 1994. Springer-Verlag.

[24] William R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, New York, NY, 1991.

[25] 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.

[26] Jeffrey Dean, David Grove, and Craig Chambers. Efficient dynamic look-up strategy for multi-methods. In Walter Olthoff, editor, *ECOOP '95, European Conference on Object-Oriented Programming, Aarhus, Denmark*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, New York, NY, August 1995. Springer-Verlag.

[27] L.G. DeMichiel and R.P. Gabriel. Common Lisp Object System overview. In Bézivin, Hullot, Cointe, and Lieberman, editors, *Proc. of ECOOP '87 European Conference on Object-Oriented Programming*, number 276 in LNCS, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[28] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings of OOPSLA '95*, pages 169–184, 1995.

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

[30] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*,

New Orleans, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. http://www.elsevier.nl:80/mcs/tcs/pc/volume01.htm.

[31] Giorgio Ghelli. A static type system for message passing. In *OOPSLA '91 Conference Proceedings*, pages 129–145, 1991.

[32] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[33] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 1995. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[34] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In Norman Meyrowitz, editor, *OOPSLA '86 Conference Proceedings, Portland, Oregon, September 1986*, volume 21(11) of *ACM SIGPLAN Notices*, pages 347–349. ACM, November 1986.

[35] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *Conference Record of POPL '94: 21st ACM SIGPLAN–SIGACT Symposium of Principles of Programming Languages, Portland, Oregon*, pages 138–150. ACM, January 1994.

[36] Gregor Kiczales and Luis H. Rodriguez Jr. Efficient method dispatch in PCL. In Andreas Paepcke, editor, *Object-Oriented Programming: the CLOS Perspective*, chapter 14, pages 335–348. MIT Press, Cambridge, Mass., 1993.

[37] Gary T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[38] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In N. Meyrowitz, editor, *OOPSLA ECOOP '90 Proceedings*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223. ACM, October 1990.

[39] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 1994. To appear. An expanded version is Department of Computer Science, Iowa State University, Technical Report 92-28d, August 1994.

[40] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Andrew C. Myers. Theta reference manual. Technical Report Programming Methodology Group Memo 88, MIT, February 1995.

[41] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[42] Narciso Martí-Oliet and José Meseguer. Inclusions and subtypes. Technical Report SRI-CSL-90-16, Computer Science Laboratory, SRI International, December 1990.

[43] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, 1992.

[44] John Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988.

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

[46] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[47] Benjamin C. Pierce and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[48] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[49] John Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development*. Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.

[50] N. Rodriguez, R. Ierusalimschy, and J. L. Rangel. Types in school. *SIGPLAN Notices*, 28(8), 1993.

[51] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In Norman Meyrowitz, editor, *OOPSLA '86 Conference Proceedings, Portland, Oregon, September 1986*, volume 21(11) of *ACM SIGPLAN Notices*, pages 9–16. ACM, November 1986.

[52] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass, 1986.

[53] Larry Tesler. Object Pascal report. Technical Report 1, Apple Computer, 1985.