Clos solutions to binary methods

Didier Verna*

Abstract—**Implementing binary methods in traditional object-oriented languages is difficult: numerous problems arise, such as typing (covariance *vs.* contravariance of the arguments), polymorphism on multiple arguments (lack of multi-methods) *etc.* The purpose of this paper is to demonstrate how those problems are either solved, or nonexistent in the Common Lisp Object System (Clos). Several solutions for implementing binary methods in Clos are proposed. They mainly consist in re-programming a binary method specific object system through the Clos meta-object protocol (Mop).**

Keywords: ***Binary methods, Common Lisp,*** **Clos,** ***Meta-Object Protocol***

1 Introduction

Binary operations work on two arguments of the same type (regardless of the return type). Common examples include arithmetic operations (=, +, − *etc.*) and ordering relations (=, <, > *etc.*). In the context of object-oriented programming, it is often desirable to implement binary operations as methods applying to two objects of the same class in order to benefit from polymorphism. Such methods are hence called *binary methods* .

However, implementing binary methods in many traditional object-oriented languages is a difficult task, most of the difficulty having to do with the relationship between types and classes in the context of inheritance. In this paper, we approach the concept of binary method from the object-oriented perspective of Common Lisp.

In section 2, we present a vain attempt in C++, explain why it is actually not possible to implement binary methods directly in such a language, and why the issue does not exist in Common Lisp. Section 3 refines the initial Common Lisp implementation by presenting a first customization feature of Clos. In section 4, we delve a bit more into Clos and present ways to make sure that binary methods are used properly. In section 5, we delve even more into the Clos Mop and present ways to make sure that binary methods are defined properly.

Throughout this paper, only simplified code excerpts are presented, for the sake of conciseness and clarity. However, fully functional Common Lisp code is available for download at the author's website[1].

*Epita Research and Development Laboratory, 14–16 rue Voltaire, F-94276 Le Kremlin-Bictre, France. Email: didier@lrde.epita.fr

2 Types, classes, inheritance

In this section, we describe the major problem of binary methods in a traditional object-oriented context: mixing types and classes with an inheritance scheme[3]. We also explain why this problem simply does not exist in Common Lisp [1, 9]. In order to illustrate our matter, we take the same example as used by [3], and provide excerpts from a sample implementation in C++ [2].

2.1 A C++ implementation attempt

Consider a `Point` class representing 2D points from an image, equipped with an equality operation. A sample implementation is given in listing 1 (details omitted).

```
class Point
{
  int x, y;

  bool equal (Point& p)
  { return x == p.x && y == p.y; }
};
```

Listing 1: Excerpt from the `Point class`

Now, consider a `ColorPoint` class representing a `Point` associated with a color. A natural implementation would be to inherit from the `Point` class, as shown in listing 2 (details omitted).

```
class ColorPoint
  : public Point
{
  std::string color;

  bool equal (ColorPoint& cp)
  {
    return color == cp.color
      && Point::equal (cp);
  }
};
```

Listing 2: Excerpt from the `ColorPoint class`

Unfortunately, this implementation doesn't work: if you happen to manipulate objects of class `ColorPoint` through pointers to `Point` (which *is* perfectly legal by definition of inheritance), only definition for `equal` from the base class will be seen. That is because the definition

[1] `http://www.lrde.epita.fr/~didier/`

in the derived class simply *overloads* the one in the base class.

What is actually needed is that the definition pertaining to the *exact* class of the object be used. This is, in the C++ jargon, exactly what *virtual methods* are for. A proper implementation would then be to make the `equal` method virtual, as shown in listing 3.

```
// In the Point class:
virtual bool equal (Point& p)
{ return x == p.x && y == p.y; }

// ...

// In the ColorPoint class:
virtual bool equal (ColorPoint& cp)
{
  return color == cp.color
    && Point::equal (cp);
}
```

Listing 3: The `equal` virtual method

Unfortunately again, this implementation doesn't behave as expected. Indeed, C++ does not allow the arguments of a virtual method to change type in this fashion, because this would break typing.

2.2 The typing problem

Here, we describe informally the problem that we face when typing binary methods in presence of inheritance. For a more theoretical description, see [3].

When subclassing the `Point` class, we expect to get a *subtype* of it: any object of type `ColorPoint` can be seen as an object of type `Point`, and thus used where a `Point` is expected. This has the implication that our `equal` method must also satisfy a subtyping relationship across the hierarchy. So we must understand the semantics of subtyping for functions: if we are to allow the use of *ColorPoint::equal* where *Point::equal* is expected what are the implications ?

The most important one is that we are then likely to give a *real* `Point` object (which would not be an actual `ColorPoint` one) to the method. In other words, *ColorPoint::equal* cannot assume to get anything more specific than a `Point` object, and in particular, not a `ColorPoint` one. This is precisely the opposite of what we are trying to achieve.

We see that in order to maintain static type safety, the arguments of a polymorphic method must be *contravariant*[4]: subtyping a function implies supertyping the arguments. C++ actually imposes an even stronger constraint: the arguments of a virtual method must be *invariant*. Other languages such as Eiffel[7] allow the arguments to behave in a covariant[4] manner; however, this leads to executable code that may trigger typing errors at runtime.

Things are getting even worse in C++ since the sample implementation presented in listing 3 is actually *valid* C++ code. However, the contravariance constraint being unsatisfied, the behavior is that of overloading (not polymorphism), regardless of the presence of the `virtual` keyword, which can be very confusing.

2.3 A non-issue in Common Lisp

We saw that because of static type safety constraints, it is not possible to implement binary methods directly (explicitely) in C++. In CLOS, the Common Lisp Object System[5] however, the issue simply does not exist. The crucial point here is that the object model for polymorphism in CLOS is quite different from that of traditional object-oriented languages such as C++.

In C++, methods belong to classes. The polymorphic dispatch depends on one parameter only: the object through which the method is called (represented by the "hidden" pointer `this` in C++; sometimes referred to as `self` in other languages).

In CLOS, on the other hand, methods do not belong to classes (classes merely contain only data): calls to methods appear like ordinary function calls. The support for polymorphism is implemented with what is called *multi-methods* (in the CLOS jargon: *generic functions*). A generic function looks like an ordinary one, except for the fact that the actual body to be executed is dynamically selected according to the type of one or more of the function's arguments. A generic function can provide a default implementation, and can be specialized by writing *methods* on it (the meaning of which is quite different from that of a C++ method).

In order to clarify this, listing 4 provides a sample implementation of the `Point` hierarchy in Common Lisp.

```
(defclass point ()
  ((x :initarg :x :reader point-x)
   (y :initarg :y :reader point-y)))

(defclass color-point (point)
  ((color :initarg :color :reader point-color)))


(defgeneric point= (a b))

(defmethod point= ((a point) (b point))
  (and (= (point-x a) (point-x b))
       (= (point-y a) (point-y b))))

(defmethod point= ((a color-point) (b color-point))
  (and (string= (point-color a) (point-color b))
       (call-next-method)))
```

Listing 4: The `Point` hierarchy in Common Lisp

As you can see, a `point` class is defined, but only with data members (called *slots* in the CLOS jargon) and names for their accessors. The `color-point` class is then defined to inherit from `point` and adds a `color` slot.

And now comes the interesting part: the generic function `point=` is defined by a call to `defgeneric`. This call is actually optional, but you may provide a default behavior here. Two *specializations* of the generic function are subsequently provided by calls to `defmethod`. As you can see, a special argument syntax lets you precise the expected class of the arguments: we provide a method to be used when both arguments are of class `point`, and one to be used when both arguments are of class `color-point`.

Now, for testing equality between two points, one simply calls the generic function as an ordinary one: `(point= p1 p2)`. According to the exact classes of the objects, the proper method (in other words function body) is selected and executed automatically. It is worth noting that in the case of CLOS, the polymorphic dispatch (the actual method selection) depends on the class of *both* arguments to the generic function. In other words, a multiple dispatch is used, whereas in traditional object-oriented languages where methods belong to classes, the dispatch occurs only on the first argument. That is the reason for the name "multi-method".

2.4 Corollary

It should now be clear why the use of multi-methods makes the problem with binary methods a non-issue in Common Lisp. Binary methods can be defined just as easily as any other kind of polymorphic function. There is also another advantage that, while being marginal, is still worth mentioning: with binary methods, objects are usually treated equally, so there is no reason to privilege one of them. For instance, in C++, should we call `p1.equal(p2)` or `p2.equal(p1)` ? It is more pleasant aesthetically, and more conformant to the concept to write `(point= p1 p2)`.

In the remainder of this paper, we will gradually improve our support for the concept of binary methods thanks to the expressiveness of CLOS and the flexibility of the CLOS MOP. For the sake of coherence, we will speak of "binary functions" instead of "binary methods" when in the context of Common Lisp.

3 Method combinations

In listing 4, the reader may have noticed an unexplained call to `(call-next-method)` in the `color-point` method of `point=`, and may have guessed that it is used to execute the previous one (hence completing the equality test by comparing point coordinates).

3.1 Applicable methods

In order to avoid code duplication in the C++ code (listing 2), we used a call to `Point::equal` in order to complete the equality test by calling the method from the super-class. In CLOS, things happen somewhat differently. Given a generic function call, more than one method might correspond to the classes of the arguments. These methods are called the *applicable methods*. In our example, when calling `point=` with two `color-point` objects, both our specializations are applicable, because a `color-point` object can be seen as a `point` one.

When a generic function is called, CLOS computes the list of applicable methods and sorts it from the most to the least specific one. We are not going to describe precisely what "specific" means in this context; suffice to say that specificity is a measure of proximity between the classes on which a method specializes and the exact classes of the arguments.

Within the body of a generic function method in CLOS, a call to `(call-next-method)` triggers the execution of the next most specific applicable method. In our example, the semantics of this should now be clear: when calling `point=` with two `color-point` objects, the most specific method is the second one, which specializes on the `color-point` class, and `(call-next-method)` within it triggers the execution of the other, hence completing the equality test.

3.2 Method combinations

An interesting feature of CLOS is that, contrary to most other object-oriented languages where only one method is applied (this is also the default behavior in CLOS), it is possible to use all the applicable methods to form the global execution of the generic function (note that CLOS knows the sorted list of all applicable methods anyway).

This concept is known as *method combinations*: a method combination is a way to combine the results of all applicable methods in order to form the result of the generic function call itself. CLOS provides several predefined method combinations, as well as the possibility for the programmers to define their own.

In our example, one particular (predefined, as a matter of fact) method combination is of interest to us: our equality concept is actually defined as the logical *and* of all local equalities in each class. Indeed, two `color-point` objects are equal if their `color-point`-specific parts are equal *and* if their `point`-specific parts are also equal.

This can be directly implemented by using the `and` method combination, as shown in listing 5.

As you can see, the call to `defgeneric` is modified in order to specify the method combination we want to use, and both calls to `defmethod` are modified accordingly. The advantage of this new scheme is that each method can now concentrate on the local behavior only: note that there is no more call to `(call-next-method)`, as the logical *and* combination is performed automatically. This also has the advantage of preventing possible bugs

```
(defgeneric point= (a b)
  (:method-combination and))

(defmethod point= and
    ((a point) (b point))
  (and (= (point-x a) (point-x b))
       (= (point-y a) (point-y b))))

(defmethod point= and
    ((a color-point) (b color-point))
  (string= (point-color a) (point-color b)))
```

Listing 5: The `and` method combination

resulting from an unintentional omission of this very same call.

It is important to realize that what we have done here is actually modify the semantics of the dynamic dispatch mechanism. While most other object-oriented languages offer one single, hard-wired dispatch procedure, CLOS lets you (re)program it.

4 Enforcing a correct usage of binary functions

In this section, we start enforcing the concept of binary function by addressing another problem from our previous implementation. Our equality concept requires that only two objects of the same exact class be compared. However, nothing prevents a program from comparing a `color-point` with a `point` for instance. Worse, such a comparison would be perfectly valid code and would go unnoticed, because the first method (specialized on the `point` class) applies.

4.1 Introspection in Clos

We can solve this problem by using the introspection capabilities of CLOS: it is possible to retrieve the class of a particular object at run-time (just as it is possible to retrieve the type of any Lisp object). Consequently, it is very simple to check that two objects have the same exact class, an trigger an error otherwise. Listing 6 shows the use of the function `class-of` to retrieve the exact class of an object, in order to perform such a check.

```
(unless (eq (class-of a) (class-of b))
  (error "Objects not of the same class."))
```

Listing 6: Introspection example in CLOS

In order to avoid code duplication, we can simply perform this check in the basic specialization of `point=`, since we know that this method will be used for any of its subclasses. One drawback of this approach is that since this method is always called last, it is a bit unsatisfactory to perform the check in the end, after all more specific methods have been applied.

Another solution would be to perform the check inside a "before-method" (see section 4.3), but before-methods are not available with the `and` method combination type. In order to really fix this problem, we would then have to write our own method combination type. This possible workaround will not be described here, because there is a better solution to this problem, explained in the next section.

4.2 A meta-class for binary functions

There is something conceptually wrong with the solutions proposed or suggested in the previous section: the fact that it makes no sense to compare objects of different classes belongs to the concept of binary function, not to the `point=` operation. In other words, if we ever add a new binary function to the `point` hierarchy, we don't want do duplicate the code from listing 6 yet again.

What we really need is to be able to express the concept of binary function directly. A binary function is a generic function with a special, constrained behavior (taking only two arguments of the same class). In other words, it is a *specialization* of the general concept of generic function. This strongly suggests an object-oriented model, in which binary functions are subclasses of generic functions. This model happens to be accessible if we delve a bit more into the CLOS internals.

CLOS itself is written on top of a *Meta Object Protocol*, simply known as the CLOS MOP [8, 6]. Although not part of the ANSI specification, the CLOS MOP is a *de facto* standard well supported by many Common Lisp implementations, in which CLOS elements are themselves modeled in an object-oriented fashion (one begins to perceive here the reflexive nature of CLOS): for instance, a call to `defgeneric` creates a CLOS object of class `standard-generic-function`. We are hence able to implement binary functions by subclassing standard generic functions, as shown in listing 7.

```
(defclass binary-function
    (standard-generic-function)
  ()
  (:metaclass funcallable-standard-class))

(defmacro defbinary
    (function-name lambda-list &rest options)
  `(defgeneric ,function-name ,lambda-list
     (:generic-function-class binary-function)
     ,@options))
```

Listing 7: The binary function class

Without going into too many details, remember that a call to `defclass` actually creates a (meta-)object of some (meta-)class. Since instances of class `binary-function` are meant to be called as functions, it is required to precise that the `binary function` class meta-object is an instance of a funcallable meta-class. This is done through the `:metaclass` option.

The next step is to make sure that generic functions are instantiated from the correct class. (here, `binary-function`). This is done by passing a `:generic-function-class` argument to `defgeneric`. With a few lines of macrology, we make this process easier by providing a `defbinary` macro to be used instead of `defgeneric`.

4.3 Back to introspection

Now that we have an explicit implementation of the binary function concept, let us get back to our original problem: how and when can we check that only points of the same class are compared ? As seen before, when a generic function is called, CLOS must compute the sorted list of applicable methods for this particular call. In most cases, this can be figured out from the classes of the arguments to the generic call. The CLOS MOP implements this by calling a generic function called `compute-applicable-methods-using-classes` (`c-a-m-u-c` for short). This function takes the concerned generic function meta-object as first argument, and the list of classes derived from the call as the second one. It is thus possible to specialize its behavior for binary functions, as demonstrated in listing 8.

```
(defmethod compute-applicable-methods-using-classes
    :before ((bf binary-function) classes)
  (assert (equal (car classes) (cadr classes))))
```

Listing 8: Back to introspection

Here, we introduce a new kind of method from CLOS. The methods we have seen so far are actually called *primary methods*. They resemble methods from traditional object-oriented languages such as C++. CLOS also provides other kinds of methods, mainly used for side-effects, such as *before-methods*. As there name suggests, these methods are executed before the primary ones. Listing 8 show a specialization of `c-a-m-u-c` for binary functions as a before-method. The result is that the standard primary method is used, just as for general generic functions, but in addition, and beforehand, our introspective check is performed. This effectively removes the need to perform this check in the binary methods themselves, which is much cleaner at the conceptual level.

5 Enforcing a correct implementation of binary functions

In the previous section, we have seen how to make sure that binary functions are *used* as intended to, and we have made that part of the their implementation. In this section, we show how to make sure that binary functions are *implemented* as intended to. This is an occasion to delve even deeper into the CLOS MOP.

5.1 Properly defined methods

Just as it makes no sense to compare points of different classes, it makes even less sense to *implement* methods to do so. The CLOS MOP is expressive enough to make it possible to implement this constraint directly.

When a call to `defmethod` is issued, CLOS must register this new method into the concerned generic function. This is done in the MOP through a call to `add-method` which takes the generic function and the method as arguments. `add-method` is itself a generic function, which means that we can specialize it for our binary functions. This is demonstrated in listing 9.

```
(defmethod add-method :before
    ((bf binary-function) method)
  (assert (apply #'equal
                 (method-specializers method))))
```

Listing 9: Binary method definition check

We want to preserve the behavior of standard generic functions, so we let the primary method alone. However, before it is actually executed, we check that the specialization is correct: the function `method-specializers` returns the list of argument specializations from the method's prototype. In our examples, that would be `(point point)` or `(color-point color-point)`, so all we have to do is check that the members of this list are actually `equal`.

5.2 Strong binary functions

One might realize that our `point=` concept is not yet completely enforced, if for instance, the programmer forgets to implement the `color-point` specialization: when comparing to points at the same coordinates but with different colors, only the coordinates would be checked and the test would silently yet mistakenly succeed.

It would be an interesting safety measure to ensure that for each defined subclass of the `point` class, there is also a corresponding specialization of the `point=` function (we call that a *strong* binary function), and it should be no surprise that the CLOS MOP lets you do just that. In order to sacrifice to the "fully dynamic" tradition of Lisp, we want to perform this check at the latest possible stage: this will avoid the need for block compilation, let the program be executable even if not completed yet *etc.*

The latest possible stage to perform our consistency check is actually when the binary function is called, and we already have seen how this works: the function `c-a-m-u-c` is used to sort out the list of applicable methods. This is precisely the data on which we have to introspect, so we can specialize on the *primary* method this time, and retrieve the list in question simply by calling `call-next-method`.

Our test involves two different things: first we have to assert that there exists a specialization for the exact classes of the objects we are comparing. This is demonstrated in listing 10: we retrieve the specializers for the first method in the list (the most specific one) and compare that with the classes of the arguments given in the binary function call.

```
(let* ((method (car methods))
       (class (car (method-specializers method))))
  (assert (equal (list class class) classes))
  ;; ...
```

Listing 10: Strong binary function check n.1

Next, we have to check that the whole super-hierarchy has properly specialized methods (none were forgotten). This is demonstrated in listing 11 which defines a temporary recursive function that we start by applying on the class of the objects passed to the binary function call.

```
;; ...
(labels ((find-binary-method (class)
    (find-method bf (method-qualifiers method)
                 (list class class))
    (dolist
        (cls (remove-if
             #'(lambda (elt)
                 (eq elt (find-class
                          'standard-object)))
             (class-direct-superclasses class)))
      (find-binary-method cls))))
  (find-binary-method class))
```

Listing 11: Strong binary function check n.2

The function `find-method` retrieves a method meta-object for a particular generic function satisfying a set of qualifiers and a set of specializers. In our case, there is one qualifier: the `and` method combination type (that can be retrieved by the function `method-qualifiers`), and the specializers are twice the class of the objects.

Once we have made sure this method exists, we must perform the same check on the whole super-hierarchy (the bottommost, standard class excepted). As its name suggests, the function `class-direct-superclasses` returns a list of direct superclasses for some class. We can then recursively call our test function on this list.

By hooking the code excerpts from listings 10 and 11 into a specialization of `c-a-m-u-c` for binary functions, we have completed our check for the "strong" property.

6 Conclusion

In this paper we have approached the concept of binary method which is quite problematic in traditional object-oriented languages. We have seen that because of conflicts between types and classes in the presence of inheritance, this concept is not directly implementable in languages such as C++. We also have demonstrated that support for multi-methods renders the problem nonexistent, as in the case of the Common Lisp Object System. By taking advantage of the introspective capabilities of CLOS, and the expressiveness of the CLOS MOP, we have shown that more than just implementing mere binary methods, it is possible to enforce a correct *usage* of them, and even a correct *implementation* of them.

Please note that we do not claim that enforcing such or such behavior is necessarily a good thing. As a matter of fact, it rather goes against the usual liberal philosophy of Lisp in which the programmer is allowed to do anything (including writing bugs). Rather, our point was to demonstrate that your freedom extends to being able to implement a rigid or even dictatorial (but maybe safer) concept if you wish to do so.

It is also important to realize that we have not just made the concept of binary methods available; we have implemented it directly and explicitly. To this aim, we have actually programmed a new object system which behaves quite differently from the default CLOS. CLOS, along with its MOP, is not only an object system. It is an object system designed to let you program your own object systems.

Acknowledgments

The author would like to thank Pascal Costanza for his insight in the CLOS MOP.

References

[1] American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[2] International Standard: Programming Language – C++. ISO/IEC 14882:1998(E), 1998.

[3] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

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

[5] Sonja E. Keene. *Object-Oriented Programming in Common Lisp: a Programmer's Guide to* CLOS. Addison-Wesley, 1989.

[6] Gregor J. Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[7] Bertrand Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

[8] Andreas Paepcke. User-level language crafting – introducing the CLOS metaobject protocol. In Andreas Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, chapter 3, pages 65–99. MIT Press, 1993. Downloadable version at `http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps`.

[9] Guy L. Steele. *Common Lisp the Language, 2nd edition*. Digital Press, 1990. Online and downloadable version at `http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html`.