Journal of Universal Computer Science, vol. 14, no. 20 (2008), 3389-3411
submitted: 23/6/08, accepted: 24/8/08, appeared: 28/11/08 © J.UCS

Binary Methods Programming: the CLOS Perspective

Didier Verna
(EPITA Research and Development Laboratory, Paris, France
didier@lrde.epita.fr)

Abstract: Implementing binary methods in traditional object-oriented languages is difficult. Numerous problems arise regarding the relationship between types and classes in the context of inheritance, or the need for privileged access to the internal representation of objects. Most of these problems occur in the context of statically typed languages that lack multi-methods (polymorphism on multiple arguments). The purpose of this paper is twofold: to show why some of these problems are either non-issues, or easily solved in Common Lisp, and to demonstrate how the Common Lisp Object System (CLOS) allows us to simply define, implement and enforce type-safe binary methods. These last considerations involve re-programming a binary method-specific object system through the CLOS Meta-Object Protocol (MOP).
Key Words: Binary methods, Common Lisp, object orientation, meta-programming
Category: D.1.5, D.3.3

1 Introduction

Binary functions operate on two arguments of the same type. Common examples include arithmetic functions ($=, +, -$ *etc.*) and ordering relations ($=, <, >$ *etc.*). In the context of object-oriented programming, it is often desirable to implement binary functions as methods applied on two objects of the same class, to benefit from polymorphism. Such methods are hence called "binary methods".

Implementing binary methods in many traditional object-oriented languages is a difficult task. The relationship between types and classes in the context of inheritance and the need for privileged access to the internal representation of objects are the two most prominent problems. In this paper, we approach the concept of binary method from the perspective of Common Lisp.

The paper is composed of two main parts. In section 2, we show how the two problems mentioned above are either non-issues or easily solved. In section 3, we go further and show how to support the concept of binary methods *directly* in the language. Finally, we demonstrate how to ensure a correct usage and implementation of it.

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

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

2 Binary methods non-issues

In this section, we describe the two major problems with binary methods in a traditional object-oriented context, as pointed out by [Bruce et al., 1995]: mixing types and classes within an inheritance scheme, and the need for privileged access to the internal representation of objects. We show why the former is a non-issue in Common Lisp, and how the latter can be solved. We take the same examples as used by [Bruce et al., 1995], and provide excerpts from a sample implementation in C++ [C++, 1998] for comparison.

Please note that C++ should by no means be considered as the official representative of "traditional" object-oriented languages. There is actually no such thing as a "traditional" object-oriented language, as each language has its own specificities. Our point is not to provide a precise feature comparison between Common Lisp and C++ (or any other language for that matter). With so different programming philosophies, we believe that would be pointless. Rather, our point is to demonstrate the expressive power of CLOS. The C++ examples given here and there should only be taken as illustrations of difficulties that might occur elsewhere.

2.1 Types, classes, inheritance

Consider a C++ `Point` class representing 2D image points, equipped with an equality operation. Consider further a C++ `ColorPoint` class representing a `Point` associated with a color. A natural implementation would be to inherit from the `Point` class, as shown in listing 1 (details omitted).

```
class Point
{
  int x, y;

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

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

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

Listing 1: The `Point` class hierarchy in C++

Note that instead of manually comparing point coordinates (hence duplicating code from the base class), the `ColorPoint` equality method explicitly calls the one from the `Point` class through the `Point::` qualifier.

This implementation does not behave as expected, however, because what we have done in the `ColorPoint` class is simply *overload* the `equal` method. If

one happens to manipulate objects of class `ColorPoint` through references to `Point` (which *is* perfectly legal by definition of inheritance), only the definition for `equal` from the base class will be seen, as demonstrated in listing 2.

```
bool foo (Point& p1, Point& p2)
{
  // Point::equal is called
  return p1.equal (p2);
}
```

```
ColorPoint p1 (1, 2, "red");
ColorPoint p2 (1, 2, "blue");

foo (p1, p2); // => true. Wrong!
```

Listing 2: Method overloading

On the contrary, we would like to use the definition pertaining to the *exact* (in the C++ sense) class of the object. This is, in C++ jargon, exactly what *virtual methods* are for. In a proper implementation, we should make the `equal` method virtual, as shown in listing 3. Unfortunately, this implementation doesn't behave as expected either. C++ does not allow the arguments of a virtual method to change type in this fashion. This example would not statically type check.

```
// 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* methods

2.1.1 The static type safety problem

Here we informally describe the problem that we face when typing binary methods in the presence of inheritance. For a more theoretical description, see [Bruce et al., 1995].

```
bool foo (Point& cp, Point& p)
{
  return cp.equal (p);
}
```

```
ColorPoint cp (1, 2, "red");
Point       p (1, 2);

foo (cp, p); // => ???
```

Listing 4: The static type safety problem

By the definition of inheritance, a `ColorPoint` *is* a `Point`. So it should be possible to use a `ColorPoint` where a `Point` is expected. Consider the situation described in listing 4. The function `foo` expects two `Point` arguments, but actually gets a `ColorPoint` as the first one. Assuming that the `equal` method from the exact class is called (hence `ColorPoint::equal`), we see that this method attempts to access the `color` field in a `Point`, which does not exist. Therefore, if we want to preserve static type safety, this code should not compile.

To prevent this situation from happening, we see that the `ColorPoint::equal` method should not expect to receive anything more specific than a `Point` object and, in particular, not a `ColorPoint` one — exactly the opposite of what we are trying to achieve. More precisely, maintaining static type safety in a context of inheritance implies that polymorphic methods must follow a *contravariance* rule [Castagna, 1995] on their arguments: a derived method in a subclass can be prototyped as accepting only arguments of the same class or of a superclass of the original arguments.[2]

Note that C++ requires the arguments of a virtual method to be *invariant* and not just contravariant. Things become even more confusing once you realize that the sample implementation presented in listing 3 is actually *valid* C++ code. The invariance constraint being unsatisfied, the behavior is that of overloading (not polymorphism), regardless of the presence of the `virtual` keyword. One simply ends up with two separate methods that can individually be virtual, so the compiler doesn't even issue a warning.

Other languages such as O_2 [Bancilhon et al., 1992] and Eiffel [Meyer, 1992] allow the arguments of a polymorphic method to behave in a covariant manner. However, this leads to executable code that may trigger typing errors at run-time as described above.

One final note on this problem. The fact that it is not straightforward to implement binary methods while maintaining static type safety can arguably be considered a lack in expressiveness. However, that does not make the contravariance rule a *defect* in the object model. In fact, as clearly pointed out by [Castagna, 1995]:

- the contravariance rule is useful when you want to express *subtyping* (by subclassing),
- the covariance one is appropriate when you want to express *specialization*.

These concepts are not antagonistic; they just express different things. [Castagna, 1995] also demonstrates that the lack of support for the latter is not due to the *record-based* object model [Cardelli, 1988] (where methods belong to classes), but rather to the failure of languages using it to capture the

[2] Conversely, and for reasons not detailed here, the return type of a polymorphic method must follow a *covariance* rule.

notion of multiple dispatch, supported natively by Common Lisp, as described in the next section.

2.1.2 A non-issue in Common Lisp

This crucial difference makes correct construction of binary methods in CLOS, the Common Lisp Object System[Bobrow et al., 1988, Keene, 1989], free from the previous problems.

In languages using the record-based model such as C++, methods belong to classes and the polymorphic dispatch depends only on one parameter: the class of the object through which the method is called (represented by the "hidden" pointer `this` in C++; sometimes referred to as `self` in other languages). CLOS, on the other hand, differs in two important ways.

1. Methods do not belong to classes (there is no privileged object receiving the messages). A polymorphic call *appears* in the code like an ordinary function call. Functions whose behavior is provided by such methods are called *generic functions*.

2. More importantly, CLOS supports *multi-methods*, that is, polymorphic dispatch based on *any* number of arguments (not only the first one).

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

(defgeneric point= (a b))

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

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

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

Listing 5: The `Point` hierarchy in Common Lisp

To clarify this, listing 5 provides a sample implementation of the `Point` hierarchy in Common Lisp (details omitted)[3] As you can see, a `point` class is defined with only 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.

[3] Note that `point=` is a perfectly valid name in Lisp, and is along the lines of standard Common Lisp functions like `string=`.

The remainder is more interesting. A 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 syntax for the arguments allows the specification of the expected class of each. 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`.

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 *both* arguments, the proper method is used. The case where the generic function would be called with two arguments of different classes (for example, `point` and `color-point`) will be addressed in section 3.2.

2.1.3 Aesthetic note

It should now be clear why the use of multi-methods makes the problem 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, albeit marginal, advantage which 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++, one has to choose between `p1.equal(p2)` and `p2.equal(p1)`. This is unsatisfactory because the two points play a different role here: one is a caller, the other is an argument, and there is no reason to choose one form or the other. On the contrary, by using either `(point= p1 p2)` or `(point= p2 p1)`, both points play the same role: they are both arguments to a function call. This is more pleasant aesthetically, and more conformant to the concept (just like in mathematics, where you could equally write $a = b$ or $b = a$).

There are cases, such as `set.addElement(elt)`, where methods as class members do not seem to be aesthetically disturbing. Even in those cases, it is likely that lispers would rather stick to the ordinary function call syntax, such as `(add-element set elt)`. We won't develop this discussion any further, as we believe it is mainly a matter of personal taste. What is important is to remember that the record based model does *not* prohibit the coexistence of subtyping and specialization: only deficient implementations of it do.

2.2 Privileged access to objects internals

The second problem exposed by [Bruce et al., 1995] involves more complex situations where the need for accessing the objects internals (normally hidden from public view) arises.

Consider a type `IntegerSet`, representing sets of integers, with an interface providing methods such as the following (their purpose should be obvious):

```
add    (i: Integer): Void
member (i: Integer): Boolean
```

and also a binary method like the one below:

```
superSet (a: IntegerSet, b: IntegerSet): Boolean
```

Consider further that several implementations are available, perhaps storing the set as a list or array of integers, a bitstring or whatever else. While the implementations of `add` and `member` are not an issue, the binary method *is* problematic. Indeed, this method needs to access the individual elements of the sets. It is possible to enrich the above interface with a method returning the sets elements in a single format (for instance, a list). However, the concern expressed by [Bruce et al., 1995] is that it may be preferable to work directly on the internal representation for efficiency reasons. The conclusion they draw from this example is twofold:

1. a mechanism is needed to constrain both arguments of the binary method to be not only of the same *type*, but also of the same *implementation*,
2. an additional mechanism is also needed to allow access to this internal representation while keeping it hidden from general public view.

They also admit that this problem exists whether binary methods (in the record based model) or traditional (external) procedures are used.

2.2.1 Types *vs.* implementation

It would be slightly misleading to state that the first problem above is a "non-issue" in Common Lisp because the situation does not arise exactly in the same terms. When considering constraining both type and implementation to be the same, [Bruce et al., 1995] are silently assuming that there is (or should be) a clear distinction between them. However, CLOS does not explicitly provide any such distinction.

Note that even in the case of statically typed languages, support for this paradigm is achieved with various degrees of success. Consider for instance the notion of *abstract class* in C++: a non-instantiable class but yet a class. Consider also the notion of *interface vs. implementation* in Java [Gosling et al., 2005], which goes one step further, but which you are not obliged to use, and which also has its own limitations.

In dynamic languages such as Common Lisp however, we might think of solutions in which this distinction is *intentionally* blurred. For instance, we can

define a *single* `integer-set` class equipped with a `set` slot, and let different instances of this class use different `set` types (lists, arrays, bitstrings *etc.*) at run-time. In such a case, the `super-set` function need not be generic anymore (since we only have one class to deal with), but will in turn involve a generic call to effectively compare sets, once their actual type is known.

Also, note that the multiple dispatch offered by Common Lisp generic functions will allow us to implement this comparison even for different kinds of sets, something that [Bruce et al., 1995] give up when constraining both arguments of the binary method to be not only of the same *type*, but also of the same *implementation*.

2.2.2 Data encapsulation

The last problem we have to address is the need for accessing the internal representation of objects while still following the general principle of information hiding. The assumption is that, in the general case, only the type (or the interface) of an object should be public; its implementation should not.

Information hiding or data encapsulation is a concern that predates object orientation. Languages such as Modula-2 [Modula, 1996, King, 1988] and Ada [Ada, 1995, Skansholm, 1996] draw a clear line between accessible interface and hidden implementation. For some reason, traditional object-oriented languages provide additional mechanisms for classes, again, with different levels of success. For instance, C++ inherits the limited module support of the C language (`extern` declarations in headers, `static` definitions in source files) and provides further access control to class members via the `public`, `protected` and `private` keywords. Because of the inherent limitations to this mechanism (it doesn't handle access across different hierarchies), C++ also provides an additional machinery for letting "friends" functions or classes access the private part of another class. A typical situation where this machinery is needed would be our very example of integer sets.

We believe that information hiding and object orientation are two separate issues that should remain orthogonal, and that the need for an object-oriented specific information hiding machinery only demonstrates the lack of a more general and expressive enough mechanism. Common Lisp seems to follow these lines, as CLOS itself does not provide any functionality for data encapsulation. However, Common Lisp *packages* can be used to this aim.

The Common Lisp package system may look familiar at a first glance, because it lets you define *packages*, *import* and *export* things, a vocabulary that should sound familiar for instance to people aware of Java packages, or, to some extend, C++ namespaces. However, this system is very special in at least two ways.

1. Common Lisp packages are collections of *symbols* only (more precisely, a bijective mapping between names and symbols). These symbols can in turn

be used for storing functional definitions, classes, instances or any kind of value, but it does not make any sense to say that a function belongs to a package.

2. The package system operates at *read-time*, that is, the mapping from a name to the corresponding symbol occurs when code is read. This can make the behavior of Common Lisp somewhat tricky to understand, but is also quite powerful.

Back to our original example (the `point` class), we will now roughly describe how one would use the package system to perform implementation hiding. Many important aspects of Common Lisp packages are omitted because our point is not to describe them thoroughly. For more information on packages, see [Gat, 2003, Seibel, 2005].

```
(defpackage :point
    (:use :cl)
  (:export :point
           :point-x
           :point-y))
```

```
(in-package :point)

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

Listing 6: The `point` class package

The right side of listing 6 shows the definition for the `point` class, that would typically go in its own file `point.lisp`. Note that the class definition is exactly the same as in listing 5; there is nothing in the class definition to separate interface from implementation. Only the first line of code is new. It merely tells Common Lisp that the current package should be a certain one named `point`. Packages have their own namespace so there is no clash with the class name. Names beginning with a colon refer to special symbols belonging to the `keyword` package. When the reader encounters a name for a symbol which is not found, it automatically creates the corresponding symbol and adds it to the current package. In our case, the effect is to add 5 new symbols into the `point` package: `point`, `x`, `y`, `point-x` and `point-y`. Again, remember that we are talking about symbols. Associated variables or functions do *not* belong to packages.

In order to effectively declare what is "public" and "private" in a package, one has to provide a package definition such as the one depicted on the left side of listing 6. The `:use` clause specifies that while in the `point` package, all public symbols from the `cl` package (the one that provides the Common Lisp standard) are also directly accessible. Consider that if this clause had been missing, we could not have accessed the macro definition associated with the symbol `defclass` directly. The `:export` clause specifies which symbols are public. As

you can see, the class name and the accessors are made so, but the slot names remain private.

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

Listing 7: Access to exported symbols

Now, to access the public (exported) symbols of the `point` package, one has two options. The first one is to use symbol names *qualified* by the package name, as shown in listing 7. A qualified name contains the package name, a colon and the symbol name. The second option is to `:use` the package, in which case all exported symbols become directly accessible, without qualification. Hence, the `point=` method in listing 5 can be used as-is.

As for the question of accessing private information, that is where the surprise is the most striking for people accustomed to other package systems or information hiding mechanisms: any private (not exported) symbol from a package can be accessed with a double-colon qualified name from anywhere. Thus, one could access the slot values in the `point` class at any place in the code using the symbol names `point::x` and `point::y`.

Accessing a package's private symbols should be considered bad programming style, and used with caution because it effectively breaks modularity. But it is nevertheless easy to do so, and although maybe surprising, is typical of the Lisp philosophy: be very permissive in the language and put more trust on the programmer's skills. The advantage of this approach is that not only the package system and the object-oriented layer are completely orthogonal, but no additional mechanism (such as C++'s "friends") is needed for privileged access either. One simply uses an additional colon when one has to.

3 Binary methods enforcement

In the previous section, we have examined binary methods and some of their notorious problems, and shown that, from the Common Lisp/CLOS perspective, implementing them presents no particular difficulty. However, there is no explicit support for binary methods in the language. In the remainder of this paper, we go further and gradually add support for the concept *itself*, thanks to the expressiveness of CLOS and the flexibility of the CLOS MOP. From now on, we will use the term "binary function" as a shorthand for "binary generic function".

3.1 Method combinations

In listing 5, the reader may have noticed an unexplained call to `call-next-method` in the `color-point` specialization of the `point=` generic function, and may have guessed that it is used to execute the previous one (specialized on plain `point`s), hence completing the equality test by comparing point coordinates.

3.1.1 Applicable methods

In order to avoid code duplication in the C++ case (listing 1), we used a call to `Point::equal` for completing the equality test by calling the method from the superclass. In CLOS, things happen somewhat differently. Given a generic function call, more than one method might fit 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 actual classes of the arguments.

Within the body of a method, 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.

3.1.2 Method combinations

An interesting feature of CLOS is that, contrary to other object-oriented languages where only one method is applied (this is also the default behavior in CLOS), it is possible to use some or all of the applicable methods to form the global execution of the generic function (resulting in what is called an *effective method*). Note that CLOS knows the sorted list of all applicable methods anyway, since this is required to implement `call-next-method`.

This concept is known as *method combinations*. A method combination is a way to combine the results of some or all of the applicable methods to form the result of the generic function call itself. CLOS provides several predefined method combinations and supports the possibility of the programmer to define his own.

In our example, one particular (predefined) 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 8.

```
(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 8: The `and` method combination

As you can see, the call to `defgeneric` is modified 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 call to `call-next-method`, as the logical *and* combination is performed automatically. This also has the advantage of preventing possible bugs resulting from an unintentional omission of this very same call.

This point is arguably quite specific to the particular example of binary function we are using, but nevertheless, it illustrates an interesting feature of CLOS. It is important to realize that what we have done is to actually modify the semantics of the dynamic dispatch mechanism. While other object-oriented languages offer one single, hard-wired, dispatch procedure, CLOS lets you program your own dispatch procedure to fit your needs.

3.2 Enforcing a correct usage of binary functions

In this section, we demonstrate explicit support for the concept of binary function itself by addressing another problem of our previous implementation. Our equality concept requires that only two objects of the same class be compared. However, nothing prevents one from using the `point=` binary function for comparing a `color-point` with a `point`. Our current implementation of `point=` is unsafe because such a comparison is perfectly valid code and the error would go unnoticed. Indeed, since a `color-point` is a `point` by definition of inheritance,

the first specialization (the one on the `point` class) *is* an applicable method, so the comparison will work, but only check for point coordinates.

3.2.1 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 also very simple to check that two objects have the same exact class, and trigger an error otherwise. In listing 9, we show a new implementation of `point=` making use of the function `class-of` to retrieve the exact class of an object, to perform such a check.

```
(defmethod point= and ((a point) (b point))
  (assert (eq (class-of a) (class-of b)))
  (and (= (point-x a) (point-x b))
       (= (point-y a) (point-y b))))
```

Listing 9: Introspection example in Clos

We chose to perform this check only in the least specific method in order to avoid code duplication, because we know that this method will be used for all `point` objects, including instances of subclasses. One drawback of this approach is that since this method is always called last, it is a bit unsatisfactory to perform the check as the last operation, after all more specific methods have been applied, possibly for nothing.

There is something in the design of Clos that can help us here: it is possible to create a specific method combination and have it execute the applicable methods in reverse order (by using the `:most-specific-last` option to `define-method-combination`). There is an even better way, however, as described in the next section.

3.2.2 Before-methods

Clos has a feature perfectly suited to (actually, even designed for) this kind of problem. The methods we have seen so far are called *primary methods*. They resemble methods from traditional object-oriented languages (with the exception that they can be combined together, as we saw in section 3.1.2). Clos also provides other kinds of methods, such as *before* and *after-methods*. As there name suggests, these methods are executed before or after the primary ones.

It should be noted that even when before or after-methods exist, the global return value from the generic call only depends on the primary methods involved. Thus, before and after-methods are typically used for side-effects, such as performing checks.

Unfortunately, before and after-methods cannot be used with the `and` method combination described in section 3.1.2. Thus, assuming that we are back to the initial implementation described in listing 5, listing 10 demonstrates how to properly place the check for class equality. Note the presence of the `:before` keyword in the method definition.

```
(defmethod point= :before ((a point) (b point))
  (assert (eq (class-of a) (class-of b))))
```

Listing 10: Using before-methods

We want this check to be performed for all `point` objects, including instances of subclasses, so this method is specialized only on `point`, and hence applicable to the whole potential `point` hierarchy. But note that even when passing `color-point` objects to `point=`, the before-method is executed before the primary ones, so an improper usage error is signaled as soon as possible. This scheme effectively removes the need to perform the check in the first method itself, which is much cleaner at the conceptual level.

3.2.3 A meta-class for binary functions

There is still something conceptually wrong with the solutions proposed in the previous sections. The fact that it makes no sense to compare objects of different classes belongs to the very 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 to duplicate the code from listing 9 or 10 yet again.

What we really need is 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 conceptual model is 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 [Paepcke, 1993, Kiczales et al., 1991]. Although not part of the ANSI specification, the CLOS MOP is a *de facto* standard well supported by many Common Lisp implementations. Within the MOP, CLOS elements are themselves modeled in an object-oriented fashion (one begins to perceive here the

reflexive nature of CLOS). For instance, classes (the result of calling `defclass`) are CLOS meta-objects that are instances of other meta-classes. By default, new classes are instances of the class `standard-class`. Similarly, a user-defined generic function (the result of calling `defgeneric`) is 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 11.

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

(defmacro defbinary
    (function-name lambda-list &rest options)
  (when (assoc ':generic-function-class options)
    (error
     ":generic-function-class option prohibited"))
  `(defgeneric ,function-name ,lambda-list
    (:generic-function-class binary-function)
    ,@options))
```

Listing 11: The binary function class

The `binary-function` class is defined as a subclass of `standard-generic-function`, and does not provide any additional slots. Since instances of this class are meant to be called as functions, it is also required to specify that the `binary function` meta-class (the class of the `binary-function` class meta-object) is a "funcallable" meta-object. This is done through the `:metaclass` option, which is given the value of `funcallable-standard-class` and not just `standard-class`.

Now that we have a proper meta-class for binary functions, we need to make sure that our binary generic functions are instantiated from it. Normally, one specifies the class of newly created generic functions by passing a `:generic-function-class` argument to `defgeneric`. If this argument is omitted, generic functions meta-objects are instantiated from the `standard-generic-function` class. With a few lines of macrology, we make this process easier by providing a `defbinary` macro that is to be used instead of `defgeneric`. This macro checks that the user did not provide an explicit generic function class. `defbinary` is designed as a syntactic clone of `defgeneric` (so it will look familiar to lispers), but we could also think of all sorts of modifications, including enforcing the lambda-list (the generic call prototype) to be of exactly two arguments, *etc.*

One note on implementation. Since the CLOS MOP is a *de facto* standard only (the reference document being [Kiczales et al., 1991]), some portability issues

arise across implementations (missing or deficient features *etc.*). Even the implementation's package name is not portable (SBCL [Newman, 2000] uses `sb-mop` for instance). For simplicity, these problems are silently skipped in our examples. Pascal Costanza's "Closer to MOP" package [Closer, www] comes in handy for harmonizing the behavior of the CLOS MOP across Common Lisp implementations.

3.2.4 Back to introspection

Now that we have an explicit implementation of the binary function concept, let us return to our original problem: how and when can we check that only points of the same class are compared?

With some knowledge of the internal process involved in calling a generic function, the answer becomes apparent. For each generic function call, CLOS must calculate 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 (otherwise, the arguments themselves are involved). The CLOS MOP handles these two situations by calling `compute-applicable-methods-using-classes` (`c-a-m-u-c` for short), and `compute-applicable-methods` (`c-a-m` for short) if that is not enough.

The one of particular interest to us is `c-a-m-u-c`. It is not an ordinary function, but a *generic* one, taking two arguments:

1. the generic function meta-object involved in the call (in our case, that would be the `point=` one created by the call to `defgeneric`),
2. the list of the arguments classes involved in the generic call (in our case, that would be a list of two element, either `point` or `color-point` class meta-objects).

```
(defmethod c-a-m-u-c :before ((bf binary-function) classes)
  (assert (= (length classes) 2))
  (assert (apply #'eq classes)))
```

Listing 12: Back to introspection

This generic function is interesting to us because, conceptually speaking, before even calculating the applicable methods given the arguments classes, we should make sure that these two classes are the same. This strongly suggests a specialization with a before-method (see section 3.2.2), as demonstrated in listing 12. As you can see, this new method applies to binary functions only,

thanks to the specialization of its first argument on the `binary-function` class created earlier in listing 11. The advantage is that the check now belongs to the binary function concept itself, and not anymore to each individual function one might want to implement.

One note on implementation. In our example, the applicable methods can always be deduced from the classes of the arguments, so `c-a-m` should not intervene (only `c-a-m-u-c` is involved).[4] For the reader already familiar with the MOP, we are assuming here that only class specializers are used with our binary functions (no `eql` specializers), which justifies that `c-a-m` is unimportant. More on this in sections 3.3.1 and 5.

3.3 Enforcing a correct implementation of binary functions

In the previous section, we have made sure that binary functions are *used* as intended, and we have made that requirement part of their implementation. In this section, we make sure that binary functions are *implemented* as intended, and we also make this requirement part of their implementation.

3.3.1 Properly defined methods

Just as it makes no sense, in our particular context, 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`. It is not an ordinary function, but a *generic* one, taking two arguments:

1. the generic function meta-object involved in the call (in our case, that would be `point=`),
2. the newly created method meta-object.

This generic function is interesting to us because before registering the new method, we should make sure that it specializes on two identical classes. This strongly suggests a specialization with a before-method (see section 3.2.2), as demonstrated in listing 13.

Again, this new method applies to binary functions only, thanks to the specialization of its first argument on the `binary-function` class created earlier in

[4] We have implemented these ideas for SBCL, which seems to suffer from a bug (or at least an excessively rigid interpretation of the standard) that renders our specialization of `c-a-m-u-c` inoperative if there is not also one for `c-a-m`. We thus had to define one explicitly so that it simply exists, although this method actually does nothing.

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

Listing 13: Binary method definition check

listing 11. As before, the advantage is that the check belongs directly to the binary function concept itself, and not to every individual function one might want to implement. 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 the same.

Again, for the reader already familiar with the MOP, note that this implementation is actually simplified. To be absolutely correct and honor the assumption made in section 3.2.4, we should also check that the specializers are class ones, and not `eql` ones. More on `eql` specializers in section 5.

3.3.2 Binary completeness

One might realize that our `point=` concept is not yet completely enforced if, for instance, the programmer forgets to implement the `color-point` specialization. In this scenario, when comparing two 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 *binary completeness*), and it should be no surprise that the CLOS MOP lets you do just that.

One of the attractive points of Lisp in a development cycle is its "fully dynamic" aspect. In CLOS, class definitions are allowed to change at run-time and propagate their new definitions to already existing instances. Lisp dialects also provide a "read-eval-print" loop (REPL for short) in which you have full control over your lisp environment, including the ability run, compile, interpret or even modify any part of your program interactively. This also means that it is possible to run parts of an incomplete program, and explains why the notion of block compilation is not as important in Lisp as it is in other languages.

To sacrifice to the "fully dynamic" tradition of Lisp, we want to perform the check for binary completeness at the latest possible stage: obviously when the binary function is called. We have seen how this works already: the function `c-a-m-u-c` is used to sort out the list of applicable methods.

This is very interesting to us because the check for binary completeness involves introspection on exactly this list (to see if some methods are missing). What we can do is specialize on the *primary* method this time, retrieve the list

in question simply by calling `call-next-method`, and then do our own work. This is presented in listing 14.

```
(defmethod c-a-m-u-c ((bf binary-function) classes)
  (multiple-value-bind (methods ok)
      (call-next-method)
    (when ok
      ;; Check for binary completeness
      )
    (values methods ok)))
```

Listing 14: Binary completeness skeleton

The built-in `c-a-m-u-c` returns two values: one (`methods`) is the sorted list of applicable methods and the other (`ok`) indicates whether calling `c-a-m` is needed. After we perform our check for completeness (and possibly trigger an error), we simply return the values we got from the default method. Our check involves two different things.

1. We have to assert that there exists a specialization for the exact class of the objects we are comparing (otherwise, as previously mentioned, a missing specialization for `color-point` would go unnoticed).
2. We have to check that the whole super-hierarchy has properly specialized methods (that is, none were forgotten).

The test for an existing bottommost specialization is demonstrated in listing 15. The most specialized applicable method is the first one in the sorted list of applicable methods (the `methods` variable). The classes on which it specializes are retrieved by calling `method-specializers` (there should be two of them, and we already know from listing 13 that both are identical). We then check that the classes of the arguments involved in the generic call (the `classes` parameter) match the most specific specialization.

```
(let* ((method (car methods))
       (specializers (method-specializers method)))
  (assert (= (length specializers) 2))
  (assert (equal specializers classes))
  ;; ...
```

Listing 15: Binary completeness check n.1

The actual test for completeness of the whole super-hierarchy is demonstrated in listing 16. 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 a 2-element list containing the objects class twice. If such a method does not exist, `find-method` triggers and error.

```
;; ...
(loop :for class :in (class-precedence-list (car classes))
      :until (eq class (find-class 'standard-object))
      :do (find-method bf (method-qualifiers method)
                       (list class class)))
```

Listing 16: Binary completeness check n.2

What we have to do is thus simply call `find-method` on the whole hierarchy we are checking. This is what the `loop` construct does. The function `class-precedence-list` returns the sorted list of super-classes (with duplicates removed, in case our hierarchy is a graph) for our bottommost class. We check each of these classes, the topmost class (`standard-object`) excepted.

By hooking the code excerpts from listings 15 and 16 into the skeleton of listing 14, we have completed our check for the "binary completeness" property.

One note on performance: CLOS implementations are explicitly allowed ([Kiczales et al., 1991], p.175) to optimize their computations by memoizing the results of `c-a-m-u-c` instead of calling it under some precise invariance conditions. As a result, our check for binary completeness does not induce any performance loss in such implementations. More precisely, this check will be performed only once, the first time a generic function is called, for a given set of arguments'classes.

4 Conclusion

In this paper, we have described why binary methods are a problematic concept in traditional object-oriented languages. The relationship between types and classes in the context of inheritance, or the need for privileged access to the internal representation of objects makes it difficult to implement.

From the CLOS perspective, we have demonstrated that implementing binary methods is a straightforward process, for at least two reasons.

1. The covariance / contravariance problem does not exist, because CLOS generic functions support multiple dispatch natively.

2. When privileged access to internal information is needed, the dynamic nature of Common Lisp provides solutions that are unavailable in statically typed languages. Besides, the package system is completely orthogonal to the object-oriented layer and is pretty liberal in what you can access and how (admittedly, at the expense of breaking modularity just as in other languages).

From the MOP perspective, it is also important to realize that we have not just made the concept of binary methods accessible. We have implemented it *directly* and *explicitly*. We have shown ways not only to implement it, but also to enforce a correct *usage* of it, and even a correct *implementation* of it. 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.

Starting at section 3.2.4, the author is well aware of the fact that the material in this paper might be controversial for some lispers. Enforcing such or such behavior rather goes against the usual liberal philosophy of Lisp in which the programmer is allowed to do anything (including writing bugs). For instance, just as we defined a `defbinary` macro, why not simply provide another one like `defbinmethod`, that would perform all our checks instead of all this intricate MOP programming? Our answer to this is as follows.

- First, we never claimed that enforcing anything is necessarily a good thing. However, one still might wish to do so in some circumstances.
- Our point was rather to demonstrate that the freedom Lisp provides extends to being able to implement a rigid or even dictatorial (but maybe safer) concept if you wish to do so.
- There is no incompatibility between providing a layer of macros and *document* the fact that the user should use it, and implementing a concept directly behind the scenes. You can do both. But note that with only a macro layer, the concept is only expressed in the documentation.
- While the Lisp tradition is to trust the skills of the programmer, the situation might not be as ideal as wished for, typically when you write a library or an interactive application to be used by non-experts or even not computer scientists at all. In such situations, it *is* a good thing to provide safeguards in the code, especially if it costs only a few lines of code and has a small impact on performance. Only a direct implementation of the concept can give you that.
- Finally, one advantage of using the MOP (for instance to check for binary completeness) is that all sorts of things can be done as late as possible (for

instance, when the generic call occurs), something much more difficult if at all possible to do with macros. And *this*, for once, is along the lines of the Lisp tradition.

5 Perspectives

The `point=` example was chosen on purpose because it allowed us to illustrate many possible features of binary functions that a programmer would like to get from a rich implementation (equality in itself is a rich and complicated concept [Pitman, 1993]). Studying other examples of binary functions used in real-life applications would probably lead to the discovery of other interesting features one would like to have support for.

The implementation of the ideas presented in this paper require so few lines of code that the usefulness of a properly packaged, general purpose implementation remains arguable. However, it would be a good base for extending the concept, as new scenarios are studied, including relaxation of strictness at runtime (for instance, one could decide on the fly whether binary completeness is wanted or not).

As stated earlier, the question of non-classic specialization remains to be studied. Thanks to the `eql` specializer, CLOS allows methods to specialize on something else than classes. Binary functions should have the right to do so, just like ordinary generic functions. However, even in simple and academic examples like ours, we can see that some problems arise. For instance, it could make sense to specialize `point=` on a particular point, like

```
(point= pt *origin*)
```

but then, it is redundant to execute both this specialization and the one for the corresponding class. We then have to get rid of the `and` method combination type again, or possibly try to write a dedicated one. The lack for explicit superclass reference in CLOS is known to have some limitations [Ducournau and Habib, 1987, Ducournau et al., 1992]. Maybe this is one of them.

Acknowledgments

The author would like to thank Pascal Costanza for his insight in the CLOS MOP, as well as the reviewers for their very constructive remarks.

References

[Ada, 1995] Ada (1995). International Standard: Programming Language – Ada. ISO/IEC 8652:1995(E).

[Bancilhon et al., 1992] Bancilhon, F., Delobel, C., and Kanellakis, P. C., editors (1992). *Building an Object-Oriented Database System, The Story of O_2*. Morgan Kaufmann.

[Bobrow et al., 1988] Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. (1988). Common lisp object system specification. *ACM SIGPLAN Notices*, 23(SI):1–142.

[Bruce et al., 1995] Bruce, K. B., Cardelli, L., Castagna, G., Eifrig, J., Smith, S. F., Trifonov, V., Leavens, G. T., and Pierce, B. C. (1995). On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242.

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

[Cardelli, 1988] Cardelli, L. (1988). A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66.

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

[Closer, www] Closer (www). Closer to MOP. `http://common-lisp.net/project/closer/`.

[Ducournau and Habib, 1987] Ducournau, R. and Habib, M. (1987). On some algorithms for multiple inheritance in object-oriented programming. In *European conference on object-oriented programming on ECOOP '87*, pages 243–252, London, UK. Springer-Verlag.

[Ducournau et al., 1992] Ducournau, R., Habib, M., Huchard, M., and Mugnier, M. L. (1992). Monotonic conflict resolution mechanisms for inheritance. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 16–24, New York, NY, USA. ACM Press.

[Gat, 2003] Gat, E. (2003). The complete idiot's guide to Lisp packages. Downloadable version at `http://www.flownet.com/gat/packages.pdf`.

[Gosling et al., 2005] Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java (tm) Language Specification*. Prentice Hall, third edition.

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

[Kiczales et al., 1991] Kiczales, G. J., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA.

[King, 1988] King, K. (1988). *Modula-2, a Complete Guide*. Jones and Bartlett.

[Meyer, 1992] Meyer, B. (1992). *Eiffel: the Language*. Prentice Hall.

[Modula, 1996] Modula (1996). International Standard: Programming Language – Modula-2. ISO/IEC 10514-1:1996.

[Newman, 2000] Newman, W. H. (2000). Steel Bank Common Lisp user manual. `http://www.sbcl.org`.

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

[Pitman, 1993] Pitman, K. M. (1993). The best of intentions, EQUALS rights – and wrongs – in Lisp. *Lisp Pointers*, 6(4). Online version at `http://www.nhplace.com/kent/PS/EQUAL.html`.

[Seibel, 2005] Seibel, P. (2005). *Practical Common Lisp*. Apress, Berkeley, CA, USA. Online version at `http://www.gigamonkeys.com/book/`.

[Skansholm, 1996] Skansholm, J. (1996). *Ada 95 from the beginning*. Addison Wesley, third edition.