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: first, we show why some of these problems are either non-issues, or easily solved in Common Lisp. Then, we demonstrate how the Common Lisp Object System (Clos) allows us not only to implement binary methods in a straightforward way, but also to support the concept directly, and even enforce it at different levels (usage and implementation).

Key Words: Binary methods, Common Lisp, object orientation, meta-programming
Category: D.1.5, D.3.3

1 Introduction

Binary operations work on two arguments of the same 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 applied on two objects of the same class in order 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 two problems mentioned above are either non-issues, or easily solved. In section 3, we show how to support the concept of binary methods *directly* into the language, and demonstrate how to ensure not only a correct *usage* of it, but also a correct *implementation* of it.

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. In order to illustrate our matter, we take the same examples as used by [Bruce et al., 1995], and provide excerpts from a sample implementation in C++ for comparison.

2.1 Types, classes, inheritance

Consider a `Point` class representing 2D points from an image, equipped with an equality operation. Consider further 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 1 (C++ version, 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: Excerpt from the `Point class`

However, this implementation does not behave as expected because what we have done in the `ColorPoint` class is simply *overload* the `equal` method: `ColorPoint` objects manipulated as `Point` ones will only see the definition for `equal` from the base class, 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

In order to find the proper method definition at run-time in C++, one needs *virtual methods* (obtained by simply prefixing the methods declarations in figure 1 with the keyword `virtual`). Unfortunately, such an implementation doesn't behave as expected. Indeed, C++ does not allow the arguments of a virtual

method to change type as in figure 1, because this would not statically type check.

2.1.1 The static type safety problem

By 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 3. 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 could try 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.

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

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

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

Listing 3: The static type safety problem

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

2.1.2 A non-issue in Common Lisp

In languages 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. The Common Lisp Object System (CLOS [Keene, 1989]), on the other hand, differs in two important ways.

1. Firstly, methods do not belong to classes: a polymorphic call *appears* in the code like an ordinary function call. Functions the behavior of which are provided by such methods are called *generic functions*.

2. Secondly, and more importantly, CLOS supports *multi-methods*, that is, polymorphic dispatch based on any number of arguments, not only the first one (`this` in C++).

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

(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 4: The `Point` hierarchy in Common Lisp

In order to clarify this, listing 4 provides a sample implementation of the `Point` hierarchy in Common Lisp (details omitted). As you can see, `point` and `color-point` classes are defined with only data members (called *slots* in the CLOS jargon). Instead of being class members, two methods on the *generic function* `point=` are defined by calls to `defmethod`. As you can see, a special argument syntax lets you specify the expected class of each: we provide a method for comparing two `point` objects, and one for comparing two `color-point` ones. Testing for equality between two points is now simply a matter of calling the generic function as follows:

```
(point= p1 p2)
```

According to the *exact* classes of *both* of the objects, the correct 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.

2.2 Privileged access to objects internals

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

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

```
add    (i: Integer): Unit
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 (for instance, for efficiency reasons), effectively storing the set as a list or array of integers, as a

bitstring or whatever else. While implementing `add` and `member` is not an issue at all, 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), but the concern expressed by [Bruce et al., 1995] is that it might 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 for constraining both arguments of the binary method to be not only of the same type, but also of the same implementation,
2. another mechanism is also needed to allow access to this internal representation while keeping it hidden from general public view.

2.2.1 Types *vs.* implementation

It would be slightly abusive to claim that point 1 above is a "non-issue" in Common Lisp because the question does not arise exactly in the same terms. When considering constraining both type and implementation to be the same, the authors are silently assuming that there is (or should be) a clear distinction between them. As a matter of fact, CLOS does not explicitly provide any such distinction.

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 have only 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 contrary to the first conclusion drawn by [Bruce et al., 1995], the multiple dispatch offered by Common Lisp generic functions will allow us to implement this comparison even for different kinds of sets (however, this cannot be considered a "binary" operation anymore).

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. Common Lisp itself does not provide any functionality for data encapsulation, but the *package* system is perfectly suited to this task.

Back to our original example (the `point` class), we 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.

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

```
(in-package :point)

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

Listing 5: The `point` class package

The right side of listing 5 shows a definition of the `point` class, which is no different from the one in listing 4; 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`. When Common Lisp encounters, at read-time, 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`. Note that we are only talking about *symbols* here. 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 5. 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`. 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.

Now, in order 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, such as `point:point-x`. The second option is to `:use` the package, in which case all exported symbols become directly accessible, without any qualification. Hence, the `point=` method in listing 4 can be used as-is.

As for the question of accessing private information, this 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.

One important design consideration here is that the package system and the object-oriented layer are completely orthogonal: compare this with languages such as C++ in which information hiding is done by the object-oriented layer itself (`public`, `protected` and `private` members). Also, note that no additional mechanism is needed for privileged access either. One simply uses an additional colon when one really wants to. Again, compare this with languages such as C++ in which an additional machinery is needed (`friend` functions, methods or classes).

For the record, note that Common Lisp allows for completely hiding symbols (they are said to be *uninterned*), but doing that is definitely not the "Lisp way".

3 Binary methods enforcement

While the previous section demonstrated how straightforward it is to implement binary methods, there is no explicit support for them in the language. In the remainder of this paper, we gradually add support for the concept *itself*, thanks to the expressiveness of CLOS and the flexibility of the CLOS Meta-Object Protocol (MOP). From now on, we will use the term "binary function" as a shorthand for "binary generic function".

3.1 Method combinations

When calling `point=` with two `color-point` objects, both of the methods we defined are applicable because a `color-point` object can be seen as a `point` one. More generally, for each generic function call, several methods might fit the classes of the arguments. These methods are called "applicable methods".

When a generic function is called, CLOS computes the list of applicable methods and sorts it from the most to the least specific one. 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 (listing 4), 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 (this is roughly the equivalent of calling `Point::equal` in the C++ version).

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 all, or some of the applicable methods to form the global execution of the generic function (resulting in what is called an *effective method*).

This concept is known as *method combination*: a way to combine the results of all or some of the 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 programmer to define his own.

In our example, one particular (predefined, for that matter) 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 6.

```
(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 6: The `and` method combination

As you can see, the call to `defgeneric` (otherwise optional) specifies 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: 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 resulting from an unintentional omission of this very same call.

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

3.2 Enforcing a correct usage of binary functions

In this section, we start providing explicit support for the concept of binary function itself by addressing another problem from our previous implementa-

tion. Our equality concept requires that only two objects of the same exact class be compared. However, nothing prevents one from using the `point=` binary function for comparing a `color-point` with a `point` for instance. 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. Consequently, it is also very simple to check that two objects have the same exact class, an trigger an error otherwise. In listing 7, we show a new implementation of `point=` making use of the function `class-of` to retrieve the exact class of an object, in order 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 7: 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 in the end, after all more specific methods have been applied, possibly for nothing.

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). 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, and are typically used for side-effects.

Unfortunately, before and after-methods cannot be used with the `and` method combination described in section 3.1. Thus, assuming that we are back to the

initial implementation described in listing 4, listing 8 demonstrates how to properly place the check for class equality. Note the presence of the `:before` keyword in the method definition.

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

Listing 8: 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 occasional 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 concept of binary function itself, 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 listings 7 or 8 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 conceptual 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 [Paepcke, 1993, Kiczales et al., 1991], which architects CLOS itself in an object-oriented fashion: classes (the result of calling `defclass`) are CLOS (meta-)objects, that is, instances of certain (meta-)classes. 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 9.

The `binary-function` class is defined as a subclass of `standard-generic-function`, and does not provide any additional slot. Since

```
(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 9: The binary function class

instances of this class are meant to be called as functions, it is also required to state 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 `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 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 is designed as a syntactic clone of `defgeneric`, 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.*

3.2.4 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 ?

For each generic function call, we saw that 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. The CLOS MOP implements this by calling `compute-applicable-methods-using-classes` (`c-a-m-u-c` for short).

`c-a-m-u-c` is not an ordinary function, but a *generic* one, taking two arguments: first, the generic function meta-object involved in the call (in our case, that would be the `point=` one created by the call to `defgeneric`), and 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 (apply #'eq classes)))
```

Listing 10: 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), and this is demonstrated in listing 10. As you can see, this new method only applies to binary functions, thanks to the specialization of its first argument on the `binary-function` class. The advantage is that the check now belongs to binary function concept itself, and not anymore to each individual function one might want to implement.

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 part of the 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 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 the 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: first, the generic function meta-object involved in the call (in our case, that would be the `point=` one created by the call to `defgeneric`), and the newly created method object.

This generic function is interesting to us because, conceptually speaking, 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), and this is demonstrated in listing 11.

Again, this new method only applies to binary functions, thanks to the specialization of its first argument on the `binary-function` class. And again, 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

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

Listing 11: Binary method definition check

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.

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: 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 *binary completeness*), and it should be no surprise that the CLOS MOP lets you do just that.

Remember that the function `c-a-m-u-c` is used to sort out the list of applicable methods. Again, 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 thus specialize on the *primary* method this time, retrieve the list in question simply by calling `call-next-method`, and then do our own work, as depicted in listing 12. The built-in `c-a-m-u-c` returns two values, the first of which is the list of applicable methods. After we perform our check for completeness (and possibly trigger an error), we simply return the values we got from the default method.

```
(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 12: Binary completeness skeleton

Our check involves two different things: first we have to assert that there exists a specialization for the exact classes of the objects we are comparing (otherwise, as previously mentioned, a missing specialization for `color-point`

would go unnoticed). This is demonstrated in listing 13. The most specialized applicable method is the first one in the list. The classes on which it specializes are retrieved by calling `method-specializers` (it suffices to retrieve the first one because we already know that both are identical; see listing 11). 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))
       (class (car (method-specializers method))))
  (assert (equal (list class class) classes))
  ;; ...
```

Listing 13: Binary completeness 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 14. We define a local recursive function `find-binary-method` that we first apply on the bottommost class in the hierarchy we are checking (the `class` binding from listing 13).

```
;; ...
(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 14: Binary completeness 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 (it 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 (`find-method` would trigger an error otherwise), we must perform the same check on the whole super-hierarchy (the topmost, 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 13 and 14 into the skeleton of listing 12, we have completed our check for the "binary completeness" property.

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, and the need for privileged access to the internal representation of objects make it difficult to implement.

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

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

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 to not only implement it, but also 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.

References

[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.

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

[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.

[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`.