

On the Interaction of Partial Evaluation and Inheritance [★]

Gustavo Bobeff and Jacques Noyé

École des Mines de Nantes
4, rue Alfred Kastler - 44307 Nantes Cedex 3, France
{Gustavo.Bobeff, Jacques.Noye}@emn.fr

Abstract. In this paper, we consider a direct, source-to-source, specialization of Java programs. In this setting, specialization does not boil down to partially evaluating functions (here called methods) any longer. Indeed, specialized methods have also to be encapsulated into residual classes. We show that inheritance offers new specialization opportunities but that these opportunities are not so easy to benefit from because of some deep incompatibilities between specialization as partial evaluation and specialization as inheritance in standard object-oriented languages.

1 Introduction

Building generic programs has many benefits, for instance in terms of reuse, since such programs can be used in many different contexts. The interest of a generic program is however limited if genericity impairs specific uses of this program. Such a situation can be avoided by resorting to specialization techniques, such as partial evaluation [6, 7], in order to derive, from a generic program and a specific context, a program adapted to the context.

In a functional setting, a program can be seen as a set of functions, one of these functions being designated as the entry point of the program. As soon as some parameters of the entry point are known, a specialized program can be generated using partial evaluation. This specialized program is then a set of specialized functions, obtained through symbolic computation, unfolding and program point specialization [6, 7], from which all computations depending on the known parameters have been compiled away. This can easily be extended to a program with multiple entry points as well as to *module-oriented specialization*, introduced in Tempo [5], where entry points correspond to provided interfaces that should not be specialized, and required interfaces to external functions whose code is not available to the specializer.

When looking at the overall structure of a specialized program, dealing with imperative programming does not change the picture much. A program is now a set of procedures, functions with a hidden parameter and a hidden return value describing the state of the program. A specialized program is then a set of specialized procedures. It is not so difficult then to build a specializer for an imperative language either using a translation approach, based on an existing partial evaluator for a functional language [8], or a dedicated approach, where the translation is somehow buried either in a non-standard interpreter (on-line specialization) or in the analyzer (off-line specialization) [5].

The question is then: what happens when dealing with object-oriented programming as exemplified by Java [1]? This may, at first sight, look very much the same. An object-oriented program can be seen as a set of functions, called constructors (returning new objects) and methods (with, as first argument, the receiver of the invocation). Instance variables can simply be seen as constant methods. A specialized object-oriented program is then a set of specialized methods. Of course, these specialized methods cannot live in isolation, they are encapsulated in classes: the ones associated with their receiver or newly created class for the sake of encapsulation of the specialization. At this point, specialization as partial evaluation meets specialization as inheritance [11].

This paper shows that this is not as easy as it could have seemed by exploring some of the intricacies of using inheritance when specializing Java programs.

The problems related to the insertion of the specialized methods into classes, the class hierarchy and the use of polymorphism are introduced progressively. Section 2 describes specialization in a context where

[★] This work was partially funded by the European Commission in the FET Open Domain of the IST Programme under contract no. IST-1999-14191 (EASYCOMP - Easy Composition in Future Generation Component Systems). Any opinions, findings, and conclusions or recommendations expressed in this material are the authors and not necessarily reflect the view of the sponsors.

inheritance is not used at all, neither in the program to be specialized nor in the result of specialization. Section 3 introduces inheritance in both, as well as method overriding and changes in the class hierarchy. Section 4 discusses these results and concludes.

For the sake of simplicity, we will consider a simple subset of Java, excluding in particular inner classes and overloading (which can anyway be translated away) as well as interfaces, which means that the type of a non primitive object can be equated with its class.

2 Specializing Class-based Programs

2.1 The Basic Case

In the absence of inheritance, a method cannot be overridden and is therefore defined in a unique class (a method with the same name in another class is simply another method). In a first step, we do not consider the possibility of new classes. This means that any specialized version of a method has to be inserted in the class of the initial method.

Let us consider the classical example (see [1]) of a class modeling a point on a two-dimensional plane with a method `distance` computing the Euclidian distance between the point and another point, given as a parameter:

```
public class Point {
    double x;
    double y;

    public double distance(Point that) {
        double xdiff = x - that.x;
        double ydiff = y - that.y;
        return Math.sqrt(xdiff * xdiff + ydiff * ydiff);
    }
}
```

We assume that this class is part of a larger program that is specialized, leading to specific invocation contexts for the method `distance`.

A first possibility is that some contexts include knowledge on the value of the parameter `that`, for instance that the abscissa of `that` is equal to 0. This leads to a specialized version of `distance`, let us say, `distance_x0` to be inserted in the class `Point`. The resulting class, although still called `Point`, is actually a *specialized* (with respect to partial evaluation) version of the initial class `Point`.

Another possibility is that some other contexts include knowledge on the value of the receiver, for instance that its abscissa is equal to 0. This leads to another variant of `distance`, `x0_distance`, to be also inserted in the class `Point`. A more extreme case happens when both the abscissa and the ordinate of the receiver are known. In that case, the receiver is no longer necessary; the specialized version of `distance` is a class method. A specialized class `Point` including these three cases of specialized methods follows.

```
public class Point {
    <generic definition of Point>

    public static double x0_y0_distance(Point that) {
        double xdiff = - that.x;
        double ydiff = - that.y;
        return Math.sqrt(xdiff * xdiff + ydiff * ydiff);

    public double x0_distance(Point that) {
        double xdiff = - that.x;
        double ydiff = y - that.y;
        return Math.sqrt(xdiff * xdiff + ydiff * ydiff);

    public double distance_x0(Point that) {
```

```

    double xdiff = x ;
    double ydiff = y - that.y;
    return Math.sqrt(xdiff * xdiff + ydiff * ydiff);
}
}

```

A complication occurs when module-oriented specialization is taken into account and the class `Point` is used from outside the specialized module. In that case, it may be interesting to link the generic definition of a method to its specialized versions (which would be used directly from within the specialized module). For instance, the generic version of `distance` could be redefined in class `Point` in order to benefit from its specialized version `distance_x0`:

```

public double distance(Point that) {
    if (that.x == 0)
        distance_x0(that);
    else
        < generic definition of distance >
}

```

Of course, such a transformation is not always worthwhile as invocations of the generic method now incur some overhead.

Let us also note that inserting specialized methods in the class to be specialized is not possible when the structure of the class should not be changed, for instance in the presence of reflection. In that case, the specialized code can be replugged directly within the method to be specialized. Consider, for instance, the example above where the invocation of `distance_x0` would have been inlined.

2.2 Creating New Classes

Creating new specialized classes makes it possible to reduce the memory footprint of running instances. This happens when some objects turn out to have some invariant instance variables. Let us imagine that some points are initialized with their abscissa equal to 0 and remain so. What about creating a new class `Point_x0` which would correspond to these points and encapsulate all the variants of `distance` specialized for `x` equal to 0? If `x` is known at specialization time, it is no longer useful to keep it at execution time, it can disappear from the structure of `Point_x0`. Hence, the data size required to store an instance of `Point_x0` will be smaller than that of an instance of `Point`. Of course, as many variants of `Point` as invariant values of `x` can be generated.

The complication here comes from the creation of new types. A reference of type `Point` can only be specialized into a reference of type `Point_x0` if all its potential aliases may also be transformed in the same way. Here, inheritance would help and make objects of type `Point_x0` and `Point` compatible (see below).

In general, creating new classes instead of inserting specialized code in existing classes facilitates inspection of the specialized code. In the state of the technology, our experience is that looking at the specialized code is very often necessary in order to check the quality of the specialization¹. An alternative, suggested by U.P. Schultz [10], is to first generate specialized code into a *specialization aspect*, which is only woven into the initial class in a second step.

3 Specializing Object-oriented Programs

3.1 The Basic Case

Let us now consider programs using inheritance. As in the previous section, we shall first assume that specialized methods are inserted in existing classes. We shall also assume that overriding is not used in the program to be specialized.

Let us extend our previous class `Point` with two new classes, `3DPoint`, and `Colored3DPoint`:

¹ Actually, it can also be very helpful to debug the specializer!

```

public class 3DPoint extends Point{
    double z;
}
public class Colored3DPoint extends 3DPoint {
    int color;
}

```

If no new class is created, the question is simply to decide where to insert a specialized method. With inheritance, this depends on the type of the receiver, more precisely, on the *concrete (vs. declared)* type of the receiver. If no type information is available, a safe approximation has to be taken: the concrete type is a subtype of the declared type, which means that the specialized method must be inserted in the same class as its definition.

More precision can be obtained if type information is available, for instance the set of possible concrete types. If this set is a singleton, i.e. the concrete class of the object is known, then the specialized method can be inserted in this class. The point is that this approach allows specializing computations depending on types, e.g. uses of the operator `instanceOf`, which may happen during the invocation of the method being specialized. If the set of possible concrete types is not a singleton, the general strategy is to partition the set of possible types with one specialization generated for each element of the partition. When the elements of a partition include more than one type, a single specialized version of `distance` is generated and inserted in the most general type (i.e. common supertype) in this element. For example, if an object `p` is one of the types in `{Point, 3DPoint, Colored3DPoint}` and this set is partitioned as follows `{{Point}, {3DPoint, Colored3DPoint}}`, then a specialized method is defined in class `Point` and another specialized method is defined in class `3DPoint`. In the latter case `3DPoint` is the class that includes the specialized method because it is the supertype in the given element `{3DPoint, Colored3DPoint}`. The objective is, of course, to generate the optimal partition, making it possible to take the best advantage of specialization without creating duplicated code.

Let us consider a call to the method `distance` where the receiver is either a point or a colored 3D point, with the argument `that` known to be a colored 3D point such that $x = 0$. As the receiver may be a point, at the top of the hierarchy, and no more specialization opportunity can be gained by considering a colored 3D point, the best is to insert a single specialized version of the method in the class `Point`.

3.2 Specialization with Inheritance and Overriding

Let us now consider overriding in the program to be specialized. For instance, it makes sense to redefine the method `distance` in the class `3DPoint` so that the z-coordinate of the 3D point is taken into account, as follows:

```

public Class 3DPoint {
    double z;

    public double distance(Point that) {
        if (that instanceof 3DPoint) {
            double xdiff = x - (3DPoint)that.x;
            double ydiff = y - (3DPoint)that.y;
            double zdiff = z - (3DPoint)that.z;
            return Math.sqrt(xdiff * xdiff + ydiff * ydiff + zdiff * zdiff);
        } else {
            double xdiff = x - that.x;
            double ydiff = y - that.y;
            return Math.sqrt(xdiff * xdiff + ydiff * ydiff + z * z);
        }
    }
}

```

The implementation of the overriding method above allows one to conform to the *invariance* rule stated in method overriding in Java: the argument type cannot be changed in the overriding method. The particular example above corresponds to the *binary method problem* as explained by Bruce et al. in [2].

The insertion strategy discussed in section 3.1 is still valid, but the partitioning is constrained by overriding. In our example, if the parameter `that` is known to be a colored 3D point such that $x = 0$, and the receiver `p` is either a point or a 3D colored point, it does not make much sense any longer to insert a single specialized version of `distance` in the class `Point`. Indeed, this would require to build two specialized blocks of code corresponding each to one of the possible types of the receiver and glue them together within a conditional selecting the right block depending on the type of the receiver. This amounts to recoding dynamic dispatch, which then would be performed twice at runtime. The idea is rather to create two specialized versions of `distance`, sharing the same signature, in `Point` and `Colored3DPoint`, each version being based on the specific definition of `distance`. The code below shows the corresponding specialized methods, to be inserted in the classes `Point` and `Colored3DPoint`, respectively:

```
public double point_distance_x0(Point that) {
    double xdiff = x;
    double ydiff = y - that.y;
    return Math.sqrt(xdiff * xdiff + ydiff * ydiff);
}
public double colored3Dpoint_distance_x0(Point that) {
    double xdiff = x;
    double ydiff = y - (Colored3DPoint)that.y;
    double zdiff = z - (Colored3DPoint)that.z;
    return Math.sqrt(xdiff * xdiff + ydiff * ydiff + zdiff * zdiff);
}
```

In this case, the receiver types are used to define the insertion sites, and the knowledge about the arguments (including the receiver) constitutes the context in which the specialization is performed.

3.3 Creating New Classes

Let us finally add the possibility of creating new classes. It is now possible to come back to the simple situations described in section 2.1, keep the initial classes and relate the specialized classes to these classes via inheritance. This makes specialized instances type-compatible with generic instances, without duplicating code. Here, specialization as partial evaluation meets specialization as inheritance.

Unfortunately, this idea cannot be generalized. Let us, for instance, consider an invocation of `distance` such that the concrete type of the receiver is one of `{Point, Colored3DPoint}`, with $x=1$, and $z=0$ (when the receiver is a 3D colored point). Let us assume, moreover that the concrete type of the argument is `Point`. This requires specializing the definition of `distance` in both `Point` and `Colored3DPoint`, hence the creation of two specialized classes `SPoint` and `SColored3DPoint`. General typing considerations would require `SPoint` to be a subclass of `Point`, `SColored3DPoint` a subclass of `Colored3DPoint` and `SColored3DPoint` a subclass of `SPoint`, but, without multiple inheritance, `SColored3DPoint` cannot be a subclass of both `Point` and `SPoint`. Figure 1 shows the set of classes added into the original hierarchy.

One inheritance link has to be relaxed. In some situations one of the links may not be useful (this could be detected by generating typing constraints), otherwise one could imagine a general transformation strategy simulating multiple inheritance.

Note also that using inheritance only partly solve the problems encountered in module-oriented specialization. Indeed, it makes it possible to export an instance of `SPoint`, which could receive `distance` invocations, but, of course, dispatching on the arguments or on the receiver's state has still to be done in the generic interface method (see section 2.2).

As a final point, let us note that there are situations where it makes sense to use *interclassing* (see [9]), i.e. make the specialized class a superclass of the initial class. One case is when part of the receiver state is invariant. It is then still possible to remove the invariant fields from the specialized class, as was suggested in section 2.2 for the specialized class `Point_x0` of class `Point`, representing the points of constant abscissa 0. Of course, using subclassing (as suggested in [10]) is correct but fails to specialize structures as well as behaviors.

4 Conclusion

This paper has presented some intricacies of mixing partial evaluation and inheritance. On the one hand, a simple scheme consists of inserting the specialized methods in their receiver's class, on the other hand

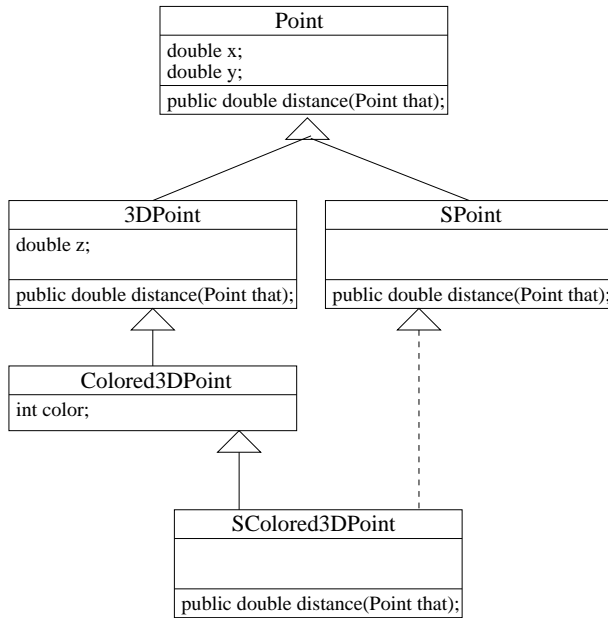


Fig. 1. Resulting hierarchy after the specialization.

this simple scheme may create problems in the presence of reflection and does not take the most out of partial evaluation. Another possibility is to create proper *specialized classes*, which can then specialize structure as well as behavior.

However, it turns out that connecting these new classes to the existing hierarchy is not easy, especially in the presence of single inheritance. This is due to the fact that specialization as inheritance and specialization as partial evaluation are very different in nature, in spite of a similar feel. This can be illustrated by the fact that partial evaluation can reduce the structure of a class whereas subclassing extends it. Another striking point is that a specialized method obeys a covariant rule whereas an inherited method obeys a contravariant rule (invariant in Java). This suggests that taking specialization as partial evaluation into account right from the start during language design may be worth. This also suggests to look more closely at multimethods [3, 4], which, thanks to multiple dispatch and its use of covariance to override parameters driving method selection, could facilitate the introduction of specialized methods.

Acknowledgment

We would like to thank Mathias Braux for its initial contribution to our work.

References

1. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 2nd edition edition, 1997.
2. K. Bruce, L. Cardelli, and G. Castagna. On binary methods. *TAPOS - Theory and Practice of Object Systems*, 1(3):221–242, 1995.
3. Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhauser, Boston, MA, 1997.
4. C. Clifton, G. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00, Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, Minneapolis, Minnesota, USA, October 2000. ACM Press. ACM SIGPLAN Notices, 35(10).
5. C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noy, S. Thibault, and E.N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys*, 30(3), September 1998.
6. N.D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.
7. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, 1993.

8. B. Moura. *Bridging the gap between functional and imperative languages*. PhD thesis, Universit de Rennes I, Rennes, France, 1997.
9. P. Rapicault and A. Napoli. Evolution d'une hirarchie de classes par interclassement. In R. Godin and I. Borne, editors, *LMO 2001 - Langages et modles objets*, pages 215–230, Le Croisic, France, January 2001. Herms. L'Objet, 7(1-2).
10. U.P. Schultz. *Object-Oriented Software Engineering Using Partial Evaluation*. PhD thesis, Universit de Rennes I, December 2000.
11. A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.