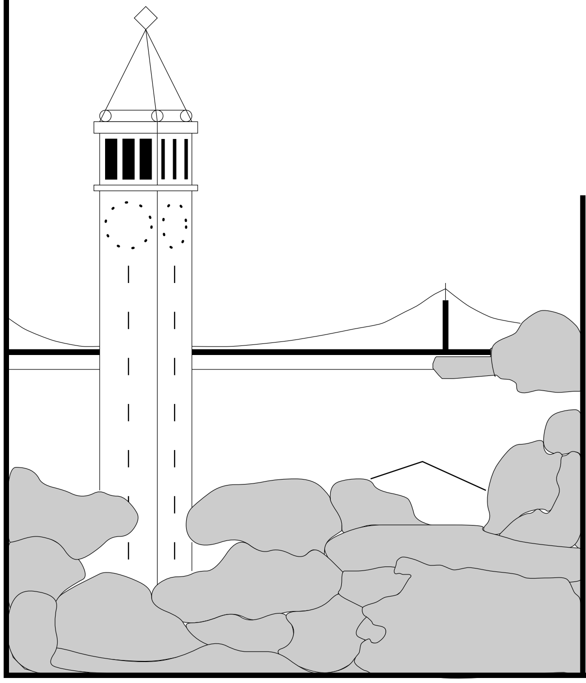
Type-Safe Compilation of Covariant Specialization: A Practical Case

John Boyland *Giuseppe Castagna*



Report No. UCB/CSD-95-890

November 1995

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Type-Safe Compilation of Covariant Specialization: A Practical Case

John Boyland*

Computer Science Division, EECS
University of California
e-mail: boyland@cs.berkeley.edu

Giuseppe Castagna

CNRS, LIENS-DMI
École Normale Supérieure
e-mail: castagna@dmi.ens.fr

November 1995

Abstract

Despite its lack of type safety, some typed object-oriented languages use *covariant* specialization for methods. In this work, we show how one may modify the semantics of languages that use covariant specialization in order to improve their type safety. We demonstrate our technique using O_2, a strongly and statically typed object-oriented database programming language which uses covariant specialization. We propose a modification to the O_2 compiler that adds code to correct previously ill-typed computations that arise from the use of covariant specialization. The modification we propose does not affect the semantics of those computations without type errors. Furthermore, the new semantics of the previously ill-typed computations is defined in a very "natural" way and ensures the type safety (w.r.t. covariance) of the program. Since the solution consists of a conservative backward-compatible modification of the compiler, it does not require, unlike other solutions, any modification of existing O_2 code. Our solution is based solely on a type-theoretic analysis and thus is general. Therefore, although this paper applies it to a particular programming language, the same ideas could easily be applied to other languages that uses covariant specialization.

1 Introduction

Strongly-typed object-oriented languages impose conditions in order to statically ensure type safety for method overriding. In particular, in the presence of subtyping, type safety requires that the type of the result of the overriding method be a subtype of the type of the result of the overridden one, and that the types of the parameters of the overridden method are subtypes of those of the corresponding parameters in the overriding one. In type theory, this rule is said to be *covariant* in the result type (since it preserves the sense of the subtype relation) and *contravariant* on the parameters' types (since it inverts the sense of the relation). Since the parameter behavior is taken as representative, one speaks in this case of *contravariant specialization*.

Although type-safe, contravariant specialization restricts the flexibility of a language considerably. For this reason, some object-oriented language designers put expressiveness above strict type safety and instead adopt the more flexible *covariant specialization* rule, which requires exactly the opposite subtyping relation for parameters, namely that the type of the parameters in the overriding method are subtypes of those of the corresponding parameters in the overridden one. It would seem

*The work of John Boyland was supported in part by Advanced Research Projects Agency grant MDA972-92-J-1028, and by NSF Infrastructure grant CDA-9401156. The content of this paper does not necessarily reflect the position or the policy of the U. S. Government, and no official endorsement should be inferred.

that one cannot combine covariant specialization and inheritanced-based subtyping in a type-safe manner.

In the last years, several solutions have been suggested to retain some of the flexibility of covariant specialization in a type-safe framework. These solutions propose new languages constructs (e.g. [Cas95a]), or new relations on types (e.g. [Bru94]), or the use of different typing techniques for the methods (e.g. [BHJL86]): see [BCC+96] for a wide review. However, none of them can be directly applied to programs written in languages that use covariant specialization. To use these solutions, one has to throw away the programs written with covariant specialization, and to rewrite them from scratch either in a different language or in a extended version of the old language. In many practical cases, such as in large databases, a complete rewrite of the existing code would be much too expensive, even if feasible. Such cases are not merely hypothetical, since the world's third most popular commercial object-oriented database system, O_2, uses covariant specialization.

Despite its great practical implications, nearly no work to our knowledge has tried to handle the problem of ensuring the type safety of existing code that uses covariant specialization. A notable exception to this is given by the work done for the language Eiffel [Mey91, Mey96]. Eiffel uses covariant redefinition, but uses an additional check, the "system validity" check to detect possible type errors admitted due to covariant specialization. Currently, the definition of Eiffel gives a link-time (i.e. global) data-flow analysis for this purpose, but Meyer has proposed a new check that can be done locally. Due to its complexity, the current definition is not incorporated into any widely available compiler. The new check is much less complex, but may possibly prove too strict. Either analysis guarantees type-safety but can only reject programs with potential type errors.

In this article we propose another solution, different from Eiffel's, but in the same spirit, since it can be directly applied to the existing code. What we show is that it is not necessary to rewrite some programs or to discard any of them, since by slightly changing the interpretation of the programs, we can ensure complete type safety. Furthermore, this change will affect only those computations that would otherwise have undefined semantics due to a type error.

In other words, type theoretic research so far addresses the question: *How should I have written my program to obtain type safety and also the flexibility of covariant specialization without using it?* The analysis for Eiffel answers the question: *Is my program type-safe even if I used covariant specialization?* The problem addressed by this article is substantially different. This work answers the question: *How can I change the semantics of my programs that use covariant specialization in order to ensure type safety?*

Of course, the new semantics must satisfy some minimal requirements:

1. The new semantics must be conservative over type-safe programs, in the sense that the programs that worked safely even with covariance must not be affected by the change. In other words, programs that have no type problems must have, with the new semantics, the same meaning as with the old semantics.
2. The new semantics of ill-typed computations[1] must be somehow "natural", in the sense that the definition of the new semantics must take into account why an ill-typed situation has been reached and what could reasonably be its intended meaning. The new semantics must recover from the errors, not simply hide them.

We describe our solution which fulfils these requirements by applying it to O_2, a strongly-typed object-oriented database programming language [BDK92] that uses covariant specialization together with run-time checks to ensure type safety. The choice is not arbitrary since the question we address is acutely pertinent in the presence of persistent data. In particular, we sketch how to modify the standard O_2 compiler to make O_2's covariant specialization rule type-safe. Our solution does

[1]By an *ill-typed computation* we mean a computation that includes at least one *ill-typed application*: an application of a function or a method to an argument whose type is not compatible with the parameter type (the domain) of the function/method.

not require any source code to be changed, neither does it require the "bases" to be recompiled; a "schema" recompilation will suffice.[2]

Although in this paper, we apply our ideas to a particular programming language, our solution is general. Being based solely on type theoretic observations, it can be applied to other languages that uses covariant specialization, to yield a type-safe semantics, or it can be used to define a *new* type-safe object-oriented language that provides covariant specialization.

The work is organized as follows. Section 2 introduces O_2 and shows why covariant specialization is not type-safe. Section 3 reviews the work on multi-methods done in [Cas95a]. These ideas are used in Section 4 to illustrate the solution we propose here in the case of single inheritance. Section 5 shows that the naive extension of the previous solution to the multiple inheritance doesn't work and uses this analysis to describe a more sophisticated solution that does work. We close our presentation by comparing our work with other solutions presented in the literature.

2 The Language O_2 and Covariance

O_2 is a strongly-typed, object-oriented database programming language. The most important aspect of O_2's type system, as with most typed object-oriented languages, is its *subtyping relation*. Subtyping allows a value of a *subtype* to be used anywhere a value of the *supertype* is expected. The use of this relation enhances the flexibility of the language since it allows the values of a given type to use operations originally defined for a different type.

However, the O_2 type discipline is not safe; type errors may occur at run-time, even if a program has successfully passed static type checking.[3] In particular, type errors may arise due to the use of covariant specialization. We demonstrate the problem by an example:[4]

```
class Point
   type tuple (x:real,
               y:real)
   method equal(p:Point):boolean
end;

method body equal(p:Point):boolean in class Point
   {return( (self->x == p->x) && (self->y == p->y) );};


class ColorPoint inherit Point
   type tuple(c:string)                /* x and y are inherited from Point */
   method equal(p:ColorPoint):boolean
end;

method body equal(p:ColorPoint):boolean in class ColorPoint
   {return( (self->x == p->x) && (self->y == p->y) && (self->c == p->c) );}
```

We have two classes that represent two-dimensional points and colored points respectively. The class `Point` has two instance variables `x` and `y` of type `real` and a method for the message `equal` that returns a boolean when applied to an argument of class `Point`. Its definition compares the instance variables of the receiver of the message (`self`) with the ones of the argument. The class `ColorPoint` is obtained by inheritance from `Point`. To represent the color, it adds the instance variable `c` to

[2] In O_2 jargon, the *schema* of a database is the description of the structure and the behavior of the data; it essentially consists of class and method definitions. A collection of data (objects and values) whose structure and behavior conforms to the definitions in a schema is called *base*. A schema may be created and modified without a reference to a particular base and it can be shared by several bases, while a base must always refer to a single schema.

[3] In classical languages, the definition of a type error is well-known. In addition, in object-oriented languages, sending a message to an object that cannot respond to it is considered a type error.

[4] We use version 4.0 of O_2 and we specify the methods in O_2C.

those inherited from `Point`. It redefines (or, *overrides*) the method for the message `equal`, in order to take into account the color of the point.

In O_2, an overriding method must satisfy the following covariant specialization rule:

1. It must have the same number of parameters as the method it overrides.
2. The type of each parameter must be a (possibly improper) subtype of the type of the corresponding parameter in the overridden method.
3. The type of the result must be smaller than or equal to the type of the result of the overridden method.

The classes `Point` and `ColorPoint` satisfy these conditions. The rule is quite intuitive, but it is not type-safe, as can be demonstrated by adding the following method to the class `Point`:

```
method break_it(p:Point) in class Point;
method body  break_it(p:Point) in class Point {p->equal(self);}
```

This method[5] sends the message `equal` to the parameter `p` and with `self` as argument. It is easy to verify that, although the command `(new Point)->break_it(new ColorPoint)` is well typed according to the rules of O_2 (the command `new` creates a new instance of the specified class), it leads to a type error: its execution produces the ill-typed application of the `equal` method in `ColorPoint` to an argument of class `Point`.[6]

The problem arises from the covariance introduced in condition (2). Type theory states that a class C_2 is a subtype of another class C_1 (namely, that objects of C_2 can be type-safely used wherever an object of C_1 is expected) only if the methods defined in C_1 can be type-safely replaced by those of C_2 [Car88]. If a method for a message is inherited in C_2, then no typing issue is raised, since the method is the same for C_1 and C_2. If the method is redefined in C_2, type safety is ensured only if the new method can replace the method defined in C_1 *in every context*. In particular, the method in C_2 must be able to handle at least the same arguments that can be passed to the method in C_1. In our example with points, the method for `equal` in `Point` accepts arguments of type *Point* while the one in `ColorPoint` does not. The type error in the call of `break_it` is caused by the fact that the method for `equal` in `ColorPoint` is applied to an argument of type *Point*, a legal argument for the method it overrides.

More formally, the problem is that the type $ColorPoint \to boolean$ of the equal method in `ColorPoint` is not a subtype of $Point \to boolean$.[7] Indeed recall that by definition of subtyping, $S \to T \leq S' \to T'$ holds only if every function f of type $S \to T$ can be used where a function g of type $S' \to T'$ is expected. This implies that the result of f (of type T) must be able to replace the result of g (of type T'), i.e. $T \leq T'$; and that f can handle any argument g can, i.e. $S' \leq S$. Thus the orientation of the inequality of the arrow is inverted on the domains: the subtyping rule is "contravariant" in the left component of the arrow.

In conclusion, to ensure type safety in an object-oriented language (such as used in O_2), one must require that the types of the parameters of an overriding method are *supertypes* of those of the parameters of the overridden method. Otherwise the subclass cannot be considered a subtype of its superclass, and thus objects of the subclass cannot be used where objects of the superclass are expected. This rule disallows a useful specialization of `equal` in `ColorPoint`, because there is no way to compute the color component of an argument known only to be a *Point*. Going ahead with the specialization would mean that the class `ColorPoint` would not be a subtype of the class

[5] In O_2, methods can be declared outside class definitions, as in the example above. This allows one to add new methods to an existing class or to redefine the old methods.

[6] Note that the kind of error produced by break_it is not so hard to generate. Every function that needs to test the equality of two parameters of type Point may generate a type error of this sort.

[7] According to the O_2's notation, the types of the methods for equal in ColorPoint and Point are, respectively, $ColorPoint \times ColorPoint \to boolean$ and $Point \times Point \to boolean$ since the type of the receiver (i.e. of self) is included among the parameters of the method. In order to simplify the exposition, we will omit the receiver's type from the type of the methods. This omission does not affect the core of our discussion. For more details on receiver types, see [Cas95a].

`Point`, and thus that colored point could not be used where points are expected. Thus, it appears one is left with the choice between a useful subtype relation and a useful specialization.

The designers of O_2 have preferred to give up some type safety, and to adopt covariant specialization. In the next section, we show that there is way to allow covariant specialization without sacrificing type safety.

3 Multi-methods

Our solution for making covariant specialization type-safe uses multi-methods. Multi-methods appear in the CLOS language [DG87] and their typing issues have been studied in [ADL91, CGL95, CL94]. However, none of these approaches can be directly applied to the case of O_2, since they do not retain the notion of method encapsulation: there is no privileged receiver —as in O_2— to which a message is sent. Thus in this article, we utilize a different kind of multi-methods, those studied in [Cas95a, MHH91] (and, implicitly, also in [Ing86]). These multi-methods allow the use of multiple dispatching (i.e., the possibility that the selection of a method is also based on other arguments of the message) even in presence of a privileged receiver. (A detailed comparison between the two kinds of multi-methods can be found in [BCC+96]).

In particular, it is possible to have type-safe covariant specialization when using this second kind of multi-method with late-binding [Cas95a]. We demonstrate the idea with the `Point/ColorPoint` example introduced in the previous section. The problem with the definition of equal in `ColorPoint` is that it cannot handle arguments of type *Point*. The intuition of our solution is that some code can be added to compensate for this deficiency. The original method definition is executed for arguments of type *ColorPoint* while the new code handles arguments of type *Point*. In practice, this corresponds to changing the definition of `ColorPoint`:

```
class ColorPoint inherit Point
   type tuple(c:string)            /* x and y are inherited from Point */
   method equal(p:Point):boolean,
          equal(p:ColorPoint):boolean
end;

method body equal(p:Point):boolean in class ColorPoint
   {return( (self->x == p->x) && (self->y == p->y) );};

method body equal(p:ColorPoint):boolean in class ColorPoint
   {return( (self->x == p->x) && (self->y == p->y) && (self->c == p->c) );};
```

There are now two different *method branches* (*branches* for short) for the message `equal` in `ColorPoint`. Both method branches together can be considered a single *multi-method*.[8] When the message `equal` is sent to an object of type *ColorPoint*, one of the two method branches is selected for execution according to the type of the argument. If the argument is of type *Point*, the first branch is used; if it is of type *ColorPoint*, or of a subtype, the the second method branch is used. The selection of the branch is performed at run-time after the argument of the message has been fully evaluated. This is a crucial feature that differentiates multi-methods from C++'s overloaded methods (where the selection is performed at compile time), and ensures the appropriate use of the covariant specialization. Consider the fragment `p->equal(self)` in the code of `break_it` in `Point`; `self` has static type *Point*, therefore if the selection of the method were performed at compile time, the first branch of `equal` would be always executed, even for

```
(new ColorPoint)->break_it(new ColorPoint)
```

Here, we would expect the two points to be compared also in their color component.

[8]These kinds of multi-methods are called *multivariant* in [MHH91] and *encapsulated multi-methods* in [BCC+96] in order to distinguish them from CLOS's multi-methods.

What then are the rules that ensure the type safety of this approach? Recall that type safety is guaranteed only if every overriding method possesses a subtype of the type of the method it overrides. The type of a multi-method is the set of the types of its various branches. The subtyping relation between sets of types (not to be confused with the type of sets) states that one set of types is smaller than another set of types only if for every type contained in the latter, there exists a type in the former smaller than it.[9] This fits the intuition that one multi-method can be replaced by another multi-method of different type, when for every method branch that can be selected in the former, there is one in the latter that can replace it (for subtyping, an ordinary method is considered to be a multi-method with just one branch: its type is a singleton).

In the new version of the point example, the type of the multi-method for `equal` in `ColorPoint` is $\{Point \to boolean, ColorPoint \to boolean\}$. By the rule above we have

$$\{Point \to boolean, ColorPoint \to boolean\} \leq \{Point \to boolean\}$$

Since $\{Point \to boolean\}$ is the type of the method associated with `equal` in `Point`, the subtyping condition is fulfilled and type safety ensured.

More generally, if one wishes to override a method[10] and covariantly specialize the type of its parameters, it is necessary also to add another branch to handle the arguments that could be passed to the overridden method.

Adding multi-method branches to achieve type safety does not require a large amount of programming: in case of single inheritance, the number of branches sufficient to override covariant methods is independent from both the size and the depth of the inheritance hierarchy; it is always equal to two. Indeed, when a multi-method of type $\{S_1 \to T_1, \ldots, S_n \to T_n\}$ is applied to an argument of type U, the branch executed is the one defined for the type $S_j = \min_{i=1..n}\{S_i \mid U \leq S_i\}$. Thus a single branch with a sufficiently high (in the inheritance hierarchy) type may handle all the remaining arguments that are not handled by the specializing code. For example, suppose that we further specialize our point hierarchy by adding dimensions, each with their own `equal` methods:

```
class Point3D inherit Point
     type tuple(z:real)
     method equal(p:Point3D) : boolean
end;

class Point4D inherit Point3D
     type tuple(w:real)
     method equal(p:Point4D) : boolean
end;
```

and so on, up to dimension n. The new classes form a chain in the inheritance hierarchy. Each class covariantly overrides the `equal` method inherited from its superclass. It may appear to be necessary to add $n-2$ more branches to each class Point$n$D in order to guarantee safety; however, one additional branch with a parameter of type *Point* suffices. This branch will handle all other possible points. For example, in the case of $n = 4$, one could define

```
class Point4D inherit Point3D
     type tuple(w:real);
     method equal(p:Point4D) : boolean,
            equal(p:Point) : boolean
end;

method body equal(p:Point4D):boolean in class Point4D
   {return( (self->x == p->x) && (self->y == p->y)
         && (self->z == p->z) && (self->w == p->w) );};
```

[9] The subtyping relation for function types is, of course, the contravariant rule defined before.
[10] We assume here that overriding a multi-method overrides all its branches.

```
method body equal(p:Point):boolean in class Point4D
    {return( p->equal(self) ); }
```

Type safety stems from the fact that the subtyping condition is satisfied.[11]

In conclusion, if we extend the syntax of O_2 with multi-methods, we can have type safety and covariant specialization. Every time we perform a covariant specialization in a class with a single direct superclass, it will suffice to add one (and one only) branch to handle all the arguments "inherited" from the superclasses, although one may wish to add multiple branches for semantic reasons.

This still does not solve the problem we want to address with this work and which concerns existing O_2 code. In the rest of this paper, we show that a compiler can automatically add branches that make a program type-safe. Note that the type safety is obtained without any modification of the source code.

A more formal treatment of multi-methods can be found in [Cas95a]. For the formal type system and the proof of its type safety see [CGL95]. Examples of type safe use of multi-methods in programming languages with a privileged receiver are proposed in [MHH91] and [Cas95b].

4 Single inheritance

We first describe our solution as it applies when there is only single inheritance. We generalize this solution to multiple inheritance hierarchies in Section 5.

For this section, we use $\varsigma(C)$ to denote the unique *direct superclass* of C, i.e. the class that follows the `inherit` keyword in the definition of C. Besides this notation, we need to introduce some more O_2 syntax.

4.1 The `@` notation

In O_2, it is possible to invoke the method attached to a specific class by the following `@`-notation: r`->`C`@`m. This invokes the method attached to the message m in the class C instead of the one attached in the class of the receiver r, provided that the latter is a subclass of C. In particular, in this work we use the `@`-notation in commands of the form `self->`A`@`m which inside a proper subclass of A dispatches the message m to `self` but it starts the search for the method associated to m from the class A. This mechanism is different from the `super` mechanism of, say, Smalltalk: suppose that the class A defines a method for the message m, that the class B is defined by inheritance from A (i.e. $A = \varsigma(B)$) and that B inherits from A the method for m. Then `self->`A`@`m always begins the search for the method for m from the class A (thus, in this case it is equivalent to `self->`m), while `super->`m begins the search from the superclass of the class containing the method, namely from the superclass of A (therefore, in this example it is equivalent to `self->`$\varsigma(A)$`@`m). In other words, the `@`-notation expresses absolute addressing, whereas `super` is a relative addressing.

4.2 The solution

We start the presentation of our solution in the simplified case of methods with just one parameter. Suppose we have two classes C_0 and C_n, $C_n < C_0$, defined by the following program

[11] In general, according to the subtyping rule for multi-methods, if $T_1 \geq T_2 \geq \ldots \geq T_n$ and $S_1 \geq S_2 \geq \ldots \geq S_n$ then we have the following type inequalities:

$$\{S_n \to T_n, S_1 \to T_{n-1}\} \leq \ldots \leq \{S_{i+1} \to T_{i+1}, S_1 \to T_i\} \leq \{S_i \to T_i, S_1 \to T_{i-1}\} \leq \ldots \leq \{S_1 \to T_1\}$$

The declarations of the classes for points are a special case of this, where $S_1 = Point$, for $i \in [3..n]$ $S_{i-1} = PointiD$, and for $i \in [1..n\text{-}1]$ $T_i = boolean$.

```
class C_0
    type ...                /* The type is not important */
    method m(x:S_0):T_0
end;

class C_n inherit C_{n-1}   /* C_{n-1} is a subclass of C_0 */
    method m(x:S_n):T_n
end
```

where $S_n < S_0$ and $T_n \leq T_0$. Suppose also that the message m has not been redefined in the inheritance hierarchy between C_n and C_0. Our solution has the compiler add the following method branch to C_n:

```
method m(x:S_0):T_0 in class C_n;
method body m(x:S_0):T_0 in class C_n { return(self->C_{n-1}@m(x)); }
```

The program is now type-safe since by the subtyping rule of multi-methods $\{S_0 \to T_0, S_n \to T_n\} \leq \{S_0 \to T_0\}$, that is, the type of the (multi-)method in C_n is smaller than the type of the method in C_0 it overrides. Note also that since m has not been redefined between C_0 and C_n, the type of `self->`C_{n-1}`@`m`(x)` is the type of the result of the method in C_0 (that is, T_0) and that therefore the body of the new method branch conforms to its signature. This same branch must also be added to all subclasses of C_n which *invariantly override* m, by which we mean that the parameter type of m remains the same in the redefinition.

In the case in which m has been covariantly redefined multiple times in the hierarchy between C_0 and S_0, our solution has the compiler add multiple branches. For example, suppose that for a class C_i, $C_n < C_i < C_0$, the method for m has been redefined with signature $S_i \to T_i$, where $S_n < S_i < S_0$, but that it is not redefined elsewhere between C_0 and C_n. Then the compiler will add to the class C_i a method branch of type $S_0 \to T_0$, and will add two method branches to C_n: one of type $S_0 \to T_0$ and another of type $S_i \to T_i$.

```
method m(x:S_i):T_i in class C_n;
method body m(x:S_i):T_i in class C_n { return(self->C_{n-1}@m(x)); }

method m(x:S_0):T_0 in class C_n;
method body m(x:S_0):T_0 in class C_n { return(self->C_{n-1}@m(x)); }
```

By the multi-method subtyping rule we have

$$\underbrace{\{S_0 \to T_0, S_i \to T_i, S_n \to T_n\}}_{\text{type of } m \text{ in } C_n} \quad \leq \quad \underbrace{\{S_0 \to T_0, S_i \to T_i\}}_{\text{type of } m \text{ in } C_i} \quad \leq \quad \underbrace{\{S_0 \to T_0\}}_{\text{type of } m \text{ in } C_0}$$

The subtyping condition is fulfilled since the type of every overriding (multi-)method is a subtype of the type of the (multi-)method it overrides. This guarantees type safety.[12] Both added method branch bodies have the same code; they are type-safe because `self->`C_{n-1}`@`m is of multi-method type. We will return to this fact after describing the algorithm more precisely.

Viewed as a top-down process over the inheritance hierarchy, our solution descends the inheritance hierarchy looking the first class containing a covariant redefinition of a given message. When it finds one, it adds to the class a branch that points to the last definition of the message, and continues to descend the hierarchy looking for the next class where the message is redefined (either covariantly or invariantly). If the redefinition is invariant, it adds to the class all the branches that were added to the last definition; if the redefinition is covariant, it adds the same branches plus one branch that points to the last definition.

The general task of the compiler can be described as follows

[12] For those familiar with [Cas95a] note that O_2's covariance rule ensures that the overloaded types are well formed.

Algorithm 1 For every class C, for every message m overridden in C with type $S \to T$, and for every superclass C_i (for which m has type $S_i \to T_i$) where m is covariantly redefined in the direct subclass C_{i+1} to $S_{i+1} \to T_{i+1}$ (that is $C \leq C_{i+1} < C_i$, $\varsigma(C_{i+1}) = C_i$, $S \leq S_{i+1} < S_i$), add the following method branch:

```
method m(x:Si):Ti in class C;
method body m(x:Si):Ti in class C { return(self->ς(C)@m(x)); }
```

■

The intuition underlying this rule is that the compiler adds branches to handle all possible arguments that are not handled by the original redefinition. There is a one-to-one correspondence between the superclasses of a class (including itself) that covariantly redefine a method, and method branches one has to add to it.

Note that the method body declarations are well typed (this can be proved by induction on the depth of the inheritance hierarchy). Note also that all the added branches perform the same thing: they search up in the inheritance hierarchy for the first definition of a method that can handle the argument. In practice, the compiler can collapse all the added branches into a single branch to be selected when the argument is of a supertype of that in the covariant method. This branch simply performs a lookup in the inheritance hierarchy. This observation is used in Section 4.5 to give an implementation of our solution. Since our methods already have multi-method type, with a little abuse of notation, we could use multi-method typed branches as well. For the second example with C_n, C_i and C_0, the addition would look like

```
method m:{S0 → T0, Si → Ti} in class Cn
method body m(x) in class Cn { return(self->ς(Cn)@m(x)); }
```

Intuitively, such a (multi-)method is selected when the type of the argument is a subtype of S_i (in which case the result will be of type T_i), or is a type included between S_0 and S_i (in which case the result is of type T_0)[13].

More generally, when a message m has been covariantly overridden, the compiler adds a single branch (possibly of multi-method type) whose type is the type of the superclass' definition of the method (ignoring any arrow type whose domain is the same as the one defined for this class). This can be expressed by reformulating the previous algorithm in the following implementation-oriented way:

Algorithm 1 (Implementation-Oriented Version) For every class C, for every message m that is redefined in C whose type is not a subtype of the method defined in some superclass, add the following branch:

```
method m : typeof(ς(C)@m)/{S} in class C
method body m(x) in class C { return(self->ς(C)@m(x)); }
```

where S is the parameter type of m in C, *typeof*(.) is a meta-operator that returns the type of a (multi-)method, and $T/\{S\}$ denotes the multi-method type T in which a possible arrow of domain S has been erased. ■

It is not necessary that the syntax of the language being made safe actually allow these new multi-methods, since the method is added by the compiler as a part of the implementation, as shown in Section 4.5. This notation is only used to express implementation at the source level.

[13]Note that it is not necessary to restate the type of the branch in the body declaration since there will be only one such branch.

4.3 Naturalness

In the introduction, we state that our solution is natural. First of all, note that the semantics of well-typed programs is not modified: indeed all the (non-functional) expressions have, after the compiler's completion, the same type as before the completion. Thus in the case of well-typed programs, the original method definitions are always selected. For example, if we compare a *ColorPoint* with another *ColorPoint*, after the completion, the method written by the programmer in the `ColorPoint` class is executed. We give a new semantics only to those computations that produce a run-time type error. Because of covariant specialization, it may happen that a method is applied to an argument that it cannot handle. In that case, an added method branch is executed: it ascends the inheritance hierarchy to look for the last definition of that method that can handle the argument (it knows that one exists). Thus, we have an intelligent compiler that inserts the code the programmer has forgotten to write, thus ensuring type safety for covariant specialization. The naturalness of our solution is given by the fact the method executed is always the most specialized one written by the programmer for the arguments in the call. Of course, no solution for adding multi-methods automatically can be as natural as one in which the multi-methods are hand-written, but our solution is the most natural fix that can be done automatically. Furthermore, the new semantics takes into account the reason for an ill-typed application, namely the application can be ill-typed only if the receiver is statically considered an object of a superclass of its actual class. Our solution has the method lookup mechanism ascend the inheritance hierarchy from the receiver's dynamic class (where it would otherwise stop) towards the static class, looking for a definition that supports the arguments given.

4.4 Multiple-argument methods

It is straightforward to extend this solution to methods with multiple parameters. As before, the algorithm leads to the most specific method definition being used:

Algorithm 2 For every class C, for every message m with k parameters overridden in C with type $(S^1 \times \cdots \times S^k) \to T$, and for every superclass C_i (for which m has type $(S_i^1 \times \cdots \times S_i^k) \to T_i$) where m is covariantly defined in the direct subclass C_{i+1} (that is, $C \leq C_{i+1} < C_i, C_i = \varsigma(C_{i+1}), (S_{i+1}^1 \times \cdots \times S_{i+1}^k) < (S_i^1 \times \cdots \times S_i^k)$), add the following method branch:

method m(x1:S_i^1,...,xk:S_i^k):T' in class C;
method body m(x1:S_i^1,...,xk:S_i^k):T' in class C { return(self->$\varsigma(C)$@m(x1,...,xk)); }

■

The implementation-oriented version of Algorithm 1 can be extended similarly so that the solution works by adding a single (multi-method) branch.

The formal justification of the type safety of this second algorithm is straightforwardly obtained by using cartesian products to type multi-argument methods.

4.5 Implementation

The solution admits at least two different implementation techniques.

One could change the compiler to use the observation that all the method branches added to a method definition have the same body, thus applying the implementation-oriented versions of Algorithms 1 and 2. The (new) compiler "marks" methods needing extra branches, and compiles these methods differently. Either extra code may be added which tests the argument types, or a description of the types may be used at run-time by the message dispatcher. In any case, if a marked method does not handle its arguments, the dispatch mechanism searches for a new method definition starting from the superclass. If this method is also marked then the argument types must be checked again and so on. Our solution ensures that as long as a method is marked, there is another method for the same message higher up in the inheritance hierarchy that can handle more argument types.

It also ensures that if a method definition is not marked, then it can handle all arguments that the static type system permits. Effectively, a marked method overrides a previous definition for only *some* of its arguments.

A more conservative, but less efficient way to implement the algorithms is to simulate multi-methods in the source language, using the technique proposed by Ingalls [Ing86]. Ingalls' simulation, offered in the context of single-dispatching languages such as Smalltalk-80 [GR83], uses a second message dispatch to obtain the dynamic selection on an extra argument. Every multi-method can be simulated by several normal method dispatches. After the first dispatch, only the type of the receiver is known. After the second dispatch, the type of the first argument is known, and so on. The realization of this method for the example from Section 3 is presented in Appendix A. The reader can try to follow the execution of `p->equal(q)`: for all the possible combinations of `p` and `q` (both arguments instances of `Point`, `p` instance of `Point` and `q` instance of `ColorPoint`, and so on) the code executed is always the same as that executed with the multi-methods defined in Section 3. Also, note that all the methods have the same types for `Point` and `ColorPoint`. This means that covariant specialization is not used and therefore, as expected, the class definitions are type-safe.

The advantage of using Ingalls' simulation is that it can be implemented by a preprocessor, rather than changing the core of the standard O_2 compiler. The preprocessor would transform the covariant specialization of a method with one argument into one dispatching method plus one more for the original method and one more for each additional branch determined by Algorithm 1 in any of its subclasses. This advantage, however, must be weighed against several disadvantages. The method-marking implementation is more efficient both in terms of space (there is no code duplication) and of time (the overhead to select the branch is much more prominent in Ingalls' simulation). Furthermore, Ingalls' simulation is neither modular nor incremental. If we compiled some classes using the marking implementation and later added some new subclasses, we do not need to recompile the first ones. With an implementation based on Ingalls' simulation, every new covariant redefinition would require the recompilation of every class that implements a method for the message. Another problem is that, as shown in [BCC+96], Ingalls' simulation does not work properly when the result types of overriding methods differ from the methods they override. This problem can be fixed using parametric polymorphism, but since O_2 does not provide this kind of polymorphism, it would be necessary to bypass the type-checking phase when compiling pre-processed code.

Finally, note that although marked methods require extra checking to implement the added method branches, the current O_2 compiler already generates code to perform these checks, in order to detect the run-time type-errors caused by covariant specialization.

5 Multiple inheritance

Multiple inheritance presents several obstacles to our solution as defined for single inheritance. The most pertinent is that we do not have a privileged superclass; therefore the notation $\varsigma(.)$ is undefined. To put it otherwise, there is no longer a standard place from where to start the search for a method definition for an ill-typed application.

A second problem concerns the application of multi-methods. In Section 3, we said that if a multi-method of type $\{S_1 \to T_1, \ldots, S_n \to T_n\}$ is applied to an argument of type U, the branch executed is the one defined for the type $S_j = \min_{i=1..n}\{S_i \,|\, U \leq S_i\}$. With multiple inheritance, some conditions are needed to ensure that the set $\{S_i \,|\, U \leq S_i\}$ has a least element.

These two problems are connected. Indeed, if we generalize the algorithm given for single inheritance in a straightforward way, we run into pathological cases that break naturalness and type safety.

In this section, we first study the cases in which the straightforward extension of the solution for single inheritance fails. Next, we define an extension of the multi-method syntax and behavior that allows us to generalize the single inheritance solution to multiple inheritance in a type-safe and natural way.

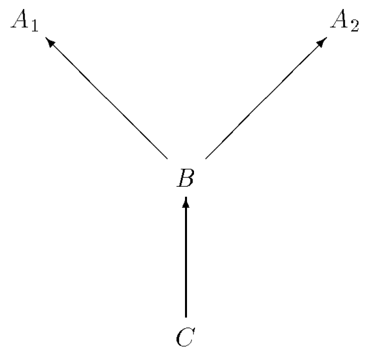
5.1 Pathological cases

Consider four classes A_1, A_2, B, and C, with C defined by inheritance from B and B defined by multiple inheritance from A_1 and A_2. This situation is graphically represented in the figure below. Consider now the following program, where $T \leq T_1, T_2$ (as before, we omit the `type` declarations):

```
class E
   method m(x:A1):T1;
   method n(x:B):T1
end;

class F
   method m(x:A2):T2;
   method n(x:B):T2
end;

class G inherit E, F
   method m(x:C):T;
   method n(x:C):T
end;
```



The class G inherits from E and F. Since the message m is defined in both E and F, O_2 requires the programmer to redefine the message inside G (otherwise there would be a conflict in the choice of the method to inherit). In G, the method for m covariantly overrides the two previous methods. In order to make this redefinition type-safe, one must add new method branches to handle potential arguments of type A_1, A_2 and B. For the arguments of type A_1, our solution has the compiler use the method defined in E; it will add to G the following method branch:

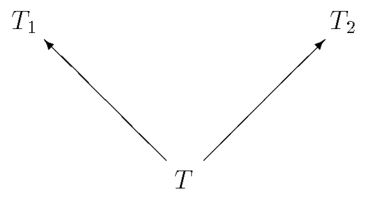
`method body m(x:`A_1`):`T_1` in class `G` {return(self->`E`@`m`(x))}`

Similarly for arguments of type A_2 it will add

`method body m(x:`A_2`):`T_2` in class `G` {return(self->`F`@`m`(x))}`

Note also that the compiler *must* add a branch for B, otherwise the type of the multi-method in G would be $\{A_1 \to T_1, A_2 \to T_2, C \to T\}$ and for an argument of type B, there would be no branch with least parameter type. So the compiler also adds a branch for B. But what shall the compiler use as the body for this branch? There are only two possible choices: either it uses the method in E or it uses the method in F, but both choices are equally good (or bad). The return type for the method branch will reflect this choice, either T_1 or T_2. A similar problem arises when trying to add a method branch for n to handle arguments of type B. If one wants to use multi-methods as defined in Section 3 then the only way out is to perform arbitrary choices that break the type safety of our solution and, perhaps more seriously, its naturalness.

To see how arbitrary choices break type safety, consider again the example of m. Imagine that T_1 and T_2 are incomparable (as in the figure to the right), and that for the body of the code for B in G, the compiler has arbitrarily chosen the method in E. Let b be an object of class B and consider the expression o`->m(`b`)`. If the static type of o is F, then the static type of this expression is T_2. But if the dynamic type of o is G (which is feasible since $G \leq F$) then the method inserted for B in G is selected and, as a consequence of the arbitrary choice, the method in E is executed. Thus the dynamic type of the expression is T_1 which is incompatible with the static type T_2. This inconsistency may lead to a run-time type error. In this case, the natural method to call would be the one in F, but of course, choosing that method for the branch could also lead to a type error. No automatic addition of method branches (as defined so far) can give complete type safety.



All these pathological cases can only occur in the following situation: some class has two incomparable superclasses each defining a method for the same message and the domains of these two methods have a common subtype not handled by the redefinition in the class. For some particular configurations of the result types of the two methods, there exists a type-safe and natural choice, but in the remaining cases one cannot avoid an arbitrary choice.

This arbitrary choice breaks the naturalness of the solution and, in particular, when it is necessary to make an arbitrary choice between methods with incomparable result types, a run time type error may occur.

In conclusion, the analysis performed for single inheritance is no longer sufficient to solve the problem with multiple inheritance.

5.2 The intuition of our solution

To give a solution for multiple inheritance we revisit the causes of type errors due to the covariant specialization.

Recall that covariant specialization can lead to ill-typed applications only in the case that an object has a (non-trivial) superclass of its true class as its static type. Consider the following fragment in the context of the example in the Section 5.1 (where t_1 is some message that can be sent only to objects of the T_1 class, and t_2 is legal only for T_2):

```
o2 B b = new B;                    /* create an object of the B class */
o2 G g = new G;                    /* create an object of the G class */
o2 E e = g;                      /* treat g as an object of the E class */
o2 F f = g;                      /* treat g as an object of the F class */
(e->n(b))->t1;                                  /* Application #1 */
(f->n(b))->t2;                                  /* Application #2 */
```

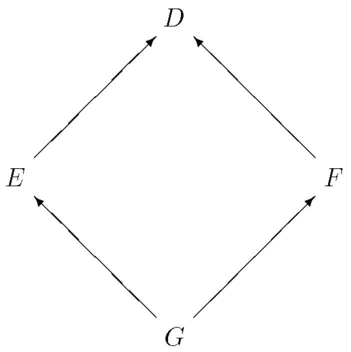
First, note that both applications are ill-typed in unmodified O_2, despite being legal under covariant specialization. Note also that the only way to avoid type errors in both applications is to select a different method for each, despite the fact that both applications have the same receiver and argument objects.

Our multi-method solution works for single inheritance by ascending the inheritance hierarchy starting from the dynamic class of the receiver object. If the method definition for the class can handle the parameters, it is used, otherwise the direct superclass is examined, and so on. This solution works because eventually the search will reach the static class of the receiver which must have a method definition that can handle the arguments. This solution does not work for multiple inheritance because when the search must continue from a class with multiple direct superclasses, it does not "know" which of the direct superclasses is on a path to the static class of the receiver, and so can get "lost." The intuition behind our solution for multiple inheritance is to use the static class of the receiver to direct the ascension of the hierarchy.

So the first idea for a solution for multiple inheritance is to limit the search of the method to that part of the inheritance hierarchy that is included between the static and the dynamic type of the receiver. Applying this restriction in the fragment above, the method for the receiver `e` will not be looked for in F and the one for `f` will not be looked for in E.

This idea is enough to avoid type errors, but it does not remove the need for arbitrary choices. Arbitrary choices interfere with naturalness and, more seriously, with the predictabilty and understandability of the semantics.

Consider again the example at the beginning of Section 5.1 and suppose that both E and F are subclasses of some class D (as in the figure on the side) which has methods for both m and n. Consider further that we are sending the message m or n to a receiver with static type D and dynamic type G. If the type of the argument is B, the methods of E and F are equally applicable and only an arbitrary choice can choose between them. Predictability can be restored in a type-safe manner by choosingthe method defined for D, but at the expense of some naturalness, since the methods in E and F are not considered.[14]



The solution we present for multiple inheritance uses both these ideas to assure type-safety and (a certain degree of) naturalness, namely:

1. The search of the method is restricted to the portion of the inheritance hierarchy included between the static type and the dynamic type of the receiver.

2. If this portion of the inheritance hierarchy includes a zone in which a pathological cases may happen, this zone is skipped by the search.

Note that the decision to skip a zone is based on the static type of the receiver. For example, for m or n messages, if the receiver's dynamic type is G then the class E must be skipped if the receiver's static type is D, but E must be searched if the static type of the receiver is E.

To see the solution in a different way, consider the inheritance hierarchy between the static and the dynamic class of the receiver. The hierarchy forms a directed acyclic graph. There are several paths that lead from the dynamic to the static class. Consider the set of nodes that belong to every such path. Because of the acyclicity of the graph, this set is totally ordered (w.r.t. the subtyping relation). Therefore if we consider only the classes of this set, we have a single-inheritance-like hierarchy going from the dynamic class to the static class of the receiver. Our solution applies the single inheritance solution from Section 4 to this hierarchy.

The multi-methods described in Section 3 do not suffice to implement this strategy. A further extension and semantics must be given that permit the selection to take into account the receivers' static type. Before defining this extension, some further notation is needed.

5.3 Notation

A *chain* from a class C up to another class C_0 is a set of comparable classes $\{C_{n-1}, \ldots, C_0\}, n > 0$, where $C = C_n < C_{n-1} < \ldots < C_0$. If each C_i is the direct superclass of C_{i+1}, we call it a *path*. There are many chains in a multiple inheritance hierarchy. Thus, we pick a particular chain (denoted $\kappa(C, C_0)$), which is defined as the set of all classes that appear in every path from C to C_0, or more precisely:

$$\kappa(C, C_0) \equiv \{C' \mid C < C' \leq C_0, \forall T.\, (C < T \leq C_0 \Rightarrow T \leq C' \vee T \geq C')\}$$

We distinguish the least class in this chain, C_{n-1}, as $\varsigma_{C_0}(C)$, the *direct join superclass* of C for C_0, and the greatest class (other than the superclass itself), C_1, as $\varsigma'_C(C_0)$, the *direct join subclass* of C_0 for C:

$$\begin{aligned} \varsigma_{C_0}(C) &\equiv \min \kappa(C, C_0) \\ \varsigma'_C(C_0) &\equiv \max \kappa(C, C_0) \backslash C_0 \end{aligned}$$

[14] Of course, predictability could be also be achieved by, say, searching from the first parent that is on a path to the static class of the receiver. Such a definition, however, makes the semantics dependent on the order of the parents in the inheritance clause which can be confusing and inelegant.

Intuitively, the direct join superclass of a class C for a class A is the next class up the inheritance hierarchy that is comparable with every other class between A and C. Note that a direct join superclass (there may be several) is not necessarily a direct superclass and a direct join subclass is not necessarily a direct subclass. Note that in the case of single inheritance, the direct join superclasses are simply the direct superclasses (that is, $C_0 > C \Rightarrow \varsigma_{C_0}(C) = \varsigma(C)$)

The method branches added in the enhanced solution are restricted to calls depending on the static class of the receiver. The notation

method ... in class C < C'

is used to specify that this method branch should only be considered if C' is in the chain from C to the static class A of the receiver, that is, $C' \in \kappa(C, A)$. The intuition is that the body of this method has been defined in the class C' and therefore it should be executed (that is, C' can be searched) only if C' is on all paths going from the dynamic to the static class of the receiver.

Note that in the case of single inheritance, the branch is applicable when the receiver is of the class C' or any superclass of C'. Since the typing rules of O_2 ensure that branches added by our solution are only needed in such situations, a restriction of this form is vacuous.

5.4 The solution

Our solution works very similarly to the case of single inheritance, a method branch is added for each definition in a superclass which is covariantly redefined in a subclass. The difference if that the added branch is restricted to apply only to certain static classes of the receiver.

For the example in the Section 5, the compiler would add the following methods

method body m(x:A_1):T_1 in class G < E {return(self->E@m(x));}
method body m(x:A_2):T_2 in class G < F {return(self->F@m(x));}
method body n(x:B):T_1 in class G < E {return(self->E@n(x));}
method body n(x:B):T_2 in class G < F {return(self->F@n(x));}

Note that this completion avoids the arbitrary choices imposed by the pathological cases: if an object of class G receives the message m or n with an argument of class B then the method will be selected on the base of the static type of the receiver (if the static type is a superclass of D —see Section 5.2— then the algorithm will add other branches that handle it).

In general, the solution (for the single argument case) works as follows:

Algorithm 3 For every class C, for every message m overridden in C with type $S \to T$, and for every superclass C_i (for which m has type $S_i \to T_i$) where m is covariantly redefined in the direct join subclass C_{i+1} to $S_{i+1} \to T_{i+1}$ (that is, $C \leq C_{i+1} < C_i, C_{i+1} = \varsigma'_C(C_i), S \leq S_{i+1} < S_i$), add the following method branch:

method m(x:S_i):T_i in class C < C_i;
method body m(x:S_i):T_i in class C < C_i { return(self->$\varsigma_{C_i}(C)$@m(x)); }

■

Notes:

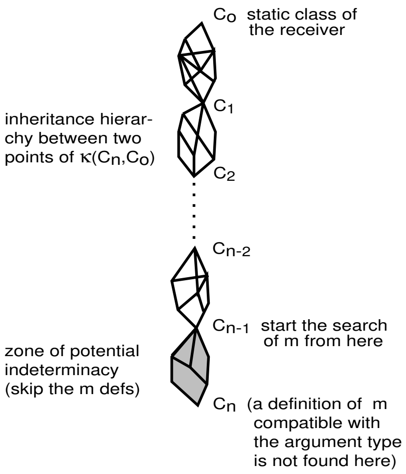
First note that in the case of single inheritance, this algorithm yields exactly the same method branches as Algorithm 1 aside from vacuous restrictions. More generally, if C has a single superclass, then despite any multiple inheritance among its ancestors, its direct superclass is always its direct join superclass.

Secondly, note that the type safety of the method branch bodies is proved by induction down the inheritance hierarchy. The class $\varsigma_{C'}(C)$ is the first in the chain to C', and if it is not equal to C', it will have a similar method branch added to it. In general, $\varsigma_{C'}(C)$@m has multi-method type.

Lastly, and most importantly, for every receiver with static class A, the set of applicable method branches includes only the original method definition in C and branches for classes along the chain

$\kappa(C, A)$. By including the original definition, the solution ensures that well-typed applications have the same semantics as previously. The covariant specialization rules for O_2 guarantee that the domains of these branches will form a chain themselves, and a minimum applicable branch is ensured.

So what is the exact behavior of this new algorithm? Imagine that the message m has been sent to some object whose static type was C_0, but whose dynamic type was C_n. The systems looks for the method in class C_n but because of covariant specialization, the method for m cannot handle the actual argument of the method. Our algorithm adds the branches that make the system continue the search up in the inheritance hierarchy. This search continues from the direct join superclass of C_n for C_0 (C_{n-1} in the figure on the right). Note that in the inheritance hierarchy included between C_n and C_{n-1}, (darker in the figure) there may be some definitions for m able to handle the argument of the method, but also that precisely this possibility makes this particular part of the inheritance hierarchy a place (better, *the* place) of potential indeterminacies, where an arbitrary choice might be required. Therefore the search skips this dangerous zone.



Since the search goes up through subtypes of the static type of the receiver eventually reaching this class, there will always be a method that can handle the argument. Note that in the limit case, where $\varsigma_{C_0}(C_n) = C_0$, the algorithm simply executes the method that was statically predicted for the message, that is, the one defined for the static class of the receiver.

The solution given here does maintain a degree of naturalness: it may not perform the most specialized method for the argument (e.g. any method that is defined in the grey zone) but it never has to make an arbitrary choice. This property makes it behave predictably. In the case of single inheritance, the semantics of the multi-method solution is that each covariant specialization is only overriding part of the definition of its direct superclass. This concept is clear and natural. Here, we are overriding the definition of the direct join superclass, in order to avoid choosing between two direct superclasses. Note that we have done nothing but to expand the solution we gave for single inheritance; indeed the single inheritance solution is nothing but the special case in which all inheritance hierarchies between two classes of the chain are replaced by a single link (in single inheritance $\varsigma_{C_0}(C_n) = \varsigma(C_n)$).

Thus in summary, this solution is good, not merely because it avoids type errors, but more importantly because it makes a natural and predictable completion of the class.

As with the first algorithm, it is straightforward to extend Algorithm 3 to methods with multiple arguments by considering the arguments to be a single argument with cartesian product type.

5.5 Implementation

There are at least two possible implementation possibilities for the enhanced solution.

The first implementation possibility involves compiling all message sends to implicitly also send some indication of the static class of the receiver. As with the marking method for single inheritance, all methods that have additional branches added are marked. If the arguments are not handled by the method definition, then the method must compute the direct join superclass of the class which defined the method for the static class passed implicitly. This computation could involve a table lookup compiled into the marked method. Once a new class was determined, the method lookup mechanism could proceed from there. A compiled table lookup requires that every method in a class be recompiled if there is any change in the inheritance hierarchy above the class. Alternately, if the structure of the hierarchy was available at runtime, the computation of the direct join superclasses could be deferred to this point.

The second implementation possibility involves having a different method lookup table depending on the static class of the receiver. For example, if an object of class G were given static class F, it would be given a different set of method definitions. The compiler can detect these occurrences statically and arrange that the correct method definition table be used. The original definitions would not need to be marked, because the static typing rules of O_2 would prevent any type errors. A compiled method for a particular static non-trivial superclass would have to check its arguments and then if the defined method was unable to handle them, would defer to the appropriate method of the direct join superclass (known statically). The method tables for a class tailored for two different superclasses on the same $\kappa(.,.)$ chain could be identical, thus avoiding massive code duplication. This implementation possibility does not require method sends to implicitly send the static class of the receiver and thus should not impact the efficiency of the dispatch mechanism. However, this possibility would require a class to be recompiled if any class above it in the hierarchy was changed.

These two possibilities both share the advantage that they would not require the data to have a new representation, and thus large bases would not need to be recompiled.

6 Comparison with other works and conclusion

We want to stress, once more, that our main concern for this work is to define a technique that could be applied to the existing programs without requiring any modification of them. This is a crucial characteristic in the field of large databases, where the rewriting of the code would be much too expensive, even if feasible. Indeed, there already exist several solutions that handle the problem of covariant specialization: some very "ad hoc" like the run-time handling of exceptions [ABDS95], others much more elegant and formal such as a relation that replaces subtyping [Bru94], or the use of less precise types for the methods [BHJL86]. In the same spirit, we could have further developed Section 3 and defined an extension of O_2 to handle multi-methods. But all these solutions require at least the modification of the existing code, if not the use of a totally different paradigm. Therefore they cannot be strictly compared with the solution we propose here.

The only other solution that, like ours, does not require any modification of existing code is Eiffel's one. The former definition of system validity [Mey91] would use global data-flow analysis to ensure that arguments to procedures such as `break_it` in Section 3 can only be passed expressions of dynamic type *Point*. The newer definition [Mey96] would disallow routines like `break_it` outright. If a compiler using one of these rules detects a violation, it can only issue warnings (to little effect) or reject the program. It cannot fix the (potential) error.

Our solution takes a different tack. Rather than disallowing potentially ill-typed applications, our solution patches them so that they are well-typed. The patch is executed only upon an actual ill-typed application, and could optionally generate a warning message at this point as well. The added code uses the conditions under which application occurred to choose the most appropriate method definition. To use a medical metaphor, Eiffel performs a much more accurate screening process to search for type errors, while our solution addresses more the prophylaxis by vaccinating risky situations. It is important to stress that the solution affects only the definition of the methods. All the existing bases are unaffected and can be used as before.

Another important remark is that our solution does not merely fix existing code, it also provides a fix that works for possible evolutions of the system. Every use of covariant specialization is a potential time bomb that can explode a long time after that the code has been written. This may happen, for example, because a new version of a library is released or merely because the run-time values of the data are different. Eiffel's solution blocks all possible explosive situations (thus the use of a new library version may occasion that an old program no longer type-checks). Our solution instead defuses the time bombs.

In other words, our solution gives a predictable and type-safe semantics for covariant specialization. Covariant specialization may be seen as *partial overriding* of a previous definition, contrasting with the *full overriding* by an invariant (or contravariant) specialization. Our solution could be incorporated into a new language that permits type-safe covariant specialization, possibly along with contravariant specialization as well.

Incidentally, our paper also proposes a possible extension of O_2 to incorporate multi-methods. This idea deserves more extensive treatment, especially concerning modularity issues, which constitute a peculiar problem of multi-methods (see [Coo91]). The great advantage of multi-methods is that the programmer can choose the definition to be used in the case of failure of the covariant specialization, instead of delegating this choice to the compiler.

Last but not least, our analysis is founded on well-established type-theoretic bases, so that the correctness of our solution is formally proved and type-safety is guaranteed.

Acknowledgments

Giuseppe Castagna is very grateful to Kim Bruce, Luca Cardelli, Gary Leavens, Scott Smith and Benjamin Pierce. The joint paper [BCC+96] and the several e-mails exchanges were crucial to the development of this work. In particular, Gary gave the pointer to Ingalls' simulation and Scott was the first who noticed that this simulation might be used to implement our ideas. William Maddox and Tim A. Wagner read early drafts of this paper and provided useful criticism.

References

[ABDS95] E. Amiel, M.-J. Bellosta, E. Dujardin, and E. Simon. Type-safe relaxing of schema consistency rules for flexible modelling in OODBMS. *VLDB journal*, 1995. To appear.

[ADL91] R. Agrawal, L. DeMichiel, and B. Lindsay. Static type checking of multi-methods. *SIGPLAN Notices*, 26(11):113–128, 1991. Proceedings of OOPSLA'91.

[BCC+96] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1996. To appear. Currently available: `ftp://ftp.ens.fr/pub/dmi/users/castagna/binary.ps.Z`

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Implementing an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, 1992.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. *ACM SIGPLAN Notices*, 21(11):78–86, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[Bru94] K.B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. A previous version can be found in Semantics of Data Types, LNCS 173, 51-67, Springer-Verlag, 1984.

[Cas95a] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[Cas95b] G. Castagna. A meta-language for typed object-oriented languages. *Theoretical Computer Science*, 151(2):297–352, November 1995. Extended abstract in the Proceedings of the *13th Conference on the Foundations of Software Technology and Theoretical Computer Science*; Lecture Notes in Computer Science number 761, December 1993.

[CGL95] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995. A preliminary version was presented at the *1992 ACM Conference on LISP and Functional Programming*, San Francisco, June 1992.

[CL94] Craig Chambers and Gary T. Leavens. Typechecking and modules for multi-methods. In *OOPSLA'94*, 1994.

[Coo91] William R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, 1991.

[DG87] L.G. DeMichiel and R.P. Gabriel. Common Lisp Object System overview. In Bézivin, Hullot, Cointe, and Lieberman, editors, *Proc. of ECOOP '87 European Conference on Object-Oriented Programming*, number 276 in Lecture Notes in Computer Science, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.

[Ing86] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In Norman Meyrowitz, editor, *OOPSLA '86 Conference Proceedings, Portland, Oregon, September 1986*, volume 21(11) of *SIGPLAN Notices*, pages 347–349, November 1986.

[Mey91] Bertrand Meyer. *Eiffel: The Language.* Prentice-Hall, 1991.

[Mey96] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 2nd edition, to appear 1996.

[MHH91] W. B. Mugridge, J. G. Hosking, and J. Hamer. Multi-methods in a statically-typed programming language. In Pierre America, editor, *ECOOP '91 Conference Proceedings, Geneva, Switzerland*, volume 512 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

A Ingalls' simulation for the `Point/ColorPoint` problem

```
class Point
   type tuple (x:real,
               y:real)
   method equal(p:Point):boolean,
   method equalPoint(p:Point):boolean,
   method equalColorPoint(p:ColorPoint):boolean
end;

method body equal(p:Point):boolean in class Point
   {return(p->equalPoint(self));};

method body equalPoint(p:Point):boolean in class Point
   {return( (self->x == p->x) && (self->y == p->y) );};

method body equalColorPoint(p:ColorPoint):boolean in class Point
   {return( self->equalPoint(p) );};
```

```
class ColorPoint inherit Point
   type tuple(c:string)       /* x and y are inherited from Point and the */
end;                          /* signature of the methods does not change */

method body equal(p:Point):boolean in class ColorPoint
   {return( p->equalColorPoint(self) );};

/* ColorPoint inherits equalPoint from Point */

method body equalColorPoint(p:ColorPoint):boolean in class ColorPoint
   {return( (self->x == p->x) && (self->y == p->y) && (self->c == p->c) );};
```