Covariance and Contravariance: Conflict without a Cause

GIUSEPPE CASTAGNA
C.N.R.S.

In type-theoretic research on object-oriented programming, the issue of "covariance versus contravariance" is a topic of continuing debate. In this short note we argue that covariance and contravariance appropriately characterize two distinct and independent mechanisms. The so-called contravariance rule correctly captures the *subtyping* relation (that relation which establishes which sets of functions can replace another given set *in every context*). A covariant relation, instead, characterizes the *specialization* of code (i.e., the definition of new code which replaces old definitions *in some particular cases*). Therefore, covariance and contravariance are not opposing views, but distinct concepts that each have their place in object-oriented systems. Both can (and should) be integrated in a type-safe manner in object-oriented languages. We also show that the independence of the two mechanisms is not characteristic of a particular model but is valid in general, since covariant specialization is present in record-based models, although it is hidden by a deficiency of all existing calculi that realize this model. As an aside, we show that the $\lambda\&$-calculus can be taken as the basic calculus for both an overloading-based and a record-based model. Using this approach, one not only obtains a more uniform vision of object-oriented type theories, but in the case of the record-based approach, one also gains multiple dispatching, a feature that existing record-based models do not capture.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*object-oriented languages*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*

General Terms: Theory, Languages

Additional Key Words and Phrases: Object-oriented languages, type theory

1. INTRODUCTION

In type-theoretic research on object-oriented programming, the issue of "covariance versus contravariance" has been, and still is, the core of a heated debate. The discussion goes back, in our ken, to at least 1988, when Lécluse, Richard, and Vélez used "covariant specialization" for the methods in the O_2 data model [Lécluse et al. 1988]. Since then, it has been disputed whether one should use covariant or contravariant specialization for the methods in an object-oriented language. The fact that this debate is still heated is witnessed by the excellent tutorial on object-oriented type systems given by Michael Schwartzbach at the last POPL con-

This work was partially supported by grant no. 203.01.56 of the Consiglio Nazionale delle Ricerche, Comitato Nazionale delle Scienze Matematiche, Italy, to work at LIENS.
Author's address: LIENS, 45 rue d'Ulm 75005 Paris, France; email: castagna@dmi.ens.fr.

ference [Schwartzbach 1994]: in the abstract of his tutorial Schwartzbach fingers the "covariance versus contravariance" issue as a key example of the specificity of object-oriented type systems.

In this short note we argue that the choice between covariance and contravariance is a false problem. Covariance and contravariance characterize two completely distinct mechanisms: subtyping and specialization. The confusion of the two made them appear mutually exclusive. In fact, covariance and contravariance are not conflicting views but distinct concepts that can be integrated in a type-safe formalism. Finally, we argue that it would be an error to exclude either of them, since then the corresponding mechanism could not be properly implemented.

This result is clear in the model of object-oriented programming defined by Giuseppe Longo, Giorgio Ghelli, and the author in Castagna et al. [1995]; it is already present in Ghelli's seminal work [Ghelli 1991], and it is somehow hidden in the work on `OBJ` [Goguen and Meseguer 1989; Jouannaud et al. 1992; Martí-Oliet and Meseguer 1990]. In these notes we want to stress that this result is independent of the particular model of object-oriented programming one chooses, and that covariance and contravariance already coexist in the record-based model proposed by Luca Cardelli in Cardelli [1988], and further developed by many other authors (see the collection [Gunter and Mitchell 1994] for a wide review of the record-based model).

The article is organized as follows. In Section 2, we recall the terms of the problem and we hint at its solution. In Section 3, we introduce the overloading-based model for object-oriented programming and give a precise explanation of *subtyping* and *specialization*. We then show how and why covariance and contravariance can coexist within a type-safe calculus. We use this analysis to determine the precise role of each mechanism and to show that there is no conflict between them. Section 4 provides evidence that this analysis is independent of the particular model by revealing the (type-safe) covariance in the record-based model. Section 5 contains our conclusions and the golden rules for the type-safe usage of covariance and contravariance.

We assume that the reader is familiar with the objects-as-records model of object-oriented programming and is aware of the typing issues it raises.

The presentation is intentionally kept informal: no definitions, no theorems. It is not a matter of defining a new system but of explaining and comparing existing ones: indeed, all the technical results have already been widely published.

2. THE CONTROVERSY

The controversy concerning the use of either covariance or contravariance can be described as follows. In the record-based model, proposed by Luca Cardelli in 1984 [Cardelli 1988], an object is modeled by a record, whose fields contain all the methods of the object and whose labels are the corresponding messages that invoke the methods. An object can be specialized to create a new object in two different ways: either by adding new methods —i.e., new fields— or by redefining the existing ones —i.e., overriding old methods.[1] A specialized object can be used

[1] It is unimportant in this context whether the specialization is performed at object level (delegation) or at class level (inheritance).

wherever the object it specializes can be used. This implies that method overriding must be restricted if type safety is desired. A sufficient condition to assure type safety (at least for method specialization) is the requirement that each field can be specialized only by terms whose types are "subtypes" of the type of the field.

The core of the covariance/contravariance controversy concerns methods that have a functional type. The subtyping relation for functional types is defined in Cardelli [1988] as follows:

$$\text{if} \quad T_1 \leq S_1 \quad \text{and} \quad S_2 \leq T_2 \quad \text{then} \quad S_1 \to S_2 \leq T_1 \to T_2$$

If we consider the arrow "$\to$" as a type constructor, then, borrowing the terminology of category theory, "$\to$" is a functor covariant on the right argument (since it preserves the direction of "$\leq$") and contravariant on the left argument (since it reverses the direction of "$\leq$"). Taking the behavior of the left argument as characteristic, this rule has been called the *contravariant rule*.[2] If an arrow "$\to$" is covariant on the left argument (i.e., if in the rule above the sense of the first inequality is reversed), then type safety is lost. With this modified rule, it is quite easy to write a statically well-typed term that produces a run-time type error.

Despite its unsoundness, covariant specialization has its tenacious defenders, and not without cause. (Eiffel [Meyer 1991] and the O_2 system [Bancilhon et al. 1992], for example, use covariant specialization.) The contravariant rule, besides being less intuitive than the covariant one, is the source of many problems. The most surprising one appears with binary methods and can be exemplified as follows. Consider an object o_1 of a given type T, from which we create another object o_2 of type S via specialization. Suppose we have defined a method *equal* for these objects, which compares the object at issue with another object of the same type. This *equal* method has type $T \times T \to Bool$ for the object o_1 and $S \times S \to Bool$ for the object o_2. In the record-based approach, the fields labeled *equal* will have the type $T \to Bool$ in o_1 and $S \to Bool$ in o_2 since the method *belongs* to the object, and thus it already knows its first argument, usually denoted by the keyword `self`. If the contravariant rule is used, the type associated with *equal* for S-objects is not a subtype of the type for *equal* in T-objects. Thus, in order to have type safety, one must not use o_2 as a specialization of o_1. In other words, S must not be a subtype of T. This is quite unintuitive. Imagine that you have objects for real numbers and for natural numbers. As soon as a number can respond to a message that asks it whether it is equal to another number, then a natural number can no longer be used where a real number is expected! Furthermore, experience with O_2 (which is the third most sold object-oriented database management system in the world) shows that the unsoundness of the type-checker has not caused many problems in practice. Thus, many people prefer to give up type safety and use the covariant subtyping rule for specialization. The general conclusion is that one has to use contravariance when static type safety is required, but otherwise covariance is more natural, flexible, and expressive.

[2] Although *co-contravariant rule* would be a better name for this rule, we prefer to adopt the name in usage in the object-oriented community. Therefore, in the rest of the article we will use "contravariance," "contravariant rule," and "contravariant specialization" to denote the co-contravariant behavior of the arrow.

The viewpoint of the covariance advocates and the one of the contravariance advocates are both very appealing, and yet they seem totally incompatible. However, there is a flaw in the comparison made above: covariance in O_2's (nearly) overloading-based model is compared with contravariance in the record-based model. The difference between the two models is in the type of the parameter `self`, which appears in the former model but disappears in the latter one (see the type of *equal* in the previous example). The conclusion drawn above is wrong because, as we will show in the next two sections, it does not take into account the disappearance of this type from one model to the other. Thus, we will proceed by studying both covariance and contravariance, first in the overloading-based model (Section 3) and then in the record-based one (Section 4). We will show that both covariance and contravariance can be used in a way that guarantees type safety. To achieve this end, we need not impose any further restrictions, just point out what the two concepts serve for.

Before proceeding, let us fix some terminology. Recall that each object has a set of private operations associated with it, called *methods* in Smalltalk [Goldberg and Robson 1983], Objective-C [Pinson and Wiener 1992], and CLOS [DeMichiel and Gabriel 1987], and *member functions* in C++ [Stroustrup 1986]. These operations can be executed by applying a special operator to the object itself: the object is the receiver of a message in Smalltalk and Objective-C, the argument of a generic function in CLOS, and the left argument of a dot selection in C++. In order to simplify the exposition we refer to all of these different ways of selecting a method as "message-sending" operations: the message is the name of the generic function in CLOS and the right argument of dot selection in C++. Additionally, a message may have some parameters. They are introduced by *keywords* in Smalltalk and Objective-C; they are the arguments of an n-ary generic function in CLOS, [3] and they are surrounded by parenthesis in C++.

Now (and here we enter the core of our discussion) the type (or class) of the actual parameters of a message may or may not be considered in the run-time selection of the method to execute. For example in CLOS, the type of *each* argument of a generic function is taken into account in the selection of the method. In C++, Smalltalk, and Objective-C, no arguments are considered: the type of the receiver alone drives the selection.[4] In the following sections, we formally show that given a method m selected by a message with parameters, when m is overridden, the parameters that determine the (dynamic) selection must be covariantly overridden (i.e., the corresponding parameters in the overriding method must have a lesser type). Those parameters that are not taken into account in the selection must be contravariantly overridden (i.e., the corresponding parameters in the overriding method must have a greater type).

[3] Strictly speaking, it is not possible in CLOS to identify a privileged "receiver" for the generic function.

[4] The use of overloading in C++ requires a brief remark. C++ resolves overloading at compile time, using static types; dynamic method look-up does not affect which code is executed for an overloaded member function. At run-time, the code for such functions has already been expanded. For this reason, the overloading in C++ is quite different from the one we describe in Section 3.

3. THE FORMAL STATEMENT

In this section we give a formal framework in which to state precisely the elements of the problem intuitively explained in the section before. We first analyze the problem in the overloading-based model [Castagna et al. 1995] since in this model the covariance-contravariance issue has a clearer formalization. In Section 4 we will discuss the record-based model.

The idea in the overloading-based model is to type messages rather than objects. More precisely, we assume that messages are special functions composed of several (ordinary) functions: the methods. When a message is sent to an object of a given class, the method defined for objects of that class is selected from among those composing the message. The object is passed to the selected method, which is then executed. This model is quite natural for programmers used to languages with *generic functions* such as CLOS or Dylan [Apple Computer Inc. 1992] (generic functions of CLOS coincide to our special functions), while its understanding requires an effort of abstraction to programmers used to other object-oriented languages that group methods inside their host objects —as formalized in the record-based model— instead of inside the messages

However, if we ignore implementation issues, these two ways of grouping methods, either by object or by message, are essentially equivalent, since they are simply two different perspectives of the same scene. This is also true from the type-theoretic point of view, as suggested by Section 4.

Class definitions are used to describe objects. A class is generally characterized by a name, a set of instance variables, and a set of methods. Each method in a class is associated to a message. In the overloading-based model we further assume that classes are used to type their instances.[5] Under this assumption, messages are special functions composed of several codes (the methods); when one special function is applied to an argument (i.e., the messages is sent to the argument), the code to execute is chosen according to the class, i.e., the type, of the argument. In other words, messages are *overloaded functions*. When such functions are applied, code selection is not performed at compile time, as is usual, but must instead be done at run-time using a *late binding* or *late selection* strategy (this run-time selection is sometimes also called *dynamic binding* or *dynamic dispatch*). We can see why run-time selection is necessary by considering the following example. Suppose that we code a graphical editor in an object-oriented style. Our editor uses the classes *Line* and *Square*, which are subclasses (subtypes) of *Picture*. Suppose that we have defined a method *draw* on all three classes. If method selection is performed at compile time, then the following message *draw*

$$\lambda x^{Picture}.(\ldots x \Leftarrow draw \ldots)$$

is always executed using the *draw* code defined for *Pictures*, since the compile-time type of x is *Picture*. With late binding, the code for *draw* is chosen only when the x parameter has been bound and evaluated, on the basis of the run-time type of x, i.e., according to whether x is bound to an instance of *Line* or *Square* or *Picture*.

[5] We prefer to be a little vague, for the moment, about the precise definition of typing for objects: in the case of name subtyping, the name of the class is used as its type. In the case of structural subtyping, the functionality of the object is used instead

Overloaded functions with late binding are the fundamental feature of the overloading-based model, in the same way that records are the fundamental feature of the record-based model. To study the latter, Cardelli extended the simply typed lambda calculus with subtyping and records. To study the former, we extended the simply typed lambda calculus with subtyping and overloaded functions. This extension led to the definition of the $\lambda\&$-calculus, the intuitive ideas of which can be described as follows (for a detailed presentation see Castagna [1994] and Castagna et al.[1995]; see also Castagna [1995a] for the second-order case).

An overloaded function consists of a collection of ordinary functions (i.e., λ-abstractions), each of which is called a *branch* of the overloaded function. We chose the symbol & (whence the name of the calculus) to glue together ordinary functions into an overloaded one. Thus we add to the simply typed lambda calculus terms of the form

$$(M\&N)$$

which intuitively denotes an overloaded function of two branches, M and N. When $(M\&N)$ is applied to an argument, one of the two branches will be selected according to the type of the argument. We must distinguish ordinary application from the application of an overloaded function because they are fundamentally different mechanisms. The former is implemented by substitution while the latter is implemented by selection. We use "$\bullet$" to denote overloaded application and "$\cdot$" for the usual one.

We build overloaded functions as lists: we start with the *empty* overloaded function, denoted by ε, and concatenate new branches via &. Hence in the term above, M is an overloaded function while N is an ordinary one, i.e., a branch of the resulting overloaded function. We can write an overloaded function with n branches $M_1, M_2, \ldots M_n$ as

$$((\ldots((\varepsilon\&M_1)\&M_2)\ldots)\&M_n).$$

The type of an overloaded function is the set of the types of its branches. Thus if $M_i{:}U_i \to V_i$, then the overloaded function above has type

$$\{U_1 \to V_1, U_2 \to V_2, \ldots, U_n \to V_n\}.$$

If we pass to this function an argument N of type U_j, then the selected branch will be M_j. More formally:

$$(\varepsilon\&M_1\&\ldots\&M_n)\bullet N \;\rhd^*\; M_j\cdot N \qquad (*)$$

where $\rhd^*$ means "rewrites in zero or more steps into."

In short, we add the terms ε, $(M\&N)$, and $(M\bullet N)$ to the terms of the simply typed lambda calculus, and we add sets of arrow types to the types of the simply typed lambda calculus.

We also add a subtyping relation on types. Intuitively, if $U \leq V$ then any expression of type U can be used "safely" (w.r.t. types) wherever an expression of type V is expected; with this definition, a calculus will not produce run-time type errors as long as its evaluation rules maintain or reduce the types of its terms. The subtyping relation for arrow types is the one of Cardelli [1988]: covariance on the right and contravariance on the left. The subtyping relation for overloaded types

can be deduced from the observation that an overloaded function can be used in the place of another overloaded one when, for each branch of the latter, there is one branch in the former that can replace it. Thus, an overloaded type U is smaller than another overloaded type V if and only if, for any arrow type in V, there is at least one smaller arrow type in U. Formally:

$$\frac{U_2 \leq U_1 \quad V_1 \leq V_2}{U_1 \to V_1 \leq U_2 \to V_2} \qquad \frac{\forall i \in I, \exists j \in J \qquad U'_j \to V'_j \leq U''_i \to V''_i}{\{U'_j \to V'_j\}_{j \in J} \leq \{U''_i \to V''_i\}_{i \in I}}$$

Because of subtyping, the type of N in (*) may not match any of the U_i but just be a subtype of one of them. In this case, we choose the branch whose U_i "best approximates" the type of N. More precisely, if the type of N is U, we select the branch h such that $U_h = \min\{U_i | U \leq U_i\}$.

In our system, not every set of arrow types can be considered an overloaded type, however. In particular, a set of arrow types $\{U_i \to V_i\}_{i \in I}$ is an overloaded type if and only if for all i, j in I it satisfies these two conditions:

(1) U maximal in $LB(U_i, U_j) \;\Rightarrow\;$ there exists a unique $h \in I$ such that $U_h = U$
(2) $U_i \leq U_j \;\Rightarrow\; V_i \leq V_j$

where $LB(U_i, U_j)$ denotes the set of common lower bounds of U_i and U_j.

Condition (1) concerns the selection of the correct branch. We said earlier that if we apply an overloaded function of type $\{U_i \to V_i\}_{i \in I}$ to a term of type U, then the selected branch has type $U_j \to V_j$ where $U_j = \min_{i \in I}\{U_i | U \leq U_i\}$. Condition (1) guarantees the existence and uniqueness of this branch (it is a necessary and sufficient condition for existence, as proved in Castagna [1994]).

More interesting for the purposes of this article is the second condition, which we call the *covariance condition*. Condition (2) guarantees that during computation the type of a term may only decrease. More concretely, if we have a two-branch overloaded function M of type $\{U_1 \to V_1, U_2 \to V_2\}$ with $U_2 < U_1$, and we pass to it a term N, which at compile-time has type U_1, then the compile-time type of $M \bullet N$ will be V_1. If the normal form of N has type U_2, however, (which is possible, since $U_2 < U_1$) then the run-time type of $M \bullet N$ will be V_2. Condition(2) requires that $V_2 \leq V_1$.

So far, we have shown how to include overloading and subtyping in the calculus. Late binding still remains. A simple way to obtain it is to impose the condition that a reduction like (*) can be performed only if N is a closed normal form. With this restriction, the most precise type for N is apparent whenever the argument is used to select the appropriate branch from an overloaded function.

Let us stress, once more, that it is important to understand that overloaded functions with late binding are significantly different from the form of overloaded functions found in C or definable C++, for example. With late binding, overloading is resolved at run-time, while C/C++ overloaded functions are resolved at compile time.

At this point we can intuitively show how to use this calculus to model object-oriented languages by noting that in $\lambda\&$ it is possible to encode surjective pairings, simple records (those of Cardelli [1988]) —as described in Section 4— and extensible records (see Cardelli and Mitchell [1991], Rémy [1989], and Wand [1987]). These encodings can be found in Castagna [1994].

Conditions (1) and (2) have a very natural interpretation in object-oriented languages. Suppose that *mesg* is the identifier of an overloaded function with the following type:

$$mesg \ : \{C_1 \to T_1, C_2 \to T_2\}.$$

In object-oriented jargon, *mesg* is then a message containing two methods, one defined in the *class* C_1 and the other in the class C_2: class C_1's method returns a result of type T_1, while class C_2's method returns a result of type T_2. If C_1 is a subclass of C_2 (more precisely a subtype: $C_1 \leq C_2$), then the method of C_1 overrides the one of C_2. Condition (2) requires that $T_1 \leq T_2$. That is to say, the covariance condition expresses the requirement that a method that *overrides* another one must return a smaller type. If instead C_1 and C_2 are unrelated, but there exists some subclass C_3 of both of them ($C_3 \leq C_1, C_2$), then C_3 has been defined by *multiple inheritance* from C_1 and C_2. Condition (1) requires that a branch be defined for C_3 in *mesg*, i.e., in case of multiple inheritance, methods defined for the same message in more than one ancestor must be explicitly redefined.

Let us see how this all fits together by an example. Consider the class `2DPoint` with two integer instance variables `x` and `y` and subclass `3DPoint`, which has an additional instance variable `z`. These relationships can be expressed with the following definitions:

```
class 2DPoint                    class 3DPoint is 2DPoint
  {                                {
    x:Int;                           x:Int;
    y:Int                            y:Int;
  }                                  z:Int
     :                             }
     :                                 :
                                       :
```

where in place of the dots are the definitions of the methods. To a first approximation, these classes can be modeled in $\lambda\&$ by two atomic types *2DPoint* and *3DPoint* with *3DPoint*$\leq$*2DPoint*, whose respective representation types are the records $\langle\!\langle x\!:\!\text{Int} \ ; y\!:\!\text{Int} \rangle\!\rangle$ and $\langle\!\langle x\!:\!\text{Int} \ ; y\!:\!\text{Int} \ ; z\!:\!\text{Int} \ \rangle\!\rangle$. Note that the assumption *3DPoint*$\leq$*2DPoint* is "compatible" with the subtyping relation on the corresponding representation types.

One method that we could include in the definition of `2DPoint` is

```
norm = sqrt(self.x^2 + self.y^2)
```

where `self` denotes the receiver of the message. We may override this method in `3DPoint` with the following method

```
norm = sqrt(self.x^2 + self.y^2 + self.z^2).
```

In $\lambda\&$, these methods are written as a two-branch overloaded function:

$$\begin{aligned} norm \equiv (\ & \lambda self^{\,2DPoint}.\sqrt{self.x^2 + self.y^2} \\ & \&\ \lambda self^{\,3DPoint}.\sqrt{self.x^2 + self.y^2 + self.z^2} \\ &) \end{aligned}$$

where ε is omitted for brevity. The type of this overloaded function is $\{\mathit{2DPoint} \to \mathit{Real}\ ,\ \mathit{3DPoint} \to \mathit{Real}\}$. Note that `self` becomes in $\lambda\&$ the first parameter of the overloaded function, i.e., the one whose class determines the selection.

Covariance appears when, for example, we define a method that modifies the instance variables. For example, a method initializing the instance variables of *2DPoint* and *3DPoint* objects will have the following type

$$\mathit{initialize}\ : \{\mathit{2DPoint} \to \mathit{2DPoint}\ ,\ \mathit{3DPoint} \to \mathit{3DPoint}\}.$$

In this framework, the inheritance mechanism is given by subtyping plus the branch selection rule. If we send a message of type $\{C_i \to T_i\}_{i \in I}$ to an object of class C, then the method defined in the class $\min_{i=1..n}\{C_i | C \leq C_i\}$ will be executed. If this minimum is exactly C, then the receiver uses the method defined in its own class; if the minimum is strictly greater than C, then the receiver uses the method that its class, C, has *inherited* from the minimum. Note that the search for the minimum corresponds exactly to Smalltalk's "method look-up," where one searches for the least superclass (of the receiver's class) for which a given method has been defined.

Modeling messages by overloaded functions has some advantages. For example, since these functions are first-class values, so are messages. It becomes possible to write functions (even overloaded ones) that take a message as an argument or return one as result. Another interesting characteristic of this model is that it allows methods to be added to an already existing class C without modifying the type of its objects. Indeed, if the method concerned is associated with the message m, it suffices to add a new branch for the type C to the overloaded function denoted by m.[6]

In the context of this article, however, the most notable advantage of using overloaded functions is that it allows multiple dispatch.[7] As we hinted in the previous section, one of the major problems of the record model is that it is impossible to combine satisfactorily subtyping and binary methods (i.e., methods with a parameter of the same class as the class of the receiver). This problem gave rise to the proposed use of the unsound covariant subtyping rule. Let us reconsider the point example above, adding the method *equal*. In the record-based models, two-dimensional and three-dimensional points are modeled by the following recursive records:

$$\mathit{2EqPoint} \equiv \langle\!\langle x\colon \mathrm{Int};\ y\colon \mathrm{Int};\ \mathit{equal}\colon \mathit{2EqPoint} \to \mathrm{Bool} \rangle\!\rangle$$
$$\mathit{3EqPoint} \equiv \langle\!\langle x\colon \mathrm{Int};\ y\colon \mathrm{Int};\ z\colon \mathrm{Int};\ \mathit{equal}\colon \mathit{3EqPoint} \to \mathrm{Bool} \rangle\!\rangle.$$

Because of the contravariance of arrow, the type of the field *equal* in *3EqPoint* is not a subtype of the type of *equal* in *2EqPoint*. Therefore $\mathit{3EqPoint} \not\leq \mathit{2EqPoint}$.[8] Let us consider the same example in $\lambda\&$. We have already defined the atomic types *2DPoint* and *3DPoint*. We can still use them since, unlike what happens in

[6] It is important to remark that the new method is available at once to all the instances of C, and thus it is possible to send the message m to an object of class C even if this object has been defined *before* the branch for C in m.

[7] That is, the capability of selecting a method taking into account other classes besides that of the receiver of the message.

[8] The subtyping rule for recursive types says that if from $X \leq Y$ one can deduce that $U \leq V$ then $\mu X.U \leq \mu Y.V$ follows. In the example above, $2EqPoint \equiv \mu X\,.\,\langle\!\langle x\colon \mathrm{Int};\ y\colon \mathrm{Int};\ equal\colon X \to \mathrm{Bool} \rangle\!\rangle$.

the record case, adding a new method to a class does not change the type of its instances. In $\lambda\&$, a declaration such as

$$\mathit{equal}\colon \{\mathit{2DPoint} \to (\mathit{2DPoint} \to \text{Bool})\ ,\ \mathit{3DPoint} \to (\mathit{3DPoint} \to \text{Bool})\}$$

is not well defined either: because $\mathit{3DPoint} \leq \mathit{2DPoint}$, condition (2) —the covariance condition— requires that $\mathit{3DPoint} \to \text{Bool} \leq \mathit{2DPoint} \to \text{Bool}$, which does not hold because of the contravariance of arrow on the left argument. It must be noted that such a function would choose the branch according to the type of just the first argument. Now, the code for *equal* cannot be chosen until the types of *both* arguments are known. This is the essential reason why the type above must be rejected (in any case, it is easy to write a term with the above type producing an error). In $\lambda\&$, however, it is possible to write a function that takes into account the types of two (or more) arguments for branch selection. For *equal*, this is obtained as follows:

$$\mathit{equal}\colon \{(\mathit{2DPoint} \times \mathit{2DPoint}) \to \text{Bool}\ ,\ (\mathit{3DPoint} \times \mathit{3DPoint}) \to \text{Bool}\}.$$

If we send to this function two objects of class *3DPoint*, then the second branch is chosen; when one of the two arguments is of class *2DPoint* (and the other is of a class smaller than or equal to *2DPoint*), the first branch is chosen.

At this point, we are able to make precise the roles played by covariance and contravariance in subtyping: contravariance is the correct rule when you want to substitute a function of a given type for another one of a different type; covariance is the correct condition when you want to specialize (in object-oriented jargon "override") a branch of an overloaded function by one with a smaller input type. It is important to notice that, in this case, the new branch *does not replace* the old branch, but rather it *conceals* it from the objects of some classes. Our formalization shows that the issue of "contravariance versus covariance" was a false problem caused by the confusion of two mechanisms that have very little in common: substitutivity and overriding.

Substitutivity establishes when an expression of a given type S can be used *in place of* an expression of a different type T. This information is used to type ordinary applications. More concretely, if f is a function of type $T \to U$, then we want to characterize a category of types whose values can be passed as arguments to f; it must be noted that these arguments will be *substituted*, in the body of the function, for the formal parameter of type T. To this end, we define a subtyping relation such that f accepts every argument of type S smaller than T. Therefore, the category at issue is the set of subtypes of T. When T is $T_1 \to T_2$ it may happen that, in the body of f, the formal parameter is applied to an expression of type T_1. Hence, we deduce two facts: the actual parameter must be a function (thus, if $S \leq T_1 \to T_2$, then S has the shape $S_1 \to S_2$), and furthermore, it must be a function to which we can pass an argument of type T_1 (thus $T_1 \leq S_1$, yes! ... contravariance). It is clear that if one is not interested in passing functions as arguments, then there is no reason to define the subtyping relation on arrows (this is the reason why O_2 works well even without contravariance[9]).

[9] Eiffel compensates the holes resulting from the use of covariance by a link-time data-flow analysis of the program.

Overriding is a totally different feature. Suppose we have an identifier m (in the circumstances, a message) that identifies two functions $f : A \to C$ and $g : B \to D$ where A and B are incomparable. When this identifier is applied to an expression e, then the meaning of the application is f applied to e if e has a type smaller than A (in the sense of substitutivity explained above), or g applied to e if e has type smaller than B. Suppose now that $B \leq A$. The application in this case is resolved by selecting f if the type of e is included between A and B, or by selecting g if the type is smaller than or equal to B. There is a further problem, however. The types may decrease during computation. It may happen that the type checker sees that e has type A, and infers that m applied to e has type C (f is selected). But if, during the computation, the type of e decreases to B, the application will have type D. Thus, D must be a type whose elements can be substituted for elements of type C (in the sense of substitutivity above), i.e., $D \leq C$. You may call this feature covariance, if you like, but it must be clear that it is not a subtyping rule: g does not replace f since g will never be applied to arguments of type A. Indeed, g and f are independent functions that perform two precise and different tasks: f handles the arguments of m whose type is included between A and B, while g handles those arguments whose type is smaller than or equal to B. In this case, we are not defining substitutivity; instead, we are giving a formation rule for sets of functions in order to ensure the type consistency of the computation. In other words, while contravariance characterizes a (subtyping) *rule*, i.e., a tool to deduce an existing relation, covariance characterizes a (formation) *condition*, i.e., a law that programs must observe.

Since these arguments are still somewhat too abstract for object-oriented practitioners, let us write them in "plain" object-oriented terms as we did at the end of Section 2. A message may have several parameters, and the type (class) of each parameter may or may not be taken into account in the selection of the appropriate method. If a method for that message is overridden, then the parameters that determine the selection must be covariantly overridden (i.e., the corresponding parameters in the overriding method must have a lesser type). Those parameters that are not taken into account for the selection must be contravariantly overridden (i.e., the corresponding parameters in the overriding method must have a greater type).

How is all this translated into object-oriented type systems? Consider a message m applied (or "sent") to n objects $e_1 \ldots e_n$ where e_i is an instance of class C_i. Suppose we want to consider the classes of only the first k objects in the method selection process. This dispatching scheme can be expressed using the following notation:

$$m(e_1, \ldots, e_k | e_{k+1}, \ldots, e_n).$$

If the type of m is $\{S_i \to T_i\}_{i \in I}$, then the expression above means that we want to select the method whose input type is the $\min_{i \in I}\{S_i \mid (C_1 \times \ldots \times C_k) \leq S_i\}$ and then to pass it all the n arguments. The type, say $S_j \to T_j$, of the selected branch must have the following form:

$$\underbrace{(A_1 \times \ldots \times A_k)}_{S_j} \to \underbrace{(A_{k+1} \times \ldots \times A_n) \to U}_{T_j}$$

where $C_i \leq A_i$ for $1 \leq i \leq k$ and $A_i \leq C_i$ for $k < i \leq n$.[10] If we want to override the selected branch by a more precise one, then, as explained above, the new method must covariantly override $A_1 \dots A_k$ (to specialize the branch) and contravariantly override $A_{k+1} \dots A_n$ (to have type safety).

4. COVARIANCE IN THE RECORD-BASED MODEL

We said in the previous section that covariance must be used to specialize the arguments that are taken into account during method selection. In record-based models, no arguments are taken into account in method selection: the method to use is uniquely determined by the record (i.e., the object) that the dot selection is applied to. Thus in these models, it appears that we cannot have a covariance condition.

Strictly speaking, this argument is not very precise, since the record-based model does possess a limited form of "covariance" (in the sense of a covariant dependency that the input and the output of a message must respect), but it is hidden by the encoding of objects. Consider a label ℓ. By the subtyping rule for record types, if we "send" this label to two records of type S and T with $S \leq T$, then the result returned by the record of type S must have a type smaller than or equal to the type of the one returned by T. This requirement exactly corresponds to the dependency expressed by the covariance condition (2), [11] but its form is much more limited because it applies only to record types (since we "sent" a label), but not to products (i.e., multiple dispatch) nor to arrows. We may see this correspondence by treating a record label ℓ as a potentially infinitely branching overloaded function that takes as its argument any record with *at least* a field labeled by ℓ and returns a value of the corresponding type:

$$\ell : \{\, \langle\!\langle \ell\!:\!T \rangle\!\rangle \to T \,\}_{T \in \mathbf{Types}}$$

Note that this treatment respects the covariance condition (2) since $\langle\!\langle \ell\!:\!T \rangle\!\rangle \leq \langle\!\langle \ell\!:\!T' \rangle\!\rangle$ implies $T \leq T'$. Though, all the types of the arguments are records of the same form; no other kind of type is allowed. Hence record-based models possess only a limited form of covariance, an "implicit" covariance.

However the idea is that "explicit" covariance without multiple dispatching does not exist. Actual record-based models do not possess multiple dispatching. This lack does not mean that the analogy "objects as records" is incompatible with multiple dispatching, however. The problem is simply that the formalisms that use this analogy are not expressive enough to model it.

In the rest of this section, therefore, we show how to construct a record-based model of object-oriented programming using the $\lambda\&$-calculus, i.e., we use $\lambda\&$ to describe a model in which objects will be modeled by records. In the model we obtain, it will be possible to perform multiple dispatch, and hence we will recover the covariance relation. Thus, we will have shown by example that covariance and

[10] Indeed, by the covariance condition, all methods whose input type is compatible with the one of the arguments must be of this form.

[11] Recall that in the overloading-based model, covariance has exactly the same meaning as here. That is, the smaller the type of the object that a message (label) is sent to, the smaller the type of the result.

contravariance can cohabit in type-safe systems based on the analogy of "objects as records."

The key point is that records can be encoded in $\lambda\&$. By using this encoding, we can mimic any model based on simple records, but with an additional benefit: we have overloaded functions. For the purposes of this article, simple records suffice. Let us recall their encoding in $\lambda\&$ as given in Castagna et al. [1995].

Let $L_1, L_2, \ldots$ be an infinite list of atomic types. Assume that they are *isolated* (i.e., for any type T, if $L_i \leq T$ or $T \leq L_i$, then $L_i = T$), and introduce for each L_i a *constant* $\ell_i : L_i$. It is now possible to encode record types, record values, and record field selection, respectively, as follows:

$$\begin{array}{lll} \langle\!\langle \ell_1 : V_1; \ldots; \ell_n : V_n \rangle\!\rangle & \equiv & \{L_1 \to V_1, \ldots, L_n \to V_n\} \\ \\ \langle \ell_1 = M_1; \ldots; \ell_n = M_n \rangle & \equiv & (\varepsilon \,\&\, \lambda x^{L_1}.M_1 \,\&\, \ldots \,\&\, \lambda x^{L_n}.M_n) \qquad (x^{L_i} \notin FV(M_i)) \\ M.\ell & \equiv & M \bullet \ell \end{array}$$

In words, a record value is an overloaded function that takes as its argument a label —each label belongs to a different type— that is used to select a particular branch (i.e., field) and then is discarded (since $(x^{L_i} \notin FV(M_i))$. Since $L_1 \ldots L_n$ are isolated, the typing, subtyping, and reduction rules for records are *special cases* [12] of the rules for overloaded types. Henceforth, to enhance readability, we will use the record notation rather then its encoding in $\lambda\&$. All the terms and types written below are encodable in $\lambda\&$.[13]

Consider again the *equal* message. The problem, we recall, was that it is not possible to select the right method by knowing the type of just one argument. The solution in the overloading-based approach was to use multiple dispatching and to select the method based on the class of both arguments. We can use the same solution with records. Thus, the method defined for *2EqPoint* must select different code according to the class of the "second" argument (similarly for *3EqPoint*). This can be obtained by using in the field for *equal* an overloaded function. The definition of the previous two recursive types therefore becomes:

$$\begin{array}{lll} \mathit{2EqPoint} \equiv & \langle\!\langle x : \mathrm{Int}; & \\ & \quad y : \mathrm{Int}; & \\ & \quad \mathit{equal} : \{\mathit{2EqPoint} \to \mathrm{Bool}, \mathit{3EqPoint} \to \mathrm{Bool}\} & \\ & \rangle\!\rangle & \end{array}$$

$$\begin{array}{lll} \mathit{3EqPoint} \equiv & \langle\!\langle x : \mathrm{Int}; & \\ & \quad y : \mathrm{Int}; & \\ & \quad z : \mathrm{Int}; & \\ & \quad \mathit{equal} : \{\mathit{2EqPoint} \to \mathrm{Bool}, \mathit{3EqPoint} \to \mathrm{Bool}\} & \\ & \rangle\!\rangle & \end{array}$$

Note that now $\mathit{3EqPoint} \leq \mathit{2EqPoint}$. The objection may now be raised that when we define the class *2EqPoint*, the class *3EqPoint* may not exist yet, and so it

[12] There is an "if and only if" relation, e.g., the encodings of two record types are in subtyping relation if and only if the record types are in the same relation.

[13] More precisely, in $\lambda\&$ plus recursive types.

would be impossible to define in the method *equal* for *2EqPoint* the branch for *3EqPoint*. But note that a lambda abstraction may be considered as a special case of an overloaded function with only one branch and thus that an arrow type may be considered as an overloaded type with just one arrow (it is just a matter of notation; see Section 4.3 of Castagna [1994]). Hence, we could have first defined *2EqPoint* as

$$\begin{aligned} \mathit{2EqPoint} \equiv \langle\!\langle & x\colon \mathrm{Int}; \\ & y\colon \mathrm{Int}; \\ & \mathit{equal}\colon \{\mathit{2EqPoint} \to \mathrm{Bool}\} \\ \rangle\!\rangle & \end{aligned}$$

and then added the class *3EqPoint* with the following type:

$$\begin{aligned} \mathit{3EqPoint} \equiv \langle\!\langle & x\colon \mathrm{Int}; \\ & y\colon \mathrm{Int}; \\ & z\colon \mathrm{Int}; \\ & \mathit{equal}\colon \{\mathit{2EqPoint} \to \mathrm{Bool}, \mathit{3EqPoint} \to \mathrm{Bool}\} \\ \rangle\!\rangle & \end{aligned}$$

Note that again $\mathit{3EqPoint} \leq \mathit{2EqPoint}$ holds. An example of objects with the types above is

$$\begin{aligned} \mathbf{Y}\ (\lambda \mathit{self}^{\mathit{2EqPoint}}.\ & \\ & \langle x = 0; \\ & \ y = 0; \\ & \ \mathit{equal} = \lambda p^{\mathit{2EqPoint}}.(\mathit{self}.x = p.x) \wedge (\mathit{self}.y = p.y) \\ & \rangle) \end{aligned}$$

$$\begin{aligned} \mathbf{Y}\ (\lambda \mathit{self}^{\mathit{3EqPoint}}.\ & \\ & \langle x = 0; \\ & \ y = 0; \\ & \ z = 0; \\ & \ \mathit{equal} = (\ \lambda p^{\mathit{2EqPoint}}.(\mathit{self}.x = p.x) \wedge (\mathit{self}.y = p.y) \\ & \qquad \&\lambda p^{\mathit{3EqPoint}}.(\mathit{self}.x = p.x) \wedge (\mathit{self}.y = p.y) \wedge (\mathit{self}.z = p.z)) \\ & \rangle) \end{aligned}$$

where $\mathbf{Y}$ is the fixpoint operator (which is encodable in $\lambda\&$: see Castagna [1994]).

The type safety of expressions having the types above is assured by the type safety of the $\lambda\&$-calculus. Indeed, the type requirements for specializing methods as in the case above can be explained in a simple way: when specializing a binary (or general n-ary) method for a new class C' from an old class C, the specialized method must specify not only its behavior in the case that it is applied to an object of the the new class C', but also its behavior in the case that it is applied to an object of the old class C. Going back to our example of Section 2, this is the same as saying that when one specializes the class of natural numbers from the real numbers, then type safety can be obtained by specifying not only how to compare a natural number to another natural number, but also how to compare it to a real number. The conclusion is that in the record-based approach, specialization of functional fields is done by using (contravariant) subtypes, but to make specialization type-safe

and convenient with binary (and general n-ary) methods, we must more accurately specialize binary (and general n-ary) methods by defining their behavior not only for the objects of the new class, but also for all possible combinations of the new objects with the old ones.

One could object that if the subtyping hierarchy were very deep, this approach would require us to define many branches, one for each ancestor, and that in most cases these definitions would never be used. Actually, many of these definitions are not necessary. Indeed, in all cases, two branches will suffice to assure type safety.[14] For example, suppose that we further specialize our equality-point hierarchy by adding further dimensions. When we define the *nEqPoint*, it is not necessary to define the behavior of the *equal* method for *nEqPoint*, *(n-1)EqPoint*,..., *2EqPoint*; two branches are more than enough: one for *2EqPoint* (the only one really necessary), the other for *nEqPoint*. Why? The reason is that from the subtyping rules, it follows that if for all $i \in I$, $T_i \leq T$ then $\{T \to S\} \leq \{T_i \to S\}_{i \in I}$. If we take *2EqPoint* for T and the various *(n-k)EqPoint* for T_i we may see that the branch *2EqPoint*$\to$ Bool suffices in the definition of *nEqPoint* to guarantee type safety; all the other branches are not strictly necessary, but they may be added at will. Furthermore, if the branch that guarantees type safety is missing, it can be added in an automatic way. Therefore, multiple dispatch can be embedded directly into the compiler technology in order to "patch" programs of languages that, like O_2, use covariant subtyping, without modifying the language's syntax. In that case type safety is obtained without any modification of the code that already exists: a recompilation is enough (see Castagna [1995b]).

Finally, we want to stress that, in this record-based model, covariance and contravariance naturally coexist. This is not apparent in the example above with *equal* since all the branches of *equal* return the same type Bool. To see that the two concepts coexist, imagine that instead of the method for *equal* we had a method *add*. Then we would have objects of the following types:

$$
\begin{aligned}
2AddPoint \equiv \langle\!\langle & x\colon \mathrm{Int};\\
& y\colon \mathrm{Int};\\
& add\colon \{2AddPoint \to 2AddPoint\}\\
\rangle\!\rangle &
\end{aligned}
$$

$$
\begin{aligned}
3AddPoint \equiv \langle\!\langle & x\colon \mathrm{Int};\\
& y\colon \mathrm{Int};\\
& z\colon \mathrm{Int};\\
& add\colon \{2AddPoint \to 2AddPoint, 3AddPoint \to 3AddPoint\}\\
\rangle\!\rangle &
\end{aligned}
$$

The various branches of the multimethod[15] *add* in *3AddPoint* are related in a covariant way, since the classes of their arguments determine the code to be executed.

[14] This observation does not depend on the size or the depth of the hierarchy, and it is valid also for n-ary methods. More than two branches may be required only if we use multiple inheritance.

[15] A multimethod is a collection of methods (or branches). When a multimethod is applied to argument objects, the appropriate method to execute is selected according to the type of one or more of the arguments. Multimethods correspond to our overloaded functions.

5. CONCLUSION

With this article we hope to have contributed decisively to the debate about the use of covariance and contravariance. We have tried to show that the two concepts are not antagonistic, but that each has its own use: covariance for specialization and contravariance for substitutivity. Also, we have tried to convey the idea that the independence of the two concepts is not characteristic of a particular model but is valid in general. The fact that covariance did not appear explicitly in the record-based model was not caused by a defect of the model but rather by a deficiency of all the calculi that used the model. In particular, they were not able to capture multiple dispatching. Indeed, it is only when one deals with multiple dispatching that the differences between covariance and contravariance become apparent. The use of overloaded functions has allowed us to expose the covariance hidden in records.

As an aside, we have shown that the $\lambda\&$-calculus can be taken as the basic calculus both of an overloading-based and of a record-based model. With it, we not only obtain a more uniform vision of object-oriented type theories but, in the case of the record-based approach, we also gain multiple dispatching, which is, we believe, *the* solution to the typing of binary methods.

To end this note we give three "golden rules" that summarize our discussion.

The Golden Rules

(1) Do not use (left) covariance for arrow subtyping.

(2) Use covariance to override parameters that drive *dynamic* method selection.

(3) When overriding a binary (or n-ary) method, specify its behavior not only for the actual class but also for its ancestors.

ACKNOWLEDGMENTS

I want to thank Véronique Benzaken who encouraged me to write this article and Kathleen Milsted for her patient reading and many suggestions. Special thanks to John Mitchell and to Kathleen Fisher whose revisions made these notes more readable.

REFERENCES

Apple Computer Inc. 1992. *Dylan: An Object-Oriented Dynamic Language*. Eastern Research and Technology, Apple Computer Inc., Cambridge, Mass.

Bancilhon, F., Delobel, C., and Kanellakis, P. (Eds.) 1992. *Implementing an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, San Mateo, Calif.

Cardelli, L. 1988. A semantics of multiple inheritance. *Inf. Comput. 76*, 138–164. A previous version can be found in *Semantics of Data Types*. Lecture Notes in Computer Science, vol. 173. Springer-Verlag, New York, 1984, pp. 51–67.

Cardelli, L. and Mitchell, J. 1991. Operations on records. *Math. Struct. Comput. Sci. 1,* 1, 3–48.

Castagna, G. 1995a. A meta-language for typed object-oriented languages. *Theor. Comput. Sci.* To be published. An extended abstract appears in *Proceedings of the 13th Conference on the Foundations of Software Technology and Theoretical Computer Science*. Lecture Notes in Computer Science, vol. 761. Springer-Verlag, New York, 1993.

Castagna, G. 1995b. A proposal for making O_2 more type safe. Tech. Rep. LIENS-95-4, LIENS, Paris, France. Available by anonymous ftp from ftp.ens.fr in file /pub/dmi/users/castagna/o2.dvi.Z.

Castagna, G. 1994. Overloading, subtyping and late binding: Functional foundation of object-oriented programming. Ph.D. thesis, Université Paris 7, Paris, France. Appeared as LIENS Tech. Rep.

Castagna, G., Ghelli, G., and Longo, G. 1995. A calculus for overloaded functions with subtyping. *Inf. Comput. 117,* 1, 115–135. A preliminary version has been presented at the *1992 ACM Conference on LISP and Functional Programming* (San Francisco, June).

DeMichiel, L. and Gabriel, R. 1987. Common lisp object system overview. In *Proceeding of ECOOP '87 European Conference on Object-Oriented Programming* (Paris, France). Lecture Notes in Computer Science, vol. 276. Springer-Verlag, Berlin, 151–170.

Ghelli, G. 1991. A static type system for message passing. In *Proceedings of OOPSLA '91.* ACM, New York.

Goguen, J. and Meseguer, J. 1989. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Tech. Rep. SRI-CSL-89-10, Computer Science Laboratory, SRI International, Menlo Park, Calif. July.

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

Gunter, C.A. and Mitchell, J.C. 1994. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* The MIT Press, Cambridge, Mass.

Jouannaud, J.-P., Kirchner, C., Kirchner, H., and Megrelis, A. 1992. OBJ: Programming with equalities, subsorts, overloading and parametrization. *J. Logic Program. 12,* 257–279.

Lécluse, C., Richard, P., and Vélez, F. 1988. O_2, an object-oriented data model. In *Proceedings of the ACM SIGMOD Conference* (Chicago, Ill.). ACM, New York.

Martí-Oliet, N. and Meseguer, J. 1990. Inclusions and subtypes. Tech. Rep. SRI-CSL-90-16, Computer Science Laboratory, SRI International, Menlo Park, Calif. Dec.

Meyer, B. 1991. *Eiffel: The Language.* Prentice-Hall, Englewood Cliffs, N.J.

Pinson, L. and Wiener, R. 1992. *Objective-C: Object-Oriented Programming Techniques.* Addison-Wesley, Reading, Mass.

Rémy, D. 1989. Typechecking records and variants in a natural extension of ML. In the *16th Annual ACM Symposium on the Principles of Programming Languages.* ACM, New York.

Schwartzbach, M. 1994. Developments in object-oriented type systems. Tutorial given at POPL'94. Unpublished.

Stroustrup, B. 1986. *The C++ Programming Language.* Addison-Wesley, Reading, Mass.

Wand, M. 1987. Complete type inference for simple objects. In the *2nd Annual Symposium on Logic in Computer Science.* IEEE Computer Society Press, Los Alamitos, Calif.

Received April 1994; revised January 1995; accepted February 1995