COVARIANCE AND CONTRAVARIANCE:CONFLICT WITHOUT A CAUSE

Giuseppe CASTAGNA

LIENS - 94 - 18

October 1994

Laboratoire d'Informatique de l'Ecole Normale Supérieure
45 rue d'Ulm 75230 PARIS Cedex 05

Tel : (33)(1) 44 32 00 00
Adresse électronique : castagna@dmi.ens.fr
URL: http://www.dmi.ens.fr/~castagna

Covariance and contravariance: conflict without a cause

Giuseppe Castagna[*]
LIENS(CNRS)-DMI
45 rue d'Ulm, 75005 Paris. FRANCE
e-mail: castagna@dmi.ens.fr

Abstract

In type theoretic research on object-oriented programming the "covariance versus contravariance issue" 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 *substitutivity*, or subtyping relation (that establishes which sets of codes can replace *in every context* another given set). A covariant relation, instead, characterizes the *specialization* of code (i.e. the definition of new code that replaces the old one *in some particular cases*). Therefore, covariance and contravariance are not opposing views, but distinct concepts that each have their place in object-oriented systems and that both can (and should) be type safely integrated in an object-oriented language.

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 also in record-based models, but is hidden by a deficiency of all calculi that realize this model.

As an aside, we show that the $\lambda\&$-calculus [CGL95] can be taken as the basic calculus both for an overloading-based and for a record-based model. In that case, 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, which is not captured by the existing record-based models.

1 Introduction

In type theoretic research on object-oriented programming the "covariance versus contravariance issue" 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 a "covariant specialization" for the methods in the O_2 data model [LRV88]. Since then, it has been disputed in object-oriented languages whether one has to use covariant or contravariant specialization of the methods. The fact that this debate is still very felt is witnessed by the excellent tutorial on object-oriented type systems given by Michael Schwartzbach in the last POPL conference [Sch94]: already 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.

With this short note we argue that the choice between covariance and contravariance is a false problem. That covariance and contravariance characterize two completely distinct mechanisms, subtyping and specialization, whose confusion made appear covariance and contravariance as mutually exclusive. That, therefore, covariance and contravariance are not opposing views but distinct concepts that can both be integrated in a type safe formalism. That, finally,

[*]This work was upported by grant no. 203.01.56 of the Consiglio Nazionale delle Ricerche, Comitato Nazionale delle Scienze Matematiche, Italy, to work at LIENS

it would even be an error to exclude either of them, since, then, the corresponding mechanism could not be properly implemented.

This result is patent in the model of object-oriented programming defined by Giuseppe Longo, Giorgio Ghelli and the author in [CGL95], it is already present in Ghelli's seminal work [Ghe91] and it is somehow hidden in the work on `OBJ` [GM89, MOM90, JKKM92]. Here, we want to stress that this result is independent from 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 [Car88], and further developed by many other authors (see the collection [GM94] for a wide review on the record-based model).

The paper 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, we give a precise connotation of *subtyping* and *specialization*, and we 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 from the particular model, by revealing the (type-safe) covariance of the record-based model. Section 5 contains our conclusion and the golden rules for type-safe use of covariance and contravariance.

We assume that the reader is familiar with the model of object-oriented programming based on the analogy "object as records" and aware of the typing issues it implies.

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 [Car88], an object is modeled by a record value, 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 a different object in two different ways: either by adding new methods —i.e. new fields—, or by redefining the existing ones —i.e. overriding the old method.[1] A specialized object can be used wherever the object it specializes can be used. This implies that method overriding must be restricted if type safety is wanted. A sufficient condition that assures type safety (at least for method specialization) is the requirement that a field can be specialized only by terms whose types are "subtypes" of the type of the field. The core of the problem concerns methods that have a functional type. Indeed, the subtyping relation for functional types is defined in [Car88] 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 inverses the direction of "$\leq$"). The behavior of the left argument having been taken 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 inversed), then type safety is lost. In that case, indeed, it is quite easy to write a statically well-typed term producing a run-time type error. Despite its unsoudness, covariant specialization has its tenacious defenders, and not without cause. The O_2 system [BDK92], for example, uses covariant specialization. The

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

[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 paper we will use "contravariance", "contravariant rule" and "contravariant specialization" to denote the co-contravariant behavior of the arrow.

contravariant rule, besides being less natural than the covariant one, is indeed 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 which is specialized to another object o_2 of type S. Imagine that for these objects we have defined a method *equal*, which compares the object at issue with another object of the same type. Thus, *equal* has type $T \times T \rightarrow Bool$ for the object o_1 and $S \times S \rightarrow Bool$ for the object o_2. In the record-based approach the fields labeled *equal* will have the type $T \rightarrow Bool$ in o_1 and $S \rightarrow Bool$ in o_2 since the method *belongs* to the object and thus it already knows its first argument, usually denoted by the keyword `self`. In any case, if the contravariant rule is used, the type associated with *equal* for S-objects is not a subtype of the one of *equal* for 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 unnatural. 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, the 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 does not cause much problem in practice. Thus, many people prefer to give up type safety and use the covariant subtyping rule for specialization. And the general conclusion is that one has to use contravariance when static type safety is really required, since otherwise covariance is more natural, flexible and expressive.

Both viewpoints seem as much reasonable as incompatible. Though, there is a little flaw in the comparison made above: covariance in the O_2's (nearly) overloading-based model is compared with contravariance in the record-based model. The difference between the two models is the type of the parameter `self`, which appears in the former model and disappears in the latter (see the type of *equal* in the previous example). The conclusion drawn above is wrong right 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 model (section 4). We will show that both covariance and contravariance can be used and type safety still granted. To that end one does not need to impose any restriction, but just to point out what each concept serves for.

Before hinting at the solution let us fix some terminology. Recall that each object has associated a set of private operations, called *methods* in Smalltalk [GR83], Objective-C [PW92] and CLOS [DG87] and *member functions* in C++ [Str86]. These operations can be executed by applying a special operator to the object itself: the object is the left argument of a dot selection in C++, the argument of a generic function in CLOS and the receiver of a message in Objective-C and Smalltalk. In order to simplify conceptually the exposition we can refer to all these different ways of selecting a method, as operations of "message sending", where the message is the name of the generic function in CLOS or the right argument of the dot selection in C++. Last but not least, a message may have some parameters. They are introduced by *keywords* in Smalltalk and Objective-C, they are enclosed between parenthesis in C++, they are the arguments of an n-ary generic function in CLOS [3]. Now, and here we enter the core of the discussion, the type (class) of the actual parameters of a message may or may not be taken into account to select, at run-time, the right method. For example in CLOS the type of *all* arguments of a generic function is taken into account for the selection of the method. In C++, Smalltalk and Objective-C, instead, no argument is considered: only the type of the receiver drives the selection.[4] What we formally show in the sequel is that, given a method

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

[4]The use of overloading in C++ imposes a brief remark; overloading in C++ is resolved by the compiler, therefore the dynamic look-up for the methods does not concern the selection of the code of an overloaded member function: at that moment its code has already been expanded. At run-time only the class of the receiver will discriminate between the different methods. For that reason the overloading of C++ is quite different from the one we describe in section 3.

selected by a message with parameters, when this method 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).

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 [CGL95] since in this model the covariance-contravariance issue has a clearer formalization. In section 4 we will also deal with the record-based model.

The idea of 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 passed to an object of a given class, then the method defined for objects of that class is selected from among those composing the message. The object is then passed to the selected method which is executed. This model is quite natural for programmers used to languages with *generic functions* such as CLOS or Dylan [Dyl92] (generic functions of CLOS conincide with our special functions). It needs some work of abstraction for programmers used to other object-oriented languages, for which the methods are grouped inside the objects—as formalized in the record-based model—rather than inside the messages.

However, if we abstract from effective implementation of a language, the two visions are equivalent since they constitute two different perspectives of the same scenery. 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 associated to their messages. Another assumption that we make to define the overloading-based model is that the class is used to type its instances[5]. Under this assumption, messages are special functions composed of several codes; when these functions are applied to an 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*. In this case the selection of the code is not performed as usual at compile time but, instead, must be done at run-time using a *late binding* or *late selection* strategy. The reason for this can be shown by an example. Suppose that a graphical editor is coded using an object-oriented style; it uses the classes *Line* and *Square* which are subclasses (subtypes) of *Picture*; suppose also that a method *draw* is defined on all three classes. If the selection of the methods 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 for pictures, since the compile time type of x is $Picture$. Using 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 $Picture$.

Overloaded functions with late binding are the fundamental feature of the overloading-based model, in the same way as records are the fundamental feature of the record-based model. To study the latter, Cardelli extended the simply typed lambda calculus by subtyping and records. To study the former we extended the simply typed lambda calculus by subtyping and overloaded functions; this led to the definition of the $\lambda\&$-calculus, whose intuitive idea can be described as follows (for a detailed presentation see [CGL95, Cas94]).

An overloaded function is constituted by a collection of ordinary functions (i.e. λ-abstractions), each one forming a different branch. To glue together these functions in an overloaded one, we

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

have chosen the symbol & (whence the name of the calculus); thus we add to the simply typed lambda calculus the term

$$(M\&N)$$

which intuitively denotes an overloaded function of two branches, M and N, one of which will be selected according to the type of the argument. We must distinguish ordinary application from the application of an overloaded function since they constitute different mechanisms[6]. Thus we use "$\bullet$" to denote overloaded application and "$\cdot$" for the usual one.

We build overloaded functions as lists, starting with an *empty* overloaded function, that we denote ε, and concatenating new branches by means of &. Thus, an overloaded function will be a list of ordinary functions, and in the term above, M is an overloaded function while N is an ordinary function, that we call a *branch* of the resulting overloaded function. Therefore, an overloaded function with n branches $M_1, M_2, \dots M_n$ can be written as

$$((\dots((\varepsilon\&M_1)\&M_2)\dots)\&M_n)$$

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

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

and if we pass to this function an argument N of type U_j then the selected branch will be M_j. That is:

$$(\varepsilon\&M_1\&\dots\&M_n)\bullet N \;\rhd^*\; M_j\cdot N \tag{1}$$

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

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

We also have a subtyping relation on types. Its intuitive meaning is that if $U \leq V$ then any expression of U can be "safely" (w.r.t. types) used wherever an expression of V is expected; thus a calculus will not produce run time type errors as long as it maintains or reduces the types of terms. The subtyping relation for arrow types is the one of [Car88]: 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. This is translated into the following rules:

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

Due to subtyping, the type of N in (1) may match none of the U_i but it may be a subtype of one of them. In this case, we choose the branch whose U_i "best approximates" the type, say U, of N; i.e. 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. 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:

$$U \text{ maximal in } LB(U_i, U_j) \;\Rightarrow\; \text{there exists a unique } h \in I \text{ such that } U_h = U \tag{2}$$

$$U_i \leq U_j \;\Rightarrow\; V_i \leq V_j \tag{3}$$

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

[6]The former is implemented by substitution while the latter is implemented by selection.

Condition (2) concerns the selection of the correct branch: we said before 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 $U_j \to V_j$ such that $U_j = \min_{i\in I}\{U_i | U \leq U_i\}$; condition (2) guarantees the existence and uniqueness of this branch (it is a necessary and sufficient condition for the existence, as proved in [Cas94]).

Much more interesting for the purposes of this paper is the other condition which is called the *covariance condition*. Condition (3) guarantees that during computation the type of a term may only decrease. More specifically, 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; but if the normal form of N has type U_2 (which is possible, since $U_2 < U_1$) then the run-time type of $M \bullet N$ will be V_2 and therefore $V_2 \leq V_1$ must hold.

Up to now, 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 (1) can be performed only if N is a closed normal form.

At this point we can intuitively show how to use this calculus to model object-oriented languages. First of all, note that, in $\lambda\&$, it is possible to encode surjective pairings, simple records (those of [Car88]) —as described in section 4— and extensible records (see [Wan87, Rém89, CM91]). These encodings can be found in [Cas94].

Conditions (2) and (3) 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 denoting two methods, one defined in the *class* C_1 and returning a result of type T_1, the other in the class C_2 and returning 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 (3) imposes that $T_1 \leq T_2$. That is to say, covariance simply 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 a 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 (2) imposes that a branch must 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 show how this all fits together by an example. Consider the class `2DPoint` with two integer instance variables `x` and `y`, and with subclass `3DPoint`, which has, in addition, the instance variable `z`. This can be expressed by the following definitions:

```
class 2DPoint
  {
    x:Int;
    y:Int;
  }
     :
     :
```

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

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

A first method that we can include in the definition of `2DPoint` is

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

overridden in `3DPoint` by the following method

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

In $\lambda\&$, this is obtained by a two-branch overloaded function

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

whose type is $\{\mathit{2DPoint} \to \mathit{Real}\ ,\ \mathit{3DPoint} \to \mathit{Real}\}$. Note that `self`, which denotes in the body of a method the receiver of the message, 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 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, this means that the receiver uses the method defined in its own class; if this minimum is strictly greater than C, then the receiver uses the method that its class, C, has *inherited* from that minimum. Note that the search for the minimum exactly corresponds to Smalltalk's "method look-up" where one searches for the least super-class (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, then so are messages. It then becomes possible to write functions (even overloaded) that take a message as 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.[7]

But, in the context of this paper, the most notable advantage of using overloaded functions is that it allows multiple dispatch[8]. As we hinted in the previous section, one of the major problems of the model with records is that it is impossible to combine, in a satisfactory way, 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 proposal of using a covariant subtyping rule for specialization. Let us recast the example for points with the method *equal*. In the record-based models, two-dimensional and three-dimensional points are modeled by the following recursive records:

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

Because of contravariance of arrow, the type of the field *equal* in *3EqPoint* is not a subtype of the type of *equal* in *2EqPoint*, therefore $\mathit{3EqPoint} \not\leq \mathit{2EqPoint}$.[9] Let us consider the same example in $\lambda\&$. We already defined the atomic types *2DPoint* and *3DPoint*. We can still use them since, contrary to what happens in the record case, the addition of a new method to a class does not change the type of its instances. In $\lambda\&$, a definition such as

$$\mathit{equal}{:}\,\{\mathit{2DPoint} \to (\mathit{2DPoint} \to \mathrm{Bool})\ ,\ \mathit{3DPoint} \to (\mathit{3DPoint} \to \mathrm{Bool})\}$$

[7]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.

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

[9]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 $2EqPoint \equiv \mu X\,.\,\langle\!\langle x{:}\,\mathrm{Int};\ y{:}\,\mathrm{Int};\ \mathit{equal}{:}\,X \to \mathrm{Bool}\rangle\!\rangle$

is not well defined either: *3DPoint*$\leq$ *2DPoint*, thus condition (3) —i.e. covariance— requires *3DPoint* $\to$ Bool $\leq$ *2DPoint* $\to$ Bool, which does not hold because of 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 reason why the type above must be rejected (in any case, it would be easy to write a term producing an error). However, in $\lambda\&$, it is possible to write a function that takes into account the types of two arguments for branch selection. For *equal*, this is obtained in the following way

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

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 by 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 substitute* the old branch but rather it *conceals* it from the objects of some given classes. Indeed, our formalization shows that the issue of "contravariance vs. 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 application: let f be a function of type $T \to U$, 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 *substitute* in the body of the function, 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 $T_1 \to T_2$, then 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 too (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).

Overriding is a totally different feature: say, we have an identifier m (in the circumstances, a message) that identifies two functions $f : A \to C$ and $g : B \to D$ with A and B incomparable. This identifier can be applied to an expression e, and the meaning of this application is f applied to e if e has a type smaller than A (in the sense of substitutivity explained above), else g applied to e if e has type smaller than B. Suppose now that $B \leq A$. The application in this case is solved by selecting f if the type of e is included between A and B, else g is selected if the type is smaller than or equal to B. But there is a further problem: the types may decrease during computation. Thus, it may happen that the type checker sees e of 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 that can substitute C (in the sense of substitutivity above), i.e. $D \leq C$. You can call it covariance, if you like, but it must be clear that it is not a subtyping rule: g does not substitute 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, but we are giving a formation rule for sets of functions in order to ensure the type consistency of the computation. Indeed, 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.

But these arguments may still be too abstract for object-oriented practitioners. Thus, let us write it 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 to select the right 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 in object-oriented type systems? Take a message m and apply (or "send") it to n objects $e_1 \dots e_n$ where e_i is an instance of class C_i. Suppose that to select the method you want to take into account the classes of the first k objects only. This can be expressed, say, by the following notation

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

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 $\min_{i \in I}\{S_i \mid (C_1 \times \dots \times C_k) \leq S_i\}$ 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:

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

with $C_i \leq A_i$ 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 selection of the method. In record-based models, no argument at all is taken into account to select the method: the method is uniquely determined by the record (i.e. the object) that the dot selection is applied to. Thus, in these models, it appears that you can have only contravariance and no covariance condition.

Strictly speaking, this 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 you "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 exactly corresponds to the covariant dependency stated by the covariance condition (3) [11], but its form is much more limited because it applies only to record types (since we "send" a label), and not to products (i.e. multiple dispatch) nor to arrows. This is shown by the fact that a record-label ℓ can be seen as a potentially infinitely branching overloaded function that takes as argument any record with *at least* a field labeled by ℓ and returns a value of the corresponding type:

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

[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 object that a message (label) is sent to, the smaller the type of the result.

Note then that covariance condition (3) is respected since $\langle\!\langle \ell{:}T \rangle\!\rangle \leq \langle\!\langle \ell{:}T' \rangle\!\rangle$ implies $T \leq T'$. But all the types of the arguments are records of the same form ... no other kind of type is allowed.

However the idea is that "explicit" covariance without multiple dispatching does not exist. Actual record-based models do not possess multiple dispatching. But, this does not mean that the analogy "objects as records" is incompatible with multiple dispatching. The problem is 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 obtained, it will be possible to perform multiple dispatch and thus we will recover the covariance relation. Thus, we show by an example that covariance and contravariance can cohabit in type-safe systems based on the "objects as records" analogy.

The key point is that records can be encoded in $\lambda\&$. Thus, by using this encoding, we can mimic any model based on simple records, but with an additional benefit: we also have overloaded functions. For the purposes of this paper, simple records suffice. Let us recall their encoding in $\lambda\&$ as given in [CGL95]

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:

$$\begin{array}{lll}
\langle\!\langle \ell_1{:}\,V_1; \ldots; \ell_n{:}\,V_n \rangle\!\rangle & \equiv & \{L_1 \to V_1, \ldots, L_n \to V_n\} \\[1em]
\langle \ell_1 = M_1; \ldots; \ell_n = M_n \rangle & \equiv & (\varepsilon \;\&\; \lambda x^{L_1}.M_1 \;\&\ldots\&\; \lambda x^{L_n}.M_n) \qquad {\scriptstyle (x^{L_i} \notin FV(M_i))} \\
M.\ell & \equiv & M\bullet\ell
\end{array}$$

In words, a record value is an overloaded function that takes as argument a label —each label belongs to a different type— that is used to selected 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, then 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 $\lambda\&$. Just remember that 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 code by knowing the type of just one argument. The solution in the overloading-based approach was to use multiple dispatching and to select the code 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:

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

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

[12] There is a "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.

Note that now *3EqPoint*$\leq$*2EqPoint*. But the objection may be raised that when we define the class *2EqPoint* the class *3EqPoint* may not exist yet, and so it would be impossible to define in the method *equal* for *2EqPoint* the branch for *3EqPoint*. But note that a lambda abstraction can be considered as the special case of an overloaded function with only one branch and, thus, that an arrow type can be considered as an overloaded type with just one arrow (it is just a matter of notation; see section 4.3 of [Cas94]). Thus, one 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 *3EqPoint*$\leq$*2EqPoint* holds. An example of objects with the types above is

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

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

where **Y** is the fixpoint operator (which is encodable in $\lambda\&$: see [Cas94]).

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 plain 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, in order to make it type-safe with binary methods, one has to specialize more accurately a method by defining its behaviour not only for the objects of the new class, but also for all possible combinations of the new objects with the old ones.

Finally, we want to stress once more 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 return the same type Bool. But imagine that instead of the method for *equal* we had a method *add*: then we would have had 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 multi-method[14] *add* in $3AddPoint$ are related in a covariant way, since the classes of their arguments do determine the code to be executed.

5 Conclusion

With this paper we hope to have contributed decisively to the debate about the use of covariance or contravariance. We tried to show that the two concepts are not antagonist, but that each one has its own use: covariance for specialization and contravariance for substitutivity. Also, we have tried to convey the intuition 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 due to a defect of this model but rather to a deficiency of all the calculi that used this model, which 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 take the covariance hidden in records out of its shell.

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. In that case, 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, which is *the* solution to the typing of binary methods.

The Golden rules

1. Do not use (left) covariance for arrow subtyping.
2. Use covariance to override parameters that drive the selection of the method.
3. When overriding a binary (or n-ary) method, specify its behavior not only for the actual class but also for its ancestors.

acks

I want to thank Véronique Benzaken who encouraged me to write this paper and Kathleen Milsted for her patient reading and many suggestions.

References

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

[14] A multi-method is a method that takes some objects as argument and selects the code according to the classes of these objects.

[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, 1984.

[Cas94] G. Castagna. *Overloading, subtyping and late binding: functional foundation of object-oriented programming.* PhD thesis, Université Paris 7, January 1994. Appeared as LIENS technical report.

[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.

[CM91] L. Cardelli and J.C. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1(1):3–48, 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.

[Dyl92] Apple Computer Inc., Eastern Research and Technology. *Dylan: an object-oriented dynamic language*, April 1992.

[Ghe91] G. Ghelli. A static type system for message passing. In *Proc. of OOPSLA '91*, 1991.

[GM89] J.A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report SRI-CSL-89-10, Computer Science Laboratory, SRI International, July 1989.

[GM94] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* The MIT Press, 1994.

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

[JKKM92] Jean-Pierre Jouannaud, Claude Kirchner, Hélène Kirchner, and Aristide Megrelis. OBJ: Programming with equalities, subsorts, overloading and parametrization. *Journal of Logic Programming*, 12:257–279, 1992.

[LRV88] C. Lécluse, P. Richard, and F. Vélez. O_2, an object-oriented data model. In *Proceedings of the ACM SIGMOD conference*, Chicago, Illinois, June 1988.

[MOM90] N. Martí-Oliet and J. Meseguer. Inclusions and subtypes. Technical Report SRI-CSL-90-16, SRI International, Computer Science Laboratory, December 1990.

[PW92] L.J. Pinson and R.S. Wiener. *Objective-C: Object-Oriented Programming Techniques.* Addison-Wesley, 1992.

[Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. In *16th Ann. ACM Symp. on Principles of Programming Languages*, 1989.

[Sch94] Michael Schwartzbach. Developments in object-oriented type systems. Tutorial given at POPL'94, January 1994.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *2nd Ann. Symp. on Logic in Computer Science*, 1987.