Contravariance for the Rest of Us

Warren Harris
Software and Systems Laboratory
HPL-90-121
August, 1990

e-mail address: harris@hpl.hp.com

contravariance; overloading;
CTT; object-oriented
programming
Introduction

Recent research has demonstrated that subtyping and inheritance are distinct relationships [1]. Primarily, the difference arises because of something called contravariance and its effects on object-oriented programming. Contravariance is a phenomenon which occurs as an interaction between subtyping and higher-order functions, and has important implications for object-oriented programming. It affects all object-oriented programming languages, including C++, and is usually circumvented by overloading. However, overloading does not always have the desired effect, which we will illustrate with actual C++ examples. Finally, we will discuss what a better – more expressive and type safe – language might look like.

What is contravariance?

We all have an intuitive notion of what it means for one type to be a subtype of another. We would expect that a value of a subtype can be used anywhere a value of a supertype is expected. Values of a subtype though can potentially do more, i.e. support a richer set of operations, than values of the supertype. The difference between the subtype and supertype reflects the increased functionality of the values. In some sense a subtype is more specific than its supertypes. What does it mean to be more specific?

Let us approach this question intuitively. From an implementation standpoint, a data structure is more specific if it has all the fields of its parent, but adds additional fields. From an interface standpoint, we would expect a data type to be more specific if it has all the operations of its parent, but adds additional operations. However, in object-oriented programming it is often necessary not only to add new operations, but also to restrict operations which are inherited. The question then arises: what does it mean for one operation to be more specific than another?

For simplicity’s sake, we can think of the operations on objects simply as functions (we will ignore the dispatching aspect of sending a message temporarily). We can now ask what it means for one function to be more specific than another. The type of a function is expressed in terms of the types of its arguments (if any) and the type of its result. We can summarize the subtype relationship between functions as:

The type of a function is a subtype of the type of another function
if (all else being the same)
the result type is more specific, or
any of the argument types are more general.  

\[1\]

This is also true of functions which return no values (void) in which case we simply ignore restrictions
Result types are said to be *covariant* – they vary in the same way as the function type. Result types must be more specific for the function type to be more specific. Argument types are said to be *contravariant* – they vary in the opposite way as the function type. Argument types must be more general for the function type to be more specific.

This seems counterintuitive. One would expect an operation defined over employees to be more specific than one defined over all people. The following example will illustrate why this is not true.

**Example**

The whole issue of contravariance comes into play when we manipulate functions from within programs. Functions which manipulate other functions are called *higher-order*. Higher-order functions typically are passed other functions as arguments and *apply* the functional argument to some values.\(^2\)

When a language involves subtyping, we become concerned about higher-order functions being passed functions which are subtypes of the type required. We would like to check that a function’s type is indeed a subtype of the required type and thereby verify that the program will not get runtime errors from being passed and subsequently invoking an inappropriate function.

This is a simple (contrived) example involving some subtypes and a higher-order function. Let us define a “person” to have a “name,” an “employee” to have a “salary” and inherit from person (thereby also having a name), and a “manager” to have someone s/he “manages” (to keep it simple, we will make this a single employee rather than a set) and also inherit from employee (thereby also having a name and salary). We will use C++ classes to specify some structural inheritance (i.e. all the fields from a superclass will also be available in a subclass):

```cpp
class Person
{
    public:
        char* name;
};

class Employee : public Person
{
    // code
}
```

on the results, and in functions which return multiple values in which case each of the results must be either the same or more specific.

\(^2\)Higher-order functions may also obtain a function to apply by other means – either as a piece of literal data, or by retrieving one from an external data structure.
public:
    int salary;
};

class Manager : public Employee
{
    public:
        Employee* manages;
};

Now suppose there exists a collection of functions over these data types. To keep it simple, we will define a set of print functions to print out various fields of the objects. Of course, we could just as well use member functions (methods), but regular functions will be sufficient to illustrate how contravariance works:

    void print_name(Person* p)
    {
        cout << p->name;
    }

    void print_salary(Employee* e)
    {
        cout << e->salary;
    }

    void print_manages(Manager* m)
    {
        cout << m->manages->name;
    }

Now let us define a higher-order function (a function which takes another function as a parameter and applies it). The higher-order function do_with_banner could take an operation applicable to Employees (such as one of the print functions) and an instance which was at least of type Employee. It would first print some banner, then apply the function:

    void do_with_banner(void (*action)(Employee*), Employee* employee)
    {
        print_banner();
        (*action)(employee);
    }

Suppose there is a single distinguished Employee instance called employee_of_the_month:

    Employee* employee_of_the_month;
A working example of this simple function is:

```c
  do_with_banner(print_salary, employee_of_the_month);
```

Now, one would suspect that the following piece of code should signal a compile-time error:

```c
  do_with_banner(print_manages, employee_of_the_month);
```

because we have no way of knowing whether the employee of the month will be a manager or not until runtime (with a specific Employee instance).

Conversely, the following code should work just fine:

```c
  do_with_banner(print_name, employee_of_the_month);
```

because we know that employee_of_the_month will always at least be an Employee, and therefore will always have a name (inherited from the Person class).

From this example we can see that functions which are acceptable as arguments to the higher-order function `do_with_banner` must themselves take arguments of type `Employee`, or a more general type. The arguments to `print_name` are more general than the arguments to `print_salary`, therefore the type of the `print_name` function is more specific than the type of the `print_salary` function. The `print_name` function can be used anywhere `print_salary` can be used. In other words, in order to be used by `do_with_banner`, the function must at least be defined on `Employees` (i.e. take `Employees` or a more specific type as an argument). This is contravariance.

Ultimately contravariance has ramifications for object-oriented programming. We will examine this in the next section.

**How is contravariance relevant to object-oriented programming?**

Object-oriented programming’s message passing paradigm inherently involves higher-order functions. Even though the user may not write higher-order functions directly, messages act as higher-order functions that invoke individual methods according to the particular

---

3 C++ unfortunately does not allow this code to pass through the compiler even though it really should work. This is because it does not permit function subtyping at all. Functions must be of exactly the right type to be passed as arguments.
object involved. When objects are passed as arguments or returned as values, their methods are actually being passed around too, just as with higher-order functions.

Let us look at the message dispatch process in detail. When an object is sent a message with some arguments, a method which will handle the message is looked up. This method is associated with the particular object, and is usually fetched from a table that is accessible from the object. The method is then applied to the arguments and any result returned from the method is also returned from the message dispatcher to the caller. Therefore sending a message is calling a higher-order function.

Since arguments to a message ultimately become arguments to the method, and since the method is invoked from within the (higher-order) message dispatcher, method arguments are subject to contravariance.

Now, when we type check a method of a subclass that overrides a method of a superclass with the same name, we should observe the contravariance rule. This way we can guarantee that the new method will apply to everything that the overridden method applied to, and therefore the subclass can be used anywhere the superclass can be used. Basically:

A method of a subclass is more specific than the method it overrides from a superclass if (all else being the same) its result type is more specific, or any of the argument types are more general.

When all the methods of a subclass are equally specific or more specific than the methods of a superclass, the interface of the subclass (the method names and their types) is said to contain the interface of the superclass [2]. When one interface contains another, instances of that interface can be used wherever instances of the other interface are required. This notion of containment is exactly the same as the notion of subtyping.

This seems simple so far. However, in practice it is not always the case that we want the interface of a subclass to contain the interface of a superclass. What is important is to be able to inherit some methods from the parent class and restrict other methods which must be overridden in order to make the new class work. One case of this restriction is when arguments to methods must be more specific (be a subtype of the type of the corresponding argument in the parent class) in order for the new implementation to work properly. Since method arguments are contravariant, making them more specific actually

---

4 Whether or not this method lookup is done at runtime (as with C++'s virtual methods), or at compile time (as with its regular methods), the higher-order nature still exists. Contravariance still plays a crucial role in the type checking of methods.
causes the subclass interface not to contain the interface of the parent class. In other words, inheritance is not subtyping, at least in some cases.

Perhaps the most common occurrence of this phenomenon, where inheriting does not produce subtyping, is when a method must take an argument which is the same type as “self” (i.e. the type of “this” in C++). The following example will illustrate.

Example

The following example illustrates what we might like to achieve with some code that implements windows and presenters (windows which display an associated object). For convenience, we will write this code in C++ although C++ actually behaves a bit differently. Later we will describe this difference and what the programmer must do to get around it.

```c++
class Window
{
    public:
        virtual void insert(Window*);
    ...
};

class Presenter : public Window
{
    public:
        virtual void insert(Presenter*);
        virtual void layout();
    ...
};
```

The intention of this example is that Presenter's insert method override the method inherited from Window while at the same time introducing an additional restriction: Presenters can only have children added to them which are themselves Presenters. One might want to do this because insert will invoke another method (like layout) on each of the inserted children.

A problem arises with this interpretation of the above code in that the interface to Presenter no longer contains the interface to Window. This is because all Windows allow other Windows to be inserted as children, whereas Presenters only allow other Presenters. A Presenter cannot be passed to any arbitrary piece of code which expects

---

6In C++ we are not allowed to say “the type of this, however this may have been inherited.” The language EIFFEL does support this notion via “like Current.”
to receive a Window because it may try to add a child window to it which is a Window rather than a Presenter:

```cpp
Window* add_a_child(Window* w)
{
    Window* child = new Window();
    w->insert(child);
    return w;
}
```

In some sense, the definition of Presenter has taken away the insert operation inherited from Window. It is not really a subtype anymore because of this missing operation. It instead includes a more specific operation (also called insert) which only applies to other Presenters.

In actuality, C++ does not take away the inherited operation. Instead, it overloads the name “insert” and allows both definitions to exist simultaneously. Even though we read both methods as “insert,” the compiler treats them as two separate methods. It is in this way that C++ guarantees that subclasses satisfy the interface of the parent.

There is a problem with overloading, however. Even though the code will not get a runtime error because a Window was inserted as a child of a Presenter, what will happen is that the wrong method will be invoked (the inherited insert method). From within add_a_child the Window will indeed be inserted, but the layout method will not be called. Such a maneuver can seriously violate the intended semantics of the program.

Sometimes it is the case that we really do want to override a method, and restrict its usage. In these cases, the new class is not really a subtype of the parent.

In such cases, the compiler should not allow subclasses to be used wherever the superclass is specified. In the above example, the correctness of the program does in fact depend on Windows not being inserted as children of Presenters.

What do C++ programmers really do?

There are five ways in which C++ programmers typically circumvent the problem of subclasses not being subtypes, and overloading not performing what is actually desired:

1. Often times in C++ we are unfortunately inclined to loosen type restrictions. In this case we can change the argument to Presenter's insert method so that the Presenter class becomes:
class Presenter : public Window
{
    public:
    virtual void insert(Window*);
    virtual void layout();
    ...
};

The programmer must assume that at runtime insert will indeed be called with a Presenter rather than a Window. Then if Presenter operations are to be performed on the w parameter, "casts" must be used to short-circuit the type checker. As a result, the type checker performs the role of verifying that the programmer indeed declared what operations s/he was interested in (via casts to classes which support those operations) rather than verifying that the entire program hangs together as a consistent whole. This really nullifies much of the benefit of type checking.6

2. A cleaner solution in this case would be to define a third class from which Window and Presenter both inherit. This class, SimpleWindow, could provide everything Window provided except the insert method. Window and Presenter would then be disjoint classes each with their own version of insert, and the compiler would be able to detect that one is an unacceptable data type to a routine which expected the other.

This solution is infeasible when we consider that classes like Window are often contained in libraries and that it is not possible to repartition its set of methods so that we could inherit some and override others. A completely usable and type correct library would have to consist of a large number of classes, each containing a single method. These classes would then be combined together with multiple inheritance to form the desired classes. This is highly impractical, and defeats the primary benefit of object-oriented programming – ease of programming through inheritance.

3. Rather than trying to split the Window class into two portions so that we can inherit from the part we need, we could instead use private inheritance:

class Presenter : private Window
{
    public:
    virtual void insert(Presenter*);
    virtual void layout();
    ...
};

6In fact, several large C++ applications have been forced into this style of coding where all variables in the system are basically of the most general type (e.g. the NIH Class Library of Smalltalk-like classes). The safety of such applications leaves much to be desired.
Private inheritance allows the implementation of Window to be used inside the implementation of Presenter, but does not allow the Window methods to be available to clients of Presenter. Effectively, this makes Presenter inherit from Window, but not be a subtype of it. This is exactly what we want in this case – with one exception. Although clients of Presenter are completely protected from inadvertently invoking Window's insert method, the Presenter implementation itself is not. If inside one of Presenter's methods the insert method is invoked, the problem arises again. This is because Window's insert method is still privately available. Programs can thereby type check, but produce the wrong behavior at runtime.

4. Another solution which is often used is the encoding of runtime "type" information into objects. Routines like Presenter's insert would first check some sort of tag field within the object before proceeding to assume the object actually is a Presenter, even though the compile-time type information declared the object to be only a Window. Not only are such solutions time consuming to implement and decrease the performance of the running system, but also introduce the question of how to recover from type errors at runtime.

5. Perhaps the solution used most often is to further overload methods in order to keep unwanted methods from applying. In the Presenter example we would define yet another insert method:

```cpp
class Presenter : public Window
{
 public:
   virtual void insert(Window*);
   virtual void insert(Presenter*);
   virtual void layout();
   ...;
};
```

The first insert method, insert(Window*), would simply prevent the Window class's insert(Window*) from being used. This method would either ignore the attempt to insert, or signal some form of runtime error. The second insert method, insert(Presenter*), would actually implement the desired semantics.

This solution seems unsatisfying in that these dummy methods must be around at runtime simply because the compiler could not catch at compile time the cases where they would be invoked. A correct application should never call them. This solution also has problems in that the choice of whether to use the insert(Window*) method or the insert(Presenter*) is determined at compile time. This choice is based on the declared type of arguments at the call sites of insert rather than the actual type of the arguments at runtime. Since C++ preserves no type information at runtime, the programmer is forced into one of the previously mentioned solutions.
What else can be done?

Some of the problem with C++’s overloading mechanism stems from the fact that only the object can be used to discriminate methods at runtime (i.e. virtual methods). The types of all other arguments are factored away at compile time when the overloaded names are resolved. Single argument dispatch allows a simple table to be used for the method lookup process.

The language CLOS [3] allows any number of arguments to be used in the runtime method lookup process, and terms these multimethods. Multimethods also eliminate the problem with contravariance (i.e. that subclasses may not be subtypes) because, like C++, they overload message names. Multimethods defer the entire lookup process until runtime, not just the lookup associated with the “first” argument, and therefore permit many correct method invocations that C++ would reject.

Although multimethods are more general, they carry along with them all the same problems with overloading found in C++. Basically, if a more general method is not found which corresponds to the types of the actual parameters (obeying contravariance), a method from a superclass which is not a supertype may be used instead. As we have already seen, in most cases this method will not be able to preserve the intended semantics of an application, and in general is always the incorrect method to call. However, rather than immediately generating a “no applicable method” error, subsequent errors will arise which are much removed from the actual problem (e.g. such as sending a Window a layout message rather than disallowing the call to insert a Window into a Presenter in the first place).

With each CLOS method invocation, there must always be some method in the system with every formal parameter at least as general as each actual parameter in the invocation. Without a type checker, it is possible to have some actual parameters be more specific while others are too general, and consequently no method will be found at runtime. Programmers are left to visualize the cross-product of all possible parameter types, both to ensure that some method will exist, and to determine exactly which method will apply in a given situation. The simple conceptual model of inheriting methods from a class lattice can no longer be used.

Why has contravariance not been a problem before?

For one thing, contravariance only arises when subtyping is involved. Since languages like C do not have subtypes (i.e. the arrangement of types into a generalization/specialization hierarchy), contravariance does not come up as a problem.
Languages like SMALLTALK [4] and CLOS do indeed exhibit contravariant behavior, but types are not checked statically. At runtime it is possible to be get a type error because the wrong type of function was passed as an argument. This may not seem to happen in most working programs, but it is not possible to guarantee that it will not happen in general without essentially type checking. Sometimes certain bugs are not encountered for months or years simply because the right combination of data has not been encountered which would cause a certain portion of code or method body to be executed. When the faulty code is finally executed, a type error which could have been caught statically finally occurs. Also as a program becomes larger, it becomes increasingly difficult to ensure that portions of it (possibly written by different programmers) will work together reliably.

What can be done to make programming type safe?

Research underway at Hewlett-Packard is striving to make object-oriented programming type safe without being too restrictive as are C++ and SIMULA. In other words, we want to guarantee that a piece of code will not break at runtime because it was handed a piece of data of the wrong type. In order to do this we are careful to make a distinction between classes (which specify implementations) and types (which specify interfaces). By observing the rules of contravariance (and a few others), we can statically determine when a class is an acceptable implementation for a piece of code that expects a certain type.

Checking that certain pieces of code are type safe is only half of the problem though. We also desire that the language be expressive enough to concisely encode the problem we are trying to solve. This includes allowing generic code to be stored in libraries and reused. This is accomplished in two ways. The first is by allowing implementation (class) inheritance to be independent from interface inheritance (subtyping). The second is through the property of parametric polymorphism. Parametric polymorphism is the ability to parameterize a piece of code over the types that it can potentially handle. In some sense, it establishes constraints between the types in a piece of code. Parametric polymorphism can further be broken down into simple (unquantified) parametric polymorphism, bounded quantification, and \( f \)-bounded quantification. We will examine each of these features in turn.

Let us reconsider the Window and Presenter types to show how we can separate the subtype and subclass notions:

```
interface Window
{
  methods:
    insert(Self) returns Void
  ...
};
```
interface Presenter
{
    inherits: Window
    methods:
        layout() returns Void
};

These interface definitions define the operations available on the types Window and Presenter respectively. Window defines an insert operation (method) which take another Window as a parameter and return nothing. Self indicates that the same type as this interface is required. If the Window interface is inherited, the type Self will change to reflect the inheritance. In the case of Presenter, insert will be available but will require another Presenter as an argument.

At first the distinction between this and C++ may seem nominal, but it allows the type checker to insure that both the calls to insert one Window into another and to insert one Presenter into another will succeed, whereas attempting to mix the two types will be caught at compile time. This is because of the contravariant use of Self in a method signature.

Moreover, the type checker will catch an inadvertent mixing of the two types even if a Presenter class (a specific implementation of the Presenter interface) inherits most of the code from a Window class. The type checker can also determine that the programmer will have to supply a new insert method for Presenters because of the contravariant use of Self.

Here is what some working examples of insert might look like:

```plaintext
w1 : Window = make_Simple_Window(...);
w2 : Window = make_Bordered_Window(...);
w1.insert(w2);
```

```plaintext
p1 : Presenter = make_Column_Presenter(...);
p2 : Presenter = make_Graph_Presenter(graph1, ...);
p1.insert(p2);
```

As previously mentioned, parametric polymorphism can be used to parameterize a piece of code over the types that it can potentially handle. Using parametric polymorphism we can rewrite the do_with_banner function as:

```plaintext
function do_with_banner[T : TYPE](fn : T -> Void, arg : T) returns Void
```
This polymorphic function establishes a constraint that the type of the parameter to the `fn` argument must be the same as the type of `arg`. The square brackets specify the type parameter `T`, which is evaluated at compile time. We could use this function as follows:

```cpp
do_with_banner[Employee](print_name, employee_of_the_month);
```

or if we knew that in a certain section of code `employee_of_the_month` was bound to a `Manager`:

```cpp
do_with_banner[Manager](print_manages, employee_of_the_month);
```

However when writing reusable routines, it is often necessary not only to specify that two arguments must be the same type, but also to specify that that type must support at least a certain interface. This is because we know that the argument will be used in a certain way, such as being sent a specific message. The object had better be able to support that message. This can be done by what we call bounded quantification. Bounded quantification is just a way of saying that an object must be at least a certain type. For example:

```cpp
function add_a_child[Win : CONTAINS[Window]](w : Win) returns Win
{
    child : Window = make_Window();
    w.insert(child);
    return w;
}
```

Here, `CONTAINS[Window]` specifies that the type variable `Win` must be at least as specific as the type `Window`. We may now call `add_a_child` to add a child window to any `Window` or subtype of `Window` that contains the `Window` interface:

```cpp
w1 : Window = make_Window();
add_a_child[Window](w1);

w2 : Bordered_Window = make_Bordered_Window();
add_a_child[Bordered_Window](w2);
```

---

7It is possible that the explicit type application (e.g. to `Employee` or to `Manager`) at the call site can be eliminated. This is because in most cases it can be inferred from the arguments given that we know the function's signature.
where the Bordered_Window interface contains the Window interface. We could not however write:

```cpp
p1 : Presenter = make_Presenter();
add_a_child[Presenter](p1);
```

because as we have seen in the previous section, Presenter does not contain the Window interface because its `insert` method requires an argument which is too specific.

Interestingly, because of polymorphism this new definition of `add_a_child` knows that the result of calling `add_a_child` will be the same type as its argument. The C++ definition will only know that the result is a `Window*`.

Sometimes however, it is desirable to write functions which operate over not only all interfaces which contain a given interface, but also over all interfaces which are recursive in the same way, i.e. which inherit one another. In other words, these functions can operate on a class and its subclasses, rather than operate over a type and its subtypes. For this, we use what we call \textit{f-bounded quantification} \cite{5}. F-bounded quantification specifies that any implementation which was derived from a parent is an acceptable type for a function:

```cpp
function foo[Win : INHERITS[Window]](w1 : Win, w2 : Win) returns Void
{
    w1.insert(w2);
};
```

This function, \textit{foo}, type checks because \textit{w1} and \textit{w2} will always have compatible implementations. \textit{INHERITS[Window]} guarantees that both variables will either be \textit{Windows} or \textit{Presenters}, but not one of each:

```cpp
foo[Window](some_window, another_window);
foo[Presenter](some_presenter, another_presenter);
```

\textbf{Conclusions}

This article has shown how contravariance affects object-oriented programming. We have seen that contravariance only comes into play when subtypes and higher-order functions are involved, but that these are the exact conditions under which all object-oriented programming languages must operate. We have seen how overloading can be used to alleviate the problems associated with contravariance, but that it carries its own problems. Finally, it has been suggested what a better programming language might look like, one in which parametric polymorphism and the separation of implementations and interfaces plays a
crucial role. These ideas can be used to make object-oriented programming both safer and more expressive.

References

[1] Cook, W., Hill, W., Canning, P.
_Inheritance is Not Subtyping_
POPL '90 Proceedings, pp. 125-135
Report STL-89-17 (revision 1), Hewlett-Packard Labs, 1989

[2] Canning, P., Cook, W., Hill, W., Olthoff, W.
_Interfaces for Strongly-Typed Object-Oriented Programming_
OOPSLA '89 Proceedings, pp. 457-467
Report STL-89-6 (revision 1), Hewlett-Packard Labs, 1989

_Common Lisp Object System Specification_
SIGPLAN Notices, Special Issue, September 1988

_Smalltalk-80: the Language and Its Implementation_
Addison-Wesley, 1983

_F-Bounded Quantification for Object-Oriented Programming_
Proc. of Conf. on Functional Programming Languages and Computer Architecture, pp. 273-280
Report STL-89-5, Hewlett-Packard Labs, 1989