Type-safe covariant routine overriding for the general-purpose programming language

Abstract

The paper presents the problem with the covariant overriding of routine arguments and potential solution when the compiler can statically detect type-unsafe calls and provide clear diagnostics to software developers.

Keywords

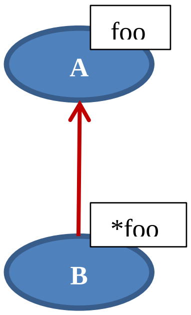
Variance, covariant redefinition-overriding, class, type safety, conformance.

CCS → Software and its engineering → Software notations and tools → General programming languages → Language types → Multi-paradigm programming languages

1. INTRODUCTION

Static typing is treated as the foundation for reliable software development which raises the productivity of the software developers and reduces the cost of the development. Such aspects as inheritance and polymorphism allow creating reusable and extensible software components. The ability to redefine or override (the latter term will be used in this article) inherited routines (procedures and functions) is another important aspect of efficient reuse. While we override the inherited routine we may keep its signature unchanged – that will be invariant overriding, or change it to the conformant signature – that will be covariant overriding or replace it with the signature to which the signature of the parent routine will conform to. The latter is known as contravariant overriding. More details are available in [2].

The term conformance (which is applied for types and signatures) can be explained with the help of a very simple picture. This picture also introduces the graphical notation and a code example used further in the paper. Terms like class and type are used as programmers used to use them.



```
class A {
    foo (a: T1)
}
class B inherits A {
    override foo(a: T2)
}
```

Picture 1.

Ovals represent classes (note, that every class defines a type). Arrows between ovals reflect inheritance relation between classes. So, B inherits A is the meaning of the picture and code. Class A has some routine foo with some signature and it is overridden in the descendant class with the new version of foo. This is the classic case of object-oriented programming: a base class, a derived class, a function of the base class is overridden with a function of the derived class.

Conformance is defined like this. If there is a path in the inheritance graph from class B to class A we say that type B conforms to type A meaning that it is possible to assign objects of type B to the objects of type A. As a routine signature is a group of types then one signature conforms to another when every type of one signature conforms to the corresponding type of the other signature.

The majority of programming languages support invariant overriding or support invariant overriding for routine arguments while keeping covariant overriding for the return type. But some languages risk to deal with covariant overriding for both arguments and return type. This form of variance goes well with the logic that when we provide a more specialized class we need also to have a more specialized signature for overridden routines. For example, Eiffel [1] and Dart [3] use covariant overriding.

But this approach has a very unpleasant implication known as *catcalls*. In fact, that is a breakage in the static typing. So, let's start with the problem description.

2. PROBLEM DESCRIPTION

The example in Picture 1 uses covariant routine overriding as type T2 conforms to the type T1 for the only argument of routine foo. Classes A and B as well as T1 and T2 can be compiled separately. Classes T1 and T2 look similar to A and B

```
class T1 { ... }
class T2 inherits T1 { ... }
```

It does not matter what we have inside of T1 and T2. Now consider the routine goo with two parameters of types A and T1 like this:

```
goo(a: A; x: T1) {
    a.foo(x)
}
```

The trap is ready. The call a.foo(x) can be compromised when we call goo. There are four possible ways to call goo selecting all type combinations for two arguments. All four are valid in terms of type conformance: B conforms to A, T2 conforms to T1 and every type conforms to itself. (If to scale to N>2 arguments the number of variants will just grow exponentially)

```
goo(new A, new T1)
goo(new A, new T2)
goo(new B, new T1)
goo(new B, new T2)
```

So, call to a.foo(x) replacing a and x with types will look like

A.foo (T1) – type-safe

A.foo (T2) – type-safe

B.foo (T2) – type-safe

B.foo (T1) – B's foo expects objects of type T2 or its descendants, not T1. This could lead to the program crash. 'Could' means that if the code of foo tries to access the routines and/or attributes specific for T2 only then the program crashes but if not, no crash may occur although an object of incorrect type was provided. So, the program behavior depends on the nature of the runtime check. If the check is done for all dynamic types of all arguments it dramatically reduces performance. If it is done only for dynamically dispatched calls then it works with the wrong types as described above. It seems it's quite hard to find a good runtime solution.

It is a rather dramatic situation: we have valid classes A, B, T1, and T2, and all calls to goo are aligned with the conformance test but boom our program is stopped with an error message that there is a catcall. So, we have to run a lot of tests to catch with help of testing all such calls. Who will use covariant routines overriding after that? Who will develop a reusable library with a very useful routine goo that may fail because it is called with some particular combination of types?

This is the serious reason why many programming languages do not support the covariant redefinition of routines. Their authors simply do not want millions of programmers to complain that the language is not type-safe.

The solution proposed nearly 35 years ago was to run a system-wide type check, to identify all sets of dynamic types for all variables and then check validity for all calls for all possible combinations of dynamic types. So, for the particular example, the set of dynamic types for an argument a in goo is {A, B}, for x is {T1, T2}. Therefore, it's necessary to check the validity of the call a.foo(x) for all possible type permutations. The problem is that code of goo was already compiled and maybe not available hence we call goo 'incorrectly'. We use separate compilation for years and rely on it. So, in practice compilers generate a call to runtime functions that check for catcalls and aborts the program execution with obvious negative consequences.

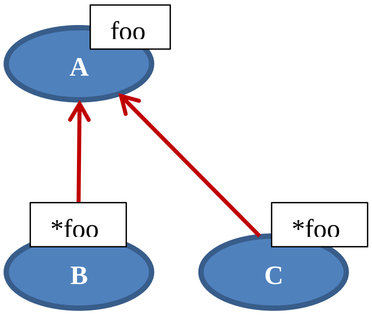
3. POTENTIAL SOLUTION

For the potentially dangerous call B.foo(T1) from the previous section, is there a version of foo that can handle this case? Well, A's foo has the proper signature, and it can process objects of type T1. This is just a previous version of foo in the inheritance graph, a precursor. So, a kind of naïve attempt to avoid run-time panic is to look for a precursor routine and call it passing B as this. As B is a descendant of A it is type-safe. The thing is that nobody expects such behavior.

On the other side, there is a logic behind: if a routine is defined in a more specialized class (descendant) and it cannot process the parameter then the object of the parent type should ask its precursor from the more general class to handle this situation. Of course, it is not that straightforward in the case of multiple inheritance as there could be several precursors for the particular routine and then recursively it will be necessary to identify the nearest common precursor going up the inheritance graph. Statically it is possible to determine such precursor and prepare run-time structures and proper behavior and/or generate necessary code to support this.

However, there is still a problem. If there is just an abstract routine at the top and this routine is this precursor then there is no routine to be called! Thus, this solution is not a universal one.

Picture 2 illustrates this case. Classes T1 and T2 stay unchanged and routine goo stays unchanged.



Picture 2.

```
abstract class A {
    abstract foo (a: T1)
}
class B inherits A {
    override foo(a: T2)
}
class C inherits A {
    override foo(a: T3)
}
class T3 inherit T1
end
```

Here we will have more catcalls

```
goo(new B, new T2) -> B.foo (T2)
goo(new B, new T3) -> B.foo (T3)
goo(new C, new T2) -> C.foo (T2)
goo(new C, new T3) -> C.foo (T3)
```

To conclude, it is perhaps better not to be too smart and report statically that some calls to goo are type-unsafe, stating that some particular types can not be used together within the body of goo.

4. PROPOSED SOLUTION

When checking the validity of the call to goo we need to know not only the signature of goo but also the list of ‘dangerous’ calls in the body of goo that should be checked for validity. Of course, this list is not visible for the developer, and whenever the implementation of goo is changed this list is to be checked for updates. But this only impacts the recompilation strategy which we do not consider here as it is a separate topic. So, the next question is should we add any routine call in the body of goo to the list of dangerous calls? Probably no: if the call does not use the routine arguments it should not be put into the list. If the call uses only one argument of goo it is also not a dangerous one because no type correlation can occur. Also, all routines with one argument including the target of the call are safe from the catcalls.

```
class X {
    foo() // catcall-safe routine
    goo(x:X) {
        //catcall-safe routine
        x.foo(x)
    }
}
```

Therefore, all routine calls within the routine body which use two or more routine arguments are to be registered.

```
goo(a: A; x: T1) {
   a.foo(x)
   // two ‘goo’ arguments used;
   // it is a dangerous call
}
```

This means that the internal interface description of routine goo could look like the example below (JSON notation is used):

```
{
  “name”: goo,
  “signature”: {{“_1”:”A”}, {“_2”,”T1”}},
  “bad calls”:{{_1.foo (_2)}}
}
```

For each call, we take static types of all parameters and try to verify if all dangerous calls when we replace placeholders (underscored number) with actual static types of parameters be valid in terms of type conformance. To make the example a bit more general we replace an explicit creation of objects with some expression of the same type. So, we deal only with types and conformance.

```
goo(exprB, exprT2) ->
    goo(B,T2)->
       _1 := B; _2 := T2 ->
          B.foo(T2)  is a valid call

goo(exprB, exprT3) ->
     goo(B,T3) ->
        _1 := B; _2 := T3 ->
           B.foo(T3)  is type invalid call,
                      compile-time error!

goo(exprC, exprT2) ->
     goo(C, T2) ->
        _1 := C; _2 := T2 ->
           C.foo (T2)  is type invalid call,
                       compile-time error!

goo(exprC, exprT3) ->
     goo(C, T3) ->
        _1 := C; _2 := T3 ->
           C.foo(T3)  is a valid call
```

Considering a more general case in the routine body we have a call in the form like

```
someRoutine (a1: T1; a2: T2; … an: Tn) {
    ...
    ai.someCall(
          ..., aj, ...,
          someFunction(... , ak, ...),
          ...
    )
    ...
}
```

So, the target of the call can be an argument (like a_i) or it has to be replaced just with the static type of other entity that is used as a target of the call. For parameters, of the dangerous call there are three possible situations:

1. An expression that is passed as a parameter to someCall does not use any someRoutine arguments: it’s a type-safe parameter. It can just be skipped.
2. a_j is used directly as a parameter: that could lead to type errors, so, this is to be checked
3. a_k is used as a parameter to a function call which is a parameter to someCall. In this case, the call to someFunction is to be analyzed for the type-safety, but it is not a threat for the call of someCall.

Then the list of dangerous calls for someRoutine will look like

```
{
  “name”: someRoutine,
  “signature”: {{“_1”:”T1”}, {“_2”,”T2”},
…},
  “bad calls”:{{_1.someCall (,,,_2,,,,
returnTypeOfsomeFunction}}
}
```

Such generalization allows building a list of dangerous calls for any routine. And the initial creation of the list can be started at parse time and later optimized by removing duplicating entries during semantics checks.

5. CONCLUSION

Decorating every routine with a list of dangerous calls allows detecting catcalls and generate precise compile-time errors pointing to a particular place in the source code programmer is developing and stating that particular types cannot be used together within the particular call. There is no more need for the system-wide type analysis. As a result, it is type-safe to use covariant routine overriding. Please note that this scheme does not cover the similar type breakage issue with the covariant conformance of generic types but this is a different issue.

6. REFERENCES

[1] Bertrand Meyer: Object-Oriented Software Construction, Second Edition. Prentice-Hall. ISBN 0-13-629155-4.

[2] Covariance and contravariance (computer science). https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science).

[3] Dart programming language. https://dart.dev/ and Frank Zammetti, Practical Flutter, Apress, 2019, ISBN 978-1-4842-4971-0.