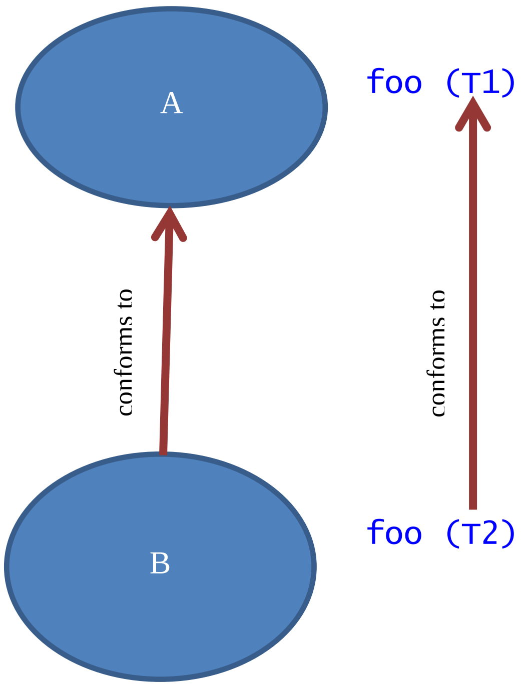
Call validity and run-time semantics in case of covariant overriding

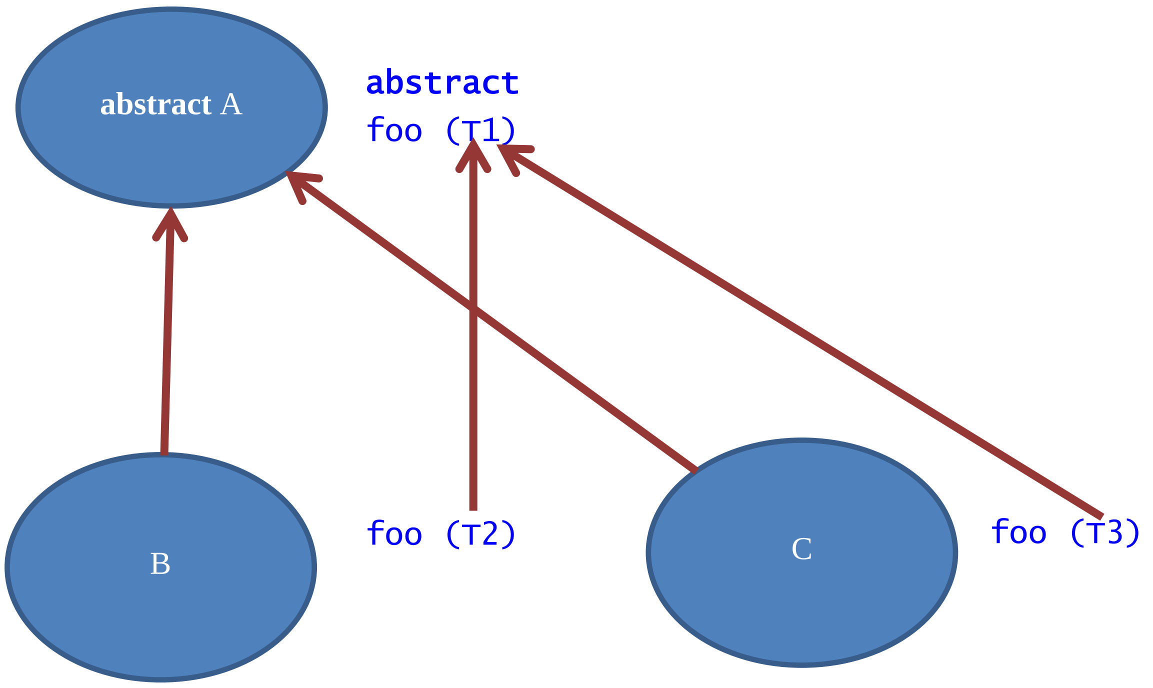
Covariant overriding of routines has some caveats and one of them leads to the type-unsafe code. This problem is known as catcalls. The solution, which was suggested in 1985 implies system-wide checks of every member call, for every combination of dynamic types of the call target and parameters the call should be valid. Such check is rather heavy but the problem is that it negatively impacts the paradigm of reusable software components, idea of separate compilation. And in the reality particular implementations of compilers for the programming language (Eiffel) which supports covariant overriding generate code (or use language runtime support) which runs run-time checks of catcall and if found raise a run-time error aborting program execution with a very subtle diagnostics which does not help to the programmer to find the right place in the code where the problem occurred and how to fix it.

Let's consider 2 simple examples that highlight the issue and then examine the solution which allows statically, at compile time to detect incorrect calls. The first example is very simple. We have 2 units (classes). Unit A has routine 'foo' with a single argument of type T1, and descendant of A, unit B overrides 'foo' introducing another 'foo' version with the conformant signature with type T2. That is the classical case of object-oriented programming – a base class, a derived class, a function of the base class which is overriden with a function of the derived class. The only difference is that instead of invariant overriding the conformant or covariant one is being used. We may safely compile both units. Then we can create procedure 'goo' which has 2 arguments of types A and T1. That is it. The trap is ready. We can also compile 'goo' and get some code for it. And then in the 3rd compilation unit, we start to use 'goo' calling it with different objects. There could be 4 cases (all type combinations for 2 arguments for the case) which will be fully valid in terms of type conformance. (If to scale to N arguments the number of variants will just grow) But some calls will cause a breakage in the type system in the code of 'goo', which was already verified and compiled. Catcall! That is a 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. So, below is the source code of the first example and a graph that illustrates the inheritance relation between types.



```
unit A
        foo (a: T1) do end
end
unit B extend A
        override foo (a: T2) do end
end
goo (a: A; x: T1) do
        a.foo (x)
// This call can be damaged outside of goo
end
goo (new A, new T1) /* A.foo to be called
with T1 as argument – type safe*/
goo (new A, new T2) /* A.foo to be called
with T2 as argument – type safe */
goo (new B, new T1) /* B.foo to be called
with T1 as argument – type UNSAFE!!! */
goo (new B, new T2) /* B.foo to be called
with T2 as argument – type safe */
```

For this example, we have routine 'foo' from A which may handle all calls to 'goo' as it will be absolutely type-safe to pass an object of type B as 'this' and call 'foo' version from A. But the second example shows that such solution is not a universal one. Unit at the top can be abstract (interface in Java terms) and then we simply has no type-safe routine to be called! Dead-end.



```
goo (a: A; x: T1) do
      a.foo (x)
      // This call can be damaged outside of goo
end
goo (new B, new T2) /* B.foo to be called with T2 as argument – type safe*/
goo (new B, new T3) /* B.foo to be called with T3 as argument – type UNSAFE!!!
*/
goo (new C, new T2) /* C.foo to be called with T2 as argument – type UNSAFE!!!
*/
goo (new C, new T3) /* C.foo to be called  with T3 as argument – type safe */
```

It is a very bad situation – the code of 'goo' is correct, every call to 'goo' looks valid but your program crashes. So, we need a mechanism that will allow for the compiler to identify that 2 calls to 'goo' second and third one are incorrect due to the type combination of actual parameters as such combination forces type breakage in the good body. So, that is a kind of semantics check – it is incorrect to call 'foo' from B passing T3 as a parameter. B was designed differently!

So, when we check validity of the call to 'goo' we need not only the signature of goo but a list of dangerous calls in 'goo' which should be checked for validity. Of course, this list is not visible for the programmer and every time when 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. Should we put any routine call in the body of 'goo' in the list of dangerous calls? Probably no, if the call does not use the routine arguments it should not be put into this list. If the call uses only one argument it is also not a dangerous one. So, all routine calls in the routine body which use 2 or more routine arguments are to be registered.

```
goo (a: A; x: T1) do
      StandardIO.print ("Safe call!!!")
      StandardIO.print (a.toString) // Only one argument is used – safe call
      a.foo (x) // 2 arguments used – dangerous call
```

```
        a.too (a) // Only one argument is used - safe call
end
```

That means that the internal interface description of routine goo should look like

```
{
        goo (_1 is A; _2 is T1)
        {
                _1.foo (_2)
        }
}
```

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

```
goo (exprOfTypeB, exprOfTypeT2) -> goo (B, T2) -> _1 := B; _2 := T2 ->
        B.foo (T2) is a valid call
goo (exprOfTypeB, exprOfTypeT3) -> goo (B, T3) -> _1 := B; _2 := T3 ->
        B.foo (T3) is type invalid call -> compile time error!
goo (exprOfTypeC, exprOfTypeT2) -> goo (C, T2) -> _1 := C; _2 := T2 ->
        C.foo (T2) is type invalid call -> compile time error!
goo (exprOfTypeC, exprOfTypeT3) -> goo (C, T3) -> _1 := C; _2 := T3 ->
        C.foo (T3) is a valid call
```

Decorating every routine with the list of dangerous calls allows to detect catcalls and generate precise compile-time errors pointing to a particular place in the source code programmer is developing stating that particular types can not be used together in the particular call. There is no need for the system-wide type analysis. As a result, it is type-safe to use covariant routine overriding in the case of multiple inheritance.