What are covariance and contravariance?

July 21, 2017

Subtyping is a tricky topic in programming language theory. The trickiness comes from a pair of frequently misunderstood phenomena called *covariance* and *contravariance*. This article will explain what these terms mean.

The following notation will be used:

- `A ≼ B` means `A` is a subtype of `B`.
- `A → B` is the type of functions for which the argument type is `A` and the return type is `B`.
- `x : A` means `x` has type `A`.

A motivating question

Suppose I have these three types:

```
Greyhound ≼ Dog ≼ Animal
```

So `Greyhound` is a subtype of `Dog`, and `Dog` is a subtype of `Animal`. Subtyping is usually transitive, so we'll say `Greyhound` is also a subtype of `Animal`.

Question: Which of the following types could be subtypes of `Dog → Dog`?

1. `Greyhound → Greyhound`
2. `Greyhound → Animal`

4. `Animal → Greyhound`

How do we answer this question? Let `f` be a function which takes a `Dog → Dog` function as its argument. We don't care about the return type. For concreteness, we can say `f : (Dog → Dog) → String`.

Now I want to call `f` with some function `g`. Let's see what happens when `g` has each of the four types above.

1. Suppose `g : Greyhound → Greyhound`. Is `f(g)` type safe?

No, because `f` might try to call its argument (`g`) with a different subtype of `Dog`, like a `GermanShepherd`.

2. Suppose `g : Greyhound → Animal`. Is `f(g)` type safe?

No, for the same reason as (1).

3. Suppose `g : Animal → Animal`. Is `f(g)` type safe?

No, because `f` might call its argument (`g`) and then try to make the return value bark. Not every `Animal` can bark.

4. Suppose `g : Animal → Greyhound`. Is `f(g)` type safe?

Yes—this one is safe. `f` might call its argument (`g`) with any kind of `Dog`, and all `Dog`s are `Animal`s. Likewise, it may assume the result is a `Dog`, and all `Greyhound`s are `Dog`s.

What's going on?

So this is safe:

```
            f: (Dog -> Dog) -> String

g: Dog -> Greyhound          f(g) is typeSafe
h: Dog -> GermanShephard     f(h) is typeSafe     As long as these
i: Animal -> Greyhound       f(i) is typeSafe    takes Dog/DogSuperclass
j: Animal -> Dog             f(j) is typeSafe    returns Dog/DogSubclass
k: Dog -> Dog                f(k) is typeSafe
```

The return types are straightforward: `Greyhound` is a subtype of `Dog`. But the argument types are flipped around: `Animal` is a *supertype* of `Dog`!

To state this strange behavior in the proper jargon, we allow function types to be *covariant* in their return type and *contravariant* in their argument type. Covariance in the return type means `A ≼ B` implies `(T → A) ≼ (T → B)` (`A` stays on the left of the `≼`, and `B` stays on the right). Contravariance in the argument type means `A ≼ B` implies `(B → T) ≼ (A → T)` (`A` and `B` flipped sides).

Fun fact: In TypeScript, argument types are *bivariant* (both covariant and contravariant), which is unsound (although now in TypeScript 2.6 you can fix this with `--strictFunctionTypes` or `--strict`). Eiffel also got this wrong, making argument types covariant instead of contravariant.

What about other types?

Question: Could `List<Dog>` be a subtype of `List<Animal>`?

The answer is a little nuanced. If lists are immutable, then it's safe to say yes. But if lists are mutable, then definitely not!

Why? Suppose I need a `List<Animal>` and you pass me a `List<Dog>`. Since I think I have a `List<Animal>`, I might try to insert a `Cat` into it. Now your `List<Dog>` has a `Cat` in it! The type system should not allow this.

Formally: we can allow the type of immutable lists to be covariant in its type parameter, but the type of mutable lists must be *invariant* (neither covariant nor contravariant) in its type parameter.

Fun fact: In Java, arrays are both mutable and covariant. This is, of course, unsound.

Type safe dimensional analysis in Haskell My hobby: proof engineering