Subclassing and Subtyping

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

You should be able to ...

The idea of a subtype and a subclass are very closely related, but there is a subtle difference we would like to cover.

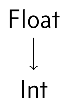
- Explain the difference between a subclass and a subtype.
- Explain the terms *covariant* and *contravariant*.
- Identify if two types have a subtyping relationship.

How do Types Relate?

- How can you tell if one type is a *subtype* of another?
  - Are integers subtypes of floats? (Or vice-versa?)
  - Characters / strings?
  - Squares / shapes?

How do Types Relate?

- How can you tell if one type is a *subtype* of another?
  - Are integers subtypes of floats? (Or vice-versa?)
  - Characters / strings?
  - Squares / shapes?
- An integer is a kind of float, so we can say that integer is a subtype of float.



Covariance

- Some types take parameters, such as lists and trees.
- If the subtype relationship varies according to the input type, the type is said to be *covariant*.
- "Most" types containing parameters are covariant.

$$\begin{tikzcd}
\text{Float} \arrow[d] & {[\text{Float}]} \arrow[d] \\
\text{Int} & {[\text{Int}]}
\end{tikzcd}$$

Functions

- Functions are an important exception!
  - The function type is covariant with respect to the output. If we are expecting a function that outputs a float, I can give you a function that outputs an integer without breaking anything. The reverse is not true!
  - The function type is *contravariant* with respect to the input. If we are expecting a function that takes a float, providing a function that takes an integer will fail or truncate the input.

$$\begin{tikzcd}
 & \text{Int} \to \text{Float} \arrow[dl] \arrow[dr] & \\
\text{Int} \to \text{Int} \arrow[dr] & & \text{Float} \to \text{Float} \arrow[dl] \\
 & \text{Float} \to \text{Int} &
\end{tikzcd}$$

The Trouble with Objects ...

Actually, there's more than just this one!

```
public class A {
  public A foo(A x) { ... }
  public A bar() { /* calls foo ... */ }
}
public class B : A {
  public B foo(B x) { ... }
}
```

- `B.bar` inherits from `A`.
- But `B.foo` overwrites `A.foo`.
- When `A.bar` calls `B.foo`, what will happen?

Conclusions

- Objects have a lot of flexibility and allow us to create useful abstractions.
- They can be implemented using functions. Users of functional programming languages tend to avoid them.
- These are useful enough in practice, and difficult enough to implement, that most modern languages now include them, including OCaml. (That's where the O comes from.)
- Inheritance can be tricky.