Subtyping

98-317: Hype for Types

February 19, 2019

1 Introduction

This week in lecture, we discussed several systems of subtyping. Some of the examples we gave, such as record and variant width and depth subtyping, are straightforward generalizations of relationships between types, and could be plausibly integrated into ML. (OCaml's *polymorphic variants* are an example of variant width subtyping in practice.)

However, most interesting subtype systems are seen in languages lying further from the ML tradition. Of the most commonly used programming languages, Java and its descendents[1] now have some of the most fanatic adherents of *inclusion polymorphism*.[2]

Therefore, we will use Java as the language for much of this assignment, though *knowledge of Java is not required* to complete it. If you have questions on the basic syntax of Java, feel free to ask on Piazza or use one of the numerous resources on the web.

[1] viz. Scala, Kotlin. The .NET languages also developed similarly rich systems in parallel.

[2] one of the forms of polymorphism; the others are *parametric polymorphism*, *ad-hoc polymorphism*, and *row polymorphism*.

2 Bounded Polymorphism

In Java, we may write an *interface* (similar to a *typeclass* in Haskell, *protocol* in Swift, *trait* in Rust, ...) expressing a type which has an associated comparison operation:

```
enum Order { Less, Greater, Equal }

interface Comparable {
  Order compareTo(Comparable other);
}
```

The interface defines one method `compareTo`, which takes an object `other` and returns an ordering. The type of `other` is constrained to be an instance of `Comparable`. In Java, because of the *subsumption* property, any subtype of `Comparable` may be implicitly treated as a `Comparable`.

It is now possible for us to write a class (which for our purposes is merely a record with one field) that can be compared:

```
class Foo implements Comparable {
  int foo;

  @Override public Order compareTo(Comparable other) {
    Foo o = (Foo) other;  // <- this is annoying
    return foo > o.foo ? Order.Greater :
      foo < o.foo ? Order.Less :
      Order.Equal;
  }
}
```

However, as you can see, the interface `Comparable` limits the information we have. In particular, we know only that the argument `other` is an instance of `Comparable`, and not that it ought to (if our comparisons are to be reasonable) be an instance of `Foo`. We must *downcast* the object to `Foo`, possibly causing an exception to be thrown at runtime.

2.1 Generics

We can improve the information we have available to us by annotating the `Comparable` interface with the actual type of the items to be compared. This invokes *parametric polymorphism*, and is roughly the same as placing a type parameter such as `'a` in ML onto the type.

```
/** @param <T> The type of the item to be compared. */
interface Comparable<T> {
  Order compareTo(T other);
}
```

Now, we can securely implement a comparison for our class `Foo`:

```
class Foo implements Comparable<Foo> {
  int foo;

  @Override public Order compareTo(Foo other) {
    return foo > other.foo ? Order.Greater :
      foo < other.foo ? Order.Less :
      Order.Equal;
  }
}
```

No more downcast is necessary, and we get the added advantage that comparing `Foo` with other `Comparable` objects will be a type error unless those objects are also `Foo`.

2.2 Interface Subtype

An issue arises if we attempt to use our new `Comparable` interface, however. Suppose we are defining a new interface `Container`, such that `Containers` are also `Comparable`. We might write the following, using the `extends` keyword to indicate subtyping:

```
interface Container<T> extends Comparable<T> {}
```

Be aware: `T` is not the type of the elements of the container (which remain abstract). It is instead the type of the container itself! Now we implement the interface:

```
class List implements Container<List> {
  @Override public Order compareTo(List other) {
    // implementation
  }
}
```

This works just fine! But there is a problem. Nothing stops the user from writing something like this:

```
class Bar implements Container<Foo> {
  @Override public Order compareTo(Foo other) {
    // implementation
  }
}
```

This is valid code, but it makes no semantic sense. How do we compare `Bar` to `Foo`? What we want to do is *constrain* the ability of the user to implement `Container`, with the correct type parameter.

2.3 Bounds

It turns out that we can accomplish this constraint using *bounded polymorphism*. Java's system of generics allows us to limit the accepted range of type arguments for a generic type. Namely, there are two bounds:

```
<? extends T>
```

```
<? super T>
```

The first specifies that the type argument must be a *subtype* of `T`, and the second specifies that it must be a *supertype*.

In Java, the `?` wildcard is merely a type variable. The only difference between `?` and a named variable like `T` is that `T` is explicitly declared and scoped, whereas `?` is usable inline. The following OCaml and Java are roughly equivalent:

```
let l: 'a list = [1; 2; 3]
```

```
List<?> l = Arrays.asList(1, 2, 3);
```

First, let's check your understanding of bounded polymorphism.

Task 1. In one sentence, describe the difference between the following two types:

```
List<? extends Animal>
```

```
List<Animal>
```

Hint: There really is an important difference! Both describe "a list of `Animals`", but how can you construct an instance of one type, versus the other?

Solution: The first is a homogeneous list of one type of animal, and the second is a heterogeneous list of any animals.

Now, we will use bounds to solve the problem.

Task 2. Give a bound to the `Container` type that enforces the notion that a `Container` must implement a comparison to itself.[3] That is, we want to fill in the blank:

```
interface Container<__________> extends Comparable<T> {}
```

Your answer should make use of F-bounded polymorphism.

Hint: That means a bound which is self-referential in the quantifier. Of course, the answer to this question may give you a better idea of what F-bounded polymorphism is, rather than the other way around...

Solution: `T extends Container<T>`
This isn't perfect, but it's quite good.

3 Behavioral Subtyping

In the fanatic world of *object-oriented programming*, the idea that subtypes should describe behavior is upheld as a maxim. That is, if we consider the principle of subsumption:

> *If $e : \sigma$ and σ is a subtype of τ, then $e : \tau$.*

[3] Technically, Java's type system will probably allow some leakiness in that condition, but there is a clean solution that clearly prevents the `Foo`/`Bar` problem above.

then we may generalize it to a description of observable behavior, known as the Liskov substitution principle:

> *If $\phi(e)$ is a property provable about all $e : \tau$, and σ is a subtype of τ, then $\phi(e')$ must be true for all $e' : \sigma$.*

We may rewrite that in a more (arguably excessively) succinct way:

> *Every σ is a τ.*
>
> Note that the converse is not necessarily true!

But what does it mean for a σ to be a τ? We may define it through *substitution*, i.e. to say that it must be possible for an instance of σ to be freely substitutible where a τ is expected while maintaining the correctness of the program.

Programmers versed in theory recognize such a claim as immediately undecidable in general, and possibly unwieldy in practice. Nevertheless, a large part of OOP software design is in service of the LSP. We will now ask whether the LSP is a good candidate for a "reasonable interpretation of subsumption".

Consider a type hierarchy (class hierarchy) of two objects: `Square`s and `Rectangle`s, defined intuitively.

Task 3. Give a reasonable argument, under the Liskov substitution principle, that `Square` should be considered a subtype (subclass) of `Rectangle`.

Task 4. Give a reasonable argument, under the Liskov substitution principle, that `Rectangle` should be considered a subtype of `Square`.

Hint: Depending on your first impression, one of these directions may be much easier than the other! Think about what interface we can write for `Square` that `Rectangle` should be able to easily implement, and vice versa. Mutability matters.

Solution: Squares are mathematically rectangles. Any routine which draws rectangles on a screen should be able to draw squares.
Rectangles expose a greater set of functionality. Namely, a rectangle can have its width and height set independently. Therefore, they can be seen as a specialization of squares ("quadrilaterals with one free dimension"), having an additional dimension field.
Note that the crux of this answer is that almost any object has mutable fields (or both getters and setters) in OOP-style code. We argued in class that ref cells, being a junction of a source and a sink for a type, must be invariant to ensure safety. Likewise, a strict interpretation would be that "behavioral subtyping" is almost certainly broken when given an object that has both getters and setters; the rectangle/square argument is just a counterintuitive example of this phenomenon.

Given the ability to compellingly argue that `Square` should be considered a subclass of `Rectangle` and that `Rectangle` should be a subclass of `Square`, we must conclude that they are equal!

Well, okay, that would be ridiculous. But, we will leave you to formulate your own opinion

of behavioral subtyping: whether a subtyping system should be considered sound as long as it is behaviorally consistent, or considered complete even if it is not.

4 Row Polymorphism

The Objective Caml programming language, designed in the intense weeks after the storming of the Bastille and the toppling of the French king Louis XVI, was infused with the republican principles of the revolution: *"fortement typé, strictement évalué, objet orienté!"*

Despite the visionary goals of the the revolutionary organization INRIA, Objective Caml, a.k.a. OCaml was suppressed during the reign of Napoleon, a notorious lover of lazy programming languages. The Congress of Vienna finally resulted in the standardization of OCaml as a language, though several grueling decades of war had decimated the ranks of the object-oriented programmers. Contributing to the loss of popularity was OCaml's interesting approach to objects: a system known as *row polymorphism*. We now study row polymorphism and how it affected the political, economic, and cache-coherency climate of western Europe in the nineteenth century.

Though the influence of the Jacobins on OCaml remains disputed, historians agree that anti-royalist sentiment led to the addition of an *object system* into OCaml, complete with classes and inheritance. Unlike the maritime republics, whose interactions with the inhabitants of the island Java (present-day Indonesia) led to a discipline of nominal, opaque objects, OCaml provides *object types* akin to records:

```
let counter : <get : int; incr : unit> = object
  val mutable n = 0
  method incr = n <- n+1
  method get = n
end
```

An object type is similar to a record in its naming and typing. However, it has one distinguishing feature termed *row polymorphism*, in which an object may be flexibly operated on using only a subset of its fields. The literature makes frequent reference to the contrast with Britain of this time period, whose relative conservatism limited the full development of so-called *flex records* in His Majesty's Standard ML. Namely, one can write functions like the following in OCaml (where `#` denotes method invocation):

```
let print_count (c : <get : int; ..>) = print_int c#get
```

And with such a function, provide a decadently many possible arguments to invoke different behaviors, as long as they uphold the basic object signature, regardless of other fields in the object:

```
let zero : <get : int> = object
  method get = 0
end in
print_count zero;
```

```
counter#incr;
print_count counter
```

Row polymorphism has been compared favorably to other political doctrines of the period, most notably *record width subtyping*. However, the landed gentry, who favored the preservation of the class-based system of inheritance that ensured the permanence of their generational wealth, resisted the system considerably.

Task 5. *Short response.* In 50 words or less, answer the following prompt. You may use any resource explaining the OCaml object system you wish, and use citations of OCaml code to support your argument.

> Despite the power of row polymorphism, in what ways is it deficient from true subtyping?

Hint: There are arbitrarily complex answers depending on how well you know OCaml, but think back to the example from class where we stored different kinds of entities in a collection. Does row polymorphism allow this to be achieved?

Solution: A heterogeneous list (or ref cell) cannot be created using row polymorphic objects, as they carry all their variable other fields with them.

5 Unsoundness of Java

Consider the following snippet of Java code.[4]

```
class Unsound {
  static class Constrain<A, B extends A> {}
  static class Bind<A> {
    <B extends A>
    A upcast(Constrain<A,B> constrain, B b) {
      return b;
    }
  }
  static <T,U> U coerce(T t) {
    Constrain<U,? super T> constrain = null;
    Bind<U> bind = new Bind<U>();
    return bind.upcast(constrain, t);
  }
  public static void main(String[] args) {
    String zero = Unsound.<Integer,String>coerce(0);
  }
}
```

It compiles under `javac`, version `1.8.0_25`. Upon execution, it produces a `ClassCastException`.

[4]From N. Amin and R. Tate, "Java and Scala's Type Systems are Unsound: The Existential Crisis of Null Pointers", *OOPSLA '16*.

Fun Task 6. Describe how this snippet of code exposes an unsoundness in the Java type system.

Hint: The inference of the type of the variable `constrain` is the culprit. The fact that it can be easily set to `null` does not do Java any favors.