Type Classes and Subtyping

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

October 5, 2015

With content by Arvind and Adam Chlipala.
Used with permission.

Hindley-Milner gives us generic functions

- Can generalize a type if the function makes no assumptions about the type:

```
const :: ∀ a b. a -> b -> a
const x y = x

apply :: ∀ a b. (a -> b) -> a -> b
apply f x = f x
```

- What do we do when we need to make an assumption?

A simple sum function

```
-- List data type
data [x] = [] | x : [x]

sum n []       = n
sum n (x:xs) = sum (n + x) xs
```

- `sum` cannot be of type `a -> [a] -> a`, we make use of the type (we need to know how to add to objects in the list).
- Pass in the notion of plus?

Avoiding constraints: Passing in +

```
sum plus n []     = n
sum plus n (x:xs) = sum (plus n x) xs
```

- Now we can get have a polymorphic type for sum

```
sum :: (a -> a -> a) ->
                  a -> [a] -> a
```

- When we call sum we have to pass in the appropriate function representing addition

Generalizing to other arithmetic functions

- A large class of functions do arithmetic operations (matrix multiply, FFT, Convolution, Linear Programming, Matrix solvers):
  - We can generalize, but we need +, -, *, /, …

- Create a Numeric "class" type:

```
data (Num a) = Num{
    (+) :: a -> a -> a
    (-) :: a -> a -> a
    (*) :: a -> a -> a
    (/) :: a -> a -> a
    fromInteger :: Integer  -> a
}
```

Generalized Functions w/ “class” types

```
matrixMul :: Num a -> Mat a -> Mat a -> Mat a
dft       :: Num a -> Vec a -> Vec a -> Vec a
```

- All of the numeric aspects of the type has been isolated to the Num type
  - For each type, we built a num instance
  - The same idea can encompass other concepts (Equality, Ordering, Conversion to/from String)

- Issues: Dealing with passing in num objects is annoying:
  - We have to be consistent in our passing of funcitons
  - Defining Num for generic types (`Mat a`) requires we pass the correct num a to a generator (`num_mat :: Num a -> Num (Mat a)`)
  - Nested objects may require a substantial number of “class” objects

Push “class” objects into type class

Type Classes

Type classes group together related functions (e.g., +, -) that are overloaded over the same types (e.g., Int, Float):

```
class Num a where
   (==), (/=)           ::   a -> a -> Bool
   (+), (-), (*)        ::   a -> a -> a
   negate               ::   a -> a
   ...

instance Num Int where
    x == y              = integer_eq x y
    x + y               = integer_add x y
       …
instance Num Float where ...
```

Type Class Hierarchy

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

class (Eq a) => Ord a where
   (<),(<=),(>=),(>) :: a -> a -> Bool
          max, min   :: a -> a -> a
```

- Each type class corresponds to one concept and class constraints give rise to a natural hierarchy on classes
- Eq is a superclass of Ord:
  - If type a is an instance of Ord, a is also an instance of Eq
  - Ord inherits the specification of (==), (/=) from Eq

Laws for a type class

- A type class often has laws associated with it
  - E.g., + in Num should be associate and commutative
- These laws are not checked or ensured by the compiler; the programmer has to ensure that the implementation of each instance correctly follows the law

more on this later

`(Num a)` as a predicate in type defintions

- We can view type classes as predicates
- Deals with all the passing we had to do in our data passing fashion
  - The type implies which objects should be passed in

Type classes is merely a type discipline which makes it easier to write a class of programs; after type checking the compiler de-sugars the language into pure λ-calculus

Subtyping

Related Reading

Chapter 15 of Pierce, “Subtyping”

Subtyping in Java: Primitive Types

```
void foo(int n) {
    float f = n;
    // ...and so on.
}
```

int ≤ float

Subtyping in Java: Interfaces

Nil ≤ List

Cons ≤ List

```
interface List {
      List append(List);
}
class Nil implements List {
      Nil() { }
      List append(ls) { return ls; }
}
class Cons implements List {
      private int data;
      private List tail;

      Cons(int d, List t) { data = d; tail = t; }
      List append(ls) {
            return Cons(data, tail.append(ls));
      }
}
```

Subtyping in Java: Inheritance

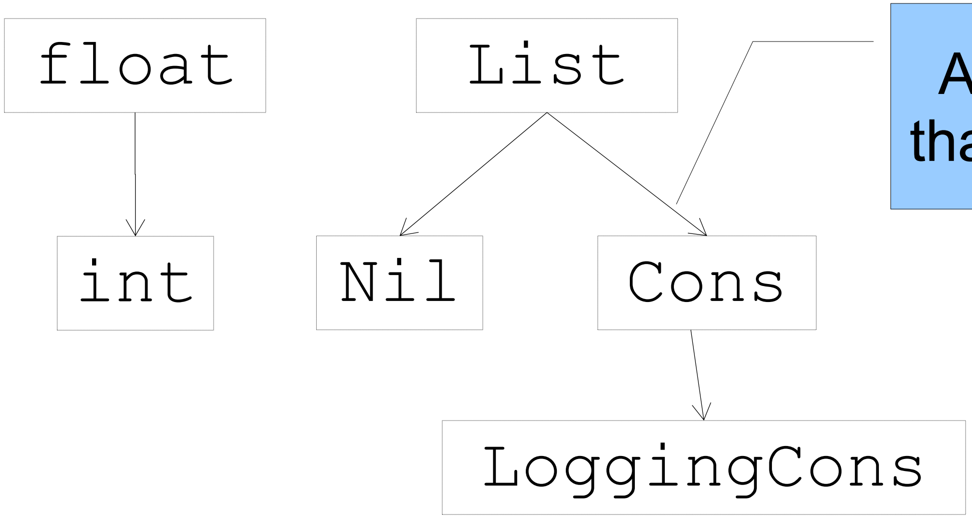
```
class Cons implements List {
    /* … */
}
class LoggingCons extends Cons {
    private int numAppends;

    LoggingCons(int d, List t) {
        super(d, t);
        numAppends = 0;
    }

    List append(ls) {
        ++numAppends;
        return super.append(ls);
    }
    int howManyAppends() { return numAppends; }
}
```

LoggingCons ≤ Cons

Subtyping as Graph Search



Arrow from A to B to indicate that B is a “direct” subtype of A

Q: How do we decide if A is a subtype of B?
A: Graph reachability! (Easy, right?)

We do need to think harder when the graph can be infinite.
(E.g., what about **generics**?)

Subtyping as a Formal Judgment

Reflexivity: $\dfrac{}{\tau \leq \tau}$ Transitivity: $\dfrac{\tau \leq \tau' \qquad \tau' \leq \tau''}{\tau \leq \tau''}$

Primitive rule: $\dfrac{}{\texttt{int} \leq \texttt{float}}$

Inheritance: $\dfrac{\texttt{class A extends B}}{\texttt{A} \leq \texttt{B}}$

Interfaces: $\dfrac{\texttt{class A implements B}}{\texttt{A} \leq \texttt{B}}$

This style of subtyping is called **nominal**, because the edges between user-defined types are all declared *explicitly*, via the *names* of those types.

What is Subtyping, Really?

Assume we have some operator $\llbracket \cdot \rrbracket$,
such that $\llbracket \tau \rrbracket$ is a mathematical **set** that represents τ.

[`int`] = $\mathbb{Z}$

[`float`] = $\mathbb{R}$

What's a natural way to formulate subtyping here?

$\tau_1 \leq \tau_2$ iff $\llbracket \tau_1 \rrbracket$? $\llbracket \tau_2 \rrbracket$

What about cases like:

```
struct s1 { int a; int b; }
struct s2 { float b; }
```

Is either of these a subtype of the other?

A More Helpful Guiding Principle

$$\tau_1 \leq \tau_2$$

if

Anywhere it is legal to use a τ_2,
it is also legal to use a τ_1.

Typing rule for subtypes

$$\frac{\Gamma \vdash e : \tau' \qquad \tau' \leq \tau}{\Gamma \vdash e : \tau}$$

Sanity-Checking the Principle

Primitive rule: $$\frac{}{\texttt{int} \leq \texttt{float}}$$

✔ Any integer N can be treated as N.0, with no loss of meaning.

Primitive rule: $$\frac{}{\texttt{float} \leq \texttt{int}}$$

✗ E.g., "`%`" operator defined for `int` but not `float`.

Primitive rule: $$\frac{}{\texttt{int} \leq \texttt{int} \rightarrow \texttt{int}}$$

✗ Can't call an `int`!

From Nominal to Structural

A **structural** subtyping system includes rules that analyze the *structure* of types, rather than just using graph edges declared by the user explicitly.

Pair Types

Consider types $\tau_1 \times \tau_2$,
consisting of (immutable) pairs of a τ_1 and a τ_2.

What is a good subtyping rule for this feature?
Ask ourselves: What operations does it support?

1. Pull out a τ_1.
2. Pull out a τ_2.

$$\frac{\tau_1 \leq \tau_1' \qquad \tau_2 \leq \tau_2'}{\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}$$

Jargon: The pair type constructor is **covariant**.

Record Types

Consider types like $\{ a_1 : \tau_1, \ldots, a_N : \tau_N \}$, consisting of, for each *i*, a field a_i of type τ_i.

What operations must we support?

1. For any *i*, pull out a τ_i from a_i.

Depth subtyping:

$$\frac{\forall i.\ \tau_i \leq \tau'_i}{\{\overrightarrow{a_i : \tau_i}\} \leq \{\overrightarrow{a_i : \tau'_i}\}}$$

Same field names, possibly with different types

Width subtyping:

$$\frac{\forall j.\ \exists i.\ a_i = a'_j \land \tau_i = \tau'_j}{\{\overrightarrow{a_i : \tau_i}\} \leq \{\overrightarrow{a'_j : \tau'_j}\}}$$

Field names may be different

Record Type Examples

$\{A : \texttt{int}, B : \texttt{float}\} \overset{?}{\leq} \{A : \texttt{float}, B : \texttt{float}\}$ **Yes!**

$\{A : \texttt{float}, B : \texttt{float}\} \overset{?}{\leq} \{A : \texttt{int}, B : \texttt{float}\}$ **No!**

$\{A : \texttt{int}, B : \texttt{float}\} \overset{?}{\leq} \{A : \texttt{float}\}$ **Yes!**

Depth:

$$\frac{\forall i.\ \tau_i \leq \tau'_i}{\{\overrightarrow{a_i : \tau_i}\} \leq \{\overrightarrow{a_i : \tau'_i}\}}$$

Width:

$$\frac{\forall j.\ \exists i.\ a_i = a'_j \wedge \tau_i = \tau'_j}{\{\overrightarrow{a_i : \tau_i}\} \leq \{\overrightarrow{a'_j : \tau'_j}\}}$$

Function Types

Consider types $\tau_1 \rightarrow \tau_2$.

What operations must we support?

1. Call with a τ_1 to receive a τ_2 as output.

Optimistic **covariant** rule:

$$\frac{\tau_1 \leq \tau'_1 \qquad \tau_2 \leq \tau'_2}{\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2}$$

Counterexample: `int` → `int` ≤ `float` → `int`

(λ x : `int`. x `%` 2) : `int` → `int`

Breaks when we call it with 1.23!

Function Types

Consider types $\tau_1 \rightarrow \tau_2$.

What operations must we support?

1. Call with a τ_1 to receive a τ_2 as output.

Swap order for function domains!

$$\frac{\tau'_1 \leq \tau_1 \qquad \tau_2 \leq \tau'_2}{\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2}$$

The function arrow is **contravariant** in the domain and *covariant* in the range!

Example: `float` → `int` ≤ `int` → `int`

Assume f : `float` → `int`

Build (λ x. f(intToFloat(x))) : `int` → `int`

Arrays

Consider types $\tau\texttt{[]}$.

What operations must we support?

1. *Read* a τ from some index.
2. *Write* a τ to some index.

Covariant rule:

$$\frac{\tau_1 \leq \tau_2}{\tau_1\texttt{[]} \leq \tau_2\texttt{[]}}$$

Counterexample:

```
int[] x = new int[1];
float[] y = x; // Use subtyping here.
y[0] = 1.23;
int z = x[0]; // Not an int!
```

Arrays

Consider types τ`[]`.

What operations must we support?

1. *Read* a τ from some index.
2. *Write* a τ to some index.

Contravariant rule: $$\frac{\tau_2 \leq \tau_1}{\tau_1\texttt{[]} \leq \tau_2\texttt{[]}}$$

Counterexample:

```
float[] x = new float[1];
int[] y = x; // Use subtyping here.
x[0] = 1.23;
int z = y[0]; // Not an int!
```

Arrays

Consider types τ `[]`.

What operations must we support?

1. *Read* a τ from some index.
2. *Write* a τ to some index.

Correct rule: None at all!

Only reflexivity applies to array types.

In other words, the array type constructor is **invariant**.

Java and many other "practical" languages use the covariant rule for convenience.
Run-time type errors (exceptions) are possible!

Subtyping Variance and Generics/Polymorphism

$$\texttt{List<}\tau_1\texttt{>} \overset{?}{\leq} \texttt{List<}\tau_2\texttt{>}$$

`List` and most other “data structures” will be **covariant**.

There are reasonable uses for **contravariant** and **invariant** generics, including mixing these modes across multiple generic parameters.

Languages like OCaml and Scala allow generics to be annotated with variance.

[Haskell doesn't have subtyping and avoids the whole mess.]

6.820 Fundamentals of Program Analysis
Fall 2015