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:

  \[\text{const} :: \forall a \ b. \ a \to b \to a\]
  \[\text{const} \ x \ y = x\]

  \[\text{apply} :: \forall a \ b. \ (a \to b) \to a \to b\]
  \[\text{apply} \ f \ x = f \ x\]

- 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 +

\[
\begin{align*}
& \text{sum \ plus \ n \ [] = n} \\
& \text{sum \ plus \ n \ (x:xs) = sum \ (plus \ n \ x) \ xs}
\end{align*}
\]

• Now we can get have a polymorphic type for sum

\[
\text{sum :: (a -> a -> a) -> a -> [a] -> a}
\]

• 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:

```haskell
data (Num a) = Num{
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  (/) :: a -> a -> a
  fromInteger :: Integer -> a
}
```
Generalized Functions w/ “class” types

\[
\text{matrixMul} :: \text{Num a} \rightarrow \text{Mat a} \rightarrow \text{Mat a} \rightarrow \text{Mat a} \\
\text{dft} :: \text{Num a} \rightarrow \text{Vec a} \rightarrow \text{Vec a} \rightarrow \text{Vec a}
\]

- 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 function
  - Defining Num for generic types (\text{Mat a}) requires we pass the correct num a to a generator (\text{num_mat} :: \text{Num a} \rightarrow \text{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):

```haskell
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 ...
```

October 5, 2015
Type Class Hierarchy

```haskell
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 associative 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 definitions

- 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 $\lambda$-calculus
Subtyping
Related Reading

Chapter 15 of Pierce, “Subtyping”
void foo(int n) {
    float f = n;
    // ...and so on.
}

int ≤ float
Subtyping in Java: Interfaces

```java
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));
    }
}
```

Nil $\leq$ List
Cons $\leq$ List
class Cons implements List {
    /* ... */
}

class LoggingCons extends Cons {
    private int numAppends;

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

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

    int howManyAppends() { return numAppends; }
}

Subtyping in Java: Inheritance
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: \( \tau \leq \tau \)

Transitivity: \( \tau \leq \tau' \) \( \tau' \leq \tau'' \) \( \tau \leq \tau'' \)

Primitive rule: \( \text{int} \leq \text{float} \)

Inheritance: \( \text{class A extends B} \)
\( A \leq B \)

Interfaces: \( \text{class A implements B} \)
\( A \leq B \)

This style of subtyping is called **nominal**, because the edges between user-defined types are all declared **explicitly**, via the **names** of those types.
Assume we have some operator $\cdot$, such that $[\tau]$ is a mathematical set that represents $\tau$.

\[
\begin{align*}
[int] &= \mathbb{Z} \\
[float] &= \mathbb{R}
\end{align*}
\]

What's a natural way to formulate subtyping here?

$\tau_1 \leq \tau_2$ iff $[\tau_1] \supseteq [\tau_2]$?

What about cases like:

```c
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 \( \tau_2 \), it is also legal to use a \( \tau_1 \).
Typing rule for subtypes

\[ \Gamma \vdash e : \tau' \quad \tau' \leq \tau \]

\[ \Gamma \vdash e : \tau \]
Sanity-Checking the Principle

**Primitive rule:** \[ \text{int} \leq \text{float} \]

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

**Primitive rule:** \[ \text{float} \leq \text{int} \]

✗ E.g., “%” operator defined for \text{int} but not \text{float}.

**Primitive rule:** \[ \text{int} \leq \text{int} \rightarrow \text{int} \]

✗ Can't call an \text{int}!
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 $\tau_1$ and a $\tau_2$.

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

1. Pull out a $\tau_1$.
2. Pull out a $\tau_2$.

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

*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 \(\tau_i\).

What operations must we support?

1. For any \(i\), pull out a \(\tau_i\) from \(a_i\).

Depth subtyping: 
\[ \forall i. \tau_i \leq \tau_i' \]
\[ \{a_i : \tau_i\} \leq \{a_i : \tau_i'\} \]
Same field names, possibly with different types

Width subtyping: 
\[ \forall j. \exists i. a_i = a'_j \land \tau_i = \tau'_j \]
\[ \{a_i : \tau_i\} \leq \{a'_j : \tau'_j\} \]
Field names may be different
Record Type Examples

\[
{A: \text{int}, B: \text{float}} \leq {A: \text{float}, B: \text{float}} \quad \text{Yes!}
\]

\[
{A: \text{float}, B: \text{float}} \not\leq {A: \text{int}, B: \text{float}} \quad \text{No!}
\]

\[
{A: \text{int}, B: \text{float}} \not\leq {A: \text{float}} \quad \text{Yes!}
\]

Depth:
\[
\forall i. \tau_i \leq \tau'_i \quad \Rightarrow \quad \{a_i : \tau_i \} \leq \{a_i : \tau'_i \}
\]

Width:
\[
\forall j. \exists i. a_i = a'_j \land \tau_i = \tau'_j \quad \Rightarrow \quad \{a_i : \tau_i \} \leq \{a'_j : \tau'_j \}
\]
Function Types

Consider types $\tau_1 \rightarrow \tau_2$.

What operations must we support?
1. Call with a $\tau_1$ to receive a $\tau_2$ as output.

Optimistic covariant rule:

\[
\frac{\tau_1 \leq \tau'_1 \quad \tau_2 \leq \tau'_2}{\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2}
\]

Counterexample: $\text{int} \rightarrow \text{int} \leq \text{float} \rightarrow \text{int}$

\[(\lambda x : \text{int}. \ x \% 2) : \text{int} \rightarrow \text{int}\]

Breaks when we call it with 1.23!
Consider types \( \tau_1 \rightarrow \tau_2 \).

What operations must we support?

1. Call with a \( \tau_1 \) to receive a \( \tau_2 \) as output.

Swap order for function domains!

\[
\frac{\tau'_1 \leq \tau_1 \quad \tau_2 \leq \tau'_2}{\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2}
\]

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

**Example:** \( \text{float} \rightarrow \text{int} \leq \text{int} \rightarrow \text{int} \)

Assume \( f : \text{float} \rightarrow \text{int} \)

Build \( (\lambda x. f(\text{intToFloat}(x))) : \text{int} \rightarrow \text{int} \)
Consider types $\tau[\ ]$.

What operations must we support?

1. *Read* a $\tau$ from some index.
2. *Write* a $\tau$ to some index.

Covariant rule: $\tau_1 \leq \tau_2 \Rightarrow \tau_1[\ ] \leq \tau_2[\ ]$

Counterexample:

```java
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 $\tau[\ ]$.

What operations must we support?

1. *Read* a $\tau$ from some index.
2. *Write* a $\tau$ to some index.

**Contravariant rule:** \[ \tau_2 \leq \tau_1 \implies \tau_1[\ ] \leq \tau_2[\ ] \]

Counterexample:

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

Consider types $\tau[\ ]$.

What operations must we support?
1. *Read* a $\tau$ from some index.
2. *Write* a $\tau$ 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!
List\(\langle \tau_1 \rangle \leq \text{List}\langle \tau_2 \rangle\)

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.]