Subtyping: It's Good!

vs.

Subtyping: It's Bad!

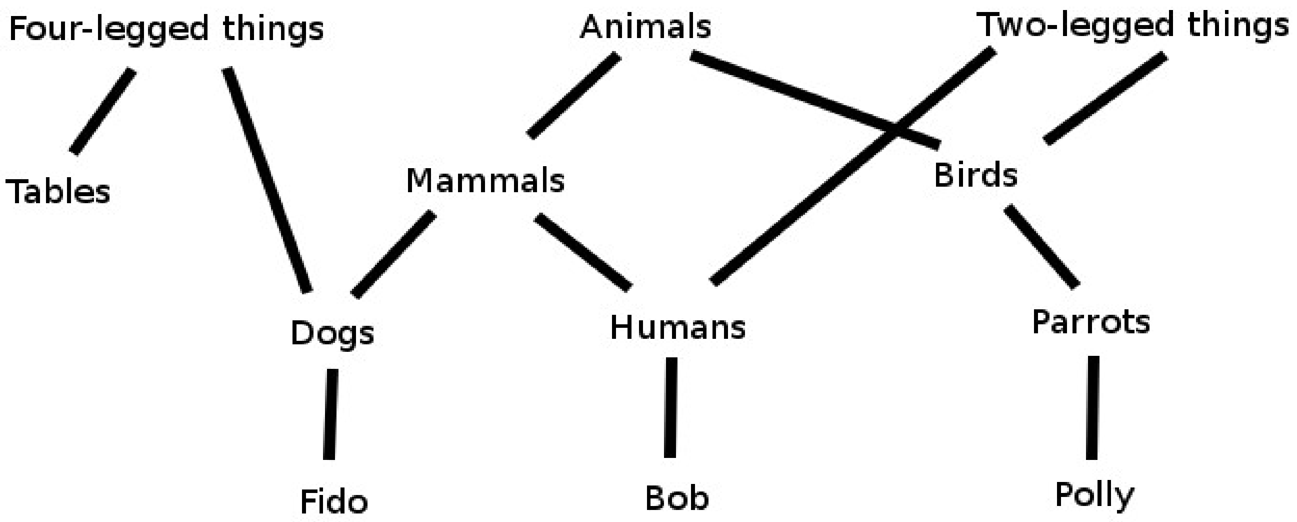
<:

Adrian King
22 September 2014

Part One

What Are Subtypes?

Overlapping Hierarchical Classifications



- We talk about things in the real world as if they were members of overlapping hierarchies
- All dogs are mammals; all mammals are animals
- This might be someone's mental model of some typical classifications
- Or a diagram from a book on OO

Substitutability

- The *intent* of subtyping is often explained in terms of the Liskov Substitution Principle:

Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

- So if something is true of mammals, it should be true of dogs

Subsumption

- The usual subtype operator is <:. So $S <: T$ means that S is a subtype of T
- The *mechanics* of subtyping are expressed in the rule of subsumption:

  $$\frac{\Gamma \vdash e : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash e : T}$$

  and in the rules that guarantee that subtypes are partially ordered (omitting the “$\Gamma \vdash$”s for concisness):

  $$\frac{}{S <: S} \qquad\qquad \frac{S <: U \quad U <: T}{S <: T} \qquad\qquad \frac{S <: T \quad T <: S}{S = T}$$

- Subsumption means that a given expression does not have a unique type; hence, “subtype polymorphism”

Function Application

- In functional programming terms, $S <: T$ generally means that a value of type S can be passed to a function declared to take a parameter of type T

- Other constraints imposed by substitutability might be of interest in more complicated systems (e.g., systems with mutable state), but here we develop our intuitions in terms of whether it makes sense to pass a value of one type to a function parameter of another type

Variance

- Subtyping of function types is contravariant in the input type and covariant in the output type:

$$\frac{I_2 <: I_1 \quad O_1 <: O_2}{I_1 \rightarrow O_1 <: I_2 \rightarrow O_2}$$

- Barbara Liskov will tell you that that's because, for a given invocation of a function f, you can substitute a function with fewer constraints on its input than f, or more constraints on its output, and the assumptions at the call site will still be satisfied

Subtyping of Recursive Types

- Type systems must allow type definitions to be recursive in order to be useful
- The usual notation for recursion looks like:

$$\mu t.x$$

  where x may contain t, and each t in x expands to µt.x

- Subtyping for recursive types:

$$\frac{\Gamma, X <: Y \vdash S <: T}{\Gamma \vdash \mu X.S <: \mu Y.T}$$

Top and Bottom

- A type system with subtyping can include special types at the top of the type hierarchy ($\top$, or Top) and at the bottom ($\bot$, or Bottom):

$$\frac{}{T <: \top} \qquad \frac{}{\bot <: T}$$

- Both of these types are optional in a type system (but it is more common to omit Bottom from a system than Top)

Part Two

What's So Great about Subtypes?

What Types Are For

- Types constrain computation
- The simply typed lambda calculus was invented to eliminate paradoxical or nonsensical behavior in the untyped lambda calculus, like infinite recursion
- Types also serve as bookkeeping to make sure programs treat data consistently
- But types are not just constraints; they also serve as commentary for human readers on the computations performed by the expressions they annotate

Types Are for People

POLYMORPHISM NOW!
WE DEMAND BOUNDED QUANTIFICATION
∀
∃
TYPES FOR THE PEOPLE
LIBERTY EQUALITY REFLEXIVITY
KEEP YOUR GOVERNMENT HANDS OFF MY DEPENDENT PRODUCTS

Subtypes Are for People

- Subtyping captures common-sense intuitions about how the world is organized
- Many common or desirable programming language features suggest subtyping:
  - Record types, OO-style object classes
  - (Untagged) union and (non-disjoint) intersection types
  - Refinements
  - Universes
  - Numbers
- If we can support subtyping for these things without introducing inconsistencies or excessive computational cost, why not do so?

Records

- Records are the classic textbook case in favor of subtyping:

```
fun f (r: {x: Int}) = … use r.x …

f({x = 17, y = false})
```

- Why should we care about the unused field y in the call to f?
- Fix by decreeing that:

```
{x: Int} <: {x: Int, y: Bool}
```

Deep Subtyping of Records

- If a field of a record r1 is a subtype of the corresponding field of r2, then we should be able to use r1 in place of r2:

```
fun f ({x: Int, r: {y: Bool, z: String}}) = …
f({x = 2, r: {y = true, z = "hi!", aa = 42}})
```

- The aa in the inner record gives the record passed to f a subtype of the parameter type in f's declaration. f doesn't use it, so this should be allowed

Record Formalities

- Consider record types with labels l_i and types T_i
- Record width subtyping:

$$\{l_i : T_i \text{ for } i \in 1..n+k\} <: \{l_i : T_i \text{ for } i \in 1..n\}$$

- Record depth subtyping:

$$\frac{\text{for } i \in 1..n,\ S_i <: T_i}{\{l_i : S_i\} <: \{l_i : T_i\}}$$

- Record types are indifferent to permutation of fields:

$$\frac{\text{for } i,j \in 1..n,\ \{k_j : S_j\} \text{ is a permutation of } \{l_i : T_i\}}{\{l_i : T_i\} = \{k_j : S_j\}}$$

Object Subtyping

- In statically typed object-oriented languages, you can think of classes as fancy record types, and subclasses as record types that have all the fields of their superclasses

- Many OO languages have additional machinery that turns this idea into a gross oversimplification, but the basic idea is usually there
- This talk has no more to say about OO

Unions and Intersections

- Programming languages with algebraic data types let us construct *tagged unions*:

```
data Wrapper =
   WrappedInt Int |
   WrappedBool Bool
```

- What if you could have *untagged* unions?:

$$S \cup T$$

- This works best in languages where all values are already tagged (typically, languages with a Top type)

- If you can have unions, why not intersections?

$$S \cap T$$

Union and Intersection Operations

- If a type is a set of values, a union of types is the union of the corresponding sets of values; likewise with intersections

- A value of S ∩ T belongs to both S and T; any operation allowed on either S or T is safe on S ∩ T
- A value of S ∪ T belongs to S or T; you can safely perform only operations allowed on *both* S and T
  - Unless you allow, say, a joint pattern match over constructors of both S and T

Union Rules, Intersection Rules

- Idempotence:

$$T \cup T = T \qquad\qquad T \cap T = T$$

- Commutativity:

$$T \cup U = U \cup T \qquad\qquad T \cap U = U \cap T$$

- Simple subtyping:

$$S <: S \cup T \qquad\qquad S \cap T <: S$$

- Combined subtyping:

$$\frac{T <: S \quad U <: S}{T \cup U <: S} \qquad\qquad \frac{S <: T \quad S <: U}{S <: T \cap U}$$

- Function types:

$$(T \to S) \cup (U \to S) = (T \cup U) \to S$$

$$(S \to T) \cap (S \to U) = S \to (T \cap U)$$

Refinement Types (aka Subset Types)

- A refinement type combines a base type with a proposition. For example:

  $$\{ n\text{: Nat} \mid n < 3 \}$$

  is the type of all natural numbers less than 3
- Of course:

  $$\{ n\text{: Nat} \mid n < 2 \} <: \{ n\text{: Nat} \mid n < 3 \}$$
- So when is one refinement type a subset of another? When both have the same base types, and the first proposition *implies* the second
- This is an immensely powerful concept. Let it soak in for a bit—we'll come back to it

Universes (aka the Infinite Tower of Types)

- A *universe* is a type whose elements are types
- You might suppose that Type: Type, that is, that Type (the type of types) is an element of itself
- However, in type systems strong enough to use for proving theorems (as in Coq, Agda, Idris), this naive formulation leads to inconsistencies. Instead, each universe at level *n*, Type_n, is an element not of itself, but of Type_{n+1}
- It is most convenient to treat all elements of Type_n as also belonging to Type_{n+1}; that is, $\text{Type}_n <: \text{Type}_{n+1}$

Numbers

- Numeric types might be the most widely known example of subtypes
- But also maybe the worst example of subtypes

- C programmers are used to thinking that all `int`s are `float`s, all `long`s are `double`s, and so on, because such numeric promotions occur without overflow
- But these promotions actual entail a change of representation, unlike most casts to a supertype
- Not to mention that conflating different types of numbers is mathematically dubious

Formal Types and Informal Types

- A given type system defines what it considers a type to be
- But the English word “type” is not so constrained, and can refer to any kind of categorization, formal or informal
- Perhaps such “folk types” should be invited into the formal type system, so we can use the system for communicating with folks

Folk Type Examples

- Consider a Haskell declaration of a List datatype:

```
data List a = Empty | Cons a (List a)
deriving (Show, Read, Eq, Ord)
```

- Haskell does not have a way to express directly “the type of all Conses”, that is, Lists built with a particular constructor
- Also note that “the type of all datatypes for which there is an Eq typeclass implementation” is not itself a type (although it can be expressed indirectly using a type variable)
- These are types that can be expressed in English but not Haskell, so they are Haskell folk types

Refined Folk

- Many folk types in simpler languages can be expressed as refinement types in languages with a more expressive type system. If we have (Coq syntax):

```
Definition isCons {X: Type} (l: list X): bool :=
  match l with
  | nil => false
  | cons _ _ => true
  end.
```

then we can build:

```
{ l: list X | isCons l = true }
```

Part Three

What's Wrong with Subtypes?

The Problems with Subtypes

- Many type systems for non-OO languages have no explicit way to express a subtype relation. Including such a relation in a type system can be complicated:
  - It can be more difficult to figure out the type of an expression
  - Subtyping may interact with other type system features to make typechecking undecidable
- If a language has an otherwise rich type system, it may be easiest to leave subtyping out

Types Are for Correctness

POLYMORPHISM NOW!
I OBJECT to objects
TYPECLASSES NOT HYPE CLASSES
TYPES WE CAN COUNT ON
∀
∃
I'VE MADE UP MY MIND ABOUT UNDECIDABILITY AND I SAY NO

Superfluous Subtypes

- The role of a type system is to ensure that programs are consistent with themselves
- Unnecessary complications to the type system make it harder for the compiler writer to ensure that the type system fulfills its purpose
- Anything that makes typechecking undecidable is *right out*

Unique Types

- A type system like Haskell's or ML's ensures that all types are disjoint
- This means that in a correctly typed program, exactly one type can be assigned to every term. This simplifies typechecking-related algorithms, like unification
- Subtyping breaks the simple rule; every value of a type T is also a value of all of T's supertypes
- So algorithms dealing with types must be phrased in terms of least upper bounds (joins) and greatest lower bounds (meets)

Meets and Joins

- A join (least upper bound) of two types S and T is written $S \cup T$. Similarly, their meet (greatest lower bound) is written $S \cap T$
- A join might not exist in a type system without Top; a meet might not exist in a type system without Bottom
- Some properties we require of joins and meets:

$$T \cup U = U \cup T \qquad\qquad T \cap U = U \cap T$$

$$S <: S \cup T \qquad\qquad S \cap T <: S$$

$$\frac{T <: S \quad U <: S}{T \cup U <: S} \qquad\qquad \frac{S <: T \quad S <: U}{S <: T \cap U}$$

- Hmm, anything look familiar here?

Bounded Quantification and Its Discontents

- In any language that includes both universal quantification (∀) and subtyping, the temptation is overwhelming to allow them to interact:

$$\frac{\Gamma \vdash S <: T \qquad \Gamma, X <: S \vdash U <: V}{\Gamma \vdash (\forall X{<:}T.U) <: (\forall X{<:}S.V)}$$

- Unfortunately, this rule makes typechecking undecidable
- The problem arises only for rather odd-looking terms—details can be found in Pierce's *Types and Programming Languages*—but undecidability is disconcerting to programming language designers

Oddness at the Bottom

- The Bottom type has unusual properties that complicate bookkeeping
- Generally, if a type does not have an arrow in it (or a type variable is not unified with an arrow type), it can't be applied to an argument. But Bottom is a subtype of all function types, so it *can* be applied

- Any assumption of the form X <: Bottom is actually an assumption that X = Bottom

Some Lies I Have Enjoyed Telling You

- I've been acting as if types are partially ordered; that is, if $A <: B$ and $B <: A$, then $A = B$
- More realistically, types form a preorder, and types that are mutual subtypes are not necessarily *equal* (as usual, equality is a troublesome concept)
- Many of the type rules shown can't be used directly in a compiler (not syntax-directed)
- Most of the rules should include some sort of validity check for types
- Worry about these things if you write a compiler

Part Three and a Half

Subtype Alternatives

(Kinda Sorta)

The Typeclass Solution: Constraining Types without Using Types

- Haskell achieves much expressivity using typeclasses:

```
member :: (Eq a) => a -> [a] -> Bool
member y [] = False
member y (x:xs) = (x == y) || member y xs
```

- In Prolog terms, available typeclass instances are like a database of facts
- Typeclass constraints on a function trigger a Prolog-style search for a matching typeclass instance
- But typeclasses are not types, nor are they terms that have types

Subtypeclassing

- Haskell allows typeclass constraints on typeclass declarations:

```
class (Eq a) => Num a where …
```

- This says that in order to create a Num instance for a, there must exist an Eq instance for a
- This sort of constraint is similar to the relationship between superclass (Eq) and subclass (Num) in OO systems: Num depends on Eq and is more specific than Eq

An Implicit Criticism of Typeclasses

- Although Scala has a rich object system that includes subtyping, it *also* has a notion of implicit conversions that has much the same effect as typeclasses:

  ```
  implicit def dbl2Cpx (d: Double) =
    Complex(d,0.0)
  implicit def pr2Cpx (ds: (Double,Double)) =
    Complex(ds._1,ds._2)
  ```

  which declares that a Double or pair of Doubles can be treated as a Complex (the methods of Complex play the role of the functions exported by a typeclass)
- The Complex class is *inside* the regular type system

Row Polymorphism

- Row polymorphism addresses the same problems as record subtyping, but with a different mechanism
- Imagine a variable that can range over sets of record fields
- A function might have a type like:

  foo: ∀r: *row without x*, {x: Int | r} → {| r}

  meaning that foo takes a record with a field x and some other fields r, and returns a record with the same fields r, but without x
- This gives you genericity in record types without a subtype in sight

Part Four

A Subtype

by Any Other Name

Curry and Howard and Their Amazing Correspondence

- Haskell Curry and William Alvin Howard noticed that a value V of a type T can be construed as a proof that T is inhabited
- (Lots of other people figured this out too, but Curry and Howard are the only ones who get the credit)
- So T can also be viewed as a proposition with proof V
- A function of type A → B is understood as a proof that A (as a proposition) implies B
- This works only for *total* functions (no exceptions!)
- Proof assistants like Coq are based on this idea

The Power of Dependent Types

- A dependent function type lets you mention the function argument's *value* in the function's output type
- Under the Curry-Howard correspondence, this means that proofs can go beyond propositional logic
- So you can prove theorems that mention variables of arbitrary types, as in (Coq syntax):

```
forall n m: nat, n + m = m + n
```

- This is the type of a *function* that takes two nats and returns a value of the *type* n + m = m + n

- A value of n + m = m + n is a *proof* that n + m = m + n

Building Type Refinements with Dependent-Type-Based Proofs

- The refinement type:

  ```
  { n: nat | n > 0 }
  ```

  can be encoded in a dependently typed language as a dependent pair, or a pair of consecutive function arguments. For example, the type of a Coq function that returns the predecessor of a positive number looks like:

  ```
  predecessor: forall n: nat, n > 0 -> nat
  ```

  which takes a natural number n and a proof that n is greater than zero (because zero has no predecessor), and returns n's predecessor

Refineder and Refineder

- A little syntactic sugar in Coq lets us use the familiar curly brace notation for refinements

- With that, we can create a predecessor function with an even more precise type than on the previous slide:

```
predecessor:
  forall n: nat, n > 0 -> {m : nat | n = S m}
```

- This type says that predecessor takes n and a proof that n is positive, and returns m such that n is m's successor

- This type completely specifies the behavior of predecessor—a type can't get more informative

But Wait,
What about the Subtyping?

- Dependently typed languages might not have an explicit notion of subtyping, but for refinement types built from dependent types, implication is just as good
- Instead of showing:

  { n: Nat | n < 2 } <: { n: Nat | n < 3 }

  we prove that:

  forall n: nat, n < 2 -> n < 3

  and *apply* the resulting proof to our proof of n < 2 to yield a proof of n < 3

Unions and Intersections by Proof

- Unions and intersections of refinement types are just types whose refinements are conjunctions or disjunctions. For example:

  ```
  { n: nat | 0 < n /\ n < 100 }
  ```

  is the intersection of:

  ```
  { n: nat | 0 < n }    and    { n: nat | n < 100 }
  ```

  where /\ is the Coq conjunction-of-propositions operator

- Now go back to the union and intersection rules page. Can you construct proofs that those rules are correct?

With Great Power
Comes Great Responsibility

- The change in mindset from subtyping to proving propositions gives you tremendous flexibility
- You can prove all sorts of interesting things, including things that have nothing to do with subtyping

- The price to be paid is that *you* now have to provide proofs of type claims that, in simpler type systems, the compiler proves for you
- (That should set your spider sense tingling)

You're the Decider

- Dependent type systems can express both parametric polymorphism (type parameters become ordinary parameters) and, using implication, subtyping

- So what happened to the undecidability problems that arose when we mixed subtyping and parametric polymorphism before? Are we still worried about them?
- No. Decidability is a property of algorithms, but constructing proof objects, although it can be done by an algorithm, is fundamentally *not algorithmic*
- The burden of proof rests on *you*

Part Five

So, Where Do We Go with All This?

Refinements Rule but Proofs Are Painful

- You might wonder just what the proof looks like for theorem:

  ```
  forall n: nat, n < 2 -> n < 3
  ```

  mentioned a few slides back. In Coq, it looks like:

  ```
  fun (n : nat) (H : n < 2) =>
    n_le_m__Sn_le_Sm n 2 (le_S n 1 (Sn_le_Sm__n_le_m n 1 H))
      : forall n : nat, n < 2 -> n < 3
  ```

  where n_le_m__Sn_le_Sm and Sn_le_Sm__n_le_m are lemmas that say n <= m -> S n <= S m and S n <= S m -> n <= m.

- Whew!

A Little Help, Please

- Imagine littering proofs like the one from the previous slide throughout your code

- It doesn't have to be that bad; for highly repetitive cases like type unions and intersections, you can write functions that generate proofs automatically
- This is not a convincing plan for more general proofs, as in the refinement types we've looked at
- Can use SMT solvers to automatically handle base types whose properties are well understood
- And then what?

Hybrid Typechecking Would Be Nice

- *Hybrid typechecking* is a technique that supplements compiletime typechecking with runtime membership type checks
- Hybrid typechecking is not airtight, but allows you to use highly expressive types without proving theorems
- Check out the experimental language Sage (https://sage.soe.ucsc.edu/)
- And my previous talk on hybrid typechecking (https://github.com/archontophoenix/hybridTypecheckingTalk/)

Can Refinements Be Both Reliable and Easy to Use?

- Sage allows non-total functions and doesn't track whether functions are total
- So a function in Sage is *not* a proof of an implication
- Idris (http://www.idris-lang.org/) tracks whether functions are total, but allows partial functions
- You can insist that functions treated as proofs be total, so that your proofs are reliable
- What if you could retrofit Idris with Sage-style runtime typechecking, or Sage with total function tracking?