```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Subtyping From the Beginning

Last major topic

Build up key ideas from first principles

- In pseudocode because:
  - No time for another language
  - Simple to first show subtyping without objects

Then, a few segments from now:

- How does subtyping relate to types for OOP?
  - Brief sketch only
- What are the relative strengths of subtyping and generics?
- How can subtyping and generics combine synergistically?

A tiny language

- Can cover most core subtyping ideas by just considering *records with mutable fields*

- Will make up our own syntax
  - ML has records, but no subtyping or field-mutation
  - Racket and Ruby have no type system
  - Java uses class/interface names and rarely fits on a slide

Records (half like ML, half like Java)

Record creation (field names and contents):

`{f1=e1, f2=e2, …, fn=en}` Evaluate **`ei`**, make a record

Record field access:

`e.f` Evaluate **`e`** to record **`v`** with an **`f`** field, get contents of **`f`** field

Record field update

`e1.f = e2` Evaluate **`e1`** to a record **`v1`** and **`e2`** to a value **`v2`**; Change **`v1`**'s **`f`** field (which must exist) to **`v2`**; Return **`v2`**

A Basic Type System

Record types: What fields a record has and type for each field

```
{f1:t1, f2:t2, …, fn:tn}
```

Type-checking expressions:

- If **`e1`** has type **`t1`**, …, **`en`** has type **`tn`**,
  then **`{f1=e1, …, fn=en}`** has type **`{f1:t1, …, fn:tn}`**
- If **`e`** has a record type containing **`f : t`**,
  then **`e.f`** has type **`t`**
- If **`e1`** has a record type containing **`f : t`** and **`e2`** has type **`t`**,
  then **`e1.f = e2`** has type **`t`**

This is safe

These evaluation rules and typing rules prevent ever trying to access a field of a record that does not exist

Example program that type-checks (in a made-up language):

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val pythag : {x:real,y:real} = {x=3.0, y=4.0}
val five : real = distToOrigin(pythag)
```

Motivating subtyping

But according to our typing rules, this program does not type-check

- It does nothing wrong and seems worth supporting

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val c : {x:real,y:real,color:string} =
   {x=3.0, y=4.0, color="green"}

val five : real = distToOrigin(c)
```

A good idea: allow extra fields

Natural idea: If an expression has type

```
{f1:t1, f2:t2, …, fn:tn}
```

Then it can *also* have a type with some fields removed

This is what we need to type-check these function calls:

```
fun distToOrigin (p:{x:real,y:real}) = …
fun makePurple (p:{color:string}) = …

val c :{x:real,y:real,color:string} =
    {x=3.0, y=4.0, color="green"}

val _ = distToOrigin(c)
val _ = makePurple(c)
```

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

The Subtype Relation

Keeping subtyping separate

A programming language already has a lot of typing rules and we do not want to change them

- Example: The type of an actual function argument must ***equal*** the type of the function parameter

We can do this by adding "just two things to our language"

- *Subtyping*: Write **`t1 <: t2`** for **`t1`** is a subtype of t2
- One new typing rule that uses subtyping:

  If **`e`** has type **`t1`** and **`t1 <: t2`**,

  then **`e`** (also) has type **`t2`**

Now all we need to do is define **`t1 <: t2`**

Subtyping is not a matter of opinion

- Misconception: If we are making a new language, we can have whatever typing and subtyping rules we want

- Not if you want to prevent what you claim to prevent [soundness]
  - Here: No accessing record fields that do not exist

- Our typing rules were *sound* before we added subtyping
  - We should keep it that way

- Principle of *substitutability*: If `t1 <: t2`, then any value of type `t1` must be usable in every way a `t2` is
  - Here: Any value of subtype needs all fields any value of supertype has

Four good rules

For our record types, these rules all meet the substitutability test:

1. "Width" subtyping: A supertype can have a subset of fields with the same types

2. "Permutation" subtyping: A supertype can have the same set of fields with the same types in a different order

3. Transitivity: If `t1 <: t2` and `t2 <: t3`, then `t1 <: t3`

4. Reflexivity: Every type is a subtype of itself

(4) may seem unnecessary, but it composes well with other rules in a full language and "does no harm"

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Depth Subtyping

More record subtyping?

[Warning: I am misleading you ☺]

Subtyping rules so far let us drop fields but not change their types

Example: A circle has a center field holding another record

```
fun circleY (c:{center:{x:real,y:real}, r:real}) =
   c.center.y

val sphere: {center:{x:real,y:real,z:real}, r:real} =
  {center={x=3.0,y=4.0,z=0.0}, r=1.0}

val _ = circleY(sphere)
```

For this to type-check, we need:

```
{center:{x:real,y:real,z:real}, r:real}
<:
{center:{x:real,y:real}, r:real}
```

Do not have this subtyping – could we?

```
{center:{x:real,y:real,z:real}, r:real}
                   <:
   {center:{x:real,y:real}, r:real}
```

- No way to get this yet: we can drop **`center`**, drop **`r`**, or permute order, but cannot "reach into a field type" to do subtyping

- So why not add another subtyping rule… "Depth" subtyping:
  If **`ta <: tb`**, then **`{f1:t1, …, f:ta, …, fn:tn} <:`**
  `{f1:t1, …, f:tb, …, fn:tn}`

- Depth subtyping (along with width on the field's type) lets our example type-check

Stop!

- It is nice and all that our new subtyping rule lets our example type-check

- But it is not worth it if it breaks soundness
  – Also allows programs that can access missing record fields

- Unfortunately, it breaks soundness ☹

Mutation strikes again

If **ta <: tb**,
then **{f1:t1, …, f:ta, …, fn:tn} <:**
{f1:t1, …, f:tb, …, fn:tn}

```
fun setToOrigin (c:{center:{x:real,y:real}, r:real})=
   c.center = {x=0.0, y=0.0}

val sphere: {center:{x:real,y:real,z:real}, r:real} =
  {center={x=3.0, y=4.0, z=0.0}, r=1.0}

val _ = setToOrigin(sphere)
val _ = sphere.center.z (* kaboom! (no z field) *)
```

Moral of the story

- In a language with records/objects with getters and setters, depth subtyping is unsound
  - Subtyping cannot change the type of fields

- If fields are immutable, then depth subtyping is sound!
  - Yet another benefit of outlawing mutation!
  - Choose two of three: setters, depth subtyping, soundness

- Remember: subtyping is not a matter of opinion

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Optional: Java/C# Arrays

Picking on Java (and C#)

Arrays should work just like records in terms of depth subtyping

- But in Java, if `t1 <: t2`, then `t1[] <: t2[]`
- So this code type-checks, surprisingly

```
class Point { … }
class ColorPoint extends Point { … }
…
void m1(Point[] pt_arr) {
  pt_arr[0] = new Point(3,4);
}
String m2(int x) {
  ColorPoint[] cpt_arr = new ColorPoint[x];
  for(int i=0; i < x; i++)
     cpt_arr[i] = new ColorPoint(0,0,"green");
  m1(cpt_arr); // !
  return cpt_arr[0].color; // !
}
```

Why did they do this?

- More flexible type system allows more programs but prevents fewer errors
  - Seemed especially important before Java/C# had generics

- Good news: despite this “inappropriate” depth subtyping
  - **`e.color`** will never fail due to there being no **`color`** field
  - Array *reads* **`e1[e2]`** always return a (subtype of) **`t`** if **`e1`** is a **`t[]`**

- Bad news: to get the good news
  - **`e1[e2]=e3`** can fail even if **`e1`** has type **`t[]`** and **`e3`** has type **`t`**
  - Array *stores* check the *run-time class* of **`e1`**'s elements and do not allow storing a supertype
  - No type-system help to avoid such bugs / performance cost

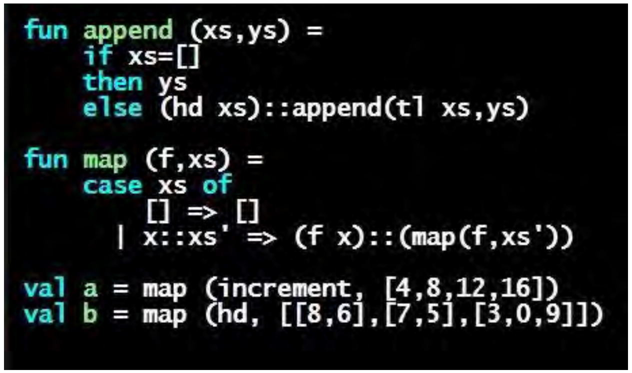
So what happens

```
void m1(Point[] pt_arr) {
  pt_arr[0] = new Point(3,4); // can throw
}
String m2(int x) {
  ColorPoint[] cpt_arr = new ColorPoint[x];
  …
  m1(cpt_arr); // "inappropriate" depth subtyping
  ColorPoint c = cpt_arr[0]; // fine, cpt_arr
    // will always hold (subtypes of) ColorPoints
  return c.color; // fine, a ColorPoint has a color
}
```

- Causes code in **`m1`** to throw an **`ArrayStoreException`**
  - Even though logical error is in **`m2`**
  - At least run-time checks occur only on array stores, not on field accesses like **`c.color`**

null

- Array stores probably the most *surprising* choice for flexibility over static checking
- But **`null`** is the most *common* one in practice
  - **`null`** is not an object; it has *no* fields or methods
  - But Java and C# let it have *any* object type (backwards, huh?!)
  - So, in fact, we do *not* have the static guarantee that evaluating **`e`** in **`e.f`** or **`e.m(…)`** produces an object that has an **`f`** or **`m`**
  - The "or **`null`**" caveat leads to run-time checks and errors, as you have surely noticed
- Sometimes **`null`** is convenient (like ML's option types)
  - But also having "cannot be **`null`**" types would be nice



Programming Languages

Dan Grossman

Function Subtyping

Now functions

- Already know a caller can use subtyping for arguments passed
  - Or on the result

- More interesting: When is one function type a subtype of another?

  - Important for higher-order functions: If a function expects an argument of type **`t1 -> t2`**, can you pass a **`t3 -> t4`** instead?

  - Coming next: Important for understanding methods
    - (An object type is a lot like a record type where “method positions” are immutable and have function types)

Example

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flip p = {x = ~p.x, y=~p.y}
val d = distMoved(flip, {x=3.0, y=4.0})
```

No subtyping here yet:

- **`flip`** has exactly the type **`distMoved`** expects for **`f`**
- Can pass in a record with extra fields for **`p`**, but that's old news

Return-type subtyping

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flipGreen p = {x = ~p.x, y=~p.y, color="green"}
val d = distMoved(flipGreen, {x=3.0, y=4.0})
```

- Return type of **`flipGreen`** is **`{x:real,y:real,color:string}`**, but **`distMoved`** expects a return type of **`{x:real,y:real}`**
- Nothing goes wrong: If **`ta <: tb`**, then **`t->ta <: t->tb`**
  - A function can return "*more* than it needs to"
  - Jargon: "Return types are *covariant*"

This is wrong

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flipIfGreen p = if p.color = "green" (*kaboom!*)
                    then {x = ~p.x, y=~p.y}
                    else {x = p.x, y=p.y}
val d = distMoved(flipIfGreen, {x=3.0, y=4.0})
```

- Argument type of **`flipIfGreen`** is **`{x:real,y:real,color:string}`**, but it is called with a **`{x:real,y:real}`**
- Unsound! **`ta <: tb`** does **NOT** allow **`ta -> t <: tb -> t`**

The other way works!

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flipX_Y0 p = {x = ~p.x, y=0.0}
val d = distMoved(flipX_Y0, {x=3.0, y=4.0})
```

- Argument type of **`flipX_Y0`** is **`{x:real}`** but it is called with a **`{x:real,y:real}`**, which is fine

- If **`tb <: ta`**, then **`ta -> t <: tb -> t`**
  - A function can assume "*less* than it needs to" about arguments
  - Jargon: "Argument types are *contravariant*"

Can do both

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flipXMakeGreen p = {x = ~p.x, y=0.0, color="green"}
val d = distMoved(flipXMakeGreen, {x=3.0, y=4.0})
```

- `flipXMakeGreen` has type

  `{x:real} -> {x:real,y:real,color:string}`
- Fine to pass a function of such a type as function of type

  `{x:real,y:real} -> {x:real,y:real}`
- If `t3 <: t1` and `t2 <: t4`, then `t1 -> t2 <: t3 -> t4`

Conclusion

- If `t3 <: t1` and `t2 <: t4`, then `t1 -> t2 <: t3 -> t4`
  - Function subtyping contravariant in argument(s) and covariant in results

- Also essential for understanding subtyping and methods in OOP

- The most unintuitive concept in this course
  - Smart people often forget and convince themselves that covariant arguments are okay
  - These smart people are always mistaken
  - At times, you or your boss or your friend may do this
  - Remember: A guy with a PhD in PL ***jumped out and down*** insisting that function/method subtyping is always contravariant in its argument -- covariant is unsound

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman
2013

Subtyping for OOP

Now…

Use what we learned about subtyping for records and functions to understand subtyping for class-based OOP

- Like in Java/C#

Recall:

- Class names are also types
- Subclasses are also subtypes
- Substitution principle: Instance of subclass should usable in place of instance of superclass

An object is…

- Objects: mostly records holding fields and methods
  - Fields are mutable
  - Methods are immutable functions that also have access to **`self`**

- So *could* design a type system using types very much like record types
  - Subtypes could have extra fields and methods
  - Overriding methods could have contravariant arguments and covariant results compared to method overridden
    - Sound only because method "slots" are immutable!

Actual Java/C#…

Compare/contrast to what our "theory" allows:

1. Types are class names and subtyping are explicit subclasses

2. A subclass can add fields and methods

3. A subclass can override a method with a covariant return type
   – (No contravariant arguments; instead makes it a non-overriding method of the same name)

(1) Is a subset of what is sound (so also sound)

(3) Is a subset of what is sound and a different choice (adding method instead of overriding)

Classes vs. Types

- A class defines an object's behavior
  – Subclassing inherits behavior and changes it via extension and overriding

- A type describes an object's methods’ argument/result types
  – A subtype is substitutable in terms of its field/method types

- These are separate concepts: try to use the terms correctly
  – Java/C# confuse them by requiring subclasses to be subtypes
  – A class name is both a class and a type
  – This confusion is convenient in practice

Optional: More details

Java and C# are sound: They do not allow subtypes to do things that would lead to “method missing” or accessing a field at the wrong type

Confusing (?) Java example:

- Subclass can declare field name already declared by superclass
- Two classes can use any two types for the field name
- Instance of subclass have two fields with same name
- “Which field is in scope” depends on which class defined the method

Optional: `self/this` *is special*

- Recall our Racket encoding of OOP-style
  - "Objects" have a list of fields and a list of functions that take **`self`** as an explicit extra argument

- So if **`self/this`** is a function argument, is it contravariant?
  - No, it is *covariant*: a method in a subclass can use fields and methods only available in the subclass: essential for OOP

```
class A {
  int m(){ return 0; }
}
class B extends A {
  int x;
  int m(){ return x; }
}
```

  - Sound because calls always use the "whole object" for **`self`**
  - This is why coding up your own objects manually works much less well in a statically typed languages

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Generics Versus Subtyping

What are generics good for?

Some good uses for parametric polymorphism:

- Types for functions that combine other functions:

```
fun compose (g,h) = fn x => g (h x)
(* compose : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c) *)
```

- Types for functions that operate over generic collections

```
val length : 'a list -> int
val map : ('a -> 'b) -> 'a list -> 'b list
val swap : ('a * 'b) -> ('b * 'a)
```

- Many other idioms
- General point: When types can "be anything" but multiple things need to be "the same type"

Generics in Java

- Java generics a bit clumsier syntactically and semantically, but can express the same ideas
  - Without closures, often need to use (one-method) objects
  - See also earlier optional lecture on closures in Java/C
- Simple example without higher-order functions (optional):

```
class Pair<T1,T2> {
  T1 x;
  T2 y;
  Pair(T1 _x, T2 _y){ x = _x; y = _y; }
  Pair<T2,T1> swap() {
     return new Pair<T2,T1>(y,x);
  }
  …
}
```

Subtyping is not good for this

- Using subtyping for containers is much more painful for clients
  - Have to downcast items retrieved from containers
  - Downcasting has run-time cost
  - Downcasting can fail: no static check that container holds the type of data you expect
  - (Only gets more painful with higher-order functions like **`map`**)

```
class LamePair {
  Object x;
  Object y;
  LamePair(Object _x, Object _y){ x=_x; y=_y; }
  LamePair swap() { return new LamePair(y,x); }
}

// error caught only at run-time:
String s = (String)(new LamePair("hi",4).y);
```

What is subtyping good for?

Some good uses for subtype polymorphism:

- Code that “needs a Foo” but fine to have “more than a Foo”

- Geometry on points works fine for colored points

- GUI widgets specialize the basic idea of “being on the screen” and “responding to user actions”

Awkward in ML

ML does not have subtyping, so this simply does not type-check:

```
(* {x:real, y:real} -> real *)
fun distToOrigin ({x=x,y=y}) =
   Math.sqrt(x*x + y*y)

val five = distToOrigin {x=3.0,y=4.0,color="red"}
```

Cumbersome workaround: have caller pass in getter functions:

```
(* ('a -> real) * ('a -> real) * 'a -> real *)
fun distToOrigin (getx, gety, v) =
   Math.sqrt((getx v)*(getx v)
             + (gety v)*(gety v))
```

– And clients still need different getters for points, color-points

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Bounded Polymorphism

Wanting both

- Could a language have generics and subtyping?
  - Sure!

- More interestingly, want to combine them
  - "Any type **`T1`** that is a subtype of **`T2`**"
  - This is bounded polymorphism
  - Lets you do things naturally you cannot do with generics or subtyping separately

Example

Method that takes a list of points and a circle (center point, radius)

- Return new list of points in argument list that lie within circle

Basic method signature:

```
List<Point> inCircle(List<Point> pts,
                     Point center,
                     double r) { … }
```

Optional: Java implementation straightforward assuming **`Point`** has a **`distance`** method

```
List<Point> result = new ArrayList<Point>();
for(Point pt: pts)
  if(pt.distance(center) <= r)
    result.add(pt);
return result;
```

Subtyping?

```
List<Point> inCircle(List<Point> pts,
                     Point center,
                     double r) { … }
```

- Would like to use **`inCircle`** by passing a **`List<ColorPoint>`** and getting back a **`List<ColorPoint>`**

- Java rightly disallows this: While **`inCircle`** would “do nothing wrong” its type does not prevent:
  - Returning a list that has a non-color-point in it
  - Modifying **`pts`** by adding non-color-points to it

Generics?

```
List<Point> inCircle(List<Point> pts,
                     Point center,
                     double r) { … }
```

- We could change the method to be

```
List<T> inCircle(List<T> pts,
                 Point center,
                 double r) { … }
```

  - Now the type system allows passing in a **`List<Point>`** to get a **`List<Point>`** returned or a **`List<ColorPoint>`** to get a **`List<ColorPoint>`** returned
  - But we cannot implement **`inCircle`** properly because method body should have no knowledge of type **`T`**

Bounds

- What we want:

```
List<T> inCircle(List<T> pts,
                 Point center,
                 double r) where T <: Point
 { … }
```

- Caller uses it generically, but must instantiate **`T`** with a subtype of **`Point`** (including **`Point`**)
- Callee can assume **`T <: Point`** so it can do its job
- Callee must return a **`List<T>`** so output will contain only list elements from input

Optional: *Real Java*

- The actual Java syntax

```
<T extends Pt> List<T> inCircle(List<T> pts,
                                Pt center,
                                double r) {
   List<T> result = new ArrayList<T>();
   for(T pt: pts)
     if(pt.distance(center) <= r)
       result.add(pt);
   return result;
}
```

- For backward-compatibility and implementation reasons, in Java there is actually always a way to use casts to get around the static checking with generics
  - With or without bounded polymorphism