# CSE341: Programming Languages

## Lecture 24
## Subtyping

Dan Grossman
Spring 2013

## Last major topic: Subtyping

Build up key ideas from first principles
- In pseudocode because:
  - No time for another language
  - Simple to first show subtyping without objects

Then:

- 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}) =
    p.color = "purple"

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

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



## *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”



## *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



## 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



## 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 **`distMoved`** 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
- Most unintuitive concept in the course
  - Smart people often forget and convince themselves covariant arguments are okay
  - These 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