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):
\[
\{f_1=e_1, f_2=e_2, \ldots, f_n=e_n\}
\]
Evaluate \(e_1\), 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
\[
e_1.f = e_2
\]
Evaluate \(e_1\) to a record \(v_1\) and \(e_2\) to a value \(v_2\);
Change \(v_1\)'s \(f\) field (which must exist) to \(v_2\);
Return \(v_2\)

A Basic Type System

Record types: What fields a record has and type for each field
\[
\{f_1:t_1, f_2:t_2, \ldots, f_n:t_n\}
\]
Type-checking expressions:
- If \(e_1\) has type \(t_1, \ldots, e_n\) has type \(t_n\),
  then \(\{f_1=e_1, \ldots, f_n=e_n\}\) has type \(\{f_1:t_1, \ldots, f_n:t_n\}\)
- If \(e\) has a record type containing \(f: t\),
  then \(e.f\) has type \(t\)
- If \(e_1\) has a record type containing \(f: t\) and \(e_2\) has type \(t\),
  then \(e_1.f = e_2\) 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):
\[
\begin{align*}
\text{fun distToOrigin} & (p:\{x:real,y:real\}) = \\
& \text{Math.sqrt}(p.x*p.x + p.y*p.y) \\
\text{val pythag} & : \{x:real,y:real\} = \{x=3.0, y=4.0\} \\
\text{val five} & : \text{real} = \text{distToOrigin}(\text{pythag})
\end{align*}
\]
Motivating subtyping

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

\[
\text{fun distToOrigin (p:{x:real,y:real}) = Math.sqrt(p.x*p.x + p.y*p.y)}
\]

\[
\text{val c : {x:real,y:real,color:string} =}
\]

\[
\{x=3.0, y=4.0, color="green"\}
\]

\[
\text{val five : real = distToOrigin(c)}
\]

A good idea: allow extra fields

Natural idea: If an expression has type

\[
{f_1:t_1, f_2:t_2, \ldots, f_n:t_n}
\]

Then it can also have a type with some fields removed

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

\[
\text{fun distToOrigin (p:{x:real,y:real}) = \ldots}
\]

\[
\text{fun makePurple (p:{color:string}) = p.color = "purple"}
\]

\[
\text{val c :{x:real,y:real,color:string} =}
\]

\[
\{x=3.0, y=4.0, color="green"\}
\]

\[
\text{val _ = distToOrigin(c)}
\]

\[
\text{val _ = makePurple(c)}
\]

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 \( t_1 <: t_2 \) for \( t_1 \) is a subtype of \( t_2 \)

– One new typing rule that uses subtyping:

  If \( e \) has type \( t_1 \) and \( t_1 <: t_2 \), then \( e \) (also) has type \( t_2 \)

Now all we need to do is define \( t_1 <: t_2 \)

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

  – 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 \( t_1 <: t_2 \), then any value of type \( t_1 \) must be usable in every way a \( t_2 \) 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 \( t_1 <: t_2 \) and \( t_2 <: t_3 \), then \( t_1 <: t_3 \)
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

\[
\text{fun circleY (c:{center:{x:real,y:real}, r:real}) = c.center.y}
\]

\[
\text{val sphere:{center:{x:real,y:real,z:real}, r:real} =}
\]

\[
\{\text{center={x=3.0,y=4.0,z=0.0}, r=1.0}\}
\]

\[
\text{val _ = circleY(sphere)}
\]

For this to type-check, we need:

\[
\text{\{center:{x:real,y:real,z:real}, r:real\} <:}
\]

\[
\text{\{center:{x:real,y:real}, r:real\}}
\]
Do not have this subtyping – could we?

\{
\text{center:} \{x: \text{real}, y: \text{real}, z: \text{real}\}, r: \text{real}\}
\subset
\{
\text{center:} \{x: \text{real}, y: \text{real}\}, r: \text{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:
\text{If } \text{ta} \subset \text{tb}, \text{ then } \{
\text{f1:} \text{t1}, \ldots, \text{f:} \text{ta}, \ldots, \text{fn:} \text{tn}\} \subset
\{
\text{f1:} \text{t1}, \ldots, \text{f:} \text{tb}, \ldots, \text{fn:} \text{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

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

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

**val** _ = setToOrigin(sphere)

val _ = sphere.\text{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 \text{t1} \subset \text{t2}, then \text{t1[]} \subset \text{t2[]}  
- So this code type-checks, surprisingly

```java
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
  - \text{e}.\text{color} will never fail due to there being no \text{color} field
  - Array reads \text{e1}[]\text{e2} always return a (subtype of) \text{t} if \text{e1} is a \text{t[]} 
- Bad news: to get the good news
  - \text{e1}[]\text{e2} can fail even if \text{e1} has type \text{t[]} and \text{e3} has type \text{t} 
  - Array stores check the run-time class of \text{e1}'s elements and do not allow storing a supertype 
  - No type-system help to avoid such bugs / performance cost
So what happens

```java
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

```java
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

```java
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

```java
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 `t -> ta <: t -> tb`
**The other way works!**

```haskell
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**

```haskell
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})
```

**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
  - 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