Review

- OOP features: interfaces, multiple inheritance, mixins, double dispatch, subtyping
- Today:
  - Subtyping for records
  - Subtyping for functions

Rename our made-up language

- Last time I called it FLJ (Flyweight Java)
- Forget that. It’s now MF (SML + mutable fields).

MF syntax and evaluation rules

Record creation (field names and contents):

\{(f_1:=e_1, f_2:=e_2, \ldots, f_n:=e_n)\} \quad \text{Evaluate } e_1, \text{ make a record}

Record field access:

\[ e.f \] \quad \text{Evaluate } e \text{ to record } v \text{ with an } f \text{ field;}
\quad \text{(otherwise, error); get contents of } f \text{ field}

Record field update:

\[ e_1.f = e_2 \] \quad \text{Evaluate } e_1 \text{ to a record } v_1 \text{ and } e_2 \text{ to a value } v_2;
\quad \text{Change } v_1\text{'s } f \text{ field (which must exist, otherwise, error) to } v_2;
\quad \text{Return } v_2

Subtyping for MF

Key idea:

If an expression has type
\{(f_1:=t_1, f_2:=t_2, \ldots, f_n:=t_n)\}
then it can be used wherever type with fewer fields
\{(f_1:=t_1, f_2:=t_2, \ldots, f_m:=t_m)\}
is required.
The larger type is a subtype of the smaller type.

Example:
\{(x:=\text{real}, y:=\text{real}, color:=\text{string})\}
is a subtype of\{(x:=\text{real}, y:=\text{real})\}

Subtyping for MF

Subtyping enables this code type-check:

\begin{verbatim}
fun distToOrigin (p:(x:real,y:real)) = ...
fun makePurple (p:(color:string)) = ...
val c : (x:real,y:real,color:string) = (x=3.0, y=4.5, color="green")
val _ = distToOrigin(c)
val _ = makePurple(c)
\end{verbatim}
Typing rules

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

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

So we add just two things to our type system:

- Subtyping relation: 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 $t_1 <: t_2$.

Barbara Liskov

Turing Award winner (2008)

“For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing.”

b. 1939

CLU language:

- abstract data types
- checked exceptions
- iterators (yield statement)

Two “universal” subtyping rules

1. **Transitivity**: If $t_1 <: t_2$ and $t_2 <: t_3$, then $t_1 <: t_3$.
2. **Reflexivity**: Every type is a subtype of itself, i.e., $t <: t$.

One rule we’ve already discovered

- **Width subtyping**: If $t_1$ has more fields than $t_2$, but otherwise agrees on field types, then $t_1 <: t_2$.

Example:

```
{x: real, y: real, color: string} <:
{x: real, y: real}
```

- Width subtyping rule obeys substitutability: using a record with more fields than expected can never lead to missing-field error.
- Good to have, enables code reuse.
What about this code reuse?

Example: A circle has a center field holding another record

```scala
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)
```

- Subtyping rules so far let us drop fields but not change their types
- For this to type-check, we need:
  
  ```scala
  {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

Could we add a new subtyping rule?

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

- So why not add another subtyping rule...
- Depth subtyping:
  
  ```scala
  if ta <: tb, then [f1: t1, ..., f: ta, ..., fn: tn] <: [f1: t1, ..., f: tb, ..., fn: tn]
  ``

- Depth+width subtyping makes our example type-check

Stop!

- Our shiny new subtyping rule makes our example type-check
- But new rule is not worthwhile unless it obeys substitutability
  - i.e., unless it still guarantees soundness
  - i.e., unless it prevents missing-field errors
- Unfortunately, depth subtyping breaks soundness 😞

Mutability, our old enemy

```scala
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 mutable fields, depth subtyping is unsound
  - Subtyping should not be permitted to change the type of fields
- But it turns out that if fields are immutable, then depth subtyping is sound
  - Yet another benefit of outlawing mutation
- So choose two of three: mutable fields, depth subtyping, soundness

Picking on Java (and C#)

Arrays should work just like records in terms of depth subtyping
- But in Java, if `t1 <: t2`, then `[x: t1] <: [x: 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
  - 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: 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

```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];
  ml(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

Subtyping for functions

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?
- Important for understanding methods (next lecture)
  - An object type is a lot like a record type where "method fields" contain functions

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 in a record with extra fields for p, but that's old news
**Return-type subtyping**

```plaintext
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) and
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 ta -> ta <: t -> tb`
  - A function can return "more than it needs to"
  - Vocabulary: "Return types are covariant"

**Argument-type subtyping, first try**

```plaintext
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`

**Argument-type subtyping, second try**

```plaintext
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
  - Vocabulary: "Argument types are contravariant"

**Subtyping on both argument and return types**

```plaintext
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**

- Function subtyping:
  - If `t3 <: t1` and `t2 <: t4`, then `t1 -> t2 <: t3 -> t4`
  - Contravariant in argument(s) and covariant in results
- Perhaps the most unintuitive concept in this course:
  - Easy to forget and convince yourself that covariant arguments are okay… but they’re not.