Subtyping: broad definitions

Job of type system:
If a program type-checks, then the program never applies an operation to an incompatible value.

New type relation: $T <: U$
Type $T$ is a subtype of type $U$ only if all operations that are valid on values of type $U$ are also valid on values of type $T$.

New type-checking rule:
If $T$ is a subtype of $U$, then an expression of type $T$ also has type $U$.

Contains material adapted from slides by Dan Grossman

Not Subtyping: type variable instantiation

ML has no subtyping

map : ('a -> 'b) -> 'a list -> 'b list
f : int -> int
xs : int list
(map f xs) : int list ← type-check
  - type variable instantiation: 'a = int, 'b = int
  - NOT subtyping

A made-up language for subtyping data

- Can cover most core subtyping ideas by just considering records with mutable fields
- Will make up our own syntax
  - ML records, no subtyping or field-mutation
  - Racket and Smalltalk: no static type system
  - Java is verbose

Mutable Records
(half like ML, half like Java/Scala)

Record creation (field names and contents):

{f1=e1, f2=e2, ..., fn=en} Evaluate eis, 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: fields a record has, 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, and \(e_n\) 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 \(t\)

Type system is sound (safe).

Does this program type check?

Can it ever try to access a non-existent field?

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

\[
\text{val pythag : \{x: real, y: real\} = \{x=3.0, y=4.0\}}
\]

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

Why not allow extra fields?

Natural idea of related types: if expression has type

\(\{f_1 : t_1, f_2 : t_2, \ldots, f_n : t_n\}\)

Then it also can have a type with a subset of those fields.

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

\[
\text{fun makePurple (p: \{color: string\}) = \ldots}
\]

\[
\text{p.color = "purple"}
\]

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

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

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

\[
\text{val _ = makePurple(c)}
\]
Changing the type system

Solution: just 2 additions, no changes
  - Subtyping relation: \( t_1 <: t_2 \) “\( t_1 \) is a subtype of \( t_2 \)”
  - 1 new typing rule:
    If \( e : t_1 \) and \( t_1 <: t_2 \), then (also) \( e : t_2 \)

Now define \( t_1 <: t_2 \)

4 good subtyping rules

For our record types, these rules all pass the substitutivity 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: \( t <: t \)

(4) may seem unnecessary, but simplifies other rules in large languages

Depth subtyping?

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)

Does this currently type-check?
Does it ever try to use non-existent fields?
How could we change the type system to allow it?
Should we?
Adding depth subtyping

New subtyping rule:

\[
\text{If } t_a <: t_b, \text{ then } \{f_1 : t_1, \ldots, f : t_a, \ldots, f_n : t_n\} <: \{f_1 : t_1, \ldots, f : t_b, \ldots, f_n : t_n\}
\]

Does it type-check now?

```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)
```

Stop!

- Added new subtyping rule to make type system more flexible.
- But is it sound?
  Does it allow any program that accesses non-existent fields?

Mutation strikes again

```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 mutable fields, depth subtyping is unsound
  - Subtyping cannot allow changing the type of fields
- If fields are immutable, then depth subtyping is sound!
- Choose two of three: mutability, depth subtyping, soundness
- Now: subtyping arrays?
- Next time: subtyping, overriding, and arguments/results
Subtyping mistakes

In Java, if \( t_1 <: t_2 \), then \( t_1[] <: t_2[] \)

- **Covariant** array subtyping

```java
class Point { ... }
class ColorPoint extends Point { ... }

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

What???

- Why allow it? Allows things like:
  - `Object[] System.arraycopy(Object[] src) { ... }`
- Seemed especially important before generics
- What goes wrong?

  - "Fix:" dynamic checking on every non-primitive array store.

From Bill Joy (Sun Cofounder)

```
Date: Fri, 09 Oct 1998 09:41:05 -0600
From: bill joy
Subject: ...[discussion about java genericity]

actually, java array covariance was done for less noble reasons ...it made some generic "bcopy" (memory copy) and like operations much easier to write...
I proposed to take this out in 95, but it was too late (...).
i think it is unfortunate that it wasn't taken out...
it would have made adding genericity later much cleaner, and [array covariance] doesn't pay for its complexity today.
wnj
```

Hypothetical:

- Allow subclass C to change type of field in superclass in scope of C
  - To unrelated type
  - To supertype of field's original type
  - To subtype of field's original type
Which ones go wrong?

  How to approach such questions:
null -- the "billion-dollar mistake"

- Chose subtyping flexibility over safety
  - null has no fields or methods
  - Java and C# let it have any object type
  - Evaluating e in e.f or e.m(...) could always produce a value without f or m!
  - ... run-time checks and errors...
- Sometimes null is convenient
  - like everything is an ML option, but not forced to check
  - But "cannot be null" types would be nice

Now functions/methods: boring part

def getLocation() : Point = { new ColorPoint(0.0, 0.0, "red") }
def plot(p: Point) : Unit = ...
plot(new ColorPoint(1.0,2.0,"red"))
def findRedDot() : ColorPoint = ...
val p : Point = findRedDot()

Functions/methods: interesting part

When is one function type a subtype of another?

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

- For overriding:
  If a superclass has a method of type t1 -> t2, can you override it with a method of type t3 -> t4 instead?

- Let's explore! See code sheet.

Method/function subtyping

Return types are covariant.
Argument types are contravariant.

\[
\begin{align*}
\text{ColorPoint} & \rightarrow \text{Point} \\
\text{ColorPoint} & \rightarrow \text{ColorPoint} \\
\text{Point} & \rightarrow \text{ColorPoint} \\
\text{Point} & \rightarrow \text{Point}
\end{align*}
\]
How special is this?

Is this contravariant (like arguments) or covariant?

```java
class A {
    def m() : Int = { 0 }
}
class B(var x: Int) extends A {
    def m() : Int = { this.x }
}
class C extends B {
    def m() : Int = { this.x }
}
```

Conclusion

- If $t_3 <: t_1$ and $t_2 <: t_4$, then $t_1 \rightarrow t_2 <: t_3 \rightarrow t_4$
  - 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: Function/method subtyping is always contravariant in its argument -- covariant is unsound