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

1

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

2

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

4

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

Record **creation** (field names and contents):

`{f1=e1, f2=e2, …, fn=en}` Evaluate `ei`s, 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`

5

A Basic Type System

Record types: fields a record has, type for each field

```
{f1:t1, f2:t2, …, fn:tn}
```

Type-checking expressions:

- If `e1` has type `t1`, …, and `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`

6

*Type system is **sound** (safe)*

Does this program type check?

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

```
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)
```

7

*Type system is **sound** (safe).*

Does this program type check?

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

```
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)
```

8

Why not allow extra fields?

Natural idea of related types: if expression has type

```
{f1 : t1, f2 : t2, ..., fn : tn}
```

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

```
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)
```

9

Changing the type system

Solution: just 2 additions, no changes

- ***Subtyping relation***: `t1 <: t2`
  "`t1` is a subtype of `t2`"
- **1 new typing rule:**
  If `e : t1` and `t1 <: t2`,
  then (also) `e : t2`

Now define `t1 <: t2`

10

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 `t1 <: t2` and `t2 <: t3`, then `t1 <: t3`.
4. **Reflexivity:**
   Every type is a subtype of itself: `t <: t`

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

12

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?

13

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)
```

Type checks only if:

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

14

Adding depth subtyping

New subtyping rule:

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

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)
```

15

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?

16

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) *)
```

17

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

18

Subtyping mistakes

In Java, if `t1 <: t2`, then `t1[] <: t2[]`

- ***Covariant*** array subtyping

```
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;
}
```

19

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.

20

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:

22

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

24

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()
```

25

Functions/methods: interesting part

When is one function type a subtype of another?

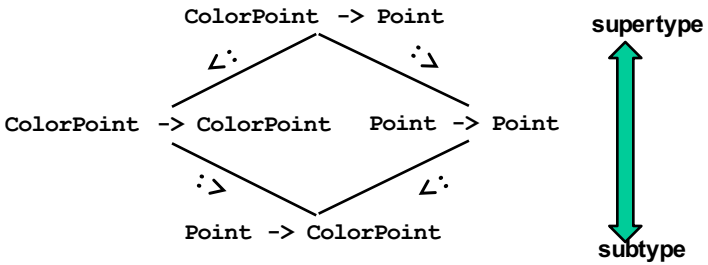
- **For higher-order functions:**
  If a function expects an argument of type `t1 -> t2`, can you pass a function of type `t3 -> t4` instead?
- **For overriding:**
  If a superclass has a method of type `t1 -> t2`, can you override it with a method of type `t3 -> t4` instead?
- Let's explore! See code sheet.

26

Method/function subtyping

Return types are **covariant.**

Argument types are **contravariant**.



32

How special is `this`?

Is `this` **contravariant** (like arguments) or **covariant?**

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

`B <: A`

`A.this <: B.this ?`

33

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: Function/method subtyping is always contravariant in its argument -- covariant is unsound**

34