Subtyping and Substitutivity

OO essence:

- Program design principles?
  - Objects model state/behavior of real-world entities/concepts? Kinda
  - Organization by classification and encapsulation
  - Reuse via implicit extensibility

- Key semantics:
  - Late binding / dynamic dispatch
  - **Substitutability and subtyping**
  - Inheritance or delegation

Will contrast function-oriented principles/semantics later.

Subtyping and substitutability

```
class Rectangle {
  private int x,y,w,h;
  void moveTo(int x, int y);
  void setSize(int width, int height);
  void show();
  void hide();
}

class FilledRectangle {
  private int x,y,w,h;
  private Color c;
  void moveTo(int x, int y);
  void setSize(int width, int height);
  void show();
  void hide();
  void setFillColor(Color color);
  Color getFillColor();
}
```

Subtyping and substitutability

```
void f() {
  Rectangle r =
    new Rectangle();
  r.moveTo(100,100);
  r.hide();
}
```

```
void f() {
  Rectangle r =
    new FilledRectangle();
  r.moveTo(100,100);
  r.hide();
}
```

Which are safe?

```
void g() {
  FilledRectangle r =
    new FilledRectangle();
  r.moveTo(100,100);
  r.setFillColor(Color.red);
  r.hide();
}
```

```
void g() {
  FilledRectangle r =
    new Rectangle();
  r.moveTo(100,100);
  r.setFillColor(Color.red);
  r.hide();
}
```

Subtyping: broad definitions

Job of type system:

If a program type-checks, **then** evaluation of the program never applies an operation to an incompatible value.

New type relation: **`T <: U`**

"Type T is a subtype of type U."

Sound **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 `e : T` and `T <: U` **then** `e : U`.

Principle: substitutability.

Type variable instantiation is NOT subtyping.

Parametric polymorphism ≠ subtype polymorphism

```
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`

ML has no subtyping

A made-up language for subtyping data

- Can cover most core subtyping ideas by considering *records with mutable fields*

- Make up our own syntax
  - ML records, no subtyping or field-mutation
  - Racket and Smalltalk: no static type system
  - Java is verbose

Mutable Records (made-up lang.)

(half like ML, half like Java)

Record creation (field names and contents):

`{f1=e1, f2=e2, …, fn=en}`

Evaluate all *`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: fields a record has, type for each field

```
{f1:t1, f2:t2, …, fn:tn}
```

Type-checking expressions:

- If `e1 : t1`, …, `en : tn`
  then `{f1=e1,…,fn=en} : {f1:t1,…,fn:tn}`
- If `e : {…,f:t,…}`
  then `e.f : t`
- If `e1 : {…,f:t,…}` and `e2 : t`,
  then `e1.f = e2 : t`

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 p : {x:real,y:real} =
    {x=3.0, y=4.0}

val five : real = distToOrigin(p)
```

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)
```

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)
```

Changing the type system

Solution: 2 additions, no changes

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

Now define `t1 <: t2`

4 reasonable subtyping rules

Principle: *substitutability*

If `t1 <: t2`, then values of type `t1` must be usable in every way values of type `t2` are.

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`
   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?

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}
```

Adding depth subtyping

New subtyping rule:

If `ta <: tb`,

then `{f1:t1, …, f:ta, …, fn:tn}`

`<: {f1:t1, …, f:tb, …, fn:tn}`

```
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 it type-check now?

Stop!

We added a 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 mutable fields.

If fields are immutable, then depth subtyping is sound!

Choose at most two of three:

- mutability
- depth subtyping
- soundness

Subtyping mistakes: Java (really)

!!!

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

"Covariant array subtyping"

```
class Point { … }
class ColorPoint extends Point { … }
…
void replaceFirst(Point[] pts) {
  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");
  replaceFirst(cpts);
  return cpts[0].color;
}
```

!!!

What???

Why allow it?

```
Object[] System.arrayCopy(Object[] src) {…}
```

Seemed especially important before generics

What goes wrong?

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

`ArrayStoreException`

!!!

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 from 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?

null – the "billion-dollar mistake"

!!!

-- C. A. R. Hoare

Chose subtyping flexibility over safety

- `null` has *no* fields or methods
- Java and C# static type systems 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… that should be static type errors.

`NullPointerException`

ML gets this right:
options make potential lack of thing explicit.

- Many languages finally moving this direction.

Function/method subtyping: boring part

```
Point getLocation() {
  return new ColorPoint(0.0, 0.0, "red");
}

void plot(Point p) {…}
…
plot(new ColorPoint(1.0,2.0,"red"));

ColorPoint findRedDot() {…}
…
Point p = findRedDot();
```

Function/method subtyping: interesting part

When is one function type a subtype of another?

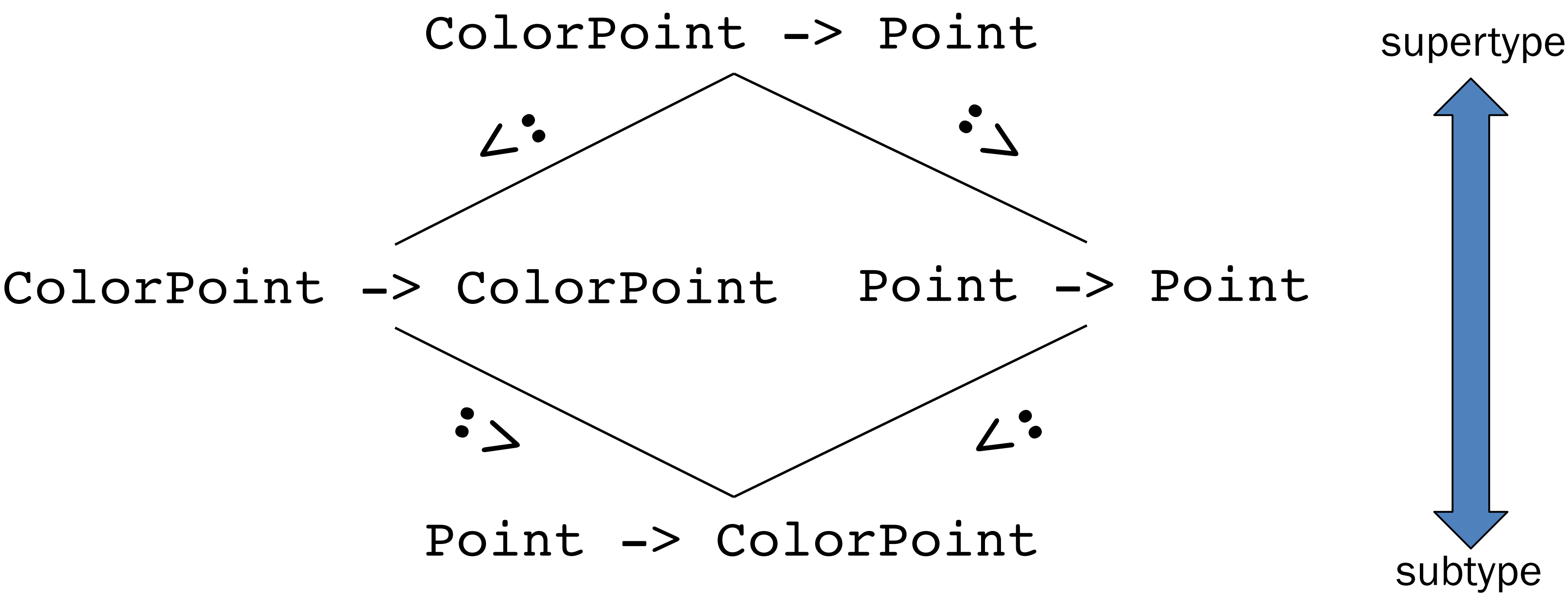
- **For higher-order functions:**
  If a function expects an argument of type `t1 -> t2`, can you pass a function of type `t3 -> t4` instead?
- **For overriding:**
  If a superclass has a method of type `t1 -> t2`, can you override it with a method of type `t3 -> t4`?
- See Subtype.java.

Function/method subtyping

Argument types are **contravariant.**

->

Return types are **covariant.**



How special is `this`?

```
class A {
  int m() { return 0; }
}
class B extends A {
  int x;
  int m() { return this.x; }
}
```

`B <: A` ✓

`A.this <: B.this` ?

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

Remember!

If t3 <: t1 and t2 <: t4,
then t1->t2 <: t3->t4

Non-negotiable:

Function/method subtyping is:

- contravariant in the argument
- covariant in the result