

# CSE 341: Programming Languages

Dan Grossman

Spring 2008

Lecture 24— Static Typing for Objects; Subtyping

# Static Typing for OO

---

Remember, any sound static type system prevents certain errors.

In ML, we never treated numbers as strings or functions, etc.

For an OO language, what's the “usual” guarantee for a type system?

- Program execution won't:
  - Send a message the receiver has no method for
  - Won't send the wrong number of arguments to a method
- Typically still allow receiver to be null (`nil`) though
  - (else too inconvenient)

Is that it?

- Pretty much; after all, all OO programs do is send messages
- With various forms of *overloading*, also prevent “no best match”

# The plan

---

- A “from first principles” approach to object-types
  - For objects with just fields, getters, and setters
  - The need for subtyping
  - “width, permutation, and depth”
    - \* Another advantage of preventing mutation
  - Objects with methods
    - \* Arguments are “contravariant”
- Next time: continue; connect this up with classes and interfaces

Warning: Lots of jargon, but ideas are important

## Types for “simple” objects

---

Assume that if an object has a field  $x$ , then it has methods  $x$  (the getter) and  $x=$  (the setter).

For an expression like  $e.x$ , type system needs to ensure  $e$  has a field  $x$ .

But what about an expression like  $e.x.y$ ?

Or  $e.x.y.z$ ?

Or  $e.x.y = e2.z$ ?

It's not enough to know  $e$  evaluates to an object with an  $x$  field. We need to know what messages the contents of that field accepts (and so on).

## Types for getter/setter objects

---

To start: a type is a list of fields and their types (recursive definition).

Let's use ML-like record syntax.

Plus named types to allow recursive structures like lists and trees.

Plus base types like `int`, `char`

- (though zero-field object can also end recursion)

Examples:

```
{ x : int, y : int }
```

```
intList = { hd : int, tl : intList }
```

```
string = { hd : char, tl : string }
```

```
name = { first : string, last : string }
```

```
{ a : {}, b : { c : int } }
```

## “Needing” nil

---

Our list types so far would need some sort of cycle.

And that makes it very hard to build/initialize the first list.

In practice, a constant `nil` *should* have type `{}` but type systems typically let it have any object type!

Revised claim on what type system soundly checks: Any message-send is understood *unless* the receiver is `nil`.

- Remember: Convenience and what-is-checked are always trade-offs.

So far: Just ML each-of types made into one-of types via `nil`.

# Wanting subtyping

---

Suppose:

- variable  $x$  has type  $\{a : \text{int}, b : \text{int}\}$
- variable  $y$  has type  $\{a : \text{int}\}$

Then:

- Should  $y=x$  be allowed? Sure, can't mess up any later use of  $y$ , for example  $y.a$ .
- Should  $x=y$  be allowed? No, could mess up later  $x.b$ .

But both would be disallowed under (just) the rule, “the two sides of an assignment must have the same type”.

We can have that rule if we have *another* rule allowing anything with type  $\{a : \text{int}, b : \text{int}\}$  to *also* have type  $\{a : \text{int}\}$ .

- Much like Java's (implicit) cast to a supertype

# Subtyping

---

Subtyping is not a matter of opinion!

*Substitutability*: If some code needs a  $t_1$ , is it always sound to give it a  $t_2$  instead? If so, we can allow  $t_2 <: t_1$ .

- “Subsumption”: If  $e$  has type  $t_2$  and  $t_2 <: t_1$ , then  $e$  has type  $t_1$
- Transitivity: If  $t_3 <: t_2$  and  $t_2 <: t_1$ , then  $t_3 <: t_1$
- Reflexivity: Every type a subtype of itself

Okay, but what are some good notions of substitutability for our getter/setter objects:

- “Width”:  $\{x_1:t_1 \dots x_n:t_n \ y:t\}$  is a subtype of  $\{x_1:t_1 \dots x_n:t_n\}$ .
- “Permutation”: order of fields in the type doesn’t matter



## What about “depth”

---

A “depth” subtyping rule says: If  $t_i <: t$ , then  $\{x_1:t_1 \dots x_i:t_i \dots x_n:t_n\}$  is a subtype of  $\{x_1:t_1 \dots x_i:t \dots x_n:t_n\}$ .

Example:  $t_1 = \{x:\{\} y:\{z:\{\}\}\}$ ,  $t_2 = \{x:\{\} y:\{\}\}$ ,  $t_1 <: t_2$ .

Sounds great: If you expect an object whose  $y$  field has no fields, it's no problem to give an object whose  $y$  field has a  $z$  field.

This is wrong because fields are mutable!!!

Example...

## Example

---

For the example, we need syntax for building objects

- not just writing down their types.

Build objects “directly”:

- `[x => [], y => [z => []]]` means an object with fields `x` and `y` where `[]` is an object with no fields, etc.

The example:

```
t1 o1 = [x=>[], y=>[z=>[]]]
t2 o2 = o1 # use _broken_ notion of subsumption
o2.y = []
o1.y.z # message not understood!
```

## Depth and Mutability

---

ML records are like our getter/setter objects without setters.

(ML doesn't have subtyping, but that's because of type inference.)

If fields are immutable, then depth subtyping is sound!

Our example — and all such examples — relies on mutation.

Yet another advantage of immutability: you get more substitutability because programs can do less.

## Methods

---

So field types must be *invariant*, else the getter or setter methods in the subtype will have an unsound type:

- The field cannot be a subtype in the subtype (see 2 slides ago).
- The field cannot be a supertype in the subtype either.

But this is really just an example of a more general phenomenon: If a supertype has a method  $m$  taking arguments of types  $t_1, \dots, t_n$  and returning an argument of type  $t_0$ , what can  $m$  take and return in a subtype?

That is, if we let object types include methods, when is

$$\{\dots t_0 \ m(t_1, \dots, t_n) \ \dots\} <: \{\dots t_0' \ m(t_1', \dots, t_n') \ \dots\}$$

# Method Subtyping, part 1

---

When is

$\{\dots t_0 \ m(t_1, \dots, t_n) \ \dots\} <: \{\dots t_0' \ m(t_1', \dots, t_n') \ \dots\}$

One sound answer: In the supertype,  $m$  must take arguments of the same type and return arguments of the same type.

- That is, for  $0 \leq i \leq n$ .  $t_i' = t_i$

(This answer corresponds to Java and C++ because they also support *static overloading*, a different topic.)

Can we be less restrictive and still sound?

Yes: We can let the return type be a subtype:  $t_0 <: t_0'$ . Why:

- Some code calling  $m$  will “know more” about what’s returned.
- Other code calling  $m$  will “still work” because of substitutability.

But what about the argument types...

## Method Subtyping, part 2

---

When is

$\{\dots t_0 \ m(t_1, \dots, t_n) \ \dots\} <: \{\dots t_0' \ m(t_1', \dots, t_n') \ \dots\}$

For  $1 \leq i \leq n$ ,  $t_i <: t_i'$  is *unsound!!!*

Example (in pseudocode):

```
{ int m({} arg) } o1 = ...
```

```
{ int m({x : int} arg) } o2 = ...
```

```
o1 = o2
```

```
o1.m([]) # calls a method that might do arg.x in its body!
```

## Method Subtyping, part 2

---

When is

$\{\dots t_0 \ m(t_1, \dots, t_n) \ \dots\} <: \{\dots t_0' \ m(t_1', \dots, t_n') \ \dots\}$

For  $1 \leq i \leq n$ ,  $t_i' <: t_i$  is *sound!!!*

Example (in pseudocode):

```
{ int m({} arg) } o1 = ...
```

```
{ int m({x : int} arg) } o2 = ...
```

```
o2 = o1
```

```
o2.m([x=7]) #fine! actual arg has x field that won't be used
```

In general, the supertype only makes it harder to call method  $m$ ; it cannot “mess up” the callee.

Jargon: Method subtyping is “contravariant” in argument types and “covariant” in return types.

## Jumping time

---

When is

$\{\dots t_0 \text{ m}(t_1, \dots, t_n) \dots\} <: \{\dots t_0' \text{ m}(t_1', \dots, t_n') \dots\}$

When  $t_0 <: t_0'$ ,  $t_1' <: t_1$ , ...  $t_n' <: t_n$ .

The point: One method is a subtype of another if the arguments are supertypes and the result is a subtype.

This is one of the most important and unintuitive points in this course.

Never, ever think argument-types are covariant. You will be tempted many times. You will never be right. Tell your friends a guy with a PhD jumped up and down!



## Connection to FP

---

Functions and methods are quite similar.

When is  $t_1 \rightarrow t_2$  a subtype of  $t_3 \rightarrow t_4$ ?

When  $t_3$  is a subtype of  $t_1$  and  $t_2$  is a subtype of  $t_4$ .

Why contravariance? For substitutability—a caller can “still” use a  $t_3$ .

Advanced point: Is there any difference? Yes, remember methods also take a `self` argument bound late.

- And in a subtype, we can assume `self` has the subtype
- But that makes it a covariant argument-type!
- This is sound because cannot change the fact that a particular value (bound to `self`) is passed
- This is why encoding late-binding in ML is awkward