CSE 341: Programming Languages

Hal Perkins

Spring 2011

Lecture 23— 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 : int, b : int}`
- variable `y` has type `{a : 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 : int, b : int}` to *also* have type `{a : int}`.

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

Subtyping

Subtyping is not a matter of opinion!

Substitutability: If some code needs a `t1`, is it always sound to give it a `t2` instead? If so, we can allow `t2<:t1`.

- "Subsumption": If `e` has type `t2` and `t2<:t1`, then `e` has type `t1`
- Transitivity: If `t3<:t2` and `t2<:t1`, then `t3<:t1`
- Reflexivity: Every type a subtype of itself

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

- "Width": `{x1:t1 ...  xn:tn y:t}` is a subtype of `{x1:t1 ...  xn:tn}`.
- "Permutation": order of fields in the type doesn't matter

What about "depth"

A "depth" subtyping rule says: If `ti<:t`, then `{x1:t1 ... xi:ti ... xn:tn}` is a subtype of `{x1:t1 ... xi:t ... xn:tn}`.

Example: `t1={x:{} y:{z:{}}}`, `t2 = {x:{} y:{}}`, `t1<:t2`.

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 $t_1, ..., t_n$ 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

```
{... t0 m(t1,...,tn) ...} <: {... t0' m(t1',...,tn') ...}
```

Method Subtyping, part 1

When is

`{... t0 m(t1,...,tn) ...} <: {... t0' m(t1',...,tn') ...}`

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

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

(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: `t0<:t0'`. 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

`{... t0 m(t1,...,tn) ...} <: {... t0’ m(t1’,...,tn’) ...}`

For $1 \leq i \leq n$, `ti<:ti’` 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

```
{... t0 m(t1,...,tn) ...} <: {... t0' m(t1',...,tn') ...}
```

For $1 \leq i \leq n$, `ti'<:ti` 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

`{... t0 m(t1,...,tn) ...} <: {... t0’ m(t1’,...,tn’) ...}`

When `t0<:t0’`, `t1’<:t1`, ... `tn’<:tn`.

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 fancy degree jumped up and down!

Connection to FP

Functions and methods are quite similar.

When is `t1->t2` a subtype of `t3->t4`?

When `t3` is a subtype of `t1` and `t2` is a subtype of `t4`.

Why contravariance? For substitutability—a caller can "still" use a `t3`.

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