CSE 341: Programming Languages

Winter 2006

Lecture 24— Static Typing for OO Languages

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 most conventional guarantee for a type system?

- Program execution will not send a not-understood message
- Except for `nil`?

Is that it?

- Pretty much; that's all Smalltalk programs do
- And it's not that easy to be sound and useful
- With multimethods or *static overloading* (coming later), we can also seek to prevent "no best match" errors

The plan

- A "from first principles" approach to object-types
  - For objects with just getters and setters
  - The need for subtyping
  - Digression to ML records
    - Immutability makes more things subtypes
  - Considering methods
    - Arguments are "contravariant"
- Next time: continue; connect this up with classes and interfaces

Warning: Lots of jargon, but ideas are very important

Types for "simple" objects

Assume for simplicity that if an object has a field `x`, then it has methods `x` (getter) and `x:` (setter).

For an expression like `e x`, our type system just 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:e2) x) y`?

The key point: 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

A type is just a list of typed fields:

```
t ::= [x1:t1, ..., xn:tn]
```

Okay, it's not quite that simple: We'll want some way to define recursive types. For example:

```
int = ... "not shown because we do not have methods"
intList = [x:int, lst:intList]
```

But wait, if we define a class `I` for making things of type `intList`, then `I new` won't have the right type. Solutions:

- Initialize an `intList` some other way. (Not good enough unless you make a cycle!)
- Let `nil` have type `intList`, though it "should" have type `[]`.

Some actual typing rules

- Let `nil` have any type.
- A class declaration should give types for instance variables; making an instance returns something with those fields and types.
- If `e` has type `[... xi:ti ...]`, then `e xi` has type `ti`.
- If `e` has type `[... xi:ti ...]` and `e2` has type `ti`, then `e xi:e2` has type `ti`.

So:

- These types are mostly "each-of" types (for the fields).
- But the first rule makes them all "one-of" types; the implementation must check for `nil` and raise an error.

So far, just like ML records, but fields are mutable.

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`.

Some more formal facts follow immediately:

- A "subsumption" rule: If `e` has type `t2` and `t2<:t1`, then `e` has type `t1`.
- Transitivity: If `t3<:t2` and `t2<:t1`, then `t3<:t1`.

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 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!!! Suppose I can build objects "directly" (where `{}` is an object with no fields.

```
t1 o1 = {x={}, y={z={}}}.
t2 o2 = o1. "use subsumption"
o2 y: {}.
(o1 y) z "error!"
```

Now you know why a Java subclass cannot change a field's type, even to a subtype (but they botched arrays)!

Depth and Mutability

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

ML doesn't have subtyping, but that's just 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:

- If the field becomes a subtype, the getter is wrong (see 2 slides ago).
- If the field becomes a supertype, the setter is wrong.

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?

Since this is more general, let's forget about fields:

```
t ::= [t10 m1:(t11,...), ..., tn0 mn(tn1,...)]
```

Now, when is `t1<:t2`?

Method Subtyping, part 1

One sound answer: A subtype can have more methods and rearrange methods, but a method m must take arguments of the same type and return arguments of the same type.

(This answer corresponds to Java and C++ because they also support *static overloading*, which we'll discuss later.)

Can we be less restrictive and still sound?

Yes: We can let the return type be a subtype. 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...

Allowing subtypes is not sound!

Method Subtyping, part 2

What if we allow argument types to be supertypes? It's sound! Why:

- Some code calling m can pass a larger collection of arguments.
- Other code calling m will "still work" because of substitutability.

The jargon: Method subtyping is "contravariant" in argument types and "covariant" in return types.

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

This is easily one of the 5 most important points in this course.

Never, ever think argument-types are covariant. You will be tempted many times. You will never be right.

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 the 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 roughly why encoding late-binding in ML is awkward.