CSE 341: Programming Languages

Dan Grossman
Spring 2004
Lecture 23— Multimethods; Static Typing for OO Languages
Multimethods

Remember our semantics for message send (with late-binding):

1. We use the receiver’s class to determine what method to call.

2. We evaluate the method body in an environment with self bound to the receiver and the arguments bound to the parameters.

The second step does not really make self so special; we could require methods to give an explicit name for this “0th” argument.

The first step does make self special; the classes of the other arguments does not affect what method we call.

Multimethods let us do just that!
Why multimethods

Consider these reasonable methods:

"in Point"

\[
distTo: \text{p2}\\
\quad ^ (((\text{self getX} - \text{p2 getX}) \text{raisedTo:2}))\\
\quad + ((\text{self getY} - \text{p2 getY}) \text{raisedTo:2})) \text{sqrt}
\]

"in 3DPoint"

\[
distTo: \text{p2}\\
\quad ^ (((\text{self getX} - \text{p2 getX}) \text{raisedTo:2}))\\
\quad + ((\text{self getY} - \text{p2 getY}) \text{raisedTo:2}))\\
\quad + ((\text{self getZ} - \text{p2 getZ}) \text{raisedTo:2}) \text{sqrt}
\]

What might happen when we do \( p \ distTo: q \)?
Multimethods Example

Neither Smalltalk nor Java has multimethods, so we have to make up syntax.

```
multimeth p1@Point distTo: p2@Point
    ^ (((p1 getX - p2 getX) raisedTo:2))
       + ((p1 getY - p2 getY) raisedTo:2)) sqrt
multimeth p1@3DPoint distTo: p2@3DPoint
    ^ (((p1 getX - p2 getX) raisedTo:2))
       + ((p1 getY - p2 getY) raisedTo:2)
       + ((p1 getZ - p2 getZ) raisedTo:2)) sqrt
```

Now we’re commutative and we can change the behavior for “one Point and one 3DPoint” by writing two more methods (and one can call the other)
Thoughts on multimethods

On the one hand, multimethods are “more OO” because they do more late-binding which is the essence of OO.

On the other hand, they are “less OO” because if the “0th” argument isn’t special, then the semantics is less “receiver-oriented” so it’s less tied to the “interacting objects” analogy.

And there are pragmatic questions like:

- where do programmers define multimethods
- how does the implementation build the necessary tables for resolving message-sends
- what if there’s “no best match”? 

Dan Grossman CSE341 Spring 2004, Lecture 23
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 \( \left( e \ x \right) \ y \)? Or \( \left( \left( e \ x \right) \ y \right) \ z \)?

Or \( \left( \left( e \ x : e2 \right) \ x \right) \ y \)?

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:

\[ t ::= [x_1 : t_1, \ldots, x_n : t_n] \]

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

\begin{align*}
\text{int} &= \ldots \text{ "not shown because we do not have methods"} \\
\text{intList} &= [x : \text{int}, \text{lst} : \text{intList}] 
\end{align*}

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

- Initialized an intList some other way. (Not good enough unless you make a cycle!)
- Let \texttt{nil} have type intList, though it “should” have type \texttt{[]}.
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 $t_1$, is it always sound to give it a $t_2$ instead? If so, we can allow $t_2 <: t_1$.

A couple more formal facts follow immediately:

- A “subsumption” rule: 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 $t_3 <: t_1$.

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

- “Width”: $[x_1 : t_1 \ldots x_n : t_n \ y : t]$ is a subtype of $[x_1 : t_1 \ldots x_n : t_n]$.
- “Permutation”: order of fields doesn’t matter
What about “depth”

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

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

$t_1\ o_1 = \{x=\{},\ y=\{z=\}\}\}$.  
$t_2\ o_2 = o_1$. "use subsumption"  
o_2\ y: {}$.  
(o_1 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 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 \textit{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 $t_1, \ldots, t_n$ 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:

$$t ::= [t_{01} \, m_{11}(t_{11}, \ldots), \ldots, t_{0n} \, m_{n1}(t_{n1}, \ldots)]$$

Now, when is $t_1 <: t_2$?
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. 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 the contravariance? For substitutability—a caller can “still” use a \( t_3 \).

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

- And in a subtype, we can assume \texttt{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 \texttt{self}) is passed.
- This is roughly why encoding late-binding in ML is awkward.