Review

- OOP features: interfaces, multiple inheritance, mixins
- Today:
  - Double dispatch and multimethods
  - an OOP feature that neither Ruby nor Java had!
  - found in Perl 6, C# 4.0, CLOS, Clojure
  - Immutable objects
  - Subtyping, part 1

Context

Remember our expression language?

<table>
<thead>
<tr>
<th>eval</th>
<th>toString</th>
<th>hasZero</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- We’ll use this as an example...
  - but this lecture is not about extensibility

DOUBLE DISPATCH

Context

Remember dynamic dispatch?

To evaluate method call e0.m(e1, ..., en),
1. Evaluate e0, e1, ..., en to objects obj0, obj1, ..., objn
2. Let C be the class of obj0
3. Use implementation of m defined in C, or if none there, look through mixins and superclasses.

...so meaning of m determined dynamically, not statically

Binary operations

- Consider our expression language over Int
- Now let’s also have String and Rational variants
  - How should Add work?
    - Could now take any pair of Int, String, Rational
    - Binary operation: takes two variants as input
    - Let’s say it should behave as string concatenation if at least one arg is a String, otherwise as math addition
  - (We won’t worry about Negate and Mult)
Implementing binary operation in SML

Add works differently for most combinations of Int, String, Rational
- Run-time error for any other kind of expression

Natural approach: pattern-match on the pair of values
- For commutative possibilities, can re-call with (v2,v1)

```sml
fun add_values (v1,v2) =
  case (v1,v2) of
  | (Int i, Int j) => Int (i+j)
  | (Int i, String s) => String (Int.toString i ^ s)
  | (Int i, Rational(j,k)) => Rational (i*k+j,k)
  | (Rational _, Int _) => add_values (v2,v1)
  | _ (* 5 more cases (3^2 total): see code *)

fun eval e =
  case e of
  | Add(e1,e2) => add_values (eval e1, eval e2)
```

Implementing binary operation in Ruby: first try

- Normal dynamic dispatch gives us separate methods for the variant of the first argument (the receiver)
- We could then abandon OOP style and use type tests for branching on the 2nd argument's variant
- 9 cases total: 3 in Int's add_values, 3 in String's add_values, 3 in Rational's add_values

```ruby
# Int
def add_values other
  if other.is_a? Int
    ...
  elsif other.is_a? Rational
    ...
  else ...
  end
end

# Add
def eval ; e1.eval.add_values e2.eval ; end
```

A more OO style

- The FP approach had 9 case-expression branches
- Our half-OOP approach had 3 methods with 3 branches
- A full-OOP would have 9 methods, with dynamic dispatch picking the right one
  - Dispatch would have to look at run-time types of both arguments, not just the receiver
    - Hence “double dispatch” or “multiple dispatch” or “multimethods”
  - Ruby (Java, C++) don’t have multimethods
  - Here’s a way (maybe awkward) to code it up...

```ruby
# Int
def add_values other
  if other.is_a? Int
    ...
  elsif other.is_a? Rational
    ...
  else ... end
end

class Add
  def eval ; e1.eval.add_values e2.eval ; end
end
```

The double-dispatch pattern

- If Int, String, and Rational all define all of addInt, addString, and addRational, that’s 9 cases
  - For example, String’s addInt is for additions of the form “i + s” where i is an int and s is a string (i.e., self is “on the right”)
- Add’s eval method calls el.eval.add_values e2.eval, which dispatches to add_values in Int, String, or Rational
  - Int’s add_values: other.addInt self
  - String’s add_values: other.addString self
  - Rational add_values: other.addRational self
  - So add_values performs “the 2nd dispatch” to the correct case!

See lec20_stage3.rb

Works in Java too

- In a statically typed language, double-dispatch works fine
  - Just need all the dispatch methods in the type

```java
abstract class Value extends Exp {  
  abstract Value addInt(Int other);  
  abstract Value addString(String other);  
  abstract Value addRational(Rational other);  
}  
class Int extends Value { ... }  
class String extends Value { ... }  
class Rational extends Value { ... }
```

IMMUTABLE OBJECTS

Is “staying OOP” here worth it?
A strange way to use objects...

- None of the code for our expression language mutates objects
  - Objects have getters, no setters
  - No methods (except initializers) ever assign to instance variables
  - Anytime object should be "changed", create a new object instead

- Idiom for programming in functional style inside OO language
  - Functional objects, aka immutable objects, aka value objects

Mutable vs. immutable objects in Java

Mutable:
```java
class Bird {
    private int x;
    private int y;
    Bird(int x, int y, ...) {
        this.x = x; ....
    }
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) {
        this.x = x; }
    void setY(int y) {
        this.y = y; }
    ....
    void move() {
        x = x + deltaX;
        y = y + deltaY;
    }
}
```

Immutable:
```java
class Bird {
    final private int x;
    final private int y;
    Bird(int x, int y, ...) {
        // assign to finals in ctor
        this.x = x; ....
    }
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) {
        this.x = x; }
    void setY(int y) {
        this.y = y; }
    ....
    void move() {
        x = x + deltaX;
        y = y + deltaY;
    }
}
```

Does this seem weird?
Java's String...

Alan Kay

- Turing Award winner (2003)
- Father of GUIs, PCs
- Co-inventor of Smalltalk (pred. of Ruby)
- Which invented "OOP" as term

"Though OOP came from many motivations, two were central. One was to find a more flexible version of assignment, and then to try to eliminate [assignment] altogether."
Subtyping

- In Java, can use a ColoredPoint anywhere a Point is required
  - Type-checking feature that enables this is subtyping
  - Different than subclassing
  - Different than generics

- How we'll study subtyping
  - In isolation: OOP languages entangle subtyping with too many other features
  - In made-up language: by necessity

Later:
- How does subtyping relate to subclasses for OOP?
- What are the relative strengths of subtyping and generics?
- How can subtyping and generics combine synergistically?

SML + Mutable Fields = MF

- Inject a really, really tiny fragment of the Java language into SML
  - objects have fields, those fields can be mutated...

- Most of subtyping can be explained by just records with mutable fields
  - ML has records, but no subtyping or field-mutation.
  - Why not other languages?
    - Ruby has no type system
    - Java uses class/interface names (and rarely fits on a slide)

...so we'll make up our own syntax (MF)

MF syntax and evaluation rules

Record creation (field names and contents):
\{(f1=e1, f2=e2, ..., fn=en)\} Evaluate e1, make a record

Record field access:
\[e.f\] Evaluate e to record v with an f field;
otherwise, error;
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.f field
(which must exist, otherwise, error) to v2;
Return v2

MF typing rules (first try)

Record types: What fields a record has and type for each field
\{(f1:t1, f2:t2, ..., fn:tn)\}

Type-checking expressions:
- If e1 has type t1, ..., en has type tn,
  then \{f1=e1, ..., fn=en\} has type \{f1:t1, ..., fn:tn\}
- If e has a record type containing f : t,
  then e.f has type t
- If e1 has a record type containing f : t and e2 has type t,
  then e1.f = e2 has type t

Example #1

These evaluation rules and typing rules prevent ever trying to access a field of a record that does not exist

Example program that type-checks in MF:

```
fun distToOrigin (p:{x:real,y:real}) = Math.sqrt(p.x*p.x + p.y*p.y)
val pythag : {x:real,y:real} = {x=3.0, y=4.0}
val five : real = distToOrigin(pythag)
```

Example #2

But according to our typing rules so far, this MF program does not type-check
- Even though it’s safe and useful

```
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)
```
Subtyping for MF

Key idea:
If an expression has type
\( \{ f_1: t_1, f_2: t_2, \ldots, f_n: t_n, f_{n+1}: t_{n+1} \} \)
then it can be used wherever type with fewer fields
\( \{ f_1: t_1, f_2: t_2, \ldots, f_n: t_n \} \)
is required.
The larger type is a subtype of the smaller type.

Example:
\( \{ x: \text{real}, y: \text{real}, \text{color}: \text{string} \} \)
is a subtype of\( \{ x: \text{real}, y: \text{real} \} \)

Subtyping vs. subset

\( t_1 = \{ x: \text{real}, y: \text{real}, \text{color}: \text{string} \} \)
is a subtype of\( t_2 = \{ x: \text{real}, y: \text{real} \} \)

- But\( t_1 \) is "bigger" than\( t_2 \). Shouldn’t "sub" mean smaller?

Subtyping for MF

Subtyping enables this code type-check:

``` scala
fun distToOrigin (p: {x:real,y:real}) = ...
fun makePurple (p: {color:string}) = ...
val c : {x:real,y:real,color:string} = (x=3.0, y=4.0, color="green")
val _ = distToOrigin(c)
val _ = makePurple(c)
```

Subtyping vs. subset

\( t_1 = \{ x: \text{real}, y: \text{real}, \text{color}: \text{string} \} \)
is a subtype of
\( t_2 = \{ x: \text{real}, y: \text{real} \} \)

- But\( t_1 \) is "bigger" than\( t_2 \). Shouldn’t "sub" mean smaller?
- Subtyping is not subset on fields
- It’s subset on all records in the universe that have at least the named fields