Breaking things down

• In functional (and procedural) programming, break programs down into **functions** that perform some operation

• In object-oriented programming, break programs down into **classes** that give behavior to some kind of data

This lecture:

– These two forms of **decomposition** are so exactly opposite that they are two ways of looking at the same “matrix”

– Which form is “better” is somewhat personal taste, but also depends on **how you expect to change/extend software**

– For some operations over two (multiple) arguments, functions and pattern-matching are straightforward, but with OOP we can do it with **double dispatch** (multiple dispatch)

The expression example

Well-known and compelling example of a common **pattern**:

– **Expressions** for a small language

– Different **variants** of expressions: ints, additions, negations, …

– Different **operations** to perform: `eval`, `toString`, `hasZero`, …

Leads to a matrix (2D-grid) of variants and operations

– Implementation will involve deciding what “should happen” for each entry in the grid regardless of the PL

<table>
<thead>
<tr>
<th></th>
<th><code>eval</code></th>
<th><code>toString</code></th>
<th><code>hasZero</code></th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Int</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Add</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Negate</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Standard approach in ML

• Define a **datatype**, with one **constructor** for each variant

– (No need to indicate datatypes if dynamically typed)

• “Fill out the grid” via **one function per column**

– Each function has one branch for each column entry

– Can combine cases (e.g., with wildcard patterns) if multiple entries in column are the same

<table>
<thead>
<tr>
<th></th>
<th><code>eval</code></th>
<th><code>toString</code></th>
<th><code>hasZero</code></th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Int</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Add</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Negate</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Standard approach in OOP

• Define a **class**, with one **abstract method** for each operation

– (No need to indicate abstract methods if dynamically typed)

• Define a **subclass** for each variant

• So “fill out the grid” via **one class per row** with one method implementation for each grid position

– Can use a method in the superclass if there is a default for multiple entries in a column

<table>
<thead>
<tr>
<th></th>
<th><code>eval</code></th>
<th><code>toString</code></th>
<th><code>hasZero</code></th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Int</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Add</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Negate</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A big course punchline

• FP and OOP often doing the same thing in **exact opposite way**

– Organize the program “by rows” or “by columns”

• Which is “most natural” may depend on what you are doing (e.g., an interpreter vs. a GUI) or personal taste

• Code layout is important, but there is no perfect way since software has many dimensions of structure

– Tools, IDEs can help with multiple “views” (e.g., rows / columns)
Extensibility

• For implementing our grid so far, SML / Racket style usually by column and Ruby / Java style usually by row
• But beyond just style, this decision affects what (unexpected?) software extensions need not change old code
• Functions [see ML code]:
  – Easy to add a new operation, e.g., noNegConstants
  – Adding a new variant, e.g., Mult requires modifying old functions, but ML type-checker gives a to-do list if original code avoided wildcard patterns

Spring 2013  CSE341: Programming Languages  7

The other way is possible

• Functions allow new operations and objects allow new variants without modifying existing code even if they didn’t plan for it
  – Natural result of the decomposition
Optional:
• Functions can support new variants somewhat awkwardly “if they plan ahead”
  – Not explained here: Can use type constructors to make datatypes extensible and have operations take function arguments to give results for the extensions
• Objects can support new operations somewhat awkwardly “if they plan ahead”
  – Not explained here: The popular Visitor Pattern uses the double-dispatch pattern to allow new operations “on the side”

Spring 2013  CSE341: Programming Languages  9

Binary operations

• Situation is more complicated if an operation is defined over multiple arguments that can have different variants
  – Can arise in original program or after extension
• Function decomposition deals with this much more simply…

Spring 2013  CSE341: Programming Languages  11

Thoughts on Extensibility

• Making software extensible is valuable and hard
  – If you know you want new operations, use FP
  – If you know you want new variants, use OOP
  – If both? Languages like Scala try; it’s a hard problem
  – Reality: The future is often hard to predict!
• Extensibility is a double-edged sword
  – Code more reusable without being changed later
  – But makes original code more difficult to reason about locally or change later (could break extensions)
  – Often language mechanisms to make code less extensible (ML modules hide datatypes; Java’s final prevents subclassing/overriding)

Spring 2013  CSE341: Programming Languages  10

Example

To show the issue:
  – Include variants String and Rational
  – (Re)define Add to work on any pair of Int, String, Rational
  • Concatenation if either argument a String, else math

Now just defining the addition operation is a different 2D grid:
**ML Approach**

Addition is different for most `Int`, `String`, `Rational` combinations
- Run-time error for non-value expressions
- For commutative possibilities, can re-call with 
  \((v2,v1)\)

```haskell
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*3 total): see the code *)

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

**Example**

To show the issue:
- Include variants `String` and `Rational`
- (Re)define `Add` to work on any pair of `Int`, `String`, `Rational`
  - Concatenation if either argument a `String`, else math

Now just defining the addition operation is a different 2D grid:

```
<table>
<thead>
<tr>
<th></th>
<th>Int</th>
<th>String</th>
<th>Rational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Worked just fine with functional decomposition -- what about OOP...

**What about OOP?**

Starts promising:
- Use OOP to call method `add_values` to one value with other value as result

```haskell
class Add
  def eval
    …
end
```

Classes `Int`, `MyString`, `MyRational` then all implement
- Each handling 3 of the 9 cases: “add self to argument”

```haskell
class Int
  def add_values v
    … # what goes here?
end
```

**First try**

- This approach is common, but is “not as OOP”
  - So do not do it on your homework

```haskell
class Int
  def add_values v
    if v.is_a? Int
      Int.new(v.i + i)
    elsif v.is_a? MyRational
      MyRational.new(v.i+v.j*i,v.j)
    else
      MyString.new(v.s + i.to_s)
end
```

- A “hybrid” style where we used dynamic dispatch on 1 argument and then switched to Racket-style type tests for other argument
  - Definitely not "full OOP"

**Another way…**

- `add_values` method in `Int` needs “what kind of thing” `v` has
  - Same problem in `MyRational` and `MyString`

- In OOP, “always” solve this by calling a method on `v` instead!

- But now we need to “tell” `v” what kind of thing” `self` is
  - We know that!
  - “Tell” `v` by calling different methods on `v`, passing `self`

- Use a “programming trick” (?) called double-dispatch...

**Double-dispatch “trick”**

- `Int`, `MyString`, and `MyRational` each define all of `addInt`, `addString`, and `addRational`
  - For example, `String`’s `addInt` is for adding concatenating an integer argument to the string in `self`
  - 9 total methods, one for each case of addition

- `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: v.addInt self`
  - `MyString’s add_values: v.addString self`
  - `MyRational’s add_values: v.addRational self`
  - So `add_values` performs “2nd dispatch” to the correct case of 9!

[Definitely see the code]
Why showing you this

- Honestly, partly to belittle full commitment to OOP
- To understand dynamic dispatch via a sophisticated idiom
- Because required for the homework
- To contrast with *multimethods* (optional)

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 add_values(Value other);
  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 { ... }
```

[See Java code]

Being Fair

Belittling OOP style for requiring the manual trick of double dispatch is somewhat unfair…

What would work better:
- Int, MyString, and MyRational each define three methods all named *add_values*
  - One *add_values* takes an Int, one a MyString, one a MyRational
  - So 9 total methods named *add_values*
  - `e1.eval.add_values e2.eval` picks the right one of the 9 at run-time using the classes of the two arguments
- Such a semantics is called *multimethods* or *multiple dispatch*

Multimethods

General idea:
- Allow multiple methods with same name
- Indicate which ones take instances of which classes
- Use dynamic dispatch on arguments in addition to receiver to pick which method is called

If dynamic dispatch is essence of OOP, this is more OOP
- No need for awkward manual multiple-dispatch

Downside: Interaction with subclassing can produce situations where there is “no clear winner” for which method to call

Ruby: Why not?

Multimethods a bad fit (?) for Ruby because:
- Ruby places no restrictions on what is passed to a method
- Ruby never allows methods with the same name
  - Same name means overriding/replacing

Java/C#/C++: Why not?

- Yes, Java/C#/C++ allow multiple methods with the same name
- No, these language do not have multimethods
  - They have *static overloading*
  - Uses static types of arguments to choose the method
    - But of course run-time class of receiver [odd hybrid?]?
    - No help in our example, so still code up double-dispatch manually
- Actually, C# 4.0 has a way to get effect of multimethods
- Many other language have multimethods (e.g., Clojure)
  - They are not a new idea