CS 3110

OOP vs FP:
The Expression Problem

Prof. Clarkson

Fall 2015

Today's music: "Express Yourself"
by Charles Wright & The Watts 103^{rd} Street Rhythm Band

Review

This semester has covered:

- Functional programming
- Modular programming
- Imperative and concurrent programming
- Reasoning about programs

Today:

- reflect on functional programming vs. object-oriented programming


FP!
OOP!

Expression Problem

- How do you express yourself in a functional language vs. an OO language?

- More specifically:
  - Suppose you're building a library of components
    - GUI library with widgets
    - Collections library with data structures
    - Library of compilers for many closely related programming languages
    - etc.
  - Problem: How do you express the *data* and the *operations*?
  - Problem: How do you evolve the library to add new data and new operations?

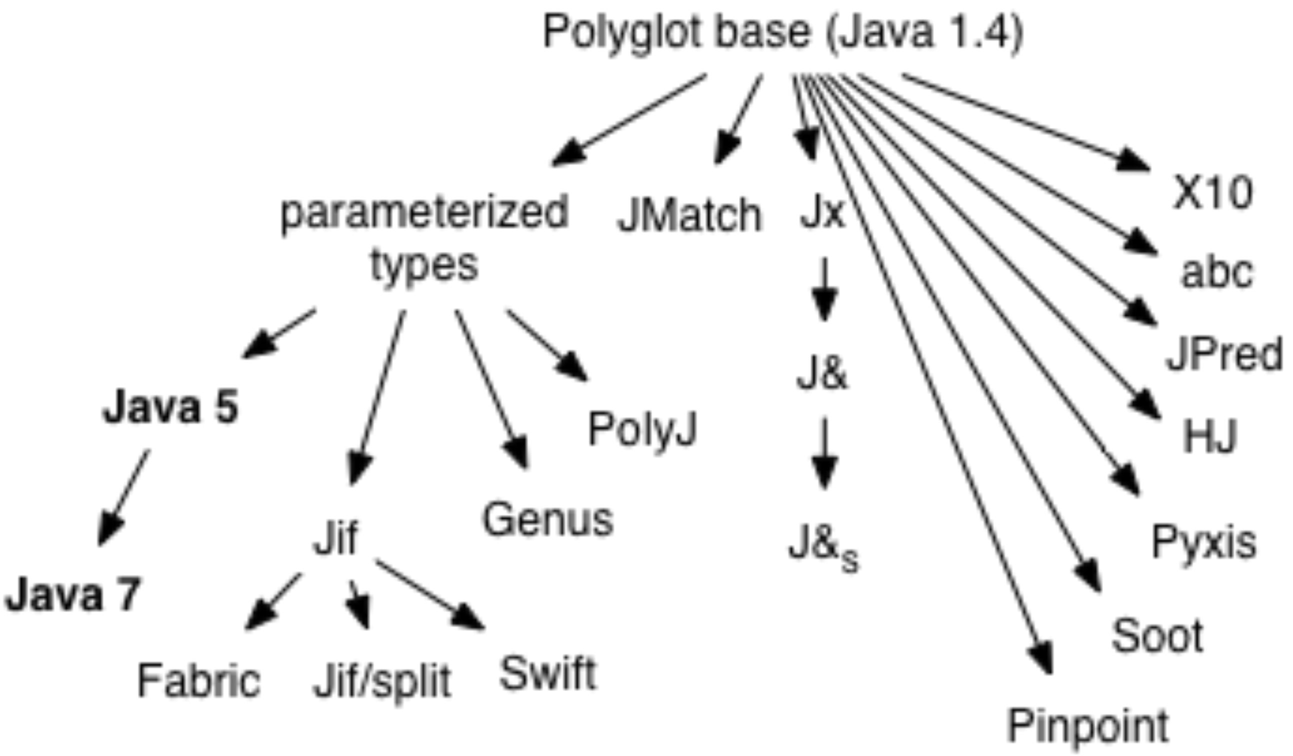
Polyglot Compiler

Suppose you want to experiment with programming language features:

- Start with a base language B
- Add a feature F1 in a modular way
- Add another feature F2 in a modular way
- Whenever B's implementation is improved, F1 and F2 should automatically get that improvement
- Later, combine F1 and F2 to get an even bigger language, just by "plugging them together", not by rewriting any code

[Polyglot: An Extensible Compiler Framework for Java. Nystrom, Clarkson, and Myers, 2003.]

Polyglot Compiler



Expression Problem

[Wadler 1998]:

- Start with an arithmetic *expression language*
- Add new forms of expressions
- Add new operations on expressions

The expression problem is: how well does your PL support this task?

Expression language

```
e ::= n | - e | e1 + e2 | ...
```

Operations:

- evaluate to integer value
- convert to string (e.g., for printing)
- determine whether zero occurs in expression
- ...

How will you design code to implement language?

Question

Which language would you choose to implement an interpreter for this simple expression language?

A. OCaml

B. Java

C. Python

D. MIPS

E. None of the above

Expression language

```
e ::= n | - e | e1 + e2 | ...
```

Operations:

- evaluate to integer value
- convert to string (e.g., for printing)
- determine whether zero occurs in expression
- ...

How will you design code to implement language?
The answer depends on your perspective on The Matrix.

The Matrix

- Rows are forms of expressions: ints, additions, negations, ...
- Columns are operations to perform: eval, toString, hasZero, …

	eval	toString	hasZero	…
Int				
Add				
Negate				
…				

Implementation will involve deciding "what should happen" for each entry in the matrix *regardless of the PL* stage1.ml

Expression Language in OCaml

```
type expr =
    | Int    of int
    | Negate of expr
    | Add    of expr * expr

let rec eval = function
    | Int i        -> i
    | Negate e     -> -(eval e)
    | Add(e1,e2)   -> (eval e1) + (eval e2)
```

Expression in FP

	eval	toString	hasZero	…
Int				
Add				
Negate				
…				

- In FP, decompose programs into functions that define operations
- Define a *variant type*, with one *constructor* for each expression form
- Fill out the matrix with one *function* per column
  - Function will pattern match on the forms
  - Can use a wildcard pattern if there is a default for multiple forms *(but maybe you shouldn't…)*

stage1.java

Expression Language in Java

```
interface Expr {
    int eval();
    String toString();
    boolean hasZero();
}
```

```
class Int implements Expr {
    private int i;
    public Int(int i) {
        this.i = i;
    }
    public int eval() {
        return i;
    }
    public String toString() {
        return Integer.toString(i);
    }
    public boolean hasZero() {
        return i==0;
    }
}
```

Expression in OOP

	eval	toString	hasZero	…
Int				
Add				
Negate				
…				

- In OOP, decompose programs into classes that define forms
- Define an *abstract class*, with an *abstract method* for each operation
  - In Java, an *interface* works for this
- Fill out the matrix with one *subclass* per row
  - Subclass will have method for each operation
  - Can use inheritance if there is a default for multiple forms
    (but maybe you shouldn't...)

FP vs. OOP

	eval	toString	hasZero	…
Int				
Add				
Negate				
…				

FP vs. OOP: first define a type, then...

- FP: express design by column
- OOP: express design by row

FP vs. OOP

- These two ways of *decomposition* are so exactly opposite that they are two ways of looking at the same matrix
- Which way is better is somewhat subjective, but also depends on how you expect to *change/extend software*

Extension

	eval	toString	hasZero	removeNegConstants
Int				
Add				
Negate				
Mult				

Suppose we need to add new:

- operations (**`removeNegConstants`**)
  - transform all syntactic occurrences of –n to Negate n
- forms (**`Mult`**)

Extension in OCaml

```
type expr =
   | Int    of int
   | Negate of expr
   | Add    of expr * expr
   | Mult   of expr * expr

let rec eval = function
   | Int i -> i
   | Negate e -> -(eval e)
   | Add(e1,e2) -> (eval e1) + (eval e2)
   | Mult(e1,e2) -> (eval e1) * (eval e2)

let rec remove_neg_constants = function
   | Int i when i<0 -> Negate (Int (-i))
   | Int _ as e -> e
   | Negate e1 -> Negate(remove_neg_constants e1)
   | Add(e1,e2) -> Add(remove_neg_constants e1, remove_neg_constants e2)
   | Mult(e1,e2) -> Mult(remove_neg_constants e1, remove_neg_constants e2)
```

Extension in FP

	eval	toString	hasZero	noNegConstants
Int				
Add				
Negate				
Mult				

- Easy to add a new operation
  - Just write a new function
  - Don’t have to modify existing functions
- Hard to add a new form
  - Have to edit all existing functions
  - But type-checker gives a todo list *if you avoid wildcard patterns*

Extension in Java

```
interface Expr {
    int eval();
    String toString();
    boolean hasZero();
    Expr removeNegConstants();
}

class Int implements Exrp {
      ...
      public Expr removeNegConstants() {
            if (i < 0) {
                return new Negate(new Int(-i));
            } else {
                return this;
            }
      }
}
```

```
class Mult implements Expr {
    private Expr e1;
    private Expr e2;
    public Mult(Expr e1, Expr e2) {
           this.e1 = e1;
           this.e2 = e2;
    }
    public int eval() {
           return e1.eval() * e2.eval();
    }
    public String toString() {
           return "(" + e1 + " * " + e2 + ")";
    }
    public boolean hasZero() {
           return e1.hasZero()
                       || e2.hasZero();
    }
    public Expr removeNegConstants() {
           return new Mult(
                 e1.removeNegConstants(),
                 e2.removeNegConstants());
    }
}
```

Extension in OOP

	eval	toString	hasZero	noNegConstants
Int				
Add				
Negate				
Mult				

- Easy to add a new form
  - Just write a new class
  - Don't have to modify existing classes
- Hard to add a new operation
  - Have to modify all existing classes
  - But Java type-checker gives a todo list *if you avoid inheritance of methods*

Planning for extension

- OOP makes new forms easy
- So if you know you want new forms, use OOP
- OOP can support new operations if you plan ahead
  - Build-in an operation whose purpose is to handle extension (Visitor pattern)

- FP makes new operations easy
- So if you know you want new operations, use FP
- FP can support new forms if you plan ahead
  - Build-in a form whose purpose is to handle extension

...once again, FP and OOP are **exact opposites**

Planning for extension in FP

```
type 'a expr =
   | Int     of int
   | Negate of 'a expr
   | Add     of 'a expr * 'a expr
   | Ext     of 'a
let rec eval_expr f = function
   | Int i        -> i
   | Negate e     -> -(eval_expr e)
   | Add(e1,e2) -> (eval_expr e1)
                    + (eval_expr e2)
   | Ext x        -> f x
let abort _ = failwith "No extension"
let eval e = eval_expr abort e
```

Extension: multiplication

```
type mexpr = (mexpr * mexpr) expr

let rec eval_mult (e1, e2) =
  eval e1 * eval e2
and eval e =
  eval_expr eval_mult e
```

needs
`-rectypes`
compiler flag

Extension: subtraction

```
type sexpr = (sexpr * sexpr) expr

let rec eval_sub (e1, e2) =
  eval e1 - eval e2
and eval e =
  eval_expr eval_sub e
```

Extension: sub. & mult.

```
type 'a sm = S of 'a*'a | M of 'a*'a
type smexpr = (smexpr sm) expr

let rec eval_sub_mult = function
  | S (e1, e2) -> eval e1 - eval e2
  | M (e1, e2) -> eval e1 * eval e2
and eval e =
  eval_expr eval_sub_mult e
```

Extensions with code reuse

Combined extension with subtraction and multiplication had to repeat implementations of those operations:

```
let rec eval_sub_mult = function
  | S (e1, e2) -> eval e1 - eval e2
  | M (e1, e2) -> eval e1 * eval e2
```

Even though we had already implemented them separately:

```
let rec eval_mult (e1, e2) =
  eval e1 * eval e2
let rec eval_sub (e1, e2) =
  eval e1 - eval e2
```

We can't call the old implementations–they have the wrong argument types :(

But we like code reuse...

Extensions with code reuse

This very clever type makes it possible:

```
type 'a expr =
   [ `Int of int
   | `Neg of 'a
   | `Add of 'a * 'a ]
```

- The use of polymorphic variants is the key: code can be parametric in "all future extensions"
- But we give up some static type checking, as usual with polymorphic variants

Thoughts on Extensibility

- Reality: the future is hard to predict
  - Might not know what kind of extensibility you need
  - Might even need both kinds!
    - Languages like Scala try; it's a hard problem

- Extensibility is a double-edged sword
  - Pro: code more reusable
  - Con: code more difficult to reason about locally or to change later (could break extensions)
  - So some language features specifically designed to make code *less* extensible
    - e.g., Java's **`final`** prevents subclassing/overriding
    - e.g., OCaml variants vs. polymorphic variants

Summary

- The Matrix is a fundamental truth about reality (of software)
- Software extensibility is heavily influenced by programming paradigm

OOP vs. FP isn't **only** a matter of taste

Upcoming events

- [Thursday] A6 due (including Project Implementation)

This is expressive.

THIS IS 3110