Lecture 22: The Expression Problem

Prof. Clarkson
Spring 2015

Today’s music: "Express Yourself" by Charles Wright & The Watts 103rd Street Rhythm Band
Review

Course so far:
• Functional programming
• Modular programming
• Imperative programming
• Reasoning about programs
• Concurrent programming

Final couple weeks: Advanced topics
• Next couple lectures:
  functional programming vs. object-oriented programming
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
    - 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?
Expression Problem

Very specific version of problem [Wadler 1998]:

– An arithmetic *expression language*
– Add new kinds of expressions
– Add new kinds of functions on expressions
Expression language

\[ e ::= n \mid -e \mid e_1 + e_2 \mid \ldots \]

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 #1

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 **variants** of expressions: ints, additions, negations, ...
- Columns are **operations** to perform: eval, toString, hasZero, ...

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

Implementation will involve deciding "what should happen" for each entry in the matrix *regardless of the PL*.
Expression Language in OCaml

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

```ocaml
type exp =
| Int of int
| Negate of exp
| Add of exp * exp
```
Expression in FP

In FP, decompose programs into functions that perform some operation

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

Fill out the matrix with one *function* per column
  
  – Function will pattern match on the variants
  – Can use a wildcard pattern if there is a default for multiple variants (*but maybe you shouldn't...*)
Expression Language in Java

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

class Int implements Exp {
    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

- In OOP, decompose programs into **classes that give behavior to some variant**
- Define an *abstract class*, with an *abstract method* for each operation
- Fill out the matrix with **one subclass per row**
  - Subclass will have method for each operation
  - Can use a method in the superclass if there is a default for multiple variants (*but maybe you shouldn't...*)
FP vs. OOP:

– Both need you to express a type to get started, then...
– FP: express design by column
– OOP: express design by row
FP vs. OOP

• 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 subjective, but also depends on how you expect to change/extend software
**Extension**

<table>
<thead>
<tr>
<th></th>
<th>eval</th>
<th>toString</th>
<th>hasZero</th>
<th>removeNegConstants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Suppose we need to add new:

- operations (**removeNegConstants**)
- variants (**Mult**)
Extension in OCaml

```ocaml
type exp =
  | Int of int
  | Negate of exp
  | Add of exp * exp
  | Mult of exp * exp

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 removeNegConstants = function
  | Int i when i<0 -> Negate(Int (-i))
  | Int _ as e -> e
  | Negate e1 -> Negate(removeNegConstants e1)
  | Add(e1,e2) -> Add(removeNegConstants e1, removeNegConstants e2)
  | Mult(e1,e2) -> Mult(removeNegConstants e1, removeNegConstants e2)
```
### Extension in FP

<table>
<thead>
<tr>
<th></th>
<th>eval</th>
<th>toString</th>
<th>hasZero</th>
<th>noNegConstants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

— Easy to add a new operation
  - Just write a new function
  - Don’t have to modify existing functions

— Hard to add a new variant
  - Have to edit all existing functions
  - But type-checker gives a todo list *if you avoid wildcard patterns*
interface Exp {
    int eval();
    String toString();
    boolean hasZero();
    Exp removeNegConstants();
}

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

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

<table>
<thead>
<tr>
<th></th>
<th>eval</th>
<th>toString</th>
<th>hasZero</th>
<th>noNegConstants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

— Easy to add a new variant
  • 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 non-abstract methods
Planning for extension

• FP makes new operations easy
  • So if you know you want new operations, use FP
  • FP can support new variants somewhat awkwardly if you plan ahead
    – Parameterize datatype and operations on "future extensions" (not discussed here)

• OOP makes new variants easy
  • So if you know you want new variants, use OOP
  • OOP can support new operations somewhat awkwardly if you plan ahead
    – Visitor Pattern (not discussed here)

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