The expression lemma

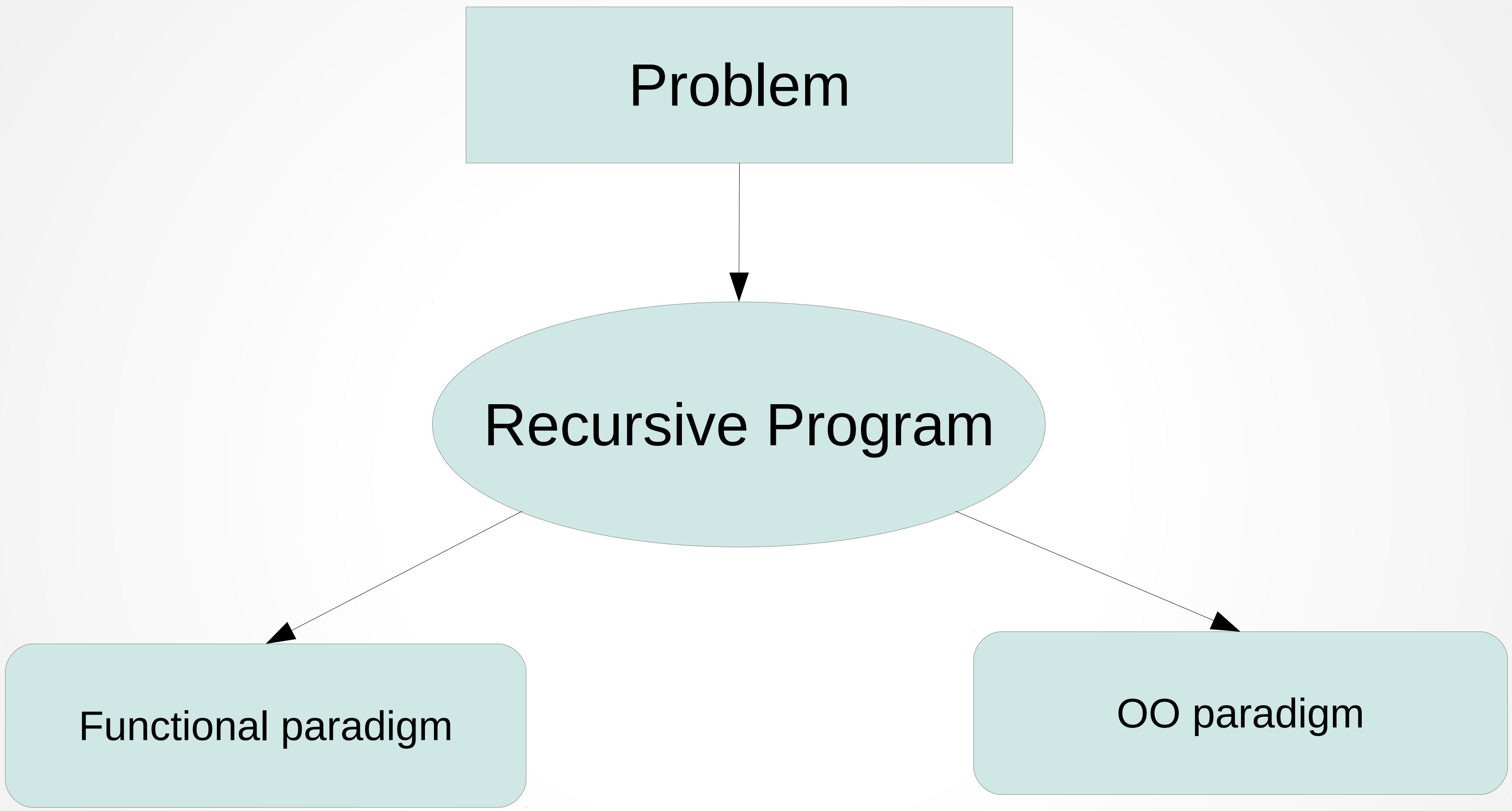
[Ralph Lämmel, Ondrej Rypacek]

João Vinagre
MAPi Thematic Seminar, June 2012

Outline

1. Introduction and motivation
2. Programming: Functional vs OO
3. Simple expression lemma
4. Generalized expression lemma
5. Related work
6. Conclusions and future work

Introduction



Functional vs OO

```
-- Arithmetic expression forms
data Expr = Num Int | Add Expr Expr

-- Evaluate expressions
eval :: Expr → Int
eval (Num i) = i
eval (Add l r) = eval l + eval r

-- Modify literals modulo v
modn :: Expr → Int → Expr
modn (Num i) v = Num (i `mod` v)
modn (Add l r) v = Add (modn l v) (modn r v)
```

Recursive functions on algebraic data type

```
public abstract class Expr {
    public abstract int eval ();
    public abstract void modn(int v);
}

public class Num extends Expr {
    private int value;
    public Num(int value) { this.value = value; }
    public int eval () { return value; }
    public void modn(int v) { this.value = this.value % v; }
}

public class Add extends Expr {
    private Expr left , right ;
    public Add(Expr left, Expr right) { this. left = left; this. right = right; }
    public int eval () { return left .eval () + right .eval (); }
    public void modn(int v) { left .modn(v); right .modn(v); }
}
```

Recursive methods on object state

We *know* (by code inspection) that these two are semantically equivalent

But can we prove it?
(Mathematically!)

Algebras and coalgebras

- Functional programs:
  - formalized in *algebras* of functional *folds (catamorphisms)*
- OO programs:
  - formalized in *coalgebras* of object *unfolds (anamorphisms)*

- *(Co)algebraic specification example (binary tree with labels):*

$$\begin{cases} \mathsf{nil}\colon 1 & \longrightarrow & X \\ \mathsf{node}\colon X \times A \times X & \longrightarrow & X \end{cases}$$

$$1 + (X \times A \times X) \to X$$

$$\begin{cases} \mathsf{leaf}\colon X & \longrightarrow & A \\ \mathsf{left}\colon X & \longrightarrow & X \\ \mathsf{right}\colon X & \longrightarrow & X \end{cases}$$

$$X \to A \times X \times X$$

Algebras and coalgebras

- Functional programs:
  - formalized in *algebras* of functional *folds (catamorphisms)*
- OO programs:
  - formalized in *coalgebras* of object *unfolds (anamorphisms)*

- *(Co)algebraic specification example (binary tree with labels):*

$$\begin{cases} \mathsf{nil}\colon 1 & \longrightarrow & X \\ \mathsf{node}\colon X \times A \times X & \longrightarrow & X \end{cases} \qquad \begin{cases} \mathsf{leaf}\colon X & \longrightarrow & A \\ \mathsf{left}\colon X & \longrightarrow & X \\ \mathsf{right}\colon X & \longrightarrow & X \end{cases}$$

$$1 + (X \times A \times X) \to X \qquad X \to A \times X \times X$$

The expression lemma

> … defines a proper correspondence between anamorphically (and coalgebraically) phrased OO programs and catamorphically phrased functional programs.

Simple expression lemma

Given a distributive law $\boldsymbol{\lambda : FB \to BF}$,
we can define an arrow $\boldsymbol{\mu F \to \nu B}$ using the derivations:

$$\lambda_{\nu B} \circ F\mathsf{out}_B : F\nu B \longrightarrow BF\nu B$$

$$[\!(\, \lambda_{\nu B} \circ F\mathsf{out}_B \,)\!]_B : F\nu B \longrightarrow \nu B$$

$$(\!|\, [\!(\, \lambda_{\nu B} \circ F\mathsf{out}_B \,)\!]_B \,|\!)_F : \mu F \longrightarrow \nu B$$

$$B\mathsf{in}_F \circ \lambda_{\mu F} : FB\mu F \longrightarrow B\mu F$$

$$(\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F : \mu F \longrightarrow B\mu F$$

$$[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,)\!]_B : \mu F \longrightarrow \nu B$$

Generalized expression lemma

- Same reasoning, but lifts to monads and comonads
- Some additional properties need to be met

Given a monad $\langle T, \eta, \mu \rangle$, a comonad $\langle D, \eta, \delta \rangle$ and a distributive law $\Lambda : TD \rightarrow DT$ of the monad T over D:

$$(\!|\, [\!(\, \Lambda_{D1} \circ T\delta_1 \,)\!]_D \,|\!)_T \quad = \quad [\!(\, (\!|\, D\mu_0 \circ \Lambda_{T0} \,|\!)_T \,)\!]_D$$

Related work

- Functional OO programming languages
  - Moby, .NET (C#/VB + LINQ), F#, Scala, OCaml, …
  - Focus is on extending existing OO languages to support the functional paradigm (or vice-versa)
  - However they do not try to establish a formal correspondence between OO and functional programs
- The expression problem [Wadler]
  - Language code extensibility problem (both term- and operation-wise)
  - Assumes the expression lemma, however with less structure

Related work (continued)

- Operational semantics [Turi and Plotkin]
  - Denotational and operational semantics corresponds to functional and OO, respectively.
- Distributive laws
  - Languages with binders in presheaf category [Fiore, Plotkin and Turi]
  - Recursive constructs [Klin]
  - Modular constructions on distributive laws [Jacobs]
  - Distributive laws for recursion and corecursion [Pardo, Uustalu, Vene et al]

Conclusions

- Given:
  - a recursive functional program expressed in catamorphisms (folds);
  - a recursive OO program expressed in anamorphisms (unfolds);
- we can prove the semantic equivalence between both

  (if they are in fact equivalent)

Limitations / Open issues

- Not everything is linearly recursive! → a more general lemma is necessary for many real world problems
- Now that we can establish semantic equivalence, we should take advantage of this (e.g. bidirectional code refactoring) → not trivial

•