Expression Problem

Shao Cheng

May 1, 2016

Overview of the talk

- Presenting the expression problem
  - Motivation
  - Common native solution in FP & OO paradigm
  - Defining the expression problem
- Some approaches to the expression problem
  - Final tagless style/object algebra
  - Polymorphic variants/row polymorphism

About the talk

- Some background knowledge on FP/OO will help
- There will be code
  - Syntax will not be covered
  - Language features used will be briefly introduced
  - Mainly in Haskell and Java
  - Other languages: OCaml, PureScript

A trivial language: AST & interpreter

```
data Expr
    = Lit Int
    | Add Expr Expr

e :: Expr
e = Add (Lit 2) (Add (Lit 3) (Lit 5))

eval :: Expr -> Int
eval (Lit lit) = lit
eval (Add x y) = eval x + eval y
```

Adding new construct/function

- ```
  data Expr
      = Lit Int
      | Add Expr Expr
      | Mul Expr Expr

  eval :: Expr -> Int
  eval (Lit lit) = lit
  eval (Add x y) = eval x + eval y
  eval (Mul x y) = eval x * eval y

  view :: Expr -> String
  view (Lit lit) = show lit
  view (Add x y) = concat [view x, "+", view y]
  view (Mul x y) = concat ["(", view x, ")*(", view y, ")"]
  ```
- The story ends here if one single module is good for you
  ```

Some encapsulation please

- ```
  module AST where
      data Expr = ..

  module Eval where
      import AST
      eval = ..

  module View where
      import AST
      view = ..
  ```
- Export semantics simplified: everything defined in the current module is exported
- Separate compilation: importing old modules without modification won't re-compile them
  ```

Houston, we have a problem

- Adding new functions to operate on the AST
  - Put them in a new module
  - Old modules still works, no re-compiling
- Adding new constructs to the AST
  - Bang! All old modules need re-compilation
  - And they don't work! Must add handlers for missing constructs one by one

Sit & relax, grab a cup of Java

- ```
  public abstract class Expr {
      public abstract int eval();
  }

  public class Lit extends Expr {
      public int lit;
      Lit(int lit) { .. }
      public int eval() { .. }
  }

  public class Add extends Expr {
      public Expr x, y;
      Add(Expr x, Expr y) { .. }
      public int eval() { .. }
  }

  Expr e = new Add(new Lit(2), new Add(new Lit(3), new Lit(5));
  ```

- We can also use `interface` in place of `abstract class` to model `Expr`; no difference in this example
  ```

Adding new construct to the AST

- ```
  public class Mul extends Expr {
      public Expr x, y;
      Mul(Expr x, Expr y) { .. }
      public int eval() { .. }
  }
  ```
- Old constructs still work and don't need re-compilation
  ```

Adding new function on the AST

- ```
  public abstract class Expr {
      public abstract int eval();
      public abstract String view();
  }

  public class .. extends Expr {
      ..
      public String view() { .. }
  }
  ```
- Bang! All sub-classes of `Expr` need to add code to implement `view()`
  ```

Wrapping up on this example

- What do we implement
  - An inductively defined AST, and functions to operate on it
- What do we want
  - Modularity: changing code in one place without breaking other places
- What do we get
  - FP: easy to add functions, hard to add constructs
  - OO: easy to add constructs, hard to add functions

The expression problem

The expression problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts). (Philip Wadler)

But you gonna add those code, one way or another

- Previous models are built on *closed-world assumption*
  - FP: constructors for a datatype is fixed
  - OO: methods for a class is fixed
  - Intuitive, but hurts extensibility
- We need to model datatypes and their functions on *open-world assumption*
  - Adding/removing constructor/function must be fully orthogonal
  - If old code is not broken, what are their new meanings?

The final encoding of AST & functions

```
class Expr expr where
    lit :: Int -> expr
    add :: expr -> expr -> expr

e :: Expr expr => expr
e = add (lit 2) (add (lit 3) (lit 5))

instance Expr Int where
    lit x = x
    add x y = x + y

e_evaled :: Int
e_evaled = e
```

Comparison with the initial encoding

- Initial vs final: concepts of category theory, won't be explained in detail here
- Initial encoding
  - AST as a datatype
  - Functions operate on AST using pattern matching
- Final encoding
  - AST as a type class
  - Functions are instances of the type class
  - Instance type denotes the *semantic domain* of AST

Adding a function

- ```
  instance Expr String where
      lit x = show x
      add x y = concat [x, "+", y]

  e_viewed :: String
  e_viewed = e
  ```
- Adding a new function is trivial, just write a new instance
- Old code is not broken, separate compilation is not compromised
  ```

Adding a construct

- ```
  class MulExpr expr where
      mul :: expr -> expr -> expr

  instance MulExpr Int where
      mul x y = x * y

  e :: (Expr expr, MulExpr expr) => expr
  e = mul (lit 2) (add (lit 3) (lit 5))

  e_evaled :: Int
  e_evaled = e
  ```
- New constructs can be added in a separate module
- The desired properties are still preserved
  ```

Preserving exhaustiveness check

- ```
  instance MulExpr String where
      mul x y = concat ["(", x, ")*(", y, ")"]

  -- comment out the instance above
  -- and see what happens!

  e_viewed :: String
  e_viewed = e
  ```
- When working with the initial encoding, we use pattern matching. The compiler checks code and issues warnings when case handling is not exhaustive
- This is also preserved in final encoding. The compiler throws a *type error* if newly added construct is not handled
  ```

Wrapping up on final tagless style

- The final tagless style is extensible in *both* dimensions
- The key to extensibility: exploiting the type class mechanism for automatically creating *sum types*
- Final tagless style has certain weaknesses, as will be introduced later
- Next, we'll introduce its cousin in the OO world: object algebra

Object algebra example in Java

```
public interface ExpAlg<T> {
    T lit(Integer x);
    T add(T x, T y);
}

public class ExpEval implements ExpAlg<Integer> {
    public Integer lit(Integer x) { return x; }
    public Integer add(Integer x, Integer y) { return x + y; }
}

ExpEval ee = new ExpEval();
Integer evaled = ee.add(ee.lit(2),ee.add(ee.lit(3),ee.lit(5)));
```

- `interface` similar to type class here, generic parameter T denotes semantic domain
- A dummy class `ExpEval` is required to organize the instance implementation. Consider it a *named instance*, as opposed to anonymous instances in Haskell

Adding a function

- ```
  public class ExpView implements ExpAlg<String> {
      public String lit(Integer x) { .. }
      public String add(String x, String y) { .. }
  }

  ExpView ev = new ExpView();
  String viewed = ev.add(ev.lit(2),ev.add(ev.lit(3),ev.lit(5)));
  ```

- Adding a new function is trivial, simply write a new class which implements `ExpAlg`
  ```

Adding a construct

```
public interface MulAlg<T> {
    T mul(T x, T y);
}

public class MulEval implements MulAlg<Integer> {
    public Integer mul(Integer x, Integer y) { return x * y; }
}

ExpEval ee = new ExpEval();
MulEval me = new MulEval();
Integer evaled = me.mul(ee.lit(2),ee.add(ee.lit(3),ee.lit(5)));
```

- `MulAlg` needn't to extend `ExpAlg`; `MulEval` needn't to inherit `ExpEval`, or implement `ExpAlg`
- The sum logic is not our concern, `ExpAlg` combines with `MulAlg` automatically
- Exhaustiveness is guaranteed, impossible to forget adding implementation to newly added construct

Wrapping up on the first approach

- Advantages
  - Extensible in both dimensions
  - Type-safety (exhaustiveness) is preserved
- Challenges
  - Implement higher-kinded ASTs (easy in Haskell, hard in Java)
  - Construct AST with parsing; put them in data structures like first-class values
  - Implement context-sensitive functions
    - Corresponds to matching nested patterns in initial encoding
    - Solution: convert back to initial encoding when needed
    - Solution: implement a datatype to explicitly encode the context
- Most suitable for compilers/staged interpreters
  - Especially with lots of passes & different (but similar) AST types

Built-in language features

- There are certain language features which are powerful constructs to attack on the expression problem
- Two of them will be introduced here
  - Polymorphic variants (as in OCaml)
  - Row polymorphism (as in PureScript)

Ordinary variants in OCaml

- ```
  type expr =
      | Lit of int
      | Add of expr * expr;;

  let e = Add(Lit(2),Add(Lit(3),Lit(5)));;

  let rec eval expr = match expr with
      | Lit lit -> lit
      | Add(x,y) -> eval(x) + eval(y);;

  type mul_expr = | Mul of expr * expr;;
  ```
- Newly added `Mul` doesn't play well with `expr`: you can't construct `Add(Lit(2),Mul(Lit(3),Lit(5)))`
  ```

Introducing polymorphic variants

- ```
  let e = `Add(`Lit(2),`Mul(`Lit(3),`Lit(5)));;

  let rec eval expr = match expr with
      | `Lit lit -> lit
      | `Add(x,y) -> eval(x) + eval(y)
      | `Mul(x,y) -> eval(x) * eval(y);;

  (* Optionally you can define expr like this *)
  type expr = [`Lit of int | `Add of expr * expr];;
  ```
- Simply prefix the constructors with a backquote character
- Constructors of *different* datatypes become reusable
  ```

Typing polymorphic variants

- Type of polymorphic variants take the form `[> `A of .. | `B of .. | .. ]`
  - In order to pattern match on it, we need to *at least* implement matching for constructors A, B, etc
- Type of functions over polymorphic variants take the form `[< `A of .. | `B of .. | .. ] -> ..`
  - Its parameter can't have constructors other than A, B, etc
- Type signatures are polymorphic and contain an implicit type variable
  - In terms of Haskell, something like `(HasA t, HasB t, .. ) => t`

Wrapping up on polymorphic variants

- Advantages
  - Very easy to use, just pattern match on them like ordinary datatypes
  - Adding new constructs is trivial
- Downsides
  - Typing information can bloat really fast and become hard to read (just try out under REPL)
  - Has certain performance penalty compared to ordinary datatypes

A different type of extensibility

- Algebraic datatypes can be defined in terms of sum types(variants) & product types(tuples, records, etc)
- So far we are dealing with extensible *sum types*
- Another challenge is implementing extensible *product types*
  - Add new fields to records, but still reuse old functions, without needing to create new type/coerce to old type

Case study in Haskell

- ```
  data Point = Point { x :: Double, y :: Double }

  f :: Point -> Double
  f p = sqrt ((x p)**2+(y p)**2)

  data Point' = Point' { x' :: Double, y' :: Double, z :: Double }

  f' :: Point' -> Double
  f' p = sqrt ((x' p)**2+(y' p)**2)
  ```
- When we extend `Point` with new fields, we must make a new datatype
  - With new type/constructor names to avoid collision
  - We can't feed a `Point'` to `f` if the same behavior is desired
- Expected behavior
  - `f` becomes less "picky" on the record type it receives: as long as it has fields `x` and `y` with desired type, any other field should be accepted
  ```

A naive OO approach

- ```
  public class Point {
      public double x, y;
      ..
  }

  public class Point3D extends Point {
      public double z;
      ..
  }
  ```
- Suffices in this case, but what if we want to *remove* fields?
  - In C++ (supports multiple inheritance): make a class for every field, and construct a record class by inheriting every field's class
  - In Java: make an interface for every field, specify getter/setter, and construct a record class by implementing every field's interface
- VERY VERBOSE, may instead increase lines of code
  ```

Extensible records in PureScript

- ```
  type Point r = { x :: Number, y :: Number | r }

  f :: forall r . Point r -> Number
  f { x:x, y:y } = sqrt (pow x 2.0 + pow y 2.0)

  f { x: 1.0, y:1.0, z:1.0 } -- this works!
  ```
- Records are *anonymous* in PureScript
- `{ x :: .., y :: .., .. | r }` denotes a *polymorphic* record type which has fields `x`, `y`, .., also with any other possible field marked with type variable `r`
- Remove type variable `r`, and ordinary closed record is back
  ```

Wrapping up on row polymorphism

- Row polymorphism solves the problem of extensible product type
- Row polymorphism does not rely on OO-style subtyping
- Has various uses in language implementation
  - Used to type polymorphic variants
  - Used to implement a fine-grained effect system in PureScript

On expression problem itself

- A general introduction: Expression problem (Wikipedia)
- What did the FP guys do about the expression problem?
  - Search for “expression problem” in ICFP & JFP publications
- What did the OO guys do about the expression problem?
  - Search for “expression problem” in ECOOP publications

On the final tagless style

- Refer to Oleg Kiselyov's blog entry for a comprehensive guide
  - Addresses lots of challenges we've mentioned before: higher-kinded ASTs, context-sensitive functions, serialization/deserialization, etc
  - Has OCaml/Haskell (Haskell '98/GHC Haskell) implementations

On object algebra

- Oliveira, Bruno C. D. S., and William R. Cook. "Extensibility for the Masses." In ECOOP 2012–Object-Oriented Programming, pp. 2-27. Springer Berlin Heidelberg, 2012.
- Oliveira, Bruno C. D. S., Tijs Van Der Storm, Alex Loh, and William R. Cook. "Feature-Oriented programming with object algebras." In ECOOP 2013–Object-Oriented Programming, pp. 27-51. Springer Berlin Heidelberg, 2013.

On various other approaches

- Swierstra, Wouter. "Data types à la carte." Journal of functional programming 18, no. 04 (2008): 423-436.
- Garrigue, Jacques. "Programming with polymorphic variants." In ML Workshop, vol. 13. 1998.
- Gaster, Benedict R. "Records, variants and qualified types." PhD diss., University of Nottingham, 1998.
- Kiselyov, Oleg, Ralf Lämmel, and Keean Schupke. "Strongly typed heterogeneous collections." In Proceedings of the 2004 ACM SIGPLAN workshop on Haskell, pp. 96-107. ACM, 2004.