Haskell's Take on the Expression Problem

Ralf Lämmel
Universität Koblenz-Landau, Software Languages Team, Koblenz, Germany

joint work with

Oleg Kiselyov
Fleet Numerical Meteorology and Oceanography Center, Monterey, CA

Elevator speech

Suppose you have some data variants (say "apples" and "oranges") and you have some operations on such data (say "drink" and "squeeze"), how would you go about the design of data and operations so that you can hopefully add new data variants and new operations later on?

Expression problem: Why the name?

- Consider such data:
  - Expressions as in programming languages:
    - Literals
    - Addition
    - …
- Consider such operations:
  - Pretty print expressions
  - Evaluate expressions
  - …
- Consider such extension scenarios:
  - Add another expression form
    - Cases for all existing operations must be added.
  - Add another operation
    - Cases for all existing expression forms must be added.

The name goes back to Phil Wadler who defined the expression problem in an email sent to mailing lists and individuals in November 1998.

Expression problem: What problem?

- In basic OO programming:
  - It is easy to add new data variants.
  - It is hard to add new operations.
- In basic functional programming:
  - It is easy to add new operations.
  - It is hard to add new data variants.
- In OO programming with visitors:
  - It is easy to add new operations.
  - It is hard to add new data variants.
- ...

Data extensibility based on simple inheritance

- class *Expr*: The base class of all expression forms
  - Virtual methods:
    - method *prettyPrint*: operation for pretty-printing
    - method *evaluate*: operation for expression evaluation
  - Subclasses:
    - class *Lit*: The expression form of "literals"
    - class *Add*: The expression form of "addition"
    - ...

Data extensibility

The class rooting all data variants

```
/**
 * The base class of all expression forms
 */
public abstract class Expr {

  /*
   * Operation for pretty printing
   */
  public abstract String prettyPrint();

  /*
   * Operation for expression evaluation
   */
  public abstract int evaluate();
}
```

Beware of code bloat!

A class for a data variant

```java
/**
 * The expression form of "literals" (i.e., constants)
 */
public class Lit extends Expr {

   private int info;
   public int getInfo() { return info; }
   public void setInfo(int info) { this.info = info; }

   public String prettyPrint() {
      return Integer.toString(getInfo());
   }

   public int evaluate() {
      return getInfo();
   }
}
```

Beware of code bloat!

That is, another data variant but with the same operations.

```
/**
 * The expression form of "addition"
 */
public class Add extends Expr {

  private Expr left, right;
  public Expr getLeft() { return left; }
  public void setLeft(Expr left) { this.left = left; }
  public Expr getRight() { return right; }
  public void setRight(Expr right) { this.right = right; }

  public String prettyPrint() { ... }

  public int evaluate() {
    return getLeft().evaluate() + getRight().evaluate();
  }
}
```

Beware of code bloat!

```java
/**
 * The expression form of "negation"
 */
public class Neg extends Expr {

  private Expr expr;
  public Expr getExpr() { return expr; }
  public void setExpr(Expr expr) { this.expr = expr; }

  public String prettyPrint() {
    return "-(" + getExpr().prettyPrint() + ")";
  }

  public int evaluate() {
    return - getExpr().evaluate();
  }
}
```

That is, yet another data variant but, again, with the same operations.

Beware of code bloat!

Operation extensibility based on public data and pattern matching

- data type *Expr*: The union of all expression forms
  - Public constructors:
    - *Lit*: The expression form of "literals"
    - *Add*: The expression form of "addition"
  - Functions defined by pattern matching on *Expr*:
    - function *prettyPrint*: operation for pretty-printing
    - function *evaluate*: operation for expression evaluation
    - ...

Operation extensibility

Algebraic data types of Haskell are closed!

```
module Data where

-- A data type of expression forms

data Exp = Lit Int | Add Exp Exp
```

2 constructor components

Algebraic data type

Constructor

Constructor component

Constructor

A new module can be designated to each new operation.

```haskell
module PrettyPrinter where

import Data

-- Operation for pretty printing

prettyPrint :: Exp -> IO ()
prettyPrint (Lit i) = putStr (show i)
prettyPrint (Add l r) = do prettyPrint l; putStr " + "; prettyPrint r
```

Argument type

Result type "side effect"

Operation defined by case discrimination

Another operation; another module.

```haskell
module Evaluator where

import Data


-- Operation for expression evaluation

evaluate :: Exp -> Int
evaluate (Lit i)   = i
evaluate (Add l r) = evaluate l + evaluate r
```

Operation extensibility based on visitors

- class *Expr*: The base class of all expression forms
  - Subclasses as before.
  - Virtual methods:
    - Accept a visitor ("apply an operation")
  - Visitors:
    - class *PrettyPrinter*: operation for pretty-printing
    - class *Evaluator*: operation for expression evaluation
    - ...

Operation extensibility

Domain operations are no longer hosted in domain classes.

```
/**
 * The base class of all expression forms
 */
public abstract class Expr {

  /*
   * Accept a visitor (i.e., apply an operation)
   */
  public abstract <R> R accept(Visitor<R> v);
}
```

Requires folklore knowledge on design patterns.

```
/**
 * A concrete visitor describe a concrete operation on expressions.
 * There is one visit method per type in the class hierarchy.
 */
public abstract class Visitor<R> {

  public abstract R visit(Lit x);
  public abstract R visit(Add x);
  public abstract R visit(Neg x);
}
```

One visitor-enabled data variant.

```java
/**
 * The expression form of "literals" (i.e., constants)
 */
public class Lit extends Expr {

   private int info;
   public int getInfo() { return info; }
   public void setInfo(int info) { this.info = info; }

   public <R> R accept(Visitor<R> v) {
      return v.visit(this);
   }
}
```

Another visitor-enabled data variant.

```java
/**
 * The expression form of "addition"
 */
public class Add extends Expr {

  private Expr left, right;
  public Expr getLeft() { return left; }
  public void setLeft(Expr left) { this.left = left; }
  public Expr getRight() { return right; }
  public void setRight(Expr right) { this.right = right; }

  public <R> R accept(Visitor<R> v) {
    return v.visit(this);
  }
}
```

Beware of code bloat!

Yet another visitor-enabled data variant.

```
/**
 * The expression form of "negation"
 */
public class Neg extends Expr {

   private Expr expr;
   public Expr getExpr() { return expr; }
   public void setExpr(Expr expr) { this.expr = expr; }

   public <R> R accept(Visitor<R> v) {
      return v.visit(this);
   }
}
```

Beware of code bloat!

An operation represented as a concrete visitor.

```
/**
 * Operation for pretty printing
 */
public class PrettyPrinter extends Visitor<String> {
  public String visit(Lit x) {
       return Integer.toString(x.getInfo());
  }
  public String visit(Add x) {
       return x.getLeft().accept(this)
            + " + "
            + x.getRight().accept(this);
  }
  public String visit(Neg x) {
     return "- (" + x.getExpr().accept(this) + ")";
  }
}
```

Beware of code bloat!

Another operation represented as a concrete visitor.

```java
/**
 * Operation for expression evaluation
 */
public class Evaluator extends Visitor<Integer> {
   public Integer visit(Lit x) {
      return x.getInfo();
   }
   public Integer visit(Add x) {
      return x.getLeft().accept(this) + x.getRight().accept(this);
   }
   public Integer visit(Neg x) {
      return - x.getExpr().accept(this);
   }
}
```

Beware of code bloat!

A final Java experiment: poor men's full extensibility

- We use **instanceof** tests and **casts** to dispatch functionality on data.
- We use a specific exception and **try-catch** blocks to extend operations.
- We encapsulate operations in objects so that extensions are self-aware.
- This is approach is **weakly typed**.
- In particular, there is no guarantee that we have covered all cases.

Domain classes are nothing but data capsules.

```
public abstract class Expr { }
```

```
public class Lit extends Expr {
   private int info;
   public int getInfo() { return info; }
   public void setInfo(int info) { this.info = info; }
}
```

```
public class Add extends Expr {
   private Expr left, right;
   public Expr getLeft() { return left; }
   public void setLeft(Expr left) { this.left = left; }
   public Expr getRight() { return right; }
   public void setRight(Expr right) { this.right = right; }
}
```

We simulate pattern matching by instanceof and cast.

```
public class EvaluatorBase {

  public int evaluate(Expr e) {
    if (e instanceof Lit) {
      Lit l = (Lit)e;
      return l.getInfo();
    }
    if (e instanceof Add) {
      Add a = (Add)e;
      return evaluate(a.getLeft()) + evaluate(a.getRight());
    }
    throw new FallThrouhException();
  }
}
```

We anticipate failure of such operations as a means to compose them.

```java
public class EvaluatorExtension
    extends EvaluatorBase {

  public int evaluate(Expr e) {
    try {
      return super.evaluate(e);
    }
    catch (FallThrouhException x) {
      if (e instanceof Neg) {
        Neg n = (Neg)e;
        return -evaluate(n.getExpr());
      }
      throw new FallThrouhException();
    }
  }
}
```

An extended operation first attempts the basic implementation.

Recursive calls are properly dispatched through “this” to the extended operation.

The notion of extensibility

- The initial system *I*:
  - Provides operations $o_1, \ldots, o_m$.
  - Handles data variants $d_1, \ldots, d_n$.
- Consider any number of extensions $e_1, \ldots, e_k$:
  - e_i could be a data extension:
    - Introduce a new data variant.
    - Provide cases for all existing operations.
  - e_i could be an operation extension:
    - Introduce a new operation.
    - Provide cases for all existing data variants.
  - Any extension can be factored into such primitive ones.
- The fully extended system e_k (… (e_1 (*I*)))

More on extensibility

- ***Code-level extensibility***:
  - Extensions can be modeled as code units.
  - Existing code units are never edited or cloned.
- ***Separate compilation***:
  - The extensions are compiled separately.
- ***Statically type-checked extensibility***:
  - The extension are statically type-checked separately.

Solutions of the expression problem

- Open classes in more dynamic languages (Smalltalk, etc.)
- Extensible predicates in Prolog (code-level extensibility only)
- Java, C# & Co.
  - Clever encodings (Torgersen, ECOOP 2004)
  - AOP-like Open classes (introductions)
  - .NET-like partial classes (code-level extensibility only)
  - Expanders (Warth et al., OOPSLA 2006)
  - JavaGI (Wehr et al., ECOOP 2007)
  - ...

Haskell supernatural type system at work: full extensibility based on type classes

Haskell 98 is sufficient at this point!

- Open (extensible) datatypes
  - Designate one type constructor per original data constructor.
  - Use a type class to describe the open union of data constructors.
- Open (extensible) functions
  - Designate one type class per operation.
  - Designate one instance of that class per data variant.

See also Lämmel and Ostermann's GPCE '06 paper: "Software extension and integration with type classes"

The initial data variants

```
data Exp = Lit Int | Add Exp Exp
```

Non-extensible

```
module DataBase where

-- Data variants for literals and addition

data Lit = Lit Int
data Add l r = Add l r

-- The open union of data variants

class Exp x
instance Exp Lit
instance Exp (Add l r)
```

Extensible

One designated data type per data variant

Type class ("set of types")

Type-class instances

The initial data variants

```
data Exp = Lit Int | Add Exp Exp
```

Non-extensible

```
module DataBase where


-- Data variants for literals and addition

data Lit = Lit Int
data Add l r = Add l r


-- The open union of data variants

class Exp x
instance Exp Lit
instance Exp (Add l r)
```

Extensible

That's not precise!

The initial data variants

```haskell
data Exp = Lit Int | Add Exp Exp
```

Non-extensible

```haskell
module DataBase where


-- Data variants for literals and addition

data Lit = Lit Int
data Add l r = Add l r


-- The open union of data variants

class Exp x
instance Exp Lit
instance (Exp l, Exp r) => Exp (Add l r)
```

Extensible

Constraints for operands of add

The initial data variants

```
data Exp = Lit Int | Add Exp Exp
```

Non-extensible

```
module DataBase where


-- Data variants for literals and addition

data Lit = Lit Int
data (Exp l, Exp r) => Add l r = Add l r


-- The open union of data variants

class Exp x
instance Exp Lit
instance (Exp l, Exp r) => Exp (Add l r)
```

Extensible

Constraints for operands of add

Fact sheet on single-parameter type classes

- Type class = nominal set of types *with common operations*
- (Type-class) instances
  - add a type to the set
  - define operations specifically for the type at hand
- Comparison to Java/C# interfaces
  - Interfaces are implemented with classes.
  - Instances may be added retroactively.

To be seen!

The initial pretty printing operation

```haskell
prettyPrint :: Exp -> IO ()
prettyPrint (Lit i) = putStr (show i)
prettyPrint (Add l r) = do prettyPrint l; putStr " + "; prettyPrint r
```

Non-extensible

```haskell
module PrettyPrinterBase where

import DataBase


-- Operation for pretty printing

class Exp x => PrettyPrint x
 where
  prettyPrint :: x -> IO ()

instance PrettyPrint Lit
 where
  prettyPrint (Lit i) = putStr (show i)

instance (PrettyPrint l, PrettyPrint r) => PrettyPrint (Add l r)
 where
  prettyPrint (Add l r) = do prettyPrint l; putStr " + "; prettyPrint r
```

Extensible

A type class for pretty-printing all expressions

The operation is a member of the type class

The initial evaluation operation

```
evaluate :: Exp -> Int
evaluate (Lit i)   = i
evaluate (Add l r) = evaluate l + evaluate r
```

Non-extensible

```
module EvaluatorBase where

import DataBase


-- Operation for expression evaluation

class Exp x => Evaluate x
 where
  evaluate :: x -> Int

instance Evaluate Lit
 where
  evaluate (Lit i) = i

instance (Evaluate l, Evaluate r) => Evaluate (Add l r)
 where
  evaluate (Add l r) = evaluate l + evaluate r
```

Extensible

A data extension

```
module DataExtension where

import DataBase
import PrettyPrinterBase
import EvaluatorBase


-- Data extension for negation

data Exp x => Neg x = Neg x

instance Exp x => Exp (Neg x)


-- Extending operation for pretty printing

instance PrettyPrint x => PrettyPrint (Neg x)
 where
  prettyPrint (Neg x) = do putStr "(- "; prettyPrint x; putStr ")"


-- Extending operation for expression evaluation

instance Evaluate x => Evaluate (Neg x)
 where
  evaluate (Neg x) = 0 - evaluate x
```

An operation extension: show expressions in prefix notation

```haskell
class Show x
 where
  show :: x -> String
```

```haskell
module OperationExtension1 where

import DataBase
import DataExtension


instance Show Lit
 where
  show (Lit i) = "Lit " ++ show i

instance (Exp x, Exp y, Show x, Show y) => Show (Add x y)
 where
  show (Add x y) = "Add (" ++ show x ++ ") (" ++ show y ++ ")"

instance (Exp x, Show x) => Show (Neg x)
 where
  show (Neg x) = "Neg (" ++ show x ++ ")"
```

Type class Show is predefined.

Another operation extension: "treealize" expression terms

```haskell
module OperationExtension2 where

import Data.Tree
import DataBase
import DataExtension


class ToTree x
 where
  toTree :: x -> Tree String

instance ToTree Lit
 where
  toTree (Lit i) = Node "Lit" []

instance (Exp x, Exp y, ToTree x, ToTree y) => ToTree (Add x y)
 where
  toTree (Add x y) = Node "Add" [toTree x, toTree y]

instance (Exp x, ToTree x) => ToTree (Neg x)
 where
  toTree (Neg x) = Node "Neg" [toTree x]
```

```
> toTree $ Add (Lit 1) (Lit 2)
Node "Add" [Node "Lit" ["1"], Node "Lit" ["2"]]
```

Noteworthy Haskell constructs and idioms covered.
(This is merely a check list: all of these constructs and idioms should have been explained through the previous slides.)

- Algebraic data types
- Pattern matching
- (Data) constructors
- Type constructors
- Constrained type constructors
- (Single-parameter) type classes
- Type-class instances
- Instance constraints
- Type-class-bounded polymorphism
- Super-class constraints
- Type-class-based open data types
- Modules

Haskell 98 has bee sufficient so far!

Summary

- The Expression Problem and software extension
- Dimensions of software extensibility
  - Data extensibility
  - Operation extensibility
- Major qualities
  - Discussed
    - Code-level extensibility
    - (Soft) static type checking
    - Separate compilation / deployment
  - Not discussed
    - Performance (“No distributed fat”)
    - Configurability