From Object Algebras to Finally Tagless Interpreters

Oleksandr Manzyuk

Overview

We are going to …

- discuss the expression problem in FP & OOP
- show the object algebras approach to solving the expression problem
- translate the object algebras approach from Java to Haskell and arrive at the finally tagless encoding

We are not going to …

- discuss the origins of the terms “object algebra” and “finally tagless”

There Will Be Code

Hutton's Razor

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


e1 = Add (Lit 1)         -- (1 + (2 + 3))
         (Add (Lit 2)
              (Lit 3))









eval :: Exp -> Int
eval (Lit n)   = n
eval (Add x y) = eval x + eval y
```

Hutton's Razor

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


e1 = Add (Lit 1)         -- (1 + (2 + 3))
         (Add (Lit 2)
              (Lit 3))









eval :: Exp -> Int
eval (Lit n)   = n
eval (Add x y) = eval x + eval y



view :: Exp -> String
view (Lit n)   = show n
view (Add x y) = "(" ++ view x ++ " + " ++ view y ++ ")"
```

Hutton's Razor

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


e1 = Add (Lit 1)         -- (1 + (2 + 3))
         (Add (Lit 2)
              (Lit 3))


e2 = Mul (Lit 4)         -- (4 * (5 + 6))
         (Add (Lit 5)
              (Lit 6))


eval :: Exp -> Int
eval (Lit n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y


view :: Exp -> String
view (Lit n)   = show n
view (Add x y) = "(" ++ view x ++ " + " ++ view y ++ ")"
view (Mul x y) = "(" ++ view x ++ " * " ++ view y ++ ")"
```

Hutton's Razor

```
interface Exp {
    int eval();

}
class Lit implements Exp {
    int n;
    int eval() { return n; }

}
class Add implements Exp {
    Exp x, y;
    int eval() { return x.eval() + y.eval(); }

}

Exp e1 = new Add(new Lit(1), new Add(new Lit(2), new Lit(3)));
```

Hutton's Razor

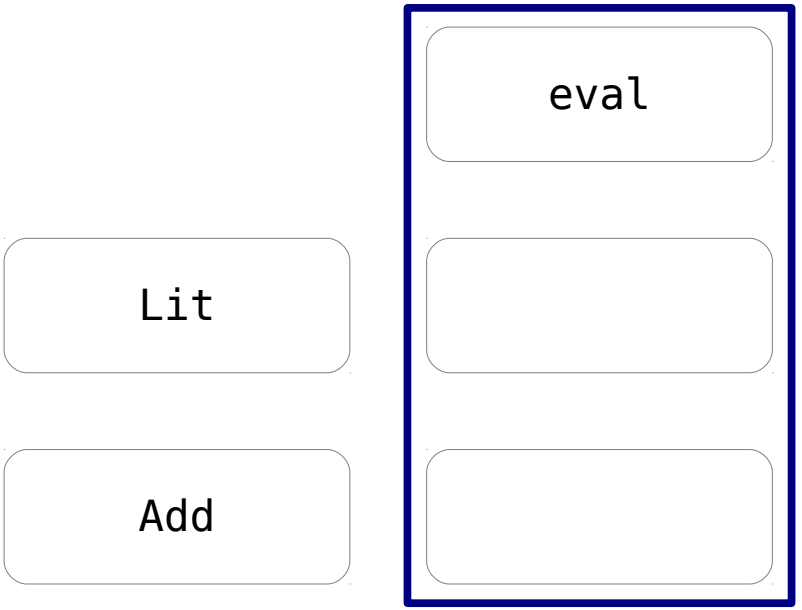

```
interface Exp {
    int eval();

}
class Lit implements Exp {
    int n;
    int eval() { return n; }

}
class Add implements Exp {
    Exp x, y;
    int eval() { return x.eval() + y.eval(); }

}

Exp e1 = new Add(new Lit(1), new Add(new Lit(2), new Lit(3)));
Exp e2 = new Mul(new Lit(4), new Add(new Lit(6), new Lit(6)));

class Mul implements Exp {
    Exp x, y;
    int eval() { return x.eval() * y.eval(); }

}
```

Hutton's Razor

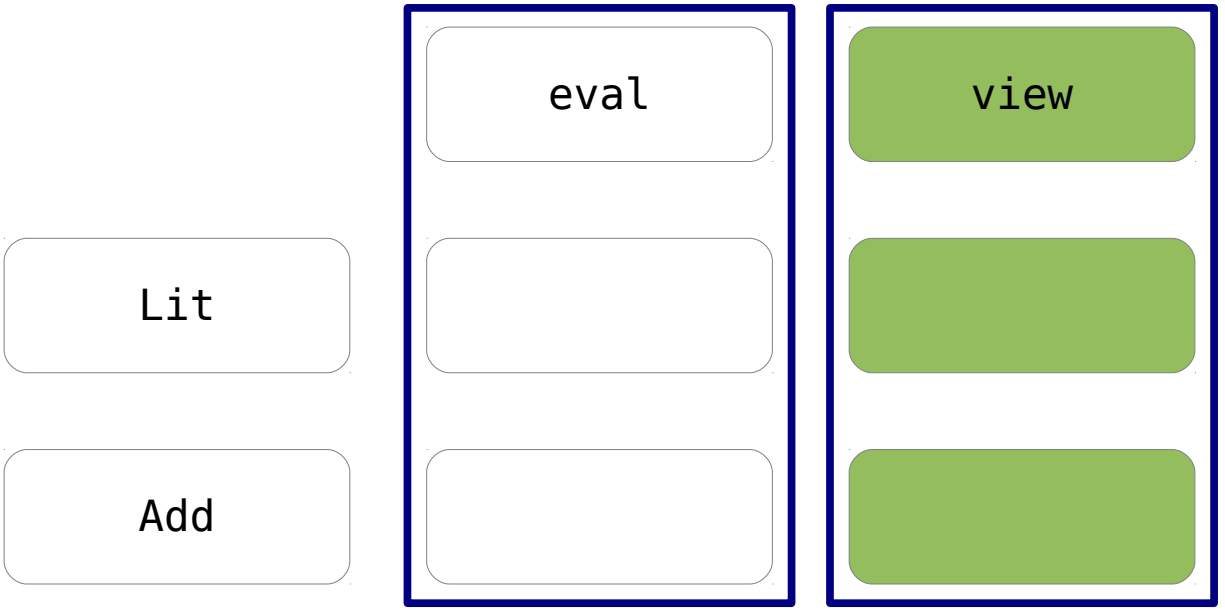

```
interface Exp {
    int eval();
    String view();
}
class Lit implements Exp {
    int n;
    int eval() { return n; }
    String view() { return Integer.toString(n); }
}
class Add implements Exp {
    Exp x, y;
    int eval() { return x.eval() + y.eval(); }
    String view() { return "(" + x.view() + " + " + y.view() + ")"; }
}

Exp e1 = new Add(new Lit(1), new Add(new Lit(2), new Lit(3)));
Exp e2 = new Mul(new Lit(4), new Add(new Lit(6), new Lit(6)));

class Mul implements Exp {
    Exp x, y;
    int eval() { return x.eval() * y.eval(); }
    String view() { return "(" + x.view() + " * " + y.view() + ")"; }
}
```

Decomposition

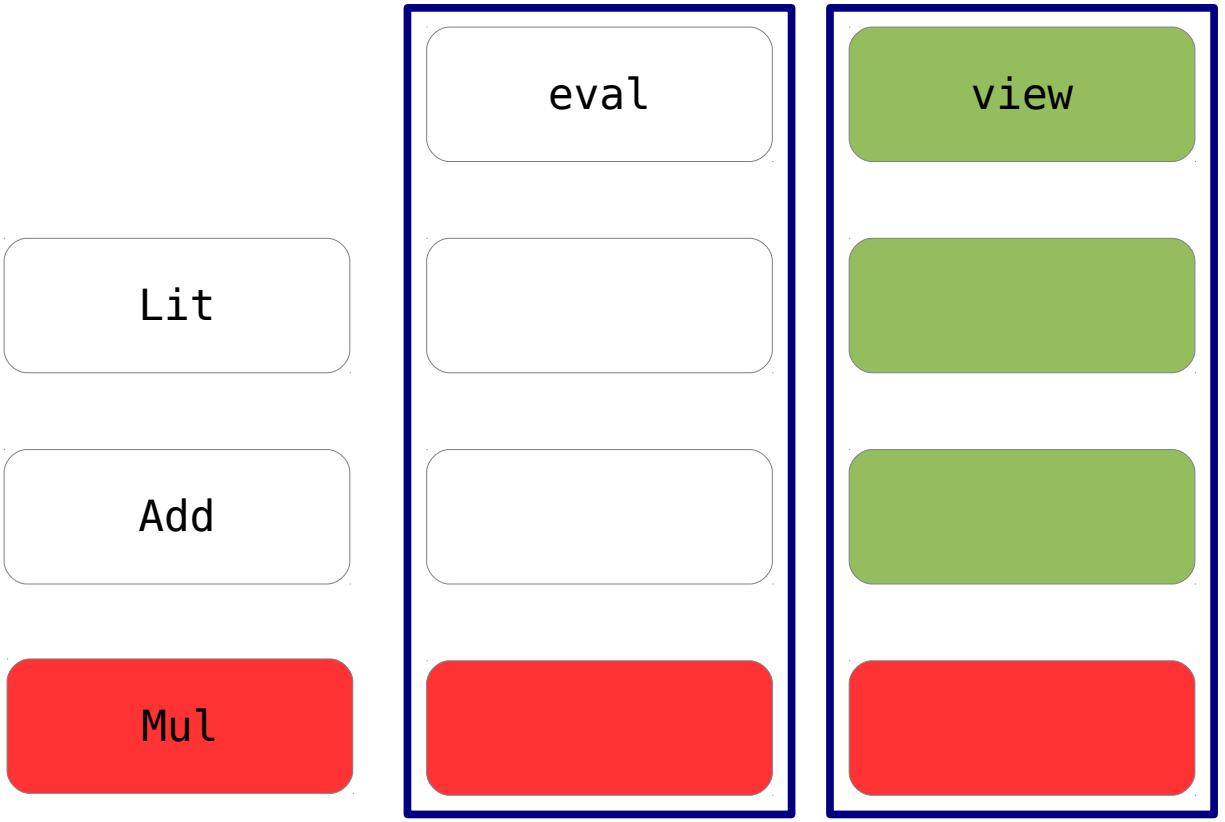
Decomposition: Functional



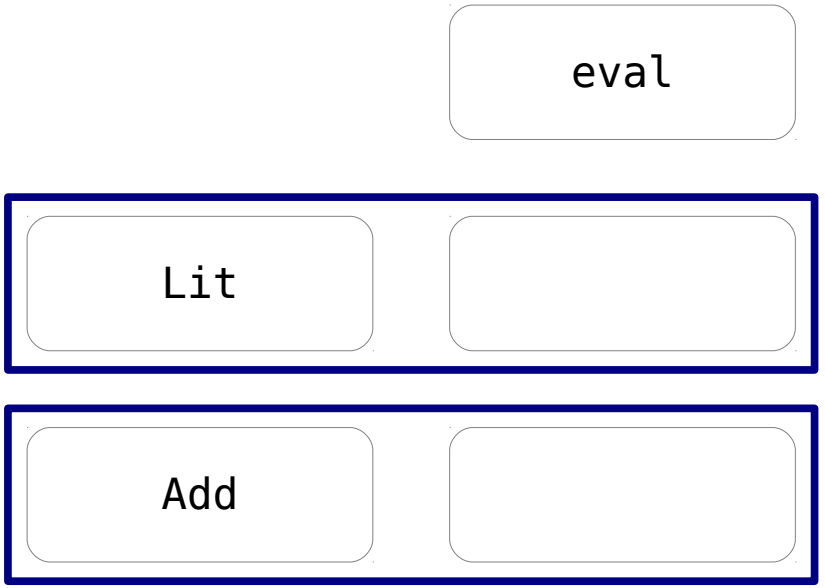
Decomposition: Functional



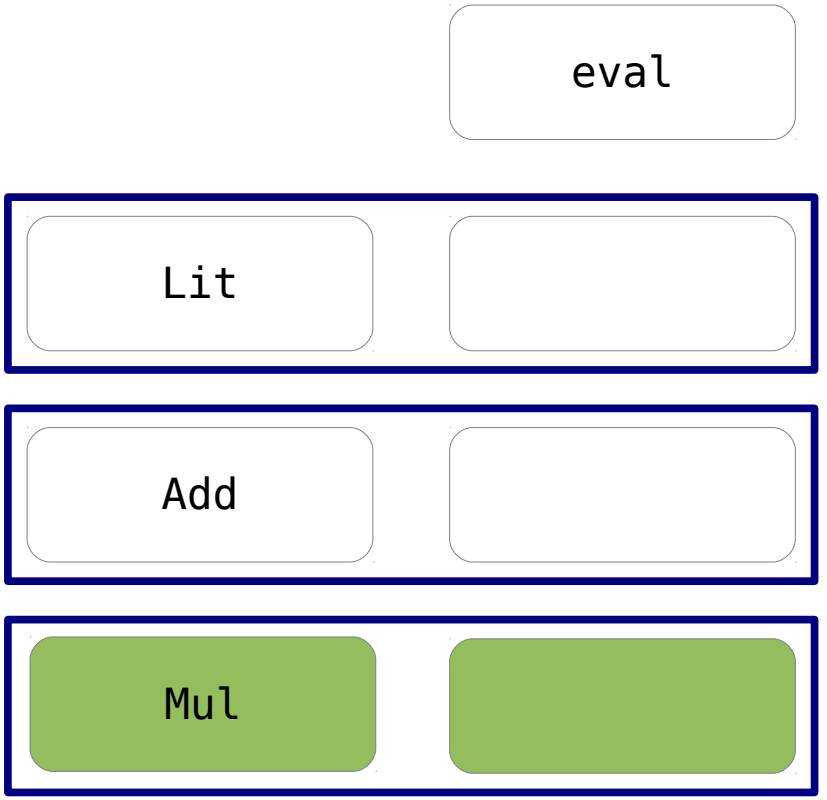
Decomposition: Functional



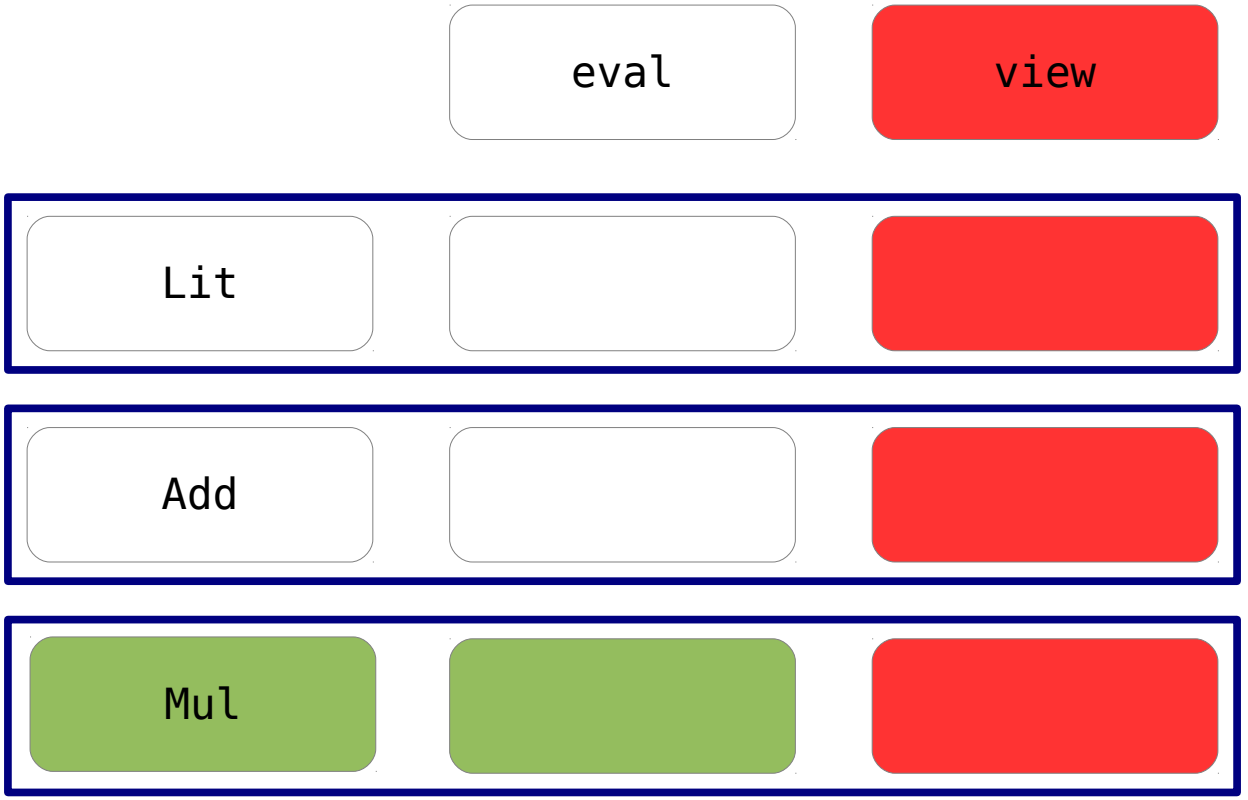
Decomposition: Object-Oriented



Decomposition: Object-Oriented



Decomposition: Object-Oriented



Expression Problem

- Extensibility in both dimensions

  Allow the addition of new data variants and new operations and support extending existing operations.

- Strong static type safety

  Prevent applying an operation to a data variant which it cannot handle using static checks.

- No modification or duplication of existing code

Object Algebras

Extensibility for the Masses

Practical Extensibility with Object Algebras

Bruno C. d. S. Oliveira[1] and William R. Cook[2]

[1]National University of Singapore
bruno@ropas.snu.ac.kr
[2] University of Texas, Austin
wcook@cs.utexas.edu

Object Algebras: Terms

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

<T> T e1(ExpAlg<T> f) {  // (1 + (2 + 3))
    return f.add(
        f.lit(1),
        f.add(
            f.lit(2),
            f.lit(3)));
}
```

Object Algebras: Operations

```
interface Eval { int eval(); }

class EvalExp implements ExpAlg<Eval> {
    Eval lit(final int n) {
        return new Eval() {
            int eval() {
                return n;
            }
        }
    }

    Eval add(final Eval x, final Eval y) {
        return new Eval() {
            int eval() {
                return x.eval() + y.eval();
            }
        }
    }
}

int v1 = e1(new EvalExp()).eval();
```

Object Algebras: Adding Variants

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

<T> T e2(MulAlg<T> f) {  // (4 * (5 + 6))
    return f.mul(
        f.lit(4),
        f.add(
            f.lit(5),
            f.lit(6)));
}
```

Object Algebras: Adding Variants

```
class EvalMul extends EvalExp implements MulAlg<Eval> {
    Eval mul(final Eval x, final Eval y) {
        return new Eval() {
            int eval() {
                return x.eval() * y.eval();
            }
        }
    }
}

int v2 = e2(new EvalMul()).eval();
```

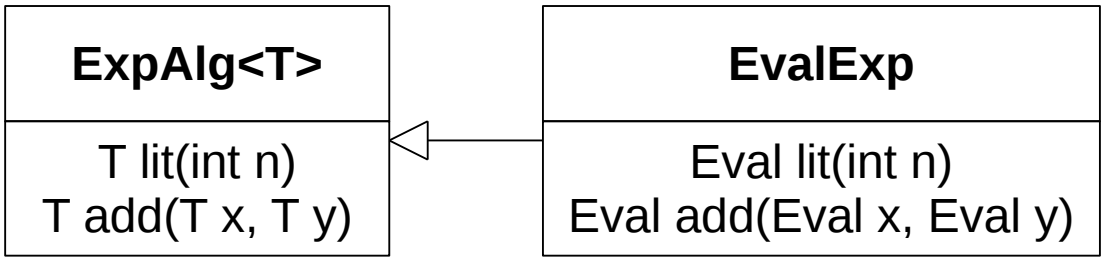
Object Algebras: Adding Operations

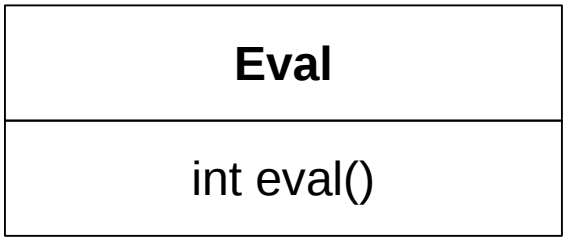
```
interface View { String view(); }

class ViewExp implements ExpAlg<Show> {
    View lit(final int n) {
        return new View() {
            String view() {
                return Integer.toString(n);
            }
        }
    }

    View add(final View x, final View y) {
        return new View() {
            String view() {
                return "(" + x.view() + " + " + y.view() + ")";
            }
        }
    }
}

String s1 = e1(new ViewExp()).view();
```

Object Algebras: Adding Operations

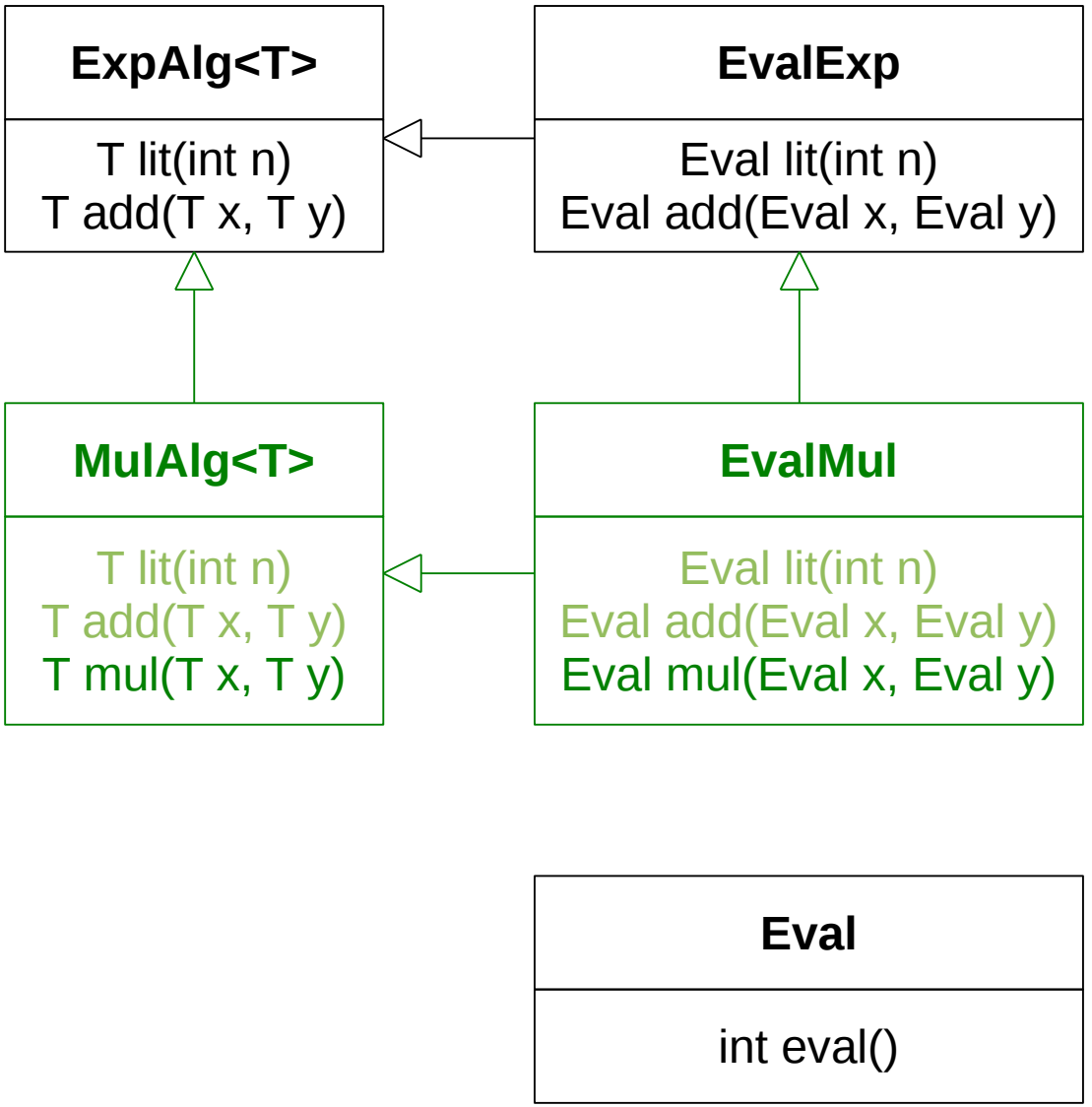
```
class ViewMul extends ViewExp implements MulAlg<View> {
    View mul(final View x, final View y) {
        return new View() {
            String view() {
                return "(" + x.view() + " * " + y.view() + ")";
            }
        }
    }
}

String s2 = e2(new ViewMul()).view();
```

Object Algebras

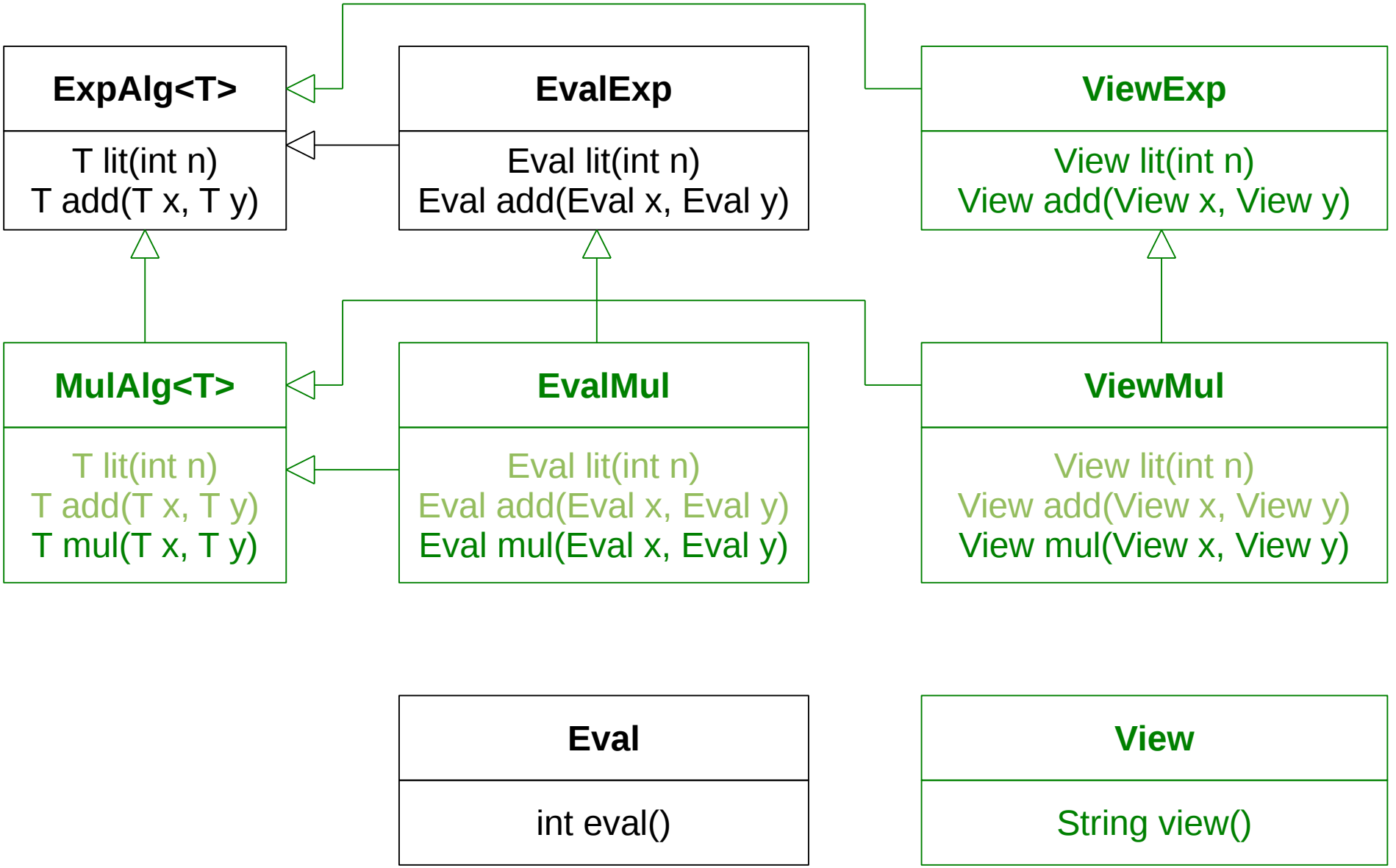




Object Algebras: Adding Variants



Object Algebras: Adding Operations



Finally Tagless

Finally Tagless, Partially Evaluated*

Tagless Staged Interpreters for Simpler Typed Languages

Jacques Carette[1], Oleg Kiselyov[2], and Chung-chieh Shan[3]

[1] McMaster University carette@mcmaster.ca
[2] FNMOC oleg@pobox.com
[3] Rutgers University ccshan@rutgers.edu

Finally Tagless: Terms

```java
interface ExpAlg<T> {
    T lit(int n);
    T add(T x, T y);
}

<T> T e1(ExpAlg<T> f) {
    return f.add(
        f.lit(1),
        f.add(
            f.lit(2),
            f.lit(3)));
}
```


```haskell
class ExpAlg t where
    lit :: Int -> t
    add :: t -> t -> t


e1 :: ExpAlg t => t
e1 = add (lit 1)
         (add (lit 2)
              (lit 3))
```

Finally Tagless: Operations

```
interface Eval { int eval(); }

class EvalExp implements ExpAlg<Eval> {
    Eval lit(final int n) {
        return new Eval() {
            int eval() {
                return n;
            }
        }
    }

    Eval add(final Eval x, final Eval y) {
        return new Eval() {
            int eval() {
                return x.eval() + y.eval();
            }
        }
    }
}

int v1 = e1(new EvalExp()).eval();
```

Finally Tagless: Operations

```
class Eval { public int eval; }

class EvalExp implements ExpAlg<Eval> {
    Eval lit(final int n) {
        return new Eval(n);
    }

    Eval add(final Eval x, final Eval y) {
        return new Eval(x.eval + y.eval);
    }

}

int v1 = e1(new EvalExp()).eval;
```

Finally Tagless: Operations

```
newtype Eval = Eval { eval :: Int }

instance ExpAlg Eval where
    lit n   = Eval n

    add x y = Eval $ eval x + eval y

v1 = eval (e1 :: Eval)
```

Finally Tagless: Adding Variants

```java
interface MulAlg<T>
extends ExpAlg<T> {
    T mul(T x, T y);
}

<T> T e2(MulAlg<T> f) {
    return f.mul(
        f.lit(4),
        f.add(
            f.lit(5),
            f.lit(6)));
}
```


```haskell
class ExpAlg t => MulAlg t where
    mul :: t -> t -> t



e2 :: MulAlg t => t
e2 = mul (lit 4)
         (add (lit 5)
              (lit 6))
```

Finally Tagless: Adding Variants

```
interface Eval { int eval(); }

class EvalMul extends EvalExp implements MulAlg<Eval> {
    Eval mul(final Eval x, final Eval y) {
        return new Eval() {
            int eval() {
                return x.eval() * y.eval();
            }
        }
    }
}


int v2 = e2(new EvalMul()).eval();
```

Finally Tagless: Adding Variants

```
class Eval { public int eval; }

class EvalMul extends EvalExp implements MulAlg<Eval> {
    Eval mul(final Eval x, final Eval y) {
        return new Eval(x.eval * y.eval);
    }




}


int v2 = e2(new EvalMul()).eval;
```

Finally Tagless: Adding Variants

```
newtype Eval = Eval { eval :: Int }

instance MulAlg Eval where
    mul x y = Eval $ eval x + eval y




v2 = eval (e2 :: Eval)
```

Finally Tagless: Adding Operations

```
interface View { String view(); }

class ViewExp implements ExpAlg<View> {
    View lit(final int n) {
        return new View() {
            int view() {
                return Integer.toString(n);
            }
        }
    }

    View add(final View x, final View y) {
        return new View() {
            int view() {
                return "(" + x.view() + " + " + y.view() + ")";
            }
        }
    }
}

String s1 = e1(new ViewExp()).view();
```

Finally Tagless: Adding Operations

```
class View { public String view; }

class ViewExp implements ExpAlg<View> {
    View lit(final int n) {
        return new View(Integer.toString(n));
    }




    View add(final View x, final View y) {
        return new View("(" + x.view + " + " + y.view + ")");
    }



}

String s1 = e1(new ViewExp()).view;
```

Finally Tagless: Adding Operations

```
newtype View = View { view :: String }

instance ExpAlg View where
    lit n   = View $ show n

    add x y = View $ "(" + view x + " + " + view y + ")"

s1 = view (e1 :: View)
```

Finally Tagless

```
class ExpAlg t where
    lit :: Int -> t
    add :: t -> t -> t
```

```
newtype Eval = Eval { eval :: Int }

instance ExpAlg Eval where
    lit n   = Eval n
    add x y = Eval $ eval x + eval y
```

Finally Tagless

```
class ExpAlg t where
    lit :: Int -> t
    add :: t -> t -> t
```

```
class ExpAlg t => MulAlg t where

    mul :: t -> t -> t
```

```
newtype Eval = Eval { eval :: Int }

instance ExpAlg Eval where
    lit n   = Eval n
    add x y = Eval $ eval x + eval y
```

```
instance MulAlg Eval where

    mul x y = Eval $ eval x * eval y
```

Finally Tagless

```
class ExpAlg t where
    lit :: Int -> t
    add :: t -> t -> t
```

```
class ExpAlg t => MulAlg t where

    mul :: t -> t -> t
```

```
newtype Eval = Eval { eval :: Int }

instance ExpAlg Eval where
    lit n   = Eval n
    add x y = Eval $ eval x + eval y
```

```
instance MulAlg Eval where

    mul x y = Eval $ eval x * eval y
```

```
newtype View = View { view :: String }

instance ExpAlg View where
    lit n   = View $ show n
    add x y =
      View $
        "(" ++ view x ++ " + "
            ++ view y ++
        ")"
```

Finally Tagless

```
class ExpAlg t where
    lit :: Int -> t
    add :: t -> t -> t
```

```
class ExpAlg t => MulAlg t where

    mul :: t -> t -> t
```

```
newtype Eval = Eval { eval :: Int }

instance ExpAlg Eval where
    lit n   = Eval n
    add x y = Eval $ eval x + eval y
```

```
instance MulAlg Eval where

    mul x y = Eval $ eval x * eval y
```

```
newtype View = View { view :: String }

instance ExpAlg View where
    lit n   = View $ show n
    add x y =
      View $
        "(" ++ view x ++ " + "
            ++ view y ++
        ")"
```

```
instance MulAlg View where

    mul x y =
      View $
        "(" ++ view x ++ " * "
            ++ view y ++
        ")"
```