Programming with Algebras

Bologna, March 2015

Bartosz Milewski

Why Algebra?

- Not just for solving equations
- Algebraic data types, recursive data types
- Computations using algebras

What is Algebra?

- Ability to form expressions with operators, numbers, and symbols: $x^2 + 2xy + y^2$
- Ability to evaluate expressions: $x=1.5$, $y=0.5$,
  - $x^2 + 2xy + y^2 = 4$
- Addition, multiplication, vectors, inner product, outer product

Expression

What is Expression?

```
data Expr = Const Int
          | Add Expr Expr
          | Mul Expr Expr
```

- Grammar for building expressions
- Recursive algebraic data structure

Algebraic Data Structures

- Unit: void, singleton `()` `Const () a`
- Const: ignore type argument
  - `data Const c a = Const c`
- Identity `a`
- Product: pair, (a, b), tuple, (a, b, c), struct, record
  - `data Pair a a’ = P a a’`
- Sum: tagged union, variant
  - `Either a a’ = Left a | Right a’`

Expression as Algebraic Data Type

```
data Expr = Const Int
          | Add Expr Expr
          | Mul Expr Expr
```

- Sum of Const and two products

```
Const Int a
```

```
Add a a
```

```
Mul a a
```

- What about recursion?

Functor

- Mapping of types: Type constructor

```haskell
data ExprF a = Const Int
             | Add a a
             | Mul a a
```

- Mapping of functions: fmap

```haskell
fmap :: (a -> b) -> (ExprF a -> ExprF b)
fmap f (Const i) = Const i
fmap f (Add x y) = Add (f x) (f y)
fmap f (Mul x y) = Mul (f x) (f y)
```

Recursion

```
data ExprF a = Const Int
             | Add a a
             | Mul a a
```

```
ExprF (ExprF a)
ExprF (ExprF (ExprF a))
ExprF (ExprF (ExprF (ExprF a)))
```

- Trees of depth 1, 2, 3, …, infinity?

Fixed Point

- f is an arbitrary functor (type constr. + fmap)

```
f a
f (f a)
f (f (f a))
f (f (f (f a)))
```

- Infinite depth: One more iteration doesn't change anything

```
newtype Fix f = In (f (Fix f))
```

Expression as Fixed Point

```
data ExprF a = Const Int
             | Add a a
             | Mul a a
```

```
type Expr = Fix ExprF
```

```
data Expr = Const Int
          | Add Expr Expr
          | Mul Expr Expr
```

Evaluation

What is Evaluation?

- Extracting value from expression
- Many ways of evaluating the same expression

```
alg :: ExprF Int -> Int

alg (Const i)    = i
alg (x `Add` y) = x + y
alg (x `Mul` y) = x * y
```

```
alg'' :: ExprF Complex ->
alg''
alg''
alg''
```

```
alg' :: ExprF String -> String
alg' (Const i)    = [chr (ord 'a' + i)]
alg' (x `Add` y) = x ++ y
alg' (x `Mul` y) = concat [[a, b] | a <- x, b <- y]
```

What is an F-Algebra?

- A functor, like ExprF
- A type, like Int (carrier type)
- A function, like alg :: ExprF Int -> Int

```
type Algebra f a = f a -> a
```

Evaluating Recursive Expressions

- Given type a and function alg :: f a -> a, evaluate an expression given by Fix f
- Given alg :: ExprF Int -> Int, (carrier type: Int) evaluate Expr = Fix ExprF

Building Catamorphism

```
newtype Fix f = In (f (Fix f))
```

```
unFix :: Fix f -> f (Fix f)
unFix (In x) = x
```

- Define cata :: Functor f => (f a -> a) -> Fix f -> a
- Use unFix to extract: f (Fix f)
- Use fmap (cata alg) to evaluate children (recursion!)
- Apply alg to evaluate top level

Catamorphism

```haskell
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix
```

```haskell
alg (Const i)     = i
alg (x `Add` y) = x + y
alg (x `Mul` y) = x * y
```

```haskell
eval :: Fix ExprF -> Int
-- eval = cata alg
eval = alg . fmap eval . unFix
```

List

```
data ListF e a = Nil | Cons e a
    deriving Functor
type List e = Fix (ListF e)
```

```
algSum :: ListF Int Int -> Int
algSum Nil = 0
algSum (Cons e acc) = e + acc
```

```
cata algSum lst =
(algSum . fmap algSum . unFix) lst =
foldr (\e acc -> e + acc) 0 lst
```

Conclusion

- Very general formula for evaluating recursive data structures
    - Data type defined as a fixed point of a functor
    - Algebra for this functor to evaluate single level
    - Catamorphism to evaluate recursive data
- More general recursion schemes (Erik Meijer et al.)