# Data types à la carte

## FP AMS – 21/6/18

Wouter Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

# Warm-up: expressions in Haskell

Suppose we're implementing a small expression language in Haskell.

We can define a data type for expressions and evaluation function easily enough:

```
data Expr = Val Int | Add Expr Expr

eval :: Expr -> Int
eval (Val x)    = x
eval (Add l r)  = eval l + eval r
```

# Warm-up: expressions in Haskell

Suppose we're implementing a small expression language in Haskell.

We can define a data type for expressions and evaluation function easily enough:

```
data Expr = Val Int | Add Expr Expr

eval :: Expr -> Int
eval (Val x)    = x
eval (Add l r)  = eval l + eval r
```

That's it – we can go home.

# Handling changes

Code is never finished – how can we handle changing requirements?

We can add **new functions** easily enough – we don't even have to modify any existing code

```
render :: Expr -> String
render (Val x)     = show x
render (Add l r)  =
  parens (show l ++ " + " ++ show r)
```

# Handling changes

Code is never finished – how can we handle changing requirements?

We can add **new functions** easily enough – we don't even have to modify any existing code

```
render :: Expr -> String
render (Val x)     = show x
render (Add l r)   =
  parens (show l ++ " + " ++ show r)
```

But we cannot add **new constructors** without modifying the datatype and all functions defined over it.

# FP vs OO

This situation is dual to that in object oriented languages.

There, we can add **new subclasses** to a class easily enough…

…but adding **new methods** requires updating every subclass.

# The Expression Problem

Phil Wadler dubbed this 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).
```

# The Expression Problem

Phil Wadler dubbed this 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).
```

How can we address the Expression Problem in Haskell?

# A naive approach

```
data IntExpr = Val Int | Add Expr Expr

data MulExpr = Mul IntExpr Intexpr

type Expr = Either IntExpr MulExpr

data Either a b = Inl a | Inr b
```

## Question

What is wrong with this approach?

# A naive approach

```
data IntExpr = Val Int | Add Expr Expr

data MulExpr = Mul IntExpr Intexpr

type Expr = Either IntExpr MulExpr

data Either a b = Inl a | Inr b
```

## Question

What is wrong with this approach?
We cannot freely mix addition and multiplication.

# The problem

```
data Expr = ...
```

What constructors should we choose?

# The problem

```
data Expr = ...
```

What constructors should we choose?

Whenever we choose the constructors, we're stuck – we won't be able to add new ones easily.

# Fixpoints

```
data Expr f = In (f (Expr f))
```

- the type variable `f` abstracts over the constructors of our data type;
- the type variable `f` has kind `* -> *` – it's a type constructor like `List` – it abstracts over the recursive occurrences of *subtrees*.
- By applying `f` to `Expr f`, we'll replace the type variables in `f` with these subtrees – similar to writing recursion explicitly using `fix` or the Y-combinator.
- I'll sometimes refer to `f` as a (pattern) functor.

# Evaluation revisited

```haskell
data AddF a = Val Int | Add a a

data Expr f = In (f (Expr f))

eval (In (Val x)) = x
eval (In (Add l r)) = eval l + eval r
```

We don't seem to have gained much, except for some syntactic noise…

# Combining functors

We can combine functors in a very similar manner to the `Either` data type:

```
data (f :+: g) r = Left (f r) | Right (g r)
```

Using this insight, we can grow our expressions step by step.

# Example: adding multiplication

```haskell
data Expr f = In (f (Expr f))

data AddF a = Val Int | Add a a
data MulF a = Mul a a

type AddExpr    = Expr AddF
type AddMulExpr = Expr (AddF :+: MulF)

addExample :: Expr (MulF :+: AddF)
addExample = In (Inl (Mul (In (Inr (Val 1)))
                          (In (Inr (Val 2)))))
```

This gives us the machinery to assemble *data types à la carte*.

## Problems

- Constructing expressions is a pain: nobody wants to write injections by hand.
- How can we define functions over these expressions?

# Functions over expressions

Usually, we write functions through pattern matching on a fixed set of branches.

But pattern matching on our constructors is painful (we have lots of injections in the way).

And pattern matching *fixes* the possible patterns that we accept.

# Functions over expressions

Usually, we write functions through pattern matching on a fixed set of branches.

But pattern matching on our constructors is painful (we have lots of injections in the way).

And pattern matching *fixes* the possible patterns that we accept.

## Idea

Use Haskell's class system to *assemble* functions for us!
Before we do this, however, we need to talk about functors and folds.

# Folds

**Folds** capture a common pattern of traversing a data structure and computing some value.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr cons nil []     = nil
foldr cons nil (x:xs) = cons x (foldr cons nil xs)
```

But this also works for other data types!

# Folding lists – contd.

```
foldr :: (a -> r -> r) -> r -> [a] -> r
```

Compare the types of the constructors with the types of the arguments:

```
(:)    ::  a  ->  [a]  ->  [a]
[]     ::  a  ->  [a]

cons   ::  a  ->  b    ->  b
nil    ::  a  ->  b
```

# Folding on trees

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

foldTree :: (b -> b -> b) -> (a -> b) -> Tree a -> b
foldTree node leaf (Leaf x)   = leaf x
foldTree node leaf (Node l r) =
  node (foldTree node leaf l) (foldTree node leaf r)
```

# Ideas in each fold

- Replace constructors by user-supplied arguments.
- Recursive substructures are replaced by recursive calls.

Can we give an account that works for any data type?

# Catamorphism generically

If we know the the recursive positions, we can express the fold or *catamorphism* generically:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

cata :: (Functor f) =>
          (f a -> a) -> Expr f -> a
cata phi (In t) = phi (fmap (cata phi) t)
```

The argument to `cata` describing how to handle each constructor, `f a -> a`, is sometimes called an *algebra*.

# Functions over expressions

We can use the `cata` function to traverse our expressions:

```
cataAdd :: Expr AddF -> Int
cataAdd = cata alg
  where
    alg (Add x y) = x + y
    alg (Val x)   = x
```

# Functions over expressions

We can use the `cata` function to traverse our expressions:

```
cataAdd :: Expr AddF -> Int
cataAdd = cata alg
  where
    alg (Add x y) = x + y
    alg (Val x)   = x
```

But can we do something more open ended?

# Algebras using classes

More generally, to define a function over an expression – without knowing the constructors – we introduce a new type class:

```
class Eval f where
  evalAlg :: f Int -> Int

eval :: Eval f => Expr f -> Int
eval = cata evalAlg
```

# Functions over expressions

We can now add instance for all the constructors that we wish to support:

```
instance Eval AddF where
  evalAlg (Add l r) = l + r
  evalAlg (Val i)   = i

instance Eval MulF where
  evalAlg (Mul l r) = l * r

...
```

# Functions over expressions

To assemble the desired algebra, however, we need one more instance:

```
instance (Eval f, Eval g) => Eval (f :+: g) where
  evalAlg x = ...
```

## Question

What should this instance be?

# Functions over expressions

To assemble the desired algebra, however, we need one more instance:

```
instance (Eval f, Eval g) => Eval (f :+: g) where
  evalAlg (Inl x) = evalAlg x
  evalAlg (Inr y) = evalAlg y
```

# The Expression Problem

- How can we write functions over expressions?
  - Use type classes
- Constructing expressions is a pain:

```
addExample :: Expr (MulF :+: AddF)
addExample = In (Inl (Mul (In (Inr (Val 1)))
                          (In (Inr (Val 2)))))
```

# The Expression Problem

- How can we write functions over expressions?
  - Use type classes
- Constructing expressions is a pain:

```
addExample :: Expr (MulF :+: AddF)
addExample = In (Inl (Mul (In (Inr (Val 1)))
                          (In (Inr (Val 2))))))
```

## Idea

Define smart constructors!

# Not so smart constructors

For any fixed pattern functor, we can define auxiliary functions to assemble datatypes:

```
data AddF a = Val Int | Add a a
type AddExpr = Expr AddF

add :: AddExpr -> AddExpr -> AddExpr
add l r = In (Add l r)
```

But how can we handle coproducts of pattern functors?

# Automating injections

To deal with coproducts, we introduce a type class describing *how* to inject some 'small' pattern functor `sub` into a larger one `sup`:

```
class (:<:) sub sup where
  inj :: sub a -> sup a
```

What instances are there?

# Instances

```
class (:<:) sub sup where
  inj :: sub a -> sup a

instance (:<:) f f where
  inj = ...
instance (:<:) f (f :+: g) where
  inj = ...
instance ((:<:) f g) => (:<:) f (h :+: g) where
  inj = ...
```

## Question

How should we complete the above definitions?

# Instances

```
class (:<:) sub sup where
  inj :: sub a -> sup a

instance (:<:) f f where
  inj = id
instance (:<:) f (f :+: g) where
  inj = Inl
instance ((:<:) f g) => (:<:) f (h :+: g) where
  inj = inj . Inr
```

# Smart constructors

```
inject :: ((:<:) g f) => g (Expr f) -> Expr f
inject = In . inj

val :: (AddF :<: f) => Int -> Expr f
val x    = inject (Val x)

add :: (AddF :<: f) => Expr f -> Expr f -> Expr f
add x y  = inject (Add x y)

mul :: (MulF :<: f) => Expr f -> Expr f -> Expr f
mul x y = inject (Mul x y)
```

# Results!

```haskell
e1 :: Expr AddF
e1 = val 1 `add` val 2

v1 :: Int
v1 = eval e1

e2 :: Expr (MulF :+: AddF)
e2 = val 1 `mul` (val 2 `add` val 3)

v2 :: Int
v2 = eval e2
```

# Extensibility

We can easily add new constructors:

```
data SubF a = SubF a a

type NewExpr = SubF :+: MulF :+: AddF
```

Or define new functions:

```
class Render f where
  render :: f String -> String
```

# General recursion

What if we would like to define recursive functions without using folds?

A first attempt might be:

```
class Render f where
  render :: f (Expr f) -> String
```

# General recursion

What if we would like to define recursive functions without using folds?

A first attempt might be:

```
class Render f where
  render :: f (Expr f) -> String
```

But this is too restrictive! We require `f` and the recursive pattern functors (`Expr f`) to be the same.

# Generalizing

A more general type seems better:

```
class Render f where
  render :: f (Expr g) -> String
```

We can try to define an instance:

```
instance Render Mul where
  render :: Mul (Expr g) -> String
  render (Mul l r) = ...
```

But now we cannot make a recursive call! We don't know that the pattern functor g can be rendered.

# General recursion

```
class Render f where
  render :: Render g => f (Expr g) -> String

instance Render Mul where
  render :: Mul (Expr g) -> String
  render (Mul l r) = renderExpr l
                  ++ " * "
                  ++ renderExpr r

renderExpr :: Render f => Expr f -> String
renderExpr (In t) = render t
```

# Recap

- Pattern functors give us the mathematical machinery to describe and recursive datatypes.
- We can define a generic fold operation (`cata`);
- We can use Haskell's type classes to assemble modular datatypes and functions!

# Looking back

- Pearl matured into bigger libraries, addressing some limitations of the injections (Patrik Bahr et al.)
- Inspired work in other languages, such as *The expression problem, trivially* (Wang & Oliveira), or *Meta-theory à la carte* (Delaware et al.).
- The key ideas were already written by Luc Duponcheel twenty years ago!

# Further topics

- So you can combine datatypes – but can you combine *monads*?
- Why did you choose the `:+:` operator? Why are Haskell's data types called algebraic?
- What are Church encodings?



Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

# Combining monads?

The `:+:` operator is the canonical way to combine the constructors of a datatype.

Can we use the same operation to combine monads?

That is, if `m1` and `m2` are monads, can we construct a monad `m1 :+: m2`?

# Combining monads?

The `:+:` operator is the canonical way to combine the constructors of a datatype.

Can we use the same operation to combine monads?

That is, if `m1` and `m2` are monads, can we construct a monad `m1 :+: m2`?

The paper 'Composing Monads Using Coproducts' explores this idea.

This construction works, but does not account for the 'interaction' between `m1` and `m2`.

Yet there is a class of monads for which this construction does work.

# Get-Put

In the labs, we saw the following data type:

```
data Teletype a =
  Get (Char -> Teletype a)
  | Put Char (Teletype a)
  | Return a

instance Monad Teletype where
  ...
```

Can we describe this using pattern functors?

# Using pattern functors

```haskell
data TeletypeF r =
  Get (Char -> r)
  | Put Char r

data Teletype a =
  In (TeletypeF (Teletype a))
  | Return a
```

# Free monads

We can capture this pattern as a so-called *free monad*:

```
data Free f a =
  In (f (Free f a))
  | Return a
```

For any functor `f` this definition is a monad.

Question

Why? What other familiar monads are free?

```haskell
instance (Functor f) => Monad (Term f) where
  return x            = Return x
  (Return x) >>= f    = f x
  (In t)  >>= f       = In (fmap (>>= f) t)
```

# Combining monads

Using the same machinery we saw previously, we can combine *free* monads in a uniform fashion.

```
data FileSystem a =
      ReadFile FilePath (String -> a)
   |  WriteFile FilePath String a

class Functor f => Exec f where
  execAlgebra :: f (IO a) -> IO a

cat ::  FilePath ->  Term (Teletype :+: FileSystem) ()
```

This gives us a more fine-grained collection of *effects* that can all be run in the IO monad.

# Algebraic datatypes

Haskell's data types are sometimes called *algebraic datatypes* – why?

# Algebraic datatypes

The `:+:` and `:*:` (pairing) operators behave similarly to `*` and `+` on numbers. The unit type `()` is a like `1`.

For example, for any type `t` we can show `1 * t` is isomorphic to `t`.

Or for any types `t` and `u`, we can show `t * u` is isomorphic to `u * t`.

Similarly, `t :+: u` is isomorphic to `u :+: t`.

## Question

What is the unit of `:+:`?

# Church encodings revisited

Using this definition, we can now give a more precise account of the *Church encoding* of algebraic data structures that we saw previously.

The idea behind Church encodings is that we identify:

- a data type (described as the least fixpoint of a functor)
- the fold over this datatype

# Church encoding: lists

```
type Church a = forall r . r -> (a -> r -> r) -> r

-- reconstruct a list by applying constructors
from :: Church a -> [a]
from f = ...

-- map a list to its fold
to :: [a] -> Church a
to xs = ...
```

# Church encoding: lists

```
type Church a = forall r . r -> (a -> r -> r) -> r

-- reconstruct a list by applying constructors
from :: Church a -> [a]
from f = f [] (:)

-- map a list to its fold
to :: [a] -> Church a
to xs = \nil cons -> foldr cons nil xs
```

# Generic Church encoding

```
type Church f = forall r . (f r -> r) -> r

cata :: Functor f => (f a -> a) -> Fix f -> a
cata f (In t) = f (fmap (cata f) t)

to :: Functor f => Fix f -> Church f
to t = \f -> cata f t

from :: Functor f => Church f -> Fix f
from f = f In
```