

Data types à la carte

Wouter Swierstra
Dutch HUG 25/8/10

Expressions

data Expr where

Add :: Expr -> Expr -> Expr

Val :: Int -> Expr

eval :: Expr -> Int

eval (Val x) = x

eval (Add l r) = eval l + eval r

Adding new features

- In Haskell it's easier to define new functions, such as:

```
print :: Expr -> String
```

- But what about adding new alternatives to the data type, such as multiplication?
- We'll need to add new cases to every function we've already defined.

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). – Phil Wadler, 1998

OO languages

- In Object Oriented languages, it is usually easy to add new data type alternatives (by defining new classes);
- But defining new functions means modifying all existing classes...

Expr revisited

data Expr = ...

What constructors should we choose?

Expr revisited

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

Expr revisited

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

Abstract over
constructors




Expr revisited

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

Abstract over
constructors



Constructors
abstract over
recursive calls



Adding constructors...

```
data Val e = Val Int
```

```
data Add e = Add e e
```

```
type ValExpr = Expr Val
```

```
type AddExpr = Expr Add
```

Building types

```
data Val e = Val Int
```

```
data Add e = Add e e
```

```
data ( :+: ) f g e =
```

```
    Inl (f e)
```

```
  | Inr (g e)
```

```
type Both = Expr (Val :+: Add)
```

Example

```
addExample :: Expr (Val :+: Add)
```

```
addExample = In (Inr (Add (In (Inl  
(Val 32))) (In (Inl (Val 10)))))
```

What next?

- We can define modular data types in this fashion.
- But how can we define modular functions?
- How can we build values easily?

Functors

```
data Val e = Val Int
```

```
data Add e = Add e e
```

```
class Functor where
```

```
    fmap :: (a -> b) -> f a -> fb
```

```
instance Functor Add where
```

```
    fmap f (Add l r) = Add (f l) (f r)
```

```
instance Functor Val where
```

```
    fmap f (Val i) = Val i
```

Why functors?

`fold :: Functor f =>`

`(f a -> a) -> Expr f -> a`

`fold f (In t) = f (fmap (fold f) t)`

Defining evaluation

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

instance Eval Val where
  evalAlg (Val i) = i

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

Putting the pieces together

```
class Functor f => Eval f where
  evalAlg :: f Int -> Int
instance (Eval f, Eval g) =>
  Eval (f :+: g) where
  evalAlg (Inl f) = evalAlg f
  evalAlg (Inr g) = evalAlg g
```

Defining eval

```
eval :: Eval f => Expr f -> Int  
eval expr = fold evalAlg expr
```

```
*Main> eval addExample
```

```
42
```

Modular functions

- Show that all your constructor types are functors.
- Define a class for every function that you want to define.
- Add instances for every constructor.
- Use the class system to assemble the pieces.

Smart constructors

- Writing out `Inl/Inr/In` by hand is tiring and error-prone.
- How can we automate this?

```
x :: Expr (Val :+: Add)
```

```
x = val 3 <+> val 5
```

A first attempt...

$val :: Int \rightarrow Expr\ Val$

$val\ x = In\ (Val\ x)$

$(<+>) :: Expr\ Add \rightarrow Expr\ Add$

$\rightarrow Expr\ Add$

$l\ <+>\ r = In\ (Add\ l\ r)$

A first attempt...

$\text{val} :: \text{Int} \rightarrow \text{Expr Val}$

$\text{val } x = \text{In } (\text{Val } x)$

$(\langle + \rangle) :: \text{Expr Add} \rightarrow \text{Expr Add}$

$\rightarrow \text{Expr Add}$

$l \langle + \rangle r = \text{In } (\text{Add } l r)$

But this is non-modular!

What we'll achieve

```
val :: Val :<: f => Int -> Expr f
```

```
val x = In (inject x)
```

```
(<+>) :: Add :<: f =>
```

```
Expr f -> Expr f -> Expr f
```

```
l <+> r = In (inject (Add l r))
```

Finding Injections

```
class sub :<: sup where  
  inject :: sub a -> sup a  
instance f :<: f where  
  inject x = x  
instance f :<: (f :+: g) where  
  inject x = Inl x  
instance f :<: g =>  
  f :<: (h :+: g) where  
  inject x = Inr (inject x)
```

Taking stock

- How hard is it to add new functions?
- Or new constructors?

Adding multiplication

```
data Mul e = Mul e e
```

```
instance Functor Mul where
```

```
  fmap f (Mul l r) = Mul (f l) (f r)
```

```
instance Eval Mul where
```

```
  evalArg (Mul x y) = x * y
```

```
(<*>) l r = In (inject (Mul l r))
```

Example

```
t :: Expr (Mul :+: Add :+: Val)
```

```
t = 1 <+> (2 <*> 3)
```

```
*Main> eval t
```

```
7
```

Adding pretty printing

```
class Render f where
```

```
  render :: Render g =>
```

```
    f (Expr g) -> String
```

```
instance Render Add where
```

```
  render (Add l r) = parens $
```

```
    render l ++ "+" ++ render r
```

```
instance Render Val where
```

```
  render (Val x) = show x
```

Adding pretty printing

```
class Render f where
```

```
  render :: Render g =>
```

```
    f (Expr g) -> String
```

```
instance (Render f, Render g) =>
```

```
  Render (f :+: g) ...
```

```
pretty :: Render f => Expr f -> String
```

```
pretty (In t) = render t
```

Conclusions

- This works well for simple data types...
- But mutually recursive/polymorphic/nested/generalized algebraic data types are harder.
- The same technology can be used to combine (a certain class of) monads.