



Fusion-powered EDSLs

Philippa Cowderoy

`flippa@flippac.org`

Outline

- What is fusion?
- Anatomy of an EDSL
- Shallow vs deep embedding
- Examples: Identity & State monads
- Self-analysing EDSL implementations
- Summary, Wishlist, Questions

What is Fusion?

Given:

- A "producer"
- A "consumer"

We can often rewrite the expression *consumer producer* so that there is no intermediate value.

e.g. `map f (map g xs) → map (f.g) xs`

Result of `producer` must only be used by `consumer`:

Otherwise we still need the intermediate!

Anatomy of an EDSL

- Language primitives:
char, >>=, catch, atomic
- Non-primitive combinators and helpers:
many, mapM, chainl1
- Execution functions:
runParser, runError, runToTheShops
- Analyses:
Error-checking, grammar analyses, possible optimisations...

Anatomy of an EDSL

Syntax



Constructors: producers, form an algebra

$\gg=$, *catch*, *mapM*, *chainl1*



Internal Representation



Deconstructors: consumers, form a coalgebra

runParser, *runError*



Semantics

Shallow Embedding

```
type Parser a = String -> [(a,String)]
```

```
char :: Char -> Parser Char
```

```
char c (c':s) | c == c' = [(c,s)]
```

```
char _ _ = []
```

```
...
```

```
runParser :: Parser a -> String ->  
          [(a,String)]
```

```
runParser = ($) 
```

Deep Embedding

```
data Parser a where
```

```
...
```

```
Char :: Char -> Parser Char
```

```
runParser :: Parser a -> String ->  
           [(a,String)]
```

```
...
```

```
runParser (Char c) (c':s)  
  | c == c' = [(c,s)]
```

```
runParser (Char c) _ = []
```

Shallow vs Deep Embedding

	Deep	Shallow
Structure	Intermediate	No intermediate
Primitive Constructors	Denote syntax Hard to add?	Denote semantics Easily added?
Primitive Deconstructors	syntax \rightarrow semantics easily added	Trivial hard to add

A Trivial Fusion Example

```
data Id a where
```

```
  Return :: a -> Id a
```

```
  Bind   :: Id a -> (a -> Id b) -> Id b
```

```
runIdentity :: Id a -> a
```

```
runIdentity (Return v) = v
```

```
runIdentity (Bind l fr) =
```

```
  let l' = runIdentity l
```

```
      r = fr l'
```

```
  in runIdentity r
```

A Trivially Transformed Example

Change types:

```
newtype Id a = {runIdentity::a}
```

Rewrite the implementation according to:

$(return, bind) = runIdentity(Return, Bind)$

Result:

```
return v = Id v
```

```
bind l fr = Id f where
```

```
  f = let l' = runIdentity l
```

```
      r = fr l'
```

```
      in runIdentity r
```

State Monad

```
data State s a where ...
```

```
  Bind :: State s r ->  
        (r -> State s a) ->  
        State s a
```

```
  Get  :: State s s
```

```
runState :: State s a -> s -> (s,a)
```

```
runState (Bind c1 f) s0 =
```

```
  let (s1,r1) = runState c1 s0
```

```
      c2 = f r1
```

```
      in runState c2 s1
```

```
runState Get s = (s,s)
```

Stating the Obvious?

```
data State s a =  
  State {runState :: s -> (s,a)}
```

```
bind c1 f = State c where  
  c s0 = let (s1,r1) = runState c1 s0  
          c2 = f r1  
          in runState c2 s1
```

```
get = State c where  
  c s = (s,s)
```

Note:

```
runState :: State s a -> s -> (s,a)  
=> State s a = State {runState :: s -> (s,a)}
```

DSLs that Think Too Much

What if we want to do self-analysis like those shiny parsing combinators? Isn't that hard?

Not really!

Parsing

```
data Parser r where
```

```
Char :: Char -> Parser Char
```

```
Ap :: Parser a -> Parser (a -> r) -> Parser r
```

```
Pure :: r -> Parser r
```

```
Choice :: Parser r -> Parser r -> Parser r
```

```
Empty :: Parser r
```

```
Many :: Parser r -> Parser [r]
```

A simple Applicative parser. Grammar is context-free - never determined by intermediate results - so we can analyse it.

Mere semantics?

Here are the semantic functions:

```
runParser :: Parser a -> String ->
           Maybe (a, String)
```

```
firsts :: Parser a -> Map Char (Parser a)
```

runParser calls firsts, using a first set analysis to speed itself up.

Moving to a shallow embedding, we need to carry two values in the record:

```
newtype Parser a =
  Parser {runParser :: String -> Maybe a,
          firsts    :: Map Char (Parser a)}
```

Results

- Memoisation of static results for free!
- Still room for further efficiency gains - e.g. swapping Maybe for a more efficient error monad, attacking the usual memory leak problem
- There's plenty of room for small optimisations, but many of them can be made to the original version of the code too or left to well-known GHC optimisations.
- Control.Applicative.many (like many other functions that use general recursion) messes up the static analysis whether in deep or shallow form :-)

Summary

To translate from deep to shallow:

- Replace the term datatype with a record type - a field for each semantic function
- Replace the term constructors with records - the fields containing the constructor's case from each semantic function
- Don't worry about sharing/exploiting laziness in the deep version, the shallow one will do it automatically!

Wishlist

- Better support for object-level recursion - this probably means sugar
- Why am I fusing all this stuff by hand anyway? I know all the conditions required to make it work!
- Pointed syntactic sugar for Applicatives would be nice
- Something for my parser's toothache?

Finish

Both the slides from this talk and a lengthier parsing example will be available on the web from <http://flippac.org/talks/> and linked to from the AngloHaskell 2008 page.

Any questions?