# Advanced Topics in Programming Languages - Paper Overview

Riley Evans (re17105)

```
{-# LANGUAGE KindSignatures, GADTs, RankNTypes, LambdaCase #-}
```

```
import Prelude hiding (or)
```

The original paper used a language for parallel prefix circuits. In this overview another simple DSL will be used to demonstate the techniques described to fold DSLs. That language is a simple parser combinator language, it contains the basic operations needed to build a larger combinator language.

In the same way as before we can construct parsers using the following operators:

```
type Parser1 a = Int

pure :: a → Parser1 a
pure = ⊥
satisfy :: (Char → Bool) → Parser1 Char
satisfy = ⊥
empty :: Parser1 a
empty = ⊥
try :: Parser1 a → Parser1 a
try = ⊥
ap :: Parser1 (a → b) → Parser1 a → Parser1 b
ap = ⊥
or :: Parser1 a → Parser1 a → Parser1 a
or = ⊥
```

It is possible to form a parser for the string 'a' using

```
a :: Parser1 Char
a = satisfy (≡ 'a')


b :: Parser1 Char
b = satisfy (≡ 'b')


aorb :: Parser1 Char
aorb = a `or` b
```

We can specify a datatype to represent this parser as a deeply embedded DSL.

```
data Parser2 :: * → * where
  Pure2 :: a → Parser2 a
  Satisfy2 :: (Char → Bool) → Parser2 Char
  Empty2 :: Parser2 a
  Try2 :: Parser2 a → Parser2 a
  Ap2 :: Parser2 (a → b) → Parser2 a → Parser2 b
  Or2 :: Parser2 a → Parser2 a → Parser2 a
```

It is simple to define functions to manipulate this deep embedding. For example, one could be used to find the size of the parser.

```
type Size = Int
size :: Parser2 a → Size
size (Pure2 _) = 1
size (Satisfy2 _) = 1
size Empty2 = 1
size (Try2 px) = 1 + size px
size (Ap2 pf px) = size pf + size px + 1
size (Or2 px py) = size px + size py + 1
```

It is clear that size is a fold over Parser2, hence it is a suitable semantics for a shallow embedding.

```
type Parser3 a = Int
pure3 _ = 1
satisfy3 _ = 1
empty3 = 1
try3 px = px + 1
ap3 pf px = pf + pf + 1
or3 px py = px + py + 1
size3 :: Parser3 a → Size
size3 = id
```

## 1 Folds

Blah blah

The shape is able to be captured in an instance of the Functor type class. In a difference to the paper Parsers are a typed DSL. Therefore, we need to define an instance of the IFunctor type class, in order to retain these types. TODO: Type indices

```
class IFunctor f where
  imap :: (forall i ∘ a i → b i) → f a i → f b i

data ParserF (k :: * → *) (a :: *) where
  PureF    :: a → ParserF k a
  SatisfyF :: (Char → Bool) → ParserF k Char
  EmptyF :: ParserF k a
  TryF     :: k a → ParserF k a
  ApF :: k (a → b) → k a → ParserF k b
  OrF :: k a → k a → ParserF k a

instance IFunctor ParserF where
  imap _ EmptyF = EmptyF
  imap _ (SatisfyF c) = SatisfyF c
  imap _ (PureF x) = PureF x
  imap f (TryF px) = TryF (f px)
  imap f (ApF pf px) = ApF (f pf) (f px)
  imap f (OrF px py) = OrF (f px) (f py)
```

The paper here attempts to hide its usage of Fix and cata by specifying specialised versions of them for Circuit4. Instead, we can just use Fix and cata for clarity.

```
newtype Fix f a = In (f (Fix f) a)
type Parser4 a = Fix ParserF a

cata :: IFunctor f ⇒ (forall i ∘ f a i → a i) → Fix f i → a i
cata alg (In x) = alg (imap (cata alg) x)
```

Now we have all the building blocks needed to start folding our parser DSL. Size can be defined as a fold, which can be determined by the sizeAlg

```
newtype Const a i = Const a
unConst :: Const a i → a
unConst (Const x) = x

sizeAlg :: ParserF (Const Size) a → Const Size a
sizeAlg (PureF _) = Const 1
sizeAlg (SatisfyF _) = Const 1
sizeAlg EmptyF = Const 1
sizeAlg (TryF (Const n)) = Const (n + 1)
sizeAlg (ApF (Const pf) (Const px)) = Const (pf + px + 1)
sizeAlg (OrF (Const px) (Const py)) = Const (px + py + 1)

size4 :: Parser4 a → Size
size4 = unConst ∘ cata sizeAlg
```

## 2 Multi

A common thing with DSLs is to evaluate multiple interpretations. For example, a parser may also want to know the maximum characters it will read. In a deep embedding this is simple, we just provide a second algebra.

```
type MaxMunch = Int

maxMunchAlg :: ParserF (Const MaxMunch) a → Const MaxMunch a
maxMunchAlg (PureF _)              = Const 0
maxMunchAlg EmptyF                 = Const 0
maxMunchAlg (SatisfyF c)           = Const 1
maxMunchAlg (TryF (Const px))      = Const px
maxMunchAlg (ApF (Const pf) (Const px)) = Const (pf + px)
maxMunchAlg (OrF (Const px) (Const py)) = Const (max px py)

maxMunch4 :: Parser4 a → MaxMunch
maxMunch4 = unConst ∘ cata maxMunchAlg
```

But what about a shallow embedding? So far we have only seen parsers be able to have single semantics, so how could we calculate both the maxMunch and size of a parser? It turns out the solution is simple, we can use a pair and calculate both interpretations simulataneously.

```
type Parser5 = (Size, MaxMunch)

size5 :: Parser5 → Size
size5 = fst

maxMunch5 :: Parser5 → Size
maxMunch5 = snd

sizeMaxMunchAlg :: ParserF (Const (Size, MaxMunch)) a → Const (Size, MaxMunch) a
sizeMaxMunchAlg (PureF _)                  = Const (1,     0)
sizeMaxMunchAlg EmptyF                     = Const (1,     0)
sizeMaxMunchAlg (SatisfyF c)               = Const (1,     1)
sizeMaxMunchAlg (TryF (Const (s, mm)))     = Const (s + 1, mm)
sizeMaxMunchAlg (ApF (Const (s, mm)) (Const (s′, mm′))) = Const (s + s′ + 1, mm + mm′)
sizeMaxMunchAlg (OrF  (Const (s, mm)) (Const (s′, mm′))) = Const (s + s′ + 1, max mm mm′)
```

Although this is an algebra, you are able to glean the shallow embedding from this, for example:

```
ap5 pf px = sizeMaxMunchAlg (ApF pf px)
```

## 3 dependent

zygomorphisms

TODO: something in parsley. [?]

## 4 Context Sensitive

Parsers themselves inherently require context sensitive interpretations - what you can parse will decide what you are able to parse in latter points of the parser.

Using the semantics from https://github.com/zenzike/yoda we are able to implement a simple parser using an accumulating fold.

```
newtype Yoda a = Yoda {unYoda :: String → [(a, String)]}
```

− > newtype Yoda a = Yoda (String -> [(a, String)]) − > unYoda :: Yoda a -> (String -> [(a, String)]) − > unYoda (Yoda px) = px

```
yodaAlg :: ParserF Yoda a → Yoda a
yodaAlg (PureF x) = Yoda (λts → [(x, ts)])
yodaAlg EmptyF = Yoda (const [])
yodaAlg (SatisfyF c) = Yoda (λcase
  []        → []
  (t : ts′) → [(t, ts′) | c t])
yodaAlg (TryF px) = px
yodaAlg (ApF (Yoda pf) (Yoda px)) = Yoda (λts → [(f x, ts″) | (f, ts′)
  (x, ts″) ← px ts′])
yodaAlg (OrF (Yoda px) (Yoda py)) = Yoda (λts → px ts ++ py ts)

parse :: Parser4 a → (String → [(a, String)])
parse = unYoda ∘ cata yodaAlg

newtype Parsec a = Parsec (String → [String])   -- not correct
```

## 5 Parameterized

Previously we saw how to add multiple types of interpretations to a shallow embedding. We used pairs to allow us to have two interpretations. However, this doesn't extend very well to many more interpretations. Language support starts to fade for larger tuples and it will begin to become messy.

We already know that shallow embeddings are folds, so we could create a shallow embedding that is in terms of a single parameterized interterpretation.

```
newtype Parser7 i = P7 {unP7 :: forall a ∘ (forall j ∘ ParserF a j →
pure7 :: i → Parser7 i
pure7 x = P7 (λh → h (PureF x))
empty7 :: Parser7 a
empty7 = P7 (λh → h EmptyF)
satisfy7 c = P7 (λh → h (SatisfyF c))
try7 :: Parser7 a → Parser7 a
try7 px = P7 (λh → h (TryF (unP7 px h)))
ap7 :: Parser7 (a → b) → Parser7 a → Parser7 b
ap7 pf px = P7 (λh → h (ApF (unP7 pf h) (unP7 px h)))
or7 px py = P7 (λh → h (OrF (unP7 px h) (unP7 py h)))
```

## 6 Implicitly Parameterized

TODO

```
main :: IO ()
main = ⊥
```