An Overview of *Folding Domain-Specific Languages: Deep and Shallow Embeddings*

Riley Evans (re17105)

1 Introduction

This is an overview of the techniques described in the paper *Folding Domain-Specific Languages: Deep and Shallow Embeddings*. The paper demonstrates a series of techniques that can be used when folding Domain Specific Languges. It does so through the use of a simple parallel prefix circuit language [3].

In this overview a small parser combinator language will be used. This language brings one key feature that was not described in the paper: how to apply these techniques to a typed language. Only a minimal functionally complete set of combinators have been included in the language to keep it simple. However, all other combinators usually found in a combinator language can be contructed from this set.

2 Background

2.1 DSLs

A Domain Specific Language (DSL) is a programming language that has a specialised domain or use-case. This differs from a General Purpose Language (GPL), which can be applied across a larger set of domains. DSLs can be split into two different categories: standalone and embedded. Standalone DSLs require their own compiler and typically have their own syntax. Embedded DSLs use a GPL as a host language, therefore they use the syntax and compiler from that GPL. This means that they are easier to maintain and are often quicker to develop than standalone DSLs.

An embedded DSL can be implemented with two main techniques. Firstly, a deep approach can be taken, this means that terms in the DSL will construct an Abstract Syntax Tree (AST) as a host language datatype. This can then be used to apply optimisations and then evaluated. A second approach is to define the terms as first class components of the language, avoiding the creation of an AST - this is known as a shallow embedding.

2.2 Parsers

A parser is a used to convert a series of tokens into another language. For example converting a string into a Haskell datatype. Parser combinators provide a flexible approach to constructing parsers. Unlike parser generators, a combinator library is embedded within a host language: using combinators to construct the grammar. This makes it a suitable to demonstrate the techniques descriped in this paper for folding the DSL to create parsers.

The langauge is made up of 6 terms, they provide all the essential operations needed in a parser.

```
empty   :: Parser a
pure    :: a                → Parser a
satisfy :: (Char → Bool)    → Parser Char
try     :: Parser a         → Parser a
ap      :: Parser (a → b)   → Parser a → Parser b
or      :: Parser a         → Parser a → Parser a
```

For example, a parser that can parse the characters `'a'` or `'b'` can be defined as,

```
aorb :: Parser Char
aorb = satisfy (≡ 'a') `or` satisfy (≡ 'b')
```

A deep embedding of this parser language is defined in the alegebraic datatype:

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

This can be interpretted by defining a function such as size, that finds the size of the AST used to construct the parser - this can be found in the appendix. size interprets the deep embedding, by folding over the datatype. See the appendix for how to add an interpretation with a shallow embedding.

3 Folds

It is possible to capture the shape of an abstract datatype through the Functor type class. It is possible to capture the shape of an abstract datatype as a Functor. The use of a Functor allows for the specification of where a datatype recurses. There is however one problem, a functor expresing the parser language is required to be typed. Parsers require the type of the tokens being parsed. For example a parser reading tokens that make up an expression could have the type Parser Expr. A functor does not retain the type of the parser, therefore it is required to define a special type class called IFunctor, which is able to maintain the type indicies [4]. A full definition can be found in the appendix.

The shape of Parser_2, can be seen in ParserF where the k a marks the recursive spots.

```
data ParserF (k :: ∗ → ∗) (a :: ∗) where
  EmptyF   :: ParserF k a
  PureF    :: a               → ParserF k a
  SatisfyF :: (Char → Bool)   → ParserF k Char
  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 _ (PureF x)    = PureF    x
  imap _ (SatisfyF c) = SatisfyF c
```

```
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)
```

Fix is used to get the fixed point of the functor. It contains the structure needed to make the datatype recursive. Parser_4 is the fixed point of ParserF.

```
type Parser4 a = Fix ParserF a
```

A mechanism is now required for folding this abstract datatype. This is possible through the use of a catamorphism, which is a generalised way of folding an abstract datatype. Therefore, the cata function can be used - a definition can be found in the appendix.

Now all the building blocks have been defined that allow for the folding of the parser DSL. size can be defined as a fold, which is determined by the sizeAlg. Due to parsers being a typed language, a constant functor is required to preserve the type indicies.

```
type ParserAlg a i = ParserF a i → a i
newtype C a i = C {unConst :: a}
sizeAlg :: ParserAlg (C Size) i
sizeAlg EmptyF       = C 1
sizeAlg (PureF _)    = C 1
sizeAlg (SatisfyF _) = C 1
sizeAlg (TryF (C n)) = C $ n + 1
sizeAlg (ApF (C pf) (C px)) = C $ pf + px + 1
sizeAlg (OrF (C px) (C py)) = C $ px + py + 1
size4 :: Parser4 a → Size
size4 = unConst · cata sizeAlg
```

3.1 Multiple Interpretations

In DSLs it is common to want to evaluate multiple interpretations. For example, a parser may also want to know the maximum characters it will read (maximum munch). In a deep embedding this is simple, a second algebra can be defined.

```
type MM = Int
mmAlg :: ParserAlg (C MM) i
mmAlg (PureF _)      = C 0
mmAlg EmptyF         = C 0
mmAlg (SatisfyF c)   = C 1
mmAlg (TryF (C px))  = C px
mmAlg (ApF (C pf) (C px)) = C $ pf + px
mmAlg (OrF (C px) (C py)) = C $ max px py
maxMunch4 :: Parser4 a → MM
maxMunch4 = unConst · cata mmAlg
```

However, in a shallow embedding it is not as easy. To be able to evaluate both semantics a pair can be used, with both interpretations being evaluated simultaneously. If many semantics are required this can become cumbersome to define.

```
type Parser5 = (Size, MM)
size5 :: Parser5 → Size
size5 = fst
maxMunch5 :: Parser5 → Size
maxMunch5 = snd
```

```
smmAlg :: ParserAlg (C (Size, MM)) a
smmAlg (PureF _)          = C (1,      0)
smmAlg EmptyF             = C (1,      0)
smmAlg (SatisfyF c)       = C (1,      1)
smmAlg (TryF (C (s, mm))) = C (s + 1, mm)
smmAlg (ApF (C (s, mm)) (C (s′, mm’)))
  = C (s + s′ + 1, mm + mm’)
smmAlg (OrF (C (s, mm)) (C (s′, mm’)))
  = C (s + s′ + 1, max mm mm’)
```

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

```
ap5 pf px = smmAlg (ApF pf px)
or5 px py = smmAlg (OrF px py)
```

3.2 Dependent Interpretations

In a more complex parser combinator library that perform optimisations on a deep embedding, it could also be possible that there is a primary fold that depends on other secondary folds on parts of the AST. Folds such as this are named mutumorphisms [1], they can be implemented by tupling the functions in the fold. Willis et al. [5] makes use of a zygomorphism - a special case where the dependency is only one-way - to perform consumption analysis.

3.3 Context-sensitive Interpretations

Parsers themselves inherently require context sensitive interpretations - what can be parsed will depend on what has previously been parsed.

Using the semantics from Wu [6], an implementation can be given for a simple parser using an accumulating fold.

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


yAlg :: ParserAlg Y i
yAlg (PureF x)     = Y $ λts → [(x, ts)]
yAlg EmptyF        = Y $ const []
yAlg (SatisfyF c)  = Y $ λcase
   []        → []
   (t : ts′) → [(t, ts′) | c t]
yAlg (TryF px)     = px
yAlg (ApF (Y pf) (Y px)) = Y $ λts →
   [(f x, ts″) | (f, ts′) ← pf ts
               , (x, ts″) ← px ts′]
yAlg (OrF (Y px) (Y py)) = Y $ λts → px ts ++ py ts
parse :: Parser4 a → (String → [(a, String)])
parse = unYoda · cata yAlg
```

3.4 Parameterized Interpretations

Previously, when defining multiple interpretations in a shallow embedding, a tuple was used. However, this does not extend well when many interpretations are needed. Large tuples tend to lack good language support and will become messy to work with. It would be beneficial if a shallow embedding could have a parameter that gives it the interpretation.

Parser7 allows for this approach, the shallow embedding is made up of first class functions that require an algebra argument. This algebra describes how the shallow embedding should fold the structure.

```
newtype Parser7 i = P7
  {unP7 :: ∀a.(∀j.ParserF a j → a j) → a i}

pure7 :: a → Parser7 a
pure7 x = P7 (λh → h (PureF x))

empty7 :: Parser7 a
empty7 = P7 (λh → h EmptyF)

satisfy7 :: (Char → Bool) → Parser7 Char
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 :: Parser7 a → Parser7 a → Parser7 a
or7 px py = P7 (λh → h (OrF (unP7 px h) (unP7 py h)))
```

One benefit of this approach is that it allows the shallow embedding to be converted to a deep embedding.

```
deep :: Parser7 a → Parser4 a
deep parser = unP7 parser In
```

Simillarly it is possible to convert a deep embedding into a parameterised shallow embedding.

```
shallow :: Parser4 a → Parser7 a
shallow = cata shallowAlg

shallowAlg :: ParserAlg Parser7 i
shallowAlg (PureF x)     = pure7 x
shallowAlg EmptyF        = empty7
shallowAlg (SatisfyF c)  = satisfy7 c
shallowAlg (TryF px)     = try7 px
shallowAlg (ApF pf px)   = ap7 pf px
shallowAlg (OrF px py)   = or7 px py
```

Being able to convert between both types of embedding, demonstrates that deep and parameterised shallow embeddings are inverses of each other.

3.5 Implicitly Parameterised Interpretations

The previous parameterised implementation still required the algebra to be specified. It would be helpful if it could be passed implicitly, if it can be determined from the type of the interpretation. This is possible in Haskell through the use of a type class.

```
class Parser8 parser where
  empty8   :: parser a
  pure8    :: a → parser a
  satisfy8 :: (Char → Bool) → parser Char
  try8     :: parser a            → parser a
  ap8      :: parser (a → b)  → parser a → parser b
  or8      :: parser a            → parser a → parser a


newtype Size8 i = Size {unSize :: Int} deriving Num
```

```
instance Parser8 Size8 where
  empty8      = 1
  pure8 _     = 1
  satisfy8 _  = 1
  try8 px     = px + 1
  ap8 pf px   = coerce pf + coerce px + 1
  or8 px py   = px + py + 1
```

coerce allows for conversion between types that have the same runtime representation. This is the case for Size8 and Int. To be able to reuse the previously defined algebras, a different type class can be defined.

```
class Parser9 parser where
  alg :: ParserAlg parser i

instance Parser9 Size8 where
  alg = coerce · sizeAlg · imap coerce
```

3.6 Modular Interpretations

There may be times when adding extra combinators would be convenient, for example adding a 'many' operator that allows for A modular technique to assembling DSLs would aid this process.

```
data Empty10 (k :: ∗ → ∗) (a :: ∗) where
  Empty10 :: Empty10 k a

data Pure10 (k :: ∗ → ∗) (a :: ∗) where
  Pure10 :: a → Pure10 k a

data Satisfy10 (k :: ∗ → ∗) (a :: ∗) where
  Satisfy10 :: (Char → Bool) → Satisfy10 k Char

data Try10 (k :: ∗ → ∗) (a :: ∗) where
  Try10 :: k a → Try10 k a

data Ap10 (k :: ∗ → ∗) (a :: ∗) where
  Ap10 :: k (a → b) → k a → Ap10 k b

data Or10 (k :: ∗ → ∗) (a :: ∗) where
  Or10 :: k a → k a → Or10 k a


data (f :+: g) (k :: ∗ → ∗) (a :: ∗) where
  L :: f k a → (f :+: g) k a
  R :: g k a → (f :+: g) k a
infixr :+:

instance (IFunctor f, IFunctor g)
   ⇒ IFunctor (f :+: g) where
  imap f (L x) = L (imap f x)
  imap f (R y) = R (imap f y)


type ParserF10 =  Empty10 :+: Pure10 :+: Satisfy10
               :+: Try10   :+: Ap10   :+: Or10
type Parser10 = Fix ParserF10


aorb10 :: Parser10 Char
aorb10 = In (R (R (R (R (R (Or10
  (In (R (R (L (Satisfy10 (≡ 'a'))))))
  (In (R (R (L (Satisfy10 (≡ 'b'))))))))))))


class (IFunctor f, IFunctor g) ⇒ f :≺: g where
  inj :: f k a → g k a
```

```
instance IFunctor f ⇒ f :≺: f where
  inj = id
instance {-# OVERLAPPING #-} (IFunctor f, IFunctor g) ⇒ f :≺: (f :+: g) where
  inj = L
instance (IFunctor f, IFunctor g, IFunctor h, f :≺: g) ⇒ f :≺: (h :+: g) where
  inj = R · inj
```

Smart constructors:

```
empty_10 :: (Empty_10 :≺: f) ⇒ Fix f a
empty_10 = In (inj Empty_10)
pure_10 :: (Pure_10 :≺: f) ⇒ a → Fix f a
pure_10 x = In (inj (Pure_10 x))
satisfy_10 :: (Satisfy_10 :≺: f) ⇒ (Char → Bool) → Fix f Char
satisfy_10 c = In (inj (Satisfy_10 c))
try_10 :: (Try_10 :≺: f) ⇒ Fix f a → Fix f a
try_10 px = In (inj (Try_10 px))
ap_10 :: (Ap_10 :≺: f) ⇒ Fix f (a → b) → Fix f a → Fix f b
ap_10 pf px = In (inj (Ap_10 pf px))
or_10 :: (Or_10 :≺: f) ⇒ Fix f a → Fix f a → Fix f a
or_10 px py = In (inj (Or_10 px py))


aorb'_10 :: (Or_10 :≺: f, Satisfy_10 :≺: f) ⇒ Fix f Char
aorb'_10 = satisfy_10 (≡ 'a') `or_10` satisfy_10 (≡ 'b')


class IFunctor f ⇒ SizeAlg f where
  sizeAlg_10 :: f Size_8 i → Size_8 i
instance (SizeAlg f, SizeAlg g) ⇒ SizeAlg (f :+: g) where
  sizeAlg_10 (L x) = sizeAlg_10 x
  sizeAlg_10 (R y) = sizeAlg_10 y
instance SizeAlg Or_10 where
  sizeAlg_10 (Or_10 px py) = px + py + 1
instance SizeAlg Satisfy_10 where
  sizeAlg_10 (Satisfy_10 _) = 1


size_10 :: SizeAlg f ⇒ Fix f a → Size_8 a
size_10 = cata sizeAlg_10
eval :: Size
eval = coerce (size_10 (aorb'_10 :: (Fix (Or_10 :+: Satisfy_10)) Char))
```

References

[1] M. Fokkinga. 1989. Tupling and Mutumorphisms.

[2] Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-Specific Languages: Deep and Shallow Embeddings. In *International Conference on Functional Programming*. 339–347. https://doi.org/10.1145/2628136.2628138

[3] Ralf Hinze. 2004. An Algebra of Scans. In *Mathematics of Program Construction*, Dexter Kozen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 186–210.

[4] Conor McBride. 2011. Functional pearl: Kleisli arrows of outrageous fortune. *Journal of Functional Programming (accepted for publication)* (2011).

[5] Jamie Willis, Nicolas Wu, and Matthew Pickering. 2020. Staged Selective Parser Combinators. *Proc. ACM Program. Lang.* 4, ICFP, Article 120 (Aug. 2020), 30 pages. https://doi.org/10.1145/3409002

[6] Nicolas Wu. 2018. Yoda: A simple combinator library. https://github.com/zenzike/yoda

A Appendix

```
type Size = Int
size :: Parser_2 a → Size
size Empty_2         = 1
size (Pure_2 _)      = 1
size (Satisfy_2 _)   = 1
size (Try_2 px)      = 1 + size px
size (Ap_2 pf px)    = 1 + size pf + size px
size (Or_2 px py)    = 1 + size px + size py


type Parser_3 a = Int
pure_3 _ = 1
satisfy_3 _ = 1
empty_3 = 1
try_3 px = px + 1
ap_3 pf px = pf + pf + 1
or_3 px py = px + py + 1
size_3 :: Parser_3 a → Size
size_3 = id


class IFunctor f where
  imap :: (∀i.a i → b i) → f a i → f b i
newtype Fix f a = In (f (Fix f) a)
cata :: IFunctor f ⇒ (∀i.f a i → a i) → Fix f i → a i
cata alg (In x) = alg (imap (cata alg) x)


instance IFunctor Empty_10 where
  imap _ Empty_10 = Empty_10
instance IFunctor Pure_10 where
  imap _ (Pure_10 x) = Pure_10 x
instance IFunctor Satisfy_10 where
  imap _ (Satisfy_10 c) = Satisfy_10 c
instance IFunctor Try_10 where
  imap f (Try_10 px) = Try_10 $ f px
instance IFunctor Ap_10 where
  imap f (Ap_10 pf px) = Ap_10 (f pf) (f px)
instance IFunctor Or_10 where
  imap f (Or_10 px py) = Or_10 (f px) (f py)
```