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 os a simple parallel prefic circuit language [2].

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 [3]. 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
```

$$
\begin{aligned}
&imap\ f\ (TryF\ px) &&= TryF\ (f\ px)\\
&imap\ f\ (ApF\ pf\ px) &&= ApF\ \ (f\ pf)\ \ (f\ px)\\
&imap\ f\ (OrF\ px\ py) &&= OrF\ \ (f\ px)\ \ (f\ py)
\end{aligned}
$$

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*.

$$\mathbf{type}\ Parser_4\ a = Fix\ ParserF\ a$$

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.

$$
\begin{aligned}
&\mathbf{newtype}\ C\ a\ i = C\ \{unConst :: a\}\\
&sizeAlg :: ParserF\ (C\ Size)\ a \to C\ Size\ a\\
&sizeAlg\ EmptyF &&= C\ 1\\
&sizeAlg\ (PureF\ \ \_) &&= C\ 1\\
&sizeAlg\ (SatisfyF\ \_) &&= C\ 1\\
&sizeAlg\ (TryF\ (C\ n)) &&= C\ \$\ n + 1\\
&sizeAlg\ (ApF\ (C\ pf)\ (C\ px)) &&= C\ \$\ pf + px + 1\\
&sizeAlg\ (OrF\ (C\ px)\ (C\ py)) &&= C\ \$\ px + py + 1\\
&size_4 :: Parser_4\ a \to Size\\
&size_4 = unConst \circ cata\ sizeAlg
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathbf{type}\ MM = Int\\
&mmAlg :: ParserF\ (C\ MM)\ a \to C\ MM\ a\\
&mmAlg\ (PureF\ \_) &&= C\ 0\\
&mmAlg\ EmptyF &&= C\ 0\\
&mmAlg\ (SatisfyF\ c) &&= C\ 1\\
&mmAlg\ (TryF\ (C\ px)) &&= C\ px\\
&mmAlg\ (ApF\ (C\ pf)\ (C\ px)) &&= C\ \$\ pf + px\\
&mmAlg\ (OrF\ (C\ px)\ (C\ py)) &&= C\ \$\ max\ px\ py\\
&maxMunch_4 :: Parser_4\ a \to MM\\
&maxMunch_4 = unConst \circ cata\ mmAlg
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathbf{type}\ Parser_5 = (Size, MM)\\
&size_5 :: Parser_5 \to Size\\
&size_5 = fst\\
&maxMunch_5 :: Parser_5 \to Size\\
&maxMunch_5 = snd\\
&smmAlg :: ParserF\ (C\ (Size, MM))\ a \to C\ (Size, MM)\ a\\
&smmAlg\ (PureF\ \_) &&= C\ (1, \quad 0)\\
&smmAlg\ EmptyF &&= C\ (1, \quad 0)\\
&smmAlg\ (SatisfyF\ c) &&= C\ (1, \quad 1)\\
&smmAlg\ (TryF\ (C\ (s, mm))) &&= C\ (s + 1, mm)\\
&smmAlg\ (ApF\ (C\ (s, mm))\ (C\ (s', mm')))\\
&\quad = C\ (s + s' + 1, mm + mm')\\
&smmAlg\ (OrF\ \ (C\ (s, mm))\ (C\ (s', mm')))\\
&\quad = C\ (s + s' + 1, max\ mm\ mm')
\end{aligned}
$$

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

$$
\begin{aligned}
&ap_5\ pf\ px = smmAlg\ (ApF\ pf\ px)\\
&or_5\ px\ py = smmAlg\ (OrF\ px\ py)
\end{aligned}
$$

3.2 Dependent Interpretations

zygomorphisms

TODO: something in parsley. [4]

3.3 Context-sensitive Interpretations

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 [5] we are able to implement a simple parser using an accumulating fold.

$$\mathbf{newtype}\ Y\ a = Y\ \{unYoda :: String \to [(a, String)]\}$$

$$
\begin{aligned}
&yAlg :: ParserF\ Y\ a \to Y\ a\\
&yAlg\ (PureF\ x) &&= Y\ \$\ \lambda ts.\ [(x, ts)]\\
&yAlg\ EmptyF &&= Y\ \$\ const\ [\,]\\
&yAlg\ (SatisfyF\ c) &&= Y\ \$\ \lambda\mathbf{case}\\
&\quad [\,] \qquad .\ [\,]\\
&\quad (t : ts') \to [(t, ts') \mid c\ t]\\
&yAlg\ (TryF\ px) &&= px\\
&yAlg\ (ApF\ (Y\ pf)\ (Y\ px)) &&= Y\ \$\ \lambda ts.\\
&\quad [(f\ x, ts'') \mid (f,\ ts') \leftarrow pf\ ts\\
&\qquad\qquad\quad ,\ (x, ts'') \leftarrow px\ ts']\\
&yAlg\ (OrF\ (Y\ px)\ (Y\ py)) &&= Y\ \$\ \lambda ts.\ px\ ts +\!\!+\ py\ ts\\
&parse :: Parser_4\ a \to (String \to [(a, String)])\\
&parse = unYoda \circ cata\ yAlg
\end{aligned}
$$

3.4 Parameterized Interpretations

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.

$$
\begin{aligned}
&\mathbf{newtype}\ Parser_7\ i = P7\\
&\quad \{unP7 :: \forall a.(\forall j.ParserF\ a\ j \to a\ j) \to a\ i\}\\
&pure_7 :: i \to Parser_7\ i\\
&pure_7\ x = P7\ (\lambda h.\ h\ (PureF\ x))\\
&empty_7 :: Parser_7\ a
\end{aligned}
$$

$$
\begin{aligned}
&empty_7 = P7\ (\lambda h.\ h\ EmptyF)\\
&satisfy_7 :: (Char \to Bool) \to Parser_7\ Char\\
&satisfy_7\ c = P7\ (\lambda h.\ h\ (SatisfyF\ c))\\
&try_7 :: Parser_7\ a \to Parser_7\ a\\
&try_7\ px = P7\ (\lambda h.\ h\ (TryF\ (unP7\ px\ h)))\\
&ap_7 :: Parser_7\ (a \to b) \to Parser_7\ a \to Parser_7\ b\\
&ap_7\ pf\ px = P7\ (\lambda h.\ h\ (ApF\ (unP7\ pf\ h)\ (unP7\ px\ h)))\\
&or_7 :: Parser_7\ a \to Parser_7\ a \to Parser_7\ a\\
&or_7\ px\ py = P7\ (\lambda h.\ h\ (OrF\ (unP7\ px\ h)\ (unP7\ py\ h)))
\end{aligned}
$$

3.5 Implicitly Parameterized Interpretations

TODO

3.6 Modular Interpretations

TODO

References

[1] Jeremy Gibbons and Nicolas Wu. "Folding Domain-Specific Languages: Deep and Shallow Embeddings". In: *International Conference on Functional Programming*. Sept. 2014, pp. 339–347. DOI: 10.1145/2628136.2628138. URL: http://www.cs.ox.ac.uk/jeremy.gibbons/publications/embedding.pdf.

[2] Ralf Hinze. "An Algebra of Scans". In: *Mathematics of Program Construction*. Ed. by Dexter Kozen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 186–210. ISBN: 978-3-540-27764-4.

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

[4] Jamie Willis, Nicolas Wu, and Matthew Pickering. "Staged Selective Parser Combinators". In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3409002. URL: https://doi.org/10.1145/3409002.

[5] Nicolas Wu. *Yoda: A simple combinator library*. URL: https://github.com/zenzike/yoda.

4 Appendix

$$
\begin{aligned}
&\mathbf{type}\ Size = Int\\
&size :: Parser_2\ a \to Size\\
&size\ Empty_2 \qquad = 1\\
&size\ (Pure_2\ \_) \quad = 1\\
&size\ (Satisfy_2\ \_) = 1\\
&size\ (Try_2\ px) \quad = 1 + size\ px\\
&size\ (Ap_2\ pf\ px) = 1 + size\ pf + size\ px\\
&size\ (Or_2\ px\ py) = 1 + size\ px + size\ py
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{type}\ Parser3\ a = Int\\
&pure3\ \_ = 1\\
&satisfy3\ \_ = 1\\
&empty3 = 1\\
&try3\ px = px + 1\\
&ap3\ pf\ px = pf + pf + 1\\
&or3\ px\ py = px + py + 1\\
&size3 :: Parser3\ a \to Size\\
&size3 = id
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{class}\ IFunctor\ f\ \mathbf{where}\\
&\quad imap :: (\forall i.a\ i \to b\ i) \to f\ a\ i \to f\ b\ i\\
&\mathbf{newtype}\ Fix\ f\ a = In\ (f\ (Fix\ f)\ a)\\
&cata :: IFunctor\ f \Rightarrow (\forall i.f\ a\ i \to a\ i) \to Fix\ f\ i \to a\ i\\
&cata\ alg\ (In\ x) = alg\ (imap\ (cata\ alg)\ x)
\end{aligned}
$$