Advanced Topics in Programming Languages - Paper Overview

Riley Evans (re17105)

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). This can then be used to apply optimisations and then evaluated. A second approach is to define the terms as their semantics, avoiding the AST. This approach is referred to as a shallow embedding.

2 Parsers

In the paper, a circuit language used to describe the different techniques for folding DSLs. For the purposes of this review a new DSL will be introduced - this is a parser DSL. This langauge is made up of 6 terms, they provide all the essential operations needed in a parser.

$$
\begin{array}{lll}
\mathit{empty} & :: & \mathit{Parser}\ a \\
\mathit{pure} & :: a \to & \mathit{Parser}\ a \\
\mathit{satisfy} & :: (\mathit{Char} \to \mathit{Bool}) \to & \mathit{Parser}\ \mathit{Char} \\
\mathit{try} & :: \mathit{Parser}\ a \to & \mathit{Parser}\ a \\
\mathit{ap} & :: \mathit{Parser}\ (a \to b) \to \mathit{Parser}\ a \to & \mathit{Parser}\ b \\
\mathit{or} & :: \mathit{Parser}\ a \to \quad \mathit{Parser}\ a \to & \mathit{Parser}\ a
\end{array}
$$

For example, a parser that can parse a or b can be defined as,

$$
\begin{array}{l}
\mathit{aorb} :: \mathit{Parser}\ \mathit{Char} \\
\mathit{aorb} = (\mathit{satisfy}\ (\equiv \texttt{'a'}))\ \text{`}\mathit{or}\text{`}\ (\mathit{satisfy}\ (\equiv \texttt{'b'}))
\end{array}
$$

A deep embedding of this parser language is defined as Parser2 in the appendix. A function size can be defined that finds the size of the AST created in the deep embedding

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{type}\ \mathit{Size} = \mathit{Int}} \\
\multicolumn{2}{l}{\mathit{size} :: \mathit{Parser2}\ a \to \mathit{Size}} \\
\mathit{size}\ (\mathit{Empty2}) & = 1 \\
\mathit{size}\ (\mathit{Pure2}\ \_) & = 1 \\
\mathit{size}\ (\mathit{Satisfy2}\ \_) & = 1 \\
\mathit{size}\ (\mathit{Try2}\ \mathit{px}) & = 1 + \mathit{size}\ \mathit{px} \\
\mathit{size}\ (\mathit{Ap2}\ \mathit{pf}\ \mathit{px}) & = 1 + \mathit{size}\ \mathit{pf} + \mathit{size}\ \mathit{px} \\
\mathit{size}\ (\mathit{Or2}\ \mathit{px}\ \mathit{py}) & = 1 + \mathit{size}\ \mathit{px} + \mathit{size}\ \mathit{py}
\end{array}
$$

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

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

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

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

3 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

$$
\begin{array}{l}
\mathbf{class}\ \mathit{IFunctor}\ f\ \mathbf{where} \\
\quad \mathit{imap} :: (\mathit{forall}\ i \circ a\ i \to b\ i) \to f\ a\ i \to f\ b\ i
\end{array}
$$

$$
\begin{array}{l}
\mathbf{data}\ \mathit{ParserF}\ (k :: * \to *)\ (a :: *)\ \mathbf{where} \\
\quad \mathit{PureF} \quad :: a \to \mathit{ParserF}\ k\ a \\
\quad \mathit{SatisfyF} :: (\mathit{Char} \to \mathit{Bool}) \to \mathit{ParserF}\ k\ \mathit{Char} \\
\quad \mathit{EmptyF} :: \mathit{ParserF}\ k\ a \\
\quad \mathit{TryF} \quad :: k\ a \to \mathit{ParserF}\ k\ a \\
\quad \mathit{ApF} :: k\ (a \to b) \to k\ a \to \mathit{ParserF}\ k\ b \\
\quad \mathit{OrF} :: k\ a \to k\ a \to \mathit{ParserF}\ k\ a
\end{array}
$$

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{IFunctor}\ \mathit{ParserF}\ \mathbf{where} \\
\quad \mathit{imap}\ \_\ \mathit{EmptyF} = \mathit{EmptyF} \\
\quad \mathit{imap}\ \_\ (\mathit{SatisfyF}\ c) = \mathit{SatisfyF}\ c \\
\quad \mathit{imap}\ \_\ (\mathit{PureF}\ x) = \mathit{PureF}\ x \\
\quad \mathit{imap}\ f\ (\mathit{TryF}\ \mathit{px}) = \mathit{TryF}\ (f\ \mathit{px}) \\
\quad \mathit{imap}\ f\ (\mathit{ApF}\ \mathit{pf}\ \mathit{px}) = \mathit{ApF}\ (f\ \mathit{pf})\ (f\ \mathit{px}) \\
\quad \mathit{imap}\ f\ (\mathit{OrF}\ \mathit{px}\ \mathit{py}) = \mathit{OrF}\ (f\ \mathit{px})\ (f\ \mathit{py})
\end{array}
$$

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.

$$
\begin{array}{l}
\mathbf{newtype}\ \mathit{Fix}\ f\ a = \mathit{In}\ (f\ (\mathit{Fix}\ f)\ a) \\
\mathbf{type}\ \mathit{Parser4}\ a = \mathit{Fix}\ \mathit{ParserF}\ a
\end{array}
$$

$$
\begin{array}{l}
\mathit{cata} :: \mathit{IFunctor}\ f \Rightarrow (\mathit{forall}\ i \circ f\ a\ i \to a\ i) \to \mathit{Fix}\ f\ i \to a\ i \\
\mathit{cata}\ \mathit{alg}\ (\mathit{In}\ x) = \mathit{alg}\ (\mathit{imap}\ (\mathit{cata}\ \mathit{alg})\ x)
\end{array}
$$

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

$$
\begin{array}{l}
\mathbf{newtype}\ \mathit{Const}\ a\ i = \mathit{Const}\ a \\
\mathit{unConst} :: \mathit{Const}\ a\ i \to a \\
\mathit{unConst}\ (\mathit{Const}\ x) = x
\end{array}
$$

$$
\begin{array}{l}
\mathit{sizeAlg} :: \mathit{ParserF}\ (\mathit{Const}\ \mathit{Size})\ a \to \mathit{Const}\ \mathit{Size}\ a \\
\mathit{sizeAlg}\ (\mathit{PureF}\ \_) = \mathit{Const}\ 1 \\
\mathit{sizeAlg}\ (\mathit{SatisfyF}\ \_) = \mathit{Const}\ 1 \\
\mathit{sizeAlg}\ \mathit{EmptyF} = \mathit{Const}\ 1 \\
\mathit{sizeAlg}\ (\mathit{TryF}\ (\mathit{Const}\ n)) = \mathit{Const}\ (n + 1) \\
\mathit{sizeAlg}\ (\mathit{ApF}\ (\mathit{Const}\ \mathit{pf})\ (\mathit{Const}\ \mathit{px})) = \mathit{Const}\ (\mathit{pf} + \mathit{px} + 1) \\
\mathit{sizeAlg}\ (\mathit{OrF}\ (\mathit{Const}\ \mathit{px})\ (\mathit{Const}\ \mathit{py})) = \mathit{Const}\ (\mathit{px} + \mathit{py} + 1)
\end{array}
$$

$$
\begin{array}{l}
\mathit{size4} :: \mathit{Parser4}\ a \to \mathit{Size} \\
\mathit{size4} = \mathit{unConst} \circ \mathit{cata}\ \mathit{sizeAlg}
\end{array}
$$

4 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)
```

5 dependent

zygomorphisms

TODO: something in parsley. [?]

6 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
```

7 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)))
```

8 Implicitly Parameterized

TODO

```
main :: IO ()
main = ⊥
```

9 Appendix

```
data Parser__2 :: ∗ → ∗ where
  Pure__2 :: a → Parser__2 a
  Satisfy__2 :: (Char → Bool) → Parser__2 Char
  Empty__2 :: Parser__2 a
  Try__2 :: Parser__2 a → Parser__2 a
  Ap__2 :: Parser__2 (a → b) → Parser__2 a → Parser__2 b
  Or__2 :: Parser__2 a → Parser__2 a → Parser__2 a
```