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 embedded into Haskell 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 constructed 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 an exisiting language as a host, therefore they use the syntax and compiler from that language. 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 known as a shallow embedding, is to define the terms as first class components of the language. An example of this could be a function in Haskell. This approach avoids the creation of an AST.

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 described in this paper for folding the DSL to create parsers.

The language 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 algebraic 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 interpreted by defining a function such as size, which finds the size of the AST used to construct the parser. size interprets the deep embedding, by folding over the datatype. See the appendix A.1 for how to add an interpretation with a shallow embedding.

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

3 Folds

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 expressing 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. Instead a type class called IFunctor will be used, which is able to maintain the type indicies [4]. This can be thought of as a functor transformer: it is able to change the structure of a functor whilst preserving the values inside it.

```
class IFunctor iF where
  imap :: (∀a.f a → g a) → iF f a → iF g a
```

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

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

The IFunctor instance can be found in the appendix A.2. It follows the same structure as a standard Functor instance.

Fix is used to get the fixed point of a functor. It provides the structure needed to allow the datatype to recursive.

```
newtype Fix iF a = In (iF (Fix iF) a)
```

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. The commutative diagram below describes how a *catamorphism* can be defined. Firstly, a layer of Fix is peeled off by removing an In to give iF (Fix iF) a. Then a recursive call is made to fold the structure below in the AST. This results in a item of type iF f a. Finally, an algebra is applied to fold this layer of the datatype, resulting in a item of type f a.

$$\begin{tikzcd}
\mathsf{iF\ (Fix\ iF)\ a} \arrow[r, "\mathsf{imap\ (cata\ alg)}"] \arrow[d, shift left, "\mathsf{In}"] & \mathsf{iF\ f\ a} \arrow[d, "\mathsf{alg}"] \\
\mathsf{Fix\ iF\ a} \arrow[r, "\mathsf{cata\ alg}"] \arrow[u, shift left, "\mathsf{inop}"] & \mathsf{f\ a}
\end{tikzcd}$$

cata is able to fold a Fix iF a and produce an item of type f a. It uses the algebra argument as a specification of how to fold the input datatype.

```
cata :: IFunctor iF ⇒ (∀a.iF f a → f a) → Fix iF a → f a
cata alg (In x) = alg (imap (cata alg) x)
```

Since the resulting type of cata for an IFunctor is f a, this requires the output to be a Functor. One example of this could be Fix ParserF. However, in the case that the datatype would like to be folded into something that is not a functor, or has kind $*$, then additional infrastructure is needed. There are two methods to allow this to take place. A new type could be defined for each output type that has a phantom type parameter. For example:

```
newtype Size′ i = Size′ {unSize :: Size} deriving Num
```

However, this could lead to lots of new type definitions. To avoid this the constant functor can be used. It allows a type with kind $*$ to have kind $* \to *$, in a similar way to how the const function works.

```
newtype C a k = C {unConst :: a}
```

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

```
type ParserAlg f a = ParserF f a → f a

sizeAlg :: ParserAlg (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)

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 number of 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) a
mmAlg EmptyF         = C 0
mmAlg (PureF _)      = 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 EmptyF             = C (1,    0)
smmAlg (PureF _)          = 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’)
```

It is possible to take an alegbra and convert it into a shallow embedding. This is possible by setting the shallow embedding equal to the result of the algebra, with the corresponding constructor from the deep embedding, 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 *zygomorphisms* [1] - a special case of a *mutomorphism* - they can be implemented by tupling the functions in the fold. Willis et al. [6] makes use of a *zygomorphism* 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.

A parser can be implemented with an accumulating fold. An accumulating fold forms series of nested functions, that collapse to give a final value once the base case has been applied. A simple example of an accumulating fold could be, implementing foldl in terms of foldr.

$$
\begin{array}{l}
\mathsf{foldl} :: (\mathsf{b} \to \mathsf{a} \to \mathsf{b}) \to \mathsf{b} \to [\mathsf{a}] \to \mathsf{b} \\
\mathsf{foldl\ f\ b\ as} = \mathsf{foldr}\ (\lambda \mathsf{a\ g\ x} \to \mathsf{g\ (f\ x\ a)})\ \mathsf{id\ as\ b}
\end{array}
$$

A simple example of foldl can be considered.

$$
\begin{array}{l}
\quad \mathsf{foldl}\ (+)\ 0\ [1,2] \\
\equiv \\
\quad (\lambda \mathsf{x} \to (\lambda \mathsf{x} \to \mathsf{id}\ (\mathsf{x}+2))\ (\mathsf{x}+1))\ 0
\end{array}
$$

Yoda [7] is a simple non-deterministic parser combinator library. The combinators are used to produce a function of type $\mathsf{parser} :: \mathsf{String} \to [(\mathsf{a}, \mathsf{String})]$. Similarities can be drawn from the previous example, the combinators form the first part of the example where a function is constructed of lambdas. The base case 0 of the fold is then passed into the constructed function, this similar to how a string is passed into the parsing function. The accumulating fold for Yoda, is implemented by yAlg.

```
newtype Y a = Y {unYoda :: String → [(a, String)]}
yAlg :: ParserAlg Y a
yAlg EmptyF      = Y (const [])
yAlg (PureF x)   = Y (λts → [(x, ts)])
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 be parameterised to take an interpretation in the form of an algebra.

Parser_7 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. The full definitions can be found in Appendix A.3.

```
newtype Parser7 a = P7
  {unP7 :: ∀f.(∀a.ParserAlg f a) → f a}
satisfy7 :: (Char → Bool) → Parser7 Char
satisfy7 c = P7 (λh → h (SatisfyF c))
or7 :: Parser7 a → Parser7 a → Parser7 a
or7 px py = P7 (λh → h (OrF (unP7 px h) (unP7 py h)))
```

Then, for example, to find the size of a parser:

```
size7 :: Parser7 a → Size
size7 p = unConst (unP7 p sizeAlg)
```

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. Where shallowAlg is setting each constructor to the corresponding shallow function - this can be seen in Appendix A.4.

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

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

3.5 Implicitly Parameterized 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
```

```
instance Parser8 Size′ where
  empty8      = Size′ 1
  pure8 _     = Size′ 1
  satisfy8 _  = Size′ 1
  try8 px     = Size′ (unSize px + 1)
  ap8 pf px   = Size′ (unSize pf + unSize px + 1)
  or8 px py   = Size′ (unSize px + unSize py + 1)
```

To be able to reuse the previously defined algebras, a different type class can be defined.

$$\begin{array}{l}\textbf{class}\ \mathsf{Parser_9\ parser}\ \textbf{where}\\ \quad \mathsf{alg :: ParserAlg\ parser\ a}\\ \textbf{instance}\ \mathsf{Parser_9\ Size'}\ \textbf{where}\\ \quad \mathsf{alg = Size' \cdot unConst \cdot sizeAlg \cdot imap\ (C \cdot unSize)}\end{array}$$

3.6 Modular Interpretations

There may be times when adding extra combinators would be convenient. For example, adding a 'string' operator. A modular technique for assembling DSLs would aid this process. This approach is described in Data types à la carte [5]. An :+: operator can be defined to specify a choice between constructors.

$$\begin{array}{l}\textbf{data}\ \mathsf{(iF :+: iG)\ (f :: * \to *)\ (a :: *)}\ \textbf{where}\\ \quad \mathsf{L :: iF\ f\ a \to (iF :+: iG)\ f\ a}\\ \quad \mathsf{R :: iG\ f\ a \to (iF :+: iG)\ f\ a}\\ \textbf{infixr}\ \mathsf{:+:}\\ \textbf{instance}\ \mathsf{(IFunctor\ iF, IFunctor\ iG)}\\ \qquad\qquad \mathsf{\Rightarrow IFunctor\ (iF :+: iG)}\ \textbf{where}\\ \quad \mathsf{imap\ f\ (L\ x) = L\ (imap\ f\ x)}\\ \quad \mathsf{imap\ f\ (R\ y) = R\ (imap\ f\ y)}\end{array}$$

For each constructor that is required the datatype and IFunctor instance can be defined.

$$\begin{array}{l}\textbf{data}\ \mathsf{Ap_{10}\ (f :: * \to *)\ (a :: *)}\ \textbf{where}\\ \quad \mathsf{Ap_{10} :: f\ (a \to b) \to f\ a \to Ap_{10}\ f\ b}\\ \textbf{instance}\ \mathsf{IFunctor\ Ap_{10}}\ \textbf{where}\\ \quad \mathsf{imap\ f\ (Ap_{10}\ pf\ px) = Ap_{10}\ (f\ pf)\ (f\ px)}\end{array}$$

All datatypes and instances can be found in Appendix A.5

The datatypes are now summed together to form a single $\mathsf{ParserF_{10}}$ type.

$$\begin{array}{l}\textbf{type}\ \mathsf{ParserF_{10} = Empty_{10} :+: Pure_{10} :+: Satisfy_{10}}\\ \qquad\qquad\qquad \mathsf{:+: Try_{10} :+: Ap_{10} :+: Or_{10}}\\ \textbf{type}\ \mathsf{Parser_{10} = Fix\ ParserF_{10}}\end{array}$$

There is however, one problem with this approach: there is now a mess of L and R's. This makes this approach inconvenient to use.

$$\begin{array}{l}\mathsf{aorb_{10} :: Parser_{10}\ Char}\\ \mathsf{aorb_{10} = In\ (R\ (R\ (R\ (R\ (R\ (Or_{10}}\\ \quad \mathsf{(In\ (R\ (R\ (L\ (Satisfy_{10}\ (\equiv \texttt{'a'}))))))}\\ \quad \mathsf{(In\ (R\ (R\ (L\ (Satisfy_{10}\ (\equiv \texttt{'b'}))))))))))))}\end{array}$$

Data types à la carte [5], however, describes a technique that allows for the injection of these L's and R's. The notion of subtypes between functors, can be specified using the :≺: operator.

$$\begin{array}{l}\textbf{class}\ \mathsf{(IFunctor\ iF, IFunctor\ iG) \Rightarrow iF :\prec: iG}\ \textbf{where}\\ \quad \mathsf{inj :: iF\ f\ a \to iG\ f\ a}\\ \textbf{instance}\ \mathsf{IFunctor\ iF \Rightarrow iF :\prec: iF}\ \textbf{where}\\ \quad \mathsf{inj = id}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \ \{\text{-\# OVERLAPPING \#-}\}\\ \qquad\qquad \mathsf{(IFunctor\ iF, IFunctor\ iG)}\\ \qquad\qquad \mathsf{\Rightarrow iF :\prec: (iF :+: iG)}\ \textbf{where}\\ \quad \mathsf{inj = L}\\ \textbf{instance}\ \mathsf{(IFunctor\ iF, IFunctor\ iG,}\\ \qquad\qquad \mathsf{IFunctor\ iH, iF :\prec: iG)}\\ \qquad\qquad \mathsf{\Rightarrow iF :\prec: (iH :+: iG)}\ \textbf{where}\\ \quad \mathsf{inj = R \cdot inj}\end{array}$$

Smart constructors are defined that allow for the L's and R's to be injected. Two examples are given, the other smart constructors can be found in Appendix A.6

$$\begin{array}{l}\mathsf{satisfy_{10} :: (Satisfy_{10} :\prec: iF) \Rightarrow (Char \to Bool)}\\ \quad \mathsf{\to Fix\ iF\ Char}\\ \mathsf{satisfy_{10}\ c = In\ (inj\ (Satisfy_{10}\ c))}\\ \mathsf{or_{10} :: (Or_{10} :\prec: iF) \Rightarrow Fix\ iF\ a \to Fix\ iF\ a \to Fix\ iF\ a}\\ \mathsf{or_{10}\ px\ py = In\ (inj\ (Or_{10}\ px\ py))}\end{array}$$

Now the smart constructors can be used to form an expression $\mathsf{aorb'_{10}}$. The type contraints on this expression allow for f to be flexible, so long as $\mathsf{Or_{10}}$ and $\mathsf{Satisfy_{10}}$ are subtypes of the functor f.

$$\begin{array}{l}\mathsf{aorb'_{10} :: (Or_{10} :\prec: iF, Satisfy_{10} :\prec: iF) \Rightarrow Fix\ iF\ Char}\\ \mathsf{aorb'_{10} = satisfy_{10}\ (\equiv \texttt{'a'})\ \grave{}or_{10}\grave{}\ satisfy_{10}\ (\equiv \texttt{'b'})}\end{array}$$

To be able to give an interpretation an algebra is still required. Simillarly to the constructors the algebra needs to be modularized. A type class can be defined that provides the algebra to fold each constructor.

$$\begin{array}{l}\textbf{class}\ \mathsf{IFunctor\ iF \Rightarrow SizeAlg\ iF}\ \textbf{where}\\ \quad \mathsf{sizeAlg_{10} :: iF\ Size'\ a \to Size'\ a}\\ \textbf{instance}\ \mathsf{(SizeAlg\ iF, SizeAlg\ iG)}\\ \qquad\qquad \mathsf{\Rightarrow SizeAlg\ (iF :+: iG)}\ \textbf{where}\\ \quad \mathsf{sizeAlg_{10}\ (L\ x) = sizeAlg_{10}\ x}\\ \quad \mathsf{sizeAlg_{10}\ (R\ y) = sizeAlg_{10}\ y}\end{array}$$

One benefit to this approach is that if an interpretation is only needed for parsers that use $\mathsf{or_{10}}$ and $\mathsf{satisfy_{10}}$, then only those instances need to be defined. Take calculating the size of the parser $\mathsf{aorb'_{10}}$, only the two instances need to be defined to do so.

$$\begin{array}{l}\textbf{instance}\ \mathsf{SizeAlg\ Or_{10}}\ \textbf{where}\\ \quad \mathsf{sizeAlg_{10}\ (Or_{10}\ px\ py) = px + py + 1}\\ \textbf{instance}\ \mathsf{SizeAlg\ Satisfy_{10}}\ \textbf{where}\\ \quad \mathsf{sizeAlg_{10}\ (Satisfy_{10}\ \_) = 1}\end{array}$$

$$\begin{array}{l}\mathsf{size_{10} :: SizeAlg\ iF \Rightarrow Fix\ iF\ a \to Size'\ a}\\ \mathsf{size_{10} = cata\ sizeAlg_{10}}\\ \mathsf{eval :: Size}\\ \mathsf{eval = unSize\ (size_{10}\ (aorb'_{10} :: (}\\ \qquad\qquad\qquad \mathsf{Fix\ (Or_{10} :+: Satisfy_{10}))\ Char))}\end{array}$$

The type of $\mathsf{aorb'_{10}}$ is required to be specified. This is so that the compiler knows the top level functor being used and the constructors included in it. There could possibly an error in the paper here, as it states that only instances for $\mathsf{Fan_{11}}$ and $\mathsf{Stretch_{11}}$ need to be defined. However, it sets the type for stretchfan to be $\mathsf{Circuit_{11}}$. This requires that the $\mathsf{WidthAlg_{11}}$ instances be defined for all constructors in $\mathsf{Circuit_{11}}$. To rectify this type error stretchfan should be given the type $\mathsf{stretchfan :: Fix\ (Fan_{11} :+: Stretch_{11})}$.

4 Conclusion

This overview has walked through the techniques described in the paper and applied them to a previously unconsidered case - typed DSLs. This now allows the techniques in the paper to be taken advantage of in typed DSLs such as parser combinators. This is done through the use of an IFunctor and special instances of Fix and cata.

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] Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

[6] 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

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

A Appendix

A.1 Shallow Embedding of size

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

A.2 IFunctor instance of ParserF

```
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)
```

A.3 Constructors for a Parameterized Shallow Embedding

```
empty_7 :: Parser_7 a
empty_7 = P_7 (λh → h EmptyF)
pure_7 :: a → Parser_7 a
pure_7 x = P_7 (λh → h (PureF x))
try_7 :: Parser_7 a → Parser_7 a
try_7 px = P_7 (λh → h (TryF (unP_7 px h)))
ap_7 :: Parser_7 (a → b) → Parser_7 a → Parser_7 b
ap_7 pf px = P_7 (λh → h (ApF (unP_7 pf h) (unP_7 px h)))
```

A.4 Converting from Deep to a Parameterized Shallow Embedding

```
shallowAlg :: ParserAlg Parser_7 a
shallowAlg EmptyF       = empty_7
shallowAlg (PureF x)    = pure_7 x
shallowAlg (SatisfyF c) = satisfy_7 c
shallowAlg (TryF px)    = try_7 px
shallowAlg (ApF pf px)  = ap_7 pf px
shallowAlg (OrF px py)  = or_7 px py
```

A.5 Datatypes and IFunctor Instances for a Modular Interpretation

```
data Empty_10 (f :: ∗ → ∗) (a :: ∗) where
  Empty_10 :: Empty_10 f a
data Pure_10 (f :: ∗ → ∗) (a :: ∗) where
  Pure_10 :: a → Pure_10 f a
data Satisfy_10 (f :: ∗ → ∗) (a :: ∗) where
  Satisfy_10 :: (Char → Bool) → Satisfy_10 f Char
data Try_10 (f :: ∗ → ∗) (a :: ∗) where
  Try_10 :: f a → Try_10 f a
data Or_10 (f :: ∗ → ∗) (a :: ∗) where
  Or_10 :: f a → f a → Or_10 f a
```

```
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 Or_10 where
  imap f (Or_10 px py) = Or_10 (f px) (f py)
```

A.6 Smart Constructors to Inject L and R's

$$\mathsf{empty}_{10} :: (\mathsf{Empty}_{10} :\prec: \mathsf{iF}) \Rightarrow \mathsf{Fix\ iF\ a}$$
$$\mathsf{empty}_{10} = \mathsf{In}\ (\mathsf{inj\ Empty}_{10})$$
$$\mathsf{pure}_{10} :: (\mathsf{Pure}_{10} :\prec: \mathsf{iF}) \Rightarrow \mathsf{a} \to \mathsf{Fix\ iF\ a}$$
$$\mathsf{pure}_{10}\ \mathsf{x} = \mathsf{In}\ (\mathsf{inj}\ (\mathsf{Pure}_{10}\ \mathsf{x}))$$
$$\mathsf{try}_{10} :: (\mathsf{Try}_{10} :\prec: \mathsf{iF}) \Rightarrow \mathsf{Fix\ iF\ a} \to \mathsf{Fix\ iF\ a}$$
$$\mathsf{try}_{10}\ \mathsf{px} = \mathsf{In}\ (\mathsf{inj}\ (\mathsf{Try}_{10}\ \mathsf{px}))$$
$$\mathsf{ap}_{10} :: (\mathsf{Ap}_{10} :\prec: \mathsf{iF}) \Rightarrow \mathsf{Fix\ iF}\ (\mathsf{a} \to \mathsf{b}) \to \mathsf{Fix\ iF\ a} \to \mathsf{Fix\ iF\ b}$$
$$\mathsf{ap}_{10}\ \mathsf{pf\ px} = \mathsf{In}\ (\mathsf{inj}\ (\mathsf{Ap}_{10}\ \mathsf{pf\ px}))$$