agdarsec -- Total Parser Combinators

Guillaume Allais

April 4, 2017

Brouwer Seminar
Radboud University Nijmegen

Parser Combinators: An Embedded DSL

- Host language's tooling & libraries
- Help from the coverage & type checkers
- Higher-Order Parser (+ fixpoints)

Parser Combinators: The Basics

```haskell
type Parser a =
    String              -- input string
    -> [(a, String)] -- possible values + leftover

parse :: Parser a -> String -> Maybe a
parse p s = case filter (null . snd) (p s) of
   [(a,[])] -> Just a
   _        -> Nothing
```

Our First Combinators

```
anyChar :: Parser Char
anyChar []       = []
anyChar (c : s) = [(c, s)]

guard :: (a -> Bool) -> Parser a -> Parser a
guard f p s = filter (f . fst) (p s)

digit :: Parser Char
digit = guard (`elem` "0123456789") anyChar
```

Our First Combinators

```
anyChar :: Parser Char
anyChar []      = []
anyChar (c : s) = [(c, s)]

guard' :: (a -> Maybe b) -> Parser a -> Parser b
guard' f p s = catMaybes $ fmap check (p s) where
  check (a, s) = fmap (,s) (f a)

digit' :: Parser Int
digit' = guard' (readMaybe . (:[])) anyChar
```

Structures

```haskell
instance Functor Parser where (...)
instance Applicative Parser where (...)
instance Monad Parser where (...)

instance Alternative Parser where
  empty :: Parser a
  empty s = []

  (<|>) :: Parser a -> Parser a -> Parser a
  (p <|> q) s = p s ++ q s
```

Fixpoints

```
some :: Parser a -> Parser [a]
some p = (:) <$> p <*> many p

many :: Parser a -> Parser [a]
many p = some p <|> pure []
```

Non Total

```
data Expr = Literal Int | Plus Expr Expr

char :: Char -> Parser Char
char c = guard (c ==) anyChar

int :: Parser Int
int = convert <$> some digit' where
  convert ds = (...)

expr :: Parser Expr
expr = Literal <$> int
  <|> Plus <$> expr <* char '+' <*> expr
```

Non Total

```
data Expr = Literal Int | Plus Expr Expr

char :: Char -> Parser Char
char c = guard (c ==) anyChar

int :: Parser Int
int = convert <$> some digit' where
  convert ds = (...)
```

```
expr :: Parser Expr
expr = Literal <$> int
  <|> Plus <$> expr <* char '+' <*> expr
```

Non Total

```
data Expr = Literal Int | Plus Expr Expr

char :: Char -> Parser Char
char c = guard (c ==) anyChar

int :: Parser Int
int = convert <$> some digit' where
  convert ds = (...)

expr' :: Parser Expr
expr' = base <|> Plus <$> base <* char '+' <*> expr'
  where base = Literal <$> int
             <|> char '(' *> expr' <* char ')'
```

Sized Lists

```
|_|≡_ : {A : Set} → List A → ℕ → Set
| []     |≡ zero  = ⊤
| x :: xs |≡ suc n = | xs |≡ n
| []     |≡ suc n = ⊥
| x :: xs |≡ zero  = ⊥

record |List_|≡_ (A : Set) (n : ℕ) : Set where
 constructor mkSizedList
 field list   : List A
       .proof : | list |≡ n
```

A Successful Computation

```
record Success (Tok : Set)
               (A : Set) (n : ℕ) : Set where
 constructor _^_,_
 field
  value      : A
  {size}     : ℕ
  .small     : size < n
  leftovers  : ∣List Tok ∣≡ size
```

The Parser Type

```
record Parser (Tok : Set) (M : Set → Set)
              (A : Set) (n : ℕ) : Set where
  constructor mkParser
  field runParser : ∀ {m} → .(m ≤ n) →
        ∣List Tok ∣≡ m → M (Success Tok A m)
```

Combinators for Indexed Sets

```
_→_ : (A B : I → Set) → (I → Set)
(A → B) n = A n → B n

_⊕_ : (A B : I → Set) → (I → Set)
(A ⊕ B) n = A n ⊎ B n

_⊗_ : (A B : I → Set) → (I → Set)
(A ⊗ B) n = A n × B n

[_] : (A : I → Set) → Set
[ A ] = ∀ {n} → A n
```

Combinator for Guarded Recursion

```
record □_ (A : ℕ → Set) (n : ℕ) : Set where
  constructor mkBox
  field call : ∀ {m} → .(m < n) → A m

map : [ A → B ] → [ □ A → □ B ]
app : [ □ (A → B) → (□ A → □ B) ]

duplicate : [ □ A → □ □ A ]
extract   : [ □ A ] → [ A ]

fix  : ∀ A → [ □ A → A ] → [ A ]
loeb : [ □ (□ A → A) → □ A ]
```

Precise Types for Parser Combinators

```
anyTok : [ Parser Tok M Tok ]
guardM : (A → Maybe B) →
         [ Parser Tok M A → Parser Tok M B ]

_<$>_ : (A → B) →
        [ Parser Tok M A → Parser Tok M B ]

return : [ Parser Tok M A → □ Parser Tok M A ]

_<*>_ : [ Parser Tok M (A → B) → Parser Tok M A
        → Parser Tok M B ]

_<|>_ : [ Parser Tok M A → Parser Tok M A
        → Parser Tok M A ]
```

Bind, Or Not, Whatever Floats Your Boat

```
_&?>>=_ : [ Parser Tok M A
          → (const A → □ Parser Tok M B)
          → Parser Tok M (A × Maybe B) ]
```

Reminder: Non Total

```
data Expr = Literal Int | Plus Expr Expr

char :: Char -> Parser Char
char c = guard (c ==) anyChar

int :: Parser Int
int = convert <$> some digit' where
  convert ds = (...)
```

```
expr :: Parser Expr
expr = Literal <$> int
  <|> Plus <$> expr <* char '+' <*> expr
```

Reminder: Non Total

```
data Expr = Literal Int | Plus Expr Expr

char :: Char -> Parser Char
char c = guard (c ==) anyChar

int :: Parser Int
int = convert <$> some digit' where
  convert ds = (...)
```

```
expr :: Parser Expr
expr = fix $ \ rec -> Literal <$> int
  <|> Plus <$> rec <* char '+' <*> rec
```

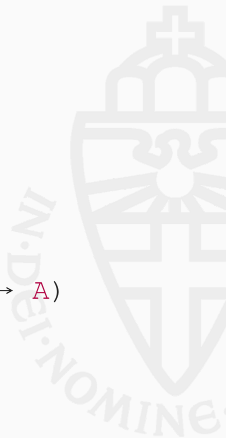
Combinator for Guarded Recursion

```
expr :: Parser Expr
expr = fix $ \ rec -> Literal <$> int
  <|> Plus <$> rec <* char '+' <*> rec
```

```
fix    : ∀ A → [ □ A → A ] → [ A ]

_<$>_ : (A → B) →
        [ Parser Tok M A → Parser Tok M B ]
_<|>_ : [ Parser Tok M A → Parser Tok M A
        → Parser Tok M A ]

_<&_  : [ Parser Tok M A → □ Parser Tok M B
        → Parser Tok M A ]
_<&>_ : [ Parser Tok M A → □ Parser Tok M B
        → Parser Tok M (A × B) ]
```

We can still recover the fixes

```
hchainl : [ Parser Tok M A
          → □ Parser Tok M (A → B → A)
          → □ Parser Tok M B
          → Parser Tok M A ]

chain(l/r)1 : [ Parser Tok M A
              → □ Parser Tok M (A → A → A)
              → Parser Tok M A ]
```



And Safely Implement a Parser for Expr

```
data Expr : Set where
  Var             : Char → Expr
  Lit             : ℕ → Expr
  Add Sub Mul Div : Expr → Expr → Expr

expr : [ Parser Char Maybe Expr ]
expr = fix (Parser Char Maybe Expr) $ λ rec →
 let var    = Var <$> alpha
     lit    = Lit <$> decimal
     addop  = Add <$ char '+' <|> Sub <$ char '-'
     mulop  = Mul <$ char '*' <|> Div <$ char '/'
     factor = parens rec <|> var <|> lit
     term   = chainl1 factor $ return mulop
     expr   = chainl1 term   $ return addop
 in expr
```

Thanks for Your Attention

https://github.com/gallais/agdarsec