Functional Parser of Markdown Language Based on Monad Combining and Monoidal Source Stream Representation

Georgiy Lukyanov(✉) and Artem Pelenitsin

Southern Federal University, Rostov-on-Don, Russia
georgiylukyanov@gmail.com

Abstract. The main goal of this work is to develop flexible and expressive methods of parsers construction based on modern techniques of structuring of effectful computations. We compare two approaches to describing effectful computations: monad transformers and extensible effects in respect to construction of parser combinator libraries. We develop two parser combinator libraries: one based on monad transformers and another on top of extensible effects, and `Markdown-to-HTML` translator with LaTeX blocks based on first library.

Keywords: Functional programming · Parser combinators
Computational effects

1 Introduction

A parser is a necessary part of wide range of software systems: from web browsers to compilers. Parsers may be automatically generated or hand-written. Like any software, parsers can carry implementation errors. One of possible methods of development of robust and correct-by-design software is using a programming language with a rich type system. Modern functional programming languages such as `Haskell` offer facilities of lightweight program verification using strict static typing discipline.

Functional programming is a programming paradigm which treats program as a computation of some mathematical function. A functional style of parser construction requires to represent parser as a function from input stream to some abstract syntactic tree. It is convenient to allow parsers to consume input stream partially. It is also necessary to have a method of incorrect or ambiguous input control. There are several methods for that, for instance user may be notified about parse error, or failure may be replaced by a list of successes [11]. With all mentioned requirements, a `Haskell` type for parser may look like following.

```
type Parser a = String -> [(a,String)]
```


V. Itsykson et al. (Eds.): TMPA 2017, CCIS 779, pp. 90–101, 2018.
https://doi.org/10.1007/978-3-319-71734-0_8

Types like `Parser a` may be treated as computations with a side effect. To extend expressiveness and convenience of parser construction set of side effects may be adjusted. Modern functional programming offer a several approaches to side effect control. **The object of research** in this work is construction of parsers using statically type functional programming languages. And a particular **subject** is methods of side effects control and their applications to parser construction.

The main goal of this work is to develop flexible and expressive method of parser construction based on modern approaches to computations effects combining. To achieve this goal, following tasks are in need to be solved:

1. Develop a parser combinator library based on monad transformers
2. Develop a parser combinator library based on extensible effects
3. Develop a parser of subset of Markdown enriched with LaTeX-blocks and `HTML` code generator.

As a starting point for development of parser combinator libraries, results of paper [8] are used. To describe computations with multiple side effects a concepts of monad transformers [10] and, as an alternative, extensible effects [9] was used. To increase flexibility of libraries and build input stream polymorphic parsers, a special kind of monoid presented in [7] is used.

2 Overview of Approaches to Combining Computational Effects

Modern typed functional languages such as Haskell, PureScript, Idris, etc. divide computations in pure and impure, enforcing statical guarantees on what computation is permitted to do and what it's not: perform IO operations, maintain mutable state, access configuration, throw exception, etc. In the wild, most computations have to carry several side-effects, thus an efficient and expressive technique of combining of effects has to be developed. This work considers to approaches: monad transformers and extensible effects and tries to compare them in terms of convenience for a programmer.

2.1 Monad Transformers

Paper [10] describes a concept of monad transformer — a building block for types describing computations with multiple side effects. Every transformer is a building block, describing one effect: mutable state, configuration, exceptions, etc. Transformers are put on top of a base monad to form a *monad stack*. Consider an example of a function in monad combining effects of mutable state and configuration:

```
adder :: StateT String (Reader Int) Int
adder = do
  str <- get
```

```
  num <- ask
  return $ num + read str

adder' :: (MonadState String m, MonadReader Int m) => m Int
adder' = ...
```

Here `adder` and `adder'` describe same computation, but first functions is bounded to specific monad stack and second just restricts effects that stack ought to provide.

One characteristic of monad stack is that order of monads is statically encoded in type, so there is no runtime control of effect interaction. Second problem of monad transformers is need to write a lot of boilerplate typeclass instances, that is, to add new effect, every possible combinations of newly added effect with existing ones must be covered with instances to provide automatic lifting, thus $\mathcal{O}(n^2)$ instances must be written, where n is a number of monad transformers provided by library.

```
class Monad m => MonadNew a m where
  action1 :: m a
  action2 :: m ()

instance MonadNew m => MonadNew (ExceptT e m) where
  action1 = lift action1
  action2 = lift action2

instance MonadNew m => MonadNew (IdentityT m) where
  action1 = lift action1
  action2 = lift action2

  ...
```

Monad transformers also doesn't provide a way to express computations that produce several homogeneous effects, e.g. two `State` effects without losing automated lifting.

One of alternative approaches that solves some problems of effect combining is Extensible Effects.

2.2 Extensible Effects

Paper [9] present extensible effects — an alternative to monad transformers approach to typing computations with several interacting side effects.

A main idea of extensible effects is in analogy between effectful computations and client-server communication. An expression that is about to introduce some side effect: perform IO, throw an exception or something else like that, must first

make a *request* to some global authority which is in charge of system resources to handle this side effect. Request describes an effectful action that need to be done and a continuation that must be executed after action is performed.

In early variants of libraries similar to extensible effects, authority that manages requests was a separate concept, like an operating system kernel, or `IO`-actions handler of `GHC` runtime. This manager possessed all the system resources (files, memory, etc.): it has been considering every request and making a decision if it should be fulfilled or rejected. This external effect interpreter had great power, but lacked flexibility.

More flexibility and modularity may be introduced with concept of algebraic effects and effects handlers [6], that inspired extensible effects. Thus, some major points of extensible effects:

- Effects handlers are parts of users program: somehow analogous to exception handlers. Every handler is authorized to manage effects of some part of program and produce effects by itself, which are going to be taken care of by some other handler.
- Effect typing system that tracks a type-level collection of effects active for every computation. For collection here stands a notion of `Open Union` — a type-indexed coproduct of functors. Action of every handler affects the type: handled effect is excluded from collection. Therefore, it could be statically checked that all effects are handled.
- Extensible effects exploits a notion of free monad to build an effectful DSLs. An instance of `Monad` typeclass provides programmer with set of familiar `Haskell` techniques such as `do`-notation and applicative programming.

One of huge advantages of extensible effects comparing to monad transformers is absence of need in boilerplate typeclass instance declaration to perform lifting between layers. And there is more: extensible effects permit computations with several similar effects without losing possibility of automatic lifting. Consider an example of function with to readable environmental constants:

```
adder :: ( Member (Reader Int) r
         , Member (Reader String) r) => Eff r Int
adder = do
  num <- ask
  str <- ask
  return $ num + read str
```

Besides, extensible effects doesn't enforce order of effects combination statically as monad transformers stack does, thus giving a precise control of effects interactions in runtime. Next listing contains a computations and two handlers: first one doesn't preserve state in case of failure and returns `Nothing`, but second one does and returns `(0,Nothing)`.

```
countdown :: ( Member Fail r
             , Member (State Int) r) => Eff r ()
countdown = do
  state <- get
  if state == (0 :: Int) then die
  else put (state - 1) >> countdown

runCountdown1 n = run $ runFail $ runState (n :: Int) $ countdown

runCountdown2 n = run $ runState (n :: Int) $ runFail $ countdown
```

2.3 Resume

Both approaches have their pros and cons. Conceptually, extensible effects are more progressive and flexible methods of effect control. But monad transformers are undoubtedly more mature and reality checked approach.

3 Methods of Parser Construction

Consider a simple type to represent a parser.

```
type Parser a = String -> [(a,String)]
```

In this representation, parser is a function, taking input stream and returning a list of possible valid variants of analysis in conjunction with corresponding input stream remains. Empty list of result stands for completely unsuccessful attempt of parsing, whereas multiple results mean ambiguity.

Types similar to `Parser a` may be treated as effectful computation. In this particular example, effect of non-determinism is exploited to express ambiguity of parsing. To represent computations with effects a concept of `Monad` is used in `Haskell` programming language. Comprehensive information about properties of parsers like one presented above may be found in paper [8].

To extend capabilities and improve convenience of syntactic analysers, set of effects of parser could be expanded: it is handy to run parsers in a configurable environment or introduce logging. In this section two approaches to combination of computational effects will be considered: monad transformers and extensible effects.

3.1 Parser as a Monad Transformer Stack

Monad transformer is a concept which lets to enrich a given monad with a property of other monad. Multiple monad transformers may be combined together to form monad stack, that is, a monad possessing all properties of it's components.

Papers [8] proposes a way of decomposition of parser type into stack of two monads: state and list, where the last one provides effect of non-determinism. Thus, type for parser takes a following form.

```
type Parser a = StateT String [] a
```

Parser combinator library developed in this work also uses two-layer monad stack.

```
newtype Parser t a = Parser (
    StateT (ParserState t) (Either (ErrorReport t)) a
  ) deriving ( Functor, Applicative, Monad
             , MonadState (ParserState t)
             , MonadError (ErrorReport t)
             )
```

This representation of a parsers also is parametrised with type of input stream. Types `ParserState` and `ErrorReport` are algebraic data types for representing parser's state and possible analysis errors respectively.

The most low-level primitive which serves as a basis for all parser combinators is a parser that consumes a single item from input stream.

```
item :: TM.TextualMonoid t => Parser t Char
item = do
  state  <- get
  let s = TM.splitCharacterPrefix . remainder $ state
  case s of
    Nothing -> throwError (EmptyRemainder "item",state)
    Just (c,rest) -> do
      let (c,rest) = fromJust s
      put (ParserState {position = updatePos (position state) c
                        , remainder = rest})
      return c
```

More advanced parsers from developed library: conditional consumer and given string consumer.

```
sat :: TM.TextualMonoid t => (Char -> Bool) -> Parser t Char
sat p = do
  state <- get
  x <- item `overrideError` (EmptyRemainder "sat")
  if p x then return x else
    throwError (UnsatisfiedPredicate "general",state)


string :: TM.TextualMonoid t => String -> Parser t String
string s = do
  state <- get
```

```
  (mapM char s) `overrideError`
    (UnsatisfiedPredicate ("string" ++ s))
```

To actually perform parsing, it's necessary to implement a function that runs a computation. It's need to be pointed out, that order of effect handling is statically encoded in type of monad stack.

```
parse :: TM.TextualMonoid t =>
  Parser t a -> t -> Either (ErrorReport t) (a,ParserState t)
parse (Parser p) s =
  runStateT p (ParserState {remainder = s, position = initPos})
    where initPos = (1,1)
```

Overall, a concept of monad transformers has a considerable convenience in programming due to its maturity and popularity. However, as it was discussed in Sect. 2, this approach lacks flexibility, doesn't allow stacks with several homogeneous effects (for instance, multiple `StateT` transformers) without losing automatic lifting (`lift`) and requires boilerplate typeclass instance declaration.

Next, different method of monadic parser combinators will be considered: one based on extensible effects — an alternative framework of construction of effectful computation.

3.2 Parsers Based on Extensible Effects

Extensible effects, presented in paper [9], are an alternative to monad transformers approach to effectful computation description.

An idea behind extensible effects, in a nutshell, is all about analogy between client-server interaction and computational effects. Commands of code is about to produce some side-effect such as IO, exception, etc. have to send a *request* for handling this effect to a special authority — an effect manager. Request describes an action that should be performed alongside with a continuation.

Consider basic primitive of the library — function that consumes a single item of input stream.

```
item :: ( Member Fail r
        , Member (State String) r) => Eff r Char
item = do
  s <- get
  case s of [] -> die
            (x:xs) -> put xs >> return x
```

Type annotation of this function declares effects performed by this function: fallible computation and presence of state. Let us take a closer look on its type annotation. Constraint `Member Fail r` points out that set of effects `r` must

contain effect `Fail`, whereas type of return value `Eff r Char` tells that function `item` yields value of type `Char` and may perform effects from set `r`.

Generally, from syntactic point of view, declaration of combinators based on extensible effects is similar to regular monadic code. This is achieved by type `Eff r a` having an instance of `Monad` typeclass. `Eff r a` is a free monad constructed on top of functor `r` which is a open union of effects. As long as `Eff r a` is a monad, regular monadic do-notation and applicative style become available.

```
sat :: ( Member Fail r
       , Member (State String) r) => (Char -> Bool) -> Eff r Char
sat p = do
  (s :: String) <- get
  x <- item
  if p x then return x else (put s >> die)
```

Extensible effects, in contrast to monad transformers, allow to set an order of effect handling just before running computation. Thus, same computation may produce different behaviour, controlled by order of application of handlers. For instance, in next listing types of handlers `parse` and `parse'` are different because `parse` handles `Fail` after `State` and yields pair of last occurred state and possibly missing result of parsing, i.e. saves last state with no respect to success of parsing. Conversely, `parse'` handles `State` first and doesn't return any state in case of unsuccessful parsing.

```
parse :: Eff (Fail :> (State s :> Void)) a -> s -> (s, Maybe a)
parse p inp = run . runState inp . runFail $ p

parse' :: Eff (State s :> (Fail :> Void)) w -> s -> Maybe (s, w)
parse' p inp = run . runFail . runState inp $ p
```

4 Design of Markdown Parser

`Markdown` is a lightweight language, widely used for small-scale writing. It comes in handy when regular markup languages such as `HTML` and LaTeX are considered an overkill. `Markdown` is popular in IT community, for instance it is extensively used on source code repositories hosting web sites, like `GitHub` [1].

4.1 Markdown Syntax

In contrast with `HTML` or `XML`, `Markdown` doesn't have a standard description. However, informal but comprehensive description of syntax exists [2]. There are also several enhanced versions, such as, for example, `GitHub Flavoured Markdown`.

In this work a subset of `Markdown` syntax is considered, specifically headers, paragraphs, unordered lists and block quotes. In addition, source code may include LaTeX-blocks with formulae.

4.2 Parser

`Haskell` programming language is know for its rich type system. It provides facilities of algebraic data types (ADTs), that could be exploited to conveniently express structure of abstract syntax tree (AST). Every `Document` is a list of blocks. Now, `Block` is a sum type, which means that each of its data constructors represents some `Markdown`-block.

```
type Document = [Block]

data Block = Blank
           | Header (Int,Line)
           | Paragraph [Line]
           | UnorderedList [Line]
           | BlockQuote [Line]
  deriving (Show,Eq)

data Line = Empty | NonEmpty [Inline]
  deriving (Show,Eq)

data Inline = Plain String
            | Bold String
            | Italic String
            | Monospace String
  deriving (Show,Eq)
```

Let's take a closer look at types from previous listing. `Block` is either empty block, or header, or paragraph, or unordered list, or block quote. Most blocks is essentially a list of lines. Every line is a collection of inline elements that are treated differently based on its style.

Next listing contains parsers for line and inline elements, parsers `bold`, `italic` and `plain` are similar to `monospace` and are omitted for the sake of briefness.

```
line :: TM.TextualMonoid t => Parser t Line
line = emptyLine `mplus` nonEmptyLine

emptyLine :: TM.TextualMonoid t => Parser t Line
emptyLine = many (sat wspaceOrTab) >> char '\n' >> return Empty

nonEmptyLine :: TM.TextualMonoid t => Parser t Line
nonEmptyLine = do
  many (sat wspaceOrTab)
  l <- sepby1 (bold <|> italic <|>
               plain <|> monospace) (many (char ' '))
  many (sat wspaceOrTab) >> char '\n'
```

```
  return . NonEmpty $ l

monospace :: TM.TextualMonoid t => Parser t Inline
monospace = do
  txt <- bracket (char '`') sentence (char '`')
  p   <- many punctuation
  return . Monospace $ txt ++ p
```

Implementation of `Markdown` parsers heavily relies on base of repetition combinators.

```
many :: Parser t a -> Parser t [a]

sepby :: Parser t a -> Parser t b -> Parser t [a]

bracket :: Parser t a -> Parser t b -> Parser t c -> Parser t b
```

1. `many` parses a list of tokens which satisfy its argument.
2. `sepby` parses a sequence of tokens which satisfy its first argument and separated by tokens which satisfy second one.
3. `bracket` parses tokens which satisfies its third argument and enclosed by tokens which satisfy first and third one respectively.

Being able to correctly parse both lines and inline elements, it's time to get to block parsers. Next listing contains parser for header. Parsers for the rest of blocks may be constructed in a similar way.

```
header :: TM.TextualMonoid t => Parser t Block
header = do
  hashes <- token (some (char '#'))
  text <- nonEmptyLine
  return $ Header (length hashes,text)
```

`Markdown` language is also used for making notes during lectures and talks, building documentation, and preparing assignments. Therefore, LaTeX blocks seem as a helpful enhancement of a language. There almost no additional work to be done here: it's needed to recognize a LaTeX block and leave its contents unmodified, so could be later treated properly by code generator.

```
  blockMath :: TM.TextualMonoid t => Parser t Block
  blockMath =
    (bracket (string "$$") (some (sat (/= '$')))
             (string "$$")) >>=
    return . Paragraph . (:[]) . NonEmpty . (:[]) . Plain .
      (\x -> "$$" ++ x ++ "$$")
```

Function doc presents top-level parser for Markdown-document as a list of blocks.

```
doc :: TM.TextualMonoid t => Parser t Document
doc = many block
  where block = blank <|> header <|> paragraph <|>
                unorderdList <|> blockquote <|> blockMath
```

4.3 HTML Generation

Having an AST, code in any markup language could be generated. In this work, HTML has been chosen as a target language. One advantage of HTML is possibility of use of JavaScript-libraries, such as MathJax [3] to render LaTeX blocks.

Code generation process follows structure of abstract syntactic tree: function serialize generated code for list of blocs and collapses result to a single string. Every block type is handled by separate pattern matching clause of genBlock function. Equally for lines elements and function genLine.

Next listing displays simplified code generators: handlers for some items are omitted for compactness.

```
serialize :: Document -> String
serialize = concatMap genBlock

genBlock :: Block -> String
genBlock Blank = "\n"
genBlock (Header h) =
  "<h"++s++">" ++ genLine (snd h) ++ "</h"++s++">"++"\n"
    where s = show (fst h)

genLine :: Line -> String
genLine Empty         = ""
genLine (NonEmpty l)  = concatMap ((++ "") . genInline) l

genInline :: Inline -> String
genInline (Plain s) = s
genInline (Monospace s) = "<code>" ++ s ++ "</code>"
```

This is, in brief, the process of Markdown parsing and HTML code generation. Full source codes of parsers and code generator may be found in GitHub repository [4].

5 Conclusion

Following **results** have been achieved:

1. Parser combinator library based on monad transformers that uses special monoids for input stream representation has been developed.
2. Prototype of parser combinator library based on extensible effects has been developed.
3. Basing on library from point one, parser for subset of Markdown enriched with LaTeX blocks has been built, together with `HTML` code generator.

All source codes are available in repositories [4,5].

In addition, Sect. 2 contains a comparative analysis of convenience of programming with two approaches to control of computational effects: monad transformers and extensible effects.

5.1 Possible Applications

Developed libraries may be used for syntax analysis of markup and programming languages.

One possible application of Markdown with LaTeX-blocks parser is a electronic lecture notes system.

5.2 Future Research

Extensible effects is a implementation of abstractions of algebraic effects and effects handlers. These abstractions are in its infancy and it could be useful to perform an approbation of its implementations as a machinery for constructing parser combinators libraries.

References

1. Github. https://github.com/
2. Markdown syntax. http://daringfireball.net/projects/markdown/syntax
3. Mathjax, js-library to render latex in web. https://www.mathjax.org/
4. Lukyanov, G.: Parsing with monad transformers, source code. https://github.com/geo2a/markdown-monparsing
5. Lukyanov, G.: Parsing with extensible effects, source code. https://github.com/geo2a/ext-effects-parsers
6. Pretnar, M., Bauer, A.: Programming with algebraic effects and handlers. arXiv:1203.1539 [cs.PL] (2012)
7. Blaevic, M.: Adding structure to monoids. In: Haskell Symposium 2013, Boston, MA, USA, 23–24 September 2013 (2013)
8. Hutton, G., Meijer, E.: Monadic parser combinators. NOTTCS-TR-96-4 (1996)
9. Swords, C., Kiselyov, O., Sabry, A.: Extensible effects: an alternative to monad transformers. In: Haskell Symposium 2013, Boston, MA, USA, 23–24 September 2013 (2013)
10. Jones, M., Liang, S., Hudak, P.: Monad transformers and modular interpreters. In: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, CA (1995)
11. Wadler, P.: How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 113–128. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-15975-4_33