Generic Parsing Combinators

Pablo E. Martínez López[*]

Dpto. de Ingeniería e Investigaciones Tecnológicas, Universidad Nacional de La Matanza
Florencio Varela 1903, (1754) San Justo, Buenos Aires, República Argentina.
URL: `http://www.unlm.edu.ar/`

Abstract

Parsing combinators are a well known technique to the functional programming community. Several definitions of them were proposed, and each one has its own advantages. From the programmer point of view, this wide range of possibilities implies that she needs to mantain several different modules, with several different namespaces, but all aimed to solve the same problem.

Type classes and constructor classes were introduced in Functional Programming as a mean to provide *overloading*, that is, sharing of names between functions of different types. Several design decisions can be made when implementing class systems. In particular, multi-parameter type classes is a difficult feature to add, and its addition makes sense only if extra expressiveness is achieved with them.

This paper proposes the use of the class system, extended with multi-parameter type classes, to unify the definition of parsing combinators. The advantage is that different solutions can share their interfaces, thus simplifying the programmer's activity. She uses this common interface, and the actual implementation is decided only changing the type signature. Moreover, this use of the class system is an example of the usefulness of multi-parameter type classes.

Keywords: Functional Programming, Parsing Combinators, Monads, Multi-parameter Type Classes

[*]The author is also member of LIFIA, UNLP, C.C.11 Correo Central, (1900) La Plata, Buenos Aires, República Argentina

Generic Parsing Combinators

1 Introduction

Parsing combinators are a well known technique for functional programmers. It is an attempt to solve the parsing problem – obtaining a structure from a elementary description of it, usually in text format – by means of combinators. Combinators are functions that take solutions to sub-problems and combines them into a solution for the problem, thus capturing very well the idea of modular programming. Several implementations of parsing combinators were proposed [Hut93, Fok95, Wad95, Röj95, HM96], each one with its own features, but providing a particular implementation of combinators, usually with different names. In general, each solution starts defining a type to represent parsers, and then gives the definition for atomic parsers and standard combinators in terms of the base type operations. The choice of names for the different parsers and combinators varies from one proposal to another one, but the intention behind them is very similar, when not the same. In [HM96], a monadic generalization in the definition of components is proposed, but this proposal is still based in the provision of a base type for parsers. From the programmer point of view, this wide range of possibilities implies that she needs to mantain several different modules, with several different namespaces, but all aimed to solve the same problem.

Type classes and constructor classes [WB89, Jon95b] were introduced in Functional Programming as a mean to provide *overloading*, that is sharing of names between functions of different types. At present the class system is an experimental feature, and thus different design decisions were made in different languages. For example, Haskell [PH$^+$96] imposes several restrictions to the declaration and use of classes, in order to mantain soundness and decidability – something that Haskell's constructor classes do not have, precisely because of some of these restrictions. On the other hand, Gofer [Jon95a] allows several extensions to the system, but interactions with other aspects of the language have resulted in a type system that is unsound and undecidable. An extensive analysis of design decisions for the class system and a suggestion of a good choice of them is developed in [JJM97]. In particular, multi-parameter type classes is a difficult feature to add, and its addition makes sense only if extra expressiveness is achieved with them, as it is stated in that work. Using the class system of Haskell, it is possible to express, for example, the sharing of the monadic operation names between different monads. But for parsing combinators it is not the case: the expressive power of the Haskell class system is not enough.

This paper proposes the use of the class system, extended with multi-parameter type classes, to unify the definition of parsing combinators. Based on the monadic unification of combinators given in [HM96], a class `Parser` is given to provide the overloading of the primitive parsers and non-monadic combinators. The advantage is that different solutions can share their interfaces, thus simplifying the programmer's activity. She uses this common interface, and the actual implementation is decided only changing the type signature. For each of the parsing combinator proposals, an instance of the `Parser` class is given. Also, the special features of each proposal are captured in subclasses of the `Parser` class.

Moreover, the `Parser` class is a good example of the usefulness of multi-parameter type classes, and thus it shows that this feature is worth to add in the class system.

The rest of this paper is organized as follows. In Sect. 2, the parsing problem is presented with some detail, and following [Wad95] and [HM96], it is stated that parsers are monads. In Sect. 3 the definition of the `Parser` class is given; for each proposal in the literature, an instance of this class and some subclasses that captures special features are also given. Finally, some conclusions and future work are presented in Sect. 4.

As, at present, only Gofer [Jon95a] allows multi-parameter type classes, the Gofer syntax will be used for the definitions and examples.

2 Parsing and Combinators

The parsing problem consists in the construction of an elaborated representation for a structure from an elementary description of it, usually in text format. A classical example of this problem is the *front end* of a compiler for a programming language: given the source code, it must returns the corresponding intermediate code. Usually, the compiler proceeds first with a lexicographical analysis, transforming the source code (list of characters) into a list of reserved words, numerical literals, identifiers, etc. (called *tokens*). The syntactical analysis is then performed using the list of tokens as input. For that reason it is usual to generalize the problem, changing the elementary description to a list of tokens whose type is abstracted away using parametric polimorphism. In the same way, the type of the elaborated representation can take different forms depending on the subject, and thus it is also abstracted away using parametric polimorphism.

The combinator technique provides the solution to a problem by providing three things: a basic type to represent solutions to the problem, a number of basic elements representing solutions to basic instances (also called atomic solutions), and a number of ways to combine solutions to subproblems to get solutions to more complex problems. It is assumed that all the solutions can be represented as combinations of smaller solutions. This technique is common in functional programming, and was used not only for the parsing problem, but also for algebraic representation of pictures [FJ95], for expressing music with functional notation [Hud96], and to construct hypermedia documents [SMLR97]. The functional implementation of the combinator technique provides functions to represent combination of solutions – these functions are called *combinators*.

To implement parsing combinators, a basic type must be chosen. This type will represent parsers, that is functions that transform an elementary description into one or more elaborated representations. In all the implementations considered, this type is fixed at the beggining. In [Hut93], [Wad95], and [HM96], the type of the elementary description is fixed to `String`, but in [Fok95], [Röj95], the more general type `[token]` is considered. In all the cases, the type of the elaborated representation is a polimorphic type variable. The type (`AParser token a`) will represent the choosed type, being `token` the variable for tokens, and `a` the variable for elaborated representations. This type captures not only the functional nature of parsers, but also another characteristics, as the possibility to fail, the possibility to return more than one representation for a given input, reporting positions of errors, etc. In the next section, the type `AParser` will be replaced by a class `Parser` grouping all the possible implementations for this type, and the type of combinators will change accordingly, using a variable `parser` that belongs to that class.

The atomic solutions for the parsing problem are four simple parsers. They are a parser that succeeds without consuming input, a parser that always fails, a parser to recognize and return a single element, and a parser that conditionally recognizes an element. Some other simple parsers can also be considered, but these four ones are common to all the works considered. Different names were used for these basic parsers, but in this work it will be called:

```
return  :: a -> AParser tok a
zero    :: AParser tok a
item    :: AParser tok tok
satisfy :: (tok -> Bool) -> AParser tok tok
```

The argument of the `return` parser is the representation to return when succeeding. The argument of the `satisfy` parser is the condition that the element must satisfy; if the first element does not satisfy it, then the parser is equivalent to `zero`. In [Röj95], an interesting basic parser is presented, `eof`, that succeeds only when there are no more tokens to recognize. It is not presented in any other work, but it can be introduced in every one, and as it is usefull and not expressible as combination of the others, it will be considered a basic one in this work.

The combinators for parsers implements sequencing, alternation and repetition. Also the transformation of the returned representation is expressed as a combinator. As in the case of basic parsers, many names were used, but in this work they will be called:

```
sequ  :: AParser tok a -> AParser tok b -> AParser tok (a,b)
alt   :: AParser tok a -> AParser tok a -> AParser tok a
many  :: AParser tok a -> AParser tok [a]
using :: AParser tok a -> (a -> b) -> AParser tok b
```

The combinators usually are more readable when used infix. For that reason, some of the combinators have an equivalent operator – for example, `(<*>)` for `sequ`, and `(<@)` for `using`.

Using the atomic parsers and the combinators, other parsers and combinators can be defined. For example, a combinator that recognizes a structured surrounded by two fixed elements can be defined using `satisfy` and `sequ`.

```
pack :: tok -> AParser tok tok -> tok
pack t1 p t2 = satisfy (==t1) <*> p <*> satisfy (==t2)
```

The basic parsers and combinators presented does not form a monad. But an operation for bind `(>>=)` can be defined and then the type of parsers becomes a monad, as it is stated, for example, in [HM96].

```
(>>=) :: AParser tok a -> (a->AParser tok b) -> AParser tok b
```

The operation `succeed` is the unit of the monad, and the operation `(>>=)` is the bind operation. Moreover, the operation `fail` is a zero for parsers, and the operation `alt` is a plus, and thus the parsers form a monad with zero and plus.

Using `(>>=)`, some of the basic parsers and combinators can be defined in terms of the others. For example, `satisfy` can be defined using `item`, and vice-versa. Also, `sequ`, `many`, and `using` can be defined using `(>>=)`. The only non-monadic operation that cannot be defined in terms of monadic ones is `item`. Additional combinators can be defined using this combinators. For example, different forms of sequential composition (differing in the results returned), a kind of functional application for parsers, etc.

```
> (*>)      :: Monad m => -> m a -> m b -> m b
> ma >> mb = ma >>= \_ -> mb
> (<*)      :: Monad m => -> m a -> m b -> m a
> ma <* mb = return const $> ma $> mb
> ($>), ap :: Monad m => m (a -> b) -> m a -> m b
> mf $> ma = mf >>= \f -> ma >>= \a -> return (f a)
> ap       = ($>)
```

In all the works studied, the type for parsers is functional, and then the way to use a parser is functional application. But in order to generalize the type of parsers and view it as abstract, there must exist a way to use a parser. That way will be represented by a function `runparser`, that given a parser and a list of tokens will return the elaborated structure, plus the unconsumed tokens.

The exact definition of the basic parsers and combinators (`return`, `zero`, `(>>=)`, `alt`, and `item`), and the type and definition of `runparser` depend completely on the type choosed to represent parsers.

The problem with all the solutions is that different interfaces are needed for different implementations – usually function names are augmented with some prefix or suffix indicating the type. Type classes are an attempt to solve that problem, but for the case of parsers no good class is known at present – perhaps due to the restrictions of the Haskell class system, that prevent a good definition for it. In the next section, a class for parsers is defined, and then, each particular type for parsers can be defined as instances of this class. In such way, there is only one interface that all the implementations share, and thus the programmer's activity is much easier. The solution uses multi-parameter type classes in a complex way, thus showing the usefulness of this feature.

3 Generic Parsing Combinators

Type classes and constructor classes [WB89, Jon95b] were introduced in Functional Programming as a mean to provide *overloading*, that is sharing of names between functions of different types. These sharing is obtained establishing that certain function names are overloaded, and that a type must provide an implementation for the operation in order to share it. The class declaration establishes the names and types for the functions that will be overloaded, and the instance declaration establishes that a certain type belongs to the class, and also provides the implementation for the overloaded functions.

One good example of the use of type classes is the definition of a class for monads, `Monad`, in order to share the operations `return` and `(>>=)`. Using those overloaded operations many overloaded functions may be provided – for example many functions of the previous section – to work with arbitrary monads. But in order to provide a class for parsers in the same way, some extra expressiveness is needed. In the rest of the section, it is assumed the definition of the classes `Monad`, `MonadZero`, and `MonadPlus` that appears in Haskell [PH$^+$96], as well as the definition of the derived combinators for monads presented in Sect. 2.

Multi-parameter type classes is a feature that allows groups of types to share a function name. It is a difficult feature to add to the class system, because it does not interact very well with other ones [JJM97].

The type `Parser` from the previous section is replaced by a type variable `parser` restricted using multi-parameter type classes. Classes are used in order to define a unique interface for parsers. The definition for the class `Parser` establishes that, in order to be a parser, a type must be first a monad with zero and plus (as it is stated in Sect. 2).

```
> class (MonadPlus m, MonadPlus (parser m tok)) =>
>       Parser parser m tok
>  where
>     item    :: parser m tok tok
>     satisfy :: (tok -> Bool) -> parser m tok tok
>     eof     :: parser m tok ()
>
>     item = satisfy (const True)
>     satisfy p = item >>= \tok ->
>                 if (p tok)
>                   then return tok
>                   else zero
```

The constructor `parser` takes three arguments. The first one is a type constructor for a monad with zero and plus, the second one is the type of tokens, and the third one is the type of the results. The first argument is used in the definition of the function `runparser`.

```
> class Parser parser m tok =>
>       RunningParser parser m tok a aux
>  where
>    runparser :: parser m tok aux -> [tok] -> m (a, [tok])
> parse :: RunningParser parser m tok a a =>
           parser m tok a -> [tok] -> m (a, [tok])
> parse = runparser
```

The function `runparser` takes a parser and a list of tokens and returns a monad containing the result and the unconsumed input. The monad parameter is used to change the features added to a parser (for example, nondeterminism, error reporting, etc.). Aditionally, the function `runparser` can change the result from the parser in some way, from type `aux` to type `a`. The function `parse` is provided as a shorthand for the case when `aux` and `a` are the same.

In order to provide instances for the class parser, there must be provided at least the functions `eof` and either `item` or `satisfy` (or both). If one of the latter two is ommited, the default implementation is used.

It is assumed that an instance of class `Parser` satisfies at least the following rules:

```
--   parse (return a) ts = return (a, ts)
--   parse (p >>= f) ts = parse p ts >>= \(a, ts') ->
--                        parse (f a) ts'
--   parse zero ts = zero
--   parse (p `alt` q) ts = parse p ts `alt` parse q ts
--   parse item [] = zero
--   parse item (t:ts) = return (t,ts)
--   parse eof [] = return ((),[])
--   parse eof (t:ts) = zero
```

These laws establishes the way in wich the monad parameter is used to add features to the basic structure.

Before giving any instance of the class `Parser`, an example will be constructed, showing that only the interface is needed to use parser combinators in the construction of parsers. But, in order for the example to run, some instance will eventually be needed. The example must convert a string representing an integer into the corresponding number. The BNF grammar defining the syntax of the numbers is the following.

```
int ::= uint | sign uint          uint ::= dig | dig uint
dig ::= 0 | .. | 9                sign ::= + | -
```

Beggining with this grammar, and using the combinators, a parser is provided for every syntactical cathegory. The parsers recognize the input, and transform the results to cooperate in the construction of the number.

```
> intP, uintP :: Parser parser m Char => parser m Char Int
> intP = (uintP `alt` signP $> uintP) <* eof
> uintP = return (\d ds -> digs2num (d:ds))
>            $> digP
>            $> many digP
> digP :: Parser parser m Char => parser m Char Char
> digP = satisfy isDigit

> signP :: Parser parser m Char => parser m Char (Int->Int)
> signP = symbol '+' <@ const id
>          `alt`
>         symbol '-' <@ const negate

> digs2num = foldl g 0 where g n c = n*10 + dig2num c
> dig2num c = ord c - ord '0'
```

The function `digs2num :: [Char] -> Int` transforms a list of digits into a number (eg. `digs2num "231"` = 231).

In order to use the parser for numbers, one specific instance of the class `Parser` must be choosed. The selection is done using an explicit type signature. For example, taking the type `FokParser` from Sect. 3.1, the following expression can be constructed:

```
       parse (intP :: FokParser Char Int) "231"
```

The value of this expression is the list `[(231,[])]`.

The definitions of parser combinators mentioned in the previous sections can be defined now as instances of the `Parser` class. In some cases minor changes are needed in order to fulfill some requirements of the class system – being the inability to work with type synonyms the most remarkable. In the following subsections are shown the implementations from [Fok95], [HM96], and [Röj95]. In each subsection appears not only the implementation of the common operations, but also extensions and enhancements proposed for the particular implementations.

3.1 Fokker's Implementation

The implementation proposed in [Fok95] – and also the one proposed in [Wad95] – is based on a function that takes a list of tokens and returns a list of pairs – each pair containing a structure described by a sub-list of tokens, and the remaining input. To represent these parsers, the type `AParser t a` is defined as `[t] -> [(a,[t])]`, and all the operations are defined accordingly. In this case, the added feature is the possibility to return more than one structure as output, represented by the list of pairs. The type `FokParser` is an instance of a more general type that, instead of returning lists, returns a monadic value, thus allowing the change of features. This type, `FunParser`, is defined as an algebraic type in order to instantiate it for the class `Parser`. Instantiating the monadic value to other monads, different features can be obtained – in the example, deterministic parsers, `DetFokParser`.

```
> data FunParser m t a = FP ([t] -> m (a, [t]))
> instance Monad m => Monad (FunParser m t) where
>    return a = FP (\ts -> return (a, ts))
>    (FP p1) >>= f = FP (\ts -> p1 ts >>= \(a,ts') ->
>                               let FP p2 = f a
>                                in p2 ts'
>                       )

> instance MonadZero m => MonadZero (FunParser m t) where
>    zero = FP (\ts -> zero)

> instance MonadPlus m => MonadPlus (FunParser m t) where
>    (FP p1) `alt` (FP p2) = FP (\ts -> p1 ts `alt` p2 ts)

> instance MonadPlus m => Parser FunParser m t where
>    item = FP (\ts -> case ts of
>                       []      -> zero
>                       (t:ts') -> return (t,ts')
>              )
>    eof = FP (\ts -> case ts of
>                       []    -> return ((), [])
>                       (_:_) -> zero
>             )

> instance RunningParser FunParser m t a a where
>    runparser (FP p) ts = p ts

> type FokParser t a = FunParser [] t a
> type DetFokParser t a = FunParser Maybe t a
```

The `runparser` function is simply the application of the function representing the parser to the given input.

The class `Parser` does not distinguish between deterministic and non-deterministic parsers. In order to do that, [Fok95] provides an operation that given any parser, returns a deterministic one. This operation is called `first`, and in this work it is provided as an overloaded function, to mantain generality, resulting in a subclass of `Parser`.

```
> class Parser parser m tok =>
>       DetParser parser m tok
>  where
>   first :: parser m tok a -> parser m tok a
>   first = id
> p `detalt` q = first (p `alt` q)

> greedy p = first (many p)
```

The combinators for alternative and repetition are re-defined to behave as deterministic, and a change on the name distinguish these from the (possibly) non-deterministic ones. The operation `first` has the identity function as default definition in order to simplify the instance declaration for parsers that are already deterministic – but it also allows that an instance ignores the deterministic requirement. For example, the types `DetFokParser` and `FokParser` can be defined as instances of `DetParser`.

```
> instance DetParser FunParser Maybe t    -- DetFokParser
> instance DetParser FunParser [] t where -- FokParser
>   first (FP p) = FP (\ts -> case p ts of
>                                []    -> []
>                                (x:_) -> [x]
>                     )
```

Another variation provided by [Fok95] and [Wad95] is a way to force a parser to succeed, even when success is not assured. The programmer is responsible for using this operation correctly – for example in the case of `many` combinator, that never fails. Again, a subclass of `Parser` is provided to capture this enhancement.

```
> class Parser parser m tok =>
>       ForceableParser parser m tok
>  where
>     force :: parser m tok a -> parser m tok a
>     force = id

> lazymany p = force $ (return (:) $> p $> lazymany p)
>                       `alt`
>                      (return [])
```

The types `DetFokParser` and `FokParser` can be defined as instances of `ForceableParser`.

```
> instance ForceableParser FunParser [] t where
>    force (FP p) = FP (\ts -> let x = p ts
>                                  hx = head x
>                                  tx = tail x
>                              in (fst hx, snd hx):tx
>                      )

> instance ForceableParser FunParser Maybe t where
>    force (FP p) = FP (\ts -> let x = p ts
>                                       hx = fromJust x
>                                    in Just (fst hx, snd hx)
>                         )
```

These two force functions use explicit constructors for list and tuples to force the results to a specific pattern – in such a way, the functions never fail.

3.2 Hutton's Implementation

In [HM96] there are several implementation for parser combinators. The most simple one is similar to the one proposed by [Fok95], save that only consider Char as the type for tokens. But the interesting ones are those based on the state monad and reader monad. In order to define that types as instances of the Parser class, some changes are needed. The state and reader monads are re-defined to have the correct number and order of parameters, and the type of parsers is generalized to provide features by means of a monad parameter. The StateMonad class is the one defined in [Jon95b].

```
> data StateM s m t a = SM (s -> m (a,s))

> instance Monad m => Monad (StateM s m t) where
>   return a = SM (\s -> return (a,s))
>   (SM ms1) >>= f = SM (\s -> ms1 s >>= \(a,s1) ->
>                                let (SM ms2) = f a
>                                 in ms2 s1
>                        )

> instance MonadZero m => MonadZero (StateM s m t) where
>   zero = SM (\s -> zero)

> instance MonadPlus m => MonadPlus (StateM s m t) where
>   (SM ms1) `alt` (SM ms2) = SM (\s -> ms1 s `alt` ms2 s)

> instance Monad m => StateMonad (StateM s m t) s where
>   update f = SM (\s -> return (s, f s))
```

The state monad StateM provides the monadic operations for the parser combinators. The type parameter t is used in the definition of the instance for the Parser class. It is needed because the state of the state monad for parsers is the list of tokens, and the type of tokens cannot be recovered. The only operations that have to be defined are the non-monadic ones.

```
> instance StateMonad (StateM [t] m t) [t] =>
>          Parser (StateM [t]) m t
>  where
>    item = update tail >>= \s ->
>           case s of
>             (x:_) -> return x
>             _     -> zero
>    eof  = update id >>= \s ->
>           case s of
>             []    -> return ()
>             (_:_) -> zero

> instance RunningParser (StateM [t]) m t a a where
>   runparser (SM p) s = p s
```

The type of the parsers presented in [HM96] can be defined instantiating the state monad.

```
> type HutParser a = StateM String [] Char a
> type DetHutParser a = StateM String Maybe Char a
```

Instances of the `DetParser` and `ForceableParser` classes can be defined for these parsers.

The reader-monad-based parser combinators defined as instances of class `Parser`, and the special features presented in [HM96] are still under development.

3.3 Röjemo's Implementation

The parser combinators presented in [Röj95] are based on continuations. This decision was taken in order to make optimizations in the memory consumption of some combinators. [Röj95] considers two kind of continuation for parsers: the failure continuation and the success continuation. In this work that types where generalized to work with an arbitrary monad – in Röjemo's work only a specific monad providing error reporting is used.

```
> type GParserFail m t a = m (a,[t])
> type GParserOK m t answer a = a -> [t] -> m (answer,[t])
```

The type of the intermediate structures and the type of the final ones are not necesarily the same, in order to provide the type representing continuations and the bind for monads at the same time. Parsers are then functions from a lists of tokens to the final answer, but represented as continuations.

```
> data GRojParser answer m token a =
>   GRP (GParserOK m token answer a ->
>        GParserFail m token answer ->
>        [token] -> m (answer, [token])
>       )
```

The monadic operations are easily defined, because of the monadic nature of continuations.

```
> instance Monad (GRojParser c m t) where
>   return a = GRP (\ok _ -> ok a)
>   (GRP p1) >>= f =
>        GRP (\ok fail -> p1 (\v -> let GRP p2 = f v
>                                    in p2 ok fail
>                            ) fail
>            )

> instance MonadZero (GRojParser c m t) where
>   zero = GRP (\_ fail -> \_ -> fail)

> instance MonadPlus (GRojParser c m t) where
>   (GRP p1) `alt` (GRP p2) =
>        GRP (\ok fail ->
>               \ts -> p1 ok
>                         (p2 ok fail ts)
>                         ts
>            )
```

Then, to define this type as an instance of the class `Parser`, the non-monadic operations should be provided. In this case, it is easier to provide the implementation of `satisfy`.

```
> instance Parser (GRojParser c) m t where
>   satisfy p = GRP (\ok fail ->
>                     \ts -> case ts of
>                             []      -> fail
>                             (t:ts') -> if (p t)
>                                         then ok t ts'
>                                         else fail
>                   )
>   eof = GRP (\ok fail ->
>               \ts -> case ts of
>                       []    -> ok () []
>                       (_:_) -> fail
>             )
```

The `runparser` function applies the function defining the parser to basic continuations and to the input.

```
> instance RunningParser (GRojParser c) m t c c where
>   runparser (GRP p) ts = p (\a -> \ts -> return (a,ts))
>                            zero
>                            ts
```

Finally, the type is instantiated to get concrete parser combinators. Indeed, in [Röj95] a more involved monad reporting the position of errors is used, but that case is still under development.

```
> type RojParser tok ans a = GRojParser ans Maybe tok a
```

This type can be instantiated for classes `DetParser` and `ForceableParser` just accepting the default definition for the operations.

The interesting feature added by the continuation based implementation of parser is the ability to 'cut' the failure continuation, thus removing a space-leak produced by that continuation. To do that, a new combinator, `cut`, is defined, and some of the old ones are re-defined in order to use the new one. In this work, `cut` is provided as an overloaded function, by means of a subclass of `Parser`.

```
> class Parser parser m t => CutableParser parser m t where
>    cut :: parser m t a -> parser m t a
>    cut = id

> ($>!) :: CutableParser parser m t =>
>          parser m t (a->b) -> parser m t a -> parser m t b
> mf $>! ma = mf >>= \f ->
>              cut (ma >>= \a -> return (f a))

> (<*!) :: CutableParser parser m t =>
>          parser m t a -> parser m t b -> parser m t a
> ma <*! mb = ma >>= \a ->
>              cut (mb >>= \_ -> return a)
```

The types `FunParser` from Sect. 3.1 and `HutParser` from Sect. 3.2 can be defined as instances of this class, accepting the default implementation – the `id` function. The type `GRojParser` can be improved defining a good `cut` combinator.

```
> instance CutableParser (GRojParser c) m t where
>    cut (GRP p) = GRP (\ok _ -> p ok zero)
```

And then, the example of integers can be redefined using the optimized combinators. Only the functions that change the definition are shown – the rest change only the type to reflect the use of `CutableParsers`.

```
> intP, uintP :: CutableParser p m Char => p m Char Int
> intP = (uintP `alt` signP $>! uintP) <* eof
> uintP = return (\d ds -> digs2num (d:ds))
>            $> digP
>            $>! many digP
```

This function are now more efficient when instantiated with `GRojParsers`, but remain the same for the other cases.

4 Conclusions

The construction of a class `Parser` was presented, in order to provide overloading of the parsing combinators. In this way, the programmer can use the combinators without thinking in any particular implementation, and only when using the parsers, a concrete instance type should be provided (by means of explicit type signatures).

The definition uses multi-parameter type classes in a non-trivial way, showing that the expressiveness of this feature makes worthwhile its inclusion in the system.

The work is not finished. There are some features that need more study – beign error reporting the most significant. Also, other implementations of parsing combinators should be expressed as instances of the class, and new features can be designed. Finally, the algebraic structure defined by the proposed laws should be studied.

References

[FJ95] Sigborn Finne and Simon Peyton Jones. Pictures: A simple structured graphics model. In *Proccedings of the Glasgow Workshop on Functional Programming*, Ullapool, 1995.

[Fok95] Jeroen Fokker. Functional parsers. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, LNCS 925*, pages 1–23. Springer-Verlag, May 1995.

[HM96] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical report, University of Nottingham, 1996.

[Hud96] Paul Hudak. Haskore music tutorial. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming, LNCS 1129*, pages 38–67. Springer-Verlag, August 1996.

[Hut93] Graham Hutton. Higher-order functions for parsing. In *Journal of Functional Programming, Vol. 1.* University of Utretch, Cambridge University Press, January 1993.

[JJM97] Simon Peyton Jones, Mark P. Jones, and Erik Meijer. Type classes: An exploration of the design space. URL: http://www.cse.ogi.edu/ simonpj/multi.ps.gz, 1197.

[Jon95a] Mark Jones. Gofer 2.30 release notes. Technical report, Yale University, 1995. http://www.cs.nott.ac.uk:80/Department/Staff/mpj/.

[Jon95b] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, LNCS 925*, pages 97–136. Springer-Verlag, May 1995.

[PH$^+$96] John Peterson, Kevin Hammond, et al. Report on the programming language Haskell, a non–strict, purely functional language. Version 1.3. Technical report, Yale University, May 1996.

[Röj95] Niklas Röjemo. Efficient parsing combinators. In *Garbage Collection, and Memory Efficiency, in Lazy Functional Languages*, Göteborg, Sweden, May 1995. Chalmers University of Technology, Department of Computer Science. Part of the Ph.D. thesis.

[SMLR97] Ignacio Gallego Sagastume, Daniel H. Marcos, Pablo E. Martínez López, and Walter Ariel Risi. Expresando hypermedia en programación funcional. Enviado para su consideración al 2do Congreso Latinoamericano de Programación Funcional, CLaPF'97, 1997.

[Wad95] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, LNCS 925*. Springer-Verlag, May 1995.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polimorphism less ad-hoc. In *16'th Symposium on Principles of Programming Languages*, Austin, Texas, January 1989. ACM Press.