Efficient Combinator Parsers

Pieter Koopman** and Rinus Plasmeijer

Computer Science
Nijmegen University, The Netherlands
pieter@cs.kun.nl, rinus@cs.kun.nl

Abstract. Parser combinators enable the construction of recursive descent parsers in a very clear and simple way. Unfortunately, the resulting parsers have a polynomial complexity and are far too slow for realistic inputs. We show how the speed of these parsers can be improved by one order of magnitude using continuations. These continuations prevents the creation of intermediate data structures. Furthermore, by using an exclusive or-combinator instead of the ordinary or-combinator the complexity for deterministic parsers can be reduced from polynomial to linear. The combination of both improvements turn parser combinators from a beautiful toy to a practically applicable tool which can be used for real world applications. The improved parser combinators remain very easy to use and are still able to handle ambiguous grammars.

1 Introduction

Parser combinators [3, 6, 5, 8] are a beautiful illustration of the use of higher order functions and currying. By using a small set of parser combinators it becomes possible to construct parsers for ambiguous grammars in a very elegant and clear way. The basis of parser combinators is the list of successes method introduced by Wadler [13]. Each parser yields a list of results: all successful parsings of the input. When the parser fails this list is empty. In this way it is very easy to handle ambiguous parsers that define multiple ways to parse a given input.

Despite the elegant formulation and the ability to handle ambiguous grammars, parser combinators are rarely used in practice. For small inputs these parsers work nice and smoothly. For realistically sized inputs the parsers consume extraordinary amounts of time and space due to their polynomial complexity.

In this paper we show that the amounts of time and memory required by the combinators parsers can drastically be reduced by improving the implementation of the parser combinators and providing a little more information about the grammar in the parser that is written. This additional information can reduce the complexity of the parser from polynomial to linear. Although the implementation of the parser combinators becomes more complex, their use in combinator parsers remains as simple as in the original setting.

This paper starts with a short review of classical parser combinators. The proposed improvements are presented hereafter. We use a running parsing example to measure the effect of the improvements.

** Sponsored by STW project NWI.4411

2 Conventional Parser Combinators

There are basically two approaches to construct a parser for a given grammar[1]. The first approach is based upon the construction of a finite state automaton determining the symbols that can be accepted. This approach is used in many parser generators like *yacc* [7, 1, 10], *Happy* [4] and *Ratatosk* [9]. In general the constructed parsers are efficient, but cumbersome to achieve and there might be a serious distance between the automaton accepting input tokens and the original grammar. If we use a generator there is a barrier between parsing and using the parsed items in the rest of the program.

The second approach to achieve a parser is to create a recursive descent parser. Such a parser follows the rules of the grammar directly. In order to ensure termination the grammars should not be left recursive. Parser combinators are a set of higher order functions that are convenient in the construction of recursive descent parsers. The parser combinators provide primitives to recognize symbols and the sequential or alternative composition of parsers.

The advantages of parser combinators are that they are easy to use, elegant and clear. Due to the fact that the obtained parsers directly correspond to the grammar there is no separate parser generator needed. Since parser combinators are ordinary functions they are easy to understand and use. It is easy to extend the set of parser combinators with new handy combinators whenever this is desired. The full power of the functional programming language is available to construct parsers, this implies for instance that it is possible to use second order grammars. Finally, there are no problems to transfer parsed items from the parser to the manipulations functions.

Conventional parser combinators are described at many places in the literature e.g. [3, 6, 5, 8]. Here we follow the approach outlined in [8], using the functional programming language Clean [11]. We restrict ourselves to a small, but complete set of parser combinators to illustrate our improvements.

A `Parser` is a function that takes a list of input symbols as argument and produces a `ParsResult`. A `ParsResult` is the list of successes. Each success is a tuple containing the rest of the list of input symbols and the item found. The types `Parser` and `ParsResult` are parameterized by the type of symbols to be recognized, `s`, and the type of the result, `r`.

```
:: Parser s r :== [s] -> ParsResult s r
:: ParsResult s r :== [([s],r)]
```

In the examples in this paper we will use characters, `Char`, as symbols in the lists of elements to be parsed, but in general they can be of any datatype.

2.1 Basic Parser Combinators

The basic combinator to recognize a given symbol in the input is `symbol`. This parser combinator takes the symbol to be recognized as argument. When the first token in the input is equal to this symbol there is a single success. In all other situations the list of successes is empty indicating that the parser failed. It is of course requested that the equality is defined for the type of symbols to be recognized, this is indicated by the phrase `| == s` in the type definition.

```
symbol :: s -> Parser s s | == s
symbol sym = p
where   p [s:ss] | s==sym = [(ss,sym)]
        p _               = []
```

A related combinator is called `satisfy`. This combinator takes a predicate on the first input symbol as argument. When the first symbol satisfies this predicate the parser combinators succeeds, otherwise it fails.

```
satisfy :: (s->Bool) -> Parser s s
satisfy pred = p
where   p [s:ss] | pred s = [(ss,s)]
        p _               = []
```

To complete the set of basic parser combinators there is a combinator, called `fail`, that always fails (i.e. yields the empty list of successes) and the combinator `yield` that always succeeds with the given item.

```
fail :: Parser s r
fail = \_ -> []

yield :: r -> Parser s r
yield r = \ss -> [(ss,r)]
```

2.2 Combining Parsers

Given two parser it is possible to create new parsers by combining them using parser combinators. For instance, the or-combinator indicates a choice between two parsers. Using the list of successes, the implementation just concatenates the results of both parsers. To enhance readability this parser combinator is defined as an infix operator named `<|>`.

```
(<|>) infixr 4 :: (Parser s r) (Parser s r) -> Parser s r
(<|>) p1 p2 = \ss -> p1 ss ++ p2 ss
```

There are several ways to compose parsers sequentially. Here we restrict us to the so-called *monadic style* where the result of the first parser is given as an argument to the second parser. This and-combinator, named `<&=>`, is defined as:

```
(<&=>) infixr 6 :: (Parser s r) (r -> Parser s t) -> Parser s t
(<&=>) p1 p2 = \ss -> [  tuple
                      \\ (ssRest,result1) <- p1 ss
                      ,  tuple            <- p2 result1 ssRest
                      ]
```

The next parser combinator applies a function to the parsed item. This combinator, called `<@`, is again an infix operator. It is defined in terms of the combinators defined above as the sequential composition of two parsers. The first parser recognize the item. The second parser is the function composition of the item transformation function `f` and the parser that yields the transformed item.

```
(<@) infixl 5 :: (Parser s r) (r->t) -> Parser s t
(<@) p f = p <&=> yield o f
```

Although the introduced set of combinators is sufficient to construct all parsers, it appears to be convenient to introduce additional combinators for repetition. In the BNF description of grammars the notation `p*` means zero or more times `p`. The parser combinator `<*>` applies a parser as often as it can. The results of repeated applications of the parser are accumulated in a list. For the definition we observe that `<*> p` is either `p` followed by `<*> p`, or the empty list.

```
<*> :: (Parser s r) -> Parser s [r]
<*> p = (    p      <&=> \r  ->
             <*> p <@    \rs -> [r:rs])
         <|> yield []
```

There is a related notion, `p+` in BNF, to applies a parser at least once. It is defined as the parser `p` followed by zero or more applications of the parser `p`.

```
<+> :: (Parser s r) -> Parser s [r]
<+> p = p <&=> \r -> <*> p <@ \rs -> [r:rs]
```

2.3 Examples and Measurements

In order to illustrate the use of parser combinators we will show some examples. Our first parser, `aANDbORc`, recognize a character `'a'` followed by a `'b'` or a `'c'`. The result of the parser is a tuple containing the parsed items.

```
aANDbORc :: Parser Char (Char,Char)
aANDbORc = symbol 'a'                    <&=> \x ->
           (symbol 'b' <|> symbol 'c') <@    \y -> (x,y)
```

We illustrate the use of this parser by applying it to some inputs.

The program	yields
`Start = aANDbORc ['abc']`	`[(['c'],('a','b'))]`
`Start = aANDbORc ['acb']`	`[(['b'],('a','c'))]`
`Start = aANDbORc ['cba']`	`[]`

Our next example is the ambiguous parser `alphaORhex`. It recognizes characters from the alphabet and hexadecimal characters.

```
alphaORhex :: Parser Char Char
alphaORhex = alpha <|> hex
```

```
alpha :: Parser Char Char
alpha = satisfy (\c -> isMember c (['a'..'z']++['A'..'Z']))

hex :: Parser Char Char
hex = satisfy (\c -> isMember c (['0'..'9']++['A'..'F']))
```

Again we illustrate the use of the parser by applying it to some inputs.

The program	yields
`Start = alphaORhex ['abc']`	`[(['bc'],'a')]`
`Start = alphaORhex ['ABC']`	`[(['BC'],'A'),(['BC'],'A')]`
`Start = alphaORhex ['123']`	`[(['23'],'1']`

The characters from the range `['A'..'F']` are accepted by the parsers `hex` and `alpha`. Hence `alphaORhex` is ambiguous and yields both results. Here the parser `alphaORhex` recognizes an `alpha`, an `alpha` and a `hex`, and a `hex` respectively.

As a more serious example we turn to a parser that recognize a sentence. A sentence is defined as a sequence of words separated by white space and/or comma's and terminated by a full stop. The result of the parser `sentence` is the list of words in the sentence. Each words is represented by a string.

```
word :: Parser Char String
word = <+> satisfy isAlpha <@ toString

sep :: Parser Char [Char]
sep = <+> (satisfy isSpace <|> symbol ',')

sentence :: Parsers Char [String]
sentence = word                        <&=> \w ->
           <*> (sep <&=> \_ -> word) <&=> \r ->
           symbol '.'                  <@   \_ -> [w:r]
```

The function `isAlpha` from the standard library checks whether a character is an element from the alphabet. Similarly, the test for white space (space, tab, newline ..) is done by `isSpace`. Note that the parser `word` is ambiguous. It does not only find the entire word, but also all non-empty initial parts.

The program	yields

`Start = word ['Hello world']`	`[([' world'],"Hello")`
	`,(['o world'],"Hell")`
	`,(['lo world'],"Hel")`
	`,(['llo world'],"He")`
	`,(['ello world'],"H")`
	`]`

The parser `sentence` looks clear and simple. It is not ambiguous since trailing parts of a word are not recognized as separators and the sentence has to end in a full stop. To investigate its efficiency we apply it to a sentence of 50,000 characters (the size of a reasonable file). As comparison we use a hand written ad-hoc parser which is described in appendix A. This ad-hoc parser uses a state automaton and accumulators to construct the words and is pretty efficient. As a consequence it is harder to determine which syntax is accepted by the parser by looking at the definition. Measurements in this paper are done on a 266 MHz Pentium II, using 10 Mb of heap space and 500 Kb of stack space. The execution time includes the generation of the input list and the consumption of the result.

Parser used	execution time (s)	garbage collect (s)	total time (s)	minimal heap (Kb)
`sentence`	120	345	465	7070
ad-hoc	0.06	0.03	0.09	625

These measurements show that the performance of the parser `sentence` is rather disappointing. It consumes one order of magnitude more memory than the ad-hoc parser and is about 5000 times slower. This disappointing performance is the reason that many people turn to parser generators, like yacc, Happy and Ratatosk mentioned above, forbid ambiguity [12], or use an ad-hoc parser. In this paper we will investigate how we can improve the performance of parsers within the given framework of potential ambiguous parser combinators. In general the time and memory needed by a parser is dependent on the parser and its input. We will use the parser `sentence` as running example.

3 Avoiding Intermediate Data Structures

As a first step to increase the speed of parser combinators we focus on the traditional implementation of these combinators. Although the list of successes method is conceptual nice and clear, it introduces a very large overhead. Each result must be packed in a list of tuples. Usually, this list of tuples is immediately taken apart in the next action of the parser.

Using continuations [2] in the implementation of the parser combinators it is possible to avoid these superfluous intermediate lists and tuples. The parser combinators will handle all continuations. The only thing that changes for the user of these parser combinators is that a parser should be initiated by `begin parser input`, instead of `parser input`.

The parser combinators using continuations can replace the combinators introduced in the previous section completely. Nevertheless, in this paper we use new names for the version of the combinators using continuations such we can identify the different implementations of the combinators.

The type of a continuation parser reflects the fact that there are two continuations. The first continuation handles the item recognized by the parser. This first continuation is very similar to the second argument of the `<&=>` operator. The second continuation contains all remaining results.

```
:: ParserC s r t :== (Success s r t) (NextRes s t) -> Parser s t
:: Success s r t :== r (NextRes s t) -> Parser s t
:: NextRes s t   :== ParsResult s t
```

The type ParserC has three arguments. The first argument, s, denotes the type of symbols parsed. The next argument, r, is the type of the result of parsing. The success continuation transforms this to the type t of the total result. One can regard this success continuation as a function applied by the apply-combinator <@. The type variables s and r correspond to the type variables we have for ordinary parser combinators, the total type variable, t, is new.

The parser combinator that always fails is now defined as the function that yields the next continuation.

```
failC :: ParserC s r t
failC = \succ next ss -> next
```

The parser combinator yieldC, that yields the given element r applies the success continuation to this element, the next continuation and the input.

```
yieldC :: r -> ParserC s r t
yieldC r = \succ next ss -> succ r next ss
```

The parser that recognizes the given symbol s, checks whether the first element of the input stream is equal to this symbol. If it is, the success continuation is applied to the symbol, the next continuation and the rest of the input stream. If the first element is not equal to the given symbol, the parser fails and hence yields the next continuation.

```
symbolC :: s -> ParserC s s t | == s
symbolC sym = p
where  p sc nc [s:ss] | s==sym = sc sym nc ss
       p sc nc _               = nc
```

The new definitions of the and-combinator and or-combinator are trivial. In the and-combinator the second parser is used as success continuation. This parser is applied to the item found by the first parser and the original success continuation.

```
(>&=<) infixr 6 :: (ParserC s u t) (u->ParserC s v t) -> ParserC s v t
(>&=<) p1 p2 = \sc -> p1 (\t -> p2 t sc)
```

In the or-combinator the next continuation is replaced by applying the second parser to the success continuation, the original next continuation and the input.

```
(>|<) infixr 4 :: (ParserC s r t) (ParserC s r t) -> ParserC s r t
(>|<) p1 p2 = \sc nc ss -> p1 sc (p2 sc nc ss) ss
```

In the combinators <@, <*> and <+> we replace the original combinators by the introduced versions. The new operators are called @<, >*< and >+< respectively.

Finally we introduce the function begin that provides the initial continuations for a parser. As success continuation we provide a function that constructs an element in the list of successes. The next continuation is initially the empty list.

```
begin :: (ParserC s t t) -> Parser s t
begin p = p (\r nc ss -> [(ss,r):nc]) []
```

To determine the effect of using continuations we apply a modified parser `sentence` again to the same input. Inside the definition of the parser we have replaced all parser combinators by the corresponding version using continuations.

Parser used	execution time (s)	garbage collect (s)	total time (s)	minimal heap (Kb)
original `sentence`	120	345	465	7070
`sentence` with continuations	9.6	36.6	46.2	5940
ad-hoc	0.06	0.03	0.09	625

From these measurements we see that using continuations improves the execution speed by a factor of ten. Also the minimal amount of memory needed to run this program is reduced a little bit. This effect is caused by omitting the construction of intermediate data structures. On the other hand it is clear that this improvement is not enough, the difference between this parser and the hand written parser is still a factor of 500.

4 Limit Backtracking

Since it looks impossible to speed up the parser combinators much more, we must turn our focus to the behavior of the parsers. As we have seen in the `word ['Hello world']` example in section 2.3 above, the parser `word` is ambiguous. Apart from the whole word, it yields also all initial fragments of this word. This implies that there are n results for a word of n characters. The same holds for the separations of words recognized by the parser `sep`. In the current application we are only interested in the most eager parsing of words and separations. All other results will not yield a successful parsing of a sentence.

This behavior is determined by the or-combinator, `<|>`, used in the definition of the repeat-combinator, `<*>`. The or-combinator yields the results of both alternatives inside the definition of `<*>`. Even when `p <&=> \r -> <*> p <@ \rs -> [r:rs]` succeeds, the or-combinator yields also the empty list. In the current situation we require an exclusive or-combinator that only applies the second parser when the first one fails.

In the plain list of successes approach the definition of the xor-combinator is very direct. The parser `p2` is only applied to the input `ss` if applying `p1` to the input yields the empty list of successes. If applying `p1` to the input produces a non-empty result, this is the result of the entire construct.

```
(<!>) infixr 4 :: (Parser s r) (Parser s r) -> Parser s r
(<!>) p1 p2 = \ss -> case p1 ss of
                        [] -> p2 ss
                        r  -> r
```

Using continuations we can achieve the same effect by replacing the next continuation of the new success-continuation by the original next continuation. This excludes the results of `p2` from the result when `p1` succeeds.

```
(>!<) infixr 4 :: (ParserC s r t) (ParserC s r t) -> ParserC s r t
(>!<) p1 p2 = \sc nc ss -> p1 (\r _ -> sc r nc) (p2 sc nc ss) ss
```

Now we can define eager versions of the repeat-combinators by replacing the ordinary or-combinator in the body of `<*>` by the xor-combinator. For the version using continuations we obtain:

```
>!*< :: (ParserC s r t) -> ParserC s [r] t
>!*< p = (      p       >&=< \r  ->
               >!*< p   @< \rs -> [r:rs])
           >!< yieldC []

>!+< :: (ParserC s r t) -> ParserC s [r] t
>!+< p = p >&=< \r -> >!*< p @< \rs -> [r:rs]
```

It is important to note that the previous optimization, using continuations, does not have any semantically influence on the parser combinators. In the current optimization we turn an ambiguous parser for words into an eager and deterministic parser for words. In the context of parsing a sentence this is allowed, but there might be (rare?) occurrences where the ambiguous nature of the parser word is used. So, it depends on the context whether this optimization can be applied. In general we need the deterministic parser combinators `>!<` and `>!*<` as well as the ambiguous combinators `>|<` and `>*<`. It is advisable to use the deterministic versions whenever the context allows this.

We repeat the measurements of parsing the sentence of 50,000 characters in order to determine the effect of the eager repeat-combinators.

Parser used	execution time (s)	garbage collect (s)	total time (s)	minimal heap (Kb)
original `sentence`	120	345	465	7070
`<!*>` in `word` and `sep`	12.4	21.0	33.4	4925
`<!*>` everywhere	0.30	0.19	0.49	4712
`sentence` with continuations	9.6	36.6	46.2	5940
`>!*<` in `word` and `sep`	1.19	1.12	2.31	1872
`>!*<` everywhere	0.28	0.14	0.43	1740
ad-hoc	0.06	0.03	0.09	625

These measurements show that using xor-combinators instead of ordinary or-combinators make a tremendous differences. The difference in speed between the ad-hoc parser and the best combinator parser is less than a factor 5. Even when the xor-combinator is used only at a limited number of places it has a huge effect. In fact we obtain the effect of an old-fashioned tokenizer when we use the eager repeat combinators in `word` and `sep`.

It is mainly the memory usage that makes it preferable to use the continuation version of the parser combinators when we use xor-combinators and eager repeats everywhere. If there is ambiguity on the parser, the ordinary or-combinator (`<|>`), the version using continuations is an order of magnitude more efficient.

5 Reducing the Memory Requirements

As a last improvement we try to reduce the memory requirements of the parser. As a side effect of the other optimizations introduced above, the memory requirements are reduced by a factor of four. We can improve this by observing that in many applications of the xor-combinator we can decide that the second alternative will not apply before the first alternative does succeed. Currently, the second alternative is not thrown away before the first alternative succeeds.

Consider again the repeat combinator. The version using continuations is:

```
>!*< :: (ParserC s r t) -> ParserC s [r] t
>!*< p = (      p       >&=< \r  ->
                >!*< p   @< \rs -> [r:rs])
           >!< yieldC []
```

As soon as the parser `p` is successfully applied, it is clear that the alternative `yieldC []` will not be used. However, this alternative is kept until the first alternative of `>!<` succeeds. This alternative succeeds after the completion of `>!*< p`, which can be a huge amount of work. By adding primitives to throw the xor-alternatives away the programmer can improve the memory requirements of the parser. In order to do this we introduce a new continuation, `XorCont s t`, containing the xor-alternatives.

```
:: CParser s r t :== (SucCont s r t) (XorCont s t) (AltCont s t)
                     -> Parser s t
:: SucCont s r t :== r (XorCont s t) (AltCont s t) -> Parser s t
:: XorCont s t   :== (AltCont s t) -> ParsResult s t
:: AltCont s t   :== ParsResult s t
```

The xor-continuation is a function that takes the ordinary alternative continuation as argument and yields the remaining parse results. Of course we have to adopt the definitions of all parser combinators to this new continuation. Again, the parsers which use these combinators remain unchanged. In order to distinguish the various versions of the combinators we introduce again new names. The basic parser combinators are:

```
Cfail :: CParser s r t
Cfail = \sc xc ac ss -> xc ac

Cyield :: r -> CParser s r t
Cyield x = \sc -> sc x
```

```
Csymbol :: s -> CParser s s t | == s
Csymbol sym = p
where   p sc xc ac [s:ss] | s==sym = sc sym xc ac ss
        p sc xc ac _               = xc ac
```

In the and-combinator the second parser, `p2`, is inserted in the first continuation, the success continuation `sc`, of the first parser, `p1`.

```
(<<&=>>) infixr 6 :: (CParser s u t) (u->CParser s v t) -> CParser s v t
(<<&=>>) p1 p2 = \sc -> p1 (\t -> p2 t sc)
```

For the xor-combinator we insert the second parser in the xor-continuation, `xc`, of the first parser.

```
(<<!>>) infixr 4 :: (CParser s r t) (CParser s r t) -> CParser s r t
(<<!>>) p1 p2
 = \sc xc ac ss -> p1 (\x xc2 -> sc x xc) (\ac3 -> p2 sc xc ac3 ss) ac ss
```

For the ordinary or-combinator we insert the second parser in the alternative continuation, `ac`, of the first parser.

```
(<<|>>) infixr 4 :: (CParser s r t) (CParser s r t) -> CParser s r t
(<<|>>) p1 p2 = \sc xc ac ss -> p1 sc (\ac2 -> ac2) (p2 sc xc ac ss) ss
```

Again there is a function `Begin` that provides the initial continuations to construct a parse result for the successful parsings.

```
Begin :: (CParser s t t) -> Parser s t
Begin p = p (\x xc ac ss -> [(ss,x):xc ac]) (\ac -> ac) []
```

After the definition of the basic combinators we can define the combinator `cut` that removes xor-alternatives by replacing the corresponding continuation, `xc`, by an identity function.

```
cut :: (CParser s r t) -> CParser s r t
cut p = \sc xc ac -> p sc (\ac -> ac) ac
```

The scope of the cut is all xor-combinators up to the first ordinary or-combinator. Multiple cuts, or cuts in a context without xor-alternatives are harmless. It appears to be convenient to combine the cut-combinator with an and-combinator.

```
(<<!&=>>) infixr 6 :: (CParser s u t) (u->CParser s v t) -> CParser s v t
(<<!&=>>) p1 p2 = \sc -> p1 (\t ac2 -> p2 t sc (\ac -> ac))
```

The semantics of this combinator reads: iff `p1` succeeds, the xor-alternatives are thrown away and we continue parsing with `p2`. If `p1` fails, we turn to the xor- and or-alternatives as usual.

An obvious application of this combinator is the eager repeat-combinator `<!*>` discussed above. When `p` succeeds we can remove the xor-alternative `Cyield []`. We use the operator `<<!&=>>` to achieve this. The definition of the eager repeat-combinator doing this is:

```
<<!*>> :: (CParser s r t) -> CParser s [r] t
<<!*>> p = (     p        <<!&=>> \r  ->
                 <<!*>> p <<@     \rs -> [r:rs])
           <<!>> Cyield []
```

To determine the effects of this optimization we repeat our measurements.

Parser used	execution time (s)	garbage collect (s)	total time (s)	minimal heap (Kb)
`>!*<` everywhere	0.28	0.14	0.43	1740
`<<!*>>` everywhere	0.27	0.11	0.38	937
ad-hoc	0.06	0.03	0.09	625

Although the improvements (12% for speed 46% for minimal memory) are of another range than the previous optimizations we consider them useful. The gain of this optimization depends on the amount of storage that is claimed by the alternative that is thrown away and the moment it is thrown away. For huge alternatives that are thrown away after a very complex (time and memory consuming) first alternative, the gain will be larger.

6 Using Accumulators

Finally, we can consider to implement the repeat-combinator outside the frame work of parser combinators. For the implementation of the combinator `<*>`, it might be useful to employ an accumulator like we have used in the ad-hoc parser. This can look like:

```
<<!!*>> :: (CParser s r t) -> CParser s [r] t
<<!!*>> p = ClistP p []

<<!!+>> :: (CParser s r t) -> CParser s [r] t
<<!!+>> p = p <<&=>> \r -> ClistP p [r]

ClistP :: (CParser s r t) [r] -> CParser s [r] t
ClistP p l = clp l
where  clp l sc xc ac ss
         = p
           (\r xc2 -> clp [r:l] sc (\ac3 -> ac3))
           (\ac4 -> sc (reverse l) xc ac4 ss)
           ac
           ss
```

This new implementation is transparent for the user of the repeat combinators. We measure the value of this modification in our running example.

Parser used	execution time (s)	garbage collect (s)	total time (s)	minimal heap (Kb)
`<<!*>>` everywhere	0.27	0.11	0.38	937
`<<!!*>>` everywhere	0.26	0.08	0.34	887
ad-hoc	0.06	0.03	0.09	625

Using this implementation shows again a small improvement. Due to the use of accumulators and the changed form of recursion, this version also requires less stack space. It appears that the xor-continuation is necessary to implement this efficiently. For the repeat-combinator such an implementation might be used. However, we cannot expect a user of the parser combinators to write such an implementation. Fortunately, the price to be paid by using parser combinators instead of tailor-made functions using accumulators is limited (about 10%).

7 Complexity

Parsers constructed by using conventional parser combinators only work well for small inputs. For larger inputs the execution time increases very rapidly. This is a strong indication that there is a problem with the complexity of the parsers.

It is impossible to make useful statements about all, potentially ambiguous, parsers created by parsers combinators. However, we can investigate the complexity of a specific parser constructed using parser combinators. We will again take the parser for sentences as example. A large part of the discussion is based on the behavior of the repeat-combinators and the recognition of tokens. This part of the discussion will apply to almost any parser written.

As shown in section 2.3 parsing words is ambiguous. The ambiguity is caused by using the standard repeat-combinator `<+>`. Apart from the longest possible word, also all initial fragments of this word are recognized as a word. The recognition of the characters in the word is shared among the various results. But, for each result a spine of the list of characters has to be constructed. These lists of characters are later transformed to strings. For each individual result the parser continues with the remaining part of the input. In our example, parsing the rest of the input will fail on the first input character for all but the most eager result of `word`. Nevertheless, there are n results for parsing a word of length n. For each of these results a list of characters of average length $n/2$ is constructed and the next character after that result is parsed. This indicates that parsing words, and hence parsing a sentence, in this way is $O(n^2)$.

In section 4 we showed how we can get rid of partial words by introducing an xor-combinator. When we construct the repeat-combinator using this xor-combinator, the parser `word` has only a single result. Parsing a word of length n implies parsing n characters and constructing a single list of length n to hold these characters. The parser is now linear, $O(n)$, in the length of the word.

When we also apply the eager repeat-combinator in the parser `sentence`, the entire parser will be linear in the length of the input. This holds for all implementations of the parser combinators. Various implementations of the same combinator result in parsers of the same complexity, only the constants are different. When we apply the ordinary or-combinators the parser `sentence` will be polynomial. Using xor-combinators the parser will be linear.

In order to verify this we measure the time consumed by our best implementation of the combinators (section 5) as function of the length of the input.

length	*or-combinators*			*xor-combinators*			speed-up
	execution time(s)	garbage collect(s)	total time(s)	execution time(s)	garbage collect(s)	total time(s)	total time
1000	0.02	0.01	0.03	0.01	0.00	0.01	3
2000	0.05	0.02	0.07	0.01	0.00	0.01	7
4000	0.12	0.04	0.16	0.02	0.01	0.03	5
10,000	0.45	0.27	0.72	0.04	0.03	0.07	10
20,000	1.46	1.73	3.19	0.09	0.05	0.15	21
40,000	5.54	16.87	22.41	0.21	0.09	0.30	75
100,000	*heap full*			0.46	0.35	0.81	-
200,000	*heap full*			0.96	1.15	2.11	-
400,000	*heap full*			1.92	5.56	7.48	-

These measurements show clearly that the time consumed by the version using the ordinary or-combinator grows polynomially. Also for the xor-combinator version there seems to be some slight super-linear effect, especially in the garbage collection times. To understand this it is important to note that the parser cannot yield any result before the full stop indicating the end of the sentence is encountered. If the full stop is not found, it is not a proper sentence and the parser should yield an empty result. Since parsers of larger inputs have to handle larger data structures, the garbage collector will reclaim less memory. Hence we need a larger number of garbage collections. This explains the super-linear growth of the garbage collection time in the parser using xor-combinators.

The changed complexity caused by using the eager repeat-combinators, like `<!*>`, instead of the usual repeat combinators, like `<*>`, explains the enormous increase of efficiency for the running example of this paper. In fact the efficiency gain of more than a factor 1000 is just an example. If we increase the size of the input, the speed-up figure grows rapidly.

8 Conclusion

Parser combinators are a nice and clear way to define ambiguous recursive descent parser in functional programming languages. Unfortunately, the constructed parsers were not suited for large inputs since they consume enormous amounts of time and memory. This is caused by a polynomial complexity and an implementation that generates much intermediate data structures.

In this paper we have introduced a new implementation technique for parser combinators based on continuations. This improves the speed of the parser by one order of magnitude and reduces the amount of memory required significantly by eliminating intermediate data structures.

By introducing an exclusive or-combinator we are able to reduce the complexity of deterministic parsers from polynomial to linear. This makes it possible to use the parsers constructed by parser combinators for large inputs. However, we stay within the framework of ambiguous parser combinators. This implies that it is always possible to use ambiguity, introduced by the ordinary or-combinator, whenever this is desired.

As a result the example parser in this paper is only four times slower for large inputs than an efficient ad-hoc parser for the same problem. Using the standard parser combinators it was 5000 times slower. This indicates that the proposed changes turn the parser combinators from a nice toy for small problems, into a beautiful and useful tool for real world parsing problems.

9 Related and Future Work

More people have experienced that parser combinators are attractive, but not suited for large inputs. Doaitse Swierstra and his group follow a different approach to tackle the problem. They exclude ambiguity from the beginning and construct parser combinators for deterministic grammars. Our work is more general since it allows ambiguity whenever the user wants it. The user can indicate that the parser is not ambiguous by using the xor-combinator instead of the or-combinator. If the parser is known to be non-ambiguous the complexity decreases from polynomial to linear. In our xor-combinator it is allowed that alternatives overlap partially. In their approach the alternatives should be distinguished at the first symbol. Moreover, our combinator `satisfy` enables the use of a conditional function on the input symbols, while the Swierstra combinators require that all allowed symbols are listed explicitly by a `symbol` combinator. Hence, our combinators are easier to use than Swierstra's unambiguous grammars. Swierstra combinators enables the generation of error messages. It is simple and cheap to extend our combinators in a similar way. In order to compare the speed of both approaches, we implemented their combinators in Clean and applied them to the running example, 2000 simple function alternatives and a huge expression.

parser	*old-combinators*			*new-combinators*			*Swierstra-combinators*		
	ex (s)	gc (s)	tot (s)	ex (s)	gc (s)	tot (s)	ex (s)	gc (s)	tot (s)
sentence	120	345	465	0.26	0.08	0.34	0.55	0.28	0.83
functions	5.5	50.5	56.0	0.43	0.22	0.65	0.98	0.37	1.35
expression	73	104	177	0.36	0.19	0.55	0.51	0.33	0.84

As a conclusion we state that our combinators are more powerful (since they allow ambiguity), easier to use (since they allow alternatives to overlap partially and conditions on input symbols) and seems to be more efficient.

Currently the user has to indicate whether the or-combinator or the xor-combinator has to be used. It would be convenient if the parser uses the xor-combinator whenever this is possible. In principle it is possible to detect this automatically in a number of situations. If the branches of the or-combinators excludes each other (i.e. the parser in unambiguous) the or-combinator can be replaced by the xor-combinator. It looks feasible and seems attractive to construct such an automatic parser optimizer.

References

1. Aho, A. Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques and Tools.* Addison-Wesley. 1986.

2. Appel, A.: *Compiling with Continuations.* Cambridge University press. 1992.
3. Fokker, J.: *Functional Parsers.* In: Advanced functional programming, Båstad summerschool Tutorial text. LNCS 925, 1995, 1 – 23.
4. Gill, A. Marlow, S.: *The parser generator for Haskell.* University of Glasgow. 1995.
5. Hill, S.: *Combinators for parsing expressions* J. Functional Programming **6**(3): 445 – 463, 1996.
6. Hutton, G.: *Higher order functions for parsing.* J. Functional Programming **2**: 323 – 343, 1992.
7. Johnson, S.C.: *Yacc: Yet Another Compiler Compiler.* UNIX on-line documentation. 1978.
8. Koopman, P.: Chapter II.5: *Parser Combinators* of the Clean book under development. The draft is available from http://www.cs.kun.nl/~clean.
9. Mogensen, T.: *Ratatosk: A parser generator and scanner generator for Gofer.* University of Copenhagen (DIKU), 1993.
10. Peyton Jones, S.L.: *Yacc in SASL, an exercise in functional programming.* Software Practice and Experience. **15**(8). 1985. 807 – 820.
11. Plasmeijer, M., van Eekelen M.: *The concurrent Clean language report. Version 1.3.* Nijmegen University, The Netherlands. 1998. http://www.cs.kun.nl/~clean.
12. Swierstra, D., Duponcheel, L.: *Deterministic, Error-Correcting Combinators Parsers.* In: Advanced Functional Programming. LNCS 1129, 1996, 185 – 207.
13. 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 Functional Programming Languages and Computer Architecture, (J.P. Jouannaud, ed.), 1985, LNCS 201, 113–128.

A The ad-hoc Parser

For the sake of completeness we include the ad-hoc parser used in the measurements listed above. The parser has three states called `word`, `sep` and `dot`. In the state `word` the parser accepts characters for the current word. When the character cannot be part of this word we go to that state `sep`. In this state we read separations. When the separation is empty and the input character is not part of the separation the parser enters state `dot`. Otherwise the parser returns to `word`. The parser uses accumulators to construct the current word `w` and the sentence `s`. The accumulators are constructed in reversed order to avoid polynomial complexity. When accumulation is finished the accumulator is reversed to obtain the right order.

```
manParse :: [Char] -> [([Char],[String])]
manParse input = word [] [] input
where word w  s [c:r] | isAlpha c = word [c:w] s r
      word w  s input             = sep [] [toString (reverse w):s] input

      sep l  s [c:r] | isSpace c || c == ',' = sep [c:l] s r
      sep [] s input = dot s input
      sep _  s input = word [] s input

      dot [] input   = []
      dot s  ['.':r] = [(r,reverse s)]
      dot _  _       = []
```