Efficient parsing combinators

Niklas Röjemo
Department of Computer Science,
Chalmers University of Technology,
S-412 96 Göteborg, Sweden
rojemo@cs.chalmers.se

May 31, 1995

Abstract

This paper describes a set of space and time efficient parsing combinators that can return the position of the first illegal token in case of failure. These combinators are intended for non-ambiguous grammars, e.g., grammars for programming languages, where an arbitrary lookahead might be needed.

The arbitrary lookahead is implemented by backtracking. The possibility to backtrack means that the parser must hold on to its input as long as it has alternative rules to try. This prevent garbage collection of the input, increasing the memory demands. However, sometimes it is possible to guarantee that backtracking cannot give a better result than the current rule, e.g., when a key word in the grammar has been recognised.

The novel part in this set of parsing combinators is the possibility to remove the backtracking capability if it is not needed. This has previously only been done for parsing combinators that cannot report the position of errors. The input that is held onto only to allow backtracking is dropped by applying a `cut` combinator. This combinator was introduction in a continuation based implementation of parsing combinators.

Introduction

One method to implement parsers in functional languages is to use parsing combinators [Bur75, Wad85, Hut92, PW94]. Parsing combinators are higher order functions which create more complex parsers from simpler ones. The behaviour of the parser, i.e., how it handles errors, ambiguous grammars, etc. can easily be changed by changing the behaviour of the parsing combinators.

This paper is concerned with parsing combinators intended to construct parsers for programming languages. Such grammars are normally not ambiguous, but they sometimes need arbitrary look ahead.[1] Arbitrary look ahead is implemented by backtracking, the parser tries one rule and if it fails backtracks and tries the next rule. Unfortunately, the backtracking forces the parser to hold on to the lexical stream when it has alternative rules so that it can try the next rule if the current one fails. This can lead to space leaks as nothing in the parser's input can be reclaimed by garbage collections. Grammars for programming languages do however often contain key words that could ensure the parser that it is using the correct rule from the grammar. If the parser finds an `if`-token then there is no need to save the input and try rules for other statements. The set of parsing combinators described in this paper contains an extra combinator that *cut* away the space leak and prevents backtracking.

The parsing combinators defined here can also report the location of parse errors, if the lexical symbols contain their positions. The error position is assumed to be the token that prevented the alternative that consumed the most tokens from continuing.

This paper is organised as follows: Section 1 contains an example grammar, together with data types for the corresponding syntax tree. These are used in the rest of the paper, together with the lexical analyser also described in this section. Section 2 describes the standard approach of parsing combinators, in the Hutton setting [Hut92], and some deficiencies of them. Hutton also described monadic combinators and they are introduced in section 3. Both of the above versions of parsing combinators hold on to the input longer than necessary in most cases. The reason is the way they are implemented, which is described in section 4. Solving the problem is not possible without a complete overhaul of the implementation. Such an implementation is in section 5. That section also introduces the new combinator that solves the problem with the retained input. The performance of the different implementations are then compared with heap profiles to visualise the 60% reduction in heap usage (section 6). The approach in this paper is then compared with related work (section 7) and concludes with a summary and future work (section 8).

1 Example grammar

Throughout this paper parsers for a simple language will be given. These parsers differ in implementation but they all parse the grammar given in Figure 1, and build an abstract syntax tree with the data types given in Figure 2.

It is possible to feed parsers with a list of characters, but more often the input string is preprocessed. The preprocessor, called lexical analyser, transforms the list of characters into a list of tokens. One example of a lexical analyser, and the

[1] The Haskell grammar [Hud92] is an example of such an grammar. The problem in Haskell is that it is not possible to distinguish a context from a type, until a `=>`-token is found, as all legal contexts also are legal types.

$$
\begin{aligned}
&program = definitions\ expression \\
\\
&definitions = definition\ definitions \\
&\qquad\qquad\ \ \mid\ \epsilon \\
\\
&definition = identifier\ identifiers\ \texttt{=}\ expression\ \ ; \\
\\
&identifiers = identifier\ identifiers \\
&\qquad\qquad\ \ \mid\ \epsilon \\
\\
&expression = identifier\ atomicExpressions \\
&\qquad\qquad\ \ \mid\ atomicExpression \\
\\
&atomicExpressions = atomicExpression\ atomicExpressions \\
&\qquad\qquad\qquad\qquad \mid\ \epsilon \\
\\
&atomicExpression = number \\
&\qquad\qquad\qquad\quad \mid\ (\ expression\ )
\end{aligned}
$$

Figure 1: The grammar used throughout this paper.

```
module Syntax where

data Program =
    Program [Definition] Expression

data Definition =
    Definition Identifier  [Identifier] Expression

data Expression =
    Number Integer
  | Application Identifier [Expression]

type Identifier = String
```

Figure 2: Data types for the abstract syntax tree that is built, according to the the grammar in Figure 1, by the parsers defined in this paper.

```
data Pos = NoPos
         | Pos Int Int
         | Eof
         deriving (Eq,Ord,Text)

lexical :: String -> [(Pos,String)]
lexical i = lex' 1 1 i
  where
   lex' r c [] = []
   lex' r c xxs@('\n':xs) = lex' (r+1) 1 xs
   lex' r c xxs@(x:xs) =
           if isSpace x then lex' r (c+1) xs
      else if isDigit x then token r c (span isDigit xxs)
      else if isAlpha x then token r c (span isAlpha xxs)
      else (Pos r c,[x]) : lex' r (c+1) xs

   token r c (t,xs) = (Pos r c,t) : lex' r (c+length t) xs
```

Figure 3: This is a lexical analyser that transforms a list of characters into a list of position-tokens pairs. Positions contain row and column numbers of the token, unless it is the end of file which has its own constructor. The `NoPos` and `Eof` positions are not used by the lexical analyser itself, the parsers use them if they reach an unexpected end of the token list or do not know where the error is.

one used in this paper, is in Figure 3. The input string is transformed from a list of characters to a a list of strings, where each string is a lexical token. The tokens are paired with their positions to make it possible to report positions of errors in the input. Pairing with position nearly doubles the amount of memory needed for the token stream, but positions are really useful in error messages.

2 BNF parsing combinators

This section describes a set of parsing combinators based on the BNF-notation, with most of the names taken from Hutton's article [Hut92].

In this standard approach, a parser takes a list of tokens and returns a value that distinguish between failure and success. The exact type of this return value is not important here, but note that in the case of success it must contain both the result of the successful parsing and the part of the input that was not consumed. A suitable type for a parser is therefore:

```
type Parser a b = [a] -> Return b [a]
```

where `Return`, for now, is an abstract type. The type variable `a` represents the input type, and the `b` represents the result type, of the parser. (A more concrete description of a parse type that can handle error positions is in section 4.)

The simplest combinator is `succeed` which corresponds to the ϵ symbol in BNF notation. The type of `succeed` is `b -> Parser a b`. Parsers are often supposed to return something, not only recognise the input, the return value from `succeed` is its argument. A `succeed` parser always succeeds and never consumes any input.

A combinator that uses the input is `satisfy` which takes a predicate, succeeding if the next token fulfills the predicate, otherwise `satisfy` fails. The type of `satisfy` is `(a->Bool)->Parser (Pos,a) a`. The return value from a `satisfy` parser is the accepted token without its position. This token is also removed from the input.

Simple parsers are combined into larger ones with the binary combinators `alt` and `seq`.[2] The `alt` combinator, with the type `Parser a b -> Parser a b -> Parser a b`, corresponds to alternation in BNF notation. The parser ``p1 `alt` p2`` recognise anything that either `p1` or `p2` would. In the implementations in this paper the second alternative is not tried if the first succeeds. It is therefore important to place the alternative that can consume the most tokens first. Trying both alternatives, and return the one that consumed the most input, is possible but prevents later space optimisations. Sequencing is done with `seq`, ``p1 `seq` p2`` recognise anything that `p1` and `p2` would if placed in succession. The type of `seq` is `Parser a b -> Parser a c -> Parser a (b,c)`.

It is also necessary to have a parser that can recognise the end of the input. The parser might otherwise ignore syntax errors, and all following tokens, returning a small abstract tree instead of an error message. The combinator `eof`, with the type `Parser (Pos,a) ()`, is therefore introduced. This combinator succeeds only at the end of the input.

It is very rare that the above combinators produce the right type of answer. The `using` combinator, with the type `Parser a b -> (b->c) -> Parser a c`, is therefore used to transform the result of a parser by applying a user defined function.

The above parsers are enough to parse any BNF-grammar, but a few more make it much easier.

```
literal :: Eq a => a -> Parser (Pos,a) a
literal p = satisfy (p==)
```

This combinator is a wrapper function for `satisfy`. It succeeds if the argument token matches the next token in the input. The return value is the token, which seldom is needed in the abstract syntax tree. Two versions of `seq`, which ignore one of the values returned by their sub-parsers, are therefore useful:

[2] Hutton uses the name `then` instead of `seq`, but `then` is a reserved word in Haskell.

```
xseq :: Parser a b -> Parser a c -> Parser a c
p1 `xseq` p2 = p1 `seq` p2 `using` snd

seqx :: Parser a b -> Parser a c -> Parser a b
p1 `seqx` p2 = p1 `seq` p2 `using` fst
```

Finally, `many` which takes a parser and repeatedly applies it until it fails, constructing a list with the return values.

```
many :: Parser a b -> Parser a [b]
many p = p `seq` many p `using` (\(x,xs) -> x:xs)
           `alt`
         succeed []
```

The priority of the combinators are defined to eliminate most parenthesis when they are used.

```
infixr 6 `alt`
infixl 7 `using`
infixr 8 `seq`, `xseq`, `seqx`
```

This gives `seq`, and its variants, the highest priority and `alt` the lowest. Making `using` left associative instead of right associative, which the other combinators are, makes it possible to use more than one `using` combinator without parenthesis. Note however that ``p `using` f `using` h`` is the same as ``p `using` (h.f)`` so it is doubtful that multiple applications of `using` will ever be used.

A parser, defined with Hutton's combinators, that parse the grammar in Figure 1 is given in Figure 4. This parser works, but it raises one question. Is it necessary to create pairs, with `seq`, that immediately are pattern matched, in the functions supplied to instances of `using`? The answer is no! One solution is to have another abstract syntax tree:

```
data Program =
    Program ([Definition],Expression)

data Definition =
    Definition (Identifier,([Identifier],Expression))

data Expression =
    Variable Identifier
  | Number Integer
  | Application (Identifier,[Expression])
```

This is *not* a nice solution, now the *abstract* syntax tree contains a structure that depend more on the implementation of the parser than the grammar, e.g., changing associativity for `seq` would enforce changes of the data type. Neither does this "solution" work for `many`. A much better solution is presented in the next section.

```
program =
  definitions `seq` expression `seqx` eof
              `using` (\(defs,exp) -> Program defs exp)

definitions = many  definition

definition =
   identifier `seq` identifiers
              `seq` literal "="
              `xseq` expression
              `seqx` literal ";"
     `using`  (\(id,(args,expr)) -> Definition id args expr)

identifiers = many identifier

expression =
   identifier `seq` atomicExpressions
              `using` (\(i,es) -> Application i es)
  `alt`
   atomicExpression

atomicExpressions = many atomicExpression

atomicExpression =
   number
  `alt`
   literal "(" `xseq` expression
               `seqx` literal ")"

identifier = satisfies (all isAlpha)

number = satisfies (all isDigit) `using` read
```

Figure 4: A parser that builds an abstract syntax tree using the example grammar.

3 Monadic parsing combinators

The previous definitions of parsing combinators are not the only ones possible. A combinator that is very useful, but missing from those mentioned, is a combinator that can use the result from the first parser in the second one. An example of such a combinator is `into` with the type `Parser a b -> (b -> Parser a c) -> Parser a c`. This combinator creates a parser by sequencing, in the same way as `seq`, but the second parser has access to the result of the first parser. The second parser is also responsible for creating the return value. This means that a sequence of two parsers do not have to create a pair, but it is possible to define `seq` with `into` if needed:

```
p1 `seq` p2 = p1 `into` \v -> p2 `using` \w -> (v,w)
```

It is also possible to define `using`:

```
p `using` f = p `into` \v -> succeed (f v)
```

However a more interesting approach is to define a new combinator instead of `using` and `seq`.

```
ap :: Parser a (b->c) -> Parser a b -> Parser a c
p1 `ap`  p2 = p1 `into` \f -> p2 `into` \x -> succeed (f x)
```

This combinator is function application, but for parsers. Note that `succeed` and `into` are unit and bind respectively in a parser monad [Wad85, Wad90, Wad92].

It is now possible to define `many` in a monadic style, without creating any pairs. The idea is to apply the cons operator `(:)` to its argument as soon as they are parsed.

```
many p = succeed (:) `ap` p `ap` many p
             `alt`
          succeed []
```

Before rewriting the whole parser, it is a good idea to define a combinator that ignore the return value for its second argument. (This is `seqx` under another name.)

```
chk :: Parser a b -> Parser a c -> Parser a b
p1 `chk` p2 = p1 `into` \f -> p2 `into` \_ -> succeed f
```

Both `ap`,`chk` and `into` are right associative and have the same priority as `seq`.

Figure 5 shows the parser written with `ap` and `chk` instead of `seq` and `using`. The difference from the parser in Figure 4 is that the "translation" functions are given first and then the pattern. There is no need to build any pairs with the

```
program =
   succeed Program ‘ap‘ definitions
                   ‘ap‘ expression
                   ‘chk‘ eof

definitions = many definition

definition =
   succeed Definition ‘ap‘ identifier
                      ‘ap‘ identifiers
                      ‘chk‘ literal "="
                      ‘ap‘ expression
                      ‘chk‘ literal ";"

identifiers = many  identifier

expression =
   succeed Application ‘ap‘ identifier
                       ‘ap‘ atomicExpressions
  ‘alt‘
   atomicExpression

atomicExpressions = many atomicExpression

atomicExpression =
   succeed Number ‘ap‘ number
  ’alt‘
   succeed id ‘chk‘ literal "("
              ‘ap‘ expression
              ‘chk‘ literal ")"

identifier = satisfy (all isAlpha)

number = succeed read ‘ap‘ satisfy (all isDigit)
```

Figure 5: This is a monadic version of the parser in Figure 4.

return values of the subparsers. Instead user defined functions are applied to one return value at a time. In many cases the user defined function is a constructor for one of the data types that define the abstract syntax tree.

The extra work needed to create pairs with the BNF combinators is now avoided. This style of parsing combinators will be used throughout the rest of this paper. Creating unnecessary pairs are however not the major problem with parsing combinators. A much worse deficiency is the space leak due to the `alt` combinator. The next section will explain why `alt` can cause a space leak, and why a major rewrite (of the parsing combinators) is necessary to solve the problem.

4 Traditional implementation of parsing combinators

The problem with `alt` is that it holds on to the stream of lexical symbols. Sometimes this is necessary, if the parser needs to backtrack, but in other cases the input ought to be reclaimed by the garbage collector. There are also many cases where it is necessary to keep the lexical stream for a while but not for as long as `alt` does it. One example is in `many`. Holding on to the input is necessary if the argument parser fails, so that the `succeed` alternative can be used, but not after the argument parser has succeeded once. It is true that the second alternative can still succeed, but the recursive call to `many` ensures that another occurrence of `succeed` is active. This occurrence is on a path that has consumed at least as many tokens, probably more, so it is the "better" alternative.

In the previous sections the parser type was mostly abstract. It is now time to look inside the type. There exist many possible realisations of the parser type. However, to parse programming languages it is enough to return one result, since grammars for programming languages are not ambiguous, or an error message. The parser also needs to return what is left of the input after the parser has consumed the token it needs. We will not bother reporting why the parser failed, only the position is returned at this point. With the above constraints a suitable type is:

```
type Parser a b = [a] -> Pos -> Return b [a]

data Return a b = FailAt Pos | OK a b Pos
```

This type is slightly different from the previously presented one as we want to report position of errors. We can also note that the type does not enforce pairing of tokens and positions in the input stream. It is possible to define a set of parser combinators that report error positions in number of parsed tokens.

The need to return the error position also when succeeding is not obvious. But it must be done, otherwise the position can be lost. Again, take `many` as an example; if the argument parser failed after consuming some tokens, then the `succeed` alternative will be used, and of course succeed. The error position reported by the first alternative is however further down in the input than the `succeed` alternative, and might be the true error position, i.e., no other alternative could consume more tokens. It must therefore be kept and returned together with the result of `succeed`.

A trivial implementation of the combinators are:

```
succeed a input pos = OK a input pos

satisfy p []            pos = FailAt Eof
satisfy p ((pos',x):xs) pos =
     if p x
     then OK x xs pos
     else FailAt (max pos' pos)

eof []            pos = OK () [] pos
eof ((pos',x):xs) pos = FailAt (max pos' pos)

into p1 p2 input pos =
   case p1 input pos of
      OK v input' pos' -> p2 v input' pos'
      notOK            -> notOK

alt p1 p2 input pos =
   case p1 input pos of
      FailAt pos'  -> p2 input pos'
      noFail       -> noFail
```

It is `satisfy` and `eof` that control which error that is furthest down in the input, the other combinators only pass along the position.

The implementation of `alt` is for non-ambiguous grammars, i.e., if the first parsers succeeds then there is no need to try the second one. *This is only true if the first parser does not accept a subset of the second one*, in which case the second parser might consume more input. The order of the subparsers is therefore relevant. This implementation of `alt` works, but it introduces a space leak. The `alt` combinator cannot drop its reference to the input until it knows that the first parser has succeeded.

As mentioned at the start of this section, it is sometimes not necessary to wait until the first parser is completely finished. In the example from `many` it was enough that a small part of the first alternative has succeeded to ensure that no backtracking is necessary. The problem is that a parser that knows it is the right alternative cannot tell any enclosing `alt` combinators that they do not need to try any other alternatives. This can however be solved by continuations, as will be explained in the next section.

5 Continuation based parsing combinators

The intuition behind continuation based parsing combinators are that a parser, instead of returning with a value encoding success or failure, has two continuations that are called for success and failure respectively. This is akin to continuation based denotational semantics with a normal and an exception continuation.

In our implementation the following types are chosen:

```
type ParserFail a b = Pos -> Return b [a]
type ParserOK a b c = b -> [a] -> Pos -> Return c [a]

type Parser a b c = ParserOK a b c -> ParserFail a c
                                   -> [a]
                                   -> Pos
                                   -> Return c [a]
```

The `Return` type in the above types is the same as in section 4. The parser type has three polymorphic type variables, the type of the tokens, the type of the object produced by this subparser and the type of the *complete* parser. The latter is the final value produced from the parser. A fail continuation needs the current position before it can return the final result, the success continuation needs the result of this subparser, the rest of the input and the furthest fail position so far.

An explanation to how this works is best done by showing the implementation of some of the combinators. The types for all combinators have changed so types are also given. The easiest one to implement is `succeed`, which only calls the success continuation.

```
succeed :: b -> Parser a b c
succeed a ok fail = ok a
```

The combinators that look at the input do a pattern match, and for `satisfy` applies a predicate, to decide if the fail or the success continuation should be called. These combinators also compare error positions to decide which is furthest in the input.

```
eof :: Parser (Pos,a) () b
eof ok fail []            pos = ok () [] pos
eof ok fail ((pos',_):xs) pos = fail (max pos pos')

satisfy :: (a->Bool) -> Parser (Pos,a) a b
satisfy p ok fail []            pos = fail Eof
```

```
satisfy p ok fail ((pos',x):xs) pos =
     if p x
     then ok x xs pos
     else fail (max pos pos')
```

The `into` combinator is not much more difficult, it applies the first parser with a changed success continuation. The new success continuation applies the second parser, given the result from the first parser, to the rest of the input before it continues with the original success continuation. If any of the parsers fails then the failure continuation is used.

```
into :: Parser a b c -> (b -> Parser a d c) -> Parser a d c
into p1 p2 = \ok fail ->
   p1 (\v -> p2 v ok fail)
      fail
```

The duplication of the fail continuation is however not nice, and will be a problem when the cut combinator is introduced below. But before introducing a new combinator we finish of defining the old ones.

An `alt` combinator with the same semantic, and memory behaviour, as in the previous section is trivial to implement with continuations. Call the first parser with the failure continuation changed to a call to the second parser.

```
alt :: Parser a b c -> Parser a b c -> Parser a b c
alt p1 p2 = \ok fail input ->
    p1 ok
       (p2 ok fail input)
       input
```

We do not get rid of the space leak here, as the failure continuation given to the first parser references the input, but that reference is now possible to remove in the subparsers with an additional combinator.

```
cut :: Parser a b c -> Parser a b c
cut p1 = \ok fail input pos -> p1 ok FailAt input NoPos
```

The above combinator drops the current fail continuation, and forgets where it happen, replacing it with an immediate failure. It is a bit cumbersome to use prefix combinators in Haskell due to their high priority, they always bind stronger than any infix combinator. The optimum choice for `cut` is to bind stronger than `alt` but weaker then `ap` and `chk`. This can be achieved by some additional combinators, which has the same priority as `ap` and `chk`.

```
apCut :: Parser a (b -> c) d -> Parser a b d -> Parser a c d
p1 `apCut` p2 = p1 `ap` cut p2

chkCut :: Parser a b c -> Parser a d c -> Parser a b c
p1 `chkCut` p2 = p1 `chk` cut p2
```

The intention is that these combinators drops the fail continuation after the first parser succeeds. Unfortunately, neither of them work in this respect, unless the compiler can do inlining and β-reduction. The reason they do not work is because the `intos` used to define them duplicate the fail continuation. This is visible if the `ap` in `apCut` is unfolded.

```
p1 `apCut` p2 =
    p1 `into` \f -> cut p2 `into` (\x -> succeed (f x))
```

The fail continuation to `p2` is stripped, but *not* the fail continuation to `succeed`! The fail continuation given to `succeed` is irrelevant for the final result but it will be kept until `succeed` is evaluated. This defies the purpose of `cut`, to get rid of the current fail continuation. A small change to the (unfolded) definition of `apCut` solves this problem.

```
p1 `apCut` p2 =
    p1 `into` \f -> cut (p2 `into` (\x -> succeed (f x)))
```

Now both the second `into` and the `succeed` combinators get the default fail continuation. It is not easy for a compiler to do this rewrite, but inlining and unfolding of both `intos` and `succeed` will give the same memory behaviour in this case. A similar implementation of `chkCut` is easily done.

The `apCut` combinator can be use in the monadic example grammar at several places. The mentioned behaviour of `many`, that no backtracking is necessary if the argument parser succeeds (end of section 4), is possible to utilise by changing one `ap` into an `apCut`.

```
many p = succeed (:) `ap` p `apCut` many p
           `alt`
         succeed []
```

Another example is in the rule for *definition*. This is the only rule that can match the equal character. If one of these are accepted then no backtracking can find a better alternative than the current one.

```
definition =
    succeed Definition `ap`    identifier
                       `ap`    identifiers
                       `chk`   literal "="
                       `apCut` expression
                       `chk`   literal ";"
```

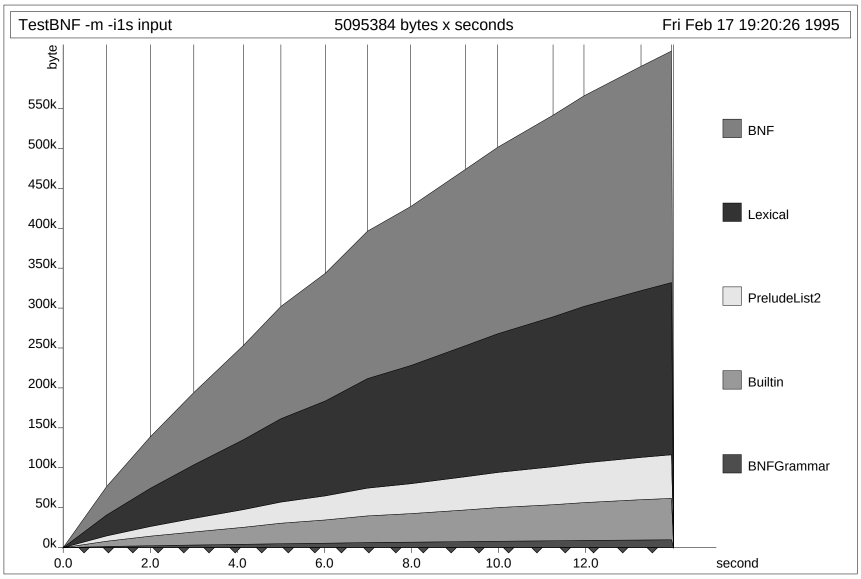

Figure 6: Parsing of a 10 000 character long string with a parser that uses the parsing combinators influenced by the BNF notation. The maximum heap space needed is 600 kByte, and it takes 14 seconds.

The final place where an `apCut` can be used in the example monadic parser is after the open parenthesis in `atomicExpression`.

6 Performance measurements

Figures 6–8 show the memory, and time, needed to parse a 10 000 character long string, with four different parsers.

The first one, Figure 6, uses the BNF influenced parsing combinators building and destructing lot of pairs. Most of the memory used by the module BNF, which defines the parsing combinators, is however *not* consumed by pair nodes. The memory is used by `using` to build application nodes for the user defined functions. These functions are not evaluated until the parser succeeds, or never in the case of failure. Nodes created by the lexical analyser (`Lexical` and `PreludeList2`) also use a significant amount of memory.

Using monadic parsing combinators do not decrease the memory usage by much, as seen in Figure 7, only the few pairs are gone. The problem with keeping lexical data is still there. Changing the implementation to continuation based does not change anything, neither time nor space decreases with much. The heap profile looks identical to Figure 7.

The large saving in space, and also in time, is when the `cut` combinator is used (see Figure 8). This time most nodes created by the lexical analyser is reclaimed

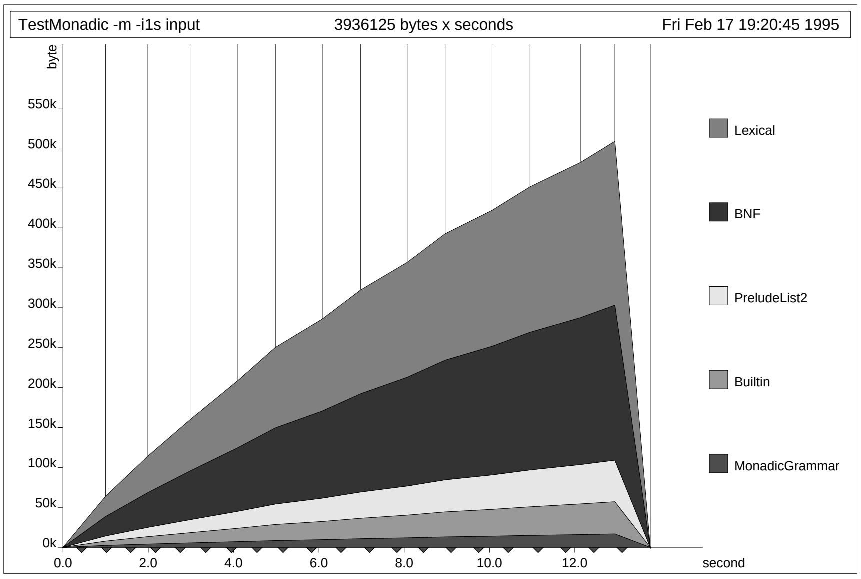

Figure 7: A parser written with monadic parsing combinators, implemented by returning different constructors for success and failure. This parser uses 100 kByte less memory than the one in Figure 6, but the same amount of time. Implementing the parsing combinators with continuations decreases the speed with less than 1 second.

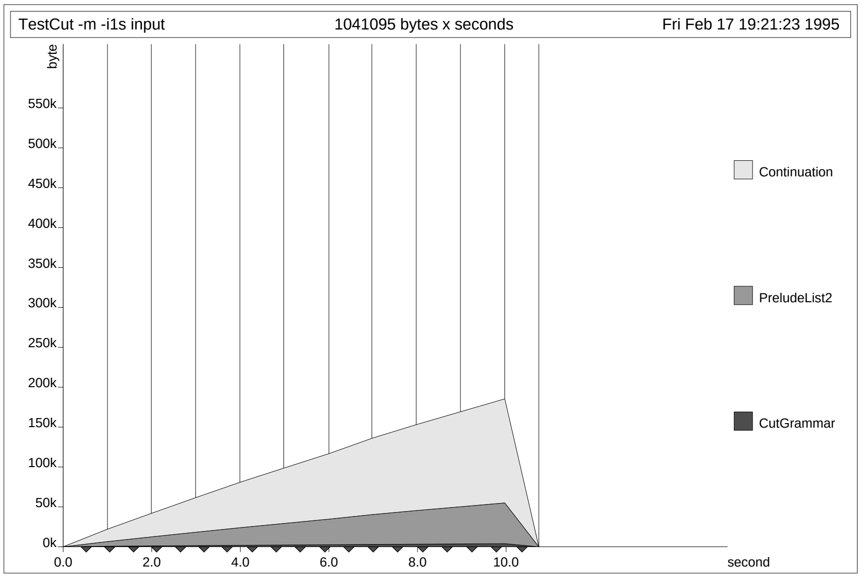

Figure 8: Using the continuation based implementation of `apCut` in the monadic parser (Figure 5), where applicable, completely removes all memory allocated by the lexical analyser from the heap profile. (Identifiers are built by a prelude functions defined in `PreludeList2`.)

by the garbage collector, it is only identifiers (created by `span` in `PreludeList2`) that is still kept. But the identifiers are needed in the abstract syntax tree, and can therefore not be called a space leak.

7 Related work

Wadler's set of parsing combinators described in [Wad85] includes a `cut` combinator. His parsing combinators can return multiple answers, and the `cut` combinator is used when one answer is enough. The `nofail` combinator he describes behaves more like the `cut` combinator in this paper. By using both of these combinators it is possible to produce space efficient parsers. These parsers can however not report where in the input an error occurred. Adding error positions is not trivial since usage of his `nofail` combinator is a guarantee that the parser will not fail later, if it does then it is a run-time error.

Hutton's work described in [Hut92] is very related to this work. This work does in fact build on that article. He describes many implementations of parsing combinators, one of which is closely related to the constructor based implementation in this paper. Error reporting is however easier to use with the implementation given here, Hutton demands user inserted `nofail` combinators that translates failures into errors. The `nofail` combinator has the same semantic as the `cut` combinator in this paper, but without the space savings.

Partridge and Wright [PW94] use continuations in their implementation, but have no cut combinator. Their combinators are however intended for an LL(1) grammar so no backtracking is needed anyway. This is enforced by an `alt` combinator that does not try any further rules if one rule has consumed any input.

8 Summary

At least two conclusions can be drawn from this work; one concerns the use of parsing combinators, the other is about space leaks in general.

Parsing combinators are very powerful, but can have a dangerous space behaviour. Given the right implementation, however, it is possible to write fast space efficient parsers with them.

Excessive use of heap profiling can highlight space leaks in a program, but they do not solve them. It can be necessary to utilise knowledge about the input (it contains key-words that ruled out all but one alternative), and completely rewrite the implementation (using continuations instead of constructors), to remove the space leak.

It is unclear how to implement a space efficient `cut` for constructor based parsing combinators and still include positions in the error messages.

References

[Bur75] W. H. Burge. *Recursive Programming Techniques*. Addison-Wesley Publishing Company, Reading, Mass., 1975.

[Hud92] Paul Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[Hut92] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2:323–343, 1992.

[PW94] Andrew Partridge and David Wright. Parser combinators need four values to report errors. Technical report, Department of Computing Science, University of Tasmania, 1994.

[Wad85] P. Wadler. How to Replace Failure by a List of Successes. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, Nancy, France, 1985.

[Wad90] P. Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–77, Nice, France, 1990.

[Wad92] P. Wadler. The essence of functional programming. In *Proceedings 1992 Symposium on principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, 1992.