Partial parsing: combining choice with commitment

Malcolm Wallace

University of York, UK

Abstract. Parser combinators, often monadic, are a venerable and widely-used solution to read data from some external format. However, the capability to return a partial parse has, until now, been largely missing. When only a small portion of the entire data is desired, it has been necessary either to parse the entire input in any case, or to break up the grammar into smaller pieces and move some work outside the world of combinators.
This paper presents a technique for mixing lazy, demand-driven, parsing with strict parsing, all within the same set of combinators. The grammar specification remains complete and unbroken, yet only sufficient input is consumed to satisfy the result demanded. It is built on a *combination* of *applicative* and *monadic* parsers. Monadic parsing alone is insufficient to allow a choice operator to coexist with the early commitment needed for lazy results. Applicative parsing alone can give partial results, but does not permit context-sensitive grammars. But used together, we gain both partiality and a flexible ease of use.
Performance results demonstrate that partial parsing is often faster and more space-efficient than strict parsing, but never worse. The trade-off is that partiality has consequences when dealing with ill-formed input.

1 Introduction

Parser combinators have been with us for a long time. Wadler was the first to notice that parsers could form a monad [12]. Tutorial papers by Hutton and Meijer [5, 6] illustrated a sequence of ever-more sophisticated monadic parsers, gradually adding state, error-reporting and other facilities. Röjemo [9] introduced applicative[1] parsers for space-efficiency, whilst Leijen's Parsec [7] aimed for good error messages with both space and time efficiency by reducing the need for backtracking except where explicitly annotated. Packrat parsing [3] eliminates backtracking altogether by memoising results (a technique that is highly space-intensive). Laarhoven's ParseP [11] also eliminates backtracking, by parsing alternative choices in parallel. Swierstra et al have shown us how to do sophisticated error-correction [10], permutation parsing [1], and on-line results through breadth-first parsing [4], all in an applicative style.

But, aside from the latter work, the particular niche of *partial parsing* is still relatively unexplored. A parser, built from almost any of the currently available

[1] The applicative functor is now recognised [8] as a structure simpler than a monad.

combinator libraries, needs to see the entire input before it can return even a portion of the result. Why is it unusual to be non-strict, demand-driven, partial? Because of the possibility of parse errors. If the document is syntactically incorrect, the usual policy is to report the error and do no onward processing of the parsed data — in order to prevent onward processing, we must wait until all possible errors could have arisen.

Sometimes this is not desirable. Imagine processing a large XML document that is already known to be well-formed. Why should the program wait until the final close-tag has been verified to match its opener, before beginning to produce output? There is also often an enormous memory cost to store the entire representation of the document internally, where lazy processing could in many cases reduce the needed live heap space to a small constant.

Even if we do not know for certain that a document is well-formed, it can still be useful to process an initial part of it. Think too, of an interactive exchange with a user, or a network communications protocol, where input and output must be interleaved.

Of course, there is a flip-side to partial processing – the parsed value may itself be partial, in the sense of containing bottom (undefinedness, or parse errors). One must be prepared to accept the possibility of notification of a parse-failure when it would be too late to undo the processing already completed.

Of all the libraries available, only the one by Hughes and Swierstra [4] has already demonstrated how to achieve partial parsing (they call it 'online' parsing). The framework is applicative in style (rather than monadic) and automatically analyses the grammar to determine when no further errors or backtracking may occur over the part of the input that has already been seen. In the absence of such errors, it becomes possible to return the initial portion of the resultant data structure with confidence that no other parse is possible. (So in fact, their partial values do not contain bottoms.)

However, the mechanism they use to implement this scheme is rather complex, involving polymorphic recursion, and both existential and rank-2 type extensions to Haskell. Whilst undoubtedly powerful, the scheme is also somewhat hard to understand, as witnessed by the fact that no parsing library (except the one which accompanies their paper) has adopted anything like it. The library itself can be fiendishly difficult to modify, even to add simple primitives found in other libraries (e.g. the 'satisfy' of Figure 2).

This paper presents a simpler, more easily understood, method to achieve partial parsing. It avoids scary higher-ranked types, instead continuing to represent parsers in a basic, slightly naive, way. The price to pay is that there is no automated analysis of the parsers, so the decision on where to be lazy or strict is left in the hands of the grammar writer.

We first outline some ordinary (strict) monadic parser combinators, then illustrate how a naive conversion to use a lazy sequencing operator is problematic. An alternative is explored, using a *commit*-based technique to limit backtracking, but this too is found to be inadequate. Finally, it is shown that by mixing

applicative and monadic combinators, the user can gain explicit control over the lazy or strict behaviour of their parsers.

All the combinator variations described here are freely available in the *polyparse* library [14].

1.1 Simple polymorphic parsers

An outline of the basic concept and implementation of monadic parsing now follows, with corresponding code in Figure 1. For a fuller treatment, the reader is directed to Hutton and Meijer's comprehensive tutorial [6].

```
newtype Parser t a = P ([t] → (Either String a, [t]))
instance Functor (Parser t) where
  fmap f (P p) = P (λts → case p ts of
                             (Right val, ts') → (Right (f val), ts')
                             (Left msg, ts')  → (Left msg, ts'))
instance Monad (Parser t) where
  return x    = P (λts →   (Right x, ts))
  fail e      = P (λts →   (Left e, ts))
  (P p) >>= q = P (λts →   case p ts of
                             (Right x,  ts') → let (P q') = q x in q' ts'
                             (Left msg, ts') → (Left msg, ts'))
runParser           :: Parser t a → [t] → (Either String a, [t])
runParser (P p)     = p
onFail              :: Parser t a → Parser t a → Parser t a
(P p) `onFail` (P q) = P (λts → case p ts of
                                  (Left _, _) → q ts
                                  right       → right)
next :: Parser t t
next = P (λts → case ts of
                  [] → (Left "Ran out of input (EOF)", [])
                  (t : ts') → (Right t, ts'))
```

Fig. 1. Basic parser combinators.

The *Parser* type is parameterised on the type of input tokens, t, and the type of the result of any given parse, a. A parser is a function from a stream of input tokens, to the desired result paired with the remaining unused tokens. If a parse fails, the failed result is reported in the String alternative of the *Either* type. Many early combinator libraries used *lists* of results to represent multiple ambiguous parses, or failure (if empty). However in practice only the first result is usually of interest, and the empty list unfortunately gives no helpful information in case of errors, hence the design choice here to use the *Either* type.

Parsers are sequenced together using monadic notation, hence the instances of *Functor* and *Monad*. It is clear by inspection of the definition of the sequence operator ($\gg\!=$), that it is strict in the result of the first parser – it performs a **case** comparison on it.

A parser can be 'run' by applying it to some input token list. The *runParser* function thus lifts embedded parsers out of the monad, back into some calling context.

Choice between different parses is expressed by *onFail*, which tries its second argument parser only if the first one fails. Note that information may be lost, since any error message from the first parser is thrown away. We return to this point later.

Finally, we need a single primitive parser called *next*, that returns the next token in the stream.

Higher-level combinators can be defined using the primitives above. For instance, those in Figure 2.

```
  -- One token satisfying a predicate.
satisfy     :: (t → Bool) → Parser t t
satisfy p   = do { x ← next
                 ; if p x then return x else fail "Parse.satisfy: failed"}
  -- Use 'Maybe' type to indicate optionality.
optional    :: Parser t a → Parser t (Maybe a)
optional p  = (fmap Just p) `onFail` return Nothing
  -- 'exactly n p' parses precisely n items, using the parser p.
exactly     :: Int → Parser t a → Parser t [a]
exactly 0 p = return []
exactly n p = do { x ← p
                 ; xs ← exactly (n − 1) p
                 ; return (x : xs)}
  -- Parse a (possibly empty) sequence. Cannot fail.
many        :: Parser t a → Parser t [a]
many p      = do { x ← p
                 ; xs ← many p
                 ; return (x : xs)} `onFail` return []
  -- Parse a sequence followed by a terminator.
manyFinally     :: Parser t a → Parser t z → Parser t [a]
manyFinally p z = do { xs ← many p
                     ; z
                     ; return xs}
```

Fig. 2. Higher-level combinators built from primitives.

A parser for some particular textual data format is then built from these combinators, and looks rather like a recursive-descent grammar. The example

in Figure 3 illustrates a grammar for a simplified form of XML. We assume the input tokens have already been lexed according to XML-like rules, and that error messages are easily augmented with positional information. Definitions for less interesting parsers such as *name* and *attribute* are omitted.

```
data Content = Elem String [Attr] [Content]
             | Text String
content   = element `onFail` text
                  `onFail` fail "unrecognisable content"
element   = do
  { token "<"
  ; n   ← name
  ; as  ← many attribute
  ; do { token "/>"
       ; return (Elem n as [])}
    `onFail`
    do { token ">"
       ; cs ← manyFinally content (endtag n)
       ; return (Elem n as cs)}
  } `onFail` fail "unrecognisable element"
endtag n = do
  { m ← bracket (token "</") name (token ">")
  ; if n ≡ m then return ()
    else fail ("tag <" ++ n ++ "> terminated by </" ++ m ++ ">")
  }
text      = fmap Text stringToken
                  `onFail` fail "unrecognisable text"
token t   = satisfy (≡ t)
```

Fig. 3. Example combinator grammar for a simplified XML.

1.2 Problems and Limitations

Complete consumption of input. If we only want a small part of the parsed data, we must still parse the whole thing first. For instance, given the XML input

```
<a><b>hello</b><c>world</c></a>
```

we may wish to extract only the contents of the `<b>` tag, yet are forced to read the `<c>` tag as well! The input could be arbitrarily large, with the fragment of sole interest close to the beginning. Not only that, but the uninteresting part of the input must be fully well-formed, which may be too restrictive for some applications.

One way to avoid complete parsing is to resort to other coding techniques outside the parsing monad. An example of such a technique is repeatedly calling

runParser on smaller units of the input, tracking unused tokens between calls. Yet manipulation of the parse state is exactly the tedious boilerplate that the monad is supposed to hide! Moving outside the monad also leads to a highly non-modular grammar, requiring much special-case code to deal with the specific fragments of interest.

Ideally, we would like to keep the original grammar, and just interpret it lazily in order to return a partial result.

Error messages are often poor. Due to backtracking over choice points, they rarely point close to the location where the input fails to match the grammar. Indeed, in the worst case, errors are often reported at the topmost outermost layer of the value's structure, i.e. column 1 of the input.

Using our example XML grammar (Figure 3), the error message from attempting to parse the incorrect input

```
<a><b>hello<b/></a>
```

is not, as one might hope,

```
"tag <b> terminated by </a> at char 18"
```

but rather

```
"unrecognisable content at char 1"
```

Why? Because failure anywhere inside the inner do-blocks of the grammar is thrown away by the enclosing nested *onFail*s, which propagate the failure outwards, but changing the error message at every stage.

One might wonder whether it suffices to re-write *onFail* to preserve and accumulate error messages, rather than ignore them? Unfortunately this only leads to a huge collection of misleading errors, amongst which it is difficult to find the single accurate one.

Backtracking over choices sometimes leads to inefficiency. Again for the example incorrect input

```
<a><b>hello<b/></a>
```

despite the fact that we have already found a valid open tag `<a>` for the element branch of the grammar, nevertheless because something further inside the element is incorrect, this parser necessarily backtracks to the top-level *content* parser and attempts to match the non-element case *text*, on which it is bound to fail.

The XML example only allows for two choices of outer construct – element or text, corresponding to the two branches of the resultant Haskell sum type – but imagine a type and its grammar having a hundred possible different constructors. A parse failure deep within the first branch could lead to the evaluation of all of the remaining 99 constructor choices, failing on all of them, before giving up. Not only is the error message imprecise, but it took much longer than necessary to deliver it!

1.3 Roadmap

In the following sections, we address some of these limitations of the basic parser combinators. First, we make a naive attempt at a lazy parsing monad, to illustrate the conflict between committing to return a value, yet retaining choice. Then we examine whether the prevention of backtracking (second and third issues above) can not only give better error messages, but also allow a more precise determination of commitment points, at which partial values can be safely returned. Finally, we give a full yet simple solution in which lazy and strict sequencing can be freely mixed.

```
newtype Parser t a = P ([t] → (a, [t]))
runParser         :: Parser t a → [t] → (a, [t])
runParser (P p) = p
instance Functor (Parser t) where
  fmap f (P p)   = P (λts → case p ts of
                              (val, ts') → (f val, ts'))
instance Monad (Parser t) where
  return x       = P (λts → (x, ts))
  fail e         = P (λts → (throwException e, ts))
  (P p) >>= q    = P (λts → let (x, ts') = p ts
                                (P q')  = q x
                            in  q' ts')
throwException  :: String → a   -- throw to enclosing I/O monad
```

Fig. 4. A futile attempt at a lazy parsing monad.

2 Naive lazy monadic sequencing

It is readily observed that the parser type presented in Figure 1 can either return an error message, or a polymorphic value, but not both. But for partial parsing, we want the parser to return the polymorphic value regardless. Any error due to parse failure could be hidden within the value as an exception, to be triggered only when the immediate subcomponent containing the error is demanded.

Thus, a naive implementation of a lazy monad (corresponding to the strict one already given) is to simply erase the *Either* type constructor, and all *Left* and *Right* value constructors. Any constructions that previously built a *Left* will instead throw an exception. Case branches that previously scrutinised a *Left* can be omitted, and those that scrutinise a *Right* now see the contained value directly – see Figure 4. Furthermore, the sequence operator ($\gg\!=$) is made lazy by scrutinising the result of its first operand with a (non-strict) **let**-binding, rather than with a **case** as before (the latter would be strict in the tuple pattern).

Sadly though, this approach leaves us with no way to code the choice operator. As the very name *onFail* suggests, the combinator must be able to detect a failure in its left argument before it can try its right argument. But the naive partial parser type no longer represents failure explicitly as a value. Instead, it is a control-flow construct – an exception. One might wonder whether the exception can be caught and handled within the *onFail* combinator, but sadly, we are in the wrong monad! Exceptions can be caught only from the I/O monad, not the parsing monad.

The lesson here is that the early commitment implicit in returning a partial value, prevents a later choice. So let us examine a different approach, where commitment is made explicit. By annotating the precise locations in the grammar where commitment is possible, it will remain possible to implement choice everywhere else.

3 Choice and commitment

The introduction of explicit commitment is initially motivated by a desire to improve error reporting. We have already seen how backtracking over choice points leads to poor error messages. But in addressing this problem, we will disallow backtracking at defined locations, and therefore also eliminate choice there too. The hope is that this will enable us to return a partial result at that same location.

Essentially, parse failures can be divided into two separate classes: recoverable and unrecoverable. Recoverable errors allow backtracking through any enclosing choice point; unrecoverable errors should always be reported to the user – they override any enclosing choice point.

We refine the original parser type to codify the different error classes – see Figure 5. Instead of the plain *Either* type, we introduce *Result*, which gives a three-valued logic: success, failure, or a committed result. The committed result is the mechanism used to prevent backtracking. Ultimately there is of course no semantic difference between a plain success or a committed success. But a commitment that ends up being a failure cannot be recovered – it must be reported. By contrast, the choice combinator can throw away an uncommitted failure, to try some other branch.

Figure 5 shows how the basic monadic machinery is modified for this new representation. The choice combinator tries alternatives only when errors are recoverable – after commitment, no alternative is possible, just as surely as if the result of the first operand were successful.

Finally, we add the new combinator *commit*, which serves as the primary mechanism for a grammar-writer to indicate where sufficient tokens have been seen to be certain that no alternative parse path is possible.

Commit is a kind of dual of the *try* combinator in Parsec [7]. In Parsec, no backtracking is allowed normally – it must be explicitly permitted with *try*. But in our framework, backtracking is normally the default, except where explicitly disallowed by *commit*. Ultimately, they have a similar effect however: the calling

```
data Result t a  = Success a [t]
                 | Failure String [t]
                 | Commit (Result t a)
newtype Parser t a = P ([t] → Result t a)
runParser         :: Parser t a → [t] → (Either String a, [t])
runParser (P p) = result ∘ p
  where
    result (Success a ts) = (Right a, ts)
    result (Failure e ts) = (Left e,  ts)
    result (Commit r)     = result r
instance Monad (Parser t) where
  return x    = P (Success x)
  fail e      = P (Failure e)
  (P p) >>= q = P (continue ∘ p)
    where continue (Success x ts) = let (P q') = q x in q' ts
          continue (Failure e ts) = Failure e ts
          continue (Commit r)     = Commit (continue r)
onFail                :: Parser t a → Parser t a → Parser t a
(P p) `onFail` (P q) = P (λts → case p ts of
                                  (Failure _ _) → q ts
                                  r             → r)
commit               :: Parser t a → Parser t a
commit (P p)         = P (Commit ∘ p)
```

Fig. 5. Parsers with commitment, for better error-reporting.

context of *try* or *commit* will never be returned to; in both cases, we have committed to any particular branch that led to the current call, yet are still willing to try different alternative branches inside the argument to *commit*.

Commit is similar to the *cut* operator used by Röjemo's combinators [9] to achieve space efficiency. Indeed, it solves the very same space-leak, which is also identified by Leijen as a primary motivator for developing Parsec [7]. *Commit* also bears a strong similarity to the extra-logical ! (cut) operator in Prolog, which serves to prevent backtracking in its implementation model.

Figure 6 refines the example XML grammar of Figure 3, re-expressing it in terms of *commit*. Note the careful placement of commitment after sufficient tokens have been read to disambiguate the cases. Now, when given the badly-formed input string

```
<a><b>hello<b/></a>
```

in contrast to the previous attempt, we receive the error message

```
"<b> terminated by </a> at char 18"
```

```
element = do
  { token "<"
  ; commit (do
      { n   ← name
      ; as  ← many attribute
      ; do { token "/>"
           ; commit (return (Elem n as [ ])) }
        `onFail`
        do { token ">"
           ; commit (do { cs ← manyFinally content (endtag n)
                        ; return (Elem n as cs) }) }
      } `onFail` fail "unrecognisable element")
  }
```

Fig. 6. The XML grammar for 'element', re-expressed using *commit*. (Other productions remain unchanged.)

as hoped.[2] The *endTag* parser is responsible for generating the message, and the nearest enclosing *commit* (in the second branch of *element*) is responsible for ensuring that it (and no other message) is reported.

It is worth noting that one of the commonest sources of bugs in Parsec grammars is that users do not know where to place the *try* combinator. Parsec grammars are *LL(1)* by default, but *try* is used to permit extra lookahead for disambiguation. It can be difficult to look at a grammar and count the required lookahead. This leads to the curious observation that Parsec grammars are not in fact compositional! When a user plugs two previously-working grammars together, the combination often turns out not to work as expected, and they resort to simply sprinkling *try* into various locations to discover a fix.

By contrast, we believe that the *commit* approach is superior, because the lack of a *commit* will not cause the grammar to fail unexpectedly, merely to be inefficient or to give unhelpful error messages. In addition, the intuition needed to place a *commit* combinator correctly within the grammar is a much lower barrier. It indicates a simple certainty that no alternative parse is possible once this marked point has been reached. This is easier to verify by inspection than deciding how many tokens of lookahead are required to disambiguate alternatives.

4 How to be lazy

Does the form of explicit commitment described above help to achieve partial results? Sadly the answer is no, at least not directly. Once the parser has emitted a *Commit* constructor, it has still not determined whether the result will be a

[2] Different implementations of the *manyFinally* combinator can yield even more detailed error messages.

success or failure. And even if it does turn out to be a success, we do not know (at the moment of commitment) which constructor of the successful polymorphic value is going to be returned. Indeed, there is no way to discover it, *because* the result of *commit* is fully polymorphic – by definition the combinator cannot know anything about the enclosed value's representation.

Thus, the insight gained is that we need a combinator which, in addition to explicitly marking the point of commitment to a value, must know enough about that value to return a portion of it immediately. Commitment must be parameterised on the thing we are committing to.

Furthermore, some new form of sequencing combinator is required, which can build a whole value from component parts, but is capable of returning a partially composed value before the end of the sequence is complete. For this, we must leave behind the monadic world, especially monadic sequence. Some strict sequencing will remain useful, but short of composing multiple monads, we cannot mix lazy and strict sequences using only the monadic framework.

It turns out that the world of *applicative functors* [8] is a more convenient place to find the kind of sequence we want. In particular, functorial *apply* can be viewed as a sequencing operator. The correspondence to monadic bind (and the difference) is clearest when the arguments to *apply* are flipped:

$$\begin{array}{lll} \mathit{apply} & :: \mathit{Applicative}\ f & \Rightarrow f\ (a \rightarrow b) \rightarrow f\ a \quad \rightarrow f\ b \\ \mathit{flip\ apply} & :: \mathit{Applicative}\ f & \Rightarrow f\ a \ \ \rightarrow f\ (a \rightarrow b) \ \ \rightarrow f\ b \\ (\gg\!=) & :: \mathit{Monad}\ m & \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \end{array}$$

Some existing parser combinator libraries are based on applicative functors, rather than monads [9, 4]. *Apply* is less powerful than monadic bind, in the sense that the former can be implemented in terms of the latter, but not vice versa. This captures the intuition that *apply* simply combines functorial values, that is, the order of evaluation of left and right arguments is not restricted, so one cannot depend on the other. By contrast, the monadic bind allows the contents of the functorial value to be examined, named, and used, in the sequel. Thus, the monadic style allows context-sensitive parsing, whilst the applicative style is context-free.

There is a straightforward and obvious definition of *apply* in terms of bind:

$$\mathit{pf}\ \text{`}\mathit{apply}\text{`}\ \mathit{pa} = \mathbf{do}\ \{f \leftarrow \mathit{pf};\ a \leftarrow \mathit{pa};\ \mathit{return}\ (f\ \ a)\}$$

but of course this is no good for returning partial results, because as we already know, the monadic bind is insufficiently partial – that is the problem we are trying to overcome. Instead, we can define *apply* to always succeed and return a result, if its left argument succeeds. For instance, if the value delivered by the left functorial argument is a partially-applied data constructor, and the right argument delivers the next component of that constructor, then we can immediately return the constructor portion of the value, before we know whether the component to be contained within it is fully parse-correct.

In the formulation of Figure 7, we revert to the original *Either* variant of the *Parser* datatype, but could equally have used the *Result* variant associated with

```
newtype Parser t a = P ([t] → (Either String a, [t]))
runParser          :: Parser t a → [t] → (a, [t])
runParser (P p) = convert ∘ p
  where convert (Right a, ts) = (a, ts)
        convert (Left e, ts)  = (throwException e, ts)
infixl 3 `apply`
apply              :: Parser t (a → b) → Parser t a → Parser t b
(P pf) `apply` pa = P (continue ∘ pf)
  where
    continue (Left e,  ts) =  (Left e, ts)
    continue (Right f, ts) =  let (a, ts') = runParser pa ts
                              in  (Right (f a), ts')
```

Fig. 7. A parser that mixes monads and applicative functors. (The instances of Monad and Functor classes, and the implementation of *onFail* remain exactly as in Figure 1.)

the *commit* combinator. The improved error-reporting of the latter is entirely independent of, and orthogonal to, the issue of partiality. A point of special note is that the use of the *Either* type for parsing continues to allow the original implementation of the choice combinator *onFail*.

But the key point in this definition of *apply* is that if the first parser succeeds, then the whole combined parse succeeds (returns a *Right* value). Both failures and successes within the second parser are stripped of their enclosing *Left* or *Right*, and used 'naked'. The new *runParser* is the place where the *Either* wrapper is discarded, leaving just the naked value (or exception).

```
element = do
  { token "<"
  ; return Elem
       `apply` name
       `apply` many attribute
       `apply` (do { token "/>"
                   ; return [] }
                `onFail`
               do { token ">"
                  ; manyFinally content (endtag n) })
  } `onFail` fail "unrecognisable element"
```

Fig. 8. The XML 'element' grammar in lazy form.

For illustration, Figure 8 re-expresses the XML grammar once again, this time in a lazy fashion. Application is of course curried, so chaining many parsers

together is as straightforward in the applicative case as in the monadic case. Note how a mixture of strict monadic sequence and lazy application is used, and how easily strict sequence (with the ability to backtrack over choices) sits inside an enclosing applicative (partial, lazy) sequence.

It is also worth making the point that this revised grammar no longer checks that XML end tags match their opening tags *in advance* of returning the prefix of the element. The check will only occur once the final inner content is demanded by the context of the parser.

So, now that we have two ways to express sequence with combinators, the user must develop their grammar to make careful use of lazy or strict sequence as appropriate. Many of the non-basic combinators must be checked carefully to ensure that they are sufficiently lazy. For example, if we want *exactly* (from Figure 2) to return a lazy list without waiting for all elements to become available, we must rewrite the earlier definition as follows:

```
exactly     :: Int → Parser t a → Parser t [a]
exactly 0 p = return []
exactly n p = do x ← p
                 return (x:) `apply` exactly (n − 1) p
```

5 Evaluation

5.1 Performance

To give a flavour of the performance of lazy partial parsing, we designed a small number of (slightly artificial) tests using the Xtract tool from the HaXml suite [15, 13]. Xtract is a grep-like utility which searches for and returns fragments of an XML document, given an XPath-like query string. Because the intention is to find small parts of a larger document, it is an ideal test case for partial parsing. The XML parser used by Xtract is switchable between the strict and lazy variations[3].

We created a number of well-formed XML documents of different sizes n (ranging on a logarithmic scale from 10 to 1,000,000) with interesting characteristics:

– linear: the document is a flat sequence of n identical elements enclosed in a single wrapper element.
```
<file> <element/> <element/> ... </file>
```
– nested: the document contains n elements of different types, with element type i containing a single element of type $i + 1$ nested inside it, except for the nth element, which is empty.
```
<file> <element0>
           <element1>
             <element2> ...
```

[3] Software releases HaXml-1.20 and polyparse-1.2 together contain all the test code.

```
            </element2>
          </element1>
        </element0>
</file>
```

– follow: the nested document, followed by a single trivial element, together enclosed in a wrapper element.

```
<file> <element0>
         <element1> ...
       </element0>
       <follow/>
</file>
```

The queries of interest are:

– `Xtract "/file/element[0]" linear`
  Find the first element in the flat sequence of elements.
– `Xtract "/file/element[$]" linear`
  Find the last element in the flat sequence of elements.
– `Xtract "//element`n`" nested`
  Find the most deeply nested element(s) in the nesting hierarchy. The difference between this test and the following one is that this test continues searching after finding the first result.
– `Xtract "//element`n`[0]" nested`
  Find only the first most deeply nested element in the nesting hierarchy.
– `Xtract "/file/follow" follow`
  Find the single top-level element that follows the large deeply-nested element.

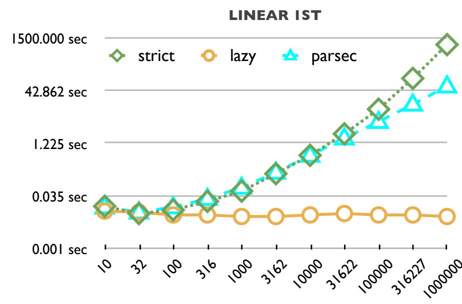
The time and memory taken to satisfy each query is given in Tables 1 and 2, using both the strict and lazy parser variations. In all cases, the lazy parser is better (both faster, and more space efficient) than the strict parser. For extremely large documents, where the strict parser often crashes due to stack overflow, the lazy parser continues to work smoothly. For the cases where the only result is a small, early, fragment of the full document, laziness reduces the complexity of the task from linear to constant, that is, it depends on the required distance into the document, not on the size of the document. Even when the searched element is at the end of the linear document, the lazy version is orders of magnitude faster, for large inputs.

The difference between the resources used by the lazy queries for the first vs. all nested elements is interesting. Taking the first element is almost exactly twice as fast, and half as space-hungry, as looking for all elements. This corresponds exactly to the intuition that the latter needs to check all closing tags against their openers (of which there are equal numbers), whilst the former only needs to look at the opening tags.

None of this is very surprising of course. Lazy streaming is well-known to improve the complexity of many algorithms operating over large datasets, often allowing them to scale to extreme sizes without exhausting memory resources,

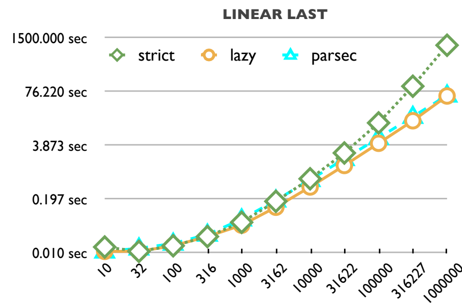
LINEAR 1ST

N	STRICT	LAZY	PARSEC
10	0.018	0.013	0.017
32	0.011	0.012	0.012
100	0.014	0.01	0.017
316	0.025	0.01	0.03
1000	0.05	0.009	0.064
3162	0.163	0.009	0.177
10000	0.563	0.01	0.558
31622	2.407	0.011	1.795
100000	12.733	0.01	5.471
316227	102.272	0.01	17.685
1000000	1001.411	0.009	60.321



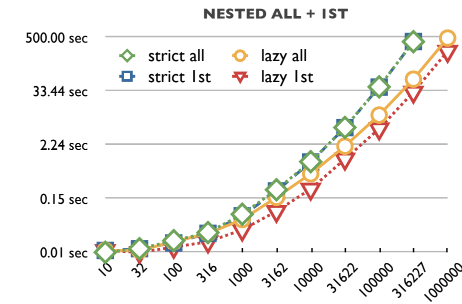
LINEAR LAST

N	STRICT	LAZY	PARSEC
10	0.014	0.011	0.011
32	0.011	0.011	0.013
100	0.015	0.015	0.017
316	0.025	0.025	0.028
1000	0.055	0.047	0.065
3162	0.176	0.127	0.186
10000	0.614	0.39	0.579
31622	2.565	1.285	1.853
100000	13.594	4.395	5.907
316227	103.752	15.299	19.182
1000000	1005.715	60.389	62.645



NESTED ALL + 1ST

N	STRICT ALL	LAZY ALL	STRICT 1ST	LAZY 1ST
10	0.01	0.011	0.011	0.011
32	0.012	0.011	0.012	0.01
100	0.018	0.017	0.016	0.013
316	0.027	0.026	0.026	0.017
1000	0.068	0.052	0.063	0.031
3162	0.233	0.159	0.218	0.079
10000	0.957	0.52	0.937	0.242
31622	5.348	2.088	5.303	1.097
100000	41.25	10.001	40.839	5.012
316227	400.36	60.854	403.932	31.86
1000000		480.729		247.842



FOLLOW

N	STRICT	LAZY	PARSEC
10	0.01	0.01	0.011
32	0.012	0.011	0.011
100	0.016	0.013	0.016
316	0.027	0.022	0.027
1000	0.061	0.045	0.066
3162	0.219	0.144	0.236
10000	0.907	0.541	0.958
31622	5.043	2.735	4.787
100000	40.813	18.354	34.251
316227	400.88	158.205	302.285
1000000		1501.799	2845.978

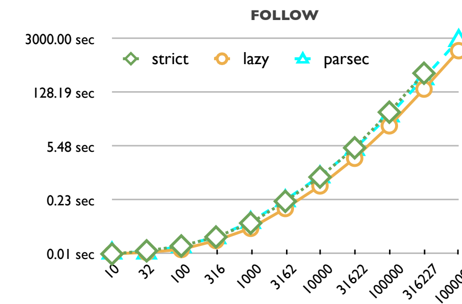
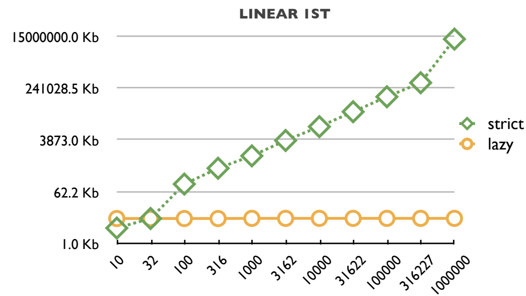


Table 1. Time performance results, measured on a twin-core 2.3GHz PowerPC G5, with 2Gb physical RAM. All timings are best-of-three, measured in seconds by the unix time command (user+system). The graph plots use a log-log scale. Blank entries indicate stack overflow.

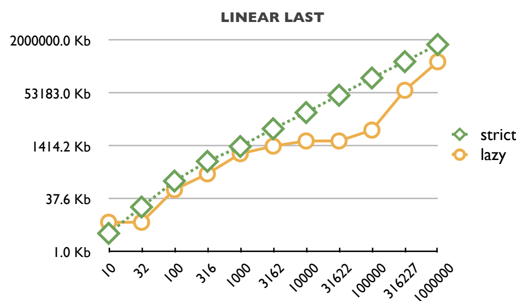
LINEAR 1ST

N	STRICT	LAZY
10	3.6	7.7
32	7.7	7.7
100	120	7.7
316	423	7.7
1000	1137	7.8
3162	3872	7.8
10000	11664	7.8
31622	38360	7.8
100000	126535	7.8
316227	386172	7.8
1000000	12539803	7.8



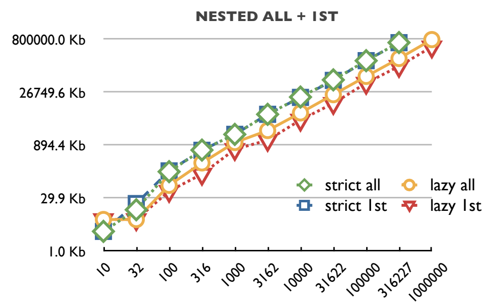
LINEAR LAST

N	STRICT	LAZY
10	3.6	7.7
32	21.9	7.7
100	130	72
316	501	213
1000	1389	848
3162	4689	1430
10000	14266	2032
31622	46595	2032
100000	152593	4284
316227	468579	65557
1000000	1514577	464335



NESTED ALL + 1ST

N	STRICT ALL	LAZY ALL	STRICT 1ST	LAZY 1ST
10	3.6	7.7	3.6	7.7
32	14.4	7.7	21.4	7.7
100	166	69	174	47.4
316	662	291	653	142
1000	1812	1077	1818	694
3162	6562	2324	6567	1250
10000	19876	7193	19445	4562
31622	59790	23181	57809	13293
100000	204901	74874	205069	48605
316227	654928	235890	650622	143416
1000000		792254		498492



FOLLOW

N	STRICT	LAZY
10	3.7	7.8
32	30	7.8
100	169	78.2
316	596	338
1000	1811	1074
3162	6562	3082
10000	17112	8713
31622	51169	30134
100000	204901	96290
316227	577197	288569
1000000		979541

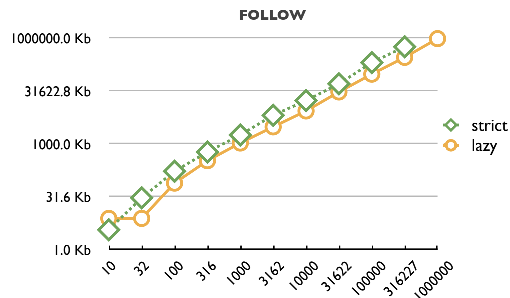


Table 2. Memory performance results. All measurements are of peak live heap usage, measured in kilobytes. The graph plots use a log-log scale.

where a more strict approach hits physical limitations. One such demonstration is given in the field of isosurface extraction for visualisation [2], where the pure lazy solution in Haskell is slower than a rival C++ implementation, only until very large inputs are considered, beyond which the Haskell overtakes the C++.

5.2 Comparisons

How does lazy parsing fare against other combinator libraries? Parsec claims to be "industrial-strength" and very fast. In contrast, the combinators presented here are somewhat simplistic, with no particular tuning for speed. So for comparison, we reimplemented our XML parser using Parsec: some selected measurements are incorporated in Table 1. Indeed, Parsec is in general faster than our strict library, but slower than our lazy library. Depending on the nature of the test, Parsec's performance aligns pretty closely to either the strict or lazy variations. Nevertheless, laziness always wins hugely when it is able to reduce a linear search to constant time.

The Utrecht combinators claim to be both partial and even faster than Parsec, so we also attempted to reimplement our XML parser in this framework too, to take advantage of the laziness. Unfortunately, the Utrecht library is entirely applicative in nature. Thus, it was not possible to implement the context-sensitive monadic parser needed for XML. (The accompanying paper [4] does give an illustrative instance of monad, but the real implementation of the library is so far removed from the paper's simplified presentation that it proved too difficult to translate.)

6 Conclusion

The main contribution of this paper is a demonstration that partial parsing is both possible, and convenient, using a framework with a mixture of monadic and applicative parser combinators. Applicative sequence is used for lazy sequencing, and monadic bind for strict sequence.

The decision on where a grammar should be strict and where lazy, is left to the programmer. This differs from the only other extant library to deliver partial parsing [4], which can automatically analyse the grammar to determine where laziness is possible.

As expected, the resources needed to partially parse a document depend on how much of the input document is consumed, not on the total size of the document. Nevertheless, if the whole document is demanded, it is still cheaper to parse it lazily than strictly.

However, partial parsing also means that the ability to report parse errors is shifted from within the parsing framework out to the world of exception handling.

A secondary contribution is the re-discovery of the *commit* combinator to prevent backtracking and enable both better error-reporting and space-efficiency. Although *commit* was previously known [9] to remove a particular space leak

associated with choice, the impact on error-reporting was not so widely appreciated. Parsec's *try*, as a dual to *commit*, is more commonly used for this purpose, but is rather less useful due to the need for a correct manual analysis of the grammar for lookahead, and the difficulty of doing this. By contrast, placement of *commit* is not required for correctness, only for efficiency, and the manual analysis involved is easy.

References

1. A. Baars, A. Löh, and D. Swierstra. Parsing permutation phrases. *Journal of Functional Programming*, 14(6):635–646, 2004.
2. D. Duke, M. Wallace, R. Borgo, and C. Runciman. Fine-grained visualization pipelines and lazy functional languages. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):973–980, Sept 2006.
3. B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *International Conference on Functional Programming*, Pittsburgh, October 2002. ACM SIGPLAN.
4. R. J. M. Hughes and S. D. Swierstra. Polish parsers, step by step. In *Proceedings of ICFP*, pages 239–248, Uppsala, 2003. ACM Press.
5. G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.
6. G. Hutton and E. Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, University of Nottingham, 1996.
7. D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, University of Utrecht, 2001.
8. C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 17(5):1–13, 2007.
9. N. Röjemo. *Garbage collection and memory efficiency in lazy functional languages*. PhD thesis, Chalmers University of Technology, 1995.
10. D. Swierstra. *Combinator Parsers: from toys to tools*, volume 41 of *ENTCS*. Elsevier, 2001.
11. T. van Laarhoven. ParseP: software distribution. http://twan.home.fmf.nl/parsep/.
12. P. Wadler. Monads for functional programming. In M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and systems sciences*. Springer-Verlag, August 1992.
13. M. Wallace. HaXml software distribution. http://haskell.org/HaXml.
14. M. Wallace. Polyparse combinators. http://www.cs.york.ac.uk/fp/polyparse, 2007.
15. M. Wallace and C. Runciman. Haskell and XML: generic combinators or type-based translation? In *Proceedings of ICFP*, Paris, 1999. ACM Press.