Lazy Functional Incremental Parsing

Jean-Philippe Bernardy

Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg

bernardy@chalmers.se

Abstract

Structured documents are commonly edited using a free-form editor. Even though every string is an acceptable input, it makes sense to maintain a structured representation of the edited document. The structured representation has a number of uses: structural navigation (and optional structural editing), structure highlighting, etc. The construction of the structure must be done incrementally to be efficient: the time to process an edit operation should be proportional to the size of the change, and (ideally) independent of the total size of the document.

We show that combining lazy evaluation and caching of intermediate (partial) results enables incremental parsing. We build a complete incremental parsing library for interactive systems with support for error-correction.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; D.2.3 [*Coding Tools and Techniques*]: Program editors; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

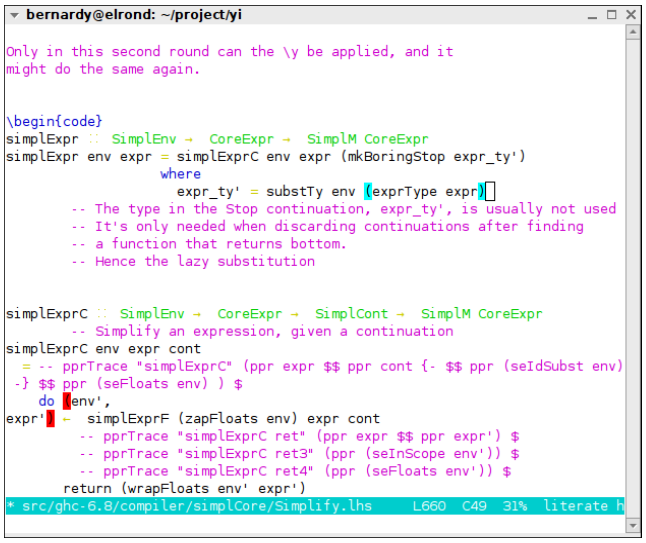
General Terms Algorithms, Languages, Design, Performance, Theory

Keywords Lazy evaluation, Incremental Computing, Parsing, Dynamic Programming, Polish representation, Editor, Haskell

1. Introduction

Yi (Bernardy, 2008; Stewart and Chakravarty, 2005) is a text editor written in Haskell. It provides features such as syntax highlighting and indentation hints for a number of programming languages (figure 1). All syntax-dependent functions rely on the abstract syntax tree (AST) of the source code being available at all times. The feedback given by the editor is always consistent with the text: the AST is kept up to date after each modification. But, to maintain acceptable performance, the editor must not parse the whole file at each keystroke: we have to implement a form of incremental parsing.

Another feature of Yi is that it is configurable in Haskell. Therefore, we prefer to use the Haskell language for every aspect of the application, so that the user can configure it. In particular, syntax is described using a combinator library.



```
Only in this second round can the \y be applied, and it
might do the same again.


\begin{code}
simplExpr :: SimplEnv → CoreExpr → SimplM CoreExpr
simplExpr env expr = simplExprC env expr (mkBoringStop expr_ty')
                   where
                     expr_ty' = substTy env (exprType expr)
        -- The type in the Stop continuation, expr_ty', is usually not used
        -- It's only needed when discarding continuations after finding
        -- a function that returns bottom.
        -- Hence the lazy substitution


simplExprC :: SimplEnv → CoreExpr → SimplCont → SimplM CoreExpr
        -- Simplify an expression, given a continuation
simplExprC env expr cont
  = -- pprTrace "simplExprC" (ppr expr $$ ppr cont {- $$ ppr (seIdSubst env)
 -} $$ ppr (seFloats env) ) $
    do (env',
expr') ← simplExprF (zapFloats env) expr cont
         -- pprTrace "simplExprC ret" (ppr expr $$ ppr expr') $
         -- pprTrace "simplExprC ret3" (ppr (seInScope env')) $
         -- pprTrace "simplExprC ret4" (ppr (seFloats env')) $
       return (wrapFloats env' expr')
* src/ghc-6.8/compiler/simplCore/Simplify.lhs     L660  C49  31%  literate h
```

Figure 1. Screenshot. The user has opened a very big Haskell file. Yi gives feedback on matching parenthesis by changing the background color. Even though the file is longer than 2000 lines, real-time feedback can be given as the user types, because parsing is performed incrementally.

Our main goals can be formulated as constraints on the parsing library:

- it must be programmable through a combinator interface;
- it must cope with all inputs provided by the user, and thus provide error correction;
- it must be efficient enough for interactive usage: parsing must be done incrementally.

To implement this last point, one could choose a stateful approach and update the parse tree as the user modifies the input structure. Instead, in this paper we explore the possibility to use a more "functional" approach: minimize the amount of state that has to be updated, and rely as much as possible on laziness to implement incrementality.

1.1 Approach

In this section we sketch how lazy evaluation can help achieve incremental parsing.

An *online* parser exhibits lazy behavior: it does not proceed further than necessary to return the nodes of the AST that are demanded. Assuming that, in addition to using an online parser to produce the AST, it is traversed in pre-order to display the decorated text

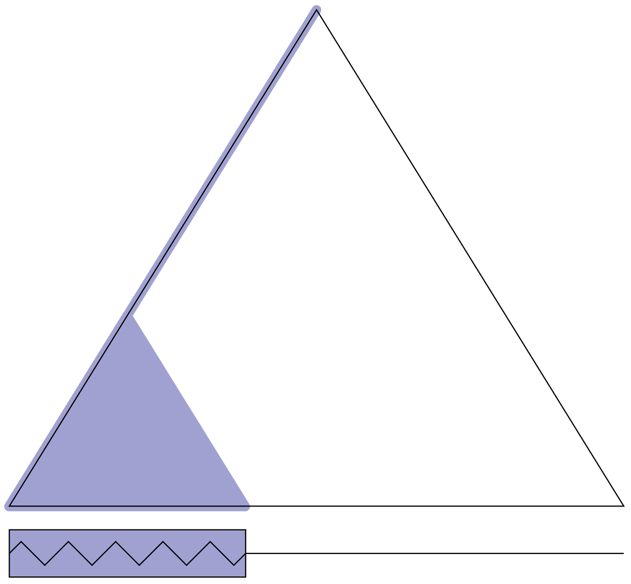


Figure 2. Viewing the beginning of a file. The big triangle represents the syntax tree. The line at the bottom represents the file. The zagged part indicates the part that is parsed. The viewing window is depicted as a rectangle.

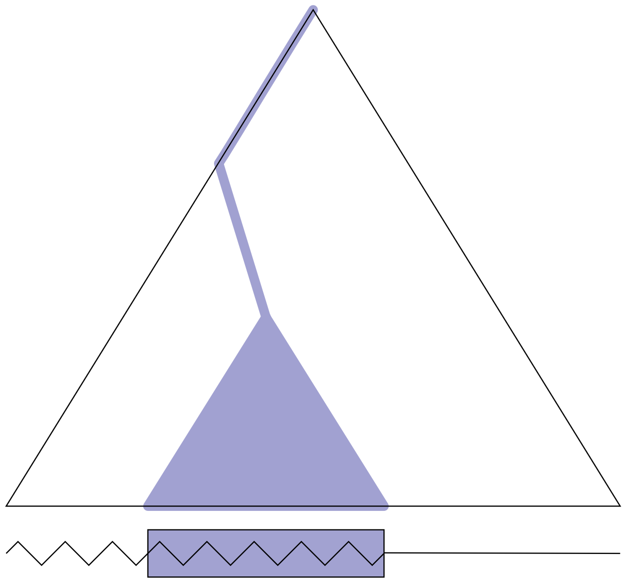


Figure 3. Viewing the middle of a file. Parsing proceeds in linear fashion: although only a small amount of the parse tree may be demanded, it will depend not only on the portion of the input that corresponds to it, but also on everything that precedes.

presented to the user, the situation right after opening a file is depicted in figure 2. The window is positioned at the beginning of the file. To display the decorated output, the program has to traverse the first few nodes of the syntax tree (in pre-order). This traversal in turn forces parsing the corresponding part of the input, but, thanks to lazy evaluation, *no further* (or maybe a few tokens ahead, depending on the amount of look-ahead required). If the user modifies the input at this point, it invalidates the AST, but discarding it and re-parsing is not too costly: only a screenful of parsing needs to be re-done.

As the user scrolls down in the file, more and more of the AST is demanded, and the parsing proceeds in lockstep (figure 3). At this stage, a user modification is more serious: re-parsing naively from the beginning can be too costly for a big file. Fortunately we can again exploit the linear behavior of parsing algorithms to our advantage. Indeed, if the editor stores the parser state for the input point where the user made the modification, we can *resume* parsing from that point. Furthermore, if it stores partial results for every point of the input, we can ensure that we will never parse more than a screenful at a time. Thereby, we achieve incremental parsing, in the sense that the amount of parsing work needed after each user interaction depends only on the size of the change or the length of the move.

1.2 Contributions

Our contributions can be summarized as follows.

- We describe a novel, purely functional approach to incremental parsing, which makes essential use of lazy evaluation;
- We complete our treatment of incremental parsing with error correction. This is essential, since online parsers need to be *total*: they cannot fail on any input;
- We have implemented such a system in a parser-combinator library and made use of it to provide syntax-dependent feedback in a production-quality editor.

1.3 Interface and Outlook

Our goal is to provide a combinator library with a standard interface, similar to that presented by Swierstra (2000).

Such an interface can be captured in a generalized algebraic data type (GADT, Xi et al. (2003)) as follows. These combinators are traditionally given as functions instead of constructors, but since we make extensive use of GADTs for modeling purposes at various levels, we prefer to use this presentation style everywhere for consistency. (Sometimes mere ADTs would suffice, but we prefer to spell out the types of the combinators explicitly, using the GADT syntax.)

```
data Parser s a where
  Pure  :: a                                      → Parser s a
  (:*:) :: Parser s (b → a) → Parser s b          → Parser s a
  Symb  :: Parser s a → (s → Parser s a)          → Parser s a
  Disj  :: Parser s a → Parser s a                → Parser s a
  Fail  ::                                          Parser s a
```

This interface supports production of results (*Pure*), sequencing (:∗:), reading of input symbols (*Symb*), and disjunction (*Disj*, *Fail*). The type parameter s stands for the type of input symbols, while a is the type of values produced by the parser.

Most of this paper is devoted to uncovering an appropriate representation for our parsing process type, and the implementation of the functions manipulating it. The core of this representation is introduced in section 3, where we merely handle the *Pure* and (:∗:) constructors. Dependence on input and the constructor *Symb* are treated in section 4. Disjunction and error correction will be implemented as a refinement of these concepts in section 5.

Parsing combinator libraries usually propose a mere *run* function that executes the parser on a given input: $run :: Parser\ s\ a \rightarrow [s] \rightarrow Either\ Error\ a$. Incremental systems require finer control over the execution of the parser. Therefore, we have to split the *run* function into pieces and reify the parser state in values of type *Process*.

We also need a few functions to create and manipulate the parsing processes:

- *mkProcess* :: *Parser s a* → *Process s a*: given a parser description, create the corresponding initial parsing process.
- *feed* :: [*s*] → *Process s a* → *Process s a*: feed the parsing process a number of symbols.
- *feedEof* :: *Process s a* → *Process s a*: feed the parsing process the end of the input.
- *precompute* :: *Process s a* → *Process s a*: transform a parsing process by pre-computing all the intermediate parsing results available.
- *finish* :: *Process s a* → *a*: compute the final result of the parsing, in an online way, assuming that the end of input has been fed into the process.

Section 2 details our approach to incrementality by sketching the main loop of an editor using the above interface. The implementation for these functions can be given as soon as we introduce dependence on input in section 4.

Sections 3 through 5 describe how our parsing machinery is built, step by step. In section 6 we discuss the problem of incremental parsing of the repetition construct. We discuss and compare our approach to alternatives in section 7 through section 10 and conclude in section 11.

2. Main loop

In this section we write an editor using the interface described in section 1.3. This editor lacks most features one would expect from a real application, and is therefore just a toy. It is however a self-contained implementation which tackles the issues related to incremental parsing.

The main loop alternates between displaying the contents of the file being edited and updating its internal state in response to user input. Notice that we make our code polymorphic over the type of the AST we process, merely requiring it to be *Show*-able.

```
loop :: Show ast ⇒ State ast → IO ()
loop s = display s ≫ update s ≫= loop
```

The *State* structure stores the "current state" of our toy editor.

```
data State ast = State
  {
    lt, rt :: String,
    ls :: [Process Char ast]
  }
```

The fields *lt* and *rt* contain the text respectively to the left and to the right of the edit point. The field *ls* is our main interest: it contains the parsing processes corresponding to each symbol to the left of the edit point. The left-bound lists, *lt* and *ls*, contain data in reversed order, so that the information next to the cursor corresponds to the head of the lists. Note that there is always one more element in *ls* than in *lt*, because we also have a parser state for the empty input.

We do not display the input document as typed by the user, but an enriched version, to hightlight syntactic constructs. Therefore, we have to parse the input and then serialize the result. First, we feed the remainder of the input to the current state and then run the online parser. The display is then trimmed to show only a window around the edit point. Trimming takes a time proportional to the position in the file, but for the time being we assume that displaying is much faster than parsing and therefore the running time of the former can be neglected.

```
display :: (Show ast) ⇒ State ast → IO ()
display s@State {ls = pst : _} = do
  putStrLn ""
  putStrLn $ trimToWindow
           $ show
           $ finish
           $ feedEof
           $ feed (rt s)
           $ pst
  where trimToWindow = take windowSize ∘
                         drop windowBegin
        windowSize = 10   -- arbitrary size
        windowBegin = length (lt s) − windowSize
```

There are three types of user input to take care of: movement, deletion and insertion of text. The main difficulty here is to keep the list of intermediate states synchronized with the text. For example, every time a character is typed, a new parser state is computed and stored. The other editing operations proceed in a similar fashion.

```
update :: State ast → IO (State ast)
update s@State {ls = pst : psts} = do
  c ← getChar
  return $ case c of
      -- cursor movements
    '<' → case lt s of    -- left
        []       → s
        (x : xs) → s {lt = xs, rt = x : rt s, ls = psts}
    '>' → case rt s of    -- right
        []       → s
        (x : xs) → s {lt = x : lt s, rt = xs
                     , ls = addState x}
      -- deletions
    ',' → case lt s of    -- backspace
        []       → s
        (x : xs) → s {lt = xs, ls = psts}
    '.' → case rt s of    -- delete
        []       → s
        (x : xs) → s {rt = xs}
      -- insertion of text
    c    → s {lt = c : lt s, ls = addState c}
  where addState c = precompute (feed [c] pst) : ls s
```

Besides disabling buffering of the input for real-time response, the top-level program has to instantiate the main loop with an initial state, and pick a specific parser to use: *parseTopLevel*.

```
main = do hSetBuffering stdin NoBuffering
          loop State {
            lt = "",
            rt = "",
            ls = [mkProcess parseTopLevel]}
```

As we have seen before, the top-level parser can return any type. In sections 4 and 5 we give examples of parsers for S-expressions, which can be used as instances of *parseTopLevel*.

We illustrate using S-expressions because they have a recursive structure which can serve as prototype for many constructs found in programming languages, while being simple enough to be treated completely within this paper.

```
data SExpr = S [SExpr] | Atom Char
```

The code presented in this section forms the skeleton of any program using our library. A number of issues are glossed over though. Notably, we would like to avoid re-parsing when moving in the file if no modification is made. Also, the displayed output is computed from its start, and then trimmed. Instead we would like to directly print the portion corresponding to the current window. Doing this is tricky to fix: the attempt described in section 6 does not tackle the general case.

3. Producing results

Hughes and Swierstra (2003) show that the sequencing operator must be applicative (McBride and Paterson (2007)) to allow for online production of results. This result is the cornerstone of our approach to incremental parsing, so we review it in this section, justifying the use of the combinators *Pure* and (:∗:), which form the applicative sub-language.

We also introduce the *Polish representation* for applicative expressions: it is the essence of our parsing semantics. This section culminates in the definition of the pipeline from applicative language to results by going through Polish expressions. Our final parser (section 5) is an extension of this machinery with all the features mentioned in the introduction.

A requirement for online production of the result is that nodes are available before their children are computed. In terms of datatypes, this means that constructors must be available before their arguments are computed. This can only be done if the parser can observe (pattern match on) the structure of the result. Hence, we make function applications explicit in the expression describing the results.

For example, the Haskell expression `S [Atom 'a']`, which stands for `S ((:) (Atom 'a') [])` if we remove syntactic sugar, can be represented in applicative form by using @ for applications.

```
S@((:)@(Atom@'a')@[])
```

The following data type captures a pure applicative language embedding Haskell values. It is indexed by the type of values it represents.

```
data Applic a where
  (:∗:) :: Applic (b → a) → Applic b → Applic a
  Pure :: a                          → Applic a
infixl 4 :∗:
```

The application annotations can then be written using Haskell syntax as follows:

```
Pure S :∗: (Pure (:) :∗: (Pure Atom :∗: Pure 'a')
                     :∗: Pure [])
```

We can also write a function for evaluation:

```
evalA :: Applic a → a
evalA (f :∗: x)  = (evalA f) (evalA x)
evalA (Pure a) = a
```

If the arguments to the *Pure* constructor are constructors, then we know that demanding a given part of the result forces only the corresponding part of the applicative expression.

Because our parsers process the input in a linear fashion, they require a linear structure for the output as well. (This is revisited in section 5). As Hughes and Swierstra (2003), we convert the applicative expressions to their Polish representation to obtain such a linear structure.

The key idea of the Polish representation is to put the application in a prefix position rather than an infix one. Our example expression (in applicative form `S@((:)@(Atom@'a')@[])`) becomes `@S (@(@(:) (@Atom 'a')) [])`

Since @ is always followed by exactly two arguments, grouping information can be inferred from the applications, and the parentheses can be dropped. The final Polish expression is therefore

```
@S@@(:)@Atom 'a' []
```

The Haskell datatype can also be linearized in the same way. Using *App* for @, *Push* to wrap values and *Done* to finish the expression, we obtain the following representation.

```
App $ Push S $ App $ App $ Push (:) $
  App $ Push Atom $ Push 'a' $ Push [] $ Done
```

```
data Polish where
  Push :: a → Polish → Polish
  App  :: Polish       → Polish
  Done ::                Polish
```

Unfortunately, the above datatype does not allow to evaluate expressions in a typeful manner. The key insight is that Polish expressions are in fact more general than applicative expressions: they represent a stack of values instead of a single one.

As hinted by the constructor names we chose, we can reinterpret Polish expressions as follows. *Push* produces a stack with one more value than its second argument, *App* transforms the stack produced by its argument by applying the function on the top to the argument on the second position and pushing back the result. *Done* produces the empty stack.

The expression `Push (:) $ App $ Push Atom $ Push 'a' $ Push [] $ Done` is an example producing a non-trivial stack. It produces the stack `(:), (Atom 'a'), []`, which can be expressed purely in Haskell as `(:) :< Atom 'a' :< [] :< Nil`, using the following representation for heterogeneous stacks.

```
data top :< rest = (:<) {top :: top, rest :: rest}
data Nil = Nil
infixr 4 :<
```

We are now able to properly type Polish expressions, by indexing the datatype with the type of the stack produced.

```
data Polish r where
  Push :: a → Polish r                → Polish (a :< r)
  App  :: Polish ((b → a) :< b :< r) → Polish (a :< r)
  Done ::                               Polish Nil
```

We can also write a translation from the pure applicative language to Polish expressions.

```
toPolish :: Applic a → Polish (a :< Nil)
toPolish expr = toP expr Done
  where toP :: Applic a → (Polish r → Polish (a :< r))
        toP (f :∗: x)  = App ∘ toP f ∘ toP x
        toP (Pure x) = Push x
```

And the value of an expression can be evaluated as follows:

```
evalR :: Polish r → r
evalR (Push a r) = a :< evalR r
evalR (App s)    = apply (evalR s)
  where apply∼(f :< ∼(a :< r)) = f a :< r
evalR (Done)     = Nil
```

We have the equality `evalR (toPolish x) ≡ evalA x :< Nil`.

Additionally, we note that this evaluation procedure still possesses the "online" property: prefixes of the Polish expression are demanded only if the corresponding parts of the result are demanded. This preserves the incremental properties of lazy evaluation that we required in the introduction. Furthermore, the equality above holds even when $\bot$ appears as argument to the *Pure* constructor. In fact, the conversion from applicative to Polish expressions can be understood as a reification of the working stack of the *evalA* function with call-by-name semantics.

4. Adding input

While the study of the pure applicative language is interesting in its own right (we come back to it in section 4.1), it is not enough to represent parsers: it lacks dependency on the input.

We introduce an extra type argument (the type of symbols, s), as well as a new constructor: *Symb*. It expresses that the rest of the expression depends on the next symbol of the input (if any): its first argument is the parser to be used if the end of input has been reached, while its second argument is used when there is at least one symbol available, and it can depend on it.

```
data Parser s a where
  Pure :: a                                   → Parser s a
  (:*:) :: Parser s (b → a) → Parser s b      → Parser s a
  Symb :: Parser s a → (s → Parser s a) → Parser s a
```

Using just this, as an example, we can write a simple parser for S-expressions.

```
parseList :: Parser Char [SExpr]
parseList = Symb
  (Pure [])
  (λc → case c of
    ')' → Pure []
    ' ' → parseList   -- ignore spaces
    '(' → Pure (λh t → S h : t) :*: parseList
           :*: parseList
    c   → Pure ((Atom c):) :*: parseList)
```

We adapt the *Polish* expressions with the construct corresponding to *Symb*, and amend the translation. Intermediate results are represented by a Polish expression with a *Susp* element. The part before the *Susp* element corresponds to the constant part that is fixed by the input already parsed. The arguments of *Susp* contain the continuations of the parsing algorithm: the first one if the end of input is reached, the second one when there is a symbol to consume.

```
data Polish s r where
  Push :: a → Polish s r                  → Polish s (a :< r)
  App  :: Polish s ((b → a) :< b :< r)    → Polish s (a :< r)
  Done ::                                   Polish s Nil
  Susp :: Polish s r → (s → Polish s r) → Polish s r

toP :: Parser s a → (Polish s r → Polish s (a :< r))
toP (Symb nil cons) =
  λk → Susp (toP nil k) (λs → toP (cons s) k)
toP (f :*: x)  = App ∘ toP f ∘ toP x
toP (Pure x) = Push x
```

Although we broke the linearity of the type, it does no harm since the parsing algorithm will not proceed further than the available input anyway, and therefore will stop at the first *Susp*. Suspensions in a Polish expression can be resolved by feeding input into it. When facing a suspension, we pattern match on the input, and choose the corresponding branch in the result.

The *feed* function below performs this duty for a number of symbols, and stops when it has no more symbols to feed. The dual function, *feedEof*, removes all suspensions by consistently choosing the end-of-input alternative.

```
feed :: [s] → Polish s r → Polish s r
feed []       p               = p
feed (s : ss) (Susp nil cons) = feed ss (cons s)
feed ss       (Push x p)      = Push x (feed ss p)
feed ss       (App p)         = App    (feed ss p)
feed ss       Done            = Done

feedEof :: Polish s r → Polish s r
feedEof (Susp nil cons) = feedEof nil
feedEof (Push x p)      = Push x (feedEof p)
feedEof (App p)         = App    (feedEof p)
feedEof Done            = Done
```

For example, *evalR* \$ *feedEof* \$ *feed* `"(a)"` \$ *toPolish* \$ *parseList* yields back our example expression: *S* [*Atom* `'a'`].

We recall from section 2 that feeding symbols one at a time yields all intermediate parsing results.

```
allPartialParses = scanl (λp c → feed [c] p)
```

If the $(n+1)^{th}$ element of the input is changed, one can reuse the n^{th} element of the partial results list and feed it the new input's tail (from that position).

This suffers from a major issue: partial results remain in their "Polish expression form", and reusing offers little benefit, because no part of the result value is shared between the partial results: the function *evalR* has to perform the the full computation for each of them. Fortunately, it is possible to partially evaluate prefixes of Polish expressions.

The following function performs this task by traversing a Polish expression and applying functions along the way.

```
evalL :: Polish s a → Polish s a
evalL (Push x r) = Push x (evalL r)
evalL (App f) = case evalL f of
  (Push g (Push b r)) → Push (g b) r
  r → App r
evalL x = x
partialParses = scanl (λp c → evalL ∘ feed [c] $ p)
```

This still suffers from a major drawback: as long as a function application is not saturated, the Polish expression will start with a long prefix of partial applications, which has to be traversed again in forthcoming partial results.

For example, after applying the S-expression parser to the string `abcdefg`, *evalL* is unable to perform any simplification of the list prefix:

```
evalL $ feed "abcdefg" (toPolish parseList)
  ≡ App $ Push (Atom 'a':) $
    App $ Push (Atom 'b':) $
    App $ Push (Atom 'c':) $
    App $ ...
```

This prefix will persist until the end of the input is reached. A possible remedy is to avoid writing expressions that lead to this sort of intermediate result, and we will see in section 6 how to do this in the particularly important case of lists. This however works only up to some point: indeed, there must always be an unsaturated application (otherwise the result would be independent of the input). In general, after parsing a prefix of size n, it is reasonable to expect a partial application of at least depth $O(\log n)$, otherwise the parser is discarding information.

4.1 Zipping into Polish

In this section we develop an efficient strategy to pre-compute intermediate results. As seen in the above section, we want to avoid the cost of traversing the structure up to the suspension at each step. This suggests to use a zipper structure (Huet, 1997) with the focus at the suspension point.

```
data Zip s out where
  Zip :: RPolish stack out → Polish s stack → Zip s out
data RPolish inp out where
  RPush :: a → RPolish (a :< r) out →
           RPolish r out
  RApp  :: RPolish (b :< r) out →
           RPolish ((a → b) :< a :< r) out
  RStop :: RPolish r r
```

Since the data is linear, this zipper is very similar to the zipper for lists. The part that is already visited ("on the left"), is reversed. Note that it contains only values and applications, since we never go past a suspension.

The interesting features of this zipper are its type and its meaning. We note that, while we obtained the data type for the left part by mechanically inverting the type for Polish expressions, it can be assigned a meaning independently: it corresponds to *reverse* Polish expressions.

In contrast to forward Polish expressions, which directly produce an output stack, reverse expressions can be understood as automata which transform a stack to another. This is captured in the type indices inp and out, which stand respectively for the input and the output stack.

Running this automaton requires some care: matching on the input stack must be done lazily. Otherwise, the evaluation procedure will force the spine of the input, effectively forcing to parse the whole input file.

```
evalRP :: RPolish inp out → inp → out
evalRP RStop acc          = acc
evalRP (RPush v r) acc = evalRP r (v :< acc)
evalRP (RApp r)~(f :< ~(a :< acc))
                          = evalRP r (f a :< acc)
```

In our zipper type, the Polish expression yet-to-visit ("on the right") has to correspond to the reverse Polish automation ("on the left"): the output of the latter has to match the input of the former.

Capturing all these properties in the types (though GADTs) allows to write a properly typed traversal of Polish expressions. The $right$ function moves the focus by one step to the right.

```
right :: Zip s out → Zip s out
right (Zip l (Push a r)) = Zip (RPush a l) r
right (Zip l (App r))    = Zip (RApp l) r
right (Zip l s)          = Zip l s
```

As the input is traversed, in the implementation of $precompute$, we also simplify the prefix that we went past, evaluating every application, effectively ensuring that each $RApp$ is preceded by at most one $RPush$.

```
simplify :: RPolish s out → RPolish s out
simplify (RPush a (RPush f (RApp r))) =
  simplify (RPush (f a) r)
simplify x = x
```

We see that simplifying a complete reverse Polish expression requires $O(n)$ steps, where n is the length of the expression. This means that the *amortized* complexity of parsing one token (i.e. computing a partial result based on the previous partial result) is $O(1)$, if the size of the result expression is proportional to the size of the input. We discuss the worst case complexity in section 6.

In summary, it is essential for our purposes to have two evaluation procedures for our parsing results. The first one, presented in section 3, provides the online property, and corresponds to call-by-name CPS transformation of the direct evaluation of applicative expressions. It underlies the *finish* function in our interface. The second one, presented in this section, enables incremental evaluation of intermediate results, and corresponds to a call-by-value transformation of the same direct evaluation function. It underlies the $precompute$ function.

5. Adding Choice

We kept the details of actual parsing out of the discussion so far. This is for good reason: the machinery for incremental computation and reuse of partial results is independent from such details. Indeed, given any procedure to compute structured values from a linear input of symbols, one can use the procedure described above to transform it into an incremental algorithm.

However, parsing the input string with the interface presented so far is highly unsatisfactory. To support convenient parsing, we can introduce a disjunction operator, exactly as Hughes and Swierstra (2003) do: the addition of the *Susp* operator does not undermine their treatment of disjunction in any way.

5.1 Error correction

Disjunction is not very useful unless coupled with *failure* (otherwise any branch would be as good as another). Still, the (unrestricted) usage of failure is problematic for our application: the online property requires at least one branch to yield a successful outcome. Indeed, since the *evalR* function *must* return a result (we want a total function!), the parser must conjure up a suitable result for *any* input.

If the grammar is sufficiently permissive, no error correction in the parsing library itself is necessary. An example is the simple S-expression parser of section 4, which performs error correction in an ad-hoc way. However, most interesting grammars produce a highly structured result, and are correspondingly restrictive on the input they accept. Augmenting the parser with error correction is therefore desirable.

Our approach is to add some rules to accept erroneous inputs. These will be marked as less desirable by enclosing them with $Yuck$ combinators, introduced as another constructor in the $Parser$ type. The parsing algorithm can then maximize the desirability of the set of rules used for parsing a given fragment of input.

```
data Parser s a where
  Pure :: a                                      → Parser s a
  (:*:) :: Parser s (b → a) → Parser s b → Parser s a
  Symb :: Parser s a → (s → Parser s a) → Parser s a
  Disj  :: Parser s a → Parser s a           → Parser s a
  Yuck :: Parser s a                             → Parser s a
```

5.2 Example

In this section we rewrite our parser for S-expressions from section 4 using disjunction and error-correction. The goal is to illustrate how these new constructs can help in writing more modular parser descriptions.

First, we can define repetition and sequence in the traditional way:

```
many, some :: Parser s a → Parser s [a]
many v = some v `Disj` Pure []
some v = Pure (:) :*: v :*: many v
```

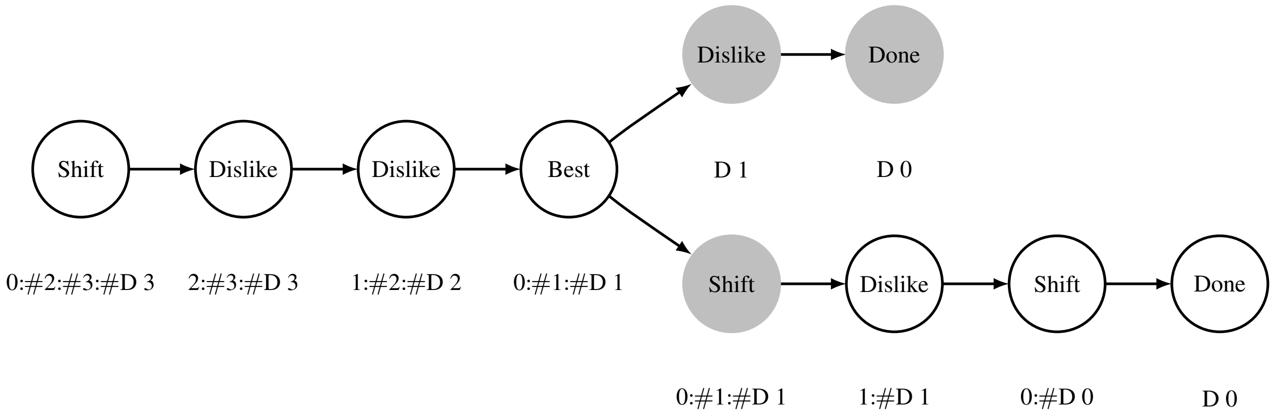


Figure 4. A parsing process and associated progress information. The process has been fed a whole input, so it is free of *Susp* constructors. It is also stripped of result information (*Push*, *App*) for conciseness, since it is irrelevant to the computation of progress information. Each constructor is represented by a circle, and their arguments are indicated by arrows. The progress information associated with the process is written below the node that starts the process. To decide which path to take at the disjunction (*Best*), only the gray nodes will be forced, if the desirability difference is 1 for look-ahead 1.

Checking for the end of file can be done as follows. Notice that if the end of file is not encountered, we keep parsing the input, but complain while doing so.

```
eof = Symb (Pure ()) (λ_ → Yuck eof)
```

Checking for a specific symbol can be done in a similar way: we accept anything but dislike (*Yuck*!) anything unexpected.

```
pleaseSymbol :: Eq s ⇒ s → Parser s (Maybe s)
pleaseSymbol s = Symb
  (Yuck $ Pure Nothing)
  (λs' → if s ≡ s' then Pure (Just s')
    else Yuck $ Pure (Just s'))
```

All of the above can be combined to write the parser for S-expressions. Note that we need to amend the result type to accommodate for erroneous inputs.

```
data SExpr
    = S [SExpr] (Maybe Char)
    | Atom Char
    | Missing
    | Deleted Char
parseExpr = Symb
  (Yuck $ Pure Missing)
  (λc → case c of
      '(' → Pure S :*: many parseExpr :*: pleaseSymbol ')'
      ')' → Yuck $ Pure $ Deleted ')'
      c   → Pure $ Atom c)
parseTopLevel
    = Pure const :*: parseExpr :*: eof
```

We see that the constructs introduced in this section (*Disj*, *Yuck*) permit to write general purpose derived combinators, such as *many*, in a traditional style.

5.3 The algorithm

Having defined our definitive interface for parsers, we can describe the parsing algorithm itself.

As before, we linearize the applications (:*:) by transforming the *Parser* into a Polish-like representation. In addition to the the *Dislike* and *Best* constructors corresponding to *Yuck* and *Disj*, *Shift* records where symbols have been processed, once *Susp* is removed.

```
data Polish s a where
  Push   :: a → Polish s r             → Polish s (a :< r)
  App    :: Polish s ((b → a) :< b :< r)
                                       → Polish s (a :< r)
  Done   ::                              Polish s Nil
  Shift  :: Polish s a                 → Polish s a
  Sus    :: Polish s a → (s → Polish s a)
                                       → Polish s a
  Best   :: Polish s a → Polish s a → Polish s a
  Dislike :: Polish s a                → Polish s a
toP :: Parser s a → (Polish s r → Polish s (a :< r))
toP (Pure x)     = Push x
toP (f :*: x)    = App ∘ toP f ∘ toP x
toP (Symb a f) = λfut → Sus (toP a fut)
                             (λs → toP (f s) fut)
toP (Disj a b)  = λfut → Best (toP a fut) (toP b fut)
toP (Yuck p)    = Dislike ∘ toP p
```

The remaining challenge is to amend our evaluation functions to deal with disjunction points (*Best*). It offers two *a priori* equivalent alternatives. Which one should be chosen?

Since we want online behavior, we cannot afford to look further than a few symbols ahead to decide which parse might be the best. (Performance is another motivation: the number of potential paths grows exponentially with the amount of look-ahead.) We use the widespread technique (Bird and de Moor, 1997, chapter 8) to *thin out* the search after some constant, small amount of look-ahead.

Hughes and Swierstra's algorithm searches for the best path by direct manipulation of the Polish representation, but this direct approach forces to transform between two normal forms: one where the *progress* nodes (*Shift*, *Dislike*) are at the head and one where the result nodes (*Pure*, :*:) are at the head. Therefore, we choose to use an intermediate datatype which represents the progress information only. This clear separation of concerns also enables to compile the progress information into a convenient form: our *Progress* data structure directly records how many *Dislike* are encountered after parsing so many symbols. It is similar to a list where the n^{th} element tells how much we dislike to take this path after shifting n symbols following it, *assuming we take the best choice at each disjunction*.

```
data Progress = S | D Int | Int :# Progress
```

The difference from a simple list is that progress information may end with success (D) or suspension (S), depending on whether the process reaches *Done* or *Susp*. Figure 4 shows a *Polish* structure and the associated progress for each of its parts. The *progress* function below extracts the information from the *Polish* structure.

```
progress :: Polish s r → Progress
progress (Push _ p) = progress p
progress (App p)    = progress p
progress (Shift p)  = 0 :# progress p
progress (Done)     = D 0
progress (Dislike p) = mapSucc (progress p)
progress (Susp _ _) = S
progress (Best p q) = snd $ better (progress p)
                                   (progress q)
mapSucc S = S
mapSucc (D x) = D (succ x)
mapSucc (x :# xs) = succ x :# mapSucc xs
```

To deal with the last case (*Best*), we need to find out which of two profiles is better. Using our thinning heuristic, given two *Progress* values corresponding to two terminated *Polish* processes, it is possible to determine which one is best by demanding only a prefix of each. The following function handles this task. It returns the best of two progress information, together with an indicator of which is to be chosen. Constructors LT or GT respectively indicates that the second or third argument is the best, while EQ indicates that a suspension is reached. The first argument (*lk*) keeps track of how much lookahead has been processed. This value is a parameter to our thinning heuristic, *dislikeThreshold*, which indicates when a process can be discarded.

```
better _ S _ = (EQ, S)
better _ _ S = (EQ, S)
better _ (D x) (D y) =
  if x ⩽ y then (LT, D x) else (GT, D y)
better lk xs@(D x) (y :# ys) =
  if x ≡ 0 ∨ y − x > dislikeThreshold lk
  then (LT, xs)
  else min x y +> better (lk + 1) xs ys
better lk (y :# ys) xs@(D x) =
  if x ≡ 0 ∨ y − x > dislikeThreshold lk
  then (GT, xs)
  else min x y +> better (lk + 1) ys xs
better lk (x :# xs) (y :# ys)
  | x ≡ 0 ∧ y ≡ 0      = rec
  | y − x > threshold = (LT, x :# xs)
  | x − y > threshold = (GT, y :# ys)
  | otherwise = rec
  where threshold = dislikeThreshold lk
        rec       = min x y +> better (lk + 1) xs ys
x +> ∼(ordering, xs) = (ordering, x :# xs)
```

Calling the *better* function directly is very inefficient though, because its result is needed every time a given disjunction is encountered. If the result of a disjunction depends on the result of further disjunction, the result of the further disjunction will be needlessly discarded. Therefore, we cache the result of *better* in the *Polish* representation, using the well known technique of *tupling*. For simplicity, we cache the information only at disjunction nodes, where we also remember which path is best to take. We finally see why the *Polish* representation is important: the progress information cannot be associated to a *Parser*, because it may depend on whatever parser *follows* it. This is not an issue in the *Polish* representation, because applications (:∗:) are unfolded.

We now have all the elements to write our final data structures and algorithms. The following code shows the final construction procedure. In the *Polish* datatype, only the *Best* constructor is amended.

```
data Polish s a where
  ...
  Best :: Ordering → Progress →
          Polish s a → Polish s a → Polish s a


toP :: Parser s a → (Polish s r → Polish s (a :< r))
toP (Symb a f) = λfut → Susp (toP a fut)
                   (λs → toP (f s) fut)
toP (f :∗: x)   = App ∘ toP f ∘ toP x
toP (Pure x)    = Push x
toP (Disj a b)  = λfut → mkBest (toP a fut) (toP b fut)
toP (Yuck p)    = Dislike ∘ toP p
mkBest :: Polish s a → Polish s a → Polish s a
mkBest p q =
  let (choice, pr) = better 0 (progress p) (progress q)
  in  Best choice pr p q
```

The evaluation functions can be easily adapted to support disjunction by querying the result of *better*, cached in the *Best* constructor. We write the the online evaluation only: partial result computation is modified similarly.

```
evalR :: Polish s r → r
evalR Done              = Nil
evalR (Push a r)        = a :< evalR r
evalR (App s)           = apply (evalR s)
  where apply∼(f :< ∼(a :< r)) = f a :< r
evalR (Shift v)         = evalR v
evalR (Dislike v)       = evalR v
evalR (Susp _ _)        = error "input pending"
evalR (Best choice _ p q) = case choice of
  LT → evalR p
  GT → evalR q
  EQ → error "Suspension reached"
```

Note that this version of *evalR* expects a process without any pending suspension (the end of file must have been reached). In this version we also disallow ambiguity, see section 5.5 for a discussion.

5.4 Summary

We have given a convenient interface for constructing error-correcting parsers, and functions to evaluate them. This is performed in steps: first we linearize applications into *Polish* (as in section 4), then we linearize disjunctions (*progress* and *better*) into *Progress*. The final result is computed by traversing the *Polish* expressions, using *Progress* to choose the better alternative in disjunctions.

Our technique can also be re-formulated as lazy dynamic programming, in the style of Allison (1992). We first define a full tree of possibilities (Polish expressions with disjunction), then we compute progress information that we tie to it, for each node; finally, finding the best path is a matter of looking only at a subset of the information we constructed, using any suitable heuristic. The cut-off heuristic makes sure that only a part of the exponentially growing data structure is demanded. Thanks to lazy evaluation, only that small part will be actually constructed.

5.5 Thinning out results and ambiguous grammars

A sound basis for thinning out less desirable paths is to discard those which are less preferable by some amount. In order to pick one path after a constant amount of look-ahead l, we must set this difference to 0 when comparing the l^{th} element of the progress information, so that the parser can pick a particular path, and return results. Unfortunately, applying this rule strictly is dangerous if the grammar requires a large look-ahead, and in particular if it is ambiguous. In that case, the algorithm can possibly commit to a prefix which will lead to errors while processing the rest of the output, while another prefix would match the rest of the input and yield no error. In the present version of the library we avoid the problem by keeping all valid prefixes. The user of the parsing library has to be aware of this issue when designing grammars: it can affect the performance of the algorithm to a great extent, by triggering an exponential explosion of possible paths.

6. Eliminating linear behavior

As we noted in section 4, the result of some computations cannot be pre-computed in intermediate parser states, because constructors are only partially applied.

This is indeed a common case: if the constructed output is a list, then the spine of the list can only be constructed once we get hold of the very tail of it.

For example, our parser for S-expressions would produce such lists for flat expressions, because the applications of (:) can be computed only when the end of the input is reached.

$$
\begin{aligned}
&\mathit{evalL}\ \$\ \mathit{feed}\ \texttt{"(abcdefg"}\ (\mathit{toPolish}\ \mathit{parseList})\\
&\quad\equiv \mathit{App}\ \$\ \mathit{Push}\ (\mathit{Atom}\ \texttt{'a'}:)\ \$\\
&\qquad \mathit{App}\ \$\ \mathit{Push}\ (\mathit{Atom}\ \texttt{'b'}:)\ \$\\
&\qquad \mathit{App}\ \$\ \mathit{Push}\ (\mathit{Atom}\ \texttt{'c'}:)\ \$\\
&\qquad \mathit{App}\ \$\ ...
\end{aligned}
$$

Section 4.1 explained how to optimize the creation of intermediate results, by skipping this prefix. Unfortunately this does not improve the asymptotic performance of computing the final result. The partial result corresponding to the end of input contains the long chain of partial applications (in reverse Polish representation), and to produce the final result the whole prefix has to be traversed.

Therefore, in the worst case, the construction of the result has a cost proportional to the length of the input.

While the above example might seem trivial, the same result applies to all repetition constructs, which are common in language descriptions. For example, a very long Haskell file is typically constituted of a very long list of declarations, for which a proportional cost must be paid every time the result is constructed.

The culprit for linear complexity is the linear shape of the list. Fortunately, nothing forces to use such a structure: it can always be replaced by a tree structure, which can then be traversed in pre-order to discover the elements in the same order as in the corresponding list. Wagner and Graham (1998, section 7) recognize this issue and propose to replace left or right recursive rules in the parsing with a special repetition construct. The parsing algorithm treats this construct specially and does re-balancing of the tree as needed. We choose a different approach: only the result type is changed, not the parsing library. We can do so for two reasons:

- Combinators can be parametrized by arbitrary values

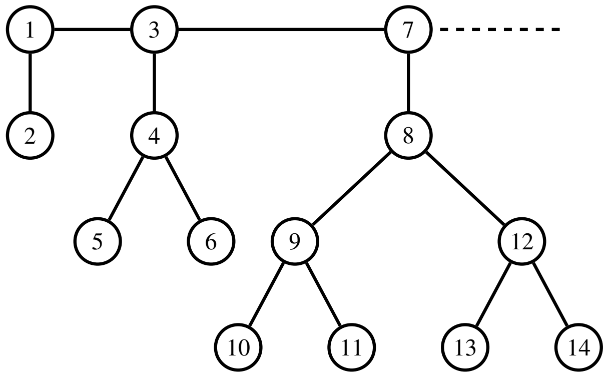


Figure 5. A tree storing the elements 1 ... 14. Additional elements would be attached to the right child of node 7: there would be no impact on the tree constructed so far.

- Since we do not update a tree, but produce a fresh version every time, we need not worry about re-balancing issues.

Let us summarize the requirements we put on the data structure:

- It must provide the same laziness properties as a list: accessing an element in the structure should not force to parse the input further than necessary if we had used a list.
- the n^{th} element in pre-order should not be further away than $O(log\ n)$ elements from the root of the structure. In other words, if such a structure contains a suspension in place of an element at position n, there will be no more than $O(log\ n)$ partial applications on the stack of the corresponding partial result. This in turn means that the resuming cost for that partial result will be in $O(log\ n)$.

The second requirement suggests a tree-like structure, and the first requirement implies that whether the structure is empty or not can be determined by entering only the root constructor. It turns out that a simple binary tree can fulfill these requirements.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Tree}\ a = \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\\
&\qquad\qquad\quad\mid\ \mathit{Leaf}
\end{aligned}
$$

The only choice that remains is the size of the sub-trees. The specific choice we make is not important as long as we make sure that each element is reachable in $O(log\ n)$ steps. A simple choice is a series of complete trees of increasing depth. The k^{th} tree will have depth k and contain $2^k - 1$ nodes. For simplicity, all these sub-trees are chained using the same data type: they are attached as the left child of the spine of a right-leaning linear tree. Such a structure is depicted in figure 5.

We note that a complete tree of total depth $2d$ can therefore store at least $\sum_{k=1}^{d} 2^k - 1$ elements, fulfilling the second requirement.

This structure is very similar to binary random access lists as presented by Okasaki (1999, section 6.2.1), but differ in purpose. The only construction primitive presented by Okasaki is the appending of an element. This is of no use to us, because the function has to analyze the structure it is appending to, and is therefore strict. We want avoid this, and thus must construct the structure in one go. Indeed, the construction procedure is the only novel idea we introduce:

$$
\begin{aligned}
&\mathit{toTree}\ d\ [\,] \qquad\ \ = \mathit{Leaf}\\
&\mathit{toTree}\ d\ (x : xs) = \mathit{Node}\ x\ l\ (\mathit{toTree}\ (d+1)\ xs')\\
&\quad \mathbf{where}\ (l, xs') = \mathit{toFullTree}\ d\ xs
\end{aligned}
$$

```
toFullTree 0 xs        = (Leaf, xs)
toFullTree d []        = (Leaf, [])
toFullTree d (x : xs) = (Node x l r, xs'')
  where (l, xs')  = toFullTree (d − 1) xs
        (r, xs'') = toFullTree (d − 1) xs'
```

In other words, we must use a special construction function to guarantee the online production of results: we want the argument of *Pure* to be in a simple value (not an abstraction), as explained in section 3. In fact, we will have to construct the list directly in the parser.

The following function implements such a parser where repeated elements are mere symbols.

```
parseTree d = Symb
  (Pure Leaf)
  (λs → Pure (Node s) :∗:
     parseFullTree d :∗:
     parseTree (d + 1))
parseFullTree 0 = Pure Leaf
parseFullTree d = Symb
  (Pure Leaf)
  (λs → Pure (Node s) :∗:
     parseFullTree (d − 1) :∗:
     parseTree (d − 1))
```

The function can be adapted for arbitrary non-terminals. One has to take care to avoid interference between the construction of the shape and error recovery. For example, the position of non-terminals can be forced in the tree, as to be in the node corresponding to the position of their first symbol. In that case the structure has to be accommodated for nodes not containing any information.

6.1 Quick access

Another benefit of using the tree structure as above is that finding the part of the tree of symbols corresponding to the edit window also takes logarithmic time. Indeed, the size of each sub-tree depends only on its relative position to the root. Therefore, one can access an element by its index without pattern matching on any node which is not the direct path to it. This allows efficient indexed access without loosing any property of laziness. Again, the technique can be adapted for arbitrary non-terminals. However, it will only work if each node in the tree is "small" enough. Finding the first node of interest might force an extra node, and in turn force parsing the corresponding part of the file.

7. Related work

The literature on parsing, incremental or not, is so abundant that a comprehensive survey would deserve its own treatment. Here we will compare our approach to some of the closest alternatives.

7.1 Development environments

The idea of incremental analysis of programs is not new. Wilcox et al. (1976) already implemented such a system. Their program works very similarly to ours: parsing states to the left of the cursor are saved so that changes to the program would not force a complete re-parse. A big difference is that it does not rely on built-in lazy evaluation. If they had produced an AST, its online production would have had to be managed entirely by hand. The system also did not provide error correction nor analysis to the right of the cursor.

Ghezzi and Mandrioli (1979) improved the concept by reusing parsing results to the right of the cursor: after parsing every symbol they check if the new state of the LR automaton matches that of the previous run. If it does they know that they can reuse the results from that point on.

This improvement offers some advantages over Wilcox et al. (1976) which still apply when compared to our solution.

1. In our system, if the user jumps back and forth between the beginning and the end of the file, every forward jump will force re-parsing the whole file. Note that we can mitigate this drawback by caching the (lazily constructed) whole parse tree: a full re-parse is required only when the user makes a change while viewing the beginning of the file.
2. Another advantage is that the AST is fully constructed at all times. In our case only the part to the left of the window is available. This means that the functions that traverse the AST should do so in pre-order. If this is not the case, the online property becomes useless. For example, if one wishes to apply a sorting algorithm before displaying an output, this will force the whole input to be parsed before displaying the first element of the input. In particular, the arguments to the *Pure* constructor must not perform such operations on its arguments. Ideally, they should be simple constructors. This leaves much risk for the user of the library to destroy its incremental properties.

While our approach is much more modest, it can be considered better in some respects.

1. One benefit of not analyzing the part of the input to the right of the cursor is that there is no start-up cost: only a screenful of text needs to be parsed to start displaying it.
2. Another important point is that a small change in the input might completely invalidate the result from the previous parsing run. A simple example is the opening of a comment: while editing an Haskell source file, typing `{-` implies that the rest of the file becomes a comment up to the next matching `-}`.

   It is therefore questionable that reusing right-bound parts of the parse tree offers any reasonable benefit in practice: it seems to be optimizing for a special case. This is not very suitable in an interactive system where users expect consistent response times.
3. Finally, our approach accommodate better to a combinator implementation. Indeed, comparing parser states is very tricky to accomplish in the context of a combinator library: since parsing states normally contain lambda abstractions, it is not clear how they can be compared to one another.

Wagner and Graham (1998) improved on the state-matching technique. They contributed the first incremental parser that took in account the inefficiency of linear repetition. We compared our approach to theirs in section 6.

Despite extensive research dating as far back as 30 years ago, these solutions have barely caught up in the mainstream. Editors typically work using regular expressions for syntax highlighting at the lexical level (Emacs, Vim, Textmate, . . .).

It is possible that the implementation cost of earlier solutions outweighed their benefits. We hope that the simplicity of our approach will permit more widespread application.

7.2 Incremental computation

An alternative to our approach to would be to build the library as a plain parser on top of a generic incremental computation system. The main drawback is that there currently exists no such off-the-shelf system for Haskell. The closest matching solution is provided by Carlsson (2002), and relies heavily on explicit threading of computation through monads and explicit reference for storage of inputs and intermediate results. This imposes an imperative description of the incremental algorithm, which does not match our goals. Furthermore, in the case of parsing, the inputs would be the individual symbols. This means that, not only their contents will change from one run to another, but their numbers will as well. One then might want to rely on laziness, as we do, to avoid depending unnecessarily on the tail of the input, but then we hit the problem that the algorithm must be described imperatively. Therefore, we think that such an approach would be awkward, if at all applicable.

7.3 Parser combinators

Our approach is firmly anchored in the tradition of parser combinator libraries (Hutton and Meijer, 1998), and particularly close to the Polish parsers of Hughes and Swierstra (2003), which were recently refined by Swierstra (2009).

The introduction of the *Susp* operator is directly inspired by the parallel parsing processes of Claessen (2004), which features a very similar construct to access the first symbol of the input and make it accessible to the rest of the computation. This paper presents our implementation as a version of Polish parsers extended with an evaluation procedure "by-value", but we could equally have started with parallel parsing processes and extended them with "by-name" evaluation. The combination of both evaluation techniques is unique to our library.

Our error correction mechanism bears many similarities with that presented by Swierstra and Alcocer (1999): they also associate some variant of progress information to parsers and rely on thinning and laziness to explore the tree of all possible parses. An important difference is that we embed the error reports in the tree instead of returning them as a separate tree. This is important, because we need to highlight errors in a lazy way. If the errors we reported separately, merely checking if an error is present could force parsing the whole file.

Wallace (2008) presents another, simpler approach to online parsing, based on the notion of *commitment*. His library features two sequencing combinators: the classic monadic bind, and a special application with commitment. The former supports backtracking in the classic way, but the latter decouples errors occurring on its left-hand side from errors occurring on its right-hand side: if there are two possible ways to parse the left-hand side, the parser chooses the first match. This scheme therefore relies on user annotations at determined points in the production of the result to prune the search tree, while we prune after the same amount of lookahead in all branches. This difference explains why we need to linearize the applications, while it can be avoided in Wallace's design. Additionally, we take advantage of the linear shape of the parsing process to to feed it with partial inputs, so we cannot spare the linearization phase. A commitment combinator would be a useful addition to our library though: pruning the search tree at specific point can speed up the parsing and improve error-reporting.

8. Discussion

Due to our choice to commit to a purely functional, lazy approach, our incremental parsing library occupies a unique point in the design space.

It is also the first time that incremental and online parsing are both available in a combinator library.

What are the advantages of using the laziness properties of the online parser? Our system could be modified to avoid relying on laziness at all. In section 4.1 we propose to apply the reverse Polish automaton (on the left) to the stack produced — lazily — by the Polish expression (on the right). Instead of that stack, we could feed the automaton with a stack of dummy values, or $\perp$s. Everything would work as before, except that we would get exceptions when trying to access unevaluated parts of the tree. If we know in advance how much of the AST is consumed, we could make the system work as such.

One could take the stance that this guesswork (knowing where to stop the parsing) is practically possible only for mostly linear syntaxes, where production of output is highly coupled with the consumption of input. Since laziness essentially liberates us from any such guesswork, the parser can be fully decoupled from the functions using the syntax tree.

The above reflexion offers another explanation why most mainstream syntax highlighters are based on regular-expressions or other lexical analysis mechanism: they lack a mechanism to decouple processing of input from production of output.

The flip side to our approach is that the efficiency of the system crucially depends on the lazy behavior of consumers of the AST. One has to take lots of care in writing them.

9. Future work

Our treatment of repetition is still lacking: we would like to retrieve any node by its position in the input while preserving all properties of laziness intact. While this might be very difficult to do in the general case, we expect that our zipper structure can be used to guide the retrieval of the element at the current point of focus, so that it can be done efficiently.

Although it is trivial to add a *failure* combinator to the library presented here, we refrained from doing so because it can lead to failing parsers. Of course, one can use our $Yuck$ combinator in place of failure, but one has to take in account that the parser continues running after the $Yuck$ occurrence. In particular, many $Yuck$s following each other can lead to some performance loss, as the "very disliked" branch would require more analysis to be discarded than an immediate failure. Indeed, if one takes this idea to the extreme and tries to use the fix-point ($fix\ Yuck$) to represent failure, it will lead to non-termination. This is due to our use of strict integers in the progress information. We have chosen this representation to emphasize the dynamic programming aspect of our solution, but in general it might be more efficient to represent progress by a mere interleaving of $Shift$ and $Dislike$ constructors.

Our library suffers from the usual drawbacks of parser combinator approaches. In particular, it is impossible to write left-recursive parsers, because they cause a non-terminating loop in the parsing algorithm. We could proceed as Baars et al. (2009) and transform the grammar to remove left-recursion. It is interesting to note however that we could represent traditional left-recursive parsers as long as they either consume or *produce* data, provided the progress information is indexed by the number of $Push$es in addition to $Shift$s.

Finally, we might want to re-use the right hand side of previous parses. This could be done by keeping the parsing results *for all possible prefixes*. Proceeding in this fashion would avoid the chaotic situation where a small modification might invalidate all the parsing work that follows it, since we take in account *all* possible prefixes ahead of time.

10. Results

We carried out development of a parser combinator library for incremental parsing with support for error correction. We argued that, using suitable data structures for the output, the complexity of parsing (without error correction) is $O(log\ m + n)$ where m is the number of tokens in the state we resume from and n is the number of tokens to parse. Parsing an increment of constant size has an amortized complexity of $O(1)$. These complexity results ignore the time to search for the nodes corresponding to the display window.

The parsing library presented in this paper is used in the Yi editor to help matching parenthesis and layout the Haskell functions, and environment delimiters as well as parenthetical symbols were matched in the LaTeX source. This paper and the accompanying source code have been edited in Yi.

11. Conclusion

We have shown that the combination of a few simple techniques achieve the goal of incremental parsing.

1. In a lazy setting, the combination of online production of results and saving intermediate results provide incrementality;
2. The efficient computation of intermediate results requires some care: a zipper-like structure is necessary to improve performance.
3. Online parsers can be extended with an error correction scheme for modularity.
4. Provided that they are carefully constructed to preserve laziness, tree structures can replace lists in functional programs. Doing so can improve the complexity class of algorithms.

While these techniques work together here, we believe that they are valuable independently of each other. In particular, our error correction scheme can be replaced by another one without invalidating the approach.

Acknowledgments

We thank Koen Claessen for persuading us to write this paper, and for his unfading support throughout the writing process. This paper was greatly improved by his comments on early and late drafts. Discussions with Krasimir Angelov helped sorting out the notions of incremental parsing. Patrik Jansson, Wouter Swierstra, Gustav Munkby, Marcin Zalewski and Michał Pałka and the anonymous reviewers of ICFP gave helpful comments on the presentation of the paper. Finally, special thanks go to the reviewers of the Haskell Symposium for their extremely helpful comments.

References

L. Allison. Lazy Dynamic-Programming can be eager. *Information Processing Letters*, 43(4):207–212, 1992.

A. Baars, D. Swierstra, and M. Viera. Typed transformations of typed abstract syntax. In *TLDI '09: fourth ACM SIGPLAN Workshop on Types in Language Design and Implementation*, New York, NY, USA, 2009.

J. Bernardy. Yi: an editor in Haskell for Haskell. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 61–62, Victoria, BC, Canada, 2008. ACM.

R. Bird and O. de Moor. *Algebra of programming*. Prentice-Hall, Inc., 1997.

M. Carlsson. Monads for incremental computing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 26–35, Pittsburgh, PA, USA, 2002. ACM.

K. Claessen. Parallel parsing processes. *Journal of Functional Programming*, 14(6):741–757, 2004.

C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Trans. Program. Lang. Syst.*, 1(1):58–70, 1979.

G. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

R. J. M. Hughes and S. D. Swierstra. Polish parsers, step by step. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 239–248, Uppsala, Sweden, 2003. ACM.

G. Hutton and E. Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(04):437–444, 1998.

C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2007.

C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, July 1999.

D. Stewart and M. Chakravarty. Dynamic applications from the ground up. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 27–38. ACM Press, 2005.

S. D. Swierstra. Combinator parsers: From toys to tools. *Electronic Notes in Theoretical Computer Science*, 41(1), 2000.

S. D. Swierstra. Combinator parsing: A short tutorial. In *Language Engineering and Rigorous Software Development*, volume 5520 of *LNCS*, pages 252–300, Piriapolis, 2009. Springer.

S. D. Swierstra and P. R. A. Alcocer. Fast, error correcting parser combinators: A short tutorial. In *Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics*, pages 112–131. Springer-Verlag, 1999.

T. A. Wagner and S. L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems*, 20(5):980–1013, 1998.

M. Wallace. *Partial Parsing: Combining Choice with Commitment*, volume 5083/2008 of *LNCS*, pages 93–110. Springer Berlin / Heidelberg, 2008.

T. R. Wilcox, A. M. Davis, and M. H. Tindall. The design and implementation of a table driven, interactive diagnostic programming system. *Commun. ACM*, 19(11):609–616, 1976.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. *SIGPLAN Not.*, 38(1):224–235, 2003.

Appendix: The complete code

The complete code of the library described in this paper can be found at: `http://github.com/jyp/topics/tree/master/FunctionalIncrementalParsing/Code.lhs` The Yi source code is constantly evolving, but at the time of this writing it uses a version of the parsing library which is very close to the descriptions given in the paper. It can be found at: `http://code.haskell.org/yi/Parser/Incremental.hs`