Packrats Parse in Packs

Mario Blažević
Jacques Légaré
Stilo International plc
Ottawa, Canada

Abstract

We present a novel but remarkably simple formulation of formal language grammars in Haskell as functions mapping a record of production parsers to itself. Thus formulated grammars are first-class objects, composable and reusable. We also provide a simple parser implementation for them, based on an improved packrat algorithm. In order to make the grammar manipulation code reusable, we introduce a set of type classes mirroring the existing type classes from Haskell base library, but whose methods have rank-2 types.

CCS Concepts • **Theory of computation → Grammars and context-free languages**; **Parsing**; • **Software and its engineering** → *Domain specific languages*;

Keywords Haskell, parser combinators, memoizing, packrat, rank-2 types

ACM Reference Format:
Mario Blažević and Jacques Légaré. 2017. Packrats Parse in Packs. In *Proceedings of 10th ACM SIGPLAN International Haskell Symposium, Oxford, UK, September 7-8, 2017 (Haskell'17),* 12 pages.
https://doi.org/10.1145/3122955.3122958

1 Introduction

Ever since Wadler [33] introduced the notion of parser combinators[1], they have been a mainstay of lazy higher-order functional programming. A number of practical parser combinator libraries for various functional languages have been developed over the years, starting with Parsec [24] and going strong ever since [30] [6] [27] [22].

Apart from the Icon and Snobol family, imperative languages offer, at best, two exclusive ways to parse inputs. For short inputs and simple languages, they may offer regular expressions as a library or a language extension. For long inputs and complex grammars, the only possibility is to use a parser generator like Yacc [23], producing a black box of unreadable code that is not to be messed with.

Parser combinator libraries, in contrast, occupy a middle ground between the two solutions on offer and encroach on both. At the short end of the scale, they are a bit more verbose but much more readable than regular expressions, and remain manageable as they grow. At the other end, a parser of a complex language described using combinators can be as readable as an input grammar for a parser generator, if not as performant as its output. Most importantly, this parser is defined, manipulated, and used as a first-class object in the language. This is doubtless one of the most convincing examples of the power and versatility of functional programming.

A gap, however, still remains between the parser combinators and grammars: a parser-grammar impedance mismatch, to stretch an already strained metaphor.

The central feature shared by all parser combinator libraries is that their combinators operate on, well, parsers. Primitive parsers are combined into more complex parsers until the parser for the whole language is constructed, but every parser is an opaque value that can be used only in two ways: to parse some input or to be combined into a larger parser.

Regular expressions notwithstanding, the syntax of any but the most trivial of languages is generally specified as a formal grammar consisting of a set mutually recursive productions, following Chomsky [5] or Backus [1] or Ford [17]. The relationship between different productions can be followed through these explicit mutual references. A grammar can be analyzed; it can be split into constituent productions and reassembled into a different form.

This semantic gap can be bridged by collapsing the multitude of grammar productions into a single parser value, replacing the grammar-level references by unobservable host-language references. Unfortunately this transformation is one-way, as the productions cannot be separated out of the collapsed parser expression. In contrast to parsers, grammars are not first-class objects.

This paper aims to bring grammars into the fold. Below is the mandatory arithmetic grammar [2] in Haskell, formulated anew:

```
arithmeticGrammar :: TextualMonoid s
  ⇒ Arithmetic (Parser g s) → Arithmetic (Parser g s)
arithmeticGrammar ~Arithmetic {..} = Arithmetic
  {additive   = (+) ⟨$⟩ multitive ⟨∗ string "+" ⟨∗⟩ additive
                ⟨|⟩ multitive,
   multitive  = (∗) ⟨$⟩ primary ⟨∗ string "*" ⟨∗⟩ multitive
                ⟨|⟩ primary,
   primary    = string "(" ∗⟩ additive ⟨∗ string ")"
                ⟨|⟩ read ⟨$⟩ digits,
   digits     = Textual.toString (const "")
                  ⟨$⟩ takeCharsWhile1 isDigit}
```

Somebody unfamiliar with Haskell could ignore the semantic and applicative operators and read this definition as a somewhat verbose context-free grammar. Its alternatives are ordered in such a way that it can also pass as a parsing expression grammar.

A different person might look at the four field assignments as if they were standalone definitions, ignoring the weird record construction, and decide that this is yet another parser definition using one of the many parsing combinator libraries in Haskell. This is

[1]The paper calls them *combinations* rather than combinators.

[2]Well, only a half of an arithmetic grammar. Subtraction and division are left out because they'd require the left-associativity, which is a long-solved problem [33] we don't need to revisit here.

not completely wrong: if we apply the standard function *fix* from the `base` library to *arithmeticGrammar*, in fact, we'll get a standard bundle of mutually recursive parsers.

This, however, is not our only option. The purpose of the record and the function that maps the record type onto itself is to make the recursive field references observable and gain new abilities to manipulate grammars. One such ability that this paper elaborates is that instead of *fix* we can apply the function *fixGrammar* that produces packrat-style parsers, using memoization to ensure $O(n)$ parsing time.

1.1 Contributions

The above definition, rather surprisingly, appears to be the first published example of a grammar defined as a pure function from a record of parsers to itself. This representation allows a seamless extension of the well-known applicative and monadic parsers to multi-production grammars with observable production references.

The grammar we have defined here is a first-class object; it can be combined with other grammars either in part or as a whole.

The particular variant of the packrat parser uses a novel modification of the original algorithm that improves its performance on large grammars and long tokens.

The above example uses the *Arithmetic* {..} pattern enabled by the *RecordWildCards* language extension. We shall use this extension in all our grammar definitions to trim down the boilerplate code, but this is a mere convenience. The only language extension we require to the Haskell 2010 standard is *RankNTypes*. To atone for this requirement, we demonstrate some new applications of rank-2 types.

2 Expanding the Packrat Midden

We shall build our grammars by gradually modifying the standard parsing expression grammars (PEGs) [17] parsed using the packrat algorithm [16]. The benefit of this starting point is that the algorithm is very compact and we don't need to worry about complications like left recursion associated with the standard context-free grammars.

2.1 The Packrat Parsing Algorithm

At its simplest, a packrat parser for arithmetic expressions can be represented by the following example in Haskell, adapted from Ford [16]:

```
data Result v = Parsed {parsedPrefix :: v,
                        parsedSuffix :: Derivs}
              | NoParse

data Derivs = Derivs {dvAdditive  :: Result Int,
                      dvMultitive :: Result Int,
                      dvPrimary   :: Result Int,
                      dvDigits    :: Result [Char],
                      dvChar      :: Result Char}

parse :: String → Derivs
parse s = d where
  d = Derivs {dvAdditive  = pAdditive d,
              dvMultitive = pMultitive d,
              dvPrimary   = pPrimary d,
              dvDigits    = pDigits d,
              dvChar      = chr}
  chr = case s
        of (c : s′) → Parsed c (parse s′)
           []       → NoParse
```

The two mutually-recursive abstract data types *Result* and *Derivs* are respectively used to contain the parsing results for a single grammar production and for the entire grammar. Every *Parsed* value contains the parsing result for an input prefix as well as *Derivs* for the remaining suffix of the input.

The function *parse* relies on Haskell's non-strict evaluation strategy to set up a circular definition of *Derivs d* at every input position. The field *dvChar* of *d* is special, as it provides the only direct access to the input. Every other field of *d* depends on the whole of *d*, passed as argument to the corresponding parser function. Each parser function encodes the right-hand side of a grammar production; every non-terminal reference translates to a pattern match of the corresponding *d* field against a *Parsed* value.

```
pAdditive, pMultitive, pPrimary :: Derivs → Result Int
pDigits :: Derivs → Result String
pAdditive d
    -- Additive → Multitive '+' Additive
  | Parsed vleft d′      ← dvMultitive d,
    Parsed '+' d″        ← dvChar d′,
    Parsed vright rest ← dvAdditive d″
      = Parsed (vleft + vright) rest
    -- Additive → Multitive
  | Parsed v rest ← dvMultitive d = Parsed v rest
  | otherwise = NoParse
pMultitive d
    -- Multitive → Primary '*' Multitive
  | Parsed vleft d′      ← dvPrimary d,
    Parsed '*' d″        ← dvChar d′,
    Parsed vright rest ← dvMultitive d″
      = Parsed (vleft ∗ vright) rest
    -- Multitive → Primary
  | Parsed v d′ ← dvPrimary d = Parsed v d′
  | otherwise = NoParse
pPrimary d
    -- Primary → '(' Additive ')'
  | Parsed '(' d′    ← dvChar d,
    Parsed v d″      ← dvAdditive d′,
    Parsed ')' rest ← dvChar d″ = Parsed v rest
    -- Primary → Digits
  | Parsed v d′ ← dvDigits d = Parsed (read v) d′
  | otherwise = NoParse
pDigits d
    -- Digits → [0 − 9] Digits?
  | Parsed c d′ ← dvChar d, '0' ⩽ c ∧ c ⩽ '9'
    = case dvDigits d′
      of Parsed digits d″ → Parsed (c : digits) d″
         NoParse          → Parsed [c] d′
  | otherwise = NoParse
```

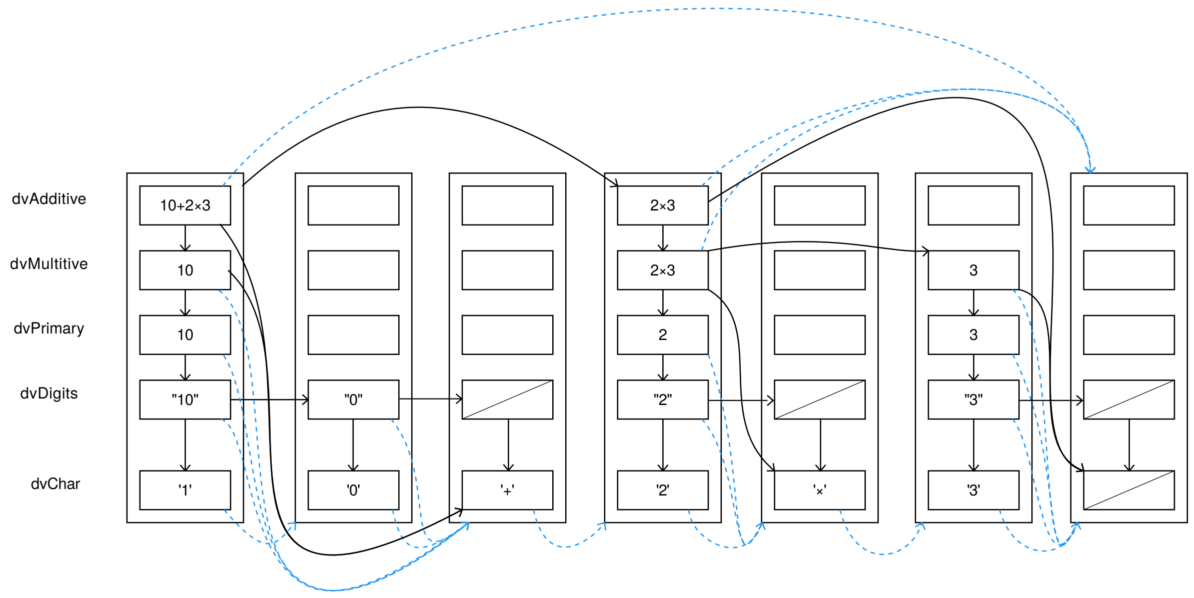

Figure 1. packrat data structure for input "10+2*3"

The diagram in figure 1 shows the entire data structure built up by the expression *parse* `"10+2*3"`. The expression evaluates to the leftmost *Derivs* in the diagram. The dashed lines represent the *parsedSuffix* field values, while the solid lines show the data flow, *i.e.*, the forcings of sub-expression evaluations during the parse. The three solid lines leaving the *dvAdditive* field in the head *Derivs*, for example, illustrate that through the parser function *pAdditive* the field depends on the *dvMultitive* field of the same record as well as *dvChar* and *dvAdditive* from other records.

The empty boxes in the diagram represent the values whose evaluation has not been forced, while the stricken-through fields have been evaluated to *NoParse*. Fully lazy evaluation strategy means that we never evaluate the same field (*i.e.*, try to match the same grammar production) at the same position twice. An important consequence of this property is that the parsing time is guaranteed to be linear against the input length.

2.2 Primitive Parsers and Non-Terminals on an Equal Footing

The above grammar definition is in many ways superior to what non-memoizing backtracking parser combinator libraries like Parsec [24] can provide. For example, a direct transcription of *pAdditive* with Parsec might look like

```
pAdditive = try (do left ← pMultitive
                    char '+'
                    right ← pAdditive
                    return (left + right))
            ⟨|⟩ pMultitive
```

In case of Parsec, *pMultitive* is not memoized, which means that *pAdditive* has to backtrack and parse *pMultitive* for the second time if its first alternative fails. In this case it would be easy enough to left-factor *pMultitive* and thus work around the problem, but the avoidance of backtracking is not always that trivial. Furthermore, such optimizations often obscure the grammar to human readers even as they make it easier for the machine to process.

On the other hand, Parsec and other combinator libraries perform better at the lexical level. A packrat parser has access only to a single character at a time, and to access that character it has to construct the entire *Derivs* container. Consider the above definition of *pDigits*. With Parsec, one could replace it with a simple

```
pDigits = read ⟨$⟩ some digit
```

Parsec enables this by giving its parsers the whole remainder of the input and letting them consume any prefix of it. Of course, this is also the reason it has to backtrack when a match only partially succeeds.

Can we have the best of both worlds? What if every parser had access not only to the current character but the entire rest of the input? Let's try to write *pDigits* that way:

```
pDigits :: String → Derivs → Result String
pDigits s d = case takeWhile isDigit s
              of  [ ]    → NoParse
                  digits → Parsed digits (error "what now?")
```

We're stuck. To properly fill in the *Parsed* constructor on the last line, we need *Derivs* corresponding to the point following the digits. The only way we can possibly obtain the right *Derivs* value is if we pick it out of a list of all *Derivs* at every point in the rest of the input. So the top function *parse* has to build a list of all *Derivs*, paired with the corresponding rest of the input, and to pass the list to every parser. Furthermore, to allow the linking of any two such parsers together, each parser has to return the same data structure. Once we satisfy these necessities, we arrive at the following definitions:

```
data Result v = Parsed {parsedPrefix :: v,
                        parsedSuffix :: [(String, Derivs)]}
              | NoParse
pDigits :: [(String, Derivs)] → Result String
pDigits ((s, d) : tail)
    = case takeWhile isDigit s
      of  [ ]     → NoParse
          digits → Parsed digits (drop (length digits − 1) tail)
```

It may seem that this formulation maintains and passes around a lot more data than the original packrat algorithm, but in fact, it's only making the inherent data dependencies more explicit. Looking at figure 1, it's obvious that all *Derivs* values already form an implicit list at least through the *dvChar* field that always points to the next *Derivs*. We can easily collect them all into an actual list data structure within the root function *parse*:

```
parse :: String → [ (String, Derivs) ]
parse s = parsed where
  parsed = (s, d) : parsedTail
  d        = Derivs { dvAdditive  = pAdditive parsed,
                      dvMultitive = pMultitive parsed,
                      dvPrimary   = pPrimary parsed,
                      dvDigits    = pDigits parsed }
  parsedTail = case s
               of  [ ]      → [ ]
                   (_ : s′) → parse s′
```

The function *parse* now returns a list of all input tails and corresponding derivatives for each input position. The derivatives in the head of the list provide the parsed results for the full input.

Parsers that use only *Derivs* suffer a little, because they need to dig it out of the head of the whole *parsed* list. On the other hand, with the input *s* in scope there is no need for *dvChar* any longer, so we can rewrite *pAdditive* to

```
  -- Parse an additive-precedence expression
pAdditive :: [ (String, Derivs) ] → Result Int
pAdditive ((s, d) : tail)
     -- Additive → Multitive '+' Additive
   | Parsed vleft (('+' : s′, d′) : (_, d′′) : _) ← dvMultitive d,
     Parsed vright rest ← dvAdditive d′′
        = Parsed (vleft + vright) rest
     -- Additive → Multitive
   | Parsed v rest ← dvMultitive d = Parsed v rest
   | otherwise                     = NoParse
```

The diagram in figure 2 again shows the data structure constructed for the expression *parse* "10+2*3", this time by the modified packrat algorithm. Notice that only three of seven *Derivs* get evaluated now, because these are the only input positions where a non-terminal needs be evaluated. The remaining input positions are consumed by primitive parsers instead. As we shall see, this translates to significant savings for larger grammars: instead of evaluating and allocating a large record, we can get away with allocating only an unevaluated thunk.

Another way of describing the result would be to equate every input chunk wholly consumed by a primitive parser with a lexical token. Where the original packrat algorithm allocated a record of *Derivs* for every input character, then, the modified algorithm allocates one per lexical token.

2.3 Applicative Functor Abstraction

Now that we have modified the type of the parsing functions, we can follow the original packrat thesis [16] and wrap them up in a *Parser* data type:

```
newtype Parser a = Parser { applyParser :: [ (String, Derivs) ]
                                              → Result a }
```

Our top-level function *parse* now has to unwrap and apply the parsers, but otherwise doesn't change:

```
parse s = parsed where
  parsed = (s, d) : parsedTail
  d = Derivs { dvAdditive  = applyParser pAdditive parsed,
               dvMultitive = applyParser pMultitive parsed,
               dvPrimary   = applyParser pPrimary parsed,
               dvDigits    = applyParser pDigits parsed }
  parsedTail = case s
               of  [ ]      → [ ]
                   (_ : s′) → parse s′
```

As in the earlier work, a *Parser* is a *Monad*, but that class has gained several super-classes in the meantime so we need to modernize the instance chain:

```
instance Functor Result where
  fmap f (Parsed a rest) = Parsed (f a) rest
  fmap f NoParse         = NoParse
instance Functor Parser where
  fmap f (Parser p) = Parser (fmap f ∘ p)
instance Applicative Parser where
  pure a = Parser (Parsed a)
  Parser p ⟨∗⟩ Parser q = Parser r where
    r rest = case p rest
             of  Parsed f rest′ → f ⟨$⟩ q rest′
                 _              → NoParse
instance Alternative Parser where
  empty = Parser (const NoParse)
  Parser p ⟨|⟩ Parser q = Parser r where
    r rest = case p rest
             of  x@Parsed { } → x
                 NoParse      → q rest
instance Monad Parser where
  return = pure
  Parser p ⟩⟩= f = Parser r where
    r rest = case p rest
             of  Parsed a rest′ → applyParser (f a) rest′
                 _              → NoParse
```

We don't follow Ford [16] in distinguishing the local deterministic choice as a new operator, because the laws of the *Alternative* class are very permissive and its method (⟨|⟩) already serves many different semantics. One more won't make a difference.

The parser combinator methods we gain from the above type class instances are not much use without some parser primitives. To complete our little grammar, three will suffice:

```
nt :: (Derivs → Result a) → Parser a
nt f = Parser p where
```

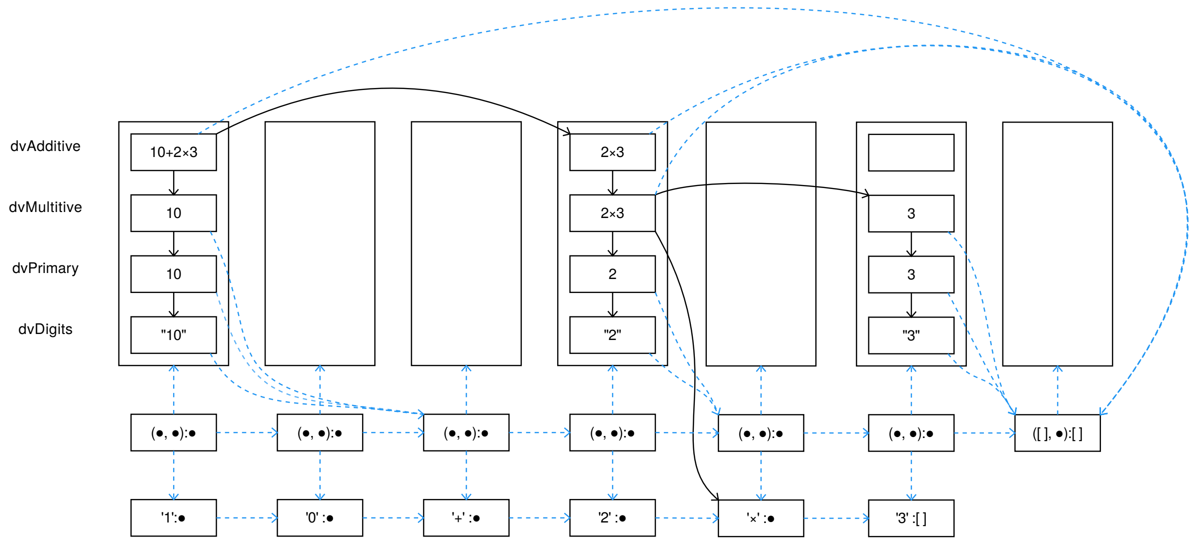

Figure 2. Modified packrat data structure for input "10+2*3"

```
  p ((_, d) : _) = f d
  p _            = NoParse
string :: String → Parser String
string s = Parser p where
  p rest@((s′, _) : _)
     | isPrefixOf s s′ = Parsed s (drop (length s) rest)
  p _                 = NoParse
takeCharsWhile1 :: (Char → Bool) → Parser String
takeCharsWhile1 pred = Parser p where
  p rest@((s, _) : _) | x@(_ : _) ← takeWhile pred s
      = Parsed x (drop (length x) rest)
  p _ = NoParse
```

The function *nt* is short for non-terminal and is the only primitive we need to access the production results memoized in *Derivs*. Functions *string* and *takeCharsWhile1* are relatively standard parser primitives.

With these definitions, the parsers become about as short as can be:

```
pAdditive :: Parser Int
pAdditive = alt1 ⟨|⟩ alt2 where
    -- Additive → Multitive '+' Additive
  alt1 = (+) ⟨$⟩ nt dvMultitive ⟨∗ string "+" ⟨∗⟩ nt dvAdditive
    -- Additive → Multitive
  alt2 = nt dvMultitive
pMultitive :: Parser Int
pMultitive = alt1 ⟨|⟩ alt2 where
    -- Multitive → Primary '*' Multitive
  alt1 = (∗) ⟨$⟩ nt dvPrimary ⟨∗ string "*" ⟨∗⟩ nt dvMultitive
    -- Multitive → Primary
  alt2 = nt dvPrimary
pPrimary :: Parser Int
pPrimary = alt1 ⟨|⟩ alt2 where
    -- Primary → '(' Additive ')'
  alt1 = string "(" ∗⟩ nt dvAdditive ⟨∗ string ")"
    -- Primary → Digits
  alt2 = read ⟨$⟩ nt dvDigits
pDigits :: Parser String
pDigits = takeCharsWhile1 isDigit
```

This abstraction of the parsing operations may cost us a constant factor of performance, but the gain in readability seems worth the price.

2.4 Generalize from *String* to Arbitrary *FactorialMonoid* Input

The core types *Derivs*, *Result*, and *Parser* refer to *String* but not to *Char*, and the same is true of the *parse* type signature. This gives us the opportunity to replace *String* by a parameter, generalizing the algorithm to operate on *Text*, *ByteString*, and other possible input types.

```
data Result s v = Parsed {parsedPrefix :: v,
                          parsedSuffix :: [(s, Derivs s)]}
                | NoParse
data Derivs s = Derivs {dvAdditive  :: Result s Int,
                        dvMultitive :: Result s Int,
                        dvPrimary   :: Result s Int,
                        dvDigits    :: Result s [Char]}
newtype Parser s a
  = Parser {applyParser :: [(s, Derivs s)] → Result s a}
```

Other than the type signatures, all definitions remain the same with the exception of *parse* and the two primitive parsers. The latter rely on the standard list-specific functions *isPrefixOf*, *length*, and *takeWhile*, which can easily be generalized to class methods from the *monoid-subclasses* library [3]:

```
string :: (FactorialMonoid s, LeftReductiveMonoid s)
       ⇒ s → Parser s s
```

```
string s = Parser p where
  p rest@((s′, _) : _)
    | Cancellative.isPrefixOf s s′
      = Parsed s (drop (Factorial.length s) rest)
  p _ = NoParse
takeCharsWhile1 :: TextualMonoid s
                ⇒ (Char → Bool) → Parser s s
takeCharsWhile1 predicate = Parser p where
  p rest@((s, _) : _)
    | x ← Textual.takeWhile False predicate s, ¬ (Null.null x)
      = Parsed x (drop (Factorial.length x) rest)
  p _ = NoParse
```

The function *parse* is not quite as obvious because it directly uses list patterns and constructors, but those can be replaced by a fold on the list of the input *tails* and then generalized using the same library:

```
parse :: TextualMonoid s ⇒ s → [(s, Derivs s)]
parse input = foldr parseTail [ ] (Factorial.tails input) where
  parseTail s parsedTail = parsed where
    parsed = (s, d) : parsedTail
    d = Derivs {dvAdditive  = applyParser pAdditive parsed,
                dvMultitive = applyParser pMultitive parsed,
                dvPrimary   = applyParser pPrimary parsed,
                dvDigits    = applyParser pDigits parsed}
```

As a point of curiosity, the modified packrat algorithm presented here does not allow as wide a choice of inputs as the backtracking parser presented by Blažević [2]. Its use of the function *tails* fixes a single ordering of atomic input fragments and thus excludes commutative monoids as parser inputs.

3 Organizing the Rat Pack

Now that we have generalized and optimized the parsers, let's widen our focus to the grammar they belong to.

3.1 Enter the Grammar

Packrat is just a particular implementation of a parser for a parsing expression grammar. The preceding code, however, does not make it obvious what grammar it implements. Although we have a bunch of *Parser* definitions, there is no grammar to group them together. The nearest we come is the rather repetitive construction of *Derivs*, repeated again below with type annotations added to fields:

```
d = Derivs {dvAdditive  = pAdditive parsed  :: Result s Int,
            dvMultitive = pMultitive parsed :: Result s Int,
            dvPrimary   = pPrimary parsed   :: Result s Int,
            dvDigits    = pDigits parsed    :: Result s [Char]}
```

The purpose of *Derivs* so far has been grouping parser *results*, not parsers themselves. The grammar we're after would have to look like this instead:

```
grammar = Derivs {dvAdditive  = pAdditive  :: Parser s Int,
                  dvMultitive = pMultitive :: Parser s Int,
                  dvPrimary   = pPrimary   :: Parser s Int,
                  dvDigits    = pDigits    :: Parser s [Char]}
```

If we compare the types of the fields of the two records above, we'll notice that they differ only in the type constructor (*Result* versus *Parser*), which means that we can unify the two types by parameterizing the constructor:

```
data Derivs f = Derivs {dvAdditive  :: f Int,
                        dvMultitive :: f Int,
                        dvPrimary   :: f Int,
                        dvDigits    :: f [Char]}
```

Now that the type *Derivs* has gained a parameter, the type of *d* becomes *Derivs* (*Result s*). The type of *grammar* is *Derivs* (*Parser s*) instead. The type parameter also propagates to the other type declarations that refer to *Derivs*.

```
data Result s v = Parsed {parsedPrefix :: v,
                          parsedSuffix :: [(s, Derivs (Result s))]}
                | NoParse
newtype Parser s a
  = Parser {applyParser :: [(s, Derivs (Result s))] → Result s a}
```

The next step is to rewrite the core function *parse*. The first attempt would simply use the grammar fields instead of the individual parser functions:

```
parse :: FactorialMonoid s ⇒ s → [(s, Derivs (Result s))]
parse input = foldr parseTail [ ] (Factorial.tails input) where
  parseTail s parsedTail = parsed where
    parsed = (s, d) : parsedTail
    d = Derivs
      { dvAdditive  = applyParser (dvAdditive grammar)
                                  parsed,
        dvMultitive = applyParser (dvMultitive grammar)
                                  parsed,
        dvPrimary   = applyParser (dvPrimary grammar)
                                  parsed,
        dvDigits    = applyParser (dvDigits grammar) parsed}
```

There's a pattern here: we seem to be applying the same function ((\$*parsed*) ∘ *applyParser*) to every single field of the record data structure. If the data structure was a standard Haskell *Functor*, we could use *fmap* instead, but sadly it cannot be because not all values have the same type. In case of *dvDigits*, we map a *Parser s* [*Char*] to *Result s* [*Char*]. All other *grammar* fields are of type *Parser s Int*, which we map to *Result s Int*. Rather than map the entire field type, we map only the type constructors. The type of the *fmap* operation we'd need is

```
fmap :: (∀a.Parser a → Result a) → Derivs Parser → Derivs Result
```

This is a rank-2 type, which is not supported by standard *Haskell* 2010, but luckily GHC has been supporting it ever since version 5.4. So we'll just define a new type class, and rather than try to invent a brand new name for it, we'll reuse the same class and method name in a new namespace called *Rank2*:

```
{-# LANGUAGE RankNTypes #-}
module Rank2 where
import qualified Prelude as Rank1
import qualified Control.Monad as Rank1
```

```
  -- Equivalent of 'Rank1.Functor' for rank-2 data types
class Functor g where
  fmap :: (∀a.p a → q a) → g p → g q
```

The instance of this class for *Derivs* turns out to be quite obvious:

```
instance Rank2.Functor Derivs where
  fmap f g = Derivs {dvAdditive   = f (dvAdditive g),
                     dvMultitive  = f (dvMultitive g),
                     dvPrimary    = f (dvPrimary g),
                     dvDigits     = f (dvDigits g)}
```

And now we can finally abstract the grammar record out of the function *parse* and pass it as an argument instead:

```
parse :: FactorialMonoid s
      ⇒ Derivs (Parser s) → s → [(s, Derivs (Result s))]

parse g input = foldr parseTail [ ] (Factorial.tails input) where
  parseTail s parsedTail = parsed where
    parsed = (s, d) : parsedTail
    d      = Rank2.fmap (($parsed) ∘ applyParser) g
```

3.2 Parameterize the Grammar Type

The definition of *parse* is now applicable to any *Rank2.Functor* instance, but it still has *Derivs* in its type signature. We'd like to parameterize it away, so we can use the same *parse* definition for different grammar types. As a consequence, we are forced to propagate the same grammar parameter to the *Parser* and *Result* types:[3]

```
data Result g s v = Parsed {parsedPrefix :: v,
                            parsedSuffix :: [(s, g (Result g s))]}
                  | NoParse
newtype Parser g s a
  = Parser {applyParser :: [(s, g (Result g s))] → Result g s a}
 {-# NOINLINE parse #-}
parse :: (Rank2.Functor g, FactorialMonoid s)
      ⇒ g (Parser g s) → s → [(s, g (Result g s))]
```

No definition needs to change, only the type signatures.

3.3 More Rank-2 Type Classes

Since *Rank2.Functor* was such a resounding success, you must be asking yourself, can we extend the same treatment to other standard type classes? Why yes we can, and indeed we should just for the sake of it. Here's some more:

```
  -- Equivalent of 'Rank1.Foldable' for rank-2 data types
class Foldable g where
  foldMap :: Rank1.Monoid m ⇒ (∀a.p a → m) → g p → m
  -- Equivalent of 'Rank1.Traversable' for rank-2 data types
class (Functor g, Foldable g) ⇒ Traversable g where
  traverse :: Rank1.Applicative m
           ⇒ (∀a.p a → m (q a)) → g p → m (g q)
newtype Arrow p q a = Arrow {apply :: p a → q a}
  -- Subclass of 'Functor' halfway to 'Applicative'
class Functor g ⇒ Apply g where
  (⟨∗⟩)  :: g (Arrow p q) → g p → g q
  liftA2 :: (∀a.p a → q a → r a) → g p → g q → g r
  (⟨∗⟩)        = liftA2 apply
  liftA2 f g h = (Arrow ∘ f) ⟨$⟩ g ⟨∗⟩ h
  -- Equivalent of 'Rank1.Applicative' for rank-2 data types
class Apply g ⇒ Applicative g where
  pure :: (∀a.f a) → g f
  -- Equivalent of 'Rank1.Distributive' for rank-2 data types
class Functor g ⇒ Distributive g where
  distributeWith :: Rank1.Functor f1
                 ⇒ (∀a.f1 (f2 a) → f a) → f1 (g f2) → g f
  distributeM :: Rank1.Monad f ⇒ f (g f) → g f
  distributeM = distributeWith Rank1.join
```

Instances of Haskell's standard type classes *Functor*, *Foldable*, *Traversable*, *Applicative*, and *Monad* are *type constructors* of kind $* \to *$: they map a data type to another data type. Instances of the rank-2 type classes defined here instead map a type constructor to a data type. Their kind signature is $(* \to *) \to *$. The class methods have rank-2 types, but apart from that they are equivalent to the methods of the standard classes, and their instances are expected to satisfy the same laws.

The instances of all presented classes can be mechanically derived for any single-constructor abstract data type with appropriate field types. This mechanical derivation is provided as a set of Template Haskell definitions that can be spliced into user modules with minimal syntactic overhead.

3.3.1 Applications

The most immediate application of the above type classes is for records with heterogenously-typed fields. That includes our grammar records but is certainly not limited to them. Another obvious candidate would be database records. A deeper discussion does not belong here, but, given a record declaration like

```
data DbRecord f = DbRecord {
    id   :: f Int,
  name :: f String,
  birth :: f Date,
    ...
}
```

and the appropriate *Rank2* class instances, we can obtain for free useful types and expressions like:

- *DbRecord Identity* is the record with plain field values,
- *getSum* ∘ *Rank2.foldMap* (*const* $ *Sum* 1) counts its fields,
- *DbRecord* (*Const ByteString*) would be the database record type with binary serialization for all fields,
- *Rank2.foldMap getConst* applied to a record of the above type encodes the whole record as a *ByteString* sequence,
- *DbRecord* (*Either String*) is a record, any field of which may be replaced by an error,
- but if we apply *Rank2.traverse* (*Identity*⟨$⟩) to it we'll get either an error or a clean record, and
- if we apply *Rank2.liftA2* (\`*either*\`*Identity*) *defaults* we'll replace erroneous fields by default field values.

[3] The *NOINLINE* pragma is necessary to improve the performance of GHC-optimized code. The intricacies of GHC are out of scope for this paper.

3.4 Shorten the Grammar Definition by Getting Rid of nt

Back to the grammar definition. Using the applicative notation, the grammar definition can be written with almost no overhead, when one takes into account that it specifies not only the syntax but the semantic actions as well:

```
data Derivs f = Derivs { additive  :: f Int,
                         multitive :: f Int,
                         primary   :: f Int,
                         digits    :: f [Char] }
arithmeticGrammar :: TextualMonoid s ⇒ Derivs (Parser Derivs s)
arithmeticGrammar
  = Derivs { additive  = (+) ⟨$⟩ nt multitive ⟨* string "+"
                               ⟨*⟩ nt additive
                         ⟨|⟩ nt multitive,
             multitive =  (*) ⟨$⟩ nt primary ⟨* string "*"
                               ⟨*⟩ nt multitive
                         ⟨|⟩ nt primary,
             primary   =  string "(" *⟩ nt additive ⟨* string ")"
                         ⟨|⟩ read ⟨$⟩ nt digits,
             digits    =  Textual.toString (const "")
                            ⟨$⟩ takeCharsWhile1 isDigit }
```

The only grating detail is the repeated use of *nt*. Its purpose is to turn a grammar field name (accessor) into a parser that reads the parse result memoized in that field. To get rid of the *nt* function, then, we need to make each field name a parser by itself. As it happens, there is a neat way to bind the field names to exactly such parsers. We'll turn the grammar definition from a record data type to a function that maps the same data type to itself, and we'll bring all field values of the argument record into scope using the {-# LANGUAGE RecordWildCards #-} language extension:

```
arithmeticGrammar :: TextualMonoid s
  ⇒ Derivs (Parser Derivs s) → Derivs (Parser Derivs s)
arithmeticGrammar ~Derivs {..}
  = Derivs { additive  = (+) ⟨$⟩ multitive ⟨* string "+"
                               ⟨*⟩ additive
                         ⟨|⟩ multitive,
             multitive = (*) ⟨$⟩ primary ⟨* string "*"
                               ⟨*⟩ multitive
                         ⟨|⟩ primary,
             primary   = string "(" *⟩ additive ⟨* string ")"
                         ⟨|⟩ read ⟨$⟩ digits,
             digits    = Textual.toString (const "")
                            ⟨$⟩ takeCharsWhile1 isDigit }
```

All that remains is to turn *arithmeticGrammar* from the function it's defined as back into just a plain record *Derivs* (*Parser Derivs s*). Expression *fix arithmeticGrammar* has the right type and, thanks to the lazy pattern match, returns a record of usable mutually-recursive parsers, but with no memoization. To restore the memoization, we need an all new fixing function instead:

```
fixGrammar :: (Derivs (Parser Derivs s)
               → Derivs (Parser Derivs s))
  → Derivs (Parser Derivs s)
fixGrammar grammar = grammar selfReferring
selfReferring :: Derivs (Parser Derivs s)
selfReferring = Derivs { additive  = nt additive,
                         multitive = nt multitive,
                         primary   = nt primary,
                         digits    = nt digits }
```

The sum of our accomplishment, as it turns out, is to eliminate the repetitive occurrences of *nt* from the grammar by putting them all in one place. It's a syntactic win as long as the grammar contains more field references than fields. But even better, a single definition of *selfReferring* can cover all instances of *Rank2.Distributive*:

```
fixGrammar :: Rank2.Distributive g
  ⇒ (g (Parser g s) → g (Parser g s)) → g (Parser g s)
fixGrammar gf = gf selfReferring
selfReferring :: Rank2.Distributive g ⇒ g (Parser g s)
selfReferring = Rank2.distributeWith nt id
instance Rank2.Distributive Derivs where
  distributeWith w f = Derivs { additive  = w (additive ⟨$⟩ f),
                                multitive = w (multitive ⟨$⟩ f),
                                primary   = w (primary ⟨$⟩ f),
                                digits    = w (digits ⟨$⟩ f) }
```

3.5 Modular Grammars

Notice that the most general type of the latest *arithmeticGrammar* definition is not

```
Derivs (Parser Derivs s) → Derivs (Parser Derivs s)
```

as we have specified, but

```
Derivs (Parser g s) → Derivs (Parser g s)
```

as the compiler would infer. This means that a grammar record can contain parsers that refer to other grammars; as a corollary, two distinct grammars can contain parsers referring to the same third grammar.

For example, suppose that in addition to the already-defined *arithmeticGrammar* we also have *booleanGrammar*, defined in a similar fashion:

```
data Boolean f = Boolean { disjunction :: f Bool,
                           conjunction :: f Bool,
                           negation    :: f Bool,
                           truth       :: f Bool }
booleanGrammar :: TextualMonoid s
  ⇒ Boolean (Parser g s) → Boolean (Parser g s)
booleanGrammar ~Boolean {..}
  = Boolean { disjunction  = (∨) ⟨$⟩ conjunction
                                ⟨* string "||"
                                ⟨*⟩ disjunction
                             ⟨|⟩ conjunction,
              conjunction  = (∧) ⟨$⟩ truth ⟨* string "&&"
                                ⟨*⟩ negation
                             ⟨|⟩ negation,
              negation     = ¬ ⟨$ string "!" ⟨*⟩ negation
                             ⟨|⟩ truth,
              truth        = False ⟨$ string "False"
                             ⟨|⟩ True ⟨$ string "True" }
```

Once we make *Boolean* an instance of both *Rank2.Functor* and *Rank2.Distributive*, in much the same way we made *Derivs* earlier, we can apply to it the same functions *fixGrammar* and *parse* that we used on *arithmeticGrammar*. Better yet, we can define a new grammar containing both *arithmeticGrammar* and *booleanGrammar* as sub-grammars.

Again we start by defining a parameterized record for the grammar. This time two of its fields will contain whole sub-grammars rather than a single production each:

```
data Combined f
  = Combined {expression           :: f (Either Bool Int),
              comparison           :: f Bool,
              arithmeticSubGrammar :: Derivs f,
              booleanSubGrammar    :: Boolean f}
```

The type class instances for this record type can be mechanically derived, but we are including them below because they differ from the ones we have seen so far:

```
instance Rank2.Functor Combined where
  fmap f g = g {expression   = f (expression g),
                comparison   = f (comparison g),
                arithmeticSubGrammar
                  = Rank2.fmap f (arithmeticSubGrammar g),
                booleanSubGrammar
                  = Rank2.fmap f (booleanSubGrammar g)}
instance Rank2.Distributive Combined where
  distributeWith w f = Combined {
    expression           = w (expression ⟨$⟩ f),
    comparison           = w (comparison ⟨$⟩ f),
    arithmeticSubGrammar = Rank2.distributeWith w
                             (arithmeticSubGrammar ⟨$⟩ f),
    booleanSubGrammar    = Rank2.distributeWith w
                             (booleanSubGrammar ⟨$⟩ f)}
```

The only non-trivial part is the combined grammar definition, once more represented as a function mapping the grammar record to itself:

```
combinedGrammar :: TextualMonoid s
  ⇒ Combined (Parser g s) → Combined (Parser g s)
combinedGrammar ~Combined {..} = Combined
  {arithmeticSubGrammar = arithmeticGrammar
                            arithmeticSubGrammar,
   comparison = additive arithmeticSubGrammar
                  ⟨**⟩ (   (⩽) ⟨$ string "<="
                         ⟨|⟩ (<) ⟨$ string "<"
                         ⟨|⟩ (⩾) ⟨$ string ">="
                         ⟨|⟩ (>) ⟨$ string ">"
                         ⟨|⟩ (≡) ⟨$ string ">="
                         ⟨|⟩ (≢) ⟨$ string "!=")
                  ⟨*⟩  additive arithmeticSubGrammar
               ⟨|⟩ truth booleanSubGrammar,
   booleanSubGrammar = booleanGrammar
                         booleanSubGrammar
                           {truth = comparison},
   expression   = Left  ⟨$⟩ disjunction booleanSubGrammar
                ⟨|⟩ Right ⟨$⟩ additive arithmeticSubGrammar}
```

The above *combinedGrammar* definition

- embeds and reuses *arithmeticGrammar* unchanged,
- adds another production *comparison* that reuses the production *additive* from *arithmeticGrammar* and *truth* from *booleanGrammar*,
- embeds *booleanGrammar* while redirecting its *truth* production to *comparison*, and finally
- defines the top production *expression* as a tagged union of *additive* from *arithmeticGrammar* and *disjunction* from *booleanGrammar*.

We can use *combinedGrammar* to parse expressions that conform to either *arithmeticGrammar* or *booleanGrammar*:

```
> let parsedExpression
    = parsedPrefix ∘ expression ∘ snd ∘ head
      ∘ parse (fixGrammar combinedGrammar)
> parsedExpression "1*2+3*4"
Right 14
> parsedExpression "False||False&&True"
Left False
```

Furthermore, *combinedGrammar* can recognize expressions that combine parts of its both constituent sub-languages:

```
> parsedExpression "True&&100>2"
Left True
```

It may be worth noting that both *arithmeticGrammar* and *booleanGrammar* in this example are full-fledged and perfectly usable grammars on their own, but we are still able to tweak their productions in order to combine them into a larger grammar.

While this combined grammar example is ad-hoc, we can also define general grammar combinators. Any two grammars can be combined into a grammar product, yielding a bigger grammar that contains productions from both original grammars. To make this grammar more than the sum of its parts, we must then modify some of the productions to refer to the other grammar, like the fragment *booleanSubGrammar* {*truth* = *comparison*} above does.

Two grammars of the same type, as long as the type is an instance of *Rank2.Apply*, can be combined by applying an arbitrary binary function to their production fields pairwise. We can use this facility, for example, to add alternatives to one or more productions.

4 Results

4.1 Packrat Performance Improvements

Figure 3 presents the run-times of three different parsers on the same series of string inputs. The input list is

```
["1000000000" ⋄ mconcat (replicate n "+1000*1000000")
  | n ← [10000, 40000, 70000, 100000]]
```

meant to simulate varying token lengths as would appear in most source code. The input sizes thus varied from 126 KiB to 1.2 MiB. All measurements were taken using the Criterion benchmarking framework compiled by GHC 8.0.2 with the standard optimizations, on a 64-bit Fedora Linux system with 12GB of RAM, running on a single core of the 3.50GHz Intel® Core™ i5-4690 CPU.

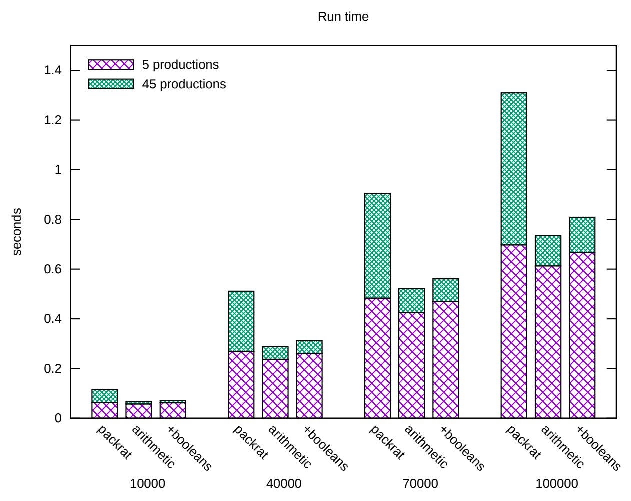

Figure 3. Benchmarks of packrat improvements

Three parsers have been tested: the plain packrat parser as presented in section 2.1 with no monad notation or any other syntax extension; the final version of the arithmetic parser from section 3.4; and the combined arithmetic+booleans parser from the preceding section. Each parser has been tested as presented, and also with 40 additional dummy productions added into the grammar (the *Derivs* record), to examine the cost of growing the grammar. Figure 3 stacks the additional run-time incurred by the 40 dummy productions on top of the regular run-time, for easier comparison.

The measurements demonstrate that all parsers' run-times are roughly proportional to the input length. The modified packrat algorithm presented by this paper, using the arithmetic grammar alone, is only slightly faster than original packrat parser; the optimizations only beat the overheads by some 12%.

We can also see that grammar extension has become much less expensive. Adding comparisons and boolean operations to the arithmetic grammar slows it down by approximately 10%. Furthermore, expanding the grammar from 5 to 45 productions, 40 of which are unused, incurs a roughly 20% run-time penalty on the modified algorithm, but a 90% penalty on the original packrat algorithm.

4.2 Grammar Representation and Reuse

To our knowledge, this is the first time a grammar has been represented using a plain function that maps the abstract data type containing the grammar productions to itself. Earlier work in Haskell has mostly relied on the recursive **do** language extension instead. In our biased opinion, the results look rather clunky compared to the formulation presented here.

Together with the rank-2 type classes, this enables the parser combinators and the functions on grammars, such as *parse*, to be generalized to operate on any grammar. They can thus be packaged into a stand-alone function library and reused across a wide range of grammars.

Another benefit of the proposed grammar representation is that grammars can be easily manipulated after they are defined, as shown in section 3.5.

4.3 Grammars and Parser Combinators

While having a more readable grammar formulation is a nice result on its own, it brings the additional benefit of bridging the gap between formal grammars and parser combinator libraries. The parser combinators we have been using are the same ones used by Parsec and similar libraries. The main point of distinction is that our production references do not rely on primitive recursion, but are under our control. Should we decide that only the *additive* and *digits* production of *arithmeticGrammar* are worth memoizing, for example, we could leave only these two fields in *Derivs* and move the rest into a **let** clause instead:

```
data Derivs′ f = Derivs′ {additive′ :: f Int,
                          digits′   :: f [Char]}
arithmeticGrammar ~Derivs′ {..} =
  let multitive = (∗) ⟨$⟩ primary ⟨∗ string "*" ⟨∗⟩ multitive
                  ⟨|⟩ primary
      primary   = string "(" ∗⟩ additive′ ⟨∗ string ")"
                  ⟨|⟩ read ⟨$⟩ digits′
  in Derivs′ {
      additive′ = (+) ⟨$⟩ multitive ⟨∗ string "+" ⟨∗⟩ additive′
                  ⟨|⟩ multitive,
      digits′   = Textual.toString (const "")
                    ⟨$⟩ takeCharsWhile1 isDigit}
```

As noted in the introduction, we can also eliminate all memoization simply by replacing the call to *fixGrammar* by *fix*. Of course, in that case it makes no sense to pay for the overhead of a parser capable of memoization, but we can switch to a simple backtracking parser by a mere change of the *arithmeticGrammar* type declaration. The grammar definition can remain the same.

```
arithmeticGrammar :: TextualMonoid s
  ⇒ Derivs (Backtracking.Parser Derivs s)
    → Derivs (Backtracking.Parser Derivs s)
```

The backtracking parser and the packrat parser are provided by the same library and they support the same set of parser combinators. The sole distinction is in the *fixGrammar* function that is supported only by the packrat parser and in the parsers' performance profiles.

4.4 Relative Performance

Figure 4 presents the run-times of four different parsers applied to the same two grammars and the same series of inputs as in figure 3. The grammars are *arithmeticGrammar* and *combinedGrammar* defined earlier with no additional dummy productions, while the parsers are

- improved packrat parser with *fixGrammar* as in figure 3,
- the same parser but using the *fix* function instead to prevent memoization,
- a backtracking PEG parser with no memoization support nor related overheads, and
- Parsec.

It should be noted that Parsec is included only to put this work in a wider context. It is not a true PEG parser, which hampers its performance: if the first alternative of ⟨|⟩ succeeds but its continuation fails, Parsec will try the second alternative. Parsec is also a

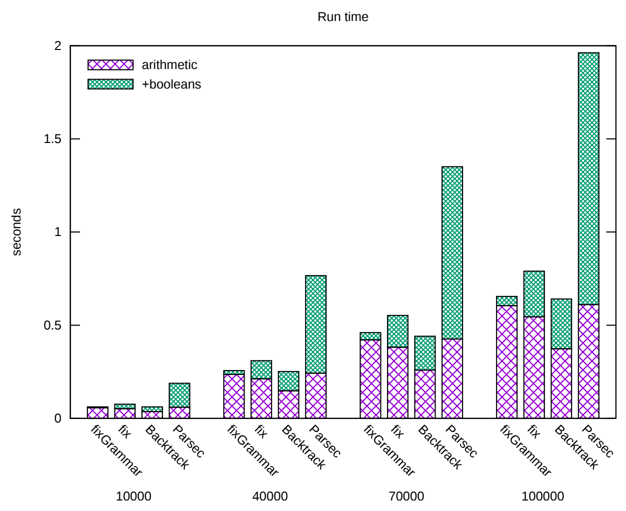

Figure 4. Benchmarks versus backtracking parsers

monad transformer and keeps track of user state, which would be impossible to do with any packrat parser.

The first three parsers use the earlier *arithmeticGrammar* and *combinedGrammar* definitions, adjusting only their type signatures. Parsec required additional modifications to the grammar definitions, namely wrapping the left-hand side of every ⟨|⟩ operator in *try* and replacing the *takeCharsWhile1* function by *many1* ∘ *satisfy*.

We can see that all parsers have linear performance. The costs of packrat memoization outweigh its benefits for the simple *arithmeticGrammar*, but it almost pays off for the larger grammar. Things would look worse for packrat if we were to left-factor the grammar productions. The packrat parsers' main claim to fame is their linear performance guarantee; in practice even a naïve backtracking, non-memoizing PEG parser can parse virtually all well-designed[4] languages in linear time.

That being said, we were able to trigger the exponential parse time only by applying some ridiculously ham-fisted tweaks to the arithmetic grammar. Suppose we wish to allow newlines in our arithmetic, and being too lazy to analyze the grammar we simply add a newline string to the end of its every production:

arithmeticGrammar′
  = *Rank2.fmap* (⟨∗ *string* "\n") ∘ *arithmeticGrammar*

Then we realize that the newlines should be optional! Luckily, there is no need to throw all that hard work away: we'll just create a union of the new grammar that requires newlines and the original one that doesn't accept them:

arithmeticGrammar″ g
  = *Rank2.liftA2* (⟨|⟩) (*arithmeticGrammar′ g*)
                         (*arithmeticGrammar g*)

If we try parsing against *arithmeticGrammar″* the same arithmetic expressions we were testing earlier but with a single newline appended, we'll trigger exponential parse time complexity in the backtracking parser. The packrat parser is idiot-proof: even though it is slower by 38% than with the original grammar, it just chugs along on its linear track.

[4]Markdown seems to be a notable exception, judging from the fate of the pegdown parser [12].

5 Related Work

5.1 Rank-2 Type Classes

The existing work to modify and reuse *Functor* and related classes largely concentrates on adding type indexes in order to improve their precision, for example to keep track of container length. McBride [25] continues in the same tradition but qualifies the type index with ∀·, thus ending up with a rank-2 type class that is similar but more general than our *Rank2.Functor*. The other classes we present here do not have such equivalents.

Another area of research where classes with methods of rank-2 type have appeared is in datatype-generic programming. de Vries and Löh [7] in particular lists almost all the methods presented here, but their type signatures are constrained to generically representable types. Their very generic-ness makes them less general.

5.2 Parser and Grammar Combinators

The semantic gap between the grammars and parsers has been recognized before [21] [10]. The proposed solutions include using the *MonadFix* type class [14] supported by the recursive **do** language extension [15], fix-point data types [34], typed abstract syntax for the grammar constructed with GADTs [31] [32], and heterogenous typed lists [28]. The solution supported by the type class *ApplicativeFix* from Devriese et al. [11] is the closest we could find to our rank-2 type classes. Interestingly, the same authors' previous work [10] appears to briefly consider representing grammars as functions on records, but abandons this idea for a different approach.

Among the published Haskell parser libraries, *Earley* [13] [19], *Grempa* [18], and *Frisby* [26] support explicit grammar representation. All three rely on the recursive **do** language extension.

The *grammar-combinators* library [8] stands apart in using grammar combinators and ad-hoc type classes [9] loosely related to the rank-2 type classes presented here. The library impressively demonstrates how the same grammar definition can target different back-end parsers.

5.3 PEGs and Packrat Parsers

Even though the original packrat algorithm [16] guarantees linear parsing complexity, the linear factor can be uncomfortably large, and the parsers' memory consumption is linear as well. There have been several improvements to the algorithm's performance since, including by Grimm [20] and Redziejowski [29]. We have not attempted to relate our modifications to the existing parsers in this tradition.

Since they require explicit references to support memoization, most packrat parsers are generated from a PEG and as such are not reusable at run-time. A notable exception in the Haskell ecosystem is the already-mentioned *Frisby* [26] library. The parser generator *Rats*! in Java is capable of combining modular grammars before generating the final parser code [21].

6 Conclusion and Future Developments

The improved packrat parser presented in this paper has been released as a Haskell library[5], together with a backtracking PEG

[5]http://hackage.haskell.org/package/grammatical-parsers

parser mentioned in section 4.4 and a few parsers for context-free grammars that fall outside the scope of this paper. Compared to the parser presented here, the released version adds proper error reporting, more parser primitives and combinators, and integrates with the rest of the library.

The same public repository provides the rank-2 type classes presented in section 3.3, as well as several examples of simple grammars expressed as endomorphic record functions and combined into a larger grammar using the technique from section 3.5.

6.1 Streaming

Rather than try to replicate the existing optimizations of the packrat algorithm at the cost of complicating it, we feel it may be more worthwhile to make the parser capable of streaming. This has already been done for *Frisby* [26], and it would work around the linear memory consumption problem.

6.2 Non-deterministic and Left-recursive Productions

The representation of grammars as an endomorphic record function is not restricted to parsing expression grammars. We already have a working parser that implements context-free grammars, left recursion included, but it awaits proper documentation and a proof of performance constraints.

6.3 Combining Grammars and Their Semantics

We believe that the new grammar representation should streamline the design and implementation of new language syntax even further. Sadly, the syntax has always been the easy part.

We have demonstrated that grammars can be easily combined in section 3.5. That particular example was not exactly contrived, but the same simple technique could not be applied to, say, adding variables to *arithmeticGrammar*. The reason is that the starting grammar fixes the base semantical domain of all expression levels to *Int*, and the variable extension would require a domain like ($Environment \rightarrow Int$).

A known solution to this problem is to use a final encoding [4] of the grammar's target semantics. Then the combined grammar can adjust the target semantics of each base grammar. We have succesfully combined five base grammars (arithmetic, boolean, untyped lambda calculus, comparisons, and conditionals) in this way as an experiment, but it remains to be seen how far this technique can be streched.

References

[1] John W Backus. 1959. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. *Proceedings of the International Conference on Information Processing, 1959* (1959).

[2] Mario Blažević. 2013. Adding structure to monoids. *ACM SIGPLAN NOTICES* 48, 12 (2013), 37–45.

[3] Mario Blažević. 2013. monoid-subclasses. http://hackage.haskell.org/package/monoid-subclasses. (2013).

[4] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 05 (2009), 509–543.

[5] Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on information theory* 2, 3 (1956), 113–124.

[6] Nils Anders Danielsson. 2010. Total Parser Combinators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 285–296. DOI: http://dx.doi.org/10.1145/1863543.1863585

[7] Edsko de Vries and Andres Löh. 2014. True Sums of Products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming (WGP '14)*. ACM, New York, NY, USA, 83–94. DOI: http://dx.doi.org/10.1145/2633628.2633634

[8] Dominique Devriese. 2010. The grammar-combinators package. http://hackage.haskell.org/package/grammar-combinators. (2010).

[9] Dominique Devriese and Frank Piessens. 2011. Explicitly Recursive Grammar Combinators: A Better Model for Shallow Parser DSLs. In *Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages (PADL'11)*. Springer-Verlag, Berlin, Heidelberg, 84–98. http://dl.acm.org/citation.cfm?id=1946313.1946326

[10] Dominique Devriese and Frank Piessens. 2012. Finally tagless observable recursion for an abstract grammar model. *Journal of Functional Programming* 22, 06 (2012), 757–796.

[11] Dominique Devriese, Ilya Sergey, Dave Clarke, and Frank Piessens. 2013. Fixing idioms: a recursion primitive for applicative DSLs. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*. ACM, 97–106.

[12] Mathias Doenitz. 2010. A pure-Java Markdown processor based on a parboiled PEG parser supporting a number of extensions. GitHub. (2010). https://github.com/sirthias/pegdown.

[13] Jay Earley. 1970. An efficient context-free parsing algorithm. *Commun. ACM* 13, 2 (1970), 94–102.

[14] Levent Erkök and John Launchbury. 2000. Recursive monadic bindings. In *ACM SIGPLAN Notices*, Vol. 35. ACM, 174–185.

[15] Levent Erkök and John Launchbury. 2002. A recursive do for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM, 29–37.

[16] Bryan Ford. 2002. Packrat parsing:: simple, powerful, lazy, linear time, functional pearl. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 36–47.

[17] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.* 39, 1 (2004), 111–122. DOI: http://dx.doi.org/10.1145/982962.964011

[18] Olle Fredriksson. 2011. The Grempa package. http://hackage.haskell.org/package/Grempa. (2011).

[19] Olle Fredriksson. 2014. The Earley package. http://hackage.haskell.org/package/Earley. (2014).

[20] Robert Grimm. 2004. *Practical packrat parsing*. Technical Report. Dept. of Computer Science, New York University.

[21] Robert Grimm. 2006. Better extensibility through modular syntax. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 38–51.

[22] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. 2016. Practical, general parser combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM, 1–12.

[23] Stephen C. Johnson. 1975. *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ.

[24] Daan Leijen and Erik Meijer. 2001. *Parsec: Direct style monadic parser combinators for the real world*. Technical Report. Department of Computer Science, Utrecht University.

[25] Conor McBride. 2011. Functional pearl: Kleisli arrows of outrageous fortune. *Journal of Functional Programming (to appear)* (2011).

[26] John Meacham. 2006. The Frisby package. http://hackage.haskell.org/package/frisby. (2006).

[27] Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: a functional pearl. In *ACM SIGPLAN notices*, Vol. 46. ACM, 189–195.

[28] Bruno C d S Oliveira and Andres Löh. 2013. Abstract syntax graphs for domain specific languages. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*. ACM, 87–96.

[29] Roman R Redziejowski. 2009. Mouse: from parsing expressions to a practical parser. In *Concurrency Specification and Programming Workshop*. Citeseer.

[30] S Doaitse Swierstra. 2009. Combinator parsing: A short tutorial. In *Language Engineering and Rigorous Software Development*. Springer, 252–300.

[31] Marcos Viera, Doaitse Swierstra, and Atze Dijkstra. 2012. Grammar fragments fly first-class. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*. ACM, 5.

[32] Marcos Viera, S. Doaitse Swierstra, and Eelco Lempsink. 2008. Haskell, Do You Read Me?: Constructing and Composing Efficient Top-down Parsers at Runtime. *SIGPLAN Not.* 44, 2 (Sept. 2008), 63–74. DOI: http://dx.doi.org/10.1145/1543134.1411296

[33] Philip Wadler. 1985. How to Replace Failure by a List of Successes. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag New York, Inc., New York, NY, USA, 113–128. http://dl.acm.org/citation.cfm?id=5280.5288

[34] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. 2009. Generic programming with fixed points for mutually recursive datatypes. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 233–244.