Parse Your Options

S. Doaitse Swierstra and Atze Dijkstra

Dept. of Computer Science, O.O. Box 80.089, 3508 TB Utrecht, the Netherlands
http://www.cs.uu.nl

Abstract. We describe the development of a couple of combinators which can be used to run applicative style parsers in an interleaved way. In the presentation we advocate a scheme for choosing identifier names which clearly shows the types of the values involved, and how to compose them into the desired result. We finish with describing how the combinators can be used to parse command line arguments and files containing options. .

Key words: Parser combinators, Haskell, Parallel parsing, Option processing, Permutation parsing

1 Introduction

In the original version of the `uulib`[1] parsing combinator library we introduced two modules which provide combinators for parsing more complicated input structures than those described by context free languages. For the sake of completeness we repeat the definition of the class *Applicative* which introduces a combinator `<*>` that combines two side-effecting functions into a single function. The important observation is that the results of running the two components are combined by applying the result of the first to that of the second. In this paper such functions will be parsers. The `<*>` operator is a function which runs its two parser arguments sequentially on the input: i.e. the second parser takes off in with the input state in which the first one has finished. The class *Alternative* introduces the companion operator `<|>` which runs either one of its operands, but not both. Its unit element is *empty*, which is not a parser which will recognise the empty string but the parser which will always fail. For the sake of completeness we also provide the *Functor* class.

```
class Functor f where
  fmap :: (b → a) → f b    → f a
class Applicative f where
  (<*>) :: f (b → a) → f b → f a
  pure  :: a                → f a
class Alternative f where
```

[1] `http://hackage.haskell.org/package/uulib`

```
(<|>) :: f a → f a → f a
empty :: f a
```

We introduce a couple of helper functions that are used to modify the result of a parser, or to throw away a result in which we are not interested:

```
(<$>) :: Functor f ⇒ (b → a) → f b → f a
(<$>)     = fmap
p <* q    = const      <$> p <*> q
f <$ p    = fmap (const f) p
p *> q    = flip const <$> p <*> q
p `opt` a = p <|> pure a
```

In addition to the operator for sequential composition `<*>` we defined [1] a permuting combinator, with the same type:

```
<||> ::Parser (b → a) → Parser b → Parser a
```

This operator also recognises both its operands, but irrespective of the order in which they occur in the input stream, i.e. we may either run its left operand and have its right operand take off where the first one finished or the other way around.

Using this new combinator we can construct parsers which recognise input in which the components are possibly permuted. Examples of such uses abound: the order of the fields in a BibTeX entry is not fixed, nor is the order of the specification of the field values in a Haskell record fixed. Suppose we have defined the type:

```
data Cart = Cart  {x   :: Float; y   :: Float}
          | Polar {rho :: Float; phi :: Float}
```

Now one might want to be able to read both `Cart{x=1,y=2}` as well as `Cart{y=2,x=1}` from the input; if a Haskell compiler accepts both representations, why shouldn't we be able to read both these representations from the input? Using the `<||>` parser this is easily achieved (*mkG* and *sepBy* are helper functions to which we will come back later):

```
pTwoFloats op constr s1 s2
  =     pToken constr *> pCurly ((op <$> pField s1 pFloat
                                     <||> pField s2 pFloat)
                                 `sepBy` pSym ','
                                )
pField s p = mkG (pToken s *> pSym '=' *> p)
pCurly   p = pSym '{' *> p <* pSym '}'
pCart =      pTwoFloats Cart  "Cart"  "x"   "y"
       <|> pTwoFloats Polar "Polar" "rho" "phi"
```

We [3] furthermore defined a combinator `<+>` which makes it possible to construct parsers which recognise merged lists. In parser *int_low* the *listOf* function

converts a parser to a parser which may run interleaved with other parsers and recognises a list values accepted by its argument parser, the function `<+>` runs both arguments in an interleaved mode, and *pMerged* converts its argument back to a normal parser again:

```
int_low :: Parser ([Int], [Char])
int_low = pMerged (listOf pInt <+> listOf pLower)
```

The function *int_low* accepts the input string `"a1bc2"` and returns the nested pair $([1,2],$`"abc"`$)$; the input is split into two sub-streams which are each accepted by one of the two operands of `<+>`.

On closer inspection we see that the list merging parsers are a generalisation of the permuting parsers: just make sure that each of the merged lists has precisely length 1 and list merging degenerates to permuting. This made us also realise that the merging of lists was just a special case of *merging any collection of structured sequences*, which in its turn raised the question whether we can define combinators which make it possible to describe this process, thus generalising the two libraries just mentioned into a single one.

In this paper we present such a library. As we will see, this library can be completely constructed using the basic *Applicative* and *Alternative* parser combinators as e.g. provided by the `uu-parsinglib` package, without having to deal with the intricate internals of the parsing process itself. Of course, A direct consequence of choosing a feature rich library as $uu - parsinglib$ is that the merging parsers constructed on top of this inherit all its nice properties: returning results in an online way, providing informative error messages, and modify the input where parsing cannot proceed. In the rest of this paper we will however put minimal requirements on the underlying parsing techniques; we only require that it is possible to split a parser in two components: one which recognises the empty sequence if the original parser can do so and one which will take care of the non-empty cases.

Before we introduce our library we will give an somewhat larger example as further motivation for the library to be derived. Finally, once we have introduced our library we conclude with some code which, based on the just defined library, facilitates the repetitive task of processing a file with options or processing command line arguments. We will see that in the end there is not much more work left for the programmer than just enumerating the possible options and the way they are to be denoted.

2 Parsing Log Files

As a motivation for our new combinators we start with a simple example of their use. It deals with unraveling a file which was created by several concurrent processes writing entries into it: each process indicates its start by writing out a `'s'` character followed by its unique identity (numbers in the example), next it will emit a couple of work entries each consisting of a `'w'` character followed by its process identity with the rest of the line containing log information, and

it closes its sequence of entries with a line consisting of a `'c'` character and its process identity. So all entries generated by a single proces will be labeled with the same process identity, which we will assume to be unique throughout the log file. So an example input will be something like:

```
s2        -- start of process 2
s1        -- start of process 1
w1 a1     -- first work entry of process 1
w2 b      -- first work entry of process 2
w1 a2     -- second work entry of process 1
c1        -- process 1 closes
s3        -- start of process 3
w3 c      -- first work entry of process 3
c2        -- process 2 closes
c3        -- process 3 closes
```

As a result we produce the following table:

$$[(\texttt{"2"},[\texttt{"b"}]),(\texttt{"1"},[\texttt{"a1"},\texttt{"a2"}]),(\texttt{"3"},[\texttt{"c"}])]$$

In the code we use names starting with a g to bind to parsers which can run in an interleaved way (which we will refer to as *grammars* from now on), whereas names starting with a p refer to conventional parsers which recognise a consecutive segment of input. The function *mkG* converts a conventional parser to a grammar; the result will still recognise a consecutive part of the input, but once it has done so it may pass control on to a competing parser.

We start out by defining the grammar for the sequence of events for a single process, using some of the basic parsers provided by common combinator libraries providing an applicative interface, and subsequently use the function *gmList* to concurrently run as many of them as needed. We have used a monadic bind to use the process identity retrieved from the starting line in constructing the parsers for the subsequent lines of this process. The function *pMunch* accepts the longest prefix of the input which passes the passed predicate.

```
gLog     = gmList gProcess
gProcess = do i  <-          mkG  pStart
              w  <- pMany .  mkG $ pWork i
              _  <-          mkG $ pClose i
              return (i, w)
pStart   = pToken "s" *> pMunch (/= '\n') <* pToken "\n"
pWork i  =    pToken ('w' : i ++ " ")
           *> pMunch (/= '\n')   -- read the rest of the line
           <* pToken "\n"
pClose i = pToken ('c' : i ++ "\n")
```

Note that the order in which the work comes out is determined by the order in which the left operands in the use of the *gmList* start: the elements are thus ordered by the ordering of their first entry in the log file.

3 Merging Parsers

3.1 Representing parsers for merged input structures

As we have seen in the example, what we are looking for is the possibility interleave parsers; one can think of this as associating a separate colour with each parser and splitting up the input in a series of coloured segments, and making that the concatenation of segments of the same colour is accepted by its correspondingly coloured parser.

The question to answer is how to split the input into uniformly coloured segments, i.e. how to find the points in the input where colour switching is to take place and what to colour to give to the segments. In order to be able to do so efficiently we have decided to construct our interleaving parsers out of *basic* parsers, which are conventional parsers which recognise a *consecutive part* of the input. We rephrase our use of the word *grammar* to stand for a *description of* a parser which can pause at a colour switch and continue when the input switches back to its colour again. Such grammars can thus be run in a competing fashion. We will use the word *parser* to refer to something which recognises a uniformly coloured segment of the input.

Once a parser gets hold of the input, and has successfully started parsing, we will let this parser run until completion. Once it has completed, all pending grammars (including the grammar for which the parser which just ran forms a constituent) can again try to continue to parse. A consequence of this choice is that at any point in the input where we may switch between grammars each of the competing grammars should present its *first-parsers*, i.e. those parsers which are candidates for accepting the next uniformly coloured segment.. These presented first-parsers play a similar role as the *first* sets resulting from an $LL(1)$ grammar analysis; the only difference is that we compute the collection of first-parsers instead of the set of first symbols.

A very important issue we have to take care of is unwanted ambiguity. Suppose we have the following permuting parser:

$$pMaybe\ s = pToken\ s\ `opt`\ \texttt{""}$$
$$pAmb = (,)\ \texttt{<\$>}\ pMaybe\ \texttt{"A"}\ \texttt{<|>}\ pMaybe\ \texttt{"B"}$$

and our input consists of the string `"A"`, then this may lead to unwanted or unexpected behaviour if we do not take any further precautions: there are two different ways in which the empty string `""` can be seen as part of the input: $\texttt{""} +\!\!+ \texttt{"A"}$ and $\texttt{"A"} +\!\!+ \texttt{""}$. Thus when running the parser *pAmb* on the input string `"A"` there are two possible parses, both returning the same result. Unless our underlying parsing library somehow knows how to deal with ambiguous parsers this will lead to an error message or a rather arbitrary choice of one of the possible parses. Even if the library could handle ambiguity it may not be able to discover that both rsults are the same. This problem has already been described in the development of the permuting combinators [1] and the solution we take here is based on a similar assumption as we have chosen there: we assume that we are able to split a parser in a part recognising the non-empty part and one

recognising the possibly empty part. The latter will be represented by the value that is to be returned as as a witness in case the parser accepts the empty input (denoted as ϵ from now on). In order to make this explicit in our interface we introduce a class:

```
class Splittable f where
  getNonPure :: f a → Maybe (f a)
  getPure    :: f a → Maybe    a
```

where we will assume the following equality to hold for any *Applicative* and *Alternative* functor f:

```
f ≡ maybe empty id (getNonPure f) <|> maybe empty pure (getPure f)
```

We are now reday to define the data type *Gram* which we use to represent *grammars*. It allows us to compute, for each possible splitting point in the input, the collection of parsers that can continue at that point, provided that the input allows this. We start out by defining a data type *Alt* which has a constructor *Seq* which explicitly represents the splitting of the grammar into in a first-parser to be run and "the rest of the work to be done". Since we also want our grammars to have a monadic interface we equip our data type *Alt* a *Bind* alternative, which again explcitly contains the first-parser to be run:

```
data Alt f a = forall c.(f (c → a)) `Seq`  (      Gram f c)
             | forall c.(f c)       `Bind` (c → Gram f a)
```

Based on the data type *Alt* we now define the data type *Gram*. It contains two components: a value of type $[Alt\ f\ a]$ which is the alternation (choice) of all non-empty parts jointly accepting a non-empty sequence of symbols, and a *Maybe a* value representing the value to be returned in case ϵ is accepted:

```
data Gram f a = Gram [Alt f a] (Maybe a)
```

The type parameter f corresponds to the conventional, non-interruptible parsers, which we use as building blocks for our grammars.

3.2 Defining the various class instances for *Gram*

We will now define the instances for these newly introduced data types for the classes *Functor*, *Applicative*, *Alternative* and *Monad*.

Gram* is a *Functor We start with the instances for *Functor* for both the data type *Gram* and *Alt*. The code is straightforward:

```
instance Functor f ⇒ Functor (Gram f) where
  fmap b2a (Gram l_b m_b) = Gram (map (b2a<$>) l_b) (b2a <$> m_b)
instance Functor f ⇒ Functor (Alt f) where
```

$$fmap\ b2a\ (f_{c2b}\ `Seq`\ g_c\ \ ) = ((b2a\circ)\ \texttt{<\$>}\ f_{c2b})\ `Seq`\ g_c$$
$$fmap\ b2a\ (f_c\ `Bind`\ c2g_b) = f_c\ `Bind`\ (\lambda c \to b2a\ \texttt{<\$>}\ c2g_b\ c)$$

Here we have chosen a naming convention which makes the types of the values involved explicitly visible in the program text: *b2a* holds a function value of the type $b \to a$, m_b stands for a value of type *Maybe* b, l_a stands for a list of *Alt* f a values, f_{c2b} is bound to a value of type $f\ (c \to b)$, g_c is bound to a value of type *Gram* f c and $c2g_b$ to a value of type $c \to$ *Gram* b. Now it is easy to see that *fmap b2a* $(fc\ `Bind`\ c2g_b)$ should result in a value of type *Gram* a, etc. Note that applying *fmap* maintains the invariant that the first-parser can be easily recognised for each *Alt* intact. We have chosen to use `<$>` as an alias for *fmap* in this code wherever possible, since this makes the names chosen even more helpful in the representation of the code.

***Gram* is an *Applicative* functor** We want to construct our merging parsers in just the same way as we construct normal parsers, using the well known *Applicative* and *Alternative* interfaces [4]. Since our data types enforce the property that we can easily identify the first-parser to be used this is where the real work takes place. The pattern we follow however is very common, and well known from various process algebras and given in Figure 1. The first-parsers in the *Seq* and *Bind* constructs of the left-hand side operand are "rotated" out. The hard work is done by the function *fwdby* which combines the remaining part of the *Seq* and *Bind* constructs to form the new right-hand side of the *Seq* and *Bind* construct in the result. The g_c and g_b grammars of a *Seq* are ran returning a value of type (c, b). We modify the value returned by the first-parser, by applying *uncurry* to it, to accept this pair of values instead of getting passed the two arguments individually. If the left-hand side parser can recognise the empty input then also the first-parsers of the right-hand side grammar are first-parsers of the resulting grammar since they can start to accept part of the input. This explains the second component in the definition of the *Alts* f b in the right-hand side of the `<*>` definition, where we use the witness value of type $b \to a$ to update the result of the right-hand side parser. The definition of *pure* speaks for itself: we have no non-empty alternatives, and the parser can recognise the empty input with a witness of type a.

A subtle point is that we had to add an irrefutable pattern match ($\sim$) to the right-hand side operand of `<*>`, since otherwise the pattern matching creates an endless loop in situations like the definition of *pMany* when f instantiated with some *Gram* g:

$$pMany\ p :: f\ a \to f\ [a]$$
$$pMany\ p = \textbf{let}\ result = (:)\ \texttt{<\$>}\ p\ \texttt{<*>}\ result\ `opt`\ [\,]\ \textbf{in}\ result$$

Here `<*>` does not have access to the top-level constructor in its right-hand side, since this constructor is produced by this very call to `<*>`. Note that the call to `<*>` is evaluated lazily, so we have no problems with recursive grammars. The unrolling of these definitions is done on a by-demand basis during the actual

```
instance Functor f ⇒ Applicative (Gram f) where
  pure a = Gram [] (Just a)
  Gram l_b2a m_b2a <*> ~g_b@(Gram l_b m_b)
    = Gram ( map (`fwdby` g_b) l_b2a
                 ++
             [b2a <$> f_b | Just b2a ← [m_b2a], f_b ← l_b]
           ) ( m_b2a <*> m_b)
fwdby :: Functor f ⇒ Alt f (b → a) → Gram f b → Alt f a
(f_c2b2a `Seq` g_c)      `fwdby` g_b = (uncurry <$> f_c2b2a) `Seq` ((,) <$> g_c <*> g_b)
(f_c     `Bind` c2g_b2a) `fwdby` g_b = f_c `Bind` (λc → c2g_b2a c <*> g_b)
uncurry f (x, y) = f x y
instance Functor f ⇒ Alternative (Gram f) where
  empty = Gram [] Nothing
  Gram ps pe <|> Gram qs qe = Gram (ps ++ qs) (pe <|> qe)
```

Fig. 1. *Gram* is a member of the *Applicative* and *Alternative* classes.

parsing process. This technique resembles the parallel parsing strategy as developed by Claessen [2] and code also bears close resemblance to the computation of the *firsts* set, as described by Swierstra and Duponcheel [5].

In case the left-hand side is a monadic construct we just use the original first-parser f_c, and again move the right-hand side $c2g_{b2a}$ of the left operator to the right-hand side of the result, where it is composed with the original right hand side using `<*>`.

***Gram* is an *Alternative* functor** The instance for *Alternative* is almost trivial: we concatenate the list of alternatives from both operands. This leaves the question what to do if the grammar is ambiguous, caused by both alternatives to be able to accept ϵ . We have chosen to use the left-biasedness of `<*>` as defined for *Maybe* to choose the left value to return. The code is given in figure 1.

Gram* is a *Monad The next thing we want to do is to equip our grammars with a monadic interface, which we again achieve by "rotating" all but the first-parser to the right so the first-parser again is presented at the top level constructor. In the case of a '*Seq*' construct as the left argument of $\gg=$ we split its monadic effect into two steps: in the first step we make the first part if the left-hand side operand explicitly visible in the resulting '*Bind*' construct, whereas the corresponding right-hand side g_c of the '*Seq*' part is, once it has been combined using $\lambda c2b \rightarrow c2b$ `<$>` g_c with the result $c2b$ of the left-hand side of the '*Seq*' in another call to $\gg=$ in the right-hand side of the resulting top level '*Bind*' constructs.

When the left-hand side accepts ϵ we have to take some extra precautions, since in this case the first-parsers created by a call to right-hand side compete for input too, since they too may accept input at this point. Since we have the

witness of this empty left-hand side available, we can use it to compute the *Gram a* value returned by the right-hand side of $\gg=$, and with this its first-parsers become available too and can take part in the competition:

```
instance Functor f ⇒ Monad (Gram f) where
  return a = Gram [] (Just a)
  Gram l_b m_b >>= b2g_a = case m_b of
    Nothing → Gram (map (`bindto` b2g_a) l_b) Nothing
    Just b  → let Gram l_a m_a = b2g_a b
              in Gram (map (`bindto` b2g_a) l_b ++ l_a) m_a
bindto :: Functor f ⇒ Alt f b → (b → Gram f a) → Alt f a
(f_c2b `Seq` g_c)      `bindto` b2g_a = f_c2b `Bind` λc2b → c2b <$> g_c >>= b2g_a
(f_c   `Bind` c2g_b) `bindto` b2g_a = f_c   `Bind` λc    → c2g_b c    >>= b2g_a
```

Constructing elementary *Gram* values The last thing we have to do is to show how we can lift a parser into an equivalent *Gram*

```
mkG :: Splittable f ⇒ f a → Gram f a
mkG p = Gram (maybe [] (λp → [(const <$> p) `Seq` pure ()]) (getNonPure p))
             (getPure p)
```

At first sight this code looks more complicated than strictly needed; this is caused by our choice of the *Alt* data type. We could have easily added a third case *Single* (*f a*) to this data type, and have used this *Single* constructor here. We have chosen for the current, minimalistic approach, in which this *Single* data type is encoded as a parser which is to be followed by an ϵ parser returning () the result of which is subsequently discarded. This adds a small constant-time overhead to our parsers, which we think is acceptable in return for the increased simplicity of the code.

3.3 <||> and <<|>

In the previous subsections we have defined the data type *Gram f a*, have shown how to lift elementary parsers to this structure, and have defined instances for this type for the *Functor*, *Applicative*, *Alternative* and *Monad* classes. As a final step we now define the <|> combinator, which describes the "interleaved" composition of two grammars. We will however express this combinator in terms of an even more primitive combinator <<|>:

```
infixl 4 <|>
infixl 4 <<|>
(<|>), (<<|>) :: Gram (b → a) → Gram b → Gram a
```

Note that we have given these operators the same type as the conventional <*> combinator, since we very much like the applicative interface for describing how

to combine the two accepted values into the result. The reason, of course, that we cannot use the <*> combinator is that it has already been used to express the more conventional sequential composition of two *Gram* values.

The idea of the <<|> combinator is that it will run one of the first-parsers of its left-hand side operand on the input first, and from that point on will behave like <|>, which does not have a preference for either of its operands to start accepting input. We can easily define <|> in terms of <<|>:

$$
\begin{aligned}
&g_{b2a} \mathbin{<|>} g_b = g_{b2a} \mathbin{<<|>} g_b \\
&\quad \mathbin{<|>} \mathit{flip}\ (\$) \mathbin{<\$>} g_b \mathbin{<<|>} g_{b2a}
\end{aligned}
$$

Here we see that the resulting parser will either run one of the first-parsers from its left-hand side operand or one of the first-parsers of its right-hand side operand. In case both grammars can accept ϵ we get the same witness value twice, and in principle our grammar becomes ambiguous; the biased choice of the *Maybe* instance of *Alternative* throws away one of these two (equal) values.

So all we have to do now is to define <<|>. We construct a new grammar which has as its first-parsers all the first-parsers of its left-hand side operand:

$$
\begin{aligned}
&(\mathbin{<<|>}) :: \mathit{Functor}\ f \Rightarrow \mathit{Gram}\ f\ (b \to a) \to \mathit{Gram}\ f\ b \to \mathit{Gram}\ f\ a \\
&g_{b2a} @ (\mathit{Gram}\ l_{b2a}\ m_{b2a}) \mathbin{<<|>} \sim g_b @ (\mathit{Gram}\ \_\ m_b) \\
&\quad = \mathit{Gram}\ (\mathit{map}\ (`\mathit{fwdby'}` g_b)\ l_{b2a})\ (m_{b2a} \mathbin{<*>} m_b) \\
&(f_{c2b2a}\ `\mathit{Seq}`\ g_c) \quad `\mathit{fwdby'}`\ g_b = (\mathit{uncurry} \mathbin{<\$>} f_{c2b2a})\ `\mathit{Seq}`\ ((,) \mathbin{<\$>} g_c \mathbin{<|>} g_b) \\
&(f_c\ `\mathit{Bind}`\ c2g_{b2a})\ `\mathit{fwdby'}`\ g_b = fc\ `\mathit{Bind}`\ (\lambda c \to c2g_{b2a}\ c \mathbin{<|>} g_b)
\end{aligned}
$$

Notice that this code is almost the same as that for the definition of <*>; only have the occurrences of <*> in the right-hand sides of the *fwdby* function been replaced by <|>, thus indicating that thus constructed parsers should run interleaved instead of sequentially.

3.4 Converting Grammars into Parsers

The only thing left to do now it to show how to construct a real parser from a *Gram* structure. We will require that the parameter f we have carried around thus far has instances for the usual *Applicative*, *Alternative* and *Monad* interfaces, so we can use the functions available from these classes in this process. For each of the alternatives we select the first-parser from it, use <|> to select one of these to run, and after that either combine its result using <*> with the parser generated from the corresponding *Gram* value in the case of a '*Seq*', or use it as an argument to the right-hand side operand in the case of a '*Bind*' and convert this result again into a proper parser.

$$
\begin{aligned}
&\mathit{mkP} :: (\mathit{Monad}\ f, \mathit{Applicative}\ f, \mathit{Alternative}\ f) \Rightarrow \mathit{Gram}\ f\ a \to f\ a \\
&\mathit{mkP}\ (\mathit{Gram}\ l_a\ m_a) = \mathit{foldr}\ (\mathbin{<|>})\ (\mathit{maybe}\ \mathit{empty}\ \mathit{pure}\ m_a) \\
&\qquad\qquad\qquad\qquad (\mathit{map}\ \mathit{mkP\_Alt}\ l_a) \\
&\quad \mathbf{where}\ \mathit{mkP\_Alt}\ (f_{b2a}\ `\mathit{Seq}`\ g_b) \qquad = f_{b2a} \mathbin{<*>} \mathit{mkP}\ g_b \\
&\qquad\quad \mathit{mkPrAlt}\ (f_b \quad `\mathit{Bind}`\ b2g_a) = f_b \quad \gg\!= (\mathit{mkP} \circ b2g_a)
\end{aligned}
$$

3.5 Inserting separators

As we have seen in the *pCart* example it is a common case that the elements which we want to recognise, and which occur in a permuted order are separated by e.g. a `';'` or a `','`. For these cases we have introduced a special version of *mkP*, which takes an additional argument telling how to parse a separator. The hard work is done by a function *insertSep* which prefixes each parser, except the first one, in the *Gram* parameter by the parser that recognises the separator:

```
sepBy :: Applicative f ⇒ Gram f a → f b → f a
sepBy g sep = mkP (insertSep sep g)
insertSep :: (Applicative f) ⇒ f b → Gram f a → Gram f a
insertSep sep (Gram l_a m_a :: Gram f a) = Gram (map insertSepInAlt l_a) m_a
  where
    insertSepInAlt (f_b2a `Seq` g_b) = f_b2a `Seq` prefixSepInGram g_b
    insertSepInAlt (f_b `Bind` b2g_a) = f_b `Bind` (insertSep sep ∘ b2g_a)
    prefixSepInGram (Gram l_a m_a) = Gram (map prefixSepInAlt l_a) m_a
    prefixSepInAlt :: Alt f b → Alt f b
    prefixSepInAlt (f_b2a `Seq` g_b) = (sep *> f_b2a) `Seq` prefixSepInGram g_b
```

Because we are making use of polymorphic recursion we had to insert a few type annotations in the code.

3.6 Parsing merged lists

Although the combinators follow the common interfaces, there are a few tricky points one has to keep in mind when using them. The fact that the interleaved parsers compete for input may lead to some complications one should be aware of. We take a look at the traditional definition of *pMany*, which converts a parser into a parser which recognises a list of elements recognised by its argument parser:

```
pList p = let pmp = (:) <$> p <*> pmp `opt` [] in pmp
```

In this definition the recursive call to *pmp* only starts to play a role once the first instance of *p* has succeeded. If we however replace the `<*>` operator by a `<||>` operator, then the recursive *pmp* can start to parse immediately too, and will spawn yet another instance of *p* which starts to compete for the input and so on recursively; apparently changing sequential execution by interleaved execution has deeper implications than is directly visible from the code. Fortunately this problem can be solved rather easily: we decide to only start with a new instance of *pmp* competing for the input, once *p* has started its work and has comsumed a bit of input. Hence we define:

```
gmList p = let pmp = (:) <$> p <<||> pmp `opt` [] in pmp
```

We see here that the availability of `<<||>` plays an essential role in this definition; in that sense it is more primitive than `<||>`, which was expressed in terms of `<<||>`.

4 Applications

In this section we will give two examples of the use of the introduced merging combinators.

4.1 Parsing options

One of the most boring tasks in writing an application is the processing of the options passed on the command line. Although there are some packages and tools to make one's life a bit easier, there always remains a lot of work to be done. Usually conversion from the strings which were passed to the kind of values one is really interested in has to be done explicitly, for optional arguments defaults have to be provided, for required arguments we have to check that they have actually been provided, and there are many conventions for passing the options, be it in short form as in `ls -l`, in long form as in `haddock --enable-documentation`, in a kind of key-value pair as in `process -o outputfile` or `process -o=outputfile`, etc. We will now present a small collection of combinators which completely takes away this burden from the programmer, using the introduced merging parser combinators.

We start out by assuming that in our program we want to put our recognised options in a record with named fields. Using Template Haskell we generate lenses to give us access to the individual fields. As an example we define the following data types and example record:

```
import Data.Lenses
import Data.Lenses.Template
data Prefers = Clean | Haskell deriving Show
data Address = Address { city_ :: String, room_ :: String }
               deriving Show
data Name    = Name { name_ :: String, prefers_ :: Prefers, ints_ :: [Int]
                    , address_ :: Address }
               deriving Show
$ (deriveLenses '' Name)
$ (deriveLenses '' Address)
$ (deriveLenses '' Prefers)
defaults = Name "Doaitse" Haskell [] (Address "Utrecht" "BBL517")
```

The *deriveLenses* calls to template Haskell generate code which will give us read and write access to the fields which a name which ends in an `'_'`. What is precisely generated does not matter too much here, but what is important is that the imported packages provide amongst others a function *alter* which can be used to update a field of type a pointed at by the first parameter in a record of type r by applying the passed function of type $a \to a$ to it:

```
alter :: MonadState a m ⇒ (m () → StateT r Identity b) → (a → a)
                                                       → (r → r)
```

Now we can update the record at say the field *clean* as follows:

```
> print ((prefers `alter` (const Clean)) defaults)
> Name { name_ = "Doaitse", prefers_ = Clean, ints_ = [],
         address_ = Address { city_ = "Utrecht", room_ = "BBL517" } }
```

So the important thing to remember is that an expression a `` `alter` `` f applies the function f to a field pointed at by the lens a.

The first thing we do is to define a function oG, which stands for *optionGrammar*, which takes a normal parser p which parses a single option and modifies the parser such that a function f is applied to its result, and finally uses the passed lens to apply this result to the field pointed at by the lens:

```
oG p a = mkG ((a`alter`) <$> p)
```

Using this code we can define an option parser which recognises a required option which takes a single extra parameter, such as e.g. filename.

```
required_ a f (string, p)
  =     oG (f <$> pSymbol ("-"          ++ [head string]) <*> lexeme p) a
    <|> oG (f <$> pSymbol ("--"         ++ string ++ "  ") <*> lexeme p) a
    <|> oG (f <$> pSymbol ("--"         ++ string ++ "=") <*> lexeme p) a
required a  string_p = required_ a const string_p
```

The call *required* `"filename"` *pFileName inp* will construct a grammar which is able to recognise one occurrence of the three possible forms of passing an option: `-f inputfile`, `--filename inputfile` and `--filename=inputfile`, and will update the field with name *inp* in the record which will hold our recognised options. The parser *pFilename* will recognise the file name part of the option.

Using this basic parser for a single option we can now define special versions of it. The function *option* makes the required field optional, as its name suggests. The function *flag* recognises an option which does not take an extra argument, but returns *const True* as result, which can be used to switch a field to *True*:

```
option a string_p    = required a string_p `opt` id
flag   a (string, v) = option   a (string, pure v)
flags  a table       = foldr (<>) (pure id) (map (flag a) table)
```

At this point one may say that the code we have presented thus far does not really depend on the fact that we have introduced grammars, and could have been implemented using the permutation parsers which have been available for a long time (see e.g. the `options-applicative` or the `uu-parsinglib` packages on hackage). We now come to the point where our somewhat more involved combinators will start to pay of. In the example record we see that we have a field which holds a list of integers, and wouldn't it be nice if these integers could each be specified by a separate item in the list of options? For this we define the functions:

```
options  a string_p = pFoldr ((∘), id) (required_ a (:) string_p)
optionsl a string_p = (last∘)  <$> options a string_p
optionsf a string_p = (head∘) <$> options a string_p
```

Use of each of these functions will make that several settings, distributed over the total list of options, will be recognised and combined into a value for a single field, to which they will be prepended. The functions *optionsl* and *optionsf* select the last, respectively the first element of this list. They can be used in the case where the same option may be set several times and we want to use the last setting or the first one. This may come in handy when the total list of options consists of e.g. a sequence of options as typed on the command line concatenated with a list of options taken from some preferences file.

Finally we want to be able to define the options for the *Address* field which is part of our *Name* record holding the recognised options. Using lenses this becomes extremely easy again. We recognise a list of options, and apply these to a specific field of the parent record pointed at by the passed lens:

```
field s opts = (s`alter`) <$> opts
```

So now we are finally ready to show how the specification of the option parser for our *name* record may look like (note that the order in which we specify the options does not matter):

```
instance Functor f ⇒ Monoid (Gram f (r → r)) where
  mappend p q = (∘) <$> p <||> q
  mempty = empty
flags table a = foldr (<>) (pure id) [flag text val a | (text, val) ← table]
oName =      name    `option`  ("name", pString)
        <> ints      `options` ("int",  pNatural)
        <> prefers   `flags`   [("clean",    Clean)
                               ,("haskell", Haskell)]
        <> address   `field`   (    city   `option` ("city", pString)
                                <> room   `option` ("room", pString)
                               )
```

By making values of type *Gram f* $(r \to r)$ an instance of the class *Monoid*, by defining *mappend* as the merge of the two parameters, and composing the record updating functions returned by both parameters when used as parser, we can use the nice <> notation to combine the options. We can now finally run our options as follows:

```
run (($ defaults) <$> mkP oName)
    "--name=Rinus --int=7 --city=Nijmegen -i 5 --clean -i3"
-- Result: Name { name_ = "Rinus"
                , prefers_ = Clean
                , ints_ = [7,5,3]
                , address_ = Address { street_ = "Nijmegen"
                                     , room_ = "BBL517"}}
```

Note that the specifications for the nested *Address* field appear distributed among the rest, and that all the integers we have specified end up together in the *ints* field.

In the definition of the derived combinators used for specifying specific variants of options we have chosen to use lenses, which return a function which updates an already existing structure containing the values to be collected. The advantage of this approach is that we may start with a record containing the default values and apply the result of reading a preferences file to it, and next apply on top of that the extra information parsed from the command line. Note that each of these structures can be easily parsed using the newly introduced combinators, be it using parsers described in an applicative style or using lenses, which correspond more closely to the familiar keyword-value way of specifying parameters.

4.2 Nested options

It is not unfamiliar to pass options for a linker that is to be involved in a later stage on the command line of a compiler. One of the problems which may arise from such an option architecture is that the option specifications of otherwise rather independent programs may start to interfere; is e.g. the `--verbose` option passed to a module installer like *cabal* meant for the installer itself, or for the *haddock* program that is used to generate the associated documentation? This problem can be easily solved by requiring these options to be surrounded by `+haddock` and `-haddock` markers on the command line, and specifying the command line parser as follows:

$$
\begin{aligned}
pHaddock = (gList \circ mkG)\ \ (\ & pToken\ \texttt{"+haddock"} \\
\texttt{*>}\ & mkP\ (<haddock\ options>) \\
\texttt{<*}\ & pToken\ \texttt{"-haddock"} \\
)
\end{aligned}
$$

5 Conclusions

We have derived a set of very general combinators which make it possible to unravel merged input structures. The library imposes very few restrictions on the underlying parser combinators used. A distinguishing feature of our combinators is that they extend beyond the now common parsers used for permuted structures. Our new combinators have both a monadic and an applicative interface. By being able to switch between the sequential and the merging view on the input we can recognise permuted structures which are embedded inside other permuted structures. In this way we may specify options for various different subsequent program stages, where the option structures for these programs would otherwise conflict.

In deriving the library we found it very helpful to choose our identifiers in such a way that the types are explicitly represented in the chosen identifiers.

As we will see his greatly helps in identifying the way we can construct the needed values out of the available values. Choosing the identifiers in the way we did greatly helped us in writing the code, and is an essential ingredient in applying the programming paradigm in which we *"Let the types do the work"*. Our experience in programming in this way helped us so much, how trivial this observation may seem, that we think it deserves being pointed at explicitly rather than just being used in the construction of this library.

6 Acknowledgement

We want to thank Bastiaan Heeren and members of the Software Technology reading club for commenting on earlier versions of this paper.

References

1. Baars, A.I., Löh, A., Swierstra, S.D.: Parsing permutation phrases. J. Funct. Program. 14(6), 635–646 (2004)
2. Claessen, K.: FUNCTIONAL PEARL Parallel Parsing Processes. Journal of Functional Programming 14(6), 741–757 (2004)
3. Guerra, R., Baars, A.I., Swierstra, S.D., Saraiva, J.: Preserving order in non-order preserving parsers. Tech. Rep. UU-CS-2005-025, Department of Information and Computing Sciences, Utrecht University (2005)
4. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming 18(01), 1–13 (2007), `http://portal.acm.org/citation.cfm?id=1348940.1348941`
5. Swierstra, S.D., Duponcheel, L.: Deterministic, error-correcting combinator parsers. In: Launchbury, J., Meijer, E., Sheard, T. (eds.) Advanced Functional Programming. LNCS-Tutorial, vol. 1129, pp. 184–207. Springer-Verlag (1996)