FUNCTIONAL PEARL

Parsing Permutation Phrases

Arthur Baars [1], Andres Löh [2], S. Doaitse Swierstra [3]

Institute of Information and Computing Sciences,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

Abstract

A permutation phrase is a sequence of elements (possibly of different types) in which each element occurs exactly once and the order is irrelevant. Some of the permutable elements may be optional. We show a way to extend a parser combinator library with support for parsing such free-order constructs. A user of the library can easily write parsers for permutation phrases and does not need to care about checking and reordering the recognised elements. Possible applications include the generation of parsers for attributes of XML tags and Haskell's record syntax.

1 Introduction

Parser combinator libraries for functional programming languages are well-known and subject to active research. Higher-order functions and the possibility to define new infix operators allow parsers to be expressed in a concise and natural notation that closely resembles the syntax of EBNF grammars. At the same time, the user has the full abstraction power of the underlying programming language at hand. Complex, often recurring patterns can be expressed in terms of higher-level combinators.

A specific parsing problem is the recognition of permutation phrases. A permutation phrase is a sequence of elements (possibly of different types) in which each element occurs exactly once and the order is irrelevant. Some of the permutable elements may be optional. Since permutation phrases are not easily expressed by a context-free grammar, the usual approach is to tackle this problem in two steps: first parse a relaxed version of the grammar, then check whether the recognised elements form a permutation of the expected elements. This method, however, has a number of disadvantages. Dealing

[1] Email: `arthurb@cs.uu.nl`
[2] Email: `andres@cs.uu.nl`
[3] Email: `doaitse@cs.uu.nl`

with a permutation of typed values is quite cumbersome, and the problem is often avoided by encoding the values in a universal representation, thus adding an extra level of interpretation. Furthermore, because of the two steps involved, error messages cannot be produced until a larger part of the input has been consumed, and special care has to be taken to make them point to the right position in the code.

Permutation phrases have been proposed by Cameron [1] as an extension to EBNF grammars, not aiming at greater expressive power, but at more clarity. Cameron also presents a pseudo-code algorithm to parse permutation phrases with optional elements efficiently in an imperative setting. It fails, however, to address the types of the constituents.

We show a way to extend any existing parser combinator library with support for parsing permutations of a number of typed, potentially optional elements. Our approach uses Haskell, relying essentially on existentially quantified types, that are used to encode reordering information that permutes the recognised elements to a canonical order. Existential types are not part of the Haskell 98 standard [6], but are, for example, implemented in GHC and Hugs. Additionally, we utilise lazy evaluation to make the resulting implementation efficient. The administrative part of parsing permutation phrases has a quadratic time complexity in the number of permutable elements. The size of the code, however, is linear in the number of permutable elements.

Possible applications include the implementation of Haskell's *read* function where it is desirable to parse the fields of data types with labelled fields in any permutation, the parsing of XML tags which have large sets of potentially optional attributes that may occur in any order, and the decomposition of a query in a URI, consisting of a number of permutable key-value pairs.

The paper is organised as follows: Section 2 explains the parser combinators we build upon. Section 3 presents the basic idea of dealing with permutations in terms of permutation trees and explains how trees are built and converted into parsers. Section 4 shows how to extend the mechanism in order to handle optional elements. In Section 5, we take a brief look at two of the applications mentioned above, the parsing of data types with labelled fields and the parsing of XML attribute sets. Section 6 concludes.

2 Parsing using combinator libraries

The use of a combinator library for describing parsers instead of writing them by hand or generating them from a separate formalism is a well-known technique in functional programming. As a result, there are several excellent libraries around. For this reason we just briefly present the interface we will assume in subsequent sections of this paper, but do not go into the details of the implementation. However, we want to stress that our extension is not tied to any specific library.

We make use of a simple arrow-style [3,9] interface that is parametrised by

```
infixl 3 <|>
infixl 4 <*>, <$>
class Parser p where
  pFail     :: p a
  pSucceed  :: a → p a
  pSym      :: Char → p Char
  (<*>)     :: p (a → b) → p a → p b
  (<|>)     :: p a → p a → p a
  (<$>)     :: (a → b) → p a → p b
  f <$> p   = pSucceed f <*> p
  parse     :: p a → String → Maybe a
```

Fig. 1. Type class for parser combinators

the result type of the parsers and assumes a list of characters as input. It can easily be implemented by straightforward list-of-successes parsers [2,10], but we also have a version based on the fast, error-correcting parser combinators of Swierstra [7,8]. A mapping to monadic-style parser combinators [4,5] or abstracting from the type of the input tokens is possible without difficulties.

The parser interface used here is given as a type class declaration in Figure 1. The function *pFail* represents the parser that always fails, whereas *pSucceed* never consumes any input and always returns the given result value. The parser *pSym* accepts solely the given character as input. If this character is encountered, *pSym* consumes and returns this character, otherwise it fails. The <*> operator denotes the sequential composition of two parsers, where the result of the first parser is applied to the result of the second. The operator <|> expresses a choice between two parsers. Finally, the application operator <$> is a parser transformer that can be used to apply a semantic function to a parse result. It can be defined in terms of *pSucceed* and <*>.

Many useful higher-level combinators can be built on top of these basic ones. A small selection that we will use later in this paper is presented in Figure 2. These parser combinators are useful if one wants to combine parsers and is interested in the result of only some of the constituents.

3 Permutation trees

3.1 Data types

Before explaining permutation parsers, we investigate how to represent permutation phrases. We decide to store the permutations of a set of elements in a rose tree.

```
data Perms p a   = Choice [Branch p a]
                 | Empty a
data Branch p a  = ∀ x. Br (Perms p (x → a)) (p x)
```

```
infixl 4 <$ , <*>, <*
(<$ )          :: Parser p ⇒ a → p b → p a
f <$ p         =  const f <$> p
(<* )          :: Parser p ⇒ p a → p b → p a
p <* q         =  const <$> p <*> q
( *>)          :: Parser p ⇒ p a → p b → p b
p *> q         =  flip const <$> p <*> q
pParens        :: Parser p ⇒ p a → p a
pParens p      =  pSym '(' *> p <* pSym ')'
```

Fig. 2. Some useful parser combinators

The data types are parametrised by a type constructor p (e.g. the parser type) and a result type a.

Each path from the root to a leaf in the tree represents a particular permutation. Figure 3 illustrates this idea for three elements a, b, c. If the permutations are grouped in such a way that different permutations with a common prefix share the same subtree, the number of choices in each node will be limited by the number of permutable elements.

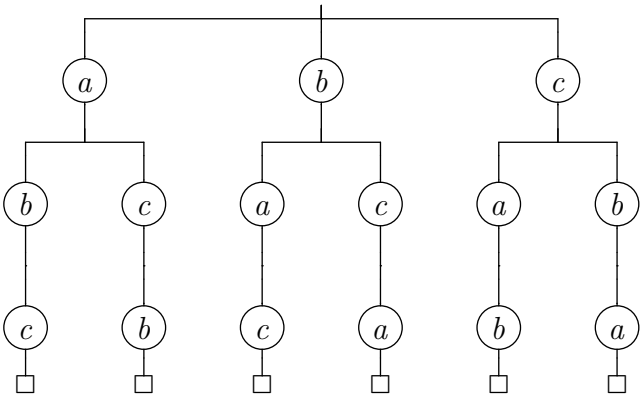

Fig. 3. A permutation tree containing three elements

A value of type *Branch* stores a subtree together with an element. The subtree returns a function that, applied to the element, computes a value of the required result type. Thus, the existentially quantified type of the element stored in a branch is used to hide the order in which the types in a subtree occur; all subtrees in a *Choice* node share a common type because all correspond to a permutation of the same set of elements. To show that reordering is almost completely determined by the type of the components, we use the convention that values, parsers and permutation trees are named v, p, and t, respectively, indexed by their type.

The idea that each path in the tree represents the parser for one of the possible permutations is reflected by the following simple conversion function from permutation trees to parsers.

$$
\begin{array}{lcl}
\mathit{pPerms} & :: & \mathit{Parser}\ p \Rightarrow \mathit{Perms}\ p\ a \to p\ a \\
\mathit{pPerms}\ (\mathit{Empty}\ v_a) & = & \mathit{pSucceed}\ v_a \\
\mathit{pPerms}\ (\mathit{Choice}\ \mathit{chs}) & = & \mathit{foldr}\ (\langle|\rangle)\ \mathit{pFail}\ (\mathit{map}\ \mathit{pars}\ \mathit{chs}) \\
\quad \textbf{where}\ \mathit{pars}\ (\mathit{Br}\ t_{x\to a}\ p_x) & = & \mathit{flip}\ (\$) \mathbin{\langle\$\rangle} p_x \mathbin{\langle\!\langle\!\$\!\rangle\!\rangle} \mathit{pPerms}\ t_{x\to a}
\end{array}
$$

It might be surprising at first sight that the last line does not read:

$$
\quad \textbf{where}\ \mathit{pars}\ (\mathit{Br}\ t_{x\to a}\ p_x) \ = \ \mathit{pPerms}\ t_{x\to a} \mathbin{\langle\!\langle\!\$\!\rangle\!\rangle} p_x
$$

But using this more obvious definition, the elements at the leaves of the permutation tree would be recognised first by the constructed parser. Therefore, the permutation tree would have to be unfolded completely before the first element could be parsed. This would result in $O(n!)$ memory usage where n is the number of permutable elements.

Fortunately, because we are parsing a permutation, reversing the order of the constituents when constructing the parser does not change the semantics. In the "flipped" variant, lazy evaluation ensures that only the path corresponding to the recognised permutation is unfolded.

3.2 Building a permutation tree

Permutation trees are created by adding the elements of the permutation one by one to an initially empty tree.

$$
\begin{array}{lcl}
\mathit{add} & :: & \mathit{Perms}\ p\ (x \to a) \to p\ x \to \mathit{Perms}\ p\ a \\
\mathit{add}\ t_{x\to a}@(\mathit{Empty}\ \_)\ p_x & = & \mathit{Choice}\ [\mathit{Br}\ t_{x\to a}\ p_x] \\
\mathit{add}\ t_{x\to a}@(\mathit{Choice}\ \mathit{chs})\ p_x & = & \mathit{Choice}\ (\mathit{first} : \mathit{others}) \\
\quad \textbf{where}\ \mathit{first} & = & \mathit{Br}\ t_{x\to a}\ p_x \\
\qquad \mathit{others} & = & \mathit{map}\ \mathit{ins}\ \mathit{chs} \\
\qquad \mathit{ins}\ (\mathit{Br}\ t_{y\to x\to a}\ p_y) & = & \mathit{Br}\ (\mathit{add}\ (\mathit{mapPerms}\ \mathit{flip}\ t_{y\to x\to a})\ p_x)\ p_y
\end{array}
$$

If we already have constructed a non-empty permutation tree, we can add a new element p_x by inserting it in all possible positions to every permutation in the tree. The function *add* explicitly constructs the tree that represents the permutation in which p_x is the top element; for each branch, the top element is left unchanged, and p_x is inserted everywhere (by a recursive call to *add*) in the subtree. Because the new element and the top element of the branch are now swapped, the function resulting from the subtree of the branch gets its arguments passed in the wrong order, which is repaired by applying *flip* to that function.

The function *mapPerms* is a map on permutation trees. In a branch, $v_{a\to b}$ is composed with the function that is resulting from the subtree.

$$
\begin{array}{lcl}
\mathit{mapPerms} & :: & (a \to b) \to \mathit{Perms}\ p\ a \to \mathit{Perms}\ p\ b \\
\mathit{mapPerms}\ v_{a\to b}\ (\mathit{Empty}\ v_a) & = & \mathit{Empty}\ (v_{a\to b}\ v_a) \\
\mathit{mapPerms}\ v_{a\to b}\ (\mathit{Choice}\ \mathit{chs}) & = & \mathit{Choice}\ (\mathit{map}\ (\mathit{mapBranch}\ v_{a\to b})\ \mathit{chs}) \\
\mathit{mapBranch} & :: & (a \to b) \to \mathit{Branch}\ p\ a \to \mathit{Branch}\ p\ b \\
\mathit{mapBranch}\ v_{a\to b}\ (\mathit{Br}\ t_{x\to a}\ p_x) & = & \mathit{Br}\ (\mathit{mapPerms}\ (v_{a\to b} \circ)\ t_{x\to a})\ p_x
\end{array}
$$

By defining the following combinators for constructing permutation parsers we can use a similar notation for permutation parsers as for normal parsers.

$$
\begin{array}{lcl}
\mathit{succeedPerms} & :: & a \to \mathit{Perms}\ p\ a \\
\mathit{succeedPerms}\ x & = & \mathit{Empty}\ x \\
\\
(\mathbin{<\!\!\!\circledast\!\!\!>}) & :: & \mathit{Parser}\ p \Rightarrow \mathit{Perms}\ p\ (a \to b) \to p\ a \to \mathit{Perms}\ p\ b \\
\mathit{perms} \mathbin{<\!\!\!\circledast\!\!\!>} p & = & \mathit{add}\ \mathit{perms}\ p \\
\\
(\mathbin{<\!\!\$\!\!>}) & :: & \mathit{Parser}\ p \Rightarrow (a \to b) \to p\ a \to \mathit{Perms}\ p\ b \\
f \mathbin{<\!\!\$\!\!>} p & = & \mathit{succeedPerms}\ f \mathbin{<\!\!\!\circledast\!\!\!>} p
\end{array}
$$

An example with three permutable elements, corresponding to the tree in Figure 3, can now be realised by:

$$\mathit{pPerms}\ ((,,)\mathbin{<\!\!\$\!\!>} \mathit{pInt} \mathbin{<\!\!\!\circledast\!\!\!>} \mathit{pChar} \mathbin{<\!\!\!\circledast\!\!\!>} \mathit{pBool})$$

Suppose *pInt*, *pChar*, and *pBool* are parsers for literals of type *Int*, *Char*, and *Bool*, respectively. Then all permutations of an integer, a character and a boolean are accepted, and the result of a successful parse will always be of type $(\mathit{Int}, \mathit{Char}, \mathit{Bool})$.

3.3 Separators

Often the permutable elements are separated by symbols that do not carry meaning—typically commas or semicolons. Consider extending the three-element example to the Haskell tuple syntax: not just the elements, but also the parentheses and the commas should be parsed. Since there is one separation symbol less than there are permutable elements, our current variant of *pPerms* cannot handle this problem.

Therefore we define *pPermsSep* as a generalisation of *pPerms* that accepts an additional parser for the separator as an argument. The semantics of the separators are ignored for the result.

$$
\begin{array}{lcl}
\mathit{pPermsSep} & :: & \mathit{Parser}\ p \Rightarrow p\ b \to \mathit{Perms}\ p\ a \to p\ a \\
\mathit{pPermsSep}\ \mathit{sep}\ \mathit{perm} & = & \mathit{p2p}\ (\mathit{pSucceed}\ ())\ \mathit{sep}\ \mathit{perm}
\end{array}
$$

The function *p2p* now converts a permutation tree into a parser almost in the same way as the former *pPerms*, except that before each permutable element a separator is parsed. To prevent that a separator is expected before the first permutable element, we make use of the following simple trick. The *p2p* function expects two extra arguments: the first one will be parsed immediately before the first element, and the second will be used subsequently. Using *pSucceed* () as first extra argument in *pPermsSep* leads to the desired result.

$$
\begin{array}{lcl}
\mathit{p2p} & :: & \mathit{Parser}\ p \Rightarrow p\ c \to p\ b \\
 & \to & \mathit{Perms}\ p\ a \to p\ a \\
\mathit{p2p}\ \_\ \_\ (\mathit{Empty}\ v_a) & = & \mathit{pSucceed}\ v_a \\
\mathit{p2p}\ \mathit{fsep}\ \mathit{sep}\ (\mathit{Choice}\ \mathit{chs}) & = & \mathit{foldr}\ (\mathbin{<\!\!|\!\!>})\ \mathit{pFail}\ (\mathit{map}\ \mathit{pars}\ \mathit{chs}) \\
\multicolumn{3}{l}{\quad \textbf{where}\ \mathit{pars}\ (\mathit{Br}\ t_{x\to a}\ p_x) = \mathit{flip}\ (\$) \mathbin{<\!\!\$} \mathit{fsep} \mathbin{<\!\!\!\circledast\!\!\!>} p_x \mathbin{<\!\!\!\circledast\!\!\!>} \mathit{p2p}\ \mathit{sep}\ \mathit{sep}\ t_{x\to a}}
\end{array}
$$

The *pPerms* function can now be implemented in terms of *pPermsSep*.

```
pPerms          :: Parser p ⇒ Perms p a → p a
pPerms          =  pPermsSep (pSucceed ())
```

To return to the small example, triples of an integer, a character, and a boolean—in any order—are parsed by:

```
pParens (pPermsSep (pSym ',') ((,,) <$> pInt <*> pChar <*> pBool))
```

4 Adding optional elements

This section shows how the permutation parsing mechanism can be extended such that it can deal with optional elements. Optional elements can be represented by parsers that can recognise the empty string and return a default value for this element. Calling the *pPermsSep* function on a permutation tree that contains optional elements leads to ambiguous parsers. Consider, for example, the tree in Figure 3 containing all permutations of a, b and c. Suppose b can be empty and we want to recognise ac. This can be done in three different ways since the empty b can be recognised before a, after a or after c. Fortunately, it is irrelevant for the result of a parse where exactly the empty b is derived, since order is not important. This allows us to use a strategy similar to the one proposed by Cameron [1]: parse nonempty constituents as they are seen and allow the parser to stop if all remaining elements are optional. When the parser stops the default values are returned for all optional elements that have not been recognised.

To implement this strategy we need to be able to determine whether a parser can derive the empty string and split it into its default value and its non-empty part, i.e. a parser that behaves the same except that it does not recognise the empty string. The splitting of parsers is represented by the *ParserSplit* class that is an extension of the normal *Parser* class. Most parser combinator libraries can be easily adapted to cover this extension.

```
class Parser p      ⇒ ParserSplit p where
  pEmpty            :: p a → Maybe a
  pNonempty         :: p a → Maybe (p a)
```

In the solution that does not deal with optional elements a parser for a permutation follows a path from the root of a permutation tree to a leaf, i.e. an *Empty* node. In the presence of optional elements, however, a parser may stop in any node that stores only optional elements. We adapt the *Perms* data type to incorporate this additional information. If all elements stored in a tree are optional then their default values are stored in *defaults*, otherwise *defaults* is *Nothing*. The parser stored in each *Branch* is not allowed to derive the empty string. Note that we do not need an *Empty* constructor anymore, since its semantics can be represented as a *Choice* node with an empty list of branches.

$$\textbf{data}\ \mathit{Perms}\ p\ a = \mathit{Choice}\ \{\mathit{defaults} :: \mathit{Maybe}\ a, \mathit{branches} :: [\mathit{Branch}\ p\ a]\}$$

The function *p2p* constructs a parser out of a permutation tree. If there are default values stored in *defaults* then the constructed parser can derive the empty string, returning those values.

$$\begin{array}{lcl}
\mathit{p2p} & :: & \mathit{Parser}\ p \\
 & \Rightarrow & p\ c \to p\ b \to \mathit{Perms}\ p\ a \to p\ a \\
\mathit{p2p}\ \mathit{fsep}\ \mathit{sep}\ t_{a\to b} & = & \mathit{foldr}\ (\lhd\rhd)\ \mathit{empty}\ \mathit{nonempties} \\
\quad \textbf{where}\ \mathit{empty} & = & \mathit{maybe}\ \mathit{pFail}\ \mathit{pSucceed}\ (\mathit{defaults}\ t_{a\to b}) \\
\qquad \mathit{nonempties} & = & \mathit{map}\ \mathit{pars}\ (\mathit{branches}\ t_{a\to b}) \\
\qquad \mathit{pars}\ (\mathit{Br}\ t_{x\to a}\ p_x) & = & \mathit{flip}\ (\$) \lhd\$\ \ \mathit{fsep} \lhd\!\!\rhd p_x \\
 & & \qquad \lhd\!\!\rhd\ \mathit{p2p}\ \mathit{sep}\ \mathit{sep}\ t_{x\to a}
\end{array}$$

A tuple, that represents a parser split into its empty part and its non-empty part, can describe four different kinds of parsers, as depicted in the following table:

empty part	non-empty part	
Nothing	*Nothing*	*pFail*
Just	*Nothing*	*pSucceed*
Nothing	*Just*	required element
Just	*Just*	optional element

The new definition of *add* reflects the four different cases. In the first case the resulting tree represents a failing permutation parser, i.e. it has no default values and no branches. In the second case the value stored in the empty part of the parser is pushed into the tree, only modifying the semantics of the tree but keeping its structure. In the cases where an element is added the non-empty part of the element is inserted in the tree in the same way as in the original definition of *add*. For an optional element the default value is combined with the *defaults* of the permutation tree.

$$\begin{array}{lcl}
\mathit{add} & :: & \mathit{Perms}\ p\ (a \to b) \\
 & & \to (\mathit{Maybe}\ a, \mathit{Maybe}\ (p\ a)) \\
 & & \to \mathit{Perms}\ p\ b \\
\mathit{add}\ t_{a\to b}@(\mathit{Choice}\ d_{a\to b}\ bs_{a\to b})\ mp_a = \textbf{case}\ mp_a\ \textbf{of} & & \\
\quad (\mathit{Nothing}, \mathit{Nothing}) & \to & \mathit{Choice}\ \mathit{Nothing}\ [\,] \\
\quad (\mathit{Just}\ v_a, \mathit{Nothing}) & \to & \mathit{Choice}\ (\mathit{fmap}\ (\$v_a)\ d_{a\to b})\ (\mathit{appSem}\ v_a) \\
\quad (\mathit{Nothing}, \mathit{Just}\ p_a) & \to & \mathit{Choice}\ \mathit{Nothing}\ (\mathit{insert}\ p_a) \\
\quad (\mathit{Just}\ v_a, \mathit{Just}\ p_a) & \to & \mathit{Choice}\ (\mathit{fmap}\ (\$v_a)\ d_{a\to b})\ (\mathit{insert}\ p_a) \\
\textbf{where}\ \mathit{insert}\ p_a & = & \mathit{Br}\ t_{a\to b}\ p_a : \mathit{map}\ \mathit{ins}\ bs_{a\to b} \\
\quad \mathit{ins}\ (\mathit{Br}\ t_{x\to a\to b}\ p_x) & = & \mathit{Br}\ (\mathit{add}\ (\mathit{mapPerms}\ \mathit{flip}\ t_{x\to a\to b})\ mp_a)\ p_x \\
\quad \mathit{appSem}\ v_a & = & \mathit{map}\ (\mathit{mapBranch}\ (\$v_a))\ bs_{a\to b}
\end{array}$$

The function *mapPerms* for the new *Perms* data type is defined as follows:

$$\begin{array}{lcl}
\mathit{mapPerms} & :: & (a \to b) \to \mathit{Perms}\ p\ a \to \mathit{Perms}\ p\ b \\
\mathit{mapPerms}\ v_{a\to b}\ t_a & = & \mathit{Choice}\ (\mathit{fmap}\ v_{a\to b}\ (\mathit{defaults}\ t_a)) \\
 & & \qquad (\mathit{map}\ (\mathit{mapBranch}\ v_{a\to b})\ (\mathit{branches}\ t_a))
\end{array}$$

Since we no longer have an *Empty* constructor the function *succeedPerms* is now defined as:

```
succeedPerms       :: a → Perms p a
succeedPerms x     =  Choice (Just x) []
```

Using the functions from the *ParserSplit* class we can straightforwardly define a new sequence operator for permutation parsers.

```
(<<$>>)            :: ParserSplit p
                   ⇒  Perms p (a → b) → p a → Perms p b
perms <<$>> p      =  add perms (pEmpty p, pNonempty p)
```

5 Applications

5.1 XML attributes

We will now demonstrate the use of the permutation parsers by showing how to parse XML tags with attributes. For simplicity, we just consider one tag (the `img` tag of XHTML) and only deal with a subset of the attributes allowed. In a Haskell program, this tag might be represented by the following data type.

```
data XHTML      =  Img {src       :: URI
                       , alt       :: Text
                       , longdesc  :: Maybe URI
                       , height    :: Maybe Length
                       , width     :: Maybe Length
                       }
                |  ...
```

Our variant of the `img` tag has five attributes of three different types. We use Haskell's record syntax to keep track of the names. The first two attributes are mandatory whereas the others are optional. We choose the *Maybe* variant of their types to reflect this optionality. Our parser should be able to parse the attributes in any order, where any of the optional arguments may be omitted. For the parsing process, we ignore whitespace and assume that there is a parser *pTok* that consumes just the given token and fails on any other input.

Using the *pPerms* combinator, writing the parser for the `img` tag is easy:

```
pImgTag     :: ParserSplit p ⇒ p XHTML
pImgTag     =  pTok "<"  *> pTok "img"  *> attrs <*  pTok "/>"
  where
  attrs     =  pPerms (Img <$> pField     "src"       pURI
                           <*> pField     "alt"       pText
                           <*> pOptField  "longdesc"  pURI
                           <*> pOptField  "height"    pLength
                           <*> pOptField  "width"     pLength
                     )
```

The order in which we denote the attributes determines the order in which the results are returned. Therefore, we can apply the *Img* constructor to form a value of the *XHTML* data type. The two helper functions *pField* and *pOptField* are used to parse a mandatory and an optional argument, respectively.

```
pField          :: Parser p ⇒ String → p a → p a
pField f p      =  pTok f  ⋗ pSym '='  ⋗ p

pOptField       :: Parser p ⇒ String → p a → p (Maybe a)
pOptField f p   =  Just <$> pField f p
                <|> pSucceed Nothing
```

5.2 Haskell's record syntax

Haskell allows data types to contain labelled fields. If one wants to construct a value of that data type, one can make use of these names. The advantage is that the user does not need to remember the order in which the fields of the constructor have been defined. Furthermore, all fields are considered as optional. If a field is not explicitly set to a value, it is silently assumed to be $\bot$.

Whereas compilers implement these features as a syntax for constructing values inside of Haskell programs, the *read* function that both GHC and Hugs generate with help of the **deriving** construct lacks this functionality. Although this behaviour is permitted by the Haskell Report, the resulting asymmetry is unfortunate.

We show here that the code that would do the job is easy to write or generate using the *pPermsSep* combinator.

```
pImg        :: ParserSplit p ⇒ p XHTML
pImg        =  pTok "Img"  ⋗ pTok "{"  ⋗ fields ⋖  pTok "}"
  where
  fields    =  pPermsSep (pSym ',')
                 (Img <$> pRecField "src"       pURI
                      <⋗> pRecField "alt"       pText
                      <⋗> pRecField "longdesc"  (pMaybe pURI)
                      <⋗> pRecField "height"    (pMaybe pLength)
                      <⋗> pRecField "width"     (pMaybe pLength)
                 )
```

The *pMaybe* combinator just parses the *Maybe* variant of a data type, and the *pRecField* makes each field optional.

```
pRecField       :: Parser p ⇒ String → p a → p a
pRecField f p   =  pField f p
                <|> pSucceed ⊥
```

6 Conclusion

We have presented a way to extend a parser combinator library with the functionality to parse free-order constructs. It can be placed on top of any combinator library that implements the *Parser* interface. A user of the library can easily write parsers for free-order constructs and does not need to care about checking and reordering the parsed elements. Due to the use of existentially quantified types the implementation of reordering is type safe and hidden from the user.

The underlying parser combinators can be used to handle errors, such as missing or duplicate elements, since the extension inherits their error-reporting or error-repairing properties. Figure 4 shows an example GHCi session that demonstrates error recovery with the UU_Parsing [8] library.

We have shown how our extension can be used to parse XML attributes and Haskell records. Other interesting examples mentioned by Cameron [1] include citation fields in BibTeX bibliographies and attribute specifiers in C declarations. Their pseudo-code algorithm uses a similar strategy. It does not show, however, how to maintain type safety by undoing the change in semantics resulting from reordering, nor can it deal with the presence of separators between free-order constituents.

```
UU_Parsing_Demo> let pOptSym x = pSym x <|> pSucceed '_'
UU_Parsing_Demo> let ptest = pPerms $ (,,) <$$> pList (pSym 'a')
                                           <||> pSym 'b'
                                           <||> pOptSym 'c'
                                 :: Parser Char (String, Char, Char)
UU_Parsing_Demo> t ptest "acb"
Result:
("a",'b','c')
UU_Parsing_Demo> t ptest ""
Symbol 'b' inserted at end of file; 'b' or 'c' or ('a')* expected.
Result:
("",'b','_')
UU_Parsing_Demo> t ptest "cdaa"
Errors:
Symbol 'd' before 'a' deleted; 'b' or ('a')* expected.
Symbol 'b' inserted at end of file; 'a' or 'b' expected.
Result:
("aa",'b','c')
UU_Parsing_Demo> t ptest "abd"
Errors:
Symbol 'd' at end of file deleted; 'c' or eof expected.
Result:
("a",'b','_')
```

Fig. 4. Example GHCi session (line breaks added for readability)

References

[1] Cameron, R. D., *Extending context-free grammars with permutation phrases*, ACM Letters on Programming Languages and Systems **2** (1993), pp. 85–94.
URL `http://www.acm.org/pubs/toc/Abstracts/1057-4514/176490.html`

[2] Fokker, J., *Functional parsers*, in: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*, LNCS **925** (1995), pp. 1–23.
URL `http://www.cs.uu.nl/~jeroen/article/parsers/parsers.ps`

[3] Hughes, J., *Generalising monads to arrows*, Science of Computer Programming **37** (2000), pp. 67–111.

[4] Hutton, G. and H. Meijer, *Monadic parser combinators*, Journal of Functional Programming **8** (1988), pp. 437–444.

[5] Leijen, D., *Parsec, a fast combinator parser* (2001).
URL `http://www.cs.uu.nl/~daan/parsec.html`

[6] Peyton-Jones, S. and J. Hughes, editors, "Report on the Programming Language Haskell 98," 1999.
URL `http://www.haskell.org/onlinereport`

[7] Swierstra, S. D., *Parser combinators: from toys to tools*, Haskell Workshop, 2000.
URL `http://www.cs.uu.nl/~doaitse/Papers/2000/HaskellWorkshop.pdf`

[8] Swierstra, S. D., *Fast, error repairing parser combinators* (2001).
URL `http://www.cs.uu.nl/groups/ST/Software/UU_Parsing`

[9] Swierstra, S. D. and L. Duponcheel, *Deterministic, error correcting combinator parsers*, in: *Advanced Functional Programming, Second International Summer School on Advanced Functional Programming Techniques*, LNCS **1129** (1996), pp. 184–207.
URL `http://www.cs.uu.nl/groups/ST/Software/UU_Parsing/AFP2.ps`

[10] Wadler, P., *How to replace failure with a list of successes*, in: *Functional Programming Languages and Computer Architecture*, LNCS **201** (1985), pp. 113–128.