*JFP* **14** (6): 741–757, 2004. © 2004 Cambridge University Press
DOI: 10.1017/S0956796804005192 Printed in the United Kingdom

# FUNCTIONAL PEARL
# *Parallel parsing processes*

KOEN CLAESSEN
*Department of Computing Science, Chalmers University of Technology, Gothenburg, Sweden*
(*e-mail:* koen@cs.chalmers.se)

**Abstract**

We derive a combinator library for non-deterministic parsers with a monadic interface, by means of successive refinements starting from a specification. The choice operator of the parser implements a breadth-first search rather than the more common depth-first search, and can be seen as a parallel composition between two parsing processes. The resulting library is simple and efficient for "almost deterministic" grammars, which are typical for programming languages and other computing science applications.

## 1 Introduction

A popular way to describe a parser in a functional language is to use a parser combinator library. In 1975, Burge (1975) had already proposed a set of functional parser combinators. Since the early 1990s, a large number of different parser combinator libraries has appeared for modern lazy functional languages (Fokker, 1995; Hutton, 1992), and their number seems to be steadily growing still.

So what contribution can this paper possibly make? To answer this question, we need to understand the different issues involved in designing and implementing parser combinator libraries today.

After Wadler's popularisation of *monads* for functional programming (Wadler, 1992), parser combinators were soon discovered to have a convenient monadic interface (Hutton & Meijer, 1998). By now, monads are well understood, there is syntactic support for them, and good library support that aids reuse of common monadic combinators. Monadic parsers are powerful enough to describe context-sensitive grammars, such as grammars where the structure of the grammar can depend on the input itself, e.g. the grammar of XML or Haskell.

However, the power that parser monads provide comes at a price. It has proven quite difficult to implement a parser monad in an efficient way. The efficiency of a parser combinator library usually revolves around a good implementation of the *choice* operator, which indicates a non-deterministic choice in the grammar. To implement the choice operator other than by a naive search, a careful analysis of the parsers involved seems to be needed. However, the use of the monadic *bind* combinator ($\gg=$), which sequentialises two parsers where the behaviour of the second parser depends on the result of the first, seems to make this impossible. For one cannot inspect the structure of the second parser before the first has produced a result.

Two different approaches have been proposed to address this problem. The first approach abandons the use of monads altogether and introduces a new class of combinators. Efficiency improvements from forbidding the use of the monadic bind, and instead introducing a weaker form of sequencing, were shown in Swierstra & Duponcheel (1996) for deterministic parsers, and later generalised for non-deterministic parsers in Swierstra (2000). This idea of a weaker form of sequencing was one of the motivations behind Hughes' *arrows* (Hughes, 2000). However, with the weaker sequencing, it is only possible to describe context-free grammars in these systems.

The second approach keeps the monads, but instead requires the user to specify at each choice point what kind of choice operator should be used (Hutton & Meijer, 1998; Leijen & Meijer, 2001). Usually, these libraries provide a number of different choice operators. For example, *asymmetric choice* means that the right hand side will not be taken if the left hand side succeeds, *deterministic choice* is only guaranteed to work if the choice can be decided by a one symbol look ahead. Most of these libraries still provide *general choice*, which has the efficiency problem mentioned earlier.

This paper presents a systematic derivation of a parser combinator library that (1) has a simple monadic interface, (2) does not need special choice annotations, and (3) is efficient in both time and space.

The derivation techniques we use are inspired by Hughes' (1995) pretty printing combinator derivation. The idea is to implement an abstract type by a datatype that sums up all the ways one can construct elements in the datatype, i.e. the operations that the library provides. This first implementation is called the *term representation*, and has trivial implementations for its operations. We successively refine this implementation by observing typical usage patterns of the constructors, giving them names, and then simplifying the datatype by using the new constructors. The implementations of the operations in terms of the new constructors can be derived using the laws that holds for the operations. The result is a very short and simple implementation of a parser monad.

The efficiency of the choice operator comes from the fact that we implement a breadth-first search (rather than a depth-first search), which works well with "almost deterministic" grammars. This informal term is usually used for grammars where choices can be decided locally or by not looking too far ahead, and where the expected number of results is small.

## 2 Specification of non-deterministic parsers

Here, we give a specification of a simple monadic interface to a non-deterministic parsing library. There is an abstract type $P\ s\ a$ of parsers that parse linear sequences of elements of type $s$ into possibly multiple structures of type $a$. The following primitive operations exist on these parsers:

$$\begin{aligned}
&\mathit{symbol} && :: P\ s\ s\\
&\mathit{fail} && :: P\ s\ a\\
&(+\!\!+\!\!+) && :: P\ s\ a \to P\ s\ a \to P\ s\ a
\end{aligned}$$

```
Form  ::=  ( Form )
        |  - Form
        |  Form & Form
        |  Var

Var   ::=  a | ... | z
```

```
data Form = Form :& Form | Not Form | Var Char

form, atom, paren, neg, var :: P Char Form
form  = do a ← atom; (conj a +++ return a)
atom  = paren +++ neg +++ var
paren = do this '('; a ← form; this ')'; return a
neg   = do this '−'; a ← atom; return (Not a)
var   = do v ← sat isAlpha; return (Var v)

conj   :: Form → P Char Form
conj a  = do this '&'; b ← form; return (a :& b)

sat  :: (s → Bool) → P s s
sat r = do c ← symbol; if r c then return c else fail

this  :: Eq s ⇒ s → P s s
this c = sat (c ==)
```

Fig. 1. A grammar and parser for a simple propositional logic.

The parser *symbol* consumes one symbol from the input (if there is one) and produces it as a result; if there is no symbol it fails. The parser *fail* does not consume any input, produces no results, and always fails. The choice operator (+++) takes two parsers and constructs one parser that produces the results of both. Further, the type $P\ s\ a$ has a so-called *monadic* interface as well:

```
return :: a → P s a
(≫=)   :: P s a → (a → P s b) → P s b
```

The parser *return* $x$ does not consume any input and produces $x$ as a result. The parser $p \gg\!= k$ (pronounced *bind*) first behaves like the parser $p$, but for every result $x$ produced by $p$, it then behaves like the parser $k\ x$. In this paper we sometimes use the *do-notation*, syntactic sugar that makes it easier to write expressions containing (≫=). The meaning of the do-notation is given by means of simple rewriting rules:

```
do x ← e; ⟨rest⟩   =   e ≫= λ x . do ⟨rest⟩
do e; ⟨rest⟩       =   e ≫= λ _ . do ⟨rest⟩
do e               =   e
```

An example use of the parser combinators provided here is given in Figure 1, where we implement a parser *form* for a simple propositional logic grammar. Note that in the parser implementation we are explicit about precedence and associativity of the operators; information which is not explicit in the grammar. We can see that we quickly find a need for defining auxiliary parser combinators, such as *sat* and *this*. The purpose of this paper is not to discuss those combinators, instead we refer to Hutton & Meijer (1998) and Leijen & Meijer (2001). Here, we restrict ourselves to developing a suitable implementation of the primitive combinators.

We can clarify what we mean by the informal explanations given above by defining a semantic mapping from the abstract type $P\ s\ a$ to a more concrete type *Parser* $s\ a$. In its definition, we use the type $\{\!|t|\!\}$ to mean the type of *bags* (also called *multisets* or *unordered lists*) of elements of type $t$. We use bags rather than lists because we want to change the order of results of parsers later on. On the expression level, empty bags are written $\{\!\!\{\}\!\!\}$, unit bags are written $\{\!\!\{x\}\!\!\}$, and we use *bag comprehension* notation, which is akin to list comprehensions.

$$\textbf{type}\ \mathit{Parser}\ s\ a\ =\ [s]\ \rightarrow\ \{\!\!\{(a,[s])\}\!\!\}$$

The type *Parser* $s$ $a$ represents the meaning of parsers: functions from strings of symbols of type $s$ to bags of results. A result is a pair of an answer of type $a$ plus the remaining part of the input.

The mapping $[\![\_]\!]$ tells us the meaning of each of the parser combinators:

$$\begin{array}{lll}
[\![\_]\!] & & ::\ P\ s\ a\ \rightarrow\ \mathit{Parsing}\ s\ a \\
[\![\mathit{symbol}]\!] & (c:s) & =\ \{\!\!\{(c,s)\}\!\!\} \\
[\![\mathit{symbol}]\!] & [\,] & =\ \{\!\!\{\}\!\!\} \\
[\![\mathit{fail}]\!] & s & =\ \{\!\!\{\}\!\!\} \\
[\![p +\!\!\!+\!\!\!+ q]\!] & s & =\ [\![p]\!]\ s\ \cup\ [\![q]\!]\ s \\
[\![\mathit{return}\ x]\!] & s & =\ \{\!\!\{(x,s)\}\!\!\} \\
[\![p \ggg k]\!] & s & =\ \{\!\!\{(y,s'')\mid (x,s')\in[\![p]\!]\ s,\ (y,s'')\in[\![k\ x]\!]\ s'\}\!\!\}
\end{array}$$

In the following we use this mapping to derive a number of implementations for the type $P$ $s$ $a$, leading to an efficient implementation of breadth-first parsing.

## 3 Traditional implementation: bags as lists

The traditional (and simplest) implementation of parser combinator libraries takes the type *Parser* $s$ $a$ as a direct implementation of $P$ $s$ $a$. The semantic mapping $[\![\_]\!]$ becomes the identity function, and the bags are implemented as simple lists.

$$\begin{array}{lll}
\mathit{symbol} & (c:s) & =\ [(c,s)] \\
\mathit{symbol} & [\,] & =\ [\,] \\
\mathit{fail} & s & =\ [\,] \\
(p +\!\!\!+\!\!\!+ q)\ s & & =\ p\ s\ +\!\!\!+\ q\ s \\
\mathit{return}\ x\ s & & =\ [(x,s)] \\
(p \ggg k)\ s & & =\ [(y,s'')\mid (x,s')\leftarrow p\ s,\ (y,s'')\leftarrow k\ x\ s']
\end{array}$$

This implementation is quite appealing since it is so close to the original definition of what the parser combinators mean. However, there are a number of inefficiency problems with the above definition.

*List concatenation* List concatenation $(+\!\!\!+)$ (used in the definition of $(+\!\!\!+\!\!\!+)$) costs linear time in the size of its left argument. So, if the $(+\!\!\!+\!\!\!+)$ combinator is nested left associatively, we have quadratic time behaviour in the depth of the nesting.

*List comprehensions* The list comprehension (used in the definition of $(\ggg)$) creates a lot of intermediate datastructures, which introduces a large constant overhead. (It is possible that an automatic compiler transformation could remove this overhead in some cases. Removing the list comprehension also allows us to perform other optimisations later.)

*Backtracking* The operational reading of the lazy lists constructed during parsing corresponds to backtracking. Backtracking works well for parsing with grammars that are highly non-deterministic. However, using backtracking for grammars that are "mostly" deterministic (i.e. non-deterministic choices can be resolved by looking but a few steps ahead in the input) leads to a nasty space-leak: at each choice point, we have to hold on to the complete input left at that point, because we might come back to that point in backtracking.

In the next couple of sections, we derive an alternative implementation that overcomes each of these problems. But first, we look at a number of laws that hold for the parsing combinators we use.

## 4 Laws for non-deterministic parsers

Here, we present a list of laws that follow from the semantics stated in section 2. First, without surprise, the well-known *monad laws* do in fact hold:

$$\begin{array}{lllcll}
L1. & \mathit{return}\ x & \gg\!\!= k & \equiv & k\ x & \\
L2. & p & \gg\!\!= \mathit{return} & \equiv & p & \\
L3. & (p \gg\!\!= k') & \gg\!\!= k & \equiv & p \gg\!\!= (\lambda x\,.\,k'\ x \gg\!\!= k) & \text{— } x \text{ not free in } k', k
\end{array}$$

A law stating $p \equiv q$ says that the parsers $p$ and $q$ have the same observable behaviour, i.e. that $[\![\, p \,]\!]\ s = [\![\, q \,]\!]\ s$ for all inputs $s$. Here is the proof of law $L1$:

$$\begin{array}{ll}
 & [\![\, \mathit{return}\ \ x \gg\!\!= k \,]\!]\ s \\
= & \{\!\!\{ (y, s'') \mid (x, s') \in [\![\, \mathit{return}\ \ x \,]\!]\ s,\ (y, s'') \in [\![\, k\ \ x \,]\!]\ s' \}\!\!\} \\
= & \{\!\!\{ (y, s'') \mid (y, s'') \in [\![\, k\ \ x \,]\!]\ s \}\!\!\} \\
= & [\![\, k\ \ x \,]\!]\ s
\end{array}$$

The other laws have similar proofs.

The above monad laws provide two laws ($L1$ and $L3$) that can be used to simplify parsers occurring on the left hand side of $(\gg\!\!=)$. Here are two more such laws; one for *fail* and one for $(+\!\!+\!\!+)$. These are actually two laws that hold for *monads with a plus*.

$$\begin{array}{lllcl}
L4. & \mathit{fail} & \gg\!\!= k & \equiv & \mathit{fail} \\
L5. & (p +\!\!+\!\!+ q) & \gg\!\!= k & \equiv & (p \gg\!\!= k) +\!\!+\!\!+ (q \gg\!\!= k)
\end{array}$$

Other laws for monads with a plus are the following, which say that choice ignores failing parsers.

$$\begin{array}{llllcl}
L6. & \mathit{fail} & +\!\!+\!\!+ & q & \equiv & q \\
L7. & p & +\!\!+\!\!+ & \mathit{fail} & \equiv & p
\end{array}$$

Moreover, the choice operator $(+\!\!+\!\!+)$ does not prefer any argument order, or order of nesting, and is therefore commutative and associative. Note that the commutativity property does not hold in general for monads with a plus, but it holds for $(+\!\!+\!\!+)$ since bags are unordered.

$$\begin{array}{llllcl}
L8. & (p +\!\!+\!\!+ q) & +\!\!+\!\!+ & r & \equiv & p +\!\!+\!\!+ (q +\!\!+\!\!+ r) \\
L9. & p & +\!\!+\!\!+ & q & \equiv & q +\!\!+\!\!+ p
\end{array}$$

These laws follow directly from the commutativity and associativity of the union operator $\uplus$ for bags.

Finally, there is one law about the *symbol* operator, which is at the heart of the algorithm we derive later. It says that a choice between two parsers that both start by consuming a symbol, also starts with consuming a symbol, and proceeds with a choice between the two remaining parts of the parsers.

$$L10. \quad (symbol \ggg k) \mathbin{+\!\!\!+\!\!\!+} (symbol \ggg k') \equiv symbol \ggg (\lambda c \,.\, k\ c \mathbin{+\!\!\!+\!\!\!+} k'\ c)$$

The case for non-empty strings of this law can be proved as follows:

$$\begin{aligned} & [\![(symbol \ggg k) \mathbin{+\!\!\!+\!\!\!+} (symbol \ggg k')]\!]\,(c : s) \\ = {} & [\![symbol \ggg k]\!]\,(c : s) \uplus [\![symbol \ggg k']\!]\,(c : s) \\ = {} & [\![k\ c]\!]\,s \uplus [\![k'\ c]\!]\,s \\ = {} & [\![k\ c \mathbin{+\!\!\!+\!\!\!+} k'\ c]\!]\,s \\ = {} & [\![symbol \ggg (\lambda c \,.\, k\ c \mathbin{+\!\!\!+\!\!\!+} k'\ c)]\!]\,(c : s) \end{aligned}$$

## 5 Implementation A: term representation

To find an efficient implementation of the parsing library specification, we use an approach pioneered by Hughes (1995). The idea is to derive an implementation of an abstract type by first representing it as a datatype that sums up all the ways one can construct elements in the datatype, i.e. the operations that the library provides. This first implementation is called the *term representation*, and has trivial implementations for its operations.

We consecutively refine this implementation by observing typical usage patterns of the constructors, giving them names, and then simplifying the datatype by using the new constructors instead of the old ones. The implementations of the operations in terms of the new constructors can be derived using the laws that hold for the operations. We start with the following term representation for $P\ s\ a$:

```
data P s a = Symbol                  — wrong!
           | Fail
           | P s a :+++ P s a
           | ∀ b . P s b :>>= (b → P s a)
           | Return a
```

The constructor functions *Fail*, (:$\mathbin{+\!\!\!+\!\!\!+}$), and *Return* directly correspond to the operators *fail*, ($\mathbin{+\!\!\!+\!\!\!+}$), and *return*. The constructor (:$\ggg$) also directly corresponds to the operator ($\ggg$), but since ($\ggg$) is polymorphic not only in its final result type, but also in its intermediate result type, we need to use local type quantification in the declaration of (:$\ggg$).

However, the type of the *Symbol* constructor shows that something is wrong; *Symbol* has type $P\ s\ a$, but $symbol :: P\ s\ s$. The type of *Symbol* really does not make much sense as a representation of the function *symbol*, since the result of *symbol* should be a symbol, not just something of any type. To fix this, we introduce a different operation to our parsing interface, but we do this just for the sake of this particular implementation. This new operation, called *symbolMap*, can be used to implement *symbol*, and it actually is a variant of *symbol* that takes an extra argument – a function that is to be applied to the parsed symbol before returning it.

$$symbolMap :: (s \rightarrow a) \rightarrow P\ s\ a$$

(Note that the type of *symbolMap* fits nicely as a constructor of the type *P s a*.) Its semantics is:

$$\begin{aligned} [\![ symbolMap\ h ]\!]\ (c : s) &= \{\!\!\{ (h\ c, s) \}\!\!\} \\ [\![ symbolMap\ h ]\!]\ [\,] &= \{\!\!\{\,\}\!\!\} \end{aligned}$$

Of course, *symbol* can trivially be defined in terms of *symbolMap* using the following law:

$$D1. \quad symbol \quad \equiv \quad symbolMap\ id$$

So, the final version of our term representation of the type *P s a* becomes:

$$\begin{aligned} \mathbf{data}\ P\ s\ a = {} & SymbolMap\ (s \rightarrow a) \\ | \ & Fail \\ | \ & P\ s\ a :\!\!+\!\!\!+\!\!\!+ P\ s\ a \\ | \ & \forall\, b\,.\, P\ s\ b :\ggg (b \rightarrow P\ s\ a) \\ | \ & Return\ a \end{aligned}$$

The definitions of the operators in our parsing interface in terms of the above constructors are straightforward:

$$\begin{aligned} symbol &= SymbolMap\ id \\ fail &= Fail \\ (+\!\!\!+\!\!\!+) &= (:\!+\!\!\!+\!\!\!+) \\ (\ggg) &= (:\ggg) \\ return &= Return \end{aligned}$$

Lastly, we can use the definition of the semantic mapping $[\![\_]\!]$ in order to give a function *parse* that takes a parser and a string of input symbols, and produces the results of the parser. However, we leave the implementation of the bag type $\{\!\!\{\_\}\!\!\}$ still abstract for now.

$$\begin{array}{lll} parse & & :: P\ s\ a \rightarrow Parsing\ s\ a \\ parse\ (SymbolMap\ h)\ (c : s) & & = \{\!\!\{ (h\ c, s) \}\!\!\} \\ parse\ (SymbolMap\ h)\ [\,] & & = \{\!\!\{\,\}\!\!\} \\ parse\ Fail & \_ & = \{\!\!\{\,\}\!\!\} \\ parse\ (p :\!+\!\!\!+\!\!\!+ q) & s & = parse\ p\ s \cup parse\ q\ s \\ parse\ (Return\ x) & s & = \{\!\!\{ (x, s) \}\!\!\} \\ parse\ (p :\ggg k) & s & = \{\!\!\{ (y, s'') \mid (x, s') \in parse\ p\ s, \\ & & \qquad\qquad (y, s'') \in parse\ (k\ x)\ s' \}\!\!\} \end{array}$$

It is not easy to come up with a good way of implementing the bags used in the above function, because of the use of bag union ($\uplus$) and the bag comprehensions.

Instead, we will consecutively refine the implementation of the datatype $P\ s\ a$ in order to remove the constructor functions that give rise to the use of the bag operators ((:+++) and (:≫), respectively).

## 6 Implementation B: removing bind

Our goal in the next implementation is to remove the possibly expensive bag comprehension in the last clause of the definition of *parse*. The approach we take here is to take away the constructor function (:≫) from our implementation altogether. This we do by trying to simplify away uses of (≫) as much as possible, and then give names to the cases that cannot be removed. These names we use to introduce new constructor functions instead of the ones we could simplify away.

There exist laws for simplifying all parsing operators on the left hand side of a (≫) operator, except *symbol*. We therefore merge the constructor (:≫) with the constructor *SymbolMap* into a new constructor, called *SymbolBind*, and then implement the two operators (≫) and *symbol* in terms of the new constructor. The law for the new constructor is:

$$D2. \quad \mathit{SymbolBind}\ k \quad \equiv \quad \mathit{symbol} \gg k$$

Note that we are abusing notation a little bit here; really we should have used a function *symbolBind* in the above law, but since we actually never implement such a function explicitly, we use the constructor function in its place. The new datatype becomes:

$$\begin{aligned}
\textbf{data}\ P\ s\ a = {} & \mathit{SymbolBind}\ (s \rightarrow P\ s\ a) \\
\mid\ & \mathit{Fail} \\
\mid\ & P\ s\ a\ {:}{+\!\!+\!\!+}\ P\ s\ a \\
\mid\ & \mathit{Return}\ a
\end{aligned}$$

So how do we implement our parsing operators? The three untouched operators are defined as before: $\mathit{fail} = \mathit{Fail}$, $({+\!\!+\!\!+}) = ({:}{+\!\!+\!\!+})$, and $\mathit{return} = \mathit{Return}$. The implementation of *symbol* follows directly from law $L2$ and definition $D2$:

$$\mathit{symbol} \ = \ \mathit{SymbolBind}\ \mathit{Return}$$

We can also derive the implementation of the (≫) operator, resulting in:

$$\begin{array}{lll}
\mathit{SymbolBind}\ f \gg k & = & \mathit{SymbolBind}\ (\lambda c\ .\ f\ c \gg k) \\
\mathit{Fail} \gg k & = & \mathit{Fail} \\
(p\ {:}{+\!\!+\!\!+}\ q) \gg k & = & (p \gg k)\ {:}{+\!\!+\!\!+}\ (q \gg k) \\
\mathit{Return}\ x \gg k & = & k\ x
\end{array}$$

To illustrate this derivation, here is the first clause in the definition of (≫):

$$\begin{array}{lll}
 & \mathit{SymbolBind}\ f \gg k & \\
= & (\mathit{symbol} \gg f) \gg k & \text{— by } D2 \\
= & \mathit{symbol} \gg (\lambda c\ .\ f\ c \gg k) & \text{— by } L3 \\
= & \mathit{SymbolBind}\ (\lambda c\ .\ f\ c \gg k) & \text{— by } D2
\end{array}$$

The other clauses can be derived in a similar fashion. Lastly, the function *parse* has one fewer case to deal with:

$$\begin{array}{llll}
\mathit{parse}\ (\mathit{SymbolBind}\ f)\ (c : s) & & = & \mathit{parse}\ (f\ c)\ s \\
\mathit{parse}\ (\mathit{SymbolBind}\ f)\ [\,] & & = & \{\!\!\}\\
\mathit{parse}\ \mathit{Fail} & \_ & = & \{\!\!\}\\
\mathit{parse}\ (p :\!+\!\!\!+\!\!\!+\ q) & s & = & \mathit{parse}\ p\ s \cup \mathit{parse}\ q\ s \\
\mathit{parse}\ (\mathit{Return}\ x) & s & = & \{\!\{(x,s)\}\!\}
\end{array}$$

The first two clauses of this definition follow directly from definition $D2$ and the semantics for ($\ggg$).

This implementation does not need to implement bag comprehensions, but the bag union operator ($\uplus$) is still there. Even though it is possible to implement bag union in an efficient way, using it here is a problem, for it is vital to be able to control the order in which the elements in the resulting bag are evaluated. For example, in the above, evaluating *parse p s* first means that *parse q s* still holds on to the whole input *s*, leading to a space leak. So, we want more fine-grained control over the evaluation order in the case of choice. In the next refinement, we would therefore like to remove the constructor function ($:\!+\!\!\!+\!\!\!+$) which gives rise to the use of ($\uplus$).

## 7 Implementation C: removing plus

Similar to the previous definition, we make the observation that there exist laws for simplifying any parser on the left and right hand side of a ($+\!\!\!+\!\!\!+$) operator, except for *return*. So, we merge ($:\!+\!\!\!+\!\!\!+$) with *Return* into a new constructor *ReturnPlus* and implement the two operators ($+\!\!\!+\!\!\!+$) and *return* in terms of the new constructor. The law for the new constructor is:

$$D3. \quad \mathit{ReturnPlus}\ x\ p \quad \equiv \quad \mathit{return}\ x +\!\!\!+\!\!\!+ p$$

The new datatype loses yet another two constructors and gains a new one:

$$\begin{array}{l}
\mathbf{data}\ P\ s\ a = \mathit{SymbolBind}\ (s \rightarrow P\ s\ a)\\
\qquad\qquad\ \mid \mathit{Fail}\\
\qquad\qquad\ \mid \mathit{ReturnPlus}\ a\ (P\ s\ a)
\end{array}$$

There is only one untouched operator defined as before: $\mathit{fail} = \mathit{Fail}$. The implementation of the *return* operator can easily be derived from law $L7$ and definition $D3$:

$$\mathit{return}\ x = \mathit{ReturnPlus}\ x\ \mathit{Fail}$$

The *symbol* operator is implemented as before, only it uses the new definition of *return*:

$$\mathit{symbol} = \mathit{SymbolBind}\ \mathit{return}$$

The choice operator ($+\!\!\!+\!\!\!+$) has to be defined by means of pattern matching on the other constructors. The complete definition of the $(+\!\!+\!\!+)$ operator becomes:

$$\begin{array}{lll}
\mathit{SymbolBind}\ f & +\!\!+\!\!+\ \mathit{SymbolBind}\ g & = \mathit{SymbolBind}\ (\lambda c\ .f\ c\ +\!\!+\!\!+\ g\ c) \\
\mathit{Fail} & +\!\!+\!\!+\ q & = q \\
p & +\!\!+\!\!+\ \mathit{Fail} & = p \\
\mathit{ReturnPlus}\ x\ p & +\!\!+\!\!+\ q & = \mathit{ReturnPlus}\ x\ (p +\!\!+\!\!+ q) \\
p & +\!\!+\!\!+\ \mathit{ReturnPlus}\ x\ q & = \mathit{ReturnPlus}\ x\ (p +\!\!+\!\!+ q)
\end{array}$$

The first clause is a very powerful case, which, by using law $L10$, allows us to combine two parsers of the form *SymbolBind f*. The second and third clauses are direct consequences of laws $L6$, and $L7$, respectively. The last two clauses can be derived using law $L8$, which is associativity of $(+\!\!+\!\!+)$, and definition $D3$. The last clause even makes use of law $L9$, which is commutativity of $(+\!\!+\!\!+)$!

The new definition of $(+\!\!+\!\!+)$ thus changes the order of results compared to a traditional implementation using lists and backtracking. The order of arguments is changed in such a way that all possible intermediate results are produced and all possible intermediate failing alternatives are discarded before the next symbol is consumed. In other words, the different choice alternatives are executed in *lock-step*, which means that none of the choice alternatives hold on to the input, as a traditional backtracking implementation would, or in fact any implementation which does not change the order of results in the choice operator. Together with the first clause in the definition of $(+\!\!+\!\!+)$, which merges two uses of *symbol* into one, this leads to a *breadth-first* (rather than the traditional *depth-first*) implementation of parsing.

The implementation of the $(\gg\!\!=)$ operator has to deal with the new constructor *ReturnPlus*. Here is a derivation of the corresponding clause:

$$\begin{array}{lll}
 & \mathit{ReturnPlus}\ x\ p \gg\!\!= k & \\
= & (\mathit{return}\ x\ +\!\!+\!\!+\ p) \gg\!\!= k & \text{— by } D3 \\
= & (\mathit{return}\ x \gg\!\!= k)\ +\!\!+\!\!+\ (p \gg\!\!= k) & \text{— by } L5 \\
= & k\ x\ +\!\!+\!\!+\ (p \gg\!\!= k) & \text{— by } L1
\end{array}$$

The complete definition of $(\gg\!\!=)$ looks as follows:

$$\begin{array}{lll}
\mathit{SymbolBind}\ f & \gg\!\!= k & = \mathit{SymbolBind}\ (\lambda c\ .f\ c \gg\!\!= k) \\
\mathit{Fail} & \gg\!\!= k & = \mathit{Fail} \\
\mathit{ReturnPlus}\ x\ p & \gg\!\!= k & = k\ x\ +\!\!+\!\!+\ (p \gg\!\!= k)
\end{array}$$

And lastly, the function *parse* has again one case fewer to deal with:

$$\begin{array}{lll}
\mathit{parse}\ (\mathit{SymbolBind}\ f) & (c : s) & = \mathit{parse}\ (f\ c)\ s \\
\mathit{parse}\ (\mathit{SymbolBind}\ f) & [\,] & = \{\!\![\,]\!\!\} \\
\mathit{parse}\ \mathit{Fail} & \_ & = \{\!\![\,]\!\!\} \\
\mathit{parse}\ (\mathit{ReturnPlus}\ x\ p)\ s & & = \{\!\![(x,s)]\!\!\} \uplus \mathit{parse}\ p\ s
\end{array}$$

The last clause follows directly from definition $D3$ and the semantics for *return* and $(+\!\!+\!\!+)$.

We have managed to express the *parse* function without using bag comprehensions, and with using bag union only in a simple case. Thus, the decision of how to implement the used bags is not difficult anymore; we use plain lists. Here is the implementation of the *parse* function above using lists:

```
parse (SymbolBind f)  (c : s) = parse (f c) s
parse (SymbolBind f)  []      = []
parse Fail            _       = []
parse (ReturnPlus x p) s      = (x, s) : parse p s
```

As we can see, to construct the lists, we are only using (:) and [], and it takes constant time to produce a result.

If we were going to use lists for bags anyway, why did we bother with doing the development with bags? The answer is that using bags in our original specification of parser semantics allowed us to change the order of the results in the list. This is what allowed us to construct a breadth-first implementation. One alternative possibility would be to use a set instead of a bag in the specification, enabling us to use idempotence as well. However, this would also restrict the possible parser result types to equality-types.

The implementation we have arrived at now is not the final one yet; there is still one source of inefficiency left that we want to remove.

## 8 Implementation D: associativity of bind

Let us take a look at the implementation of (≫=) in section 7. It is defined using recursion over its left argument. Just as for example with list concatenation, using (≫=) left-associatively in a nested fashion leads to behaviour taking quadratic time in terms of the nesting depth. This typically happens in recursive grammars for tree-like structures. Therefore, we would like to force (≫=) to be used only right-associatively.

To do this, it is not enough to simply refine the type $P\ s\ a$ further. Instead, a parser should be made aware of the way it is *used*, its *context*. Providing the context of a parser to its implementation makes the way the bind operator is used left-associatively explicit. Thus, the next implementation of the parser type becomes a function from some type representing its context to a real parser. This technique is called the *context passing implementation* in Hughes (1985).

As a note on the side, the problem of avoiding left-associative applications of (≫=) is not a problem specific to this paper. In fact, it is well known that using a so-called *continuation passing monad* solves this problem. In the form of a *monad transformer* (Liang *et al.*, 1995) one can even use an out-of-the-box solution. However, we still think it is instructive and interesting to derive a solution for our parser monad using the context passing implementation. The solution we arrive at later turns out to be exactly the continuation passing monad.

Before we look at exactly how to implement the type $P\ s\ a$ using context passing, we introduce some preliminaries. We reuse the implementation of the type $P\ s\ a$ from the previous section (section 7) as a basis for our new implementation. To avoid name confusion, we use primed (′) versions of the names to refer to the implementations in that section:

```
symbol' :: P' s s
fail'   :: P' s a
```

$$\begin{array}{ll}
(+\!\!+\!\!+') & :: P'\ s\ a \rightarrow P'\ s\ a \rightarrow P'\ s\ a \\
\mathit{return}' & :: a \rightarrow P'\ s\ a \\
(\ggg') & :: P'\ s\ a \rightarrow (a \rightarrow P'\ s\ b) \rightarrow P'\ s\ b \\
\mathit{parse}' & :: P'\ s\ a \rightarrow \mathit{Parser}\ s\ a
\end{array}$$

We use the unprimed versions of the names for the implementation of the current section. The idea is that in the new implementation, the function $(\ggg')$ is only going to be applied right-associatively.

Next, we have to decide what the context we are going to pass around looks like. For simplicity, let us assume that the parser is always used in a problematic context, i.e. where it is the left argument of an application of $(\ggg')$. So, contexts have the following shape: "$\bullet \ggg' k$" (where $\bullet$ represents a hole in which parsers can be plugged in), and can simply be represented by $k$ itself. In the case where a parser is used in a context that does not actually have this shape, we simply take $k = \mathit{return}'$, in which case we have the identity context, by law $L2$.

The type of $k$ depends on the result type of the parser that is put in the hole in the context, and also on the result type of the whole context. These types are not necessarily the same; when we construct a parser, we have no idea what the final result type of its context will be. Therefore, we introduce two different result types, $a$ and $b$, and, inside $P$, universally quantify over the result type of the context $b$.

$$\begin{array}{ll}
\textbf{type}\ \mathit{Context}\ s\ a\ b & = a \rightarrow P'\ s\ b \\
\textbf{type}\ P\ s\ a & = \forall b\ .\ \mathit{Context}\ s\ a\ b \rightarrow P'\ s\ b
\end{array}$$

What is the law that allows us to derive the implementations of the corresponding operations? For a parser $p$ in the new type and its corresponding parser $p'$ in the old type, the following correspondence should hold:

$$D4. \quad p \quad \equiv \quad \lambda k\ .\ p' \ggg' k$$

Furthermore, we want the actual results of the two parsers to be the same:

$$D5. \quad \mathit{parse}\ p \quad \equiv \quad \mathit{parse}'\ p'$$

We can now derive the definitions for the parsing operations. Here are the three primitive parsing operations:

$$\begin{array}{ll}
\mathit{symbol} & = \lambda k\ .\ \mathit{SymbolBind}\ k \\
\mathit{fail} & = \lambda k\ .\ \mathit{Fail} \\
p +\!\!+\!\!+ q & = \lambda k\ .\ p\ k +\!\!+\!\!+' q\ k
\end{array}$$

The derivations of *symbol* and *fail* are quite straightforward. Here is the derivation of $(+\!\!+\!\!+)$:

$$\begin{array}{lll}
& p +\!\!+\!\!+ q & \\
= & \lambda k\ .\ (p' +\!\!+\!\!+' q') \ggg' k & \text{— by } D4 \\
= & \lambda k\ .\ (p' \ggg' k) +\!\!+\!\!+' (q' \ggg' k) & \text{— by } L5 \\
= & \lambda k\ .\ p\ k +\!\!+\!\!+' q\ k & \text{— by } D4
\end{array}$$

The definitions of the monadic operators look like this:

$$\begin{aligned} \mathit{return}\ x &= \lambda k\ .\ k\ x \\ p \ggg f &= \lambda k\ .\ p\ (\lambda x\ .\ f\ x\ k) \end{aligned}$$

It is interesting to note that these definitions do not depend on *return′* and $(\ggg')$ at all! Let us look at the derivation of $(\ggg)$:

$$\begin{aligned} & p \ggg f \\ = \ & \lambda k\ .\ (p' \ggg' f') \ggg' k && \text{— by } D4 \\ = \ & \lambda k\ .\ p' \ggg' (\lambda x\ .\ f'\ x \ggg' k) && \text{— by } L3 \\ = \ & \lambda k\ .\ p\ (\lambda x\ .\ f\ x\ k) && \text{— by } D4 \end{aligned}$$

Note how we use the associativity law of $(\ggg)$ ($L3$) in order to ensure right-associativity — this was our original goal! Lastly, we implement the *parse* function:

$$\mathit{parse}\ p \ = \ \mathit{parse}'\ (p\ \mathit{return}')$$

Its derivation is equally simple:

$$\begin{aligned} & \mathit{parse}\ p \\ = \ & \mathit{parse}'\ p' && \text{— by } D5 \\ = \ & \mathit{parse}'\ (p' \ggg' \mathit{return}') && \text{— by } L2 \\ = \ & \mathit{parse}'\ (p\ \mathit{return}') && \text{— by } D4 \end{aligned}$$

We have now arrived at the final implementation. Note that this implementation does not make use of $(\ggg')$ anymore. Figure 2 shows the complete implementation, where we have simplified the definition of *parse′* to have fewer cases.

## 9 Extensions

Several extensions to the functionality of the derived parser combinators can be made. In this section, we discuss two such extensions.

*Look-ahead* A parsing technique that is often used is *look-ahead*, i.e. looking at the input without consuming it in order to decide what to do next. An example of an application of look-ahead is a parser for identifiers as non-empty sequences of alpha-numeric characters. Parsing the input "foo" using a direct implementation would produce the results $\{\!\!\{$("f","oo"), ("fo","o"), ("foo","")$\}\!\!\}$. However, the intention is probably to only get $\{\!\!\{$("foo","")$\}\!\!\}$, which we can obtain by looking ahead in the input and fail if there are still characters left that are alpha-numeric.

Thus, we introduce a new parsing operator in our library, $\mathit{look} :: P\ s\ [s]$, with the following definition:

$$[\![\,\mathit{look}\,]\!]\ s \ = \ \{\!\!\{(s,s)\}\!\!\}$$

It is not possible to implement *look* transparently in terms of the other combinators. This means that we have to adapt our current implementation, and simply following

```
  — types
type P s a  = ∀b . (a → P′ s b) → P′ s b
data P′ s a = SymbolBind (s → P′ s a)
            | Fail
            | ReturnPlus a (P′ s a)

  — main functions
symbol    = λk . SymbolBind k
fail      = λk . Fail
p +++ q   = λk . p k +++′ q k
return x  = λk . k x
p ≫= f    = λk . p (λx . f x k)
parse p   = parse′ (p (λx . ReturnPlus x Fail))

  — auxiliary functions
SymbolBind f   +++′ SymbolBind g    = SymbolBind (λc . f c +++′ g c)
Fail           +++′ q               = q
p              +++′ Fail            = p
ReturnPlus x p +++′ q               = ReturnPlus x (p +++′ q)
p              +++′ ReturnPlus x q  = ReturnPlus x (p +++′ q)

parse′ (SymbolBind f)   (c : s) = parse′ (f c) s
parse′ (ReturnPlus x p) s       = (x, s) : parse′ p s
parse′ _                _       = []
```

Fig. 2. Final parser implementation.

the development we have gone through for the other parser operators, we end up with an extra constructor in the $P'$ datatype:

```
data P′ s a = … | LookBind ([s] → P′ s a)
```

The function *look* is simply implemented as $\lambda\, k\, .\, \mathit{LookBind}\ k$. We also have to adapt the auxiliary functions to be able to deal with the new constructor *LookBind*. Thus, we add the following clauses to ($+\!\!+\!\!+'$):

```
LookBind f +++′ LookBind g = LookBind (λs . f s +++′ g s)
LookBind f +++′ q          = LookBind (λs . f s +++′ q)
p          +++′ LookBind g = LookBind (λs . p +++′ g s)
```

The first clause is an optimisation; it avoids creating expressions of the form $\mathit{LookBind}\ (\lambda\, s_1\, .\, \mathit{LookBind}\ (\lambda\, s_2\, .\, \ldots))$, which are unnecessary, since $s_1$ and $s_2$ will be bound to the same value anyway. Lastly, we add the following clause to *parse*′:

```
parse′ (LookBind f) s = parse′ (f s) s
```

Here is an example of how *look* can be used. The function *munch* takes a predicate $r$ over symbols, and parses as many symbols as possible satisfying $r$.

```
munch :: (s → Bool) → P s [s]
munch r = do s ← look; inspect s
  where
    inspect (c : s) | r c = do symbol; s′ ← inspect; return (c : s′)
    inspect _             = return []
```

We grab the current input with *look* and pass it to the local function *inspect*, which inspects the input, and builds a parser that precisely consumes the symbols that satisfy $r$. Note that when we decide to consume a symbol, we already know which symbol it is (namely $c$), so we can ignore the result of *symbol*. The identifier parser we introduced earlier can now be expressed using the parser *munch isAlphaNum*.

Other useful parser operations that can be implemented using *look* are *longest* $:: P\ s\ a\ \rightarrow\ P\ s\ a$, which only delivers the longest parse(s) of its argument, and *try* $:: P\ s\ a\ \rightarrow\ P\ s\ (Maybe\ a)$, which never fails but returns *Nothing* exactly once when its argument fails.

*Alternative parse functions* One design freedom which we have not explored yet is varying the kind of result that the *parse* function delivers. The assumption that the user of our parsers is interested in all intermediate results plus the left-over part of the input might not always be true, and the user might pay a performance price for it. Several alternative parse functions are possible though, and the choice of which parse function to use should be made by the user independently for each parser. These alternative parse functions can be seen as different interpreters for the same parse language.

Here is an example. When a parse error occurs, instead of simply returning [], we would like to return the position in the input where the error occurred. Let us assume that we have a type for positions *Pos*, an initial position $pos_0\ ::\ Pos$, and a function $next\ ::\ Pos\ \rightarrow\ Symbol\ \rightarrow\ Pos$, that computes the next position given a current position and a symbol. We can implement an alternative parse function by modifying the auxiliary function *parse'*. For simplicity, we only deliver the first result that manages to parse the complete input.

```
parse' :: P' s a → [s] → Either Pos a
parse' p s  =  track p s pos₀
  where
    track  (SymbolBind f)  (c : s) pos  =  track (f c) s $! next pos c
    track  (ReturnPlus x _) []     pos  =  Right x
    track  (ReturnPlus _ p) s      pos  =  track p s pos
    track  (LookBind f)     s      pos  =  track (f s) s pos
    track  _                _      pos  =  Left pos
```

We use strict function application ($!) to force evaluation of the position during parsing, because otherwise lazy evaluation builds a large position expression in the heap which consumes a lot of memory. This parse function is the first step towards adding a good error reporting mechanism.

## 10 Discussion

Through a series of successive refinements, we have created a simple and efficient implementation of a parser monad.

We have made two serious implementations of parser combinator libraries based on the ideas presented in the paper. In the first implementation, we have added error reporting combinators, and provided an interface that is compatible with Leijen's parser combinator library *Parsec* (2001), for which there exist a lot of useful auxiliary parser combinators and a comprehensive user manual. The second implementation has no error reporting, but one extra feature which allows converting back and forth between Haskell's type *ReadS a* and a parser of type *P Char a*. This library is used internally in the Glasgow Haskell Compiler to invisibly replace uses of Haskell's *read* function by calls to a more efficient parser.

Ljunglöf (2002) gives an excellent overview and comparison of different existing parser combinator implementations. One of his results is that our implementation performs best on deterministic grammars among the monadic parser combinators providing general choice. It is slightly less efficient than other non-monadic parser combinators, in particular on grammar definitions that are not left-factorised. Interesting is the relation between our implementation and Swierstra's implementation of non-monadic parser combinators (2000); both use a datatype isomorphic to $P'\ s\ a$ internally.

One other interesting observation we noted was that the datatype $P'\ s\ a$ is isomorphic to the type *SP a b* of stream processors used in the Fudgets framework (Carlsson & Hallgren, 1998). In fact, the $(+\!\!+')$ operator is one of the parallel composition operators provided for stream processors! This inspired the view of the parser combinators being parsing process combinators. The choice operator can then be seen as a parallel composition of parsing processes.

In the paper, we have so far only shown *partial correctness*: if our functions produce a result at all, it is going to be the correct result. In order to show total correctness, we also need to argue why our functions will produce actual results. We cannot simply prove termination, since for any interesting grammar, the parsers we create are infinite, because we use recursion in the definition of the parsers. Therefore, we should argue that *parse* is always *productive/destructive*: For each consumption of a constructor in the parser datatype, either one symbol from the input is consumed, a result is generated on the output list, or the output list is terminated. Doing this formally is beyond the scope of this paper.

## Acknowledgements

We are grateful to Magnus Carlsson, Ralf Hinze, Graham Hutton, Simon Peyton Jones, and the anonymous referees for useful suggestions on this work.

## References

Burge, W. (1975) *Recursive Programming Techniques.* Addison Wesley.

Carlsson, M. and Hallgren, T. (1998) *Fudgets – Purely Functional Processes with applications to Graphical User Interfaces.* PhD thesis, Chalmers University of Technology, `http://www.cs.chalmers.se/~hallgren/Thesis/`.

Fokker, J. (1995) Functional parsers. In: Jeuring, J. and Meijer, E. (editors), *Advanced Functional Programming*, vol. 925, pp. 1–23. Springer-Verlag.

Hughes, J. (1995) The design of a pretty-printing library. In: Jeuring, J. and Meijer, E. (editors), *Advanced Functional Programming*, vol. 925. Springer-Verlag.

Hughes, J. (2000) Generalising monads to arrows. *Sci. Comput. Program.* **37**, 67–111.

Hutton, G. (1992) Higher-order functions for parsing. *J. Funct. Program.* **2**(3), 323–343.

Hutton, G. and Meijer, E. (1998) Monadic parsing in Haskell. *J. Funct. Program.* **8**(4), 437–444.

Leijen, D. and Meijer, E. (2001) *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Technical report UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht. `http://www.cs.uu.nl/~daan/parsec.html`.

Liang, S., Hudak, P. and Jones, M. (1995) Monad transformers and modular interpreters. *Symposium on Principles of Programming Languages (POPL'95)*, pp. 333–343.

Ljunglöf, P. (2002) *Pure Functional Parsing – an advanced tutorial*. Lic thesis, Chalmers University of Technology, `http://www.cs.chalmers.se/~peb/papper.html`.

Swierstra, S. D. (2000) Parser combinators, from toys to tools. In: Hutton, G. (editor), *Proceedings ACM SIGPLAN Haskell Workshop*. Electronic Notes in Theoretical Computer Science 41.1. Elsevier.

Swierstra, S. and Duponcheel, L. (1996) Deterministic, error-correcting combinator parsers. In: Launchbury, J., Meijer, E. and Sheard, T. (editors), *Advanced Functional Programming*.

Wadler, P. (1992) Monads for functional programming. In: Broy, M. (editor), *Marktoberdorf Summer School on Program Design Calculi*. NATO ASI Series F: Computer and systems sciences 118. Springer-Verlag.