FUNCTIONAL PEARLS

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. The choice operator is implemented as 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 parser combinator library is a collection of combinators which can be used to describe parsers. Since the early 90's (Fokker, 1995; Hutton, 1992), a huge amount of different parser combinator libraries has suddenly appeared for modern lazy functional languages, 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.

With Wadler's popularisation of *monads* for functional programming (Wadler, 1992), parser combinators were quickly 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 which aids reuse of common monadic combinators. Monads are certainly powerful enough to be able to describe any context-sensitive grammar, meaning that the structure of the grammar can depend on parts of the input. One example of a context-sensitive grammar is XML, where open tags have to correspond to matching close tags. Another example is a programming language with fixity declarations for operations.

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 merges two parsing alternatives. 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 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 classes of solutions have been proposed to solve this problem.

The first solution abandons the use of monads altogether and introduces a new class of combinators. Huge 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 solution keeps the monads, but requires instead the user to specify more information about what kind of choice operator should be used when (Hutton & Meijer, 1998; Leijen, 2000). 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' pretty printing combinator derivation (Hughes, 1995). 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 consecutively refine this implementation by observing typical usage patterns of the constructors, giving them names, and then simplifying the datatype by using the new contructors. 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 implemented a depth-first search (rather than a breadth-first search), which works well with "almost deterministic" grammars. This informal term is usually used for grammars where choices can be decided by not looking too far ahead, and where the expected number of results is small.

```
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 parser for a simple propositional logic datatype

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 strings of elements of type s into structures of type a. The following primitive operations exist on these parsers:

```
symbol :: P s s
fail   :: P s a
(+++)  :: P s a → P s a → P s a
```

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 combines their results. 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 ≫= 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 will 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⟩
```

$$\textbf{do}\ e \quad = \quad e$$

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 datatype *Form*. 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; Leijen, 2000). Here, we will restrict ourselves to developing a suitable implementation of the combinators introduced.

We can define what we mean by the informal explanations given above more formally by defining a semantics mapping from the abstract type $P\ s\ a$ to a more concrete type $Parsing\ s\ a$. 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. On the expression level, empty bags are written $\{\!|\,|\!\}$, unit bags are written $\{\!|\, x \,|\!\}$, and we will use *bag comprehension* notation, which is akin to list comprehensions.

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

The type $Parsing\ s\ a$ 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:

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

In the following we will use this semantics mapping to derive a number of implementations 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 $Parsing\ s\ a$ as a direct implementation of $P\ s\ a$. The semantics mapping $[\![\_]\!]$ becomes the identity function, and the bags are implemented as simple lists.

$$\begin{array}{lll}
symbol & (c:s) & = [(c,s)] \\
symbol & [\,] & = [\,] \\
fail & s & = [\,] \\
(p +\!\!\!+\!\!\!+ q)\ s & & = p\ s +\!\!\!+ q\ s \\
return\ x\ s & & = [(x,s)]
\end{array}$$

$$(p \ggg k)\ s \quad = [(y, s'') \mid (x, s') \leftarrow p\ s,\ (y, s'') \leftarrow k\ x\ s'']$$

We can see that this implementation is very 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 can a quadratic time behavior 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.

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 which 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 will derive an alternative implementation that overcomes each of these problems. But first, we will 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 should hold for any implementation of the specified library that respects the given semantics. Firstly, without surprise, the well-known *monad laws* do in fact hold:

$$\begin{array}{lllll}
L1. & \mathit{return}\ x & \ggg & k & \equiv & k\ x \\
L2. & p & \ggg & \mathit{return} & \equiv & p \\
L3. & (p \ggg k') & \ggg & k & \equiv & p \ggg (\lambda x\ .\ k'\ x \ggg k)
\end{array}$$

To prove a law stating that $p \equiv q$, we simply show that $[\![p]\!]\ s = [\![q]\!]\ s$ for all s. Here is the proof of law $L1$:

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

The other laws have similar proofs.

The above monad laws provide two laws ($L1$ and $L3$) which 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 of the laws 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 $\cup$ for bags.

Lastly, there is one law about the *symbol* operator which is at the heart of the algorithm we will use 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.

$$L10. \quad (\mathit{symbol} \gg\!\!- k) \;+\!\!\!+\!\!\!+\; (\mathit{symbol} \gg\!\!- k') \;\equiv\; \mathit{symbol} \gg\!\!- (\lambda c\, .\, k\ c +\!\!\!+\!\!\!+ k'\ c)$$

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

$$\begin{array}{rl} & [\![\, (\mathit{symbol} \gg\!\!- k) +\!\!\!+\!\!\!+ (\mathit{symbol} \gg\!\!- k') \,]\!]\, (c : s) \\ = & [\![\, \mathit{symbol} \gg\!\!- k \,]\!]\, (c : s) \cup [\![\, \mathit{symbol} \gg\!\!- k' \,]\!]\, (c : s) \\ = & [\![\, k\ c \,]\!]\, s \cup [\![\, k'\ c \,]\!]\, s \\ = & [\![\, k\ c +\!\!\!+\!\!\!+ k'\ c \,]\!]\, s \\ = & [\![\, \mathit{symbol} \gg\!\!- (\lambda c\, .\, k\ c +\!\!\!+\!\!\!+ k'\ c) \,]\!]\, (c : s) \end{array}$$

5 Implementation A: Term Representation

To find an efficient implementation of the parsing library specification, we will use an approach pioneered by John Hughes in his lecture notes on Pretty Printing (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 contructors 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 $P\ s\ a$:

$$\begin{array}{lll} \textbf{data}\ P\ s\ a = & \mathit{Symbol} & \text{— wrong!} \\ & \mid \mathit{Fail} & \\ & \mid P\ s\ a :\!+\!\!\!+\ P\ s\ a & \\ & \mid \forall\, b\,.\, P\ s\ b :\!\gg\!\!- (b \to P\ s\ a) & \\ & \mid \mathit{Return}\ a & \end{array}$$

Some explanation is in place here: The constructor functions *Fail*, $(:\!+\!\!\!+)$, and *Return* directly correspond to the operators *fail*, $(+\!\!\!+)$, and *return*. The constructor $(:\!\gg\!\!-)$ also directly corresponds to the operator $(\gg\!\!-)$, but since $(\gg\!\!-)$ 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 $(:\!\gg\!\!-)$.

However, the type of the *Symbol* constructor implies that something is wrong. We can see that $\mathit{Symbol} :: P\ s\ a$, but $\mathit{symbol} :: P\ s\ s$. This difference is quite important! 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 a is variant of *symbol* which takes an extra argument – a function that is to be applied to the parsed symbol before returning it.

$$\mathit{symbolMap} :: (s \to a) \to P\ s\ a$$

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

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

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

$$D1. \quad \mathit{symbol} \ \equiv\ \mathit{symbolMap}\ \mathit{id}$$

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

$$\begin{array}{ll} \textbf{data}\ P\ s\ a = & \mathit{SymbolMap}\ (s \to a) \\ & \mid \mathit{Fail} \\ & \mid P\ s\ a :\!+\!\!\!+\ P\ s\ a \end{array}$$

$$\begin{array}{l} \mid\ \forall\, b\,.\, P\ s\ b :\!\!\succ\!\!\!\succ (b\ \rightarrow\ P\ s\ a) \\ \mid\ \mathit{Return}\ a \end{array}$$

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

$$\begin{array}{lcl} \mathit{symbol} & = & \mathit{SymbolMap}\ \mathit{id} \\ \mathit{fail} & = & \mathit{Fail} \\ (+\!\!\!+\!\!\!+) & = & (:\!+\!\!\!+\!\!\!+) \\ (\succ\!\!\!\succ) & = & (:\!\succ\!\!\!\succ) \\ \mathit{return} & = & \mathit{Return} \end{array}$$

Lastly, we can use the definition of the semantics 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.

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

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 ($\cup$) 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 $(:\!\succ\!\!\!\succ)$, 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 way we remove that clause here is by removing the constructor function $(:\!\succ\!\!\!\succ)$ from our implementation alltogether. This we do by trying to simplify away uses of $(\succ\!\!\!\succ)$ as much as possible, and then give names to the cases that cannot be removed. These names we will use to introduce new constructor functions instead of the ones we could simplify away.

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

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

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 will use the constructor function in its place. The new datatype becomes:

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

So how do we implement our parsing operators? The three untouched operators are defined as before:

$$\begin{aligned}
\mathit{fail} &= \mathit{Fail} \\
(+\!\!\!+\!\!\!+) &= (:\!+\!\!\!+\!\!\!+) \\
\mathit{return} &= \mathit{Return}
\end{aligned}$$

The implementation of the operator *symbol* follows directly from law $L2$ and definition $D2$:

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

We can also derive the implementation of the $(\ggg)$ operator, resulting in:

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

To illustrate this derivation, the first clause in the definition of $(\ggg)$ goes as follows:

$$\begin{aligned}
& \mathit{SymbolBind}\ f \ggg k \\
= {} & (\mathit{symbol} \ggg f) \ggg k && \text{— by } D2 \\
= {} & \mathit{symbol} \ggg (\lambda c\,.\, f\ c \ggg k) && \text{— by } L3 \\
= {} & \mathit{SymbolBind}\ (\lambda c\,.\, f\ c \ggg k) && \text{— by } D2
\end{aligned}$$

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

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

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

This implementation does not need to implement bag comprehensions, but the bag union operator $(\cup)$ is still there. In the next refinement, we would like to remove the constructor function $(:\!+\!\!\!+\!\!\!+)$ which gives rise to the use of $(\cup)$.

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:

$$\begin{aligned}\textbf{data}\ P\ s\ a = {} & \mathit{SymbolBind}\ (s \to P\ s\ a) \\ \mid {} & \mathit{Fail} \\ \mid {} & \mathit{ReturnPlus}\ a\ (P\ s\ a)\end{aligned}$$

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 intermediate results are returned before the next symbol is consumed. Together with the first clause, 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 $(\ggg)$ operator has to deal with the new constructor *ReturnPlus*. Here is a derivation of the corresponding clause:

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

The complete definition of $(\ggg)$ looks as follows:

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

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

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

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

The *parse* function is now rather efficient, since we have removed bag comprehensions and reduced the use of bag union to a simple case. However, there is still one source of inefficiency left, which we will discuss in Section 9.

8 Implementing Bags as Lists, Again

As we remarked, the *parse* function from the previous section does not contain any complicated operations on bags anymore. Therefore, we can now decide how to implement the used bags. The decision is not difficult; we will use plain lists.

Here is the implementation of the *parse* function above using lists:

$$\begin{array}{lll}
\mathit{parse}\ (\mathit{SymbolBind}\ f) & (c : s) & = \mathit{parse}\ (f\ c)\ s \\
\mathit{parse}\ (\mathit{SymbolBind}\ f) & [\,] & = [\,]
\end{array}$$

$$\begin{array}{lcl}
\mathit{parse}\ \mathit{Fail}\ \_ & = & [\,] \\
\mathit{parse}\ (\mathit{ReturnPlus}\ x\ p)\ s & = & (x, s)\ :\ \mathit{parse}\ p\ s
\end{array}$$

As we can see, to construct the lists, we are only using (:) and [], and it takes but 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.

So far, we have only reasoned about *partial correctness*: If our functions produce a result at all, it is going to be the correct result. To be fully correct, we also need to argue why our functions will produce actual results. It is easy to see that ($+\!\!+\!\!+$) and ($\ggg$) behave well, since they are applied to smaller arguments in their recursive calls. However, for any interesting grammar, the parsers we create are going to be infinite anyway, since we are allowed use recursion in the definition of the parsers.

Therefore, the *parse* function might not terminate at all! For it is perfectly possible for some infinite input to lead the parser into a loop where it will never come out of, simply because there are no results and it will take an infinite amount of time to find that out.

Therefore, we will (informally) argue that *parse* is *productive*. It is productive in the sense that for each consumption of a constructor in the parser datatype, a result is "produced": either one symbol from the input is consumed, or a result is generated on the output list, or the output list is terminated.

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

9 Implementation D: Associativity of Bind

Let us take a look at the implementation of ($\ggg$) in Section 7. It is defined using recursion over its left argument. Just as for example with list concatenation, using ($\ggg$) 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 ($\ggg$) to be used only right-associatively.

To solve this, we can not use the technique of refining the type $P\ s\ a$ anymore. Instead, we have to make the way a parser is *used*, its *context*, apparent to the implementation of the parser type. Passing the context can (in our case) make it explicit in what way the bind operator is used left-associatively, so that its implementation can do something about that. 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' notes (Hughes, 1995).

Before we look at exactly how to implement the type $P\ s\ a$ with this new context, we introduce some preliminaries. We will 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 will use primed (′) versions of the names to refer to the implementations in that section:

$$\begin{array}{ll}
\textbf{type}\ P'\ s\ a & \\
\mathit{symbol}' & ::\ P'\ s\ s \\
\mathit{fail}' & ::\ P'\ s\ a \\
(+\!\!+')& ::\ P'\ s\ a\ \rightarrow\ P'\ s\ a\ \rightarrow\ P'\ s\ a \\
\mathit{return}' & ::\ a\ \rightarrow\ P'\ s\ a \\
(\gg\!\!=') & ::\ P'\ s\ a\ \rightarrow\ (a\ \rightarrow\ P'\ s\ b)\ \rightarrow\ P'\ s\ b \\
\mathit{parse}' & ::\ P'\ s\ a\ \rightarrow\ \mathit{Parsing}\ s\ a
\end{array}$$

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

Next, we have to decide what the context we are going to pass around will look 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 $(\gg\!\!=')$. So, contexts have the following shape: "$\bullet \gg\!\!=' k$" (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.

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

What is the law that will allow 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:

$$D4. \quad p\ \equiv\ \lambda k\ .\ p' \gg\!\!=' k$$

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

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

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

$$\mathit{symbol}\ =\ \lambda k\ .\ \mathit{SymbolBind}\ k$$

$$\begin{aligned} \mathit{fail} &= \lambda k\,.\,\mathit{Fail} \\ p +\!\!\!+\!\!\!+\; q &= \lambda k\,.\,p\;k +\!\!\!+\!\!\!+' q\;k \end{aligned}$$

Here is the derivation of the function *symbol*:

$$\begin{array}{lll} & \mathit{symbol} & \\ = & \lambda k\,.\,\mathit{symbol}' \ggg' k & \text{— by } D4 \\ = & \lambda k\,.\,\mathit{SymbolBind}\ \mathit{return}' \ggg' k & \text{— def. } \mathit{symbol}' \\ = & \lambda k\,.\,\mathit{SymbolBind}\ (\lambda c\,.\,\mathit{return}'\ c \ggg' k) & \text{— def. } (\ggg') \\ = & \lambda k\,.\,\mathit{SymbolBind}\ k & \text{— by } L1 \end{array}$$

The derivation of *fail* goes in a similar way. 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 their derivations. Here is *return*:

$$\begin{array}{lll} & \mathit{return}\ x & \\ = & \lambda k\,.\,\mathit{return}'\ x \ggg' k & \text{— by } D4 \\ = & \lambda k\,.\,k\ x & \text{— by } L1 \end{array}$$

And here is ($\ggg$):

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

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:

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

```
  — 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 and complete parser implementation

We have now arrived at the final implementation. Note that the final implementation does not make use of (≫=) anymore. Figure 2 shows the complete implementation in a compact way.

10 Look Ahead

The five parser combinators we have dealt with so far are enough to describe parsers for any computable grammar. However, there are some parsers which are quite cumbersome to define using only these combinators.

As an example, let us imagine specifying a parser for *identifiers* in a simple language. Let us day that an identifier is a sequence of alpha-numeric characters, where the sequence can be of any strictly positive length. This definition immediately leads to problems! For example, when parsing the string "name", we get the results ⦃("n","ame"), ("na","me"), ("nam","e"), ("name","")⦄. However, our intention was probably to only get ⦃("name","")⦄.

A parsing technique that could have solved this problem is to perform *look ahead* – we can look at the input without consuming it in order to decide what to do next. In the case of the identifier parser, we can use look ahead to decide that we will fail if there are still characters left which are alpha-numeric. Thus, we introduce a new parsing operator in our library:

$$look :: P\ s\ [s]$$

The intention of this operator is to return the part of the input that has not been consumed by the parser yet, without consuming any of it itself:

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

It is not possible to implement *look* transparently in terms of the other combinators. This means that we have to adapt our current implementation. Simply following the development we have gone through for the other parser operators, this turns out to be quite straightforward. Not surprisingly, we end up with an extra constructor in the P' datatype:

$$\mathbf{data}\ P'\ s\ a = \ldots \mid LookBind\ ([s] \rightarrow P'\ s\ a)$$

The implementation of the function *look* comes out as follows:

$$look = \lambda\, k\ .\ LookBind\ k$$

We also have to adapt the auxiliary functions to be able to deal with the new constructor *LookBind*. First, we add the following clauses to $(+\!\!+\!\!+')$:

$$\begin{aligned}
LookBind\ f +\!\!+\!\!+' LookBind\ g &= LookBind\ (\lambda s\ .\ f\ s +\!\!+\!\!+' g\ s)\\
LookBind\ f +\!\!+\!\!+' q &= LookBind\ (\lambda s\ .\ f\ s +\!\!+\!\!+' q)\\
p +\!\!+\!\!+' LookBind\ g &= LookBind\ (\lambda s\ .\ p +\!\!+\!\!+' g\ s)
\end{aligned}$$

These all directly follow from laws that can be derived from the semantics definition of *look*. The first clause is an optimisation: it is not necessary but improves performance, because it avoids creating expressions of the form $LookBind\ (\lambda\ s_1\ .\ LookBind\ (\lambda\ s_2\ .\ \ldots))$, 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'*:

$$parse'\ (LookBind\ f)\ s = parse'\ (f\ s)\ s$$

The complete input s is copied, which can lead to space leaks if f holds on to s while parsing continues to consume s. So, we have to be careful when using *look*. Here is an example of how *look* can be used in a correct way. The function *munch* takes a predicate over symbols r, and parses as many symbols as possible from the input that satisfy r.

```
munch :: (s → Bool) → P s [s]
munch r = do s ← look; sim s
  where
    sim (c : s) | r c = do get; s' ← sim; return (c : s')
    sim _             = return []
```

The implementation works as follows. We grab the current input with *look* and pass it to the local function *sim*, 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 *get*. The identifier parser we introduced earlier can now simply be expressed as *munch isAlphaNum*.

Other powerful parser operations which can be implemented using *look* are

```
eof     :: P s Bool                 — checks for end of input
longest :: P s a → P s a            — only delivers longest results
try     :: P s a → P s (Maybe a)    — delivers Nothing when failing
```

We leave their implementations as an exercise to the reader.

11 Other Parse Functions

One other design freedom which we have not explored yet is varying the kind of result that the *parse* function delivers. So far, we have made the assumption that the user of our parsers is interested in all intermediate results plus the left over part of the input. This is not always the case however, and the user might pay a performance price for that. In this section, we show three alternative *parse* functions. These can be added to the library without making a change to the existing definitions.

Some of these functions can actually be implemented in terms of the current *parse′* function, but the point here is that the performance price for that might be too high, and that there is a simple alternative which does not require that price.

Complete results Suppose we are only interested in the results that managed to parse the complete input. In that case, we can adapt the *parse′* function as follows:

```
parse' :: P' s a → [s] → [a]
parse' (SymbolBind f)  (c : s)  = parse' (f c) s
parse' (ReturnPlus x p) []      = x : parse' p []
parse' (ReturnPlus _ p) s       = parse' p s
parse' (LookBind f)     s       = parse' (f s) s
parse' _                _       = []
```

The difference with the old *parse′* is in the clauses dealing with *ReturnPlus*, since we only deliver a result when the input is [ ].

Longest result Sometimes, the input cannot be parsed completely at all. In that case, we might be interested in the result that managed to parse the input furthest:

```
parse' :: P' s a → [s] → Maybe (a, [s])
```

```
parse' p s = longest p s Nothing
  where
    longest (SymbolBind f)   (c : s) mres = longest (f c) s mres
    longest (ReturnPlus x p) s       _    = longest p s (Just (x, s))
    longest (LookBind f)     s       mres = longest (f s) s mres
    longest _                _       mres = mres
```

In a similar way, we can produce a list of *all* longest results.

Keeping the position A parsing function which cannot be defined using the simple function *parse'*, is a parse function which, when a parse error occurs, also produces the position in the input where the error occurred. Let us assume we have a type for positions *Pos*, an initial position pos_0, and a function *next*, which computes the next position given a current position and a symbol:

```
type Pos
pos0 :: Pos
next :: Pos → Symbol → Pos
```

Let us assume for simplicity that we are only interested in one result which has parsed the file completely, or, alternatively, a position of where the longest parse failed if there are no complete results.

```
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 will build a large position expression in the heap which consumes a lot of memory.

The above parse function is the first step towards adding a good error reporting mechanism.

12 Discussion

We have derived a simple and efficient implementation of a parser monad. It is not possible to show a detailed benchmark comparison here, but we refer to (Ljunglöf, 2002), which gives an excellent overview of different parser combinator implementations.

The resulting implementation turns out to be quite easily extendible, as we showed in Section 10 when we added the *look* operator. We have also extended the parser monad in the same way when adding a function that can convert a function in Haskell's type *ReadS a* into a parser of type *P Chara*. This was done as part of an extension that the Glasgow Haskell Compiler implements that invisibly replaces uses of Haskell's *read* function by calls to a parser in our parser monad and vice versa.

One interesting thing we noted was that the datatype $P'\ s\ a$ is completely 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.

It is well-known that we can force a monadic computation to be built right-associatively with respect to $(\gg\!=)$ by using a *continuation monad transformer* (Liang *et al.*, 1995). In fact, a continuation monad transformer was used originally, but for the sake of presentation we decided to explain this using the context passing implementation instead.

References

Carlsson, Magnus, & Hallgren, Thomas. (1998). *Fudgets – purely functional processes with applications to graphical user interfaces.* Ph.D. thesis, Chalmers University of Technology.

Fokker, Jeroen. (1995). Functional parsers. *Pages 1–23 of:* Jeuring, J., & Meijer, E. (eds), *Advanced functional programming*, vol. 925. Springer Verlag.

Hughes, John. (1995). The design of a pretty-printing library. Jeuring, J., & Meijer, E. (eds), *Advanced functional programming*, vol. 925. Springer Verlag.

Hughes, John. (2000). Generalising monads to arrows. *Science of computer programming*, **37**, 67–111.

Hutton, Graham. (1992). Higher-order functions for parsing. *Journal of functional programming*, **2**(3), 323–343.

Hutton, Graham, & Meijer, Erik. (1998). Monadic parsing in haskell. *Journal of functional programming*, **8**(4), 437–444.

Leijen, Daan. (2000). *Parsec, a fast combinator parser.* http://www.cs.uu.nl/˜daan/parsec.html.

Liang, S., Hudak, P., & Jones, M. (1995). Monad transformers and modular interpreters. *Pages 333–343 of: Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

Ljunglöf, Peter. (2002). *Pure functional parsing – an advanced tutorial.* Lic. thesis, Chalmers University of Technology.

Swierstra, S. Doaitse. (2000). Parser combinators, from toys to tools. Hutton, Graham (ed), *Proceedings of the acm sigplan haskell workshop.*

Swierstra, S.D., & Duponcheel, L. (1996). Deterministic, error-correcting combinator parsers. Launchbury, John, Meijer, Erik, & Sheard, Tim (eds), *Advanced functional programming.*

Wadler, Phil. (1992). Monads for functional programming. Broy, M. (ed), *Marktoberdorf summer school on program design calculi.* NATO ASI Series F: Computer and systems sciences, vol. 118. Springer Verlag.