Total Parser Combinators
Draft

Nils Anders Danielsson

December 10, 2009

Abstract

A monadic parser combinator library which guarantees termination of parsing, while still allowing many forms of left recursion, is described. The library's interface is similar to that of many other parser combinator libraries, with two important differences: one is that the interface clearly specifies which parts of the constructed parsers may be infinite, and which parts have to be finite, using a combination of induction and coinduction; and the other is that the parser type is unusually informative.

The library comes with a formal semantics, using which it is proved that the parser combinators are as expressive as possible. The implementation is supported by a machine-checked correctness proof.

1 Introduction

Parser combinators (Burge 1975; Wadler 1985; Fairbairn 1987; Hutton 1992; Meijer 1992; Fokker 1995; Röjemo 1995; Swierstra and Duponcheel 1996; Koopman and Plasmeijer 1999; Leijen and Meijer 2001; Ljunglöf 2002; Hughes and Swierstra 2003; Claessen 2004; Frost et al. 2008; Wallace 2008, and many others) can provide an elegant and declarative method for specifying parsers. When compared with typical parser generators they have some advantages: it is easy to abstract over recurring grammatical patterns, and there is no need to use a separate tool just to parse something. On the other hand there are also some disadvantages: there is a risk of lack of efficiency, and parser generators can give static guarantees about termination and non-ambiguity which most parser combinator libraries fail to give. This paper addresses one of these points by defining a parser combinator library which ensures statically that parsing will terminate.

When using parser combinators the parsers/grammars are often constructed using cyclic definitions, so it is natural to see the definitions as being partly corecursive. However, a purely coinductive reading of the choice and sequencing combinators would allow definitions like $p = p \mid p$ and $p' = p' \cdot p'$, for which it is impossible to implement parsing in a total way (in a pure setting where p and p' can only be inspected via their infinite unfoldings). As shown in Section 3 totality can be ensured by reading choice inductively, and only reading an argument of the sequencing operator coinductively when the other argument does not accept the empty string. (Induction and coinduction is discussed in Section 2.)

The parser combinator library which is described below is defined in the dependently typed functional programming language Agda (Norell 2007; Agda Team 2009), which will be introduced as we go along. The library comes with a formal semantics (Section 3.2) and a machine-checked proof[1] which shows that the implementation is correct with respect to the semantics. The code which the paper is based on is at the time of writing available from the author's web page.

The main contributions of the paper are as follows:

- It is shown how parser combinators can be implemented in such a way that termination is guaranteed.
- Unlike many other parser combinator libraries these parser combinators can handle many forms of left recursion.
- It is shown that the parser combinators are as expressive as possible.
- The parser combinators come with a formal semantics, and are shown to satisfy a number of properties.

The core of the paper is Sections 3 and 4. The former section introduces the ideas by using recognisers (parsers which do not return any result other than "the string matched" or "the string did not match"), and the latter section generalises to full parser combinators.

1.1 Related work

There does not seem to be much prior work on *formally* verified termination for parser combinators (or other general parsing frameworks). McBride and McKinna (2002) define grammars inductively, and use types to ensure that a token is consumed before a non-terminal can be encountered, thereby ruling out left recursion and non-termination. Danielsson and Norell (2008) use similar ideas. Muad`Dib (2009) uses a monad annotated with Hoare-style pre- and post-conditions (Swierstra 2009) to define total parser combinators, including a fixpoint combinator whose type rules out left recursion by requiring the input to be shorter in recursive calls. Note that none of these other approaches can handle left recursion. The library defined in this paper seems to be the first one which both handles (many forms of) left recursion and guarantees termination for every parser which is accepted by the host language.[2] It also seems fair to say that, when compared to the other approaches above, this library has an interface which is closer to those of "classical" parser combinator libraries. In the classical approach the ordinary general recursion of the host language is used to implement cyclic grammars; this library uses "ordinary" corecursion (restricted by types, see Sect. 3).

There are a number of parser combinator libraries which can handle various forms of left recursion, but they all seem to come with some form of restriction.

[1]Note that the meta-theory of Agda has not been properly formalised, and Agda's type checker has not been proved to be bug-free, so take phrases such as "machine-checked proof" with a grain of salt.

[2]Danielsson and Norell (2009) define a parser using a specialised version of the library described in this paper. This version of the library can handle neither left nor right recursion, and is restricted to parsers which do not accept the empty string. A brief description of the parser interface is provided, but the implementation of the backend is not discussed.

The parser combinators defined here can handle many left recursive grammars, but not all; for instance, the definition $p = p$ is rejected statically. Lickman (1995) defines a library which can handle left recursion if a tailor-made fixpoint combinator, based on an idea due to Philip Wadler, is used. He proves (informally) that parsers defined using his combinators are terminating, as long as they are used in the right way; the argument to the fixpoint combinator must satisfy a non-trivial semantic criterion, which is not checked statically. Johnson (1995) and Frost et al. (2008) define libraries of recogniser and parser combinators, respectively, including memoisation combinators which can be used to handle left recursion. As presented these libraries can fail to terminate if used with grammars with an infinite number of non-terminals (as a simple example, consider $p\ n\ =\ \mathit{memoise}\ n\ (p\ (1 + n))$), and users of the libraries need to ensure manually that the combinators are used in the right way. The same limitations apply to a parser combinator library described by Ljunglöf (2002). This library uses an impure feature, *observable sharing* (Claessen and Sands 1999), to detect cycles in the grammar. Claessen (2001) mentions a similar implementation, attributing the idea to Magnus Carlsson.

In Section 4 it is shown that the parser combinators are as expressive as possible—every language which can be decided by the host language can also be decided using the combinators. In the case of finite token sets this holds even for non-monadic parser combinators using the applicative functor interface (McBride and Paterson 2008); see Section 3.5. The fact that monadic parser combinators can be as expressive as possible has already been pointed out by Ljunglöf (2002), who also mentions that applicative combinators can be used to parse some languages which are not context-free, because one can construct infinite grammars by using parametrised parsers. It has also been known for a long time that an infinite grammar can represent any language, decidable or not (Solomon 1977), and that the languages generated by many infinite grammars can be decided (Mazurkiewicz 1969). However, the result that monadic and applicative combinators have the same expressive strength for finite token sets seems to be largely unknown. For instance, Claessen (2004, page 742) claims that "with the weaker sequencing, it is only possible to describe context-free grammars in these systems".

Bonsangue et al. (2009, Example 2) represent a kind of regular expressions in a way which bears some similarity to the representation of recognisers in Section 3. Unlike the definition in this paper their definition is inductive, with an explicit representation of cycles: $\mu x.\varepsilon$, where ε can contain x. However, occurrences of x in ε have to be guarded by what amounts to the consumption of a token, just as in this paper.

In Section 3.3 a Brzozowski derivative operator (Brzozowski 1964) is implemented for recognisers, and in Section 3.4 this operator is used to characterise recogniser equality coinductively. Rutten (1998) performs similar tasks for regular expressions.

2 Induction and coinduction

The parser combinators defined in Section 3 use a combination of induction and coinduction which may at first sight seem bewildering, so let us begin by discussing induction and coinduction. (This discussion is rather informal. For more theoretical accounts of induction and coinduction see, for instance, the works of Hagino (1987) and Mendler (1988).)

Induction can be used to define types where the elements have finite "depth". A simple example is the type of finite lists. In Agda this data type can be defined by giving the types of all the constructors:

```
data List (A : Set) : Set where
  []  :                   List A
  _::_ : A → List A → List A
```

(The constructor _::_ is an infix operator; _ marks the argument positions. *Set* is a type of small types.) This definition should be read inductively, i.e. all lists have finite length.

Coinduction, on the other hand, can be used to define types where some elements have infinite depth. Consider the type of infinite streams, for instance:

```
data Stream (A : Set) : Set where
  _::_ : A → ∞ (Stream A) → Stream A
```

(Note that constructors can be overloaded.) The type function ∞ : *Set* → *Set* marks an argument as being coinductive. In this case this means that all streams have infinite length.

The function ∞ is analogous to the suspension type constructor which is used to implement non-strictness in strict languages (Wadler et al. 1998). Just as the suspension type constructor the function ∞ comes with delay and force operators, here called $\sharp$_ and $\flat$:

```
♯_ : {A : Set} →     A → ∞ A
♭  : {A : Set} → ∞ A →     A
```

($\sharp$_ is a tightly binding prefix operator; ordinary function application binds tighter, though. Note that $\{A : \mathit{Set}\} \to T$ is a *dependent* function space; the argument A is in scope in T. Arguments in braces, $\{\ldots\}$, are implicit, and do not need to be given explicitly as long as Agda can infer them from the context.)

Agda is a total language. This means that all computations of inductive type must be terminating, and that all computations of coinductive type must be productive. A computation is productive if the computation of the next constructor is always terminating, so even though an infinite stream cannot be computed in finite time we know that the computation of any finite prefix has to be terminating. For types which are partly inductive and partly coinductive things become a little more complicated: the inductive parts must always be computable in finite time, while the coinductive parts must always be productively computable.

To ensure termination and productivity Agda employs two basic means for defining functions: inductive values can be destructed using structural recursion, and coinductive values can be constructed using guarded corecursion. As an example of the latter, consider the following definition of *map* for streams:

```
map : ∀ {A B} → (A → B) → Stream A → Stream B
map f (x :: xs) = f x :: ♯ map f (♭ xs)
```

(Note that the code $\forall\ \{A\ B\} \to \ldots$ means that the function takes two implicit arguments A and B; it is not an application of A to B.) Agda accepts this definition because the corecursive call is guarded by the coinductive constructor $\sharp\_$, without any non-constructor function between the left-hand side and the corecursive call. It is easy to convince oneself that, if the input stream is productively computable, then the (spine of the) output stream must also be.

Let us now consider what happens if a definition uses both induction and coinduction. We can define a language of stream processors (Hancock et al. 2009), taking streams of As to streams of Bs, as follows:

```
data SP (A B : Set) : Set where
  get : (A → SP A B)      → SP A B
  put : B → ∞ (SP A B) → SP A B
```

The recursive argument of get is inductive, while the recursive argument of put is coinductive. This means that a stream processor can only read a finite number of elements from the input before having to produce some output. The semantics of stream processors can be defined as follows:

```
⟦_⟧ : ∀ {A B} → SP A B → Stream A → Stream B
⟦ get f      ⟧ (a :: as) = ⟦ f a ⟧ (♭ as)
⟦ put b sp ⟧ as           = b :: ♯ ⟦ ♭ sp ⟧ as
```

In the case of get one element from the input stream is consumed, and potentially used to guide the rest of the computation, while in the case of put one output element is produced. The definition of $[\![\_]\!]$ uses a lexicographic combination of guarded corecursion and structural recursion: in the second clause the corecursive call is guarded, while in the first clause the recursive call "preserves guardedness" (it takes place under zero coinductive constructors rather than at least one) and the stream processor argument is structurally smaller. This ensures the productivity of the resulting stream: there can only be a finite number of get constructors between any two put constructors, so the next output element can always be computed in finite time.

For more examples of the use of mixed induction and coinduction, see Danielsson and Altenkirch (2009).[3]

3 Recognisers

This section defines a small embedded language of parser combinators. The semantics of these combinators is given in Section 3.2, and Section 3.3 shows how to the parsers can be executed. For simplicity the parser combinators defined in this section can only handle recognition. Full parser combinators are discussed in Section 4.

The aim is to define a data type with (at least) the following basic combinators as constructors: $\emptyset$, which always fails; ε, which accepts the empty string; sat, which accepts tokens satisfying a predicate; $\_|\_$, symmetric choice; and $\_\cdot\_$, sequencing.

Let us first consider whether the combinator arguments should be read inductively or coinductively. An infinite choice cannot be decided (in the absence

[3]This unpublished draft contains a discussion of the recognisers presented in Section 3.

of extra information), as this is not possible without inspecting every alternative, so I choose to read choices inductively. The situation is a bit trickier for sequencing. Consider definitions like $p = p \cdot p'$ or $p = p' \cdot p$. If p' accepts the empty string, then it seems hard to make any progress with these definitions. However, if p' is guaranteed not to accept the empty string, then we know that any string accepted by the recursive occurrence of p has to be shorter than the one accepted by $p \cdot p'$ or $p' \cdot p$. To make use of this observation I will indicate whether or not a recogniser is nullable (accepts the empty string) in its type, and the left (right) argument of $\_\cdot\_$ will be coinductive iff the right (left) argument is not nullable.

The "conditional coinduction" which will be used in the definition of $\_\cdot\_$ is encoded using the following data type:

```
data ∞? (A : Set) : Bool → Set where
  ⟨_⟩  :     A → ∞? A true
  ⟪_⟫  : ∞ A → ∞? A false
```

For convenience the index true is used for the inductive case, and false for the coinductive case. The type comes with corresponding conditional delay and force operators:

```
♯? : ∀ {b A} → A → ∞? A b
♯? {true}  x = ⟨   x ⟩
♯? {false} x = ⟪ ♯ x ⟫

♭? : ∀ {b A} → ∞? A b → A
♭? ⟨ x ⟩ =   x
♭? ⟪ x ⟫ = ♭ x
```

Based on the observations and definitions above the type P of parsers (recognisers) can now be defined:

```
data P : Bool → Set where
  ∅   : P false
  ε   : P true
  sat : (Tok → Bool) → P false
  _|_ : ∀ {n₁ n₂} →       P n₁     →       P n₂     → P (n₁ ∨ n₂)
  _·_ : ∀ {n₁ n₂} → ∞? (P n₁) n₂ → ∞? (P n₂) n₁ → P (n₁ ∧ n₂)
```

(*Tok* is the token type.) Note how the conditional coinduction operator is used to express whether the two arguments to the sequencing operator should be read inductively or coinductively. Note also that $\emptyset$ and sat do not accept the empty string, while ε does. A choice $p_1 \mid p_2$ is nullable if either p_1 or p_2 is, and a sequence $p_1 \cdot p_2$ is nullable if both p_1 and p_2 are.

For convenience the following combinators are also included:

```
nonempty : ∀ {n} → P n → P false
cast     : ∀ {n₁ n₂} → n₁ ≡ n₂ → P n₁ → P n₂
```

The nonempty combinator turns a recogniser which potentially accepts the empty string into one which definitely does not, and cast can be used to coerce a recogniser indexed by n_1 into a recogniser indexed by n_2, assuming that n_1 is equal to n_2.

3.1 Examples

Using the definition above it is easy to define recognisers which are both left and right recursive:

$$\begin{aligned} &\mathit{leftRight} \;:\; P\ \mathsf{false} \\ &\mathit{leftRight} \;=\; ⟪\ \sharp\ \mathit{leftRight}\ ⟫ \cdot ⟪\ \sharp\ \mathit{leftRight}\ ⟫ \end{aligned}$$

Using the semantics in Section 3.2 it is easy to show that *leftRight* does not accept any string (so $\emptyset$ could be a derived combinator).

As a more useful example of how the combinators above can be used to define derived recognisers, consider the following definition of the Kleene star:

mutual

$$\begin{aligned} &\_\star \;:\; P\ \mathsf{false} \to P\ \mathsf{true} \\ &p\ \star \;=\; \varepsilon \mid p\ + \\ &\_+ \;:\; P\ \mathsf{false} \to P\ \mathsf{false} \\ &p\ + \;=\; \langle\ p\ \rangle \cdot ⟪\ \sharp\ (p\ \star)\ ⟫ \end{aligned}$$

The recogniser $p\ \star$ accepts zero or more occurrences of whatever p accepts, and $p\ +$ accepts one or more occurrences; this is easy to prove using the semantics in Section 3.2. Note that this definition is guarded, and hence productive. Note also that p must not accept the empty string, because if it did, then the right hand side of $p\ +$ would have to be written $\langle\ p\ \rangle \cdot \langle\ p\ \star\ \rangle$, which would make the definition unproductive (the inductive “prefix” of $p\ \star$ would be infinite). By using the nonempty combinator one can define a variant of $\_\star$ which accepts arbitrary argument recognisers:

$$\begin{aligned} &\_\bigstar \;:\; \forall\ \{n\} \to P\ n \to P\ \mathsf{true} \\ &p\ \bigstar \;=\; \mathsf{nonempty}\ p\ \star \end{aligned}$$

The example above is very small; larger examples can also be constructed. For instance, Danielsson and Norell (2009) construct mixfix operator grammars using a parser combinator library which is based on the ideas described here.

3.2 Semantics

The semantics of the recognisers is defined as an inductive family. The type $s\ \in\ p$ is inhabited iff the token string s is a member of the language defined by p:

data $\_\in\_ \;:\; \mathit{List}\ \mathit{Tok} \to P\ n \to \mathit{Set}$ **where**
  ...

The semantics is determined by the constructors of $\_\in\_$, which are introduced below. To avoid clutter the declarations of bound variables are omitted in the constructors' type signatures.

No string is a member of the language defined by $\emptyset$, so there is no constructor for it in $\_\in\_$. The empty string is recognised by ε:

$$\varepsilon \;:\; [\,] \in \varepsilon$$

The singleton t is recognised by $\mathsf{sat}\ f$ if $f\ t$ evaluates to true ($T\ b$ is inhabited iff b is true):

$$\mathsf{sat}\ :\ T\ (f\ t) \to [t]\ \in\ \mathsf{sat}\ f$$

If s is recognised by p_1, then it is also recognised by $p_1 \mid p_2$, and similarly for p_2:

$$\begin{aligned}
&|^{\mathrm{l}}\ :\ s\ \in\ p_1 \to s\ \in\ p_1 \mid p_2\\
&|^{\mathrm{r}}\ :\ s\ \in\ p_2 \to s\ \in\ p_1 \mid p_2
\end{aligned}$$

If s_1 is recognised by p_1 (suitably forced), and s_2 is recognised by p_2 (suitably forced), then the concatenation of s_1 and s_2 is recognised by $p_1 \cdot p_2$:

$$\_\cdot\_\ :\ s_1\ \in\ \flat^?\ p_1 \to s_2\ \in\ \flat^?\ p_2 \to s_1\ +\!\!+\ s_2\ \in\ p_1 \cdot p_2$$

If a nonempty string is recognised by p, then it is also recognised by $\mathsf{nonempty}\ p$:

$$\mathsf{nonempty}\ :\ t :: s\ \in\ p \to t :: s\ \in\ \mathsf{nonempty}\ p$$

Finally cast preserves the semantics of its argument:

$$\mathsf{cast}\ :\ s\ \in\ p \to s\ \in\ \mathsf{cast}\ \mathit{eq}\ p$$

It is easy to show that the semantics and the nullability index agree: if $p\ :\ P\ n$, then $[\,]\ \in\ p$ iff n is equal to true (both directions can be proved by induction on the structure of the semantics). Given this result it is easy to decide whether or not $[\,]\ \in\ p$; it suffices to inspect the index:

$$\mathit{nullable?}\ :\ \forall\ \{n\}\ (p\ :\ P\ n) \to \mathit{Dec}\ ([\,]\ \in\ p)$$

The type $\mathit{Dec}\ P$ states that P is decidable; an element of $\mathit{Dec}\ P$ is either a proof of P or a proof showing that P is impossible:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Dec}\ (P\ :\ \mathit{Set})\ :\ \mathit{Set}\ \mathbf{where}\\
&\quad \mathsf{yes}\ :\ \quad\ P \to \mathit{Dec}\ P\\
&\quad \mathsf{no}\ \ :\ \neg\ P \to \mathit{Dec}\ P
\end{aligned}$$

Here logical negation is represented as a function into the empty type $\bot$:

$$\begin{aligned}
&\neg\_\ :\ \mathit{Set} \to \mathit{Set}\\
&\neg\ P\ =\ P \to \bot
\end{aligned}$$

3.3 Backend

Let us now consider how the relation $\_\in\_$ can be decided, or alternatively, how the language of recognisers can be interpreted. No attempt is made to make this recogniser backend efficient, the focus is on correctness.

The backend will be implemented using so-called derivatives (Brzozowski 1964). The derivative $\partial\ p\ t$ of p with respect to t is the "remainder" of p after p has matched the token t; it should satisfy the equivalence

$$s\ \in\ \partial\ p\ t\ \ \leftrightarrow\ \ t :: s\ \in\ p.$$

By applying the derivative operator ∂ to p and t_1, then to $\partial\ p\ t_1$ and t_2, and so on for every element in the input string s, one can decide if $s\ \in\ p$ is inhabited.

The new recogniser constructed by ∂ may not have the same nullability index as the original one, so ∂ has the following type signature:

$$\partial\ :\ \forall\ \{n\}\ (p\ :\ P\ n)\ (t\ :\ \mathit{Tok}) \rightarrow P\ (\partial^{\mathrm{n}}\ p\ t)$$

The extensional behaviour of $\partial^{\mathrm{n}}\ :\ \forall\ \{n\} \rightarrow P\ n \rightarrow \mathit{Tok} \rightarrow \mathit{Bool}$ is uniquely constrained by the definition of ∂; its definition is included below.

The derivative operator is implemented as follows. The combinators $\emptyset$ and ε never accept any token, so they both have the derivative $\emptyset$:

$$\begin{aligned} &\partial\ \emptyset\ t\ =\ \emptyset \\ &\partial\ \varepsilon\ t\ =\ \emptyset \end{aligned}$$

The combinator $\mathsf{sat}\ f$ has a non-zero derivative with respect to t iff $f\ t$ is true:

$$\begin{aligned} &\partial\ (\mathsf{sat}\ f)\ t\ \mathbf{with}\ f\ t \\ &\partial\ (\mathsf{sat}\ f)\ t\ \mid\ \mathsf{true}\ \ =\ \varepsilon \\ &\partial\ (\mathsf{sat}\ f)\ t\ \mid\ \mathsf{false}\ =\ \emptyset \end{aligned}$$

(The **with** construct is used to pattern match on the result of an intermediate computation.) The derivatives of $\mathsf{nonempty}\ p$ and $\mathsf{cast}\ \mathit{eq}\ p$ are equal to the derivatives of p:

$$\begin{aligned} &\partial\ (\mathsf{nonempty}\ p)\ t\ =\ \partial\ p\ t \\ &\partial\ (\mathsf{cast}\ \_\qquad p)\ t\ =\ \partial\ p\ t \end{aligned}$$

The derivative of a choice is the choice of the derivatives of its arguments:

$$\partial\ (p_1 \mid p_2)\ t\ =\ \partial\ p_1\ t \mid \partial\ p_2\ t$$

The final and most interesting case is sequencing. If p_1 is nullable, then the result is implemented as a choice, because the remainder of $p_1 \cdot \langle\ p_2\ \rangle$ could be either the remainder of p_1 followed by p_2, or the remainder of p_2:

$$\begin{aligned} &\partial\ (\langle\ p_1\ \rangle\ \cdot \langle\ p_2\ \rangle)\ t\ =\ \langle\quad \partial\quad p_1\ \ t\ \ \rangle \cdot \sharp^{?}\ p_2\ \ \mid\ \ \partial\ p_2\ t \\ &\partial\ (\langle\!\langle\ p_1\ \rangle\!\rangle \cdot \langle\ p_2\ \rangle)\ t\ =\ \langle\!\langle\ \sharp\ (\partial\ (^{\flat}\ p_1)\ t)\ \rangle\!\rangle \cdot \sharp^{?}\ p_2\ \ \mid\ \ \partial\ p_2\ t \end{aligned}$$

(Note that we may need to delay p_2, depending on the nullability index of the derivative of p_1.) If p_1 is not nullable, then the second choice above should not be included, because the first token which is accepted (if any) has to be accepted by p_1:

$$\begin{aligned} &\partial\ (\langle\ p_1\ \rangle\ \cdot \langle\!\langle\ p_2\ \rangle\!\rangle)\ t\ =\ \langle\quad \partial\quad p_1\ \ t\ \ \rangle \cdot \sharp^{?}\ (^{\flat}\ p_2) \\ &\partial\ (\langle\!\langle\ p_1\ \rangle\!\rangle \cdot \langle\!\langle\ p_2\ \rangle\!\rangle)\ t\ =\ \langle\!\langle\ \sharp\ (\partial\ (^{\flat}\ p_1)\ t)\ \rangle\!\rangle \cdot \sharp^{?}\ (^{\flat}\ p_2) \end{aligned}$$

The derivative operator ∂ is implemented using a lexicographic combination of guarded corecursion and structural recursion. Note that in the last two sequencing cases p_2 is delayed, but ∂ is not applied recursively to p_2 because p_1 is known not to accept the empty string.

The index function ∂^{n} uses recursion over the *inductive* structure of the recogniser:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\partial^{\mathrm{n}}\ :\ \forall\ \{n\} \to P\ n \to \mathit{Tok} \to \mathit{Bool}} \\
\partial^{\mathrm{n}}\ \emptyset \hfill t & = & \mathsf{false} \\
\partial^{\mathrm{n}}\ \varepsilon \hfill t & = & \mathsf{false} \\
\partial^{\mathrm{n}}\ (\mathsf{sat}\ f) \hfill t & = & f\ t \\
\partial^{\mathrm{n}}\ (\mathsf{nonempty}\ p) \hfill t & = & \partial^{\mathrm{n}}\ p\ t \\
\partial^{\mathrm{n}}\ (\mathsf{cast}\ \_\ p) \hfill t & = & \partial^{\mathrm{n}}\ p\ t \\
\partial^{\mathrm{n}}\ (p_1 \mid p_2) \hfill t & = & \partial^{\mathrm{n}}\ p_1\ t \vee \partial^{\mathrm{n}}\ p_2\ t \\
\partial^{\mathrm{n}}\ (\langle\ p_1\ \rangle\ \cdot \langle\ p_2\ \rangle)\ t & = & \partial^{\mathrm{n}}\ p_1\ t \vee \partial^{\mathrm{n}}\ p_2\ t \\
\partial^{\mathrm{n}}\ (\langle\!\langle\ p_1\ \rangle\!\rangle \cdot \langle\ p_2\ \rangle)\ t & = & \partial^{\mathrm{n}}\ p_2\ t \\
\partial^{\mathrm{n}}\ (\langle\ p_1\ \rangle\ \cdot \langle\!\langle\ p_2\ \rangle\!\rangle)\ t & = & \partial^{\mathrm{n}}\ p_1\ t \\
\partial^{\mathrm{n}}\ (\langle\!\langle\ p_1\ \rangle\!\rangle \cdot \langle\!\langle\ p_2\ \rangle\!\rangle)\ t & = & \mathsf{false}
\end{array}
$$

Note that ∂^{n} does not force any delayed recogniser. Readers familiar with dependent types may find it interesting that this definition relies on the fact that $\_\wedge\_$ is defined by pattern matching on its *right* argument. If $\_\wedge\_$ were defined by pattern matching on its left argument, then the type checker would no longer reduce the open term $\partial^{\mathrm{n}}\ ({}^{\flat}\ p_1)\ t \wedge \mathsf{false}$ to false when checking the definition of ∂. This problem could be fixed by using cast in the definition of ∂, though.

It is straightforward to show that the derivative operator ∂ satisfies both directions of its specification:

$$
\begin{array}{lll}
\partial\textit{-sound} & : & s\ \in\ \partial\ p\ t \to t :: s\ \in\ p \\
\partial\textit{-complete} & : & t :: s\ \in\ p \to s\ \in\ \partial\ p\ t
\end{array}
$$

These statements can be proved by induction on the structure of the semantics.

Once the derivative operator is defined and proved correct it is easy to decide whether $s\ \in\ p$ is inhabited:

$$
\begin{array}{l}
\_\in?\_\ :\ \forall\ \{n\}\ (s\ :\ \mathit{List\ Tok})\ (p\ :\ P\ n) \to \mathit{Dec}\ (s\ \in\ p) \\
[\,]\quad\ \in?\ p = \ \mathit{nullable}?\ p \\
t :: s\ \in?\ p\ \mathbf{with}\ s\ \in?\ \partial\ p\ t \\
t :: s\ \in?\ p\ \mid\ \mathsf{yes}\ s{\in}\partial pt = \ \mathsf{yes}\ (\partial\textit{-sound}\ s{\in}\partial pt) \\
t :: s\ \in?\ p\ \mid\ \mathsf{no}\ \ s{\notin}\partial pt = \ \mathsf{no}\ \ (s{\notin}\partial pt \circ \partial\textit{-complete})
\end{array}
$$

In the case of the empty string the nullability index tells us whether the string should be accepted or not, and otherwise $\_\in?\_$ is recursively applied to the derivative and the tail of the string; the specification of ∂ ensures that this is correct.

As an aside, note that the proof returned by $\_\in?\_$ when a string matches is actually a parse tree, so it would not be entirely incorrect to call these recognisers parsers. However, in the case of ambiguous grammars at most one parse tree is returned.

3.4 Laws

Given the semantics above it is easy to derive some laws about the combinators. Let us first define that two recognisers are equivalent when they accept the same strings:

$$
\begin{array}{l}
\_\leqslant\_\ :\ \forall\ \{n_1\ n_2\} \to P\ n_1 \to P\ n_2 \to \mathit{Set} \\
p_1\ \leqslant\ p_2\ =\ \forall\ \{s\} \to s\ \in\ p_1 \to s\ \in\ p_2
\end{array}
$$

$$\begin{array}{l}
\_{\approx}\_ \;:\; \forall\, \{n_1\; n_2\} \to P\; n_1 \to P\; n_2 \to \mathit{Set} \\
p_1 \;\approx\; p_2 \;=\; p_1 \;\leqslant\; p_2 \times p_2 \;\leqslant\; p_1
\end{array}$$

It is straightforward to show that $\_{\approx}\_$ is an equivalence relation, if the definition of "equivalence relation" is generalised to accept *indexed* sets. (Such generalisations are silently assumed in the remainder of this text.) It is also easy to show that $\_{\approx}\_$ is preserved by all the primitive recogniser combinators, and that $\_{\leqslant}\_$ is a partial order with respect to $\_{\approx}\_$.

The following definition provides an alternative, coinductive characterisation of equality:

$$\begin{array}{l}
\textbf{data}\; \_{\approxeq}\_ \; \{n_1\; n_2\}\; (p_1 \;:\; P\; n_1)\; (p_2 \;:\; P\; n_2) \;:\; \mathit{Set}\; \textbf{where} \\
\quad \_{::}\_ \;:\; n_1 \equiv n_2 \to (\forall\; t \to \infty\; (\partial\; p_1\; t \approxeq \partial\; p_2\; t)) \to p_1 \approxeq p_2
\end{array}$$

Two recognisers are equal iff they agree on whether the empty string is accepted, and all their derivatives are equal (coinductively). This is a form of bisimilarity. It is easy to show that $\_{\approx}\_$ and $\_{\approxeq}\_$ are equivalent.

Let us introduce the following variant of the sequencing combinator:

$$\begin{array}{l}
\_{\odot}\_ \;:\; \forall\, \{n_1\; n_2\} \to P\; n_1 \to P\; n_2 \to P\; (n_1 \wedge n_2) \\
p_1 \odot p_2 \;=\; \sharp^?\; p_1 \cdot \sharp^?\; p_2
\end{array}$$

Using this combinator it is easy to prove that the recognisers form an idempotent semiring:

$$\begin{array}{llllll}
p_1 \;\mid p_2 & \approx & p_2 \;\mid p_1 & p \;\odot \varepsilon & \approx & p \\
\emptyset \;\mid p & \approx & p & \varepsilon \;\odot p & \approx & p \\
p_1 \;\mid (p_2 \mid p_3) & \approx & (p_1 \mid p_2) \mid p_3 & p_1 \odot (p_2 \odot p_3) & \approx & (p_1 \odot p_2) \odot p_3 \\
p_1 \odot (p_2 \mid p_3) & \approx & p_1 \odot p_2 \mid p_1 \odot p_3 & \emptyset \;\odot p & \approx & \emptyset \\
(p_1 \mid p_2) \odot p_3 & \approx & p_1 \odot p_3 \mid p_2 \odot p_3 & p \;\odot \emptyset & \approx & \emptyset \\
p \;\mid p & \approx & p & & &
\end{array}$$

Note that the order $\_{\leqslant}\_$ coincides with the natural order of the join-semilattice formed by $\_{\mid}\_$:

$$p_1 \;\leqslant\; p_2 \qquad \Leftrightarrow \qquad p_1 \mid p_2 \;\approx\; p_2$$

By using the generalised Kleene star $\_\bigstar$ from Section 3.1 one can also show that the recognisers form a $\bigstar$-continuous Kleene algebra (Kozen 1990): $p_1 \odot p_2 \bigstar \odot p_3$ is the least upper bound of the set $\{\, p_1 \odot p_2 \,\hat{}\, i \odot p_3 \mid i \in \mathbb{N}\,\}$, where $p \,\hat{}\, i$ is the i-fold repetition of p:

$$\begin{array}{l}
\_\hat{}^{\mathrm{n}}\_ \;:\; \mathit{Bool} \to \mathbb{N} \to \mathit{Bool} \\
\_\hat{}^{\mathrm{n}}\; \mathsf{zero} \;\;=\; \_ \\
\_\hat{}^{\mathrm{n}}\; \mathsf{suc}\; \_ \;=\; \_ \\
\_\hat{}\_ \;:\; \forall\, \{n\} \to P\; n \to (i \;:\; \mathbb{N}) \to P\; (n\, \hat{}^{\mathrm{n}}\; i) \\
p\,\hat{}\,\mathsf{zero} \;\;=\; \varepsilon \\
p\,\hat{}\,\mathsf{suc}\; i \;=\; p \odot p\,\hat{}\, i
\end{array}$$

(Note that Agda can figure out the right-hand sides of $\_\hat{}^{\mathrm{n}}\_$ automatically, given the definition of $\_\hat{}\_$.)

3.5 Expressive strength

Is the language of recognisers defined above useful? It may not be entirely obvious that the restrictions imposed to ensure totality do not rule out the definition of many useful recognisers. Fortunately this is not the case, at least not if *Tok*, the set of tokens, is finite, because then it can be proved that every function of type *List Tok* → *Bool* which can be implemented in Agda can also be realised as a recogniser. For simplicity this will only be shown in the case when *Tok* is *Bool*. The basic idea is to turn a function *f* : *List Bool* → *Bool* into a grammar representing an infinite binary tree, with one node for every possible input string, and to make a given node accepting iff *f* returns true for the corresponding string.

Let us first define a recogniser which only accepts the empty string, and only if its argument is true:

```
accept-if-true : ∀ b → P b
accept-if-true true  = ε
accept-if-true false = ∅
```

The following recogniser which only accepts a given token will also be used:

```
tok : Bool → P false
tok b = sat (λ b′ → b == b′)
```

Using these recognisers we can construct the "infinite binary tree" using guarded corecursion:

```
grammar : (f : List Bool → Bool) → P (f [])
grammar f = cast (lemma f) (
                ♯? (tok true)  · ⟪ ♯ grammar (f ∘ _∷_ true ) ⟫
              | ♯? (tok false) · ⟪ ♯ grammar (f ∘ _∷_ false) ⟫
              | accept-if-true (f []))
```

The following lemma is used above:

```
lemma : ∀ f → (false ∧ f [true] ∨ false ∧ f [false]) ∨ f [] ≡ f []
```

The final step is to show that, for any string s, $f\ s \equiv$ true iff $s \in$ *grammar f*. The "only if" part can be proved by induction on the structure of s, and the "if" part by induction on the structure of $s \in$ *grammar f*.

Note that the grammar above has a very simple structure: it is LL(1). I suspect that most parser combinator libraries can handle infinite LL(1) grammars (with finite branching), so the result above should carry over to most other parser combinator libraries.

As an aside it may be interesting to know that the proof above does not require the use of cast. The following left recursive grammar can also be used:

```
grammar : (f : List Bool → Bool) → P (f [])
grammar f = ⟪ ♯ grammar (λ xs → f (xs ++ [true ])) ⟫ · ♯? (tok true)
          | ⟪ ♯ grammar (λ xs → f (xs ++ [false])) ⟫ · ♯? (tok false)
          | accept-if-true (f [])
```

Note that this shows that nonempty and cast are not necessary to achieve full expressive strength, because neither *grammar* nor the backend use these operators.

Finally let us consider the case of infinite token sets. If the set of tokens is the natural numbers, then it is quite easy to see that it is impossible to implement a recogniser which accepts a string iff it has the form nn for some natural number n. By generalising the statement to "it is impossible that p accepts infinitely many identical pairs, and only identical pairs and the empty string" (where an identical pair is a string of the form nn) one can prove this formally by induction over the structure of p.

4 Parsers

This section outlines how the recogniser language above can be extended to actual parser combinators, which return results.

Consider the parser combinator bind, $\_{\gg\!=}\_$: The parser $p_1 \gg\!= p_2$ successfully returns a value y for a given string s if p_1 parses a prefix of s, returning a value x, and $p_2\ x$ parses the rest of s, returning y. Note that $p_1 \gg\!= p_2$ accepts the empty string iff p_1 accepts the empty string, returning a value x, and $p_2\ x$ also accepts the empty string. This shows that the values which a parser p can return without consuming any input—let us call them the *initial set* of p—can be relevant for determining whether another parser is nullable.

Parsers are defined analogously to the recognisers in Section 3, but taking the observation above into account. Parsers will be indexed on their return types and their initial sets (represented as lists), and the type $\infty^?$, a variant of the conditional coinduction data type introduced in Section 3, will be used to only allow coinduction when certain initial sets are empty:

$$
\begin{array}{l}
\textbf{data}\ \infty^?\ (A\ :\ Set_1)\ \{B\ :\ Set\}\ :\ List\ B \to Set_1\ \textbf{where} \\
\quad \langle\!\langle \_ \rangle\!\rangle\ :\ \qquad\qquad\quad \infty\ A \to \infty^?\ A\ [\,] \\
\quad \langle \_ \rangle\ \ \ :\ \forall\ \{x\ xs\} \to \quad A \to \infty^?\ A\ (x :: xs)
\end{array}
$$

(Set_1 is a type of large types; ∞ works also for Set_1.) It is straightforward to define variants of $\sharp^?$ and $\flat^?$ for this variant of $\infty^?$.

The type of parsers is defined as a data type which uses both induction and coinduction:

$$\textbf{data}\ Parser\ :\ (R\ :\ Set) \to List\ R \to Set_1\ \textbf{where}$$

The return combinator is the parser analogue of ε. When accepting the empty string it returns its argument:

$$\mathsf{return}\ :\ \forall\ \{R\}\ (x\ :\ R) \to Parser\ R\ [x]$$

(Note that $[\_]$ is the return function of the list monad.) The fail parser, which mirrors $\emptyset$, always fails:

$$\mathsf{fail}\ :\ \forall\ \{R\} \to Parser\ R\ [\,]$$

(Note that $[\,]$ is the zero of the list monad.) The token parser accepts any single token, and returns this token:

$$\mathsf{token}\ :\ \mathit{Parser}\ \mathit{Tok}\ [\,]$$

This combinator is not as general as sat, but sat is easy to define using token and bind. The initial set of the choice combinator, $\_|\_$, is the union of the initial sets of its two arguments:

$$\begin{aligned}\_|\_\ :\ & \forall\ \{R\ \mathit{xs_1}\ \mathit{xs_2}\} \rightarrow \\ & \mathit{Parser}\ R\ \mathit{xs_1} \rightarrow \mathit{Parser}\ R\ \mathit{xs_2} \rightarrow \mathit{Parser}\ R\ (\mathit{xs_1}\ +\!\!+\ \mathit{xs_2})\end{aligned}$$

The analogue of $\_\cdot\_$ is the combinator $\_\circledast\_$, applicative functor application (McBride and Paterson 2008):

$$\begin{aligned}\_\circledast\_\ :\ & \forall\ \{R_1\ R_2\ \mathit{fs}\ \mathit{xs}\} \rightarrow \\ & \infty^{?}\ (\mathit{Parser}\ (R_1 \rightarrow R_2)\ \mathit{fs})\ \mathit{xs} \rightarrow \infty^{?}\ (\mathit{Parser}\ R_1\ \mathit{xs})\ \mathit{fs} \rightarrow \\ & \mathit{Parser}\ R_2\ (\mathit{fs}\ \underline{\circledast}\ \mathit{xs})\end{aligned}$$

The left argument returns functions of type $R_1 \rightarrow R_2$, which are applied to the results of the right argument. (Here $\_\underline{\circledast}\_$ is the list monad's applicative functor application.) The bind combinator, which does not have an analogue in Section 3, has the following type signature:

$$\begin{aligned}\_\gg\!=\_\ :\ & \forall\ \{R_1\ R_2\ \mathit{xs}\}\ \{f\ :\ R_1 \rightarrow \mathit{List}\ R_2\} \rightarrow \\ & \mathit{Parser}\ R_1\ \mathit{xs} \rightarrow ((x\ :\ R_1) \rightarrow \infty^{?}\ (\mathit{Parser}\ R_2\ (f\ x))\ \mathit{xs}) \rightarrow \\ & \mathit{Parser}\ R_2\ (\mathit{xs}\ \underline{\gg\!=}\ f)\end{aligned}$$

(Here $\_\underline{\gg\!=}\_$ is the list monad's bind operator.) Note that (the result of) bind's right argument is coinductive iff the left argument does not accept the empty string. Note also that the right argument's initial set does not need to be the same for every input to the function. The final two combinators, the analogues of nonempty and cast, are unsurprising:

$$\begin{aligned}\mathsf{nonempty}\ &:\ \forall\ \{R\ \mathit{xs}\} \rightarrow \mathit{Parser}\ R\ \mathit{xs} \rightarrow \mathit{Parser}\ R\ [\,] \\ \mathsf{cast}\ &:\ \forall\ \{R\ \mathit{xs_1}\ \mathit{xs_2}\} \rightarrow \\ &\quad\ \mathit{xs_1} \equiv \mathit{xs_2} \rightarrow \mathit{Parser}\ R\ \mathit{xs_1} \rightarrow \mathit{Parser}\ R\ \mathit{xs_2}\end{aligned}$$

The semantics of the parser combinators is defined as a relation $\_\in\_\cdot\_$, such that $x\ \in\ p \cdot s$ is inhabited iff x is one of the results of parsing the string s using the parser p. This relation is defined in Figure 1. Following Section 3.3 it is easy to implement a parser backend

$$\mathit{parse}\ :\ \forall\ \{R\ \mathit{xs}\} \rightarrow \mathit{Parser}\ R\ \mathit{xs} \rightarrow \mathit{List}\ \mathit{Tok} \rightarrow \mathit{List}\ R$$

which is sound and complete with respect to $\_\in\_\cdot\_$:

$$\begin{aligned}\mathit{sound}\ &:\ \forall\ \{R\ \mathit{xs}\ x\}\ \{p\ :\ \mathit{Parser}\ R\ \mathit{xs}\}\ (s\ :\ \mathit{List}\ \mathit{Tok}) \rightarrow \\ &\quad\ x\ \in\ \mathit{parse}\ p\ s \rightarrow x\ \in\ p \cdot s \\ \mathit{complete}\ &:\ \forall\ \{R\ \mathit{xs}\ x\}\ \{p\ :\ \mathit{Parser}\ R\ \mathit{xs}\}\ (s\ :\ \mathit{List}\ \mathit{Tok}) \rightarrow \\ &\quad\ x\ \in\ p \cdot s \rightarrow x\ \in\ \mathit{parse}\ p\ s\end{aligned}$$

(Here $\_\in\_\ :\ \{A\ :\ \mathit{Set}\} \rightarrow A \rightarrow \mathit{List}\ A \rightarrow \mathit{Set}$ represents list membership.) See the accompanying code for details of this implementation.

Note that the properties above are weaker than what one might want: the list returned by $\mathit{parse}\ p\ s$ might contain several identical values (the use of

$$
\begin{array}{lll}
\textbf{data}\ \_{\in}\_{\cdot}\_ & : & \forall\ \{R\ xs\} \to R \to \mathit{Parser}\ R\ xs \to \mathit{List}\ \mathit{Tok} \to \mathit{Set}_1\ \textbf{where} \\
\quad \mathsf{return} & : & x\ \in\ \mathsf{return}\ x \cdot [\,] \\
\quad \mathsf{token} & : & t\ \in\ \mathsf{token} \cdot [\,t\,] \\
\quad |^{\mathrm{l}} & : & x\ \in\ p_1 \cdot s \to x\ \in\ p_1 \mid p_2 \cdot s \\
\quad |^{\mathrm{r}} & : & x\ \in\ p_2 \cdot s \to x\ \in\ p_1 \mid p_2 \cdot s \\
\quad \_{\circledast}\_ & : & f\ \in\ \flat^{?}\ p_1 \cdot s_1 \to x\ \in\ \flat^{?}\ p_2 \cdot s_2 \to \\
 & & f\ x\ \in\ p_1 \circledast p_2 \cdot s_1\ +\!\!+\ s_2 \\
\quad \_{\ggg\!=}\_ & : & x\ \in\ p_1 \cdot s_1 \to y\ \in\ \flat^{?}\ (p_2\ x) \cdot s_2 \to \\
 & & y\ \in\ p_1\ \ggg\!=\ p_2 \cdot s_1\ +\!\!+\ s_2 \\
\quad \mathsf{nonempty} & : & y\ \in\ p \cdot t :: s \to y\ \in\ \mathsf{nonempty}\ p \cdot t :: s \\
\quad \mathsf{cast} & : & x\ \in\ p \cdot s \to x\ \in\ \mathsf{cast}\ \mathit{eq}\ p \cdot s
\end{array}
$$

Figure 1: The semantics of the parser combinators. To avoid clutter the declarations of bound variables are omitted in the constructors' type signatures.

list membership only gives us set equality, not bag equality). However, it is fine for the output of *parse p s* to contain several instances of a value x if p is sufficiently ambiguous, i.e. if $x\ \in\ p \cdot s$ contains at least as many distinct proofs. Fortunately we can prove that *complete* is a left inverse of *sound*:

$$
\begin{array}{ll}
\textit{left-inverse} : & \forall\ \{R\ xs\ x\}\ (s\ :\ \mathit{List}\ \mathit{Tok})\ (p\ :\ \mathit{Parser}\ R\ xs) \\
 & (\mathit{proof}\ :\ x\ \in\ \mathit{parse}\ p\ s) \to \\
 & \mathit{complete}\ s\ (\mathit{sound}\ s\ \mathit{proof}) \equiv \mathit{proof}
\end{array}
$$

Hence the (finite) type $x\ \in\ \mathit{parse}\ p\ s$ contains at most as many proofs as $x\ \in\ p \cdot s$.

Many of the laws from Section 3.4 can be generalised to the setting of parser combinators, with equality defined as follows:

$$
\begin{array}{l}
\_{\leqslant}\_\ :\ \forall\ \{R\ xs_1\ xs_2\} \to \mathit{Parser}\ R\ xs_1 \to \mathit{Parser}\ R\ xs_2 \to \mathit{Set}_1 \\
p_1\ \leqslant\ p_2\ =\ \forall\ \{x\ s\} \to x\ \in\ p_1 \cdot s \to x\ \in\ p_2 \cdot s \\
\_{\approx}\_\ :\ \forall\ \{R\ xs_1\ xs_2\} \to \mathit{Parser}\ R\ xs_1 \to \mathit{Parser}\ R\ xs_2 \to \mathit{Set}_1 \\
p_1\ \approx\ p_2\ =\ p_1\ \leqslant\ p_2 \times p_2\ \leqslant\ p_1
\end{array}
$$

(Note that this equality allows us to prove that $\_|\_$ is idempotent, even though *parse* $(p \mid p)\ s$ may well give more results than *parse p s*.) For instance, one can prove that return and $\_{\odot}\_$ satisfy the applicative functor laws, where $\_{\odot}\_$ is defined as follows:

$$
\begin{array}{l}
\_{\odot}\_\ :\ \forall\ \{R_1\ R_2\ \mathit{fs}\ xs\} \to \\
\qquad \mathit{Parser}\ (R_1 \to R_2)\ \mathit{fs} \to \mathit{Parser}\ R_1\ xs \to \mathit{Parser}\ R_2\ (\mathit{fs}\ \underline{\circledast}\ xs) \\
p_1 \odot p_2\ =\ \sharp^{?}\ p_1 \circledast \sharp^{?}\ p_2
\end{array}
$$

By defining a variant of $\_{\ggg\!=}\_$ one can also prove that the monad laws are satisfied. However, assuming that the token type is inhabited it is not possible to define a Kleene-star-like combinator

$$
\_{\bigstar}\ :\ \forall\ \{R\ xs\}\ (p\ :\ \mathit{Parser}\ R\ xs) \to \mathit{Parser}\ (\mathit{List}\ R)\ (f\ p)
$$

(for any f) such that

$$\mathsf{return}\ [\,]\ |\ \mathsf{return}\ \_{::}\_ \odot p \odot (p \bigstar) \ \leqslant\ (p \bigstar)$$

holds for all p, where $\_{\leqslant}\_$ is defined as in Section 3.4. The reason is that p may be nullable, in which case the inequality above implies that $xs \in p \bigstar \cdot [\,]$ must be satisfied for infinitely many lists xs, whereas the completeness of *parse* shows that a parser can only return a finite number of results. (A combinator $\_\bigstar$ satisfying the inequality above can easily be implemented if it is restricted to non-nullable argument parsers.)

Finally let us consider the expressiveness of the parser combinators. By using bind one can strengthen the result from Section 3.5 to arbitrary sets of tokens: every function of type *List Tok* → *List R* can be realised as a parser (if the order and multiplicity of elements in the lists of results are ignored). The construction of the infinite grammar mirrors the construction in Section 3.5:

```
grammar : ∀ {R} (f : List Tok → List R) → Parser R (f [])
grammar f = token ≫= (λ t → ⟪ ♯ grammar (f ∘ _::_ t) ⟫)
          | return⋆ (f [])
```

Here *return⋆* returns any of its arguments:

```
return⋆ : ∀ {R} (xs : List R) → Parser R xs
return⋆ []        = fail
return⋆ (x :: xs) = return x | return⋆ xs
```

For finite sets of tokens the construction in Section 3.5 still works (given an implementation of *tok*). This means that, for finite sets of tokens, the inclusion of the monadic bind combinator does not provide any expressive advantage (if token is replaced by sat); the applicative functor interface is already sufficiently expressive. This comparison does not take efficiency into account, though.

5 Conclusions

A parser combinator library which guarantees termination of parsing has been presented, and it has been established that the library is sufficiently expressive: every decision procedure which can be implemented in the host language can also be realised as a parser.

Acknowledgements

I would like to thank Ulf Norell for previous joint work on total parsing, and Thorsten Altenkirch for many discussions about mixed induction and coinduction. Thorsten also suggested that I should allow left recursive parsers, which I might otherwise not have tried.

I would also like to thank EPSRC and the Royal Swedish Academy of Sciences' funds for financial support.

References

The Agda Team. The Agda Wiki. Available at `http://wiki.portal.chalmers.se/agda/`, 2009.

Marcello Bonsangue, Jan Rutten, and Alexandra Silva. A Kleene theorem for polynomial coalgebras. In *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009*, volume 5504 of *LNCS*, pages 122–136, 2009.

Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology, 2001.

Koen Claessen. Parallel parsing processes. *Journal of Functional Programming*, 14:741–757, 2004.

Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Advances in Computing Science — ASIAN'99*, volume 1742 of *LNCS*, pages 62–73, 1999.

Nils Anders Danielsson and Thorsten Altenkirch. Mixing induction and coinduction. Draft, 2009.

Nils Anders Danielsson and Ulf Norell. Structurally recursive descent parsing. Presentation (given by Danielsson) at the Dependently Typed Programming workshop, Nottingham, UK, 2008.

Nils Anders Danielsson and Ulf Norell. Parsing mixfix operators. To appear in the proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008), 2009.

Jon Fairbairn. Making form follow function: An exercise in functional programming style. *Software: Practice and Experience*, 17(6):379–386, 1987.

Jeroen Fokker. Functional parsers. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 1–23, 1995.

Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In *PADL 2008: Practical Aspects of Declarative Languages*, volume 4902 of *LNCS*, pages 167–181, 2008.

Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

Peter Hancock, Dirk Pattinson, and Neil Ghani. Representations of stream processors using nested fixed points. *Logical Methods in Computer Science*, 5 (3:9), 2009.

R. John M. Hughes and S. Doaitse Swierstra. Polish parsers, step by step. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 239–248, 2003.

Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2:323–343, 1992.

Mark Johnson. Memoization in top-down parsing. *Computational Linguistics*, 21(3):405–417, 1995.

Pieter Koopman and Rinus Plasmeijer. Efficient combinator parsers. In *IFL'98: Implementation of Functional Languages*, volume 1595 of *LNCS*, pages 120–136, 1999.

Dexter Kozen. On Kleene algebras and closed semirings. In *Mathematical Foundations of Computer Science 1990*, volume 452 of *LNCS*, pages 26–47, 1990.

Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Department of Information and Computing Sciences, Utrecht University, 2001.

Paul Lickman. Parsing with fixed points. Master's thesis, University of Cambridge, 1995.

Peter Ljunglöf. Pure functional parsing; an advanced tutorial. Licentiate thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, 2002.

Antoni W. Mazurkiewicz. A note on enumerable grammars. *Information and Control*, 14(6):555–558, 1969.

Conor McBride and James McKinna. Seeing and doing. Presentation (given by McBride) at the Workshop on Termination and Type Theory, Hindås, Sweden, 2002.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18:1–13, 2008.

Erik Meijer. *Calculating Compilers.* PhD thesis, Nijmegen University, 1992.

Paul Francis Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, 1988.

Muad`Dib. Strongly specified parser combinators. Post to the Muad`Dib blog, 2009.

Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

J.J.M.M. Rutten. Automata and coinduction (an exercise in coalgebra). In *CONCUR'98, Concurrency Theory, 9th International Conference*, volume 1466 of *LNCS*, pages 547–554, 1998.

Niklas Röjemo. *Garbage collection, and memory efficiency, in lazy functional languages.* PhD thesis, Chalmers University of Technology and University of Göteborg, 1995.

Marvin Solomon. *Theoretical Issues in the Implementation of Programming Languages*. PhD thesis, Cornell University, 1977.

S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming*, volume 1129 of *LNCS*, pages 184–207, 1996.

Wouter Swierstra. A Hoare logic for the state monad. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *LNCS*, pages 440–451, 2009.

Philip Wadler. How to replace failure by a list of successes; a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 113–128, 1985.

Philip Wadler, Walid Taha, and David MacQueen. How to add laziness to a strict language, without even being odd. In *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 1998.

Malcolm Wallace. Partial parsing: Combining choice with commitment. In *IFL 2007: Implementation and Application of Functional Languages*, volume 5083 of *LNCS*, pages 93–110, 2008.