Total Parser Combinators

Nils Anders Danielsson

School of Computer Science, University of Nottingham, United Kingdom

nad@cs.nott.ac.uk

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 those 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 dependent types and 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.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; E.1 [*Data Structures*]; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.4.2 [*Mathematical Logic and Formal Languages*]: Grammars and Other Rewriting Systems—Grammar types, Parsing

General Terms Languages, theory, verification

Keywords Dependent types, mixed induction and coinduction, parser combinators, productivity, termination

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 implementing 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 for every finite input string.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

The library has an interface which is very similar to those of classical monadic parser combinator libraries. For instance, consider the following simple, left recursive, expression grammar:

$$\begin{aligned} \mathit{term} &::= \mathit{factor} \mid \mathit{term}\ \texttt{'+'}\ \mathit{factor} \\ \mathit{factor} &::= \mathit{atom} \mid \mathit{factor}\ \texttt{'*'}\ \mathit{atom} \\ \mathit{atom} &::= \mathit{number} \mid \texttt{'('}\ \mathit{term}\ \texttt{')'} \end{aligned}$$

We can define a parser which accepts strings from this grammar, and also computes the values of the resulting expressions, as follows (the combinators are described in Section 4):

```
mutual
  term   = factor
         | ♯ term          >>= λ n₁ →
           tok '+'         >>= λ _  →
           factor          >>= λ n₂ →
           return (n₁ + n₂)
  factor = atom
         | ♯ factor        >>= λ n₁ →
           tok '*'         >>= λ _  →
           atom            >>= λ n₂ →
           return (n₁ * n₂)
  atom   = number
         | tok '('         >>= λ _  →
           ♯ term          >>= λ n  →
           tok ')'         >>= λ _  →
           return n
```

The only visible difference to classical parser combinators is the use of $^\sharp$, which indicates that the definitions are *corecursive* (see Section 2). However, we will see later that the parsers' types contain more information than usual.

When using parser combinators the parsers/grammars are often constructed using cyclic definitions, as above, 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 the following ones:

$$\begin{aligned} p &= {}^\sharp p \mid {}^\sharp p \\ p' &= {}^\sharp p' \gg\!= \lambda\, x \to {}^\sharp\, \mathsf{return}\ (f\ x) \end{aligned}$$

For these definitions it is *impossible* to implement parsing in a total way (in the absence of hidden information): a defining characteristic of parser combinator libraries is that non-terminals are implicit, encoded using the recursion mechanism of the host language, so (in a pure setting) the only way to inspect p and p' is via their infinite unfoldings. The key idea of this paper is that, even if non-terminals are implicit, totality can be ensured by reading choice inductively, and only reading an argument of the sequencing operator coinductively if the other argument does not accept the empty string (see Section 3). To support this idea the parsers' types will contain information about whether or not they accept the empty string.

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, using a combination of induction and coinduction to represent parsers, and a variant of Brzozowski derivatives (1964) to run them.
- Unlike many other parser combinator libraries these parser combinators can handle many forms of left recursion.
- The parser combinators come with a formal semantics. The implementation is proved to be correct, and the combinators are shown to satisfy a number of laws.
- It is shown that the parser combinators are as expressive as possible (see Sections 3.5 and 4.5).

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. Related work is discussed below.

As mentioned above the parser type is defined using *mixed induction and coinduction* (Park 1980). This technique is explained in Section 2, and discussed further in the conclusions. Those readers who are not particularly interested in parser combinators may still find the paper useful as an example of the use of this technique.

The parser combinator library is defined in the dependently typed functional programming language Agda (Norell 2007; Agda Team 2010), which will be introduced as we go along. The library comes with a machine-checked[1] proof 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.

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) and Koprowski and Binsztok (2010) use similar ideas; Koprowski and Binsztok also prove full correctness. 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 Section 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. The 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—for instance, consider the grammar $\{\, p_n ::= p_{1+n} \mid n \in \mathbb{N} \,\}$, implemented by the definition $p\ n = \mathit{memoise}\ n\ (p\ (1 + n))$—and users of the libraries need to ensure manually that the combinators are used in the right way. The same limitations apply to a 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. Kiselyov (2009) also presents a combinator library which can handle left recursion. Users of the library are required to annotate left recursive grammars with something resembling a coinductive delay constructor. If this constructor is used incorrectly, then parsing can terminate with the wrong answer.

Baars et al. (2009) represent context-free grammars, including semantic actions, in a well-typed way. In order to avoid problems with left recursion when generating top-down parsers from the grammars they implement a left-corner transformation. Neither correctness of the transformation nor termination of the generated parsers is proved formally. Brink et al. (2010) perform a similar exercise, giving a partial proof of correctness, but no proof of termination.

In Section 4.5 it is shown that the parser combinators are as expressive as possible—every parser which can be implemented using the host language can also be implemented 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 Sections 3.3 and 4.2 Brzozowski derivative operators (Brzozowski 1964) are implemented for recognisers and parsers, and in Sections 3.4 and 4.3 these operators are used to characterise recogniser and parser equivalence coinductively. Rutten (1998) performs similar tasks for regular expressions.

[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 words such as "machine-checked" 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.

2. Induction and coinduction

The parser combinators defined in Sections 3 and 4 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
```

This definition should be read inductively, i.e. all lists have finite length. Functions with underscore in their names are operators; _ marks the argument positions. For instance, the constructor _::_ is an infix operator. *Set* is a type of small types.

Coinduction can be used to define types where some elements have infinite depth. Consider the type of potentially infinite lists (colists), for instance:

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

(Note that constructors can be overloaded.) The type function $\infty : \mathit{Set} \to \mathit{Set}$ marks its argument as being coinductive. It is similar to the suspension type constructors which are used to implement non-strictness in strict languages (Wadler et al. 1998). Just as the suspension type constructors the function ∞ comes with delay and force functions, here called $\sharp\_$ (sharp) and $\flat$ (flat):

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

Sharp is a tightly binding prefix operator; ordinary function application binds tighter, though. (Flat is an ordinary function.) Note that $\{A : \mathit{Set}\} \to T$ is a *dependent* function space; the argument A is in scope in T. Arguments in braces, {...}, 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 colist 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 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 (Coquand 1994). As an example of the latter, consider the following definition of *map* for colists:

```
map : ∀ {A B} → (A → B) → Colist A → Colist B
map f []       = []
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 to *map* is *guarded*: it occurs under the delay constructor $\sharp\_$, without any non-constructor function application between the left-hand side and the corecursive call. It is easy to convince oneself that, if the input colist is productively computable, then the (spine of the) output colist 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" (Carlsson and Hallgren 1998; Hancock et al. 2009), taking colists of As to colists 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
  done :                    SP A B
```

The recursive argument of get is inductive, while the recursive argument of put is coinductive. The type should be read as the nested fixpoint $\nu X.\mu Y.\ (A \to Y) + B \times X + 1$, with an outer greatest fixpoint and an inner least fixpoint.[3] This means that a stream processor can only read (get) a finite number of elements from the input before having to produce (put) some output or terminate (done). As a simple example of a stream processor, consider *copy*, which copies its input to its output:

```
copy : ∀ {A} → SP A A
copy = get (λ a → put a (♯ copy))
```

Note that *copy* is guarded (lambdas do not affect guardedness).

The semantics of stream processors can be defined as follows:

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

(⟦_⟧ is a *mixfix* operator.) In the case of get one element from the input colist is consumed (if possible), 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.
- In the first clause the corecursive call is not guarded, but it "preserves guardedness": it takes place under zero occurrences of $\sharp\_$ rather than at least one (and there are no destructors involved). Furthermore the stream processor argument is structurally smaller: $f\ x$ is strictly smaller than get f for any x.

This ensures the productivity of the resulting colist: the next output element can always be computed in finite time, because the number of get constructors between any two put constructors must be finite. Agda accepts definitions which use this kind of lexicographic combination of guarded corecursion and structural recursion. For more information about Agda's criterion for accepting a program as total, and more examples of the use of mixed induction and coinduction in Agda, see Danielsson and Altenkirch (2010).

It may be interesting to observe what would happen if get were made coinductive. In this case we could define more stream processors, for instance the following one:

```
sink : ∀ {A B} → SP A B
sink = get (λ _ → ♯ sink)
```

On the other hand we could no longer define ⟦_⟧ as above (suitably modified), because the output of ⟦ *sink* ⟧ *as* would not be productive for infinite colists *as*. In other words, if we make more stream processors definable some functions become impossible to define.

[3] At the time of writing this interpretation is not correct in Agda (Altenkirch and Danielsson 2010), but the differences are irrelevant for this paper.

3. Recognisers

This section defines a small embedded language of parser combinators. To simplify the explanation the parser combinators defined in this section can only handle recognition. Full parser combinators are described in Section 4.

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

Let us first consider whether the combinator arguments should be read inductively or coinductively. An infinite choice cannot be decided (in the absence of extra information), as this is not possible without inspecting every alternative, so choices will be read 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 _·_ will be coinductive iff the right (left) argument is not nullable.

Based on the observations above the type P of parsers (recognisers) can now be defined for a given token type *Tok*:

```
mutual
  data P : Bool → Set where
    fail  : P false
    empty : P true
    sat   : (Tok → Bool) → P false
    _|_   : ∀ {n₁ n₂} → P n₁ → P n₂ → P (n₁ ∨ n₂)
    _·_   : ∀ {n₁ n₂} →
            ∞⟨ n₂ ⟩P n₁ → ∞⟨ n₁ ⟩P n₂ → P (n₁ ∧ n₂)

  ∞⟨_⟩P : Bool → Bool → Set
  ∞⟨ false ⟩P n = ∞ (P n)
  ∞⟨ true  ⟩P n =     P n
```

Here P true represents those recognisers which accept the empty string, and P false those which do not: fail and sat do not accept the empty string, while empty 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. The definition of the sequencing operator makes use of the mixfix operator $\infty\langle\_\rangle P$ to express the "conditional coinduction" discussed above: the *left* argument has type $\infty\langle\ n_2\ \rangle P\ n_1$, which means that it is coinductive iff n_2 is false, i.e. iff the *right* argument is not nullable. The right argument's type is symmetric.

The conditionally coinductive type $\infty\langle\_\rangle P$ comes with corresponding conditional delay and force functions:

```
♯? : ∀ {b n} → P n → ∞⟨ b ⟩P n
♯? {b = false} x = ♯ x
♯? {b = true}  x =   x

♭? : ∀ {b n} → ∞⟨ b ⟩P n → P n
♭? {b = false} x = ♭ x
♭? {b = true}  x =   x
```

(Here $\{b = \ldots\}$ is the notation for pattern matching on an implicit argument.) We can also define a function which returns true iff the argument is already forced:

```
forced? : ∀ {b n} → ∞⟨ b ⟩P n → Bool
forced? {b = b} _ = b
```

In addition to the constructors listed above the following constructors are also included in P:

```
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 (see Section 3.1 for an example and 3.2 for its semantics), 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 (the type $n_1 \equiv n_2$ is a type of *proofs* showing that n_1 and n_2 are equal). Both nonempty and cast are definable in terms of the other combinators—in the case of cast the definition is trivial, and nonempty can be defined by recursion over the *inductive* structure of its input—but due to Agda's reliance on guarded corecursion it is convenient to have them available as constructors.

3.1 Examples

Using the definition above it is easy to define recognisers which are both left and right recursive, for instance the following one:

```
left-right : P false
left-right = ♯ left-right · ♯ left-right
```

Given the semantics in Section 3.2 it is easy to show that *left-right* does not accept any string. This means that fail does not necessarily have to be primitive, it could be replaced by *left-right*.

As examples of ill-defined recognisers, consider *bad* and bad_2:

```
bad : P false
bad = bad

bad₂ : P true
bad₂ = bad₂ · bad₂
```

These definitions are rejected by Agda, because they are neither structurally recursive nor guarded. They are not terminating, either: an attempt to evaluate the inductive parts of *bad* or bad_2 would lead to non-termination, because the definitions do not make use of the delay operator ♯_.

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
  _⋆ : P false → P true
  p ⋆ = empty | p +
  _+ : P false → P false
  p + = p · ♯ (p ⋆)
```

(The combinator _|_ binds weaker than the other combinators.) 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.[4] 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 $p \cdot p \star$, which would make the definition unguarded and non-terminating—if $p \star$ were unfolded, then no delay operator would ever be encountered. By using the nonempty combinator one can define a variant of _⋆ which accepts arbitrary argument recognisers:

```
_★ : ∀ {n} → P n → P true
p ★ = nonempty p ⋆
```

For more examples, see Section 4.6.

[4] The call to $p +$ is not guarded in the definition of $p \star$, but all that matters for guardedness is calls from one function to itself. If $p +$ is inlined it is clear that $p \star$ is guarded.

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 _∈_ : ∀ {n} → List Tok → P n → Set where
  ...
```

The semantics is determined by the constructors of _∈_, which are introduced below. The values of type $s \in p$ are *proofs* of language membership; the constructors can be seen as inference rules. 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 fail, so there is no constructor for it in _∈_. The empty string is recognised by empty:

```
empty : [ ] ∈ empty
```

(Recall that constructors can be overloaded.) The singleton [t] is recognised by sat f if $f\ t$ evaluates to true ($T\ b$ is inhabited iff b is true):

```
sat : T (f t) → [ t ] ∈ 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:

```
|-left  : s ∈ p₁ → s ∈ p₁ | p₂
|-right : s ∈ p₂ → s ∈ p₁ | p₂
```

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 $p_1 \cdot p_2$:

```
_·_ : s₁ ∈ ♭? p₁ → s₂ ∈ ♭? p₂ →
      s₁ ++ s₂ ∈ p₁ · p₂
```

If a nonempty string is recognised by p, then it is also recognised by nonempty p (and empty strings are never recognised by nonempty p):

```
nonempty : t :: s ∈ p → t :: s ∈ nonempty p
```

Finally cast preserves the semantics of its recogniser argument:

```
cast : s ∈ p → s ∈ cast 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 (one direction can be proved by induction on the structure of the semantics, and the other by induction on the *inductive* structure of the recogniser; delayed sub-parsers do not need to be forced). Given this result it is easy to decide whether or not [ ] $\in p$; it suffices to inspect the index:

```
nullable? : ∀ {n} (p : P n) → Dec ([ ] ∈ p)
```

Note that the correctness of *nullable*? is stated in its type. An element of *Dec P* is either a proof of P or a proof showing that P is impossible:

```
data Dec (P : Set) : Set where
  yes :   P → Dec P
  no  : ¬ P → Dec P
```

Here logical negation is represented as a function into the empty type: $\neg P = P \to \bot$.

3.3 Backend

Let us now consider how the relation _∈_ 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. (Efficiency is discussed further in Section 4.2.)

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

$$s \in D\ t\ p \quad \Leftrightarrow \quad t :: s \in p.$$

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

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

```
D : ∀ {n} (t : Tok) (p : P n) → P (D-nullable t p)
```

The function *D-nullable* decides whether the derivative accepts the empty string or not. Its extensional behaviour is uniquely constrained by the definition of D; its definition is included in Figure 1.

The derivative operator is implemented as follows. The combinators fail and empty never accept any token, so they both have the derivative fail:

```
D t fail  = fail
D t empty = fail
```

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

```
D t (sat f) with f t
... | true  = empty
... | false = fail
```

(Here the **with** construct is used to pattern match on the result of $f\ t$.) The derivative of a choice is the choice of the derivatives of its arguments:

```
D t (p₁ | p₂) = D t p₁ | D t p₂
```

The derivatives of nonempty p and cast $eq\ p$ are equal to the derivative of p:

```
D t (nonempty p) = D t p
D t (cast eq p)  = D t p
```

The final and most interesting case is sequencing:

```
D t (p₁ · p₂) with forced? p₁ | forced? p₂
... | true  | false =   D t   p₁ ·♯? (♭ p₂)
... | false | false = ♯ D t (♭ p₁) ·♯? (♭ p₂)
... | true  | true  =   D t   p₁ ·♯?   p₂ | D t p₂
... | false | true  = ♯ D t (♭ p₁) ·♯?   p₂ | D t p₂
```

Here we have four cases, depending on the indices of p_1 and p_2:

- In the first two cases the *right* argument is not forced, which implies (given the type of _·_) that the *left* argument is not nullable. This means that the first token accepted by $p_1 \cdot p_2$ (if any) has to be accepted by p_1, so the remainder after accepting this token is the remainder of p_1 followed by p_2.
- In the last two cases p_1 is nullable, which means that the first token could also be accepted by p_2. This is reflected in the presence of an extra choice $D\ t\ p_2$ on the right-hand side.

In all four cases the operator $\sharp^?$ is used to conditionally delay p_2, depending on the nullability index of the derivative of p_1; the implicit argument b to $\sharp^?$ is inferred automatically.

The derivative operator D is total: it is implemented using a lexicographic combination of guarded corecursion and structural recursion (as in Section 2). Note that in the first two sequencing cases p_2 is delayed, but D is not applied recursively to $^\flat\ p_2$ because p_1 is known not to accept the empty string.

```
D-nullable : ∀ {n} → Tok → P n → Bool
D-nullable t fail          = false
D-nullable t empty         = false
D-nullable t (sat f)       = f t
D-nullable t (p₁ | p₂)     = D-nullable t p₁ ∨
                             D-nullable t p₂
D-nullable t (nonempty p)  = D-nullable t p
D-nullable t (cast _ p)    = D-nullable t p
D-nullable t (p₁ · p₂)     with forced? p₁ | forced? p₂
... | true  | false = D-nullable t p₁
... | false | false = false
... | true  | true  = D-nullable t p₁ ∨ D-nullable t p₂
... | false | true  = D-nullable t p₂
```

Figure 1. The index function *D-nullable*.

The index function *D-nullable* uses recursion on the *inductive* structure of the recogniser. Note that *D-nullable* does not force any delayed recogniser (it does not use $^\flat$). 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 *D-nullable* ($^\flat$ p_1) $t \wedge$ false to false when checking the definition of *D*. This problem could be fixed by using an equality proof in the definition of *D*, though.

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

```
D-sound    : ∀ {n s t} {p : P n} → s ∈ D t p → t :: s ∈ p
D-complete : ∀ {n s t} {p : P n} → t :: s ∈ p → s ∈ D t p
```

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 if $s \in p$ is inhabited:

```
_∈?_ : ∀ {n} (s : List Tok) (p : P n) → Dec (s ∈ p)
[]     ∈? p = nullable? p
t :: s ∈? p with s ∈? D t p
... | yes s∈Dtp = yes (D-sound s∈Dtp)
... | no  s∉Dtp = no  (s∉Dtp ∘ D-complete)
```

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 *D* ensures that this is correct. (Note that *s∈Dtp* and *s∉Dtp* are normal variables with descriptive names.)

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. The implementation of *parse* in Section 4.2 returns all possible results.

3.4 Laws

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

```
_≈_ : ∀ {n₁ n₂} → P n₁ → P n₂ → Set
p₁ ≈ p₂ = p₁ ⩽ p₂ × p₂ ⩽ p₁
```

Here $\_\times\_$ is conjunction, and $\_\leqslant\_$ encodes language inclusion:

```
_⩽_ : ∀ {n₁ n₂} → P n₁ → P n₂ → Set
p₁ ⩽ p₂ = ∀ {s} → s ∈ p₁ → s ∈ p₂
```

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 a congruence—i.e. that it 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:

```
data _≈c_ {n₁ n₂} (p₁ : P n₁) (p₂ : P n₂) : Set where
  _::_ : n₁ ≡ n₂ → (∀ t → ∞ (D t p₁ ≈c D t p₂)) →
         p₁ ≈c p₂
```

Two recognisers are equal iff they agree on whether the empty string is accepted, and for every token the respective derivatives are equal (coinductively). Note that the values of this data type are *infinite proofs*[5] witnessing the equivalence of the two parsers. Note also that this equality is a form of bisimilarity: the "transitions" are of the form

$$p \xrightarrow{(n,t)} D\ t\ p,$$

where $p : P\ n$. It is easy to show that $\_\approx\_$ and $\_\approx_c\_$ are equivalent. When proving properties of recognisers one can choose the equality which is most convenient for the task at hand. For an example of a proof using a coinductively defined equality, see Section 4.4.

The type of the sequencing combinator is not quite right if we want to state properties such as associativity, so let us introduce the following variant of it:

```
_⊙_ : ∀ {n₁ n₂} → P n₁ → P n₂ → P (n₁ ∧ n₂)
_⊙_ {n₁ = n₁} p₁ p₂ = ♯? p₁ · ♯? {b = n₁} p₂
```

(Agda does not manage to infer the value of the implicit argument *b*, but we can still give it manually.) Using the combinator $\_\odot\_$ it is easy to prove that the recognisers form an idempotent semiring:

```
p | p          ≈ p      p₁ | p₂          ≈ p₂ | p₁
fail | p       ≈ p      p₁ | (p₂ | p₃)   ≈ (p₁ | p₂) | p₃
p ⊙ empty      ≈ p      p₁ ⊙ (p₂ ⊙ p₃)   ≈ (p₁ ⊙ p₂) ⊙ p₃
empty ⊙ p      ≈ p      p₁ ⊙ (p₂ | p₃)   ≈ p₁ ⊙ p₂ | p₁ ⊙ p₃
fail ⊙ p       ≈ fail   (p₁ | p₂) ⊙ p₃   ≈ p₁ ⊙ p₃ | p₂ ⊙ p₃
p ⊙ fail       ≈ fail
```

It is also easy to show that the order $\_\leqslant\_$ coincides with the natural order of the join-semilattice formed by $\_|\_$:

$$p_1 \leqslant p_2 \quad \Leftrightarrow \quad p_1 \mid p_2 \approx p_2$$

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 $\{\, p_1 \odot (p_2 \mathbin{\hat{}} i) \odot p_3 \mid i \in \mathbb{N} \,\}$, where $p \mathbin{\hat{}} i$ is the i-fold repetition of p:

```
⟨_^_⟩-nullable : Bool → ℕ → Bool
⟨ n ^ zero ⟩-nullable = _
⟨ n ^ suc i ⟩-nullable = _

_^_ : ∀ {n} → P n → (i : ℕ) → P (⟨ n ^ i ⟩-nullable)
p ^ zero  = empty
p ^ suc i = p ⊙ (p ^ i)
```

(Here zero and suc are the two constructors of $\mathbb{N}$. Note that Agda can figure out the right-hand sides of $\langle \_\hat{}\_ \rangle$-*nullable* automatically, given the definition of $\_\hat{}\_$; see Section 4.6.)

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

[5] If *Tok* is non-empty.

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  = empty
accept-if-true false = fail
```

Using this recogniser we can construct the "infinite binary tree" using guarded corecursion:

```
grammar : (f : List Bool → Bool) → P (f [ ])
grammar f = cast (lemma f) (
    ♯? (sat id)    · ♯ grammar (f ∘ _::_ true )
  | ♯? (sat not) · ♯ grammar (f ∘ _::_ false)
  | accept-if-true (f [ ]))
```

Note that sat *id* recognises true, and sat *not* recognises false. The following lemma is also 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 infinite grammar above has a very simple structure: it is LL(1). I suspect that this grammar can be implemented using a number of different parser combinator libraries.

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

```
grammar : (f : List Bool → Bool) → P (f [ ])
grammar f =
    ♯ grammar (λ xs → f (xs ++ [ true ])) · ♯? (sat id)
  | ♯ grammar (λ xs → f (xs ++ [ false ])) · ♯? (sat not)
  | accept-if-true (f [ ])
```

This shows that nonempty and cast are not necessary to achieve full expressive strength, because neither *grammar* nor the backend rely on 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 for the language $\{ nn \mid n \in \mathbb{N} \}$. By generalising the statement to "it is impossible that p accepts infinitely many identical pairs, and only identical pairs and/or the empty string" (where an identical pair is a string of the form nn) one can prove this formally by induction on the structure of p (see the accompanying code). Note that this restriction does not apply to the monadic combinators introduced in the next section, which have maximal expressive strength also for infinite token sets.

4. Parsers

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

Consider the monadic parser combinator bind, _>>=_: The parser p_1 >>= 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 >>= 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 can return without consuming any input can be relevant for determining if another parser is nullable.

This suggests that, in analogy with the treatment of recognisers, a parser should be indexed by its "initial set"—the set of values which can be returned when the input is empty. However, sometimes it is useful to distinguish two grammars if the number of parse trees corresponding to a certain string differ. For instance, the parser backend defined in Section 4.2 returns twice as many results for the parser $p \mid p$ as for the parser p. In order to take account of this distinction parsers are indexed by their return types and their "initial bags" (or multisets), represented as lists:

```
mutual
  data Parser : (R : Set) → List R → Set₁ where
    ...
```

(Set_1 is a type of large types; Agda is predicative.)

The first four combinators have relatively simple types. The return combinator is the parser analogue of empty. When accepting the empty string it returns its argument:

```
return : ∀ {R} (x : R) → Parser R [ x ]
```

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

```
fail : ∀ {R} → Parser R [ ]
```

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

```
token : Parser Tok [ ]
```

This combinator is not as general as sat, but a derived combinator *sat* is easy to define using token and bind, see Section 4.6. The analogue of the choice recogniser is _|_:

```
_|_ : ∀ {R xs₁ xs₂} → Parser R xs₁ → Parser R xs₂ →
      Parser R (xs₁ ++ xs₂)
```

The initial bag of a choice is the union of the initial bags of its two arguments.

The bind combinator's type is more complicated than the types above. Consider p_1 >>= p_2 again. Here p_2 is a function, and we have a *function* f : R_1 → *List* R_2 which computes the initial bag of $p_2\ x$, depending on the value of x. When should we allow p_1 to be coinductive? One option is to only allow this when $f\ x$ is empty for every x, but I do not want to require the user of the library to prove such a property just to define a parser. Instead I have chosen to represent the function f with an optional function f : *Maybe* (R_1 → *List* R_2),[6] where nothing represents $\lambda\ \_ \to [\,]$, and to make p_1 coinductive iff f is nothing. The same approach is used for xs, the initial bag of p_1:

```
_>>=_ : ∀ {R₁ R₂} {xs : Maybe (List R₁)}
          {f : Maybe (R₁ → List R₂)} →
        ∞⟨ f ⟩Parser R₁ (flatten xs) →
        ((x : R₁) → ∞⟨ xs ⟩Parser R₂ (apply f x)) →
        Parser R₂ (bind xs f)
```

The helper functions *flatten*, *apply* and *bind*, which interpret nothing as the empty list or the constant function returning the

[6] The type *Maybe A* has the two constructors nothing : *Maybe A* and just : *A* → *Maybe A*.

```
flatten : {A : Set} → Maybe (List A) → List A
flatten nothing  = []
flatten (just xs) = xs

apply : {A B : Set} → Maybe (A → List B) → A → List B
apply nothing x  = []
apply (just f) x = f x

bind : {A B : Set} →
       Maybe (List A) → Maybe (A → List B) → List B
bind xs nothing  = []
bind xs (just f) = bind_L (flatten xs) f
```

Figure 2. Helper functions used in the type signature of $\_\gg\!=\_$. Note that there is a reason for not defining *bind* using the equation $bind\ xs\ f = bind_L\ (flatten\ xs)\ (apply\ f)$; see Section 4.6.

empty list, are defined in Figure 2; *bind* is defined in terms of $bind_L$, the standard list monad's bind operation. The function $\infty\langle\_\rangle Parser$ is defined as follows, mutually with *Parser*:

```
∞⟨_⟩Parser : {A : Set} → Maybe A →
             (R : Set) → List R → Set₁
∞⟨ nothing ⟩Parser R xs = ∞ (Parser R xs)
∞⟨ just _  ⟩Parser R xs =     Parser R xs
```

(∞ works also for Set_1.) It is straightforward to define a variant of $\flat^?$ for this type. It is not necessary to define $\sharp^?$, though: instead of conditionally delaying one can just avoid using nothing.

Just as in Section 3 two additional constructors are included in the definition of *Parser*:

```
nonempty : ∀ {R xs} → Parser R xs → Parser R []
cast     : ∀ {R xs₁ xs₂} → xs₁ ≈bag xs₂ →
           Parser R xs₁ → Parser R xs₂
```

Here $\_\approx_{bag}\_$ stands for *bag equality* between lists, equality up to permutation of elements; the cast combinator ensures that one can replace one representation of a parser's initial bag with another. Bag equality is defined in two steps. First list membership is encoded inductively as follows:

```
data _∈_ {A : Set} : A → List A → Set where
  here  : ∀ {x xs}     → x ∈ x :: xs
  there : ∀ {x y xs} → y ∈ xs → y ∈ x :: xs
```

Two lists *xs* and *ys* are then deemed "bag equal" if, for every value *x*, *x* is a member of *xs* as often as it is a member of *ys*:

```
_≈bag_ : ∀ {R} → List R → List R → Set
xs ≈bag ys = ∀ {x} → x ∈ xs ↔ x ∈ ys
```

Here $A \leftrightarrow B$ means that there is an invertible function from *A* to *B*, so *A* and *B* must have the same cardinality.

4.1 Semantics

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 3. Note that values of type $x \in p \cdot s$ can be seen as parse trees.

The parsers come with two kinds of equivalence. The weaker one, *language equivalence* ($\_\approx\_$), is a direct analogue of the equivalence used for recognisers in Section 3.4:

```
_≈_ : ∀ {R xs₁ xs₂} →
      Parser R xs₁ → Parser R xs₂ → Set₁
p₁ ≈ p₂ = ∀ {x s} → x ∈ p₁ · s ↔ x ∈ p₂ · s
```

```
data _∈_·_ : ∀ {R xs} →
             R → Parser R xs → List Tok → Set₁ where
  return    : x ∈ return x · []
  token     : t ∈ token · [ t ]
  ∣-left    : x ∈ p₁ · s → x ∈ p₁ ∣ p₂ · s
  ∣-right   : x ∈ p₂ · s → x ∈ p₁ ∣ p₂ · s
  _⟫=_      : x ∈ ♭? p₁ · s₁ → y ∈ ♭? (p₂ x) · s₂ →
              y ∈ p₁ ⟫= p₂ · s₁ ++ s₂
  nonempty  : x ∈ p · t :: s → x ∈ nonempty p · t :: s
  cast      : x ∈ p · s → x ∈ cast eq p · s
```

Figure 3. The semantics of the parser combinators. To avoid clutter the declarations of bound variables are omitted in the constructors' type signatures.

Here $A \Leftrightarrow B$ means that *A* and *B* are equivalent: there is a function of type $A \to B$ and another function of type $B \to A$. We immediately get that language equivalence is an equivalence relation.

As mentioned above language equivalence is sometimes too weak. We may want to distinguish between grammars which define the same language, if they do not agree on the number of ways in which a given value can be produced from a given string. To make the example given above more concrete, the parser backend defined in Section 4.2 returns one result when the empty string is parsed using return true (parse tree: return), and two results when return true ∣ return true is used (parse trees: ∣-left return and ∣-right return). Based on this observation two parsers are defined to be *parser equivalent* ($\_\cong\_$) if, for all values and strings, the respective sets of parse trees have the same cardinality:

```
_≅_ : ∀ {R xs₁ xs₂} →
      Parser R xs₁ → Parser R xs₂ → Set₁
p₁ ≅ p₂ = ∀ {x s} → x ∈ p₁ · s ↔ x ∈ p₂ · s
```

From its definition we immediately get that parser equivalence is an equivalence relation. Parser equivalence is strictly stronger than language equivalence: the former distinguishes between return true and return true ∣ return true, while the latter is idempotent.

Just as in Section 3.2 the initial bag index is correct:

```
index-correct : ∀ {R xs x} {p : Parser R xs} →
                x ∈ p · [] ↔ x ∈ xs
```

Note the use of $\_\leftrightarrow\_$: the number of parse trees for *x* matches the number of occurrences of *x* in the list *xs*. One direction of the inverse can be defined by recursion on the structure of the semantics, and the other by recursion on the structure of $\_\in\_$.

From *index-correct* we easily get that parsers which are parser equivalent have equal initial bags:

```
same-bag : ∀ {R xs₁ xs₂}
             {p₁ : Parser R xs₁} {p₂ : Parser R xs₂} →
           p₁ ≅ p₂ → xs₁ ≈bag xs₂
```

Similarly, language equivalent parsers have equal initial *sets*.

4.2 Backend

Following Section 3.3 it is easy to implement a derivative operator for parsers:

```
D : ∀ {R xs} (t : Tok) (p : Parser R xs) →
    Parser R (D-bag t p)
```

The implementation of the function *D-bag* which computes the derivative's initial bag can be seen in Figure 4. Both *D* and *D-bag* use analogues of the *forced*? function from Section 3:

```
D-bag : ∀ {R xs} → Tok → Parser R xs → List R
D-bag t (return x)      = [ ]
D-bag t fail            = [ ]
D-bag t token           = [ t ]
D-bag t (p₁ | p₂)       = D-bag t p₁ ++ D-bag t p₂
D-bag t (nonempty p)    = D-bag t p
D-bag t (cast eq p)     = D-bag t p
D-bag t (p₁ >>= p₂)     with forced? p₁ | forced?′ p₂
... | just f  | nothing = bind_L (D-bag t p₁) f
... | just f  | just xs = bind_L (D-bag t p₁) f ++
                         bind_L xs (λ x → D-bag t (p₂ x))
... | nothing | nothing = [ ]
... | nothing | just xs = bind_L xs (λ x → D-bag t (p₂ x))
```

Figure 4. The index function *D-bag*. Note that its implementation falls out almost automatically from the definition of *D*.

```
forced? : ∀ {A R xs m} → ∞⟨ m ⟩Parser R xs → Maybe A
forced? {m = m} _ = m

forced?′ : ∀ {A R₁ R₂ : Set} {m} {f : R₁ → List R₂} →
           ((x : R₁) → ∞⟨ m ⟩Parser R₂ (f x)) → Maybe A
forced?′ {m = m} _ = m
```

The non-recursive cases of *D*, along with choice, nonempty and cast, are easy:

```
D t (return x)      = fail
D t fail            = fail
D t token           = return t
D t (p₁ | p₂)       = D t p₁ | D t p₂
D t (nonempty p)    = D t p
D t (cast eq p)     = D t p
```

The last case, $\_\gg\!=\_$, is more interesting. It makes use of the combinator *return⋆*, which can return any element of its argument list:

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

The code is very similar to the code for sequencing in Section 3.3:

```
D t (p₁ >>= p₂) with forced? p₁ | forced?′ p₂
... | just f  | nothing = D t p₁ >>= (λ x → ♭ (p₂ x))
... | nothing | nothing = ♯ D t (♭ p₁) >>= (λ x → ♭ (p₂ x))
... | just f  | just xs = D t p₁ >>= (λ x → p₂ x)
                       | return⋆ xs >>= (λ x → D t (p₂ x))
... | nothing | just xs = ♯ D t (♭ p₁) >>= (λ x → p₂ x)
                       | return⋆ xs >>= (λ x → D t (p₂ x))
```

There are two main differences. One is the absence of $\sharp^?$. The other difference can be seen in the last two cases, where p_1 is potentially nullable (it is if *xs* is nonempty). The corresponding right-hand sides are implemented as choices, as before. However, the right choices are a bit more involved than in Section 3.3. They correspond to the cases where p_1 succeeds without consuming any input, returning one of the elements of its initial bag *xs*. In this case the elements of the initial bag index of p_1 are returned using *return⋆*, and then combined with p_2 using bind.

The implementation of *D-bag* is structurally recursive, while the implementation of *D* uses a lexicographic combination of guarded corecursion and structural recursion, just as in Section 3.3. It is straightforward to prove the following correctness property:

```
D-correct : ∀ {R xs x s t} {p : Parser R xs} →
            x ∈ D t p · s ↔ x ∈ p · t :: s
```

Both directions of the inverse can be defined by recursion on the structure of the semantics, with the help of *index-correct*.

Given the derivative operator it is easy to define the parser backend:

```
parse : ∀ {R xs} → Parser R xs → List Tok → List R
parse {xs = xs} p [ ]       = xs
parse           p (t :: s)  = parse (D t p) s
```

The correctness of this implementation follows easily from *index-correct* and *D-correct*:

```
parse-correct : ∀ {R xs x s} {p : Parser R xs} →
                x ∈ p · s ↔ x ∈ parse p s
```

Both directions of the inverse can be defined by recursion on the structure of the input string. Note that this proof establishes that a parser can only return a finite number of results for a given input string (because the list returned by *parse* is finite)—infinitely ambiguous grammars cannot be represented in this framework.

As mentioned in Section 4.1 we have

```
parse (return true | return true) [ ] ≡ true :: true :: [ ] .
```

It might seem reasonable for *parse* to remove duplicates from the list of results. However, the result type is not guaranteed to come with decidable equality (consider functions, for instance), so such filtering is left to the user of *parse*.

The code above is not optimised, and mainly serves to illustrate that it is *possible* to implement a *Parser* backend which guarantees termination. It is not too hard to see that, in the worst case, *parse* is *at least* exponential in the size of the input string. Consider the following parser:

```
p : Parser Bool [ ]
p = fail >>= λ (b : Bool) → fail
```

The derivative $D\ t\ p$ is $p \mid p$, for any token *t*. After taking *n* derivatives we get a parser with $2^n - 1$ choices, and all these choices have to be traversed to compute the parser's initial bag. The parser *p* may seem contrived, but similar parsers can easily arise as the result of taking the derivative of more useful parsers.

It may be possible to implement more efficient backends. For instance, one can make use of algebraic laws like $\mathsf{fail} \gg\!= p \cong \mathsf{fail}$ (see Section 4.4) to simplify parsers, and perhaps avoid the kind of behaviour described above, at least for certain classes of parsers. Exploring such optimisations is left for future work, though.

4.3 Coinductive equivalences

In Section 3.4 a coinductive characterisation of recogniser equivalence is given. This is possible also for parser equivalence:

```
data _≅c_ {R xs₁ xs₂} (p₁ : Parser R xs₁)
                      (p₂ : Parser R xs₂) : Set where
  _::_ : xs₁ ≈bag xs₂ →
         (∀ t → ∞ (D t p₁ ≅c D t p₂)) →
         p₁ ≅c p₂
```

Two parsers are equivalent if their initial bags are equal, and, for every token *t*, the respective derivatives with respect to *t* are equivalent (coinductively). Using *index-correct* and *D-correct* it is easy to show that the two definitions of parser equivalence, $\_\cong\_$ and $\_\cong_c\_$, are equivalent.

By replacing the use of $\_\leftrightarrow\_$ in the definition of bag equality with $\_\Leftrightarrow\_$ we get set equality instead. If, in turn, the use of bag equality is replaced by set equality in $\_\cong_c\_$, then we get a coinductive characterisation of language equivalence ($\_\approx\_$).

By using the coinductive characterisations of equivalence I have proved that all primitive parser combinators preserve both language and parser equivalence, i.e. the equivalences are congruences.

4.4 Laws

Let us now discuss the equational theory of the parser combinators. Many of the laws from Section 3.4 can be generalised to the setting of parser combinators. To start with we have a commutative monoid formed by fail and _|_:

$$\begin{aligned} p_1 \mid p_2 &\cong p_2 \mid p_1 \\ \mathsf{fail} \mid p &\cong p \\ (p_1 \mid p_2) \mid p_3 &\cong p_1 \mid (p_2 \mid p_3) \end{aligned}$$

If language equivalence is used this monoid is also idempotent:

$$p \mid p \approx p$$

We also have a monad, with fail as a left and right zero of bind, and bind distributing from the left and right over choice:

$$\begin{aligned} \mathsf{return}\ x \gg\!= p &\cong p\ x \\ p \gg\!= \mathsf{return} &\cong p \\ p_1 \gg\!= (\lambda\, x \to p_2\ x \gg\!= p_3) &\cong (p_1 \gg\!= p_2) \gg\!= p_3 \\ \mathsf{fail} \gg\!= p &\cong \mathsf{fail} \\ p \gg\!= (\lambda\ \_ \to \mathsf{fail}) &\cong \mathsf{fail} \\ p_1 \gg\!= (\lambda\, x \to p_2\ x \mid p_3\ x) &\cong p_1 \gg\!= p_2 \mid p_1 \gg\!= p_3 \\ (p_1 \mid p_2) \gg\!= p_3 &\cong p_1 \gg\!= p_3 \mid p_2 \gg\!= p_3 \end{aligned}$$

Unlike in Section 3.4 there is no need to define a special variant of _⩾=_ to state the laws above: if the types of the argument parsers are given (as for *⩾=-left-identity* below), then Agda automatically infers that bind's implicit arguments *xs* and *f* should have the form just *something*.

Analogues of most of the laws from Section 3.4 are listed above. However, assuming that the token type is inhabited, it is not possible to find a function

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

and a Kleene-star-like combinator

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

such that

$$\begin{aligned} &\mathsf{return}\ [\,] \mid (p \gg\!= \lambda\, x \to p \bigstar \gg\!= \lambda\, xs \to \mathsf{return}\ (x :: xs)) \\ &\quad \leqslant p \bigstar \end{aligned}$$

holds for all p. (Here _⩽_ 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 *parse-correct* shows that a parser can only return a finite number of results. (A combinator _★ satisfying the inequality above can easily be implemented if it is restricted to non-nullable argument parsers.)

Before leaving the subject of equational laws, let me take a moment to explain how one of the laws above—the left identity law for bind—can be proved. Assume that we have already proved some of the other laws, along with the following property of bind_L:

```
bindL-left-identity :
  {A B : Set} (x : A) (f : A → List B) →
  bindL [ x ] f ≈bag f x
```

I have found the coinductive characterisations of the equivalences to be convenient to work with, so I have proved the law roughly as follows:

```
⩾=-left-identity :
  {R1 R2 : Set} {f : R1 → List R2}
  (x : R1) (p : (x : R1) → Parser R2 (f x)) →
  return x ⩾= p ≅c p x
⩾=-left-identity {f = f} x p =
  bindL-left-identity x f :: λ t → ♯ (
    D t (return x ⩾= p)                                  ≅c
    fail ⩾= p | return⋆ [ x ]  ⩾= (λ x → D t (p x))      ≅c
    fail      | return x       ⩾= (λ x → D t (p x))      ≅c
                return x       ⩾= (λ x → D t (p x))      ≅c
    D t (p x)                                            □)
```

(To avoid clutter the proof above uses the equational reasoning notation ... ≅c ... ≅c ... □, and the sub-proofs for the individual steps have been omitted.) The proof has two parts. First bind_L*-left-identity* is used to show that the initial bags of return *x* ⩾= *p* and *p x* are equal, and then it is shown, for every token *t*, that *D t* (return *x* ⩾= *p*) and *D t* (*p x*) are equivalent. The first step of the latter part uses a law relating *D* and _⩾=_, the second step uses the left zero law (fail ⩾= *p* ≅c fail) and the right identity law for choice (*p* | fail ≅c *p*), the third step uses the left identity law for choice (fail | *p* ≅c *p*), and the last step uses the coinductive hypothesis.

The proof as written above would not be accepted by Agda, because the coinductive hypothesis is not guarded by constructors (due to the uses of transitivity implicit in the equational reasoning notation). However, this issue can be addressed (Danielsson 2010). For details of how all the properties above have been proved, see the code accompanying the paper.

4.5 Expressive strength

This subsection is concerned with the parser combinators' expressiveness. 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 bag equality is used for the lists of results). The grammar is similar to 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 [ ])
```

The function *grammar* satisfies the following correctness property:

```
grammar-correct : ∀ {R x s} (f : List Tok → List R) →
                  x ∈ grammar f · s ↔ x ∈ f s
```

One direction of the inverse can be defined by induction on the structure of the semantics, and the other by induction on the structure of the input string. If we combine this result with *parse-correct* we get the expressiveness result:

```
maximally-expressive : ∀ {R} (f : List Tok → List R) {s} →
                       parse (grammar f) s ≈bag f s
```

Assume for a moment that the primitive parser combinators included sat and applicative functor application (McBride and Paterson 2008) instead of token and bind. Then, for finite sets of tokens, we could have defined *grammar* roughly as in Section 3.5. This means that, for finite sets of tokens, the inclusion of the monadic bind combinator does not provide any expressive advantage; the applicative functor interface is already sufficiently expressive. This comparison does not take efficiency into account, though.

4.6 Examples

Finally let us consider some examples, along with some practical remarks.

Let us start with the left recursive grammar in the introduction. Note that it does not require any user annotations, except for the three uses of $\sharp\_$. Agda infers all the type signatures and all the implicit arguments, including several functions, automatically. Agda's inference mechanism is based on unification (a variant of pattern unification (Pfenning 1991)), and an omitted piece of code is only "filled in" if it can be *uniquely* determined from the constraints provided by the rest of the code. In general there is no guarantee that implicit arguments can be omitted, and it is not uncommon that the exact form of a definition affects how much can be inferred.

Consider the definition of *bind* in Figure 2. It is set up so that *bind xs* nothing evaluates to the empty list, *even if xs is a neutral term*. If *bind* had instead been defined by the equation

$$bind\ xs\ f\ =\ bind_L\ (flatten\ xs)\ (apply\ f),$$

then the example in the introduction would have required manual annotations: the example gives rise to the constraint $xs = bind$ (just xs) nothing, which with the alternative definition of *bind* reduces to $xs = bind_L\ xs\ (\lambda\ \_ \to [\,])$, and Agda cannot solve this unification problem.

As an example of a definition for which the initial bag is not inferred automatically, consider the following definition of *sat*:

```
sat : ∀ {R} → (Tok → Maybe R) → Parser R _
sat {R = R} p = token >>= λ t → ok (p t)
  where
  ok-bag : Maybe R → List R
  ok-bag nothing = _
  ok-bag (just x) = _

  ok : (x : Maybe R) → Parser R (ok-bag x)
  ok nothing = fail
  ok (just x) = return x
```

The parser *sat p* matches a single token *t* iff *p t* evaluates to just *x*, for some *x*; the value returned is *x*. The initial bag function *ok-bag* is not inferred by Agda. However, the right-hand sides of *ok-bag*, and the initial bag of *sat*, are inferred.

The example in the introduction uses the derived combinators *tok* and *number*. The parser *tok*, which accepts a given token, is easy to define using *sat* (assuming that equality of tokens can be decided using the function _==_):

```
tok : Tok → Parser Tok _
tok t = sat (λ t′ → if t == t′ then just t′ else nothing)
```

Given a parser for digits (which is easy to define using *sat*) the parser *number*, which accepts an arbitrary non-negative number, can also be defined:

```
number : Parser ℕ _
number = digit + >>= return ∘ foldl (λ n d → 10 * n + d) 0
```

Here *foldl* is a left fold for lists, and $p+$ parses one or more *p*s (as in Section 3.1).

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

5. Conclusions

A parser combinator library which handles left recursion and guarantees termination of parsing has been presented, and it has been established that the library is sufficiently expressive: every finitely ambiguous parser on finite input strings which can be implemented using the host language can also be realised using the combinators.

I believe that the precise treatment of induction and coinduction which underlies the definition of the parser combinators gives a good framework for understanding lazy programs. To take one example, Claessen (2004) defines the following parser data type using Haskell:

```
data P′ s a = SymbolBind (s → P′ s a)
            | Fail
            | ReturnPlus a (P′ s a)
```

He notes that it is isomorphic to the stream processor type used in Fudgets (Carlsson and Hallgren 1998), and that this isomorphism "inspired the view of the parser combinators being parsing process combinators". However, in a total setting I would define these two types differently. The stream processors were defined in Section 2, with an inductive get constructor and a coinductive put constructor. I find it natural to define P' in the *opposite* way:

```
data P′ (S A : Set) : Set where
  symbolBind : (S → ∞ (P′ S A)) → P′ S A
  fail       : P′ S A
  returnPlus : A → P′ S A → P′ S A
```

The reason for the difference is that the types are used differently. Stream processors are interpreted using $[\![\_]\!]$, and parsers using *parse′*, which works with *finite* lists:

```
parse′ : ∀ {S A} → P′ S A → List S → List (A × List S)
parse′ (symbolBind f) (c :: s) = parse′ (♭ (f c)) s
parse′ (returnPlus x p) s      = (x, s) :: parse′ p s
parse′ _                _      = []
```

The definition of $[\![\_]\!]$ in Section 2 would not be total if get were coinductive, because then we could not guarantee that the resulting colist would be productive. On the other hand, if returnPlus were coinductive and symbolBind inductive, then parsers like the one used in the proof of maximal expressiveness in Section 4.5 could not be implemented (consider the case when the argument to *grammar* is $\lambda\ \_ \to [\,]$).

The use of lazy data types and general recursion in Haskell is very flexible—for instance, Carlsson and Hallgren (1998) use their stream processors in ways which would not be accepted if the type *SP* were used in Agda—but I find it easier to understand how and why programs work when induction and coinduction are separated as in this paper. The use of *mixed* induction and coinduction has been known for a long time (Park 1980), but does not seem to be well-known among functional programmers. It is my hope that this paper provides a compelling example of the use of this technique.

Acknowledgements

I would like to thank Ulf Norell for previous joint work on total parsing, and for improving Agda's unification mechanism. Another person who deserves thanks is Thorsten Altenkirch, with whom I have had many discussions about mixed induction and coinduction. Thorsten also suggested that I should allow left recursive parsers, which I might otherwise not have tried, and gave feedback which improved the presentation; such feedback was also given by several anonymous reviewers.

Finally I would like to acknowledge financial support from EPSRC and the Royal Swedish Academy of Sciences' funds (EPSRC grant code: EP/E04350X/1).

References

The Agda Team. The Agda Wiki. Available at `http://wiki.portal.chalmers.se/agda/`, 2010.

Thorsten Altenkirch and Nils Anders Danielsson. Termination checking in the presence of nested inductive and coinductive types. Note supporting presentation given at the Workshop on Partiality and Recursion in Interactive Theorem Provers, Edinburgh, UK, 2010.

Arthur Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed grammars: The left corner transform. In *Preliminary Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications, LDTA 2009*, pages 18–33, 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.

Kasper Brink, Stefan Holdermans, and Andres Löh. Dependently typed grammars. In *Mathematics of Program Construction, Tenth International Conference, MPC 2010*, volume 6120 of *LNCS*, pages 58–79, 2010.

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.

Magnus Carlsson and Thomas Hallgren. *Fudgets – Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Chalmers University of Technology and Göteborg University, 1998.

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.

Thierry Coquand. Infinite objects in type theory. In *Types for Proofs and Programs, International Workshop TYPES '93*, volume 806 of *LNCS*, pages 62–78, 1994.

Nils Anders Danielsson. Beating the productivity checker using embedded languages. In *Workshop on Partiality and Recursion in Interactive Theorem Provers, Edinburgh, UK*, 2010.

Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively: An exercise in mixed induction and coinduction. In *Mathematics of Program Construction, Tenth International Conference, MPC 2010*, volume 6120 of *LNCS*, pages 100–118, 2010.

Nils Anders Danielsson and Ulf Norell. Structurally recursive descent parsing. Unpublished note, 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.

Oleg Kiselyov. Parsec-like parser combinator that handles left recursion? Message to the Haskell-Cafe mailing list, December 2009.

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

Adam Koprowski and Henri Binsztok. TRX: A formally verified parser interpreter. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010*, volume 6012 of *LNCS*, pages 345–365, 2010.

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.

David Park. On the semantics of fair parallelism. In *Abstract Software Specifications*, volume 86 of *LNCS*, pages 504–526, 1980.

Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, 1991.

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.