Structurally Recursive Descent Parsing (Draft)

Nils Anders Danielsson[*]
University of Nottingham

Ulf Norell
Chalmers University of Technology

December 29, 2008

Abstract

Recursive descent parsing does not terminate for left recursive grammars. We turn recursive descent parsing into structurally recursive descent parsing, acceptable by total dependently typed languages like Agda, by using the type system to rule out left recursion.

The resulting library retains much of the flavour of ordinary "list of successes" combinator parsers. In particular, the type indices used to rule out left recursion can in many cases be inferred automatically, so that library users do not have to write them out manually.

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. Because these specifications are directly executable the method is also light-weight: no separate tools are necessary, only the standard tool-chain of the host language. Furthermore parser combinator libraries are often embedded in an advanced programming language, in which case their users get the added advantage that it is easy to abstract over and reuse recurring grammatical or parsing patterns; this may be the most important advantage of parser combinator libraries, as opposed to standard parser generators. Finally parser combinators are nowadays often fast enough to be used in practice.

There is a problem with the description above, though. All parsers written using parser combinator libraries are not executable, at least not in a satisfying way. An attempt to parse something using a *left recursive* grammar often leads to non-termination. This problem can be fixed by using a more sophisticated

[*]The author would like to thank EPSRC and the Royal Swedish Academy of Sciences' funds for financial support.

parser backend which can handle left recursion (Frost et al. 2008), or it can be fixed by ruling out left recursive grammars. We take the second approach.

We use dependent types to add information in the form of type indices to the parser type, and use these indices to ensure that left recursion is not possible. The information is

1. whether or not the parser accepts the empty string, and
2. an approximation of the parser's *proper left corners*, i.e. the parsers (non-terminals) which the parser can invoke without first consuming any input (see Section 2).

A parser is left recursive iff it is a proper left corner of itself. Many parser combinator libraries are based on *recursive descent parsing* (Burge 1975), which fails to terminate if an attempt is made to descend on a left-recursive non-terminal, since this leads to an infinite loop. We make use of the structure of the proper left corner representation to turn recursive descent parsing into *structurally* recursive descent parsing, thereby ensuring termination (see Section 4).

Dependent types are in general hard to infer automatically, and the exercise outlined above risks being of mostly academic value if the programmer using the resulting library has to construct and maintain tricky type annotations. However, this is not the case. For many uses of our library the type indices can be inferred automatically. Sections 5 and 6 illustrate this through a series of examples.

Taking a step back one can note that parser combinator libraries provide good examples of domain-specific embedded languages (Hudak 1996), for all the reasons outlined above. In this setting the failure to rule out left recursion amounts to a lack of static checking for these languages. By using a host language with dependent types we are able to capture more domain-specific constraints in the embedded type system, thereby providing a more faithful embedding of the domain-specific language (Oury and Swierstra 2008).

To summarise, our contributions are as follows:

- We show how parser combinators which guarantee[1] termination can be implemented and used.
- Dependent types are used to rule out left recursion, but the extra type information can often be inferred automatically.

To keep things concrete the total, dependently typed functional language Agda (Norell 2007; Agda Team 2008) is used to explain our methods; the language is introduced as we go along. A parser combinator library implemented using these methods is (at the time of writing) available from the first author's web page.

The rest of the paper is structured as follows: Section 2 explains left recursion in more detail. Section 3 defines an embedded language of parser combinators coinductively, Section 4 shows how this language can be interpreted, and Section 5 outlines some differences between these and ordinary parser combinators through a series of examples. As an alternative or complement to the coinductive approach Section 6 introduces a grammar-based variant of total parsing, and Section 7 concludes with a discussion of related work.

[1]Assuming that the host language can be proved total and its implementation is bug-free. Similar assumptions apply to other statements below.

2 Left recursion

Parsers based on recursive descent or similar top-down, left-to-right techniques fail to terminate for left recursive grammars. Let us illustrate this through an example, expressed using a monadic parser combinator interface based on Hutton and Meijer's presentation (1998). In a language like Haskell the following simple expression parser is unproblematic:

```
expr = do
  l ← term
  theToken '+'
  r ← expr
  return (Add l r)
term = ...
```

(Here *theToken* t is a parser for the token t.) However, if the parser is changed to

```
expr = do
  l ← expr
  theToken '+'
  r ← term
  return (Add l r)
term = ...
```

in order to make addition left associative, then parsing fails to terminate. The reason is that *expr*, when evaluated, immediately descends on itself, without consuming any input tokens. In this case it is easy to see that *expr* is left recursive, but in the case of *indirect* left recursion, where a parser descends on itself after several steps without having consumed any tokens, it can be harder.

In order to define left recursion more precisely, consider a non-terminal N in a context-free grammar. If N can be rewritten, in one or more steps, to a sequence of terminals and non-terminals headed by X, then X is a *proper left corner* of N (this definition matches the one given by Moore (2000) if empty productions are disallowed). A non-terminal is left recursive if it is a proper left corner of itself. Even though monadic combinator parsing is not restricted to context-free grammars we reuse this terminology for parsers: The parsers that a given parser can evaluate to in at least one step without consuming any input are called its proper left corners; a parser is left recursive if it is "at the head" of a proper left corner of itself.

In Section 3 below we will make use of proper left corners to rule out left recursion, but first we note a second problem with the definitions of *expr* above: Agda, the language used for our implementation, would reject *both* of them, also the unproblematic one. The reason is that the definitions are cyclic; there is nothing which gets obviously smaller in the recursive calls to *expr*, so Agda's termination checker would not accept the given code.

There are at least two ways to work around this problem. One is to view a parser as a possibly infinite value of a coinductive type, and ensure that parser definitions are *productive*. However, the productivity checker of Agda currently only accepts guarded corecursion (Coquand 1994), which can be somewhat restrictive.

Another approach is to avoid the cyclic definitions altogether by representing grammars explicitly in the form of functions from non-terminals to parsers. This approach is more heavyweight, but does not require language support for coinductive types, and avoids issues with productivity. (It would perhaps be incorrect to apply the term parser *combinators* to this approach, though.)

We will use both approaches above, in an attempt to make the resulting library as easy to use as possible, starting with the coinductive approach in Section 3.

3 Coinductive parser combinators

We will aim for a standard monadic parser combinator interface, like the following one (Claessen 2004), where *Parser Token R* is the type of parsers parsing *Token* strings and returning values of type *R*:

$$\begin{array}{lll}
\mathit{return} & : & \forall\, \{T\ R\} \to R \to \mathit{Parser}\ T\ R \\
\_{\gg\!=}\_ & : & \forall\, \{T\ R_1\ R_2\} \to \\
 & & \mathit{Parser}\ T\ R_1 \to (R_1 \to \mathit{Parser}\ T\ R_2) \to \mathit{Parser}\ T\ R_2 \\
\mathit{fail} & : & \forall\, \{T\ R\} \to \mathit{Parser}\ T\ R \\
\_|\_ & : & \forall\, \{T\ R\} \to \mathit{Parser}\ T\ R \to \mathit{Parser}\ T\ R \to \mathit{Parser}\ T\ R \\
\mathit{token} & : & \forall\, \{T\} \to \mathit{Parser}\ T\ T \\
\mathit{parse} & : & \forall\, \{T\ R\} \to \mathit{Parser}\ T\ R \to \mathit{List}\ T \to \mathit{List}\ (R \times \mathit{List}\ T)
\end{array}$$

Here *return* and $\_{\gg\!=}\_$ (bind) are the monadic combinators, $\_|\_$ is symmetric choice with *fail* as unit, *token* returns the next token (if any), and *parse* applies a parser to a (finite) list of tokens, returning a list containing all possible parses, paired up with the corresponding remaining input. (Arguments enclosed in $\{\ldots\}$ are *implicit*; they do not need to be given explicitly if Agda can infer them. Names containing underscores (_) are mixfix operators; the underscores stand for argument positions.)

This section defines our variant of the *Parser* type, and Section 4 shows how *parse* can be defined. In order to ensure totality the interface defined below is more restrictive than the one above, though.

3.1 Corners

As discussed in Section 2 we want to make sure that a parser is not a proper left corner (henceforth simply *corner*) of itself. Consider the expression $p_1 \gg p_2$, where $p_1 \gg p_2 = p_1 \gg\!= \lambda\,\_ \to p_2$, and assume that c_1 and c_2 are the sets of corners of p_1 and p_2, respectively. The set of corners of $p_1 \gg p_2$ depends on whether or not p_1 accepts the empty string. If it does, then the set of corners is $c_1 \cup c_2$, and otherwise it is just c_1. In order to keep track of whether or not a given parser accepts the empty string we will use booleans:

$$\begin{array}{lll}
\mathit{Empty} & : & \mathit{Set} \\
\mathit{Empty} & = & \mathit{Bool}
\end{array}$$

The value true means that a parser accepts the empty string, and false that it does not. (Constructors of an inductive data type are typeset using a sans serif font. *Set* is a type of small types.)

We will only represent the corners of a parser approximately. Every parser will be annotated with a tree of "positions" in which it is not allowed to call itself:

```
data Corners : Set where
  ε    : Corners
  _∪_  : Corners → Corners → Corners
```

For instance, if c_1 and c_2 encode where p_1 and p_2 are not allowed to call themselves, and p_1 accepts the empty string, then $c_1 \cup c_2$ encodes where $p_1 \gg p_2$ may not call itself. If p_1 does accept the empty string, then $c_1 \cup \varepsilon$ is used instead. In Agda values of inductive types like *Corners* are always finite. This implies that it will not be possible to form directly left recursive parsers like $p = p \gg p$, because the representation of p's corners would have to satisfy either $c = c \cup \varepsilon$ or $c = c \cup c$, and these equations have no finite solutions. Indirect left recursion like

```
mutual
  p₁ = p₂ ≫ ...
  p₂ = p₁ ≫ ...
```

leads to similar equations, and is also ruled out.

One form of left recursion is not ruled out by the use of *Corners*: $p = p$ and similar definitions are accepted. However, such definitions are rejected by Agda's termination checker. Some readers may now worry that there is some corner case which we have not taken into consideration, and that therefore the termination guarantees are void. However, take note of the following: *All* programs accepted by Agda are total,[2] so the worst thing that can happen is that the parser combinators defined in this paper are unusable; they *will* guarantee termination.

The two types *Empty* and *Corners* are wrapped up in a record, along with a convenient constructor function:

```
record Index : Set where
  field
    empty   : Empty
    corners : Corners

_◇_ : Empty → Corners → Index
e ◇ c = record {empty = e; corners = c}
```

Note that Agda records come with definitional η equality. This means that the equality $empty\ i \diamond corners\ i = i$ is automatically used when type checking Agda programs. A reader familiar with dependent types may note that this equality is used repeatedly below.

3.2 Basic combinators

In order to allow cyclic definitions of parsers we let the basic parser combinators be constructors of a coinductive type:

[2]Modulo any bugs in the implementation, use of the flag turning off the termination checker, etc.

```
codata Parser (Tok : Set) : Index → Set → Set1 where
```

The *Parser* type takes three arguments: the token type, the *Index* and the return type (the parameter *Tok* scopes over all the constructors; *Set1* is a type of large types with *Set* : *Set1*). Let us consider one constructor at a time:

```
return : ∀ {R} → R → Parser Tok (true ◇ ε) R
```

The parser `return` x immediately returns x, so it does accept the empty string, and it has no sub-parsers, so ε is used as the *Corner* index. (Constructors of a coinductive type are typeset using a `typewriter` font.)

```
fail : ∀ {R} → Parser Tok (false ◇ ε) R
```

Because `fail` always fails it does not accept the empty string.

```
token : Parser Tok (false ◇ ε) Tok
```

Whenever the parser `token` succeeds it consumes one token, so `token` does not accept the empty string.

```
_|_ : ∀ {e₁ e₂ c₁ c₂ R} →
        Parser Tok (e₁      ◇ c₁      ) R →
        Parser Tok (     e₂ ◇      c₂) R →
        Parser Tok (e₁ ∨ e₂ ◇ c₁ ∪ c₂) R
```

The parser p_1 | p_2 accepts the empty string when either p_1 or p_2 does, and $c_1 \cup c_2$ represents the union of the corners of p_1 and p_2.

The only remaining combinator is _≫=_. Here we have a problem. Consider p_1 ≫= p_2. What should the type of p_2 be? We are working in a dependently typed language, so it is not unreasonable to want the index of p_2 to depend on the result of p_1, as captured by the type $(x : R_1) \to$ *Parser Tok* $(i_2\ x)\ R_2$ (where i_2 is a function of type $R_1 \to$ *Index*). However, if p_2 is allowed such a general type then we cannot give a type to p_1 ≫= p_2, because the argument x is not known statically. In an important special case p_2 can have this type, though: when p_1 does not accept the empty string the *Index* of p_2 is irrelevant, because all corners of p_1 ≫= p_2 have to come from p_1. Hence we include two variants of _≫=_:

```
_!≫=_ : ∀ {c₁ R₁ R₂} {i₂ : R₁ → Index} →
                     Parser Tok (false ◇ c₁     ) R₁  →
          ((x : R₁) → Parser Tok (i₂ x)            R₂) →
                     Parser Tok (false ◇ c₁ ∪ ε) R₂
```

In p_1 !≫= p_2 the exclamation mark is meant to indicate that p_1 is "strict"; it does not accept the empty string. Note that p_1 !≫= p_2 does not accept the empty string either, and that its corners come only from p_1.

```
_?≫=_ : ∀ {e₂ c₁ c₂ R₁ R₂} →
                 Parser Tok (true ◇ c₁      ) R₁  →
          (R₁ → Parser Tok (e₂   ◇      c₂) R₂) →
                 Parser Tok (e₂   ◇ c₁ ∪ c₂) R₂
```

The parser p_1 ?≫= p_2 can accept the empty string iff p_2 can, and its corners include both those from p_1 and those from p_2.

Given the two variants of _≫=_ above it is straightforward to define a further variant which works regardless of the first parser's index:

```
_·_ : Index → Index → Index
i₁ · i₂ = (empty i₁ ∧ empty i₂)
          ◇
          (if empty i₁ then corners i₁ ∪ corners i₂
                       else corners i₁ ∪ ε)
_≫=_ : ∀ {e₁ c₁ i₂ Tok R₁ R₂} → let i₁ = e₁ ◇ c₁ in
              Parser Tok i₁        R₁  →
       (R₁ → Parser Tok        i₂  R₂) →
              Parser Tok (i₁ · i₂) R₂
_≫=_ {false} = _!≫=_
_≫=_ {true}  = _?≫=_
```

Note that the resulting index $i_1 \cdot i_2$ is not inferred automatically by Agda here, because _≫=_ pattern matches on e_1 (one can pattern match on implicit arguments by enclosing the patterns in $\{\ldots\}$). When the index depends on the input in a less complicated way it can often be inferred automatically, though:

```
_≫_ : ∀ {Tok i₁ i₂ R₁ R₂} →
        Parser Tok i₁ R₁ → Parser Tok i₂ R₂ → Parser Tok _ R₂
p₁ ≫ p₂ = p₁ ≫= λ _ → p₂
```

The _ character in the type signature is a request to Agda to try to infer the corresponding expression; Agda complains if it cannot do this. (This limited inference is performed by Agda's unification mechanism. The value of _ can be inferred if it is *uniquely* determined, up to definitional equality, by other parts of the program.)

4 A simple backend

Let us now define a backend, i.e. a *parse* function, for the *Parser* type defined in Section 3. We will define a simple, very inefficient backtracking backend, but it should not be too hard to implement a more efficient, memoising one (Frost and Szydlowski 1996).[3]

4.1 Bounded vectors

The parser backend uses *bounded* vectors, i.e. lists with a specified *maximum* length:

```
data BoundedVec (A : Set) : ℕ → Set where
  []  : ∀ {n} → BoundedVec A n
  _∷_ : ∀ {n} → A → BoundedVec A n → BoundedVec A (suc n)
```

[3]Unfortunately our attempt at implementing a memoising backend currently triggers a performance problem in the Agda type checker.

(Here suc is the successor function on natural numbers.)

If the bound of a *BoundedVec* needs to be adjusted upwards, then ↑ can be used:

```
↑ : ∀ {A m} → BoundedVec A m → BoundedVec A (suc m)
↑ []        =  []
↑ (x :: xs) =  x :: ↑ xs
```

Using a non-constant-time function just to modify a type index may seem wasteful, but note that the backend implemented in this section is exponentially slow for many parsers anyway. The code presented in this paper is aimed at clarity rather than efficiency. Instead we take the opportunity to explain the basics of how Agda's pattern matching works. When pattern matching on a constructor the constructor's type is unified with the corresponding type in the function's type signature. In the second case of ↑ this means that the m in the type signature of ↑ is unified with suc n from the type signature of _::_. The expected type for the right-hand side is thus transformed to *BoundedVec A* (suc (suc n)), which matches the actual type of the right-hand side.

Sometimes one also needs to evaluate expressions in order to see that they are type correct. This is the case for *fromList*, which converts an ordinary list to a bounded vector:

```
fromList : ∀ {A} (xs : List A) → BoundedVec A (length xs)
fromList []        =  []
fromList (y :: ys) =  y :: fromList ys
```

(Note that lists and vectors use the same constructor names; constructors can be overloaded in Agda.) In the second case *xs* in the type signature is unified with *y* :: *ys*, so the expected type for the right-hand side is *BoundedVec A* (*length* (*y* :: *ys*)) (where *length* calculates the length of a list). The actual type of the right-hand side is *BoundedVec A* (suc (*length ys*)), which matches the expected type because *length* (*y* :: *ys*) evaluates to suc (*length ys*).

It is also possible to convert bounded vectors to ordinary lists:

```
toList : ∀ {A n} → BoundedVec A n → List A
toList []        =  []
toList (x :: xs) =  x :: toList xs
```

4.2 Parser monad

Bounded vectors representing the remaining input are used as the state component of a parser monad. This monad is built from a backtracking list monad to which a state monad transformer has been applied (this implies that changes to the state are not preserved when backtracking occurs). The monad is *parameterised* (Atkey 2006) on the initial and final upper bounds of the vector:

```
P : Set → ℕ → ℕ → Set → Set
P Tok i f A = BoundedVec Tok i → List (A × BoundedVec Tok f)
```

The implementation of the monad, which is standard, is not given here, just the type signatures of the operations. The monad operations are primed to distinguish them from the parser combinators defined above:

$$\begin{array}{lll}
\mathit{return}' & : & \forall\,\{\mathit{Tok}\ i\ A\} \to A \to P\ \mathit{Tok}\ i\ i\ A \\
\_\!\gg\!=\!'\_ & : & \forall\,\{\mathit{Tok}\ i\ m\ f\ A\ B\} \to \\
 & & \quad P\ \mathit{Tok}\ i\ m\ A \to (A \to P\ \mathit{Tok}\ m\ f\ B) \to P\ \mathit{Tok}\ i\ f\ B \\
\_\!\gg\!'\_ & : & \forall\,\{\mathit{Tok}\ i\ m\ f\ A\ B\} \to \\
 & & \quad P\ \mathit{Tok}\ i\ m\ A \to P\ \mathit{Tok}\ m\ f\ B \to P\ \mathit{Tok}\ i\ f\ B \\
\mathit{fail}' & : & \forall\,\{\mathit{Tok}\ i\ A\} \to P\ \mathit{Tok}\ i\ f\ A \\
\_|'\_ & : & \forall\,\{\mathit{Tok}\ i\ f\ A\} \to \\
 & & \quad P\ \mathit{Tok}\ i\ f\ A \to P\ \mathit{Tok}\ i\ f\ A \to P\ \mathit{Tok}\ i\ f\ A \\
\mathit{get}' & : & \forall\,\{\mathit{Tok}\ i\} \to P\ \mathit{Tok}\ i\ i\ (\mathit{BoundedVec}\ \mathit{Tok}\ i) \\
\mathit{put}' & : & \forall\,\{\mathit{Tok}\ i\ f\} \to \mathit{BoundedVec}\ \mathit{Tok}\ f \to P\ \mathit{Tok}\ i\ f\ \top \\
\mathit{modify}' & : & \forall\,\{\mathit{Tok}\ i\ f\} \to \\
 & & \quad (\mathit{BoundedVec}\ \mathit{Tok}\ i \to \mathit{BoundedVec}\ \mathit{Tok}\ f) \to P\ \mathit{Tok}\ i\ f\ \top
\end{array}$$

(Here $\top$ is a unit type.)

4.3 The *parse* function

The *parse* function can now be defined. It uses the functions *parse*↓ and *parse*↑ which are defined by mutual lexicographic structural recursion over

1. the upper bound of the length of the input string, and
2. the corners.

The corner indices are written out as subscripts below in order to make the code easier to follow, but Agda does not need this information; in the actual code the indices are inferred automatically from the *Parser* arguments.

The type of *parse*↓ forces non-zero bounds to decrease if the parser does not accept the empty string (*pred* is the predecessor function on natural numbers):

$$\begin{array}{lll}
\mathit{parse}\!\downarrow & : & \forall\,\{e\ c\ \mathit{Tok}\ R\}\ n \to \\
 & & \quad \mathit{Parser}\ \mathit{Tok}\ (e \diamond c)\ R \to P\ \mathit{Tok}\ n\ (\textbf{if}\ e\ \textbf{then}\ n\ \textbf{else}\ \mathit{pred}\ n)\ R
\end{array}$$

This type states that *parse*↓ never decreases the upper bound by more than one. However, a parser can consume more than one token from the input string, so it is necessary to be able to artificially increase the upper bound. This is taken care of by *parse*↑:

$$\begin{array}{llll}
\mathit{parse}\!\uparrow & : & \multicolumn{2}{l}{\forall\,\{e\ c\ \mathit{Tok}\ R\}\ n \to \mathit{Parser}\ \mathit{Tok}\ (e \diamond c)\ R \to P\ \mathit{Tok}\ n\ n\ R} \\
\mathit{parse}\!\uparrow_c\ \{\mathsf{true}\}\ n & p & = & \mathit{parse}\!\downarrow_c\ n\ p \\
\mathit{parse}\!\uparrow_c\ \{\mathsf{false}\}\ \mathsf{zero} & p & = & \mathit{fail}' \\
\mathit{parse}\!\uparrow_c\ \{\mathsf{false}\}\ (\mathsf{suc}\ n) & p & = & \mathit{parse}\!\downarrow_c\ (\mathsf{suc}\ n)\ p \gg\!=' \lambda\, r \to \\
 & & & \mathit{modify}'\ \uparrow \qquad\qquad \gg' \\
 & & & \mathit{return}'\ r
\end{array}$$

Note that it is safe to fail immediately if the upper bound is zero and the parser does not accept the empty string.

Let us now turn to the implementation of *parse*↓. The `return` and `fail` cases use *return*′ and *fail*′ from the parser monad:

$$\begin{array}{lll}
\mathit{parse}\!\downarrow_\varepsilon\ n\ (\mathtt{return}\ x) & = & \mathit{return}'\ x \\
\mathit{parse}\!\downarrow_\varepsilon\ n\ \mathtt{fail} & = & \mathit{fail}'
\end{array}$$

In the `token` case $\mathit{parse}{\downarrow}$ gets the current input string from the state and, if possible, replaces it with its tail and returns its head:

$$
\begin{array}{l}
\mathit{parse}{\downarrow}_{\varepsilon}\ n\ \mathtt{token}\ =\ \mathit{get}'\ \gg\!=' \ \mathit{eat} \\
\quad \textbf{where} \\
\quad \mathit{eat}\ :\ \forall\,\{\mathit{Tok}\ n\} \to \mathit{BoundedVec}\ \mathit{Tok}\ n \to P\ \mathit{Tok}\ n\ (\mathit{pred}\ n)\ \mathit{Tok} \\
\quad \mathit{eat}\ [\,]\qquad\quad\ =\ \mathit{fail}' \\
\quad \mathit{eat}\ (c :: s)\ =\ \mathit{put}'\ s\ \gg'\ \mathit{return}'\ c
\end{array}
$$

In the $p_1 \mid p_2$ case we distinguish between three different sub-cases, corresponding to different *Empty* indices for p_1 and p_2:

$$
\begin{array}{lcl}
\mathit{parse}{\downarrow}_{c_1 \cup c_2}\ n\ (\_|\_\ \{\mathsf{true}\}\qquad\quad\ \ p_1\ p_2) & = & \mathit{parse}{\downarrow}_{c_1}\ n\ p_1\ |'\ \mathit{parse}{\uparrow}_{c_2}\ n\ p_2 \\
\mathit{parse}{\downarrow}_{c_1 \cup c_2}\ n\ (\_|\_\ \{\mathsf{false}\}\ \{\mathsf{true}\}\ \ p_1\ p_2) & = & \mathit{parse}{\uparrow}_{c_1}\ n\ p_1\ |'\ \mathit{parse}{\downarrow}_{c_2}\ n\ p_2 \\
\mathit{parse}{\downarrow}_{c_1 \cup c_2}\ n\ (\_|\_\ \{\mathsf{false}\}\ \{\mathsf{false}\}\ p_1\ p_2) & = & \mathit{parse}{\downarrow}_{c_1}\ n\ p_1\ |'\ \mathit{parse}{\downarrow}_{c_2}\ n\ p_2
\end{array}
$$

When pattern matching on true and false here the return type of $\mathit{parse}{\downarrow}$ gets instantiated; for instance, in the last case it gets instantiated to

$$P\ \mathit{Tok}\ n\ (\textbf{if}\ (\mathsf{false} \lor \mathsf{false})\ \textbf{then}\ n\ \textbf{else}\ \mathit{pred}\ n)\ R$$

(for some R), and this evaluates to $P\ \mathit{Tok}\ n\ (\mathit{pred}\ n)\ R$. The expected type for the right-hand side then determines whether the recursive calls need to go via $\mathit{parse}{\uparrow}$ or not. The results are combined using the choice combinator $\_|'\_$ from the parser monad. Note also that the upper bound is preserved in the recursive calls and that the corner index always becomes structurally smaller.

Finally we have the cases for the two bind constructors. The $\_?\!\gg\!=\_$ case is straightforward:

$$\mathit{parse}{\downarrow}_{c_1 \cup c_2}\ n\ (p_1\ ?\!\gg\!=\ p_2)\ =\ \mathit{parse}{\downarrow}_{c_1}\ n\ p_1\ \gg\!='\ \mathit{parse}{\downarrow}_{c_2}\ n \circ p_2$$

The bind from the parser monad is used to combine the results from the two sub-parsers. (Here $\_\circ\_$ binds tighter than $\_\!\gg\!='\_$.) In the $\_!\!\gg\!=\_$ case we make use of the knowledge that the first parser has to consume a token when it succeeds, which implies that the upper bound is lower in the second recursive call (the corner index can be anything):

$$
\begin{array}{lcl}
\mathit{parse}{\downarrow}_{c \cup \varepsilon}\ \mathsf{zero}\qquad\ (p_1\ !\!\gg\!=\ p_2) & = & \mathit{fail}' \\
\mathit{parse}{\downarrow}_{c \cup \varepsilon}\ (\mathsf{suc}\ n)\ (p_1\ !\!\gg\!=\ p_2) & = & \mathit{parse}{\downarrow}_{c}\ (\mathsf{suc}\ n)\ p_1\ \gg\!='\ \mathit{parse}{\uparrow}\ n \circ p_2
\end{array}
$$

Given $\mathit{parse}{\downarrow}$ we can define *parse* by massaging the input and output a little:

$$
\begin{array}{l}
\mathit{parse}\ :\ \forall\,\{\mathit{Tok}\ i\ R\} \to \\
\qquad\qquad \mathit{Parser}\ \mathit{Tok}\ i\ R \to \mathit{List}\ \mathit{Tok} \to \mathit{List}\ (R \times \mathit{List}\ \mathit{Tok}) \\
\mathit{parse}\ p\ \mathit{toks}\ =\ \mathit{map}\ (\mathit{map}\text{-}{\times}\ \mathit{id}\ \mathit{toList})\ (\mathit{parse}{\downarrow}\ \_\ p\ (\mathit{fromList}\ \mathit{toks}))
\end{array}
$$

(Here $\mathit{map}\text{-}{\times}\ f\ g\ (x, y)\ =\ (f\ x, g\ y)$.)

5 Examples

Given the parser combinators defined in Section 3 we can start to build up a library of reusable combinators. For instance, we can define the Kleene star, i.e.

a combinator $\_\star$ such that $p \star$ parses p zero or more times. This combinator is defined mutually with $\_+$, which stands for *one* or more repetitions:

```
mutual
  _⋆  : ∀ {Tok c R} → Parser Tok (false ◇ c) R → Parser Tok _ (List R)
  p ⋆  ∼  return [] ∣ p +
  _+  : ∀ {Tok c R} → Parser Tok (false ◇ c) R → Parser Tok _ (List R)
  p +  ∼  p    !≫= λ x  →
          p ⋆ ?≫= λ xs →
          return (x :: xs)
```

Note that we require that the argument parser p does not accept the empty string; parsing the empty string many times is not useful, and the parsers would be left recursive if p did accept the empty string. Note also that Agda can figure out the indices of $p \star$ and $p +$ automatically (they are $\mathsf{true} \diamond \varepsilon \cup (c \cup \varepsilon)$ and $\mathsf{false} \diamond c \cup \varepsilon$, respectively). The index of p cannot be inferred, because the *Corners* index could be anything, and Agda only infers values when they are uniquely determined. The *Empty* index could be inferred, though.

The definitions of $\_\star$ and $\_+$ use *guarded corecursion*; corecursive definitions in Agda use $\sim$ instead of $=$. The corecursive calls are accepted by Agda because they are guarded, which roughly means that they take place under coinductive constructors. Unfortunately Agda's termination checker is not smart enough to see that it is safe to replace the constructors $\_!\gg\!=\_$ and $\_?\gg\!=\_$ with $\_\gg\!=\_$ in the definition of $\_+$. The following variant is rejected:

```
p +  ∼  p    ≫= λ x  →
        p ⋆  ≫= λ xs →
        return (x :: xs)
```

This problem implies that it can be tricky to use derived parser combinators, so in Section 6 we show how one can work around it by using grammars (functions from non-terminals to parsers).

It should be noted that in many cases corecursion does not need to be used. Often the parser being defined is not recursive, like the parser p *sepBy sep*, which parses one or more ps separated by *sep*arators:

```
_sepBy_ : ∀ {Tok i c R R′} →
            Parser Tok i R → Parser Tok (false ◇ c) R′ →
            Parser Tok _ (List R)
p sepBy sep  =  p               ≫= λ x  →
                (sep ≫ p) ⋆  ≫= λ xs →
                return (x :: xs)
```

In other cases the parser is defined by recursion over some argument. This is the case for *exactly* n p, which parses p exactly n times (*Vec* R n is the type of lists of length n containing values of type R):

```
exactly-index : Index → ℕ → Index
exactly-index i zero     =  _
exactly-index i (suc n)  =  _
exactly : ∀ {Tok i R} n →
```

```
                 Parser Tok i R →
                 Parser Tok (exactly-index i n) (Vec R n)
exactly zero     p  =  return []
exactly (suc n) p  =  p             >>= λ x  →
                      exactly n p   >>= λ xs →
                      return (x :: xs)
```

When a parser is defined by pattern matching Agda often fails to infer its index automatically, so a function which computes the index needs to be given. However, it often suffices to give the pattern matching structure of the index function. Note that the right-hand sides of *exactly-index* above are inferred.

Let us now define the parser *sat* p, which accepts a token *tok* if it satisfies the predicate p, i.e. if p *tok* is just *something* (and in that case *something* is returned):

```
sat : ∀ {Tok R} → (Tok → Maybe R) → Parser Tok _ R
sat {Tok} {R} p  =  token !>>= λ c → ok (p c)
  where
  ok-index : Maybe R → Index
  ok-index nothing   =  _
  ok-index (just _)  =  _

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

Note that it is crucial that the index of the second argument to _!>>=_ can depend on the return value of the first parser. By using *sat* it is easy to define the parser *theChar* c, which only accepts the character c:

```
theChar : Char  →  Parser Char _ Char
theChar c  =  sat (λ c′ → if c == c′ then just c′ else nothing)
```

The typing discipline imposed by the library can sometimes make parsers harder to implement. Consider the following attempt at defining the language $\{\, \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \in \mathbb{N} \,\}$ (which, incidentally, is not context-free):

```
aⁿbⁿcⁿ  =  theChar 'a' ⋆              >>= λ as →
           let n = length as in
           exactly n (theChar 'b') >>
           exactly n (theChar 'c') >>
           return n
```

Here Agda complains that the index of the body of the lambda expression depends on *as*, which is not allowed by the type of _>>=_. The problem can be worked around by separating out the case where n is zero:

```
aⁿbⁿcⁿ  =  return zero
        |  theChar 'a' +              !>>= λ as →
           let n = length as in
           exactly n (theChar 'b') >>
           exactly n (theChar 'c') >>
           return n
```

Workarounds like this one have the potential to be annoying, but they are only necessary when the monadic bind is used in an essential way, i.e. when the result of one parser is used to influence how to parse the remaining input. The problem does not exist for parsers defined using an applicative functor interface (McBride and Paterson 2008). Furthermore the problem disappears if the first argument to _≫=_ is known to consume at least one token, because then _!≫=_ can be used. Similarly, if the second argument to _≫=_ is known to consume a token before the result of the first argument is used, then the problem also disappears, because the dependency does not show up in the second argument's index.

The typing discipline causes other problems as well. Consider *p sepBy sep*. The requirement that *sep* must reject the empty string is fairly arbitrary; one could just as well put the requirement on *p*. However, the implementation has to be modified, because the *Empty* index of *sep* ≫ *p*, *empty i′* ∧ false, does not evaluate to false (_∧_ pattern matches on its first argument). One possible implementation is as follows:

```
_<⊛_ : ∀ {Tok i₁ i₂ R₁ R₂} →
         Parser Tok i₁ R₁ → Parser Tok i₂ R₂ → Parser Tok _ R₁
p₁ <⊛ p₂ = p₁ ≫= λ r → p₂ ≫ return r
_sepBy′_ : ∀ {Tok c i′ R R′} →
             Parser Tok (false ◇ c) R → Parser Tok i′ R′ →
             Parser Tok _ (List R)
p sepBy′ sep = (p <⊛ sep) ⋆ ≫= λ xs →
                p           ≫= λ x  →
                return (xs ++ x :: [])
```

Having two separate functions for almost the same task is not very elegant, though. The functions can be unified by requiring the user to supply a proof showing that the empty string is rejected by *at least one* of *p* and *sep*. The propositional equality type _≡_ can be used to represent such proofs:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

The unified version of _*sepBy*_ can then be defined by using a function which casts a parser's *Empty* index to a provably equal variant:

```
cast : ∀ {Tok e₁ e₂ c R} →
         e₁ ≡ e₂ → Parser Tok (e₁ ◇ c) R → Parser Tok (e₂ ◇ c) R
cast refl p = p
_sepBy⟨_⟩_ : ∀ {Tok i i′ R R′} →
               Parser Tok i R →
               empty i′ ∧ empty i ≡ false →
               Parser Tok i′ R′ →
               Parser Tok _ (List R)
p sepBy⟨ nonEmpty ⟩ sep =
  p                             ≫= λ x  →
  cast nonEmpty (sep ≫ p) ⋆ ≫= λ xs →
  return (x :: xs)
```

However, _*sepBy*⟨_⟩_ is more awkward to use than the version of _*sepBy*_ found in ordinary parser combinator libraries.

6 Grammars

As noted in Section 5 the coinductive approach, based on guarded coinduction, does not always interact well with abstraction. This section shows how one can avoid the problem by working with explicit grammars, at the expense of more verbosity for the cases which are handled well by the coinductive approach. Fortunately the two approaches can easily be combined. Note also that if the source language does not support coinductive types, then one can work solely with the grammar-based approach by using an *inductive* definition of *Parser*.

Consider the following simple expression recogniser:

```
addOp = theChar '+' | theChar '-'
mulOp = theChar '*' | theChar '/'

mutual
  expr   ~ term   sepBy addOp
  term   ~ factor sepBy mulOp
  factor ~ theChar '0'
         | theChar '(' ≫ expr ≫ theChar ')'
```

Agda does not accept the last mutual definition because the corecursion is not guarded. In the grammar-based approach this recogniser is instead defined as follows (leaving the definitions of *addOp* and *mulOp* unchanged):

```
data NT : NonTerminalType where
  expr   : NT _ _
  term   : NT _ _
  factor : NT _ _

grammar : Grammar NT Char
grammar expr   = ! term   sepBy addOp
grammar term   = ! factor sepBy mulOp
grammar factor = theChar '0'
               | theChar '(' ≫ ! expr ≫ theChar ')'
```

Note that the corecursion in the previous definition has been removed in favour of explicit references to non-terminals (values of type *NT*), and that the non-terminal indices and return types (the arguments to *NT*) are automatically inferred.

Let us now explain the grammar-based definition. *NonTerminalType*, the type of non-terminal types, just requires non-terminal types to have the same indices as parsers:

```
NonTerminalType : Set2
NonTerminalType = Index → Set → Set1
```

The type *Grammar NT Tok* specifies that a grammar for the non-terminal type *NT* maps non-terminals to parsers:

```
Grammar : NonTerminalType → Set → Set1
Grammar NT Tok = ∀ {i R} → NT i R → Parser NT Tok i R
```

Parsers are parameterised on a type of non-terminals, and there is a new constructor ! which lifts non-terminals to parsers:

```
codata Parser (NT : NonTerminalType) (Tok : Set) :
                NonTerminalType where
  ...
  ! : ∀ {e c R} → NT (e ◇ c) R → Parser NT Tok (e ◇ step c) R
```

Here step : *Corners* → *Corners* is a new *Corners* constructor. A grammar is passed around by *parse↓* and used when non-terminals are encountered:

```
parse↓ : ∀ {NT Tok} (g : Grammar NT Tok) {e c R} n →
           Parser NT Tok (e ◇ c) R →
           P Tok n (if e then n else pred n) R
...
parse↓_{step c} g n (! x)  =  parse↓_c g n (g x)
```

Note that the corner index is smaller in the recursive call. Finally the *parse* function also needs to be updated:

```
parse : ∀ {NT Tok i R} →
          Grammar NT Tok → Parser NT Tok i R →
          List Tok → List (R × List Tok)
parse g p toks  =  map (map-× id toList) (parse↓ g _ p (fromList toks))
```

The use of a grammar can sometimes be isolated to one part of a larger parser, because when the *Parser* type is coinductive it is possible to remove all non-terminals from a parser by corecursively instantiating them:

```
drop-steps : Corners → Corners
drop-steps ε           =  ε
drop-steps (step c)    =  drop-steps c
drop-steps (c1 ∪ c2)   =  drop-steps c1 ∪ drop-steps c2

⟦_⟧ : ∀ {Tok NT1 NT2 e c R} →
        Parser NT1 Tok (e ◇            c) R → Grammar NT1 Tok →
        Parser NT2 Tok (e ◇ drop-steps c) R
⟦ return x  ⟧ g  =  return x
⟦ fail      ⟧ g  =  fail
⟦ token     ⟧ g  =  token
⟦ p1 | p2   ⟧ g  =  ⟦ p1 ⟧ g | ⟦ p2 ⟧ g
⟦ p1 ?>>= p2 ⟧ g  =  ⟦ p1 ⟧ g ?>>= λ x → ⟦ p2 x ⟧ g
⟦ p1 !>>= p2 ⟧ g  ~  ⟦ p1 ⟧ g !>>= λ x → ⟦ p2 x ⟧ g
⟦ ! nt      ⟧ g  =  ⟦ g nt ⟧ g
```

Note that corecursion is only used for the _!>>=_ case; in all other cases the *Corners* index decreases. The function is defined by a lexicographic mixture of guarded corecursion and structural recursion. Note also that NT_2 can be instantiated with any non-terminal type, including an empty one. By using ⟦_⟧ one can define an expression recogniser which can be used with any non-terminal type:

```
expression : ∀ {NT} → Parser NT Char _ _
expression  =  ⟦ ! expr ⟧ grammar
```

The *expression* example does not explain how one can define parser *combinators* using the grammar-based approach. However, note that non-terminal constructors can take arguments, which means that the non-terminals can be parameterised (and hence that the grammars can be infinite). By parameterising the non-terminals on parsers one can construct grammars corresponding to parser combinator libraries. In order to make it possible to use non-terminals from other grammars as arguments to these parser combinators it is important that any non-terminal type is accepted by the parsers:

```
data LibraryNT (NT : NonTerminalType) (Tok : Set) :
                 NonTerminalType where
  _⋆  : ∀ {c R} → Parser      NT Tok (false ◇ c) R →
                  LibraryNT NT Tok _           (List R)
  _+  : ∀ {c R} → Parser      NT Tok (false ◇ c) R →
                  LibraryNT NT Tok _           (List R)
```

A library grammar can then be defined by parameterising it on a function lifting library non-terminals to other non-terminals:

```
library : ∀ {NT Tok} →
            (∀ {i R} → LibraryNT NT Tok i R → NT i R) →
             ∀ {i R} → LibraryNT NT Tok i R → Parser NT Tok i R
library lift (p ⋆)  =  return [] ∣ ! (lift (p +))
library lift (p +)  =  p              >>= λx  →
                       ! (lift (p ⋆)) >>= λxs →
                       return (x ∷ xs)
```

Finally a library grammar can be "imported" into another one by including a non-terminal lib which is parameterised on library non-terminals:

```
data NT : NonTerminalType where
  lib : ∀ {i R} → LibraryNT NT Char i R → NT i R
  a   : NT _ Char
  as  : NT _ (List Char)

grammar : Grammar NT Char
grammar (lib nt)  =  library lib nt
grammar a         =  theChar 'a'
grammar as        =  ! (lib ((! a) ⋆))
```

Note that _+ was defined using _>>=_ rather than _!>>=_ and _?>>=_ above; the problem with reuse identified in Section 5 is solved by the grammar-based approach. This approach does impose some notational overhead, but can be quite convenient, at least until the methods used to establish productivity in languages like Agda have matured. The approach has for instance been used to define a parser for mixfix operators (Danielsson and Norell 2008).

7 Discussion

We have presented a parser combinator library which guarantees termination, and discussed some consequences of working with our more rigidly typed parser combinators.

There does not seem to be much prior work on *formally* verified termination for parser combinators (or other general parsing frameworks). We only know of

McBride and McKinna (2002), which is based on similar ideas as this paper, but seems to be more of a proof of concept. McBride and McKinna also use types to rule out left recursion, but in a different way. They define parsers and grammars inductively and allow only a single (non-parameterised) non-terminal, so they get away with only keeping track of whether or not a token has been consumed. Furthermore their implementation is arguably harder to understand than ours; at least it differs more from ordinary list of successes combinator parsers.

It is also interesting to compare this paper with the work of Frost et al. (2008), who have implemented parser combinators which can handle left recursion. On the one hand their interface is more general than ours, since they do not need to avoid left recursion. On the other hand their users get fewer guarantees about termination: *if* the parser combinators are used in the right way, *and* the informal termination proof given by Frost et al. is correct, *then* parsing will terminate. Furthermore more precise type information can sometimes be useful, even if termination is not at stake. For instance, if $\_\star$ is applied to a parser which accepts the empty string, then this is likely to be a semantic error, even if the underlying parsing mechanism can cope with it. Fortunately the two approaches are not mutually exclusive; it should be possible to formally prove termination for parser combinators which can handle left recursion.

Acknowledgements

We would like to thank Andres Löh, whose presentation of this work gave some ideas about how to write this paper.

References

The Agda Team. The Agda Wiki. Available at `http://www.cs.chalmers.se/~ulfn/Agda/`, 2008.

Robert Atkey. Parameterised notions of computation. In *Workshop on Mathematically Structured Functional Programming (MSFP 2006)*, 2006.

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

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

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 and Ulf Norell. Parsing mixfix operators. Submitted for publication, 2008.

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 and Barbara Szydlowski. Memoizing purely functional top-down backtracking language processors. *Science of Computer Programming*, 27(3):263–288, 1996.

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.

Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es), 1996.

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.

Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8:437–444, 1998.

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

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.

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

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.

Robert C. Moore. Removing left recursion from context-free grammars. In *Proceedings of the 1st Meeting of the North American Chapter of the Association for Computational Linguistics*, pages 249–255, 2000.

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

Nicolas Oury and Wouter Swierstra. The power of pi. In *ICFP '08: Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 39–50, 2008.

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

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

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.

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.