Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Terminating combinator parsers in Agda

Andres Löh

based on work by Nils Anders Danielsson and Ulf Norell

Department of Information and Computing Sciences
Utrecht University

June 12, 2008

Overview

Totality

Parser combinators

Terminating combinator parsers

Totality

Total functions

A function is called **total** if it terminates and produces a valid (non-$\perp$) result for any input.

Total functions

A function is called **total** if it terminates and produces a valid (non-$\perp$) result for any input.

Many Haskell functions are not total:

Total functions

A function is called **total** if it terminates and produces a valid (non-$\perp$) result for any input.

Many Haskell functions are not total:

$$\begin{aligned} &\mathsf{head} :: [\mathsf{a}] \rightarrow \mathsf{a} \\ &\mathsf{head}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{x} \end{aligned}$$

Fails on the empty list.

Total functions

A function is called **total** if it terminates and produces a valid (non-$\perp$) result for any input.

Many Haskell functions are not total:

$$\mathsf{head} :: [\mathsf{a}] \rightarrow \mathsf{a}$$
$$\mathsf{head}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{x}$$

Fails on the empty list.

$$\mathsf{factorial} :: \mathsf{Int} \rightarrow \mathsf{Int}$$
$$\mathsf{factorial}\ 0 = 1$$
$$\mathsf{factorial}\ \mathsf{n} = \mathsf{n} * \mathsf{factorial}\ (\mathsf{n} - 1)$$

Loops on any negative input.

Agda is (in spirit) a total language

- All Agda functions are supposed to be total.

Agda is (in spirit) a total language

- All Agda functions are supposed to be total.
- Writing a function that the compiler cannot easily see to be terminating results in a **compiler error**.

Agda is (in spirit) a total language

- All Agda functions are supposed to be total.
- Writing a function that the compiler cannot easily see to be terminating results in a **compiler warning**.

Agda is (in spirit) a total language

- All Agda functions are supposed to be total.
- Writing a function that the compiler cannot easily see to be terminating results in a **compiler warning**.
- Other dependently typed systems (Epigram, Coq) are similar in this respect.

Agda is (in spirit) a total language

- All Agda functions are supposed to be total.
- Writing a function that the compiler cannot easily see to be terminating results in a **compiler warning**.
- Other dependently typed systems (Epigram, Coq) are similar in this respect.

Why?

Reasons for totality

- In Haskell, every type is **inhabited**: $\perp$ is a value of any type.

Reasons for totality

- In Haskell, every type is **inhabited**: $\perp$ is a value of any type.
- In dependently typed languages, we want to use the Curry-Howard correspondence: types are propositions, values are proofs.

```
data _≤_ : ℕ → ℕ → Set where
  ≤base : ∀{n}     →             n  ≤ n
  ≤step : ∀{m n} → m ≤ n → m ≤ suc n
trans : ∀{l m n} → l ≤ m → m ≤ n → l ≤ n
replaceInits : ∀{a m n} → m ≤ n →
               Vec a m → Vec a n → Vec a n
```

Reasons for totality

- In Haskell, every type is **inhabited**: $\perp$ is a value of any type.
- In dependently typed languages, we want to use the Curry-Howard correspondence: types are propositions, values are proofs.

```
data _≤_ : ℕ → ℕ → Set where
  ≤base : ∀{n}   →              n ≤ n
  ≤step : ∀{m n} → m ≤ n → m ≤ suc n
trans : ∀{l m n} → l ≤ m → m ≤ n → l ≤ n
replaceInits : ∀{a m n} → m ≤ n →
                Vec a m → Vec a n → Vec a n
```

- Haskell is **inconsistent**: all propositions can be proved.

Reasons for totality – contd.

- Types can contain terms:

$$
\begin{array}{lll}
\mathsf{Vec} : \mathsf{Set} \to \mathbb{N} \to \mathsf{Set} & & \\
\_ +\!\!+ \_ & : \forall\{\mathsf{a}\ \mathsf{m}\ \mathsf{n}\} & \to \mathsf{Vec}\ \mathsf{a}\ \mathsf{m} \to \mathsf{Vec}\ \mathsf{a}\ \mathsf{n} \to \mathsf{Vec}\ \mathsf{a}\ (\mathsf{m} + \mathsf{n}) \\
\mathsf{tail} & : \forall\{\mathsf{a}\ \mathsf{n}\} & \to \mathsf{Vec}\ \mathsf{a}\ (\mathsf{suc}\ \mathsf{n}) \to \mathsf{Vec}\ \mathsf{a}\ \mathsf{n}
\end{array}
$$

Reasons for totality – contd.

- Types can contain terms:

$$
\begin{aligned}
&\mathsf{Vec} : \mathsf{Set} \to \mathbb{N} \to \mathsf{Set} \\
&\_ +\!\!+ \_ : \forall\{\mathsf{a}\ \mathsf{m}\ \mathsf{n}\} \to \mathsf{Vec\ a\ m} \to \mathsf{Vec\ a\ n} \to \mathsf{Vec\ a}\ (\mathsf{m} + \mathsf{n}) \\
&\mathsf{tail} \quad : \forall\{\mathsf{a}\ \mathsf{n}\} \quad \to \mathsf{Vec\ a}\ (\mathsf{suc\ n}) \to \mathsf{Vec\ a\ n}
\end{aligned}
$$

- Consider:

$$\mathsf{tail}\ (\mathsf{v}_1 +\!\!+ \mathsf{v}_2)$$

Reasons for totality – contd.

- Types can contain terms:

$$
\begin{aligned}
&\mathsf{Vec} : \mathsf{Set} \to \mathbb{N} \to \mathsf{Set} \\
&\_ +\!\!+ \_ : \forall \{ a\ m\ n \} \to \mathsf{Vec}\ a\ m \to \mathsf{Vec}\ a\ n \to \mathsf{Vec}\ a\ (m + n) \\
&\mathsf{tail} \quad : \forall \{ a\ n \} \quad \to \mathsf{Vec}\ a\ (\mathsf{suc}\ n) \to \mathsf{Vec}\ a\ n
\end{aligned}
$$

- Consider:

$$\mathsf{tail}\ (v_1 +\!\!+ v_2)$$

- Typechecking the expression requires unification:

$$(\mathsf{length}\ v_1 + \mathsf{length}\ v_2) \sim \mathsf{suc}\ n$$

(for any n).

Consequences of totality

- Inductively defined datatypes have only **finite** values.
- Evaluation strategy (eager vs. lazy) is semantically irrelevant.
- The language cannot be Turing-complete (but still surprisingly expressive).

How to ensure totality

- Agda has a built-in **coverage** and **termination** checker.
- The coverage checker ensures that in a case analysis, all possible patterns are covered.
- The termination checker essentially checks that functions are **structurally recursive**.

Structural recursion

- Each value essentially is a constructor applied to other values:

  > $v = C\ v_1 \ldots v_n$

- All such subvalues (and their subvalues ...) are **structurally smaller**. Recursive calls must make at least one argument structurally smaller.

Structural recursion

- Each value essentially is a constructor applied to other values:

$$v = C\ v_1 \ldots v_n$$

- All such subvalues (and their subvalues ...) are **structurally smaller**. Recursive calls must make at least one argument structurally smaller.
- Many functions are trivially structurally recursive:

$$\begin{aligned} &\mathsf{length} : \forall \{a\} \to [a] \to \mathbb{N} \\ &\mathsf{length}\ [\,] \qquad\quad = 0 \\ &\mathsf{length}\ (x :: xs) = 1 + \mathsf{length}\ xs \end{aligned}$$

Others (e.g. Quicksort) require some work ...

Parser combinators

Simple parsers

We can do better, but for this talk, we choose a näive implementation (list of successes):

$$\begin{aligned}
&\mathsf{Input} : \mathsf{Set} \\
&\mathsf{Input} = [\mathsf{Char}] \\
&\mathsf{Parser} : \mathsf{Set} \rightarrow \mathsf{Set} \\
&\mathsf{Parser}\ \mathsf{r} = \mathsf{Input} \rightarrow [\mathsf{r} \times \mathsf{Input}]
\end{aligned}$$

Applicative interface

$$\text{fail} : \forall \{ r \} \rightarrow \text{Parser } r$$
$$\text{fail inp} = [\,]$$

$$\text{succeed} : \forall \{ r \} \rightarrow r \rightarrow \text{Parser } r$$
$$\text{succeed } x \text{ inp} = (x, \text{inp}) :: [\,]$$

Applicative interface

$$
\begin{aligned}
&\mathsf{fail} : \forall\{\mathsf{r}\} \rightarrow \mathsf{Parser\ r}\\
&\mathsf{fail\ inp} = [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{succeed} : \forall\{\mathsf{r}\} \rightarrow \mathsf{r} \rightarrow \mathsf{Parser\ r}\\
&\mathsf{succeed\ x\ inp} = (\mathsf{x}, \mathsf{inp}) :: [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\_\mid\_ : \forall\{\mathsf{r}\} \rightarrow \mathsf{Parser\ r} \rightarrow \mathsf{Parser\ r} \rightarrow \mathsf{Parser\ r}\\
&(\mathsf{p} \mid \mathsf{q})\ \mathsf{inp} = \mathsf{p\ inp} +\!\!+\ \mathsf{q\ inp}
\end{aligned}
$$

Applicative interface

```
fail : ∀{r} → Parser r
fail inp = []

succeed : ∀{r} → r → Parser r
succeed x inp = (x, inp) :: []

_ | _ : ∀{r} → Parser r → Parser r → Parser r
(p | q) inp = p inp ++ q inp

_ ⋆ _ : ∀{r s} → Parser (r → s) → Parser r → Parser s
(p ⋆ q) inp =
  concat (map (λf → map (λg → ((π₁ f) (π₁ g), π₂ g))
                             (q (π₂ f)))
              (p inp))
```

Applicative interface – contd.

$$\begin{array}{l}
\mathsf{symbol} : \mathsf{Char} \to \mathsf{Parser\ Char} \\
\mathsf{symbol}\ _\ [\,] \qquad\quad = [\,] \\
\mathsf{symbol}\ \mathsf{x}\ (\mathsf{i} :: \mathsf{inp}) = \mathsf{if}\ \mathsf{i} == \mathsf{x}\ \mathsf{then}\ [\mathsf{x}, \mathsf{inp}]\ \mathsf{else}\ [\,] \\
_\,\$\,_ : \forall\{\mathsf{r}\ \mathsf{s}\} \to (\mathsf{r} \to \mathsf{s}) \to \mathsf{Parser\ r} \to \mathsf{Parser\ s} \\
\mathsf{f}\ \$\ \mathsf{p} = \mathsf{succeed\ f} \star \mathsf{p}
\end{array}$$

Applicative interface – contdd.

$$
\begin{array}{l}
\mathsf{symbol} : \mathsf{Char} \to \mathsf{Parser\ Char} \\
\mathsf{symbol}\ \_\ [\,] \qquad\quad = [\,] \\
\mathsf{symbol}\ \mathsf{x}\ (\mathsf{i} :: \mathsf{inp}) = \mathsf{if}\ \mathsf{i} == \mathsf{x}\ \mathsf{then}\ [\mathsf{x}, \mathsf{inp}]\ \mathsf{else}\ [\,] \\
\_\,\$\,\_ : \forall \{\mathsf{r}\ \mathsf{s}\} \to (\mathsf{r} \to \mathsf{s}) \to \mathsf{Parser\ r} \to \mathsf{Parser\ s} \\
\mathsf{f}\ \$\ \mathsf{p} = \mathsf{succeed}\ \mathsf{f} \star \mathsf{p}
\end{array}
$$

- The combinators are not recursive and thus accepted as total functions by Agda.

Applicative interface – contd.

$$
\begin{array}{l}
\mathsf{symbol} : \mathsf{Char} \rightarrow \mathsf{Parser\ Char} \\
\mathsf{symbol}\ \_\ [\,] \qquad\quad = [\,] \\
\mathsf{symbol}\ x\ (i :: inp) = \mathsf{if}\ i == x\ \mathsf{then}\ [x, inp]\ \mathsf{else}\ [\,] \\
\_\$\_ : \forall \{r\ s\} \rightarrow (r \rightarrow s) \rightarrow \mathsf{Parser}\ r \rightarrow \mathsf{Parser}\ s \\
f\ \$\ p = \mathsf{succeed}\ f \star p
\end{array}
$$

- The combinators are not recursive and thus accepted as total functions by Agda.
- However, nearly all interesting grammars are cyclic, and the resulting combinator parsers recursive:

$$
\begin{array}{l}
\mathsf{sum} : \mathsf{Parser}\ \mathbb{N} \\
\mathsf{sum} = (\lambda m\ \_\ n \rightarrow m + n)\ \$\ \mathsf{nat} \star \mathsf{symbol}\ \text{'+'} \star \mathsf{sum} \\
\qquad\quad \mid\ \mathsf{nat}
\end{array}
$$

Not all parsers terminate

$$\mathsf{nat} : \mathsf{Parser}\ \mathbb{N}$$
$$\mathsf{nat} = (\lambda \mathsf{n}\ \mathsf{d} \rightarrow \mathsf{n} * 10 + \mathsf{d})\ \$\ \mathsf{nat} \star \mathsf{digit}$$
$$\quad\quad |\ \mathsf{digit}$$

Not all parsers terminate

$$
\begin{aligned}
&\mathsf{nat} : \mathsf{Parser}\ \mathbb{N}\\
&\mathsf{nat} = (\lambda \mathsf{n}\ \mathsf{d} \to \mathsf{n} * 10 + \mathsf{d})\ \$\ \mathsf{nat} \star \mathsf{digit}\\
&\qquad \mid \mathsf{digit}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{many} : \forall \{\mathsf{a}\} \to \mathsf{Parser}\ \mathsf{a} \to \mathsf{Parser}\ [\mathsf{a}]\\
&\mathsf{many}\ \mathsf{p} = \_ :: \_\ \$\ \mathsf{p} \star \mathsf{many}\ \mathsf{p}\\
&\qquad\qquad \mid \mathsf{succeed}\ [\,]\\
&\mathsf{optx} : \mathsf{Parser}\ \mathsf{Char}\\
&\mathsf{optx} = \mathsf{symbol}\ \texttt{'x'} \mid \mathsf{succeed}\ \texttt{' '}\\
&\mathsf{optxs} : \mathsf{Parser}\ [\mathsf{Char}]\\
&\mathsf{optxs} = \mathsf{many}\ \mathsf{optx}
\end{aligned}
$$

The rest of this talk

- We will design parser combinators so that the resulting parsers are structurally recursive.
- Left-recursive grammars (directly and indirectly) will be type-incorrect in this library.

Terminating combinator parsers

The main idea

- Look at the following graph: nodes are parsers, an edge from one node to another indicates that a parser can directly call another (without first consuming a symbol).
- For left-recursive grammars (directly or indirectly), the graph contains cycles.
- For other grammars, the graph is cycle-free, and can be expanded into a finite tree.
- If we make this tree an index of the parser type, then left-recursive parsers are no longer type-correct.

Dependency tree

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Corners} : \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{leaf} \quad\ \ : \mathsf{Corners}\\
&\quad \mathsf{node}_1 : \mathsf{Corners} \rightarrow \mathsf{Corners}\\
&\quad \mathsf{node}_2 : \mathsf{Corners} \rightarrow \mathsf{Corners} \rightarrow \mathsf{Corners}\\
&\mathsf{Parser} : \mathsf{Corners} \rightarrow \mathsf{Set} \rightarrow \mathsf{Set}
\end{aligned}
$$

Dependency tree

```
data Corners : Set where
  leaf  : Corners
  node₁ : Corners → Corners
  node₂ : Corners → Corners → Corners
Parser : Corners → Set → Set
```

```
symbol  : Char → Parser leaf Char
succeed : ∀{r}   → r → Parser leaf r
_|_     : ∀{r}   → Parser c₁ r → Parser c₂ r →
                   Parser (node₂ c₁ c₂) r
_⋆_     : ∀{r s} → Parser c₁ (r → s) → Parser c₂ r →
                   Parser ? s
```

Epsilon

It is important to know if a parser accepts the empty word:

$$\mathsf{Empty} : \mathsf{Set}$$
$$\mathsf{Empty} = \mathsf{Bool}$$
$$\mathsf{Parser} : (\mathsf{Empty} \times \mathsf{Corners}) \rightarrow \mathsf{Set} \rightarrow \mathsf{Set}$$

Epsilon

It is important to know if a parser accepts the empty word:

$$\mathsf{Empty} : \mathsf{Set}$$
$$\mathsf{Empty} = \mathsf{Bool}$$
$$\mathsf{Parser} : (\mathsf{Empty} \times \mathsf{Corners}) \rightarrow \mathsf{Set} \rightarrow \mathsf{Set}$$

$$\mathsf{symbol} \ : \mathsf{Char} \rightarrow \mathsf{Parser}\ (\mathsf{false}, \mathsf{leaf})\ \mathsf{Char}$$

Epsilon

It is important to know if a parser accepts the empty word:

$$\mathsf{Empty} : \mathsf{Set}$$
$$\mathsf{Empty} = \mathsf{Bool}$$
$$\mathsf{Parser} : (\mathsf{Empty} \times \mathsf{Corners}) \to \mathsf{Set} \to \mathsf{Set}$$

$$\mathsf{symbol}\ : \mathsf{Char} \to \mathsf{Parser}\ (\mathsf{false}, \mathsf{leaf})\ \mathsf{Char}$$
$$\mathsf{succeed} : \forall \{\, \mathsf{r} \,\} \to \mathsf{r} \to \mathsf{Parser}\ (\mathsf{true}, \mathsf{leaf})\ \mathsf{r}$$

Epsilon

It is important to know if a parser accepts the empty word:

$$\begin{aligned}
&\mathsf{Empty} : \mathsf{Set} \\
&\mathsf{Empty} = \mathsf{Bool} \\
&\mathsf{Parser} : (\mathsf{Empty} \times \mathsf{Corners}) \to \mathsf{Set} \to \mathsf{Set}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{symbol} &&: \mathsf{Char} \to \mathsf{Parser}\ (\mathsf{false}, \mathsf{leaf})\ \mathsf{Char} \\
&\mathsf{succeed} &&: \forall \{ \mathsf{r} \} \to \mathsf{r} \to \mathsf{Parser}\ (\mathsf{true}, \mathsf{leaf})\ \mathsf{r} \\
&_ \mid _ &&: \forall \{ \mathsf{e}_1\ \mathsf{c}_1\ \mathsf{e}_2\ \mathsf{c}_2\ \mathsf{r} \} \to \\
& &&\quad \mathsf{Parser}\ (\mathsf{e}_1, \mathsf{c}_1)\ \mathsf{r} \to \mathsf{Parser}\ (\mathsf{e}_1, \mathsf{c}_2)\ \mathsf{r} \to \\
& &&\quad \mathsf{Parser}\ (\mathsf{e}_1 \vee \mathsf{e}_2, \mathsf{node}_2\ \mathsf{c}_1\ \mathsf{c}_2)\ \mathsf{r}
\end{aligned}$$

Epsilon

It is important to know if a parser accepts the empty word:

$$
\begin{aligned}
&\mathsf{Empty} : \mathsf{Set}\\
&\mathsf{Empty} = \mathsf{Bool}\\
&\mathsf{Parser} : (\mathsf{Empty} \times \mathsf{Corners}) \to \mathsf{Set} \to \mathsf{Set}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{symbol} && : \mathsf{Char} \to \mathsf{Parser}\ (\mathsf{false}, \mathsf{leaf})\ \mathsf{Char}\\
&\mathsf{succeed} && : \forall \{ r \} \to r \to \mathsf{Parser}\ (\mathsf{true}, \mathsf{leaf})\ r\\
&\_\mid\_ && : \forall \{ e_1\ c_1\ e_2\ c_2\ r \} \to\\
& && \quad \mathsf{Parser}\ (e_1, c_1)\ r \to \mathsf{Parser}\ (e_1, c_2)\ r \to\\
& && \quad \mathsf{Parser}\ (e_1 \vee e_2, \mathsf{node}_2\ c_1\ c_2)\ r\\
&\_\star\_ && : \forall \{ e_1\ c_1\ e_2\ c_2\ r\ s \} \to\\
& && \quad \mathsf{Parser}\ (e_1, c_1)\ (r \to s) \to \mathsf{Parser}\ (e_1, c_2)\ r \to\\
& && \quad \mathsf{Parser}\ (e_1 \wedge e_2,\ \mathsf{if}\ e_1\ \mathsf{then}\ \mathsf{node}_2\ c_1\ c_2\ \mathsf{else}\ c_1)\ s
\end{aligned}
$$

Not done

- What about

$$\text{Parser} : (\text{Empty} \times \text{Corners}) \rightarrow \text{Set} \rightarrow \text{Set}$$

If, as before

$$\text{Parser}\ \_\ \text{r} = \text{Input} \rightarrow [\text{r} \times \text{Input}]$$

then the index information is lost!

Not done

- What about

  $$\mathsf{Parser} : (\mathsf{Empty} \times \mathsf{Corners}) \to \mathsf{Set} \to \mathsf{Set}$$

  If, as before

  $$\mathsf{Parser}\ \_\ \mathsf{r} = \mathsf{Input} \to [\mathsf{r} \times \mathsf{Input}]$$

  then the index information is lost!
- We have to turn Parser into an abstract datatype:

  $$\textbf{data}\ \mathsf{Parser} : (\mathsf{Empty} \times \mathsf{Corners}) \to \mathsf{Set} \to \mathsf{Set}\ \textbf{where}$$
  $$\quad \ldots$$

Not done – contd.

- Recursive definitions still pose a problem.
- Does not pass the termination checker, but still type-correct:

$$\mathsf{p} : \mathsf{Parser}\ (\mathsf{true}, \mathsf{leaf})\ \mathsf{Char}$$
$$\mathsf{p} = \mathsf{p}$$

Not done – contd.

- Recursive definitions still pose a problem.
- Does not pass the termination checker, but still type-correct:

$$p : \mathsf{Parser}\ (\mathsf{true}, \mathsf{leaf})\ \mathsf{Char}$$
$$p = p$$

- Recursion must change the Corners tree!

$$!\_ : \forall \{e\ c\ r\} \rightarrow \mathsf{Parser}\ (e, c)\ r \rightarrow \mathsf{Parser}\ (e, \mathsf{node}_1\ c)\ r$$

Recursion via ! fails the "occurs check":

$$p = !p$$

Not done – contd.

- Legal cyclic definitions are still far from structurally recursive:

  $$\begin{array}{l} \mathsf{p} : \mathsf{Parser} \ldots \\ \mathsf{p} = \ldots \mathsf{p} \ldots \end{array}$$

- Turn parsers (thus Corners) into function **arguments**.
- This unfortunately has quite a few implications: we turn parser combinators and also the nonterminals of grammars into datatypes, so that we can perform case analysis in a function.

Abstract parsers

$$\mathsf{ParserType} = (\mathsf{Empty} \times \mathsf{Corners}) \rightarrow \mathsf{Set} \rightarrow \mathsf{Set}_1$$

$$\begin{array}{l}
\textbf{data}\ \mathsf{Parser}\ (\mathsf{nt} : \mathsf{ParserType}) : \mathsf{ParserType}\ \textbf{where} \\
\quad \begin{array}{ll}
!_ & : \forall\{\mathsf{e}\ \mathsf{c}\ \mathsf{r}\} \rightarrow \\
 & \quad \mathsf{nt}\ (\mathsf{e}, \mathsf{c})\ \mathsf{r} \rightarrow \mathsf{Parser}\ \mathsf{nt}\ (\mathsf{e}, \mathsf{node}_1\ \mathsf{c})\ \mathsf{r} \\
\mathsf{symbol} & : \mathsf{Char} \rightarrow \mathsf{Parser}\ \mathsf{nt}\ (\mathsf{false}, \mathsf{leaf})\ \mathsf{Char} \\
\mathsf{return} & : \forall\{\mathsf{r}\} \rightarrow \mathsf{r} \rightarrow \mathsf{Parser}\ \mathsf{nt}\ (\mathsf{true}, \mathsf{leaf})\ \mathsf{r} \\
\ldots &
\end{array}
\end{array}$$

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Grammars

```
Grammar : ParserType → Set₁
Grammar nt = ∀{e c r} → nt (e, c) r → Parser nt (e, c) r
```

```
data NT : ParserType where
  nat  : NT (_, _) ℕ   -- indices can be inferred!
  sum  : NT (_, _) ℕ

grammar : Grammar NT
grammar nat  = (const 1) $ sym '1'    -- simplified
grammar sum  = (λm _ n → m + n) $ !nat ⋆ symbol '+' ⋆ !sum
             | !nat
```

Grammars

$$\mathsf{Grammar} : \mathsf{ParserType} \to \mathsf{Set}_1$$
$$\mathsf{Grammar}\ \mathsf{nt} = \forall \{\mathsf{e}\ \mathsf{c}\ \mathsf{r}\} \to \mathsf{nt}\ (\mathsf{e}, \mathsf{c})\ \mathsf{r} \to \mathsf{Parser}\ \mathsf{nt}\ (\mathsf{e}, \mathsf{c})\ \mathsf{r}$$

data NT : ParserType **where**

```
   nat : NT (_, _) ℕ    -- indices can be inferred!
   sum : NT (_, _) ℕ
grammar : Grammar NT
grammar nat  = (const 1) $ sym '1'    -- simplified
grammar sum  = (λm _ n → m + n) $ !nat ⋆ symbol '+' ⋆ !sum
             | !nat
```

The definition of grammar is type correct if no left-recursion is involved. It is no longer recursive.

Interpreting the parsers

$$
\begin{array}{l}
\mathsf{parse} : \{\mathsf{nt} : \mathsf{ParserType}\}(\mathsf{g} : \mathsf{Grammar\ nt}) \\
\qquad\quad \{\mathsf{e} : \mathsf{Empty}\}\{\mathsf{c} : \mathsf{Corners}\}\{\mathsf{r} : \mathsf{Set}\} \rightarrow \\
\qquad\quad \mathsf{Parser\ nt\ (e, c)\ r} \rightarrow \\
\qquad\quad \mathsf{LoS.Parser\ r} \quad \text{-- original parser type}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{parse\ g\ (!p)} & = & \mathsf{parse\ g\ (g\ p)} \\
\mathsf{parse\ g\ (symbol\ c)} & = & \mathsf{LoS.symbol\ c} \\
\mathsf{parse\ g\ (p_1 \mid p_2)} & = & \mathsf{LoS.\_\mid\_\ (parse\ g\ p_1)\ (parse\ g\ p_2)} \\
\mathsf{parse\ g\ (p_1 \star p_2)} & = & \mathsf{LoS.\_\star\_\ (parse\ g\ p_1)\ (parse\ g\ p_2)}
\end{array}
$$

Interpreting the parsers

$$
\begin{array}{l}
\mathsf{parse} : \{\mathsf{nt} : \mathsf{ParserType}\}(\mathsf{g} : \mathsf{Grammar\ nt}) \\
\qquad\qquad \{\mathsf{e} : \mathsf{Empty}\}\{\mathsf{c} : \mathsf{Corners}\}\{\mathsf{r} : \mathsf{Set}\} \to \\
\qquad\qquad \mathsf{Parser\ nt\ (e, c)\ r} \to \\
\qquad\qquad \mathsf{LoS.Parser\ r} \quad \text{-- original parser type}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{parse\ g\ (!p)} & = & \mathsf{parse\ g\ (g\ p)} \\
\mathsf{parse\ g\ (symbol\ c)} & = & \mathsf{LoS.symbol\ c} \\
\mathsf{parse\ g\ (p_1 \mid p_2)} & = & \mathsf{LoS.\_ \mid \_\ (parse\ g\ p_1)\ (parse\ g\ p_2)} \\
\mathsf{parse\ g\ (p_1 \star p_2)} & = & \mathsf{LoS.\_ \star \_\ (parse\ g\ p_1)\ (parse\ g\ p_2)}
\end{array}
$$

Is this definition structurally recursive?

Interpreting the parsers

```
parse : {nt : ParserType}(g : Grammar nt)
        {e : Empty}{c : Corners}{r : Set} →
        Parser nt (e, c) r →
        LoS.Parser r    -- original parser type
parse g (!p)          = parse g (g p)
parse g (symbol c)    = LoS.symbol c
parse g (p₁ | p₂)     = LoS._|_ (parse g p₁) (parse g p₂)
parse g (p₁ ⋆ p₂)     = LoS._⋆_ (parse g p₁) (parse g p₂)
```

Is this definition structurally recursive?

Interpreting the parsers

```
parse : {nt : ParserType}(g : Grammar nt)
        {e : Empty}{c : Corners}{r : Set} →
        Parser nt (e, c) r →
        LoS.Parser r   -- original parser type
parse g (!p)          = parse g (g p)
parse g (symbol c)    = LoS.symbol c
parse g (p₁ | p₂)     = LoS._|_ (parse g p₁) (parse g p₂)
parse g (p₁ ⋆ p₂)     = LoS._⋆_ (parse g p₁) (parse g p₂)
```

Is this definition structurally recursive?

No, in the ! case, the structure of the parser can get larger; in the $\star$ case, p_2 can have a large Corners tree.

A final refinement

We refine the Input type to keep an upper bound of the length of the input string:

$$\mathsf{Input} : \mathbb{N} \to \mathsf{Set}$$
$$\mathsf{Input}\ \mathsf{n} = \mathsf{BoundedVec}\ \mathsf{Char}\ \mathsf{n}$$
$$\mathsf{Parser} : \mathbb{N} \to \mathbb{N} \to \mathsf{Set} \to \mathsf{Set}$$
$$\mathsf{Parser}\ \mathsf{m}\ \mathsf{n}\ \mathsf{r} = \mathsf{Input}\ \mathsf{m} \to [\mathsf{r} \times \mathsf{Input}\ \mathsf{n}]$$

Adapting parse

$$
\begin{aligned}
&\mathsf{parse} : \{\mathsf{nt} : \mathsf{ParserType}\}(\mathsf{g} : \mathsf{Grammar}\ \mathsf{nt})\\
&\qquad (\mathsf{n} : \mathbb{N})\{\mathsf{e} : \mathsf{Empty}\}\{\mathsf{c} : \mathsf{Corners}\}\{\mathsf{r} : \mathsf{Set}\} \to\\
&\qquad \mathsf{Parser}\ \mathsf{nt}\ (\mathsf{e}, \mathsf{c})\ \mathsf{r} \to\\
&\qquad \mathsf{LoS.Parser}\ \mathsf{n}\ (\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{n}\ \mathsf{else}\ \mathsf{pred}\ \mathsf{n})\ \mathsf{r}
\end{aligned}
$$

...

Adapting parse

```
parse : {nt : ParserType}(g : Grammar nt)
        (n : ℕ){e : Empty}{c : Corners}{r : Set} →
        Parser nt (e, c) r →
        LoS.Parser n (if e then n else pred n) r

...
parse g n        (_⋆_ {e₁ = true} p₁ p₂)
   = LoS._⋆_ (parse g n p₁) (parse g n p₂)
        -- ok because p₁ and p₂ have a smaller Corners tree
```

Adapting parse

```
parse : {nt : ParserType}(g : Grammar nt)
        (n : ℕ){e : Empty}{c : Corners}{r : Set} →
        Parser nt (e, c) r →
        LoS.Parser n (if e then n else pred n) r

...
parse g n        (_⋆_ {e₁ = true} p₁ p₂)
   = LoS._⋆_ (parse g n p₁) (parse g n p₂)
        -- ok because p₁ and p₂ have a smaller Corners tree
parse g 0        (_⋆_ {e₁ = false}p₁ p₂)
   = LoS.fail
```

Adapting parse

```
parse : {nt : ParserType}(g : Grammar nt)
        (n : ℕ){e : Empty}{c : Corners}{r : Set} →
        Parser nt (e, c) r →
        LoS.Parser n (if e then n else pred n) r

...
parse g n        (_⋆_ {e₁ = true} p₁ p₂)
   = LoS._⋆_ (parse g n p₁) (parse g n p₂)
         -- ok because p₁ and p₂ have a smaller Corners tree
parse g 0        (_⋆_ {e₁ = false}p₁ p₂)
   = LoS.fail
parse g (suc n) (_⋆_ {e₁ = false}p₁ p₂)
   = LoS._⋆_ (parse g (suc n) p₁) (parse↑ g n p₂)
parse↑ : ··· →    -- like parse, but results in ...
         LoS.Parser n n r
```

Summary

- We have shown that structurally recursive parser combinators can be implemented in Agda.
- Parsers written using this library are total. Left-recursive grammars (whether directly or indirectly) are rejected at compilation time.
- More work for the implementor, not much more work for the user, except . . .
- Defining reusable recursive derived combinators (e.g. many) requires a bit of additional trickery.
- The indices (Empty and Corners) can usually be inferred.
- Efficiency in current implementations is not too good, but in principle, not much overhead is involved – most of the indices are irrelevant at run time and can be eliminated.

Advertisement

Interested in Agda?

Try the seminar on

"Dependently Typed Programming"

(INFOMDTP)

in block 1 of 2008/2009.