Total Parser Combinators

Nils Anders Danielsson (Nottingham)

ICFP, Baltimore, 2010-09-29

Total Parser Combinators *Using Mixed Induction and Coinduction*

Nils Anders Danielsson (Nottingham)

ICFP, Baltimore, 2010-09-29

Introduction

$$\begin{array}{llll}
\mathit{expr} & = & \mathit{number} & \\
 & | & \sharp\ \mathit{expr} & \gg\!= \ \lambda\ n_1 \ \to \\
 & & \mathit{tok}\ \texttt{'+'} & \gg\!= \ \lambda\ \_ \ \to \\
 & & \mathit{number} & \gg\!= \ \lambda\ n_2 \ \to \\
 & & \mathit{return}\ (n_1 + n_2) &
\end{array}$$

Introduction

$$\begin{array}{lcl} \mathit{expr} & = & \sharp\ \mathit{expr} \cdot \mathit{tok}\ \texttt{'+'} \cdot \mathit{number} \\ & \mid & \mathit{number} \end{array}$$

Introduction

$$\begin{array}{rcl} \mathit{expr} & = & \sharp\ \mathit{expr} \cdot \mathit{tok}\ \text{'+'} \cdot \mathit{number} \\ & \mid & \mathit{number} \end{array}$$

Introduction

$$\begin{array}{lcl} \textit{expr} & = & \sharp\ \textit{expr} \cdot \textit{tok}\ \texttt{'+'} \cdot \textit{number} \\ & \mid & \textit{number} \end{array}$$

- Left recursive.

Introduction

$$\begin{aligned} \mathit{expr} \;&=\; \sharp\, \mathit{expr} \;\cdot\; \mathit{tok}\ \text{'+'} \;\cdot\; \mathit{number} \\ &\mid\; \mathit{number} \end{aligned}$$

- Left recursive.
- Parsing is guaranteed to terminate (for finite input strings).

Introduction

$$
\begin{aligned}
\mathit{expr} &= \sharp\, \mathit{expr} \cdot \mathit{tok}\ \text{'+'} \cdot \mathit{number} \\
&\mid \mathit{number}
\end{aligned}
$$

Key idea

Control the shape of parsers using a careful combination of induction and coinduction.

Introduction

$$\begin{aligned} \mathit{expr} &= \sharp\,(\sharp\,\mathit{expr}\;\cdot\;\sharp\,(\mathit{tok}\ \text{'+'}))\;\cdot\;\sharp\,\mathit{number} \\ &\mid\ \mathit{number} \end{aligned}$$

Key idea

Control the shape of parsers using a careful combination of induction and coinduction.

Interface (roughly)

P : Set

empty : P
tok : $Token \to P$
_|_ : $P \to P \to P$
_·_ : $P \to P \to P$

Cyclic definitions

Want to allow cyclic definitions:

$$\begin{aligned} \mathit{zeros} &= \mathsf{tok}\ \texttt{'0'} \cdot \mathit{zeros} \\ &\mid\ \mathsf{empty} \end{aligned}$$

But not all of them:

$$\mathit{bad} = \mathit{bad} \mid \mathit{bad}$$

Solution: Make parsers partly inductive, partly coinductive.

Mixed induction and coinduction

Inductive types

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

Structural recursion:

```
length : List A → ℕ
length []        = zero
length (x :: xs) = suc (length xs)
```

Coinductive types

$$
\begin{array}{l}
\textbf{data}\ \mathit{Stream}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where} \\
\quad \_{::}\_\ :\ A\ \to\ \infty\,(\mathit{Stream}\ A)\ \to\ \mathit{Stream}\ A
\end{array}
$$

- ∞ marks coinductive arguments.
- Can be seen as a suspension.
- Delay and force:

$$
\begin{array}{lcrcr}
\sharp & : & A & \to & \infty\, A \\
\flat & : & \infty\, A & \to & A
\end{array}
$$

Coinductive types

$$\textbf{data}\ \mathit{Stream}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}$$
$$\quad \_{::}\_\ :\ A\ \rightarrow\ \infty\,(\mathit{Stream}\ A)\ \rightarrow\ \mathit{Stream}\ A$$

Guarded corecursion:

$$\mathit{infinite}\ :\ \mathit{Stream}\ \mathbb{N}$$
$$\mathit{infinite}\ =\ \mathrm{zero}\ ::\ \sharp\ \mathit{infinite}$$

Mixed induction and coinduction

$SP\ A\ B$ represents functions of type $Stream\ A \rightarrow Stream\ B$:

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

$$SP\ A\ B \approx \nu C.\, \mu I.\, ((A \rightarrow I) + B \times C)$$

Mixed induction and coinduction

Not OK:

$$\begin{aligned} \mathit{sink} &: \mathit{SP}\ A\ B \\ \mathit{sink} &= \mathrm{get}\ (\lambda\ \_\ \to\ \mathit{sink}) \end{aligned}$$

OK:

$$\begin{aligned} \mathit{copy} &: \mathit{SP}\ A\ A \\ \mathit{copy} &= \mathrm{get}\ (\lambda\ x\ \to\ \mathrm{put}\ x\ (\sharp\ \mathit{copy})) \end{aligned}$$

Mixed induction and coinduction

Lexicographic guarded corecursion and higher-order structural recursion:

$$
\begin{array}{lcl}
[\![\_]\!] & : & SP\ A\ B \ \rightarrow\ Stream\ A \ \rightarrow\ Stream\ B \\
[\![\ \mathsf{get}\ f\ ]\!]\ (a :: as) & = & [\![\ f\ a\ ]\!]\ (\flat\ as) \\
[\![\ \mathsf{put}\ b\ sp\ ]\!]\ as & = & b :: \sharp\ ([\![\ \flat\ sp\ ]\!]\ as)
\end{array}
$$

Mixed induction and coinduction

Lexicographic guarded corecursion and higher-order structural recursion:

$$
\begin{array}{lcl}
[\![\_]\!] \ : \ \mathit{SP}\ A\ B & \rightarrow & \mathit{Stream}\ A \ \rightarrow \ \mathit{Stream}\ B \\
[\![\ \mathsf{get}\ f \quad]\!]\ (a :: \mathit{as}) & = & [\![\ f\ \ a\ ]\!]\ (\flat\ \mathit{as}) \\
[\![\ \mathsf{put}\ b\ \mathit{sp}\ ]\!]\ \mathit{as} & = & b :: \sharp\ ([\![\ \flat\ \mathit{sp}\ ]\!]\ \mathit{as})
\end{array}
$$

Assume that get were coinductive:

- *sink* would be accepted.
- $[\![\_]\!]$ would not be productive.

Back to the parser combinators

Choice

Hard to decide infinite choice:

$$\mathit{bad} \;=\; \mathit{bad} \mid \mathit{bad}$$

The arguments of $\_|\_$ will be inductive.

Sequencing

Problematic if p′ is nullable, otherwise OK:

$$p \;=\; p \cdot p' \qquad\qquad \frac{s_1 \;\in\; p \qquad t :: s_2 \;\in\; p'}{s_1 \mathbin{+\!\!+} t :: s_2 \;\in\; p}$$

Allow the first argument to be coinductive if the second does not accept the empty string, and vice versa.

Sequencing

Problematic if p′ is nullable, otherwise OK:

$$p \;=\; p \cdot p'$$

$$\frac{s_1 \;\in\; p \qquad t :: s_2 \;\in\; p'}{s_1 +\!\!+\; t :: s_2 \;\in\; p}$$

Allow the first argument to be coinductive if the second does not accept the empty string, and vice versa.

Sequencing

Problematic if p′ is nullable, otherwise OK:

$$p \;=\; p \cdot p'$$

$$\frac{s_1 \;\in\; p \qquad t :: s_2 \;\in\; p'}{s_1 +\!\!+\; t :: s_2 \;\in\; p}$$

Allow the first argument to be coinductive if the second does not accept the empty string, and vice versa.

Sequencing

Problematic if p$'$ is nullable, otherwise OK:

$$p \;=\; p \cdot p'$$

$$\frac{s_1 \;\in\; p \qquad t :: s_2 \;\in\; p'}{s_1 +\!\!+\, t :: s_2 \;\in\; p}$$

Allow the first argument to be coinductive if the second does not accept the empty string, and vice versa.

Nullability index

Let us index parsers by their nullability:

- $P \; : \; \mathit{Bool} \; \rightarrow \; \mathit{Set}$.
- $p \; : \; P$ true if p accepts the empty string.
- $p \; : \; P$ false otherwise.

Interface

```
mutual
  data P : Bool → Set where
    empty : P true
    tok   : Token → P false
    _|_   : P n₁ → P n₂ → P (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
```

Examples

OK:

$$\begin{array}{lcl} zeros & = & \text{tok } \texttt{'0'} \cdot \sharp\; zeros \\ & \mid & \text{empty} \end{array}$$

Not OK:

$$\begin{array}{lcll} bad & = & bad \mid bad & \text{-- Not guarded.} \\ void & = & \text{empty} \cdot void & \text{-- Not guarded.} \end{array}$$

Examples

OK:

$$\begin{aligned} \mathit{zeros} \;&=\; \sharp\ \mathit{zeros} \cdot \mathsf{tok}\ \texttt{'0'} \\ &\mid\; \mathsf{empty} \end{aligned}$$

Not OK:

$$\begin{aligned} \mathit{bad} \;&=\; \mathit{bad} \mid \mathit{bad} && \text{-- Not guarded.} \\ \mathit{void} \;&=\; \mathsf{empty} \cdot \mathit{void} && \text{-- Not guarded.} \end{aligned}$$

Examples

OK:

$$\begin{aligned} \mathit{zeros} \quad &= \quad \sharp\ \mathit{zeros} \cdot \mathrm{tok}\ \texttt{'0'} \\ &\mid \quad \mathrm{empty} \end{aligned}$$

Not OK:

$$\begin{aligned} \mathit{bad} \quad &= \quad \mathit{bad} \mid \mathit{bad} && \text{-- Not guarded.} \\ \mathit{void} \quad &= \quad \mathrm{empty} \cdot \sharp\ \mathit{void} && \text{-- Not type correct.} \end{aligned}$$

Example

Kleene star:

mutual

$$\_\star : P\ \mathsf{false} \to P\ \mathsf{true}$$

$$p\,\star = \mathsf{empty} \mid p\,+$$

$$\_+ : P\ \mathsf{false} \to P\ \mathsf{false}$$

$$p\,+ = p \cdot \sharp\,(p\,\star)$$

The argument must not accept the empty string: the infinitely ambiguous parser empty $\star$ is not accepted.

See the paper/code. . .

How is parsing implemented?

- Breadth-first algorithm:
  Treat one token at a time,
  compute residual recogniser using
  Brzozowski derivatives.
- Combination of corecursion and recursion.
- Inefficient. Can we do better?

See the paper/code. . .

What about expressiveness?

- The parsers are as expressive as possible.
- This talk:
  Every decidable language over a
  finite alphabet can be recognised.
- Full parser combinators:
  Every finitely ambiguous
  decidable language can be parsed.

See the paper/code. . .

Formal semantics, algebraic laws,
mechanised correctness proofs.

Conclusions

- Mixed induction/coinduction:
  Precise control of size of data.
- I encourage you to add this technique to your toolbox.

Conclusions

- Mixed induction/coinduction:
  Precise control of size of data.
- I encourage you to add this technique
  to your toolbox.

?

Bonus slides

Infinite ambiguity

Parser combinators:

return x	The empty string. Result: x.
$p \ggg f$	First p, then f applied to result of p.
fail	Always fails.

- $(\mathit{return}\ \mathsf{unit}) \star \quad \mapsto \quad [\,], [\mathsf{unit}], [\mathsf{unit}, \mathsf{unit}], \ldots$
- $(\mathit{return}\ \mathsf{unit}) \star \ggg \lambda\, \mathit{xs} \to$
  $\mathbf{if}\ f\ \mathit{xs}\ \mathbf{then}\ \mathit{return}\ \mathsf{unit}\ \mathbf{else}\ \mathit{fail} \quad \mapsto \quad ?$

Semantics

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \_\in\_\ :\ \mathit{List}\ \mathit{Token}\ \to\ P\ n\ \to\ \mathit{Set}\ \textbf{where}} \\
\quad \textsc{Empty} & : & [\,]\ \in\ \mathsf{empty} \\
\quad \textsc{Tok} & : & [\,t\,]\ \in\ \mathsf{tok}\ t \\
\quad \textsc{AltL} & : & s\ \in\ p_1\ \to\ s\ \in\ p_1\ |\ p_2 \\
\quad \textsc{AltR} & : & s\ \in\ p_2\ \to\ s\ \in\ p_1\ |\ p_2 \\
\quad \textsc{Seq} & : & s_1\ \in\ \flat?\ p_1\ \to\ s_2\ \in\ \flat?\ p_2\ \to \\
 & & s_1 +\!\!+\ s_2\ \in\ p_1 \cdot p_2
\end{array}$$

Backend

$$\_{\in}?\_ : (s : \mathit{List}\ \mathit{Token})\ (p : P\ n) \rightarrow \mathit{Dec}\ (s \in p)$$

- Breadth-first algorithm:
  Treat one token at a time,
  compute residual recogniser.
- $D : (t : \mathit{Token})\ (p : P\ n) \rightarrow P\ (\mathit{D\text{-}null?}\ t\ p)$.
- $t :: s \in p \Leftrightarrow s \in D\ t\ p$.
- A variant of Brzozowski's
  regular expression derivatives.

Backend

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

$$\begin{array}{lcr}
\texttt{"abc"} & \in & p \Leftrightarrow \\
\texttt{"bc"} & \in & D\ \texttt{'a'}\ p \Leftrightarrow \\
\texttt{"c"} & \in & D\ \texttt{'b'}\ (D\ \texttt{'a'}\ p) \Leftrightarrow \\
\texttt{""} & \in & D\ \texttt{'c'}\ (D\ \texttt{'b'}\ (D\ \texttt{'a'}\ p))
\end{array}$$

Backend

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

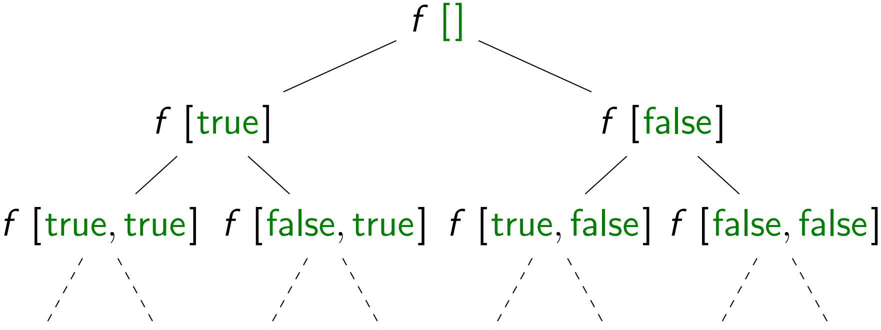
$$\begin{array}{lcr}
\texttt{"abc"} & \in & p \Leftrightarrow \\
\texttt{"bc"} & \in & D\ \texttt{'a'}\ p \Leftrightarrow \\
\texttt{"c"} & \in & D\ \texttt{'b'}\ (D\ \texttt{'a'}\ p) \Leftrightarrow \\
\texttt{""} & \in & D\ \texttt{'c'}\ (D\ \texttt{'b'}\ (D\ \texttt{'a'}\ p))
\end{array}$$

Can be very inefficient. Open question:
Can an efficient backend be implemented?

Expressive strength

For finite alphabets the combinators are as expressive as possible:

$f \;:\; \mathit{List\ Bool} \;\to\; \mathit{Bool}$



f []

f [true] f [false]

f [true, true] f [false, true] f [true, false] f [false, false]

Expressive strength

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

$$\begin{aligned} &\mathit{accept\text{-}if\text{-}true} \;:\; (b \;:\; \mathit{Bool}) \;\to\; P\ b \\ &\mathit{accept\text{-}if\text{-}true}\ \mathsf{true} \;=\; \mathsf{empty} \\ &\mathit{accept\text{-}if\text{-}true}\ \mathsf{false} \;=\; \mathit{fail} \end{aligned}$$

$$\begin{aligned} &\sharp? \;:\; P\ n \;\to\; \infty\langle\ b\ \rangle P\ n \\ &\sharp?\ \{b \;=\; \mathsf{false}\}\ x \;=\; \sharp\ x \\ &\sharp?\ \{b \;=\; \mathsf{true}\}\ x \;=\; \ x \end{aligned}$$

Expressive strength

$$
\begin{array}{l}
p \;:\; (f \;:\; \mathit{List}\ \mathit{Bool} \;\rightarrow\; \mathit{Bool}) \;\rightarrow\; P\,(f\;[\,]) \\
p\,f \quad = \\
\qquad\quad \sharp\,(p\,(\lambda\mathit{xs} \;\rightarrow\; f\;(\mathit{xs} +\!\!+ [\mathsf{true}]))) \cdot \sharp?\;(\mathsf{tok\ true}) \\
\qquad \mid\; \sharp\,(p\,(\lambda\mathit{xs} \;\rightarrow\; f\;(\mathit{xs} +\!\!+ [\mathsf{false}]))) \cdot \sharp?\;(\mathsf{tok\ false}) \\
\qquad \mid\; \mathit{accept\text{-}if\text{-}true}\,(f\;[\,])
\end{array}
$$

$$s \;\in\; p\,f \;\;\Leftrightarrow\;\; f\;s \equiv \mathsf{true}$$

Laws

- The combinators form a Kleene algebra.
- Need to generalise the Kleene star:

$$\begin{aligned} \_\bigstar &: P\ n \ \rightarrow\ P\ \textsf{true} \\ p\,\bigstar &=\ (\mathit{nonempty}\ p) \star \end{aligned}$$

- The *nonempty* combinator can be defined by structural recursion:

$$\mathit{nonempty}\ :\ P\ n\ \rightarrow\ P\ \textsf{false}$$