Total Parser Combinators Using Mixed Induction and Coinduction

Nils Anders Danielsson (Nottingham)

IFIP WG2.1 Meeting #66, Atlantic City, 2010

Introduction

Mixed induction/coinduction:

- Precise control of size of data.
- Example: parser combinators.

Parser combinator example

BNF:

$$
\begin{aligned}
\mathit{term} &::= \mathit{term}\ \text{'+'}\ \mathit{atom} \\
\mathit{term} &::= \mathit{atom} \\
\mathit{atom} &::= \text{'('}\ \mathit{term}\ \text{')'} \\
\mathit{atom} &::= \mathit{number}
\end{aligned}
$$

Parser combinators:

mutual

$$
\begin{aligned}
\mathit{term} &= \mathit{term} \cdot \mathit{tok}\ \text{'+'} \cdot \mathit{atom} \\
&\mid \mathit{atom} \\
\mathit{atom} &= \mathit{tok}\ \text{'('} \cdot \mathit{term} \cdot \mathit{tok}\ \text{')'} \\
&\mid \mathit{number}
\end{aligned}
$$

Parser combinator example

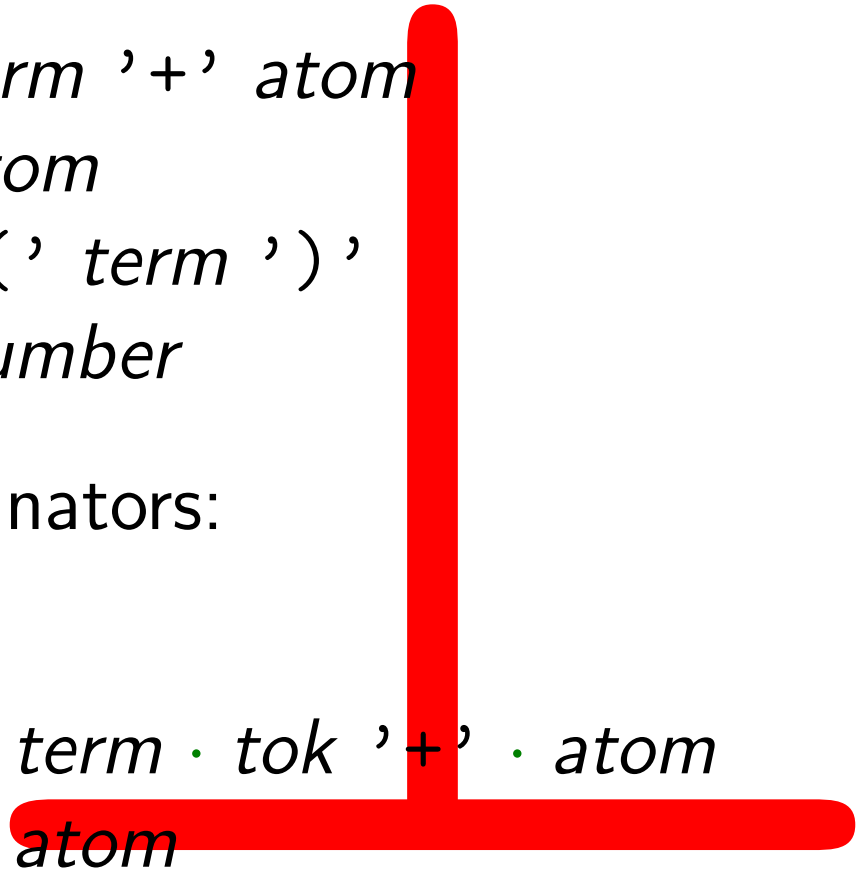
BNF:

$$
\begin{aligned}
\mathit{term} &::= \mathit{term}\ \text{'+'}\ \mathit{atom} \\
\mathit{term} &::= \mathit{atom} \\
\mathit{atom} &::= \text{'('}\ \mathit{term}\ \text{')'} \\
\mathit{atom} &::= \mathit{number}
\end{aligned}
$$

Parser combinators:

mutual

$$
\begin{aligned}
\mathit{term} &= \mathit{term} \cdot \mathit{tok}\ \text{'+'} \cdot \mathit{atom} \\
&\mid \mathit{atom} \\
\mathit{atom} &= \mathit{tok}\ \text{'('} \cdot \mathit{term} \cdot \mathit{tok}\ \text{')'} \\
&\mid \mathit{number}
\end{aligned}
$$



Mixed induction and coinduction

Inductive types

$$\begin{array}{l}\textbf{data}\ \mathit{List}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad [] \quad\ :\ \mathit{List}\ A\\ \quad \_::\_\ :\ A\ \to\ \mathit{List}\ A\ \to\ \mathit{List}\ A\end{array}$$

List A can be seen as a fixpoint:

$$\mathit{List}\ A\ =\ 1 + A \times \mathit{List}\ A$$

Read $T\ =\ F\ T$ as $T\ =\ \mu I.\ F\ I$:

$$\mathit{List}\ A\ =\ \mu I.\ 1 + A \times I$$

Inductive types

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


finite : List ℕ
finite = 0 :: 0 :: 0 :: []


map : (A → B) → List A → List B
map f []       = []
map f (x :: xs) = f x :: map f xs
```

Coinductive types

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

Stream A can be seen as a fixpoint:

$$\mathit{Stream}\ A\ =\ A \times \infty\ (\mathit{Stream}\ A)$$

Read $T\ =\ F\ (\infty\ T)$ as $T\ =\ \nu C.\ F\ C$:

$$\mathit{Stream}\ A\ =\ \nu C.\ A \times C$$

Coinductive types

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

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

$$\sharp\ :\quad A\ \to\ \infty\ A$$
$$\flat\ :\ \infty\ A\ \to\quad A$$

Coinductive types

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

```
infinite : Stream ℕ
infinite = 0 :: ♯ infinite
```

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

Mixed induction and coinduction

$$\begin{aligned}&\textbf{data}\ SP\ (A\ B\ :\ Set)\ :\ Set\ \textbf{where}\\ &\quad \mathrm{get}\ :\ (A\ \to\ SP\ A\ B)\ \ \ \ \ \ \to\ SP\ A\ B\\ &\quad \mathrm{put}\ :\ B\ \to\ \infty\,(SP\ A\ B)\ \to\ SP\ A\ B\end{aligned}$$

Read $T\ =\ F\ T\ (\infty\ T)$ as $T\ =\ \nu C.\ \mu I.\ F\ I\ C$:

$$SP\ A\ B\ =\ \nu C.\ \mu I.\ (A\ \to\ I) + B \times C$$

Mixed induction and coinduction

Not OK:

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

OK:

$$
\begin{aligned}
&\mathit{const} \;:\; B \;\to\; \mathit{SP}\ A\ B \\
&\mathit{const}\ x \;=\; \mathsf{put}\ x\ (\sharp\ (\mathit{const}\ x))
\end{aligned}
$$

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

Mixed induction and coinduction

Lexicographic guarded corecursion and higher-order structural recursion:

$$
\begin{aligned}
&[\![\_]\!] \;:\; \mathit{SP}\ A\ B \;\to\; \mathit{Stream}\ A \;\to\; \mathit{Stream}\ B \\
&[\![\ \mathrm{get}\ f \quad\ ]\!]\ (a :: as) \;=\; [\![\ f\ a\ ]\!]\ (\flat\ as) \\
&[\![\ \mathrm{put}\ b\ sp\ ]\!]\ as \qquad\quad\ \;=\; b :: \sharp\ ([\![\ \flat\ sp\ ]\!]\ as)
\end{aligned}
$$

Parser combinators

Recogniser combinators

Interface (roughly)

$$
\begin{aligned}
&P \;\; : \; Set \\
&\_\in\_ \;\; : \; List\ Token \;\rightarrow\; P \;\rightarrow\; Set \\
&\_\in?\_ \;\; : \; (s \; : \; List\ Token)\,(p \; : \; P) \;\rightarrow\; Dec\,(s \;\in\; p)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{data}\ Dec\ (A \; : \; Set) \; : \; Set\ \textbf{where} \\
&\quad \mathsf{yes} \; : \quad A \;\rightarrow\; Dec\ A \\
&\quad \mathsf{no} \;\; : \neg\, A \;\rightarrow\; Dec\ A
\end{aligned}
$$

Interface (roughly)

$$
\begin{array}{lll}
\mathsf{fail} & : & P \\
\mathsf{empty} & : & P \\
\mathsf{sat} & : & (\mathit{Token} \to \mathit{Bool}) \to P \\
\_|\_ & : & P \to P \to P \\
\_\cdot\_ & : & P \to P \to P
\end{array}
$$

Some arguments should be coinductive.

Choice

Hard to decide infinite choice:

$$p = p \mid p'$$
$$p = p' \mid p$$

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

Sequencing

Problematic if p' is nullable, otherwise OK:

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

Let us index parsers by their nullability:

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

Recognisers

```
mutual
  data P : Bool → Set where
    fail  : P false
    empty : P true
    sat   : (Token → Bool) → 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:

$$\mathit{anything} \;=\; \mathsf{sat}\;(\lambda\;\_\;\rightarrow\;\mathsf{true}) \cdot \sharp\;\mathit{anything}$$
$$\mid\;\; \mathsf{empty}$$

Not OK:

$$\mathit{void} \;=\; \mathsf{empty} \cdot \mathit{void}$$

Example

Kleene star:

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

Example

Left recursive Kleene star:

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

Example

Left recursive Kleene star:

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

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

Infinite ambiguity

Parser combinators:

return x The empty string. Result: x.

$p \gg\!= f$ First p, then f applied to result of p.

- $(\mathit{return}\ \mathsf{unit})\star \mapsto [], [\mathsf{unit}], [\mathsf{unit}, \mathsf{unit}], \ldots$
- $(\mathit{return}\ \mathsf{unit})\star \gg\!= \lambda\, xs \to$
  if $f\ xs$ **then** $\mathit{return}\ \mathsf{unit}$ **else** $\mathit{fail} \mapsto ?$

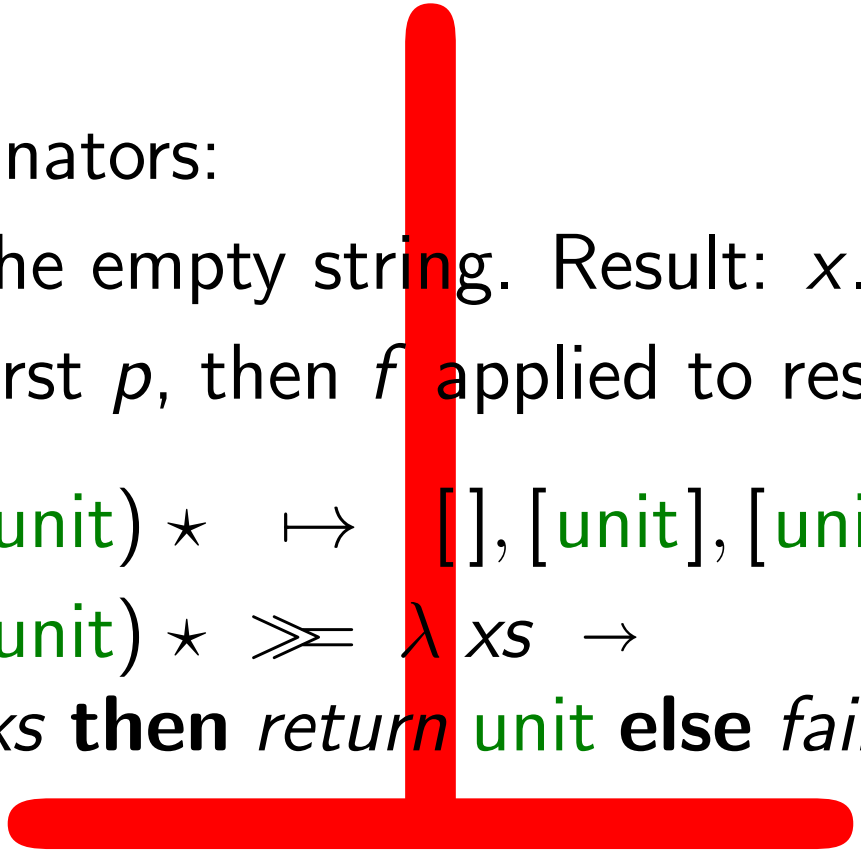
Infinite ambiguity

Parser combinators:

return x The empty string. Result: x.

$p \gg\!= f$ First p, then f applied to result of p.

- $(\mathit{return}\ \mathsf{unit})\ \star \quad \mapsto \quad [], [\mathsf{unit}], [\mathsf{unit}, \mathsf{unit}], \ldots$
- $(\mathit{return}\ \mathsf{unit})\ \star \gg\!= \lambda\, \mathit{xs} \rightarrow$
  if $f\ \mathit{xs}$ **then** *return* unit **else** *fail* $\quad \mapsto \quad ?$



Semantics

Semantics

$$\infty\langle\_\rangle P \;:\; \mathit{Bool} \;\to\; \mathit{Bool} \;\to\; \mathit{Set}$$
$$\infty\langle\ \mathsf{false}\ \rangle P\ n \;=\; \infty\,(P\ n)$$
$$\infty\langle\ \mathsf{true}\ \rangle P\ n \;=\; P\ n$$

$$\flat? \;:\; \infty\langle\ b\ \rangle P\ n \;\to\; P\ n$$
$$\flat?\ \{b \;=\; \mathsf{false}\}\ x \;=\; \flat\ x$$
$$\flat?\ \{b \;=\; \mathsf{true}\}\ x \;=\; x$$

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

Semantics

Inductive:

$$
\begin{aligned}
&\textbf{data}\ \_\in\_ : \mathit{List}\ \mathit{Token} \to P\ n \to \mathit{Set}\ \textbf{where}\\
&\quad \mathsf{empty} : [] \in \mathsf{empty}\\
&\quad \mathsf{sat} : f\ t \equiv \mathsf{true} \to [t] \in \mathsf{sat}\ f\\
&\quad \mid\text{-}\mathsf{left} : s \in p_1 \to s \in p_1 \mid p_2\\
&\quad \mid\text{-}\mathsf{right} : s \in p_2 \to s \in p_1 \mid p_2\\
&\quad \_\cdot\_ : s_1 \in \flat?\ p_1 \to s_2 \in \flat?\ p_2 \to\\
&\qquad\quad s_1 +\!\!+ s_2 \in p_1 \cdot p_2
\end{aligned}
$$

For $p : P\ n$:

$[] \in p$ iff $n \equiv \mathsf{true}$.

Backend

Backend

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

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

Backend

$$t :: s \in p \Leftrightarrow s \in D\ t\ p\text{:}$$

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

Top-level

$$
\begin{aligned}
&\mathit{nullable?} &&: (p : P\ n) \to \mathit{Dec}\ ([] \in p)\\
&\mathit{D\text{-}sound} &&: s \in D\ t\ p \to t :: s \in p\\
&\mathit{D\text{-}complete} &&: t :: s \in p \to s \in D\ t\ p
\end{aligned}
$$

$$
\begin{aligned}
&\_\in?\_ &&: (s : \mathit{List}\ \mathit{Token})\ (p : P\ n) \to \mathit{Dec}\ (s \in p)\\
&[] &&\in?\ p = ?\\
&t :: s &&\in?\ p = ?
\end{aligned}
$$

Top-level

$$
\begin{aligned}
&\mathit{nullable?} &&: (p : P\ n) \to \mathit{Dec}\ ([] \in p)\\
&\mathit{D\text{-}sound} &&: s \in D\ t\ p \to t :: s \in p\\
&\mathit{D\text{-}complete} &&: t :: s \in p \to s \in D\ t\ p
\end{aligned}
$$

$$
\begin{aligned}
&\_\in?\_ : (s : \mathit{List}\ \mathit{Token})\ (p : P\ n) \to \mathit{Dec}\ (s \in p)\\
&[] \in?\ p = \mathit{nullable?}\ p\\
&t :: s \in?\ p = ?
\end{aligned}
$$

Top-level

$$
\begin{aligned}
&\mathit{nullable?} &&: (p : P\ n) \to \mathit{Dec}\ ([] \in p)\\
&\mathit{D\text{-}sound} &&: s \in D\ t\ p \to t :: s \in p\\
&\mathit{D\text{-}complete} &&: t :: s \in p \to s \in D\ t\ p
\end{aligned}
$$

$$
\begin{aligned}
&\_\in?\_ : (s : \mathit{List}\ \mathit{Token})\ (p : P\ n) \to \mathit{Dec}\ (s \in p)\\
&[] \in?\ p = \mathit{nullable?}\ p\\
&t :: s \in?\ p\ \textbf{with}\ s \in?\ D\ t\ p\\
&\ldots \mid \mathrm{yes}\ s{\in}Dtp = ?\\
&\ldots \mid \mathrm{no}\ s{\notin}Dtp = ?
\end{aligned}
$$

Top-level

$$
\begin{aligned}
&\mathit{nullable?} &&: (p : P\ n) \to \mathit{Dec}\ ([] \in p)\\
&\mathit{D\text{-}sound} &&: s \in D\ t\ p \to t :: s \in p\\
&\mathit{D\text{-}complete} &&: t :: s \in p \to s \in D\ t\ p
\end{aligned}
$$

$$
\begin{aligned}
&\_\in?\_ : (s : \mathit{List}\ \mathit{Token})\ (p : P\ n) \to \mathit{Dec}\ (s \in p)\\
&[] \in?\ p = \mathit{nullable?}\ p\\
&t :: s \in?\ p\ \textbf{with}\ s \in?\ D\ t\ p\\
&\ldots \mid \mathsf{yes}\ s{\in}Dtp = \mathsf{yes}\ (\mathit{D\text{-}sound}\ s{\in}Dtp)\\
&\ldots \mid \mathsf{no}\ s{\notin}Dtp = ?
\end{aligned}
$$

Top-level

$$
\begin{aligned}
&\mathit{nullable?} &&: (p : P\ n) \to \mathit{Dec}\ ([] \in p)\\
&\mathit{D\text{-}sound} &&: s \in D\ t\ p \to t :: s \in p\\
&\mathit{D\text{-}complete} &&: t :: s \in p \to s \in D\ t\ p
\end{aligned}
$$

$$
\begin{aligned}
&\_\in?\_ : (s : \mathit{List}\ \mathit{Token})\ (p : P\ n) \to \mathit{Dec}\ (s \in p)\\
&[] \in?\ p = \mathit{nullable?}\ p\\
&t :: s \in?\ p\ \textbf{with}\ s \in?\ D\ t\ p\\
&\ldots \mid \mathsf{yes}\ s{\in}Dtp = \mathsf{yes}\ (\mathit{D\text{-}sound}\ s{\in}Dtp)\\
&\ldots \mid \mathsf{no}\ s{\notin}Dtp = \mathsf{no}\ (s{\notin}Dtp \circ \mathit{D\text{-}complete})
\end{aligned}
$$

Derivative

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

$$\begin{aligned}
&D : (t : \mathit{Token})\ (p : P\ n) \to P\ (\mathit{D\text{-}null?}\ t\ p) \\
&D\ t\ \mathsf{fail} = ? \\
&D\ t\ \mathsf{empty} = ? \\
&D\ t\ (\mathsf{sat}\ f) = ?
\end{aligned}$$

Derivative

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

$$\begin{aligned}
&D : (t : \mathit{Token})\ (p : P\ n) \to P\ (\mathit{D\text{-}null?}\ t\ p) \\
&D\ t\ \mathsf{fail} = \mathsf{fail} \\
&D\ t\ \mathsf{empty} = ? \\
&D\ t\ (\mathsf{sat}\ f) = ?
\end{aligned}$$

Derivative

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

$$\begin{aligned}
&D : (t : \mathit{Token})\ (p : P\ n) \to P\ (\mathit{D\text{-}null?}\ t\ p) \\
&D\ t\ \mathsf{fail} = \mathsf{fail} \\
&D\ t\ \mathsf{empty} = \mathsf{fail} \\
&D\ t\ (\mathsf{sat}\ f) = ?
\end{aligned}$$

Derivative

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

$$
\begin{aligned}
&D : (t : \mathit{Token})\ (p : P\ n) \to P\ (\mathit{D\text{-}null?}\ t\ p) \\
&D\ t\ \mathsf{fail} = \mathsf{fail} \\
&D\ t\ \mathsf{empty} = \mathsf{fail} \\
&D\ t\ (\mathsf{sat}\ f)\ \textbf{with}\ f\ t \\
&\ldots \mid \mathsf{true} = ? \\
&\ldots \mid \mathsf{false} = ?
\end{aligned}
$$

Derivative

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

$$
\begin{aligned}
&D : (t : \mathit{Token})\ (p : P\ n) \to P\ (\mathit{D\text{-}null?}\ t\ p) \\
&D\ t\ \mathsf{fail} \quad = \mathsf{fail} \\
&D\ t\ \mathsf{empty} = \mathsf{fail} \\
&D\ t\ (\mathsf{sat}\ f)\ \textbf{with}\ f\ t \\
&\ldots \mid \mathsf{true} = \mathsf{empty} \\
&\ldots \mid \mathsf{false} = ?
\end{aligned}
$$

Derivative

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

$$
\begin{aligned}
&D : (t : \mathit{Token})\ (p : P\ n) \to P\ (\mathit{D\text{-}null?}\ t\ p) \\
&D\ t\ \mathsf{fail} \quad = \mathsf{fail} \\
&D\ t\ \mathsf{empty} = \mathsf{fail} \\
&D\ t\ (\mathsf{sat}\ f)\ \mathbf{with}\ f\ t \\
&\ldots \mid \mathsf{true} = \mathsf{empty} \\
&\ldots \mid \mathsf{false} = \mathsf{fail}
\end{aligned}
$$

Choice

$$s \in D\ t\ (p_1 \mid p_2) \Leftrightarrow$$

$$t :: s \in p_1 \mid p_2$$

Choice

$$s \in D\ t\ (p_1 \mid p_2) \Leftrightarrow$$

$$\begin{array}{l} t :: s \in p_1 \\ \quad \vee \\ t :: s \in p_2 \end{array}$$

Choice

$$s \in D\ t\ (p_1 \mid p_2) \Leftrightarrow$$

$$s \in D\ t\ p_1$$
$$\vee$$
$$s \in D\ t\ p_2$$

Choice

$$s \in D\ t\ (p_1 \mid p_2) \Leftrightarrow$$

$$s \in D\ t\ p_1 \mid D\ t\ p_2$$

Sequencing

(Somewhat sloppy.)

$$s \in D\ t\ (p_1 \cdot p_2) \Leftrightarrow$$

$$t :: s \in p_1 \cdot p_2$$

Sequencing

(Somewhat sloppy.)

$$s \in D\ t\ (p_1 \cdot p_2) \Leftrightarrow$$

$$\exists\, s_1, s_2.\ s \equiv s_1 +\!\!+ s_2 \wedge t :: s_1 \in p_1 \wedge s_2 \in p_2$$
$$\vee$$
$$[\,] \in p_1 \wedge t :: s \in p_2$$

Sequencing

(Somewhat sloppy.)

$$s \in D\ t\ (p_1 \cdot p_2) \Leftrightarrow$$

$$\exists\, s_1, s_2.\ s \equiv s_1 +\!\!+ s_2 \ \wedge\ s_1 \in D\ t\ p_1 \ \wedge\ s_2 \in p_2$$
$$\vee$$
$$[\,] \in p_1 \ \wedge\ s \in D\ t\ p_2$$

Sequencing

(Somewhat sloppy.)

$$s \in D\ t\ (p_1 \cdot p_2) \Leftrightarrow$$

$$s \in D\ t\ p_1 \cdot p_2$$
$$\vee$$
$$[\,] \in p_1 \wedge s \in D\ t\ p_2$$

Sequencing

(Somewhat sloppy.)

$$s \in D\ t\ (p_1 \cdot p_2) \Leftrightarrow$$

$$s \in D\ t\ p_1 \cdot p_2$$
$$\vee$$
$$p_1\ \mathit{nullable} \wedge\ s \in D\ t\ p_2$$

Sequencing

(Somewhat sloppy.)

$$s \in D\ t\ (p_1 \cdot p_2) \Leftrightarrow$$

$$\begin{array}{l} p_1 \text{ nullable} \quad \wedge\ s \in D\ t\ p_1 \cdot p_2 \mid D\ t\ p_2 \\ \quad \vee \\ p_1 \text{ not nullable} \wedge\ s \in D\ t\ p_1 \cdot p_2 \end{array}$$

Derivative

$$
\begin{aligned}
&D\ t\ (p_1 \mid p_2) = D\ t\ p_1 \mid D\ t\ p_2 \\
&D\ t\ (p_1 \cdot p_2) \quad \textbf{with}\ \mathit{forced?}\ p_1 \mid \mathit{forced?}\ p_2 \\
&\ldots \mid \mathsf{true} \mid \mathsf{false} = D\ t\ p_1 \cdot \sharp?\ (\flat\ p_2) \\
&\ldots \mid \mathsf{false} \mid \mathsf{false} = \sharp\ D\ t\ (\flat\ p_1) \cdot \sharp?\ (\flat\ p_2) \\
&\ldots \mid \mathsf{true} \mid \mathsf{true} = D\ t\ p_1 \cdot \sharp?\ p_2 \mid D\ t\ p_2 \\
&\ldots \mid \mathsf{false} \mid \mathsf{true} = \sharp\ D\ t\ (\flat\ p_1) \cdot \sharp?\ p_2 \mid D\ t\ p_2
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{forced?} : \infty\langle\ b\ \rangle P\ n \to \mathit{Bool} \\
&\mathit{forced?}\ \{b = b\}\ \_ = b
\end{aligned}
$$

Future work

- The algorithm is inefficient.
- Optimise using algebraic identities?

Other uses of derivatives

Equality

Inductive definition:

$$
\begin{aligned}
&\_\approx\_ \;:\; P\ n_1 \;\to\; P\ n_2 \;\to\; \mathit{Set}\\
&p_1 \;\approx\; p_2 \;=\; (s \;:\; \mathit{List}\ \mathit{Tok}) \;\to\; s \;\in\; p_1 \;\Leftrightarrow\; s \;\in\; p_2
\end{aligned}
$$

Equivalent, coinductive definition:

$$
\begin{aligned}
&\textbf{data}\ \_\approx_{\mathrm{c}}\_\ (p_1 \;:\; P\ n_1)\ (p_2 \;:\; P\ n_2) \;:\; \mathit{Set}\ \textbf{where}\\
&\quad \_::\_ \;:\; n_1 \equiv n_2 \;\to\\
&\qquad\quad ((t \;:\; \mathit{Tok}) \;\to\; \infty\ (D\ t\ p_1 \;\approx_{\mathrm{c}}\; D\ t\ p_2)) \;\to\\
&\qquad\quad p_1 \;\approx_{\mathrm{c}}\; p_2
\end{aligned}
$$

Latter definition: corecursive proofs.

Derived combinators

$p \gg\!= f$ First p, then f applied to result of p.

token Accepts and returns a token.

return⋆ xs Returns an arbitrary element from *xs*.

initial-bag Results when applied to empty string.

Left-leaning asymmetric choice:

$$p_1 \lhd p_2 = (\mathit{token} \gg\!= \lambda\, t \to \sharp\, (D\ t\ p_1 \lhd D\ t\ p_2)) \mid \mathit{return}\star\, (\mathit{first\text{-}nonempty}\, (\mathit{initial\text{-}bag}\ p_1)\ (\mathit{initial\text{-}bag}\ p_2))$$

Derived combinators

$p \gg\!= f$ First p, then f applied to result of p.

token Accepts and returns a token.

return⋆ xs Returns an arbitrary element from *xs*.

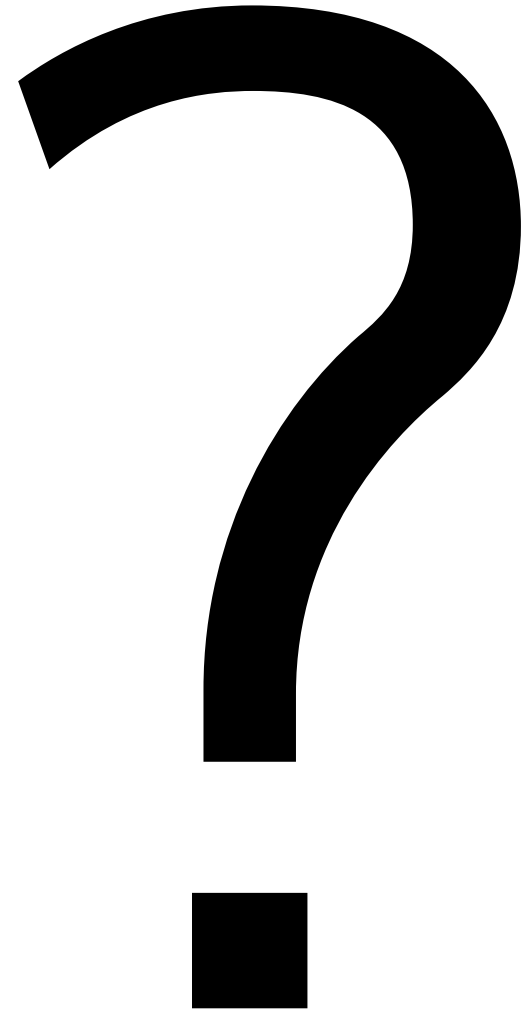
initial-bag Results when applied to empty string.

Conjunction:

$$p_1 \mathbin{\&} p_2 = (\mathit{token} \gg\!= \lambda\, t \to \sharp\, (D\ t\ p_1 \mathbin{\&} D\ t\ p_2)) \mid \mathit{return}\star\, (\mathit{initial\text{-}bag}\ p_1 \otimes \mathit{initial\text{-}bag}\ p_2)$$

Conclusions

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



Laws

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

$$\_\bigstar : P\ n \to P\ \mathsf{true}$$
$$p \bigstar = (\mathit{nonempty}\ p) \star$$

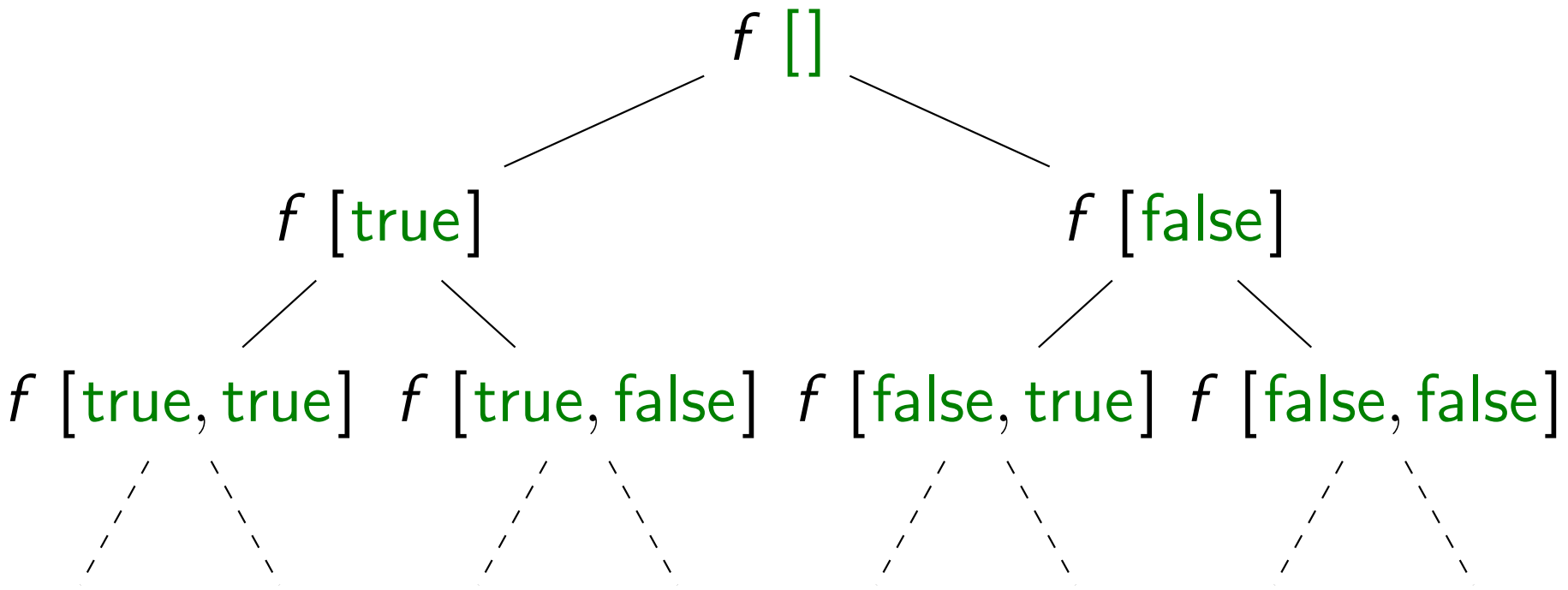
- The *nonempty* combinator can be defined by structural recursion:

$$\mathit{nonempty} : P\ n \to P\ \mathsf{false}$$

Expressive strength

For finite alphabets the combinators are as expressive as possible:

$f \ : \ \mathit{List}\ \mathit{Bool} \ \rightarrow \ \mathit{Bool}$



Expressive strength

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

$$
\begin{aligned}
&p : (f : \mathit{List}\ \mathit{Bool} \to \mathit{Bool}) \to P\ (f\ [\,]) \\
&p\ f = \\
&\quad \sharp\ (p\ (\lambda xs \to f\ (xs +\!\!+ [\mathsf{true}\,]))) \cdot \sharp?\ (\mathsf{sat}\ \mathit{id}) \\
&\quad \mid \sharp\ (p\ (\lambda xs \to f\ (xs +\!\!+ [\mathsf{false}]))) \cdot \sharp?\ (\mathsf{sat}\ \mathit{not}) \\
&\quad \mid \mathit{accept\text{-}if\text{-}true}\ (f\ [\,])
\end{aligned}
$$

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