Structurally Recursive Descent Parsing

Nils Anders Danielsson (Nottingham)
Joint work with Ulf Norell (Chalmers)

Cambridge, 2009-03-20

Parser combinator example

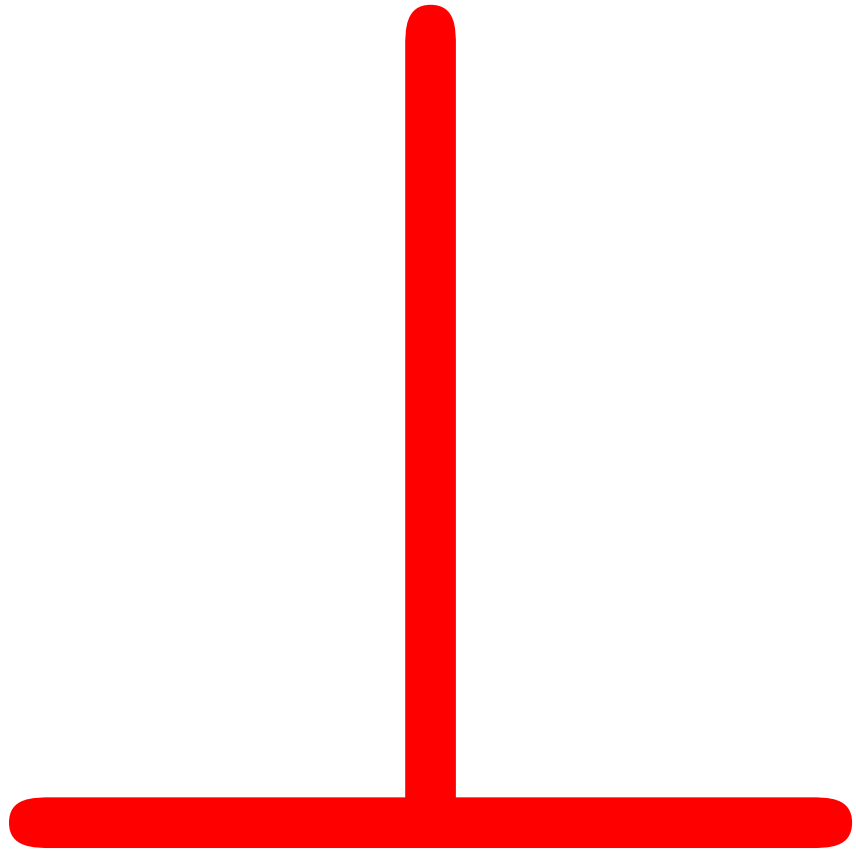
$$
\begin{aligned}
\mathit{expr} &= \mathit{term} \cdot \mathit{tok}\ \texttt{"+"} \cdot \mathit{expr} \\
&\mid \mathit{term} \\
\mathit{term} &= \ldots
\end{aligned}
$$

Parser combinator propaganda

- Parser combinator libraries are great!
- Elegant code.
- Executable grammars.
- Easy to abstract out recurring patterns.
- Light-weight.
- Nowadays often fast enough.

Parser combinator example

$$
\begin{aligned}
\mathit{expr} &= \mathit{expr} \cdot \mathit{tok}\ \texttt{"+"} \cdot \mathit{term} \\
&\mid \mathit{term} \\
\mathit{term} &= \ldots
\end{aligned}
$$



Risk of non-termination

- Combinator parsing is typically not guaranteed to terminate.
- Most combinator parsers fail for left-recursive grammars.
- *Executable* grammars?
- Some errors are not caught at compile-time.

Goal

Parser combinator library implemented in total language

- For simplicity: recursive descent.

Language: Agda

- Functional.
- Dependent types.
- Total.
- Support for mixed induction/coinduction and mutual structural recursion/guarded corecursion.

Recognisers

Termination

- Recursive descent $\Rightarrow$
  left recursion has to be avoided.
- Grammars often cyclic $\Rightarrow$ corecursion.
- Recognition returns inductive value ($Bool$) $\Rightarrow$
  structural recursion.

Interface

$$
\begin{array}{lcl}
\emptyset & : & G \\
\varepsilon & : & G \\
\mathsf{tok} & : & \mathit{Token} \to G \\
\_|\_ & : & G \to G \to G \\
\_\cdot\_ & : & G \to G \to G
\end{array}
$$

Corecursion will be used $\Rightarrow$
some arguments have to be coinductive.

Choice

Hard to decide infinite choice:

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

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

Sequencing

Also problematic (due to top-down, left-to-right implementation):

$$p = p \cdot p'$$

OK if p' does not accept the empty string:

$$p = p' \cdot p$$

Grammars

Index true iff empty string accepted:

$$
\begin{array}{l}
\textbf{data}\ G\ :\ \mathit{Bool} \to \mathit{Set}\ \textbf{where} \\
\quad \emptyset \quad\ :\ G\ \mathsf{false} \\
\quad \varepsilon \quad\ :\ G\ \mathsf{true} \\
\quad \mathsf{tok} \ :\ \mathit{Token} \to G\ \mathsf{false} \\
\quad \_|\_ \ :\ \forall\ \{e_1\ e_2\} \to G\ e_1 \to G\ e_2 \to G\ (e_1 \vee e_2) \\
\quad \_?\cdot\_ \ :\ \forall\ \{e_2\} \to G\ \mathsf{true} \to \quad G\ e_2 \ \to G\ e_2 \\
\quad \_!\cdot\_ \ :\ \forall\ \{e_2\} \to G\ \mathsf{false} \to \infty\ (G\ e_2) \to G\ \mathsf{false}
\end{array}
$$

Coinductive types

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

$$\sharp : \forall\, \{A\} \to A \to \infty\, A$$
$$\flat : \forall\, \{A\} \to \infty\, A \to A$$

- $T = F\ (\infty\ T)\ T \quad \approx \quad T = \nu C.\mu I.F\ C\ I.$

Coinductive types

$$
\begin{aligned}
&G \;:\; \mathit{Bool} \to \mathit{Set} \\
&G\ e \cong \nu C.\mu I. \\
&\qquad\qquad\quad e \equiv \mathsf{false} \\
&\qquad\quad + \; e \equiv \mathsf{true} \\
&\qquad\quad + \; \mathit{Token} \;\times\; e \equiv \mathsf{false} \\
&\qquad\quad + \; \Sigma\ e_1\ e_2.\ I\ e_1 \;\times\; I\ e_2 \;\times\; e \equiv e_1 \vee e_2 \\
&\qquad\quad + \; \Sigma\ e_2.\ \;\; I\ \mathsf{true} \;\times\; I\ e_2 \;\times\; e \equiv e_2 \\
&\qquad\quad + \; \Sigma\ e_2.\ \;\; I\ \mathsf{false} \;\times\; C\ e_2 \;\times\; e \equiv \mathsf{false}
\end{aligned}
$$

Semantics

Inductive:

$$
\begin{aligned}
&\textbf{data}\ \_\in\_ : \forall\ \{e\} \to \mathit{List}\ \mathit{Token} \to G\ e \to \mathit{Set}\ \textbf{where}\\
&\quad \varepsilon \quad\ : [] \in \varepsilon\\
&\quad \mathsf{tok} \ : [t] \in \mathsf{tok}\ t\\
&\quad \mid^{\ell} \quad : s \in p_1 \to s \in p_1 \mid p_2\\
&\quad \mid^{\mathsf{r}} \quad : s \in p_2 \to s \in p_1 \mid p_2\\
&\quad \_?\cdot\_ \ : s_1 \in p_1 \to s_2 \in\ \ p_2 \to s_1 +\!\!+ s_2 \in p_1\ ?\cdot\ p_2\\
&\quad \_!\cdot\_ \ : s_1 \in p_1 \to s_2 \in \flat\, p_2 \to s_1 +\!\!+ s_2 \in p_1\ !\cdot\ p_2
\end{aligned}
$$

For $p : G\ e$:

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

Example

Kleene star:

```
mutual
  _⋆ : G false → G true
  p ⋆ = ε | p +
  _+ : G false → G false
  p + = p !· ♯ (p ⋆)
```

The argument must not accept the empty string;
$\varepsilon \star$ is not very useful.

Implementation

Implementation

Bounded vectors:

$$
\begin{aligned}
&\textbf{data}\ \mathit{BVec}\ (A\ :\ \mathit{Set})\ :\ \mathbb{N} \to \mathit{Set}\ \textbf{where}\\
&\quad [] \quad\ : \forall\ \{n\} \to \mathit{BVec}\ A\ n\\
&\quad \_::\_\ : \forall\ \{n\} \to A \to \mathit{BVec}\ A\ n \to \mathit{BVec}\ A\ (\mathit{suc}\ n)
\end{aligned}
$$

$$
\begin{aligned}
&\uparrow\ :\ \forall\ \{A\ n\} \to \mathit{BVec}\ A\ n \to \mathit{BVec}\ A\ (\mathit{suc}\ n)\\
&\uparrow [] \qquad\ = [] \\
&\uparrow (x :: xs)\ =\ x :: \uparrow xs
\end{aligned}
$$

Implementation

$$
\begin{aligned}
&\mathit{parse} \;:\; \forall\, \{e\ n\} \to G\ e \to \mathit{BVec}\ \mathit{Token}\ n \to \\
&\qquad \mathit{List}\ (\mathit{BVec}\ \mathit{Token}\ (\mathit{if}\ e\ \mathit{then}\ n\ \mathit{else}\ \mathit{pred}\ n)) \\
&\mathit{parse}\ \emptyset \qquad s \qquad = \mathit{fail} \\
&\mathit{parse}\ \varepsilon \qquad s \qquad = \mathit{return}\ s \\
&\mathit{parse}\ (\mathsf{tok}\ t)\ [] \qquad = \mathit{fail} \\
&\mathit{parse}\ (\mathsf{tok}\ t)\ (t' :: s) \;= \\
&\quad \mathit{if}\ t == t'\ \mathit{then}\ \mathit{return}\ s\ \mathit{else}\ \mathit{fail}
\end{aligned}
$$

Implementation

$$
\begin{aligned}
&\mathit{parse} \;:\; \forall\, \{e\ n\} \to G\ e \to \mathit{BVec}\ \mathit{Token}\ n \to \\
&\qquad \mathit{List}\ (\mathit{BVec}\ \mathit{Token}\ (\mathit{if}\ e\ \mathit{then}\ n\ \mathit{else}\ \mathit{pred}\ n)) \\
&\mathit{parse}\ (p_1 \mid_{e_1} p_2)\ s \;=\; (\uparrow_1 e_1\ \mathbin{<\$>}\ \mathit{parse}\ p_1\ s) +\!\!+ \\
&\qquad\qquad\qquad\qquad (\uparrow_2 e_1\ \mathbin{<\$>}\ \mathit{parse}\ p_2\ s) \\
&\mathit{parse}\ (p_1\ ?\cdot\ p_2)\ s \;=\; \mathit{parse}\ p_2 \mathbin{=\!\!\ll} \mathit{parse}\ p_1\ s \\
&\mathit{parse}\ (p_1\ !\cdot\ \ p_2)\ [] \qquad\quad = \; \mathit{fail} \\
&\mathit{parse}\ (p_1\ !\cdot_{e_2}\ p_2)\ (t :: s) \;= \\
&\quad \uparrow_3 e_2\ \mathbin{<\$>}\ (\mathit{parse}\ (\flat\ p_2) \mathbin{=\!\!\ll} \mathit{parse}\ p_1\ (t :: s))
\end{aligned}
$$

Implementation

$$
\begin{aligned}
&\mathit{parse} \;:\; \forall\, \{e\ n\} \to G\ e \to \mathit{BVec}\ \mathit{Token}\ n \to \\
&\qquad \mathit{List}\ (\mathit{BVec}\ \mathit{Token}\ (\mathit{if}\ e\ \mathit{then}\ n\ \mathit{else}\ \mathit{pred}\ n)) \\
&\mathit{parse}\ (p_1 \mid_{e_1} p_2)\ s \;=\; (\uparrow_1 e_1\ \mathrel{<\$>}\ \mathit{parse}\ p_1\ s) +\!\!+ \\
&\qquad\qquad\qquad\qquad (\uparrow_2 e_1\ \mathrel{<\$>}\ \mathit{parse}\ p_2\ s) \\
&\mathit{parse}\ (p_1\ ?\cdot\ p_2)\ s \;=\; \mathit{parse}\ p_2 \mathrel{=\!\!\!\ll} \mathit{parse}\ p_1\ s \\
&\mathit{parse}\ (p_1\ !\cdot\ \ p_2)\ [] \qquad\quad = \mathit{fail} \\
&\mathit{parse}\ (p_1\ !\cdot_{e_2} p_2)\ (t :: s) \;= \\
&\quad \uparrow_3 e_2\ \mathrel{<\$>}\ (\mathit{parse}\ (\flat\ p_2) \mathrel{=\!\!\!\ll} \mathit{parse}\ p_1\ (t :: s))
\end{aligned}
$$

Implementation

$$\_\in?\_ : \forall \{e\} \to \mathit{List}\ \mathit{Token} \to G\ e \to \mathit{Bool}$$
$$s \in?\ p = \mathit{any}\ \mathit{null}\ (\mathit{parse}\ p\ (\mathit{fromList}\ s))$$

Parsers

Actual parsers

$$
\begin{aligned}
&\textbf{data}\ P\ :\ \mathit{Bool} \to \mathit{Set} \to \mathit{Set}\ \textbf{where}\\
&\quad \mathsf{return}\ :\ \forall\ \{R\} \to R \to P\ \mathsf{true}\ R\\
&\quad \mathsf{fail}\ \ :\ \forall\ \{R\} \to P\ \mathsf{false}\ R\\
&\quad \mathsf{token}\ :\ P\ \mathsf{false}\ \mathit{Token}\\
&\quad \_|\_\ \ :\ \forall\ \{e_1\ e_2\ R\} \to\\
&\qquad\qquad P\ e_1\ R \to P\ e_2\ R \to P\ (e_1 \vee e_2)\ R
\end{aligned}
$$

Actual parsers

$$
\begin{aligned}
&\textbf{data}\ P\ :\ \mathit{Bool} \to \mathit{Set} \to \mathit{Set}\ \textbf{where}\\
&\quad \_?\gg\!=\_\ :\ \forall\ \{e_2\ R_1\ R_2\} \to\\
&\qquad\qquad P\ \mathsf{true}\ R_1 \to (R_1 \to P\ e_2\ R_2) \to P\ e_2\ R_2\\
&\quad \_!\gg\!=\_\ :\ \forall\ \{R_1\ R_2\}\ \{e_2\ :\ R_1 \to \mathit{Bool}\} \to\\
&\qquad\qquad\qquad\qquad P\ \mathsf{false}\quad R_1\quad \to\\
&\qquad\qquad ((x\ :\ R_1) \to \infty\ (P\ (e_2\ x)\ R_2)) \to\\
&\qquad\qquad\qquad\qquad P\ \mathsf{false}\quad R_2
\end{aligned}
$$

Tokens satisfying predicate

```
sat : ∀ {R} → (Token → Maybe R) → P false R
sat {R} p = token !>>= λ t → ♯ (ok (p t))
  where
  ok-index : Maybe R → Bool
  ok-index nothing = false
  ok-index (just _) = true

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

Derived combinators

$$
\begin{aligned}
&\_\gg\!=\_ \;:\; \forall\, \{e_1\ e_2\ R_1\ R_2\} \to \\
&\qquad P\ e_1\ R_1 \to (R_1 \to P\ e_2\ R_2) \to \\
&\qquad P\ (e_1 \wedge e_2)\ R_2 \\
&\_\gg\!=\_\ \{\mathsf{false}\}\ p_1\ p_2 \;=\; p_1\ !\gg\!= \lambda\ x \to \sharp\ (p_2\ x) \\
&\_\gg\!=\_\ \{\mathsf{true}\}\ \ p_1\ p_2 \;=\; p_1\ ?\gg\!= p_2
\end{aligned}
$$

$$
\begin{aligned}
&\_\gg\_ \;:\; \forall\, \{e_1\ e_2\ R_1\ R_2\} \to \\
&\qquad P\ e_1\ R_1 \to P\ e_2\ R_2 \to P\ \_\ R_2 \\
&p_1 \gg p_2 \;=\; p_1 \gg\!= \lambda\ \_ \to p_2
\end{aligned}
$$

Exactly n occurrences

No corecursion:

$$
\begin{aligned}
&\mathit{exactly} \;:\; \forall\, \{e\ R\}\ (n \;:\; \mathbb{N}) \to P\ e\ R \to P\ ?\ (\mathit{Vec}\ R\ n) \\
&\mathit{exactly}\ \mathsf{zero}\quad\ \ p \;=\; \mathsf{return}\ [] \\
&\mathit{exactly}\ (\mathsf{suc}\ n)\ p \;=\; p \gg\!= \lambda\, x \to \\
&\qquad\qquad\qquad\qquad\quad \mathit{exactly}\ n\ p \gg\!= \lambda\, \mathit{xs} \to \\
&\qquad\qquad\qquad\qquad\quad \mathsf{return}\ (x :: \mathit{xs})
\end{aligned}
$$

Exactly n occurrences

$$
\begin{aligned}
&\mathit{exactly\text{-}index} \;:\; \mathit{Bool} \to \mathbb{N} \to \mathit{Bool} \\
&\mathit{exactly\text{-}index}\ e\ \mathsf{zero} \;=\; \_ \\
&\mathit{exactly\text{-}index}\ e\ (\mathsf{suc}\ n) \;=\; \_ \\
&\mathit{exactly} \;:\; \forall\ \{e\ R\}\ (n \;:\; \mathbb{N}) \to P\ e\ R \to \\
&\qquad\qquad P\ (\mathit{exactly\text{-}index}\ e\ n)\ (\mathit{Vec}\ R\ n) \\
&\mathit{exactly}\ \mathsf{zero}\ \ p \;=\; \mathsf{return}\ [] \\
&\mathit{exactly}\ (\mathsf{suc}\ n)\ p \;=\; p \gg\!= \lambda\ x \to \\
&\qquad\qquad\qquad\qquad \mathit{exactly}\ n\ p \gg\!= \lambda\ xs \to \\
&\qquad\qquad\qquad\qquad \mathsf{return}\ (x :: xs)
\end{aligned}
$$

Problems

Monadic interface

Index of second parser depends on result of first:

$$
\begin{aligned}
a^n b^n c^n \;=\; & \mathsf{tok}\ \texttt{"a"} \star \;\gg\!=\; \lambda\, as \rightarrow \\
& \textbf{let}\ n \;=\; \mathit{length}\ as\ \textbf{in} \\
& \mathit{exactly}\ n\ (\mathsf{tok}\ \texttt{"b"}) \gg \\
& \mathit{exactly}\ n\ (\mathsf{tok}\ \texttt{"c"}) \gg \\
& \mathsf{return}\ n
\end{aligned}
$$

Monadic interface

Workaround:

$$
\begin{aligned}
a^n b^n c^n \;=\;& \mathsf{return}\ \mathsf{zero} \\
\mid\;& \mathsf{tok}\ \texttt{"a"} + \qquad\qquad\qquad\qquad !\gg\!= \lambda\, as \rightarrow \sharp\ (\\
& \mathbf{let}\ n \;=\; \mathit{length}\ as\ \mathbf{in} \\
& \mathit{exactly}\ n\ (\mathsf{tok}\ \texttt{"b"}) \gg \\
& \mathit{exactly}\ n\ (\mathsf{tok}\ \texttt{"c"}) \gg \\
& \mathsf{return}\ n)
\end{aligned}
$$

Limitations of productivity checker

- Agda requires corecursion to be guarded by *constructors*.
- Not OK:

```
mutual
  expr   = term   sepBy addOp
  term   = factor sepBy mulOp
  factor = number
         | tok "(" ≫ expr ≫ tok ")"
```

Limitations of productivity checker

Workaround:

```
data NT : Index → Set → Set where
  expr   : NT _ _
  term   : NT _ _
  factor : NT _ _

grammar : ∀ {i R} → NT i R → P NT i R
grammar expr   = ! term   sepBy addOp
grammar term   = ! factor sepBy mulOp
grammar factor = number
               | tok "(" ≫ ! expr ≫ tok ")"
```

Limitations of productivity checker

Workaround:

$$
\begin{aligned}
&\mathit{grammar} \;:\; \forall\, \{i\ R\} \to \mathit{NT}\ i\ R \to P\ \mathit{NT}\ i\ R \\
&\mathit{grammar}\ \mathsf{expr} \;=\; !\ \mathsf{term}\ \ \mathit{sepBy}\ \mathit{addOp} \\
&\mathit{grammar}\ \mathsf{term} \;=\; !\ \mathsf{factor}\ \mathit{sepBy}\ \mathit{mulOp} \\
&\mathit{grammar}\ \mathsf{factor} \;=\; \mathit{number} \\
&\qquad\qquad\qquad \mid\ \mathsf{tok}\ \text{"("} \gg\ !\ \mathsf{expr} \gg\ \mathsf{tok}\ \text{")"}
\end{aligned}
$$

$$
! \;:\; \forall\, \{e\ c\ R\} \to \mathit{NT}\ (e, c)\ R \to P\ \mathit{NT}\ (e, \mathsf{step}\ c)\ R
$$

Limitations of productivity checker

Workaround:

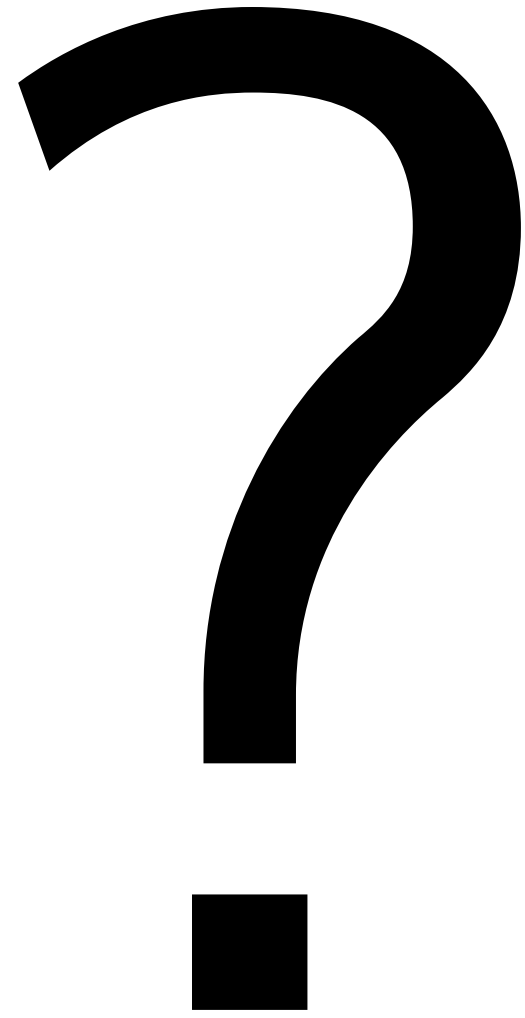
$$! \;:\; \forall\, \{e\ c\ R\} \to NT\ (e, c)\ R \to P\ NT\ (e, \mathsf{step}\ c)\ R$$

- Index includes *inductive* approximation of left corners (c).
- Left recursive definitions are disallowed:
  $$\mathit{grammar}\ \mathsf{nt} = ! \ \mathsf{nt} \quad \Rightarrow \quad c = \mathsf{step}\ c$$
  $$\mathit{grammar}\ \mathsf{nt} = ! \ \mathsf{nt} \gg\!= p$$
  $$\Rightarrow \quad c = \mathsf{step}\ c \cup c'$$
- Can be implemented without coinduction.
- Not quite parser *combinators*.

Conclusions

Conclusions

- Simple definition of grammars without left recursion.
- Combinator parsing with termination guarantees.
- Productivity checker needs to be improved.
- Larger examples possible:
  - Grammar scheme for mixfix operators.
- Mixed induction/coinduction is fun.



Bonus

“Declarative” coinductive inference systems

$$
\begin{aligned}
&\textbf{data}\ \_\leqslant\_ : \mathit{Ty} \to \mathit{Ty} \to \mathit{Set}\ \textbf{where}\\
&\quad \mathsf{arrow} : \infty\,(\tau_1 \leqslant \sigma_1) \to \infty\,(\sigma_2 \leqslant \tau_2) \to \sigma_1 \to \sigma_2 \leqslant \tau_1 \to \tau_2\\
&\quad \mathsf{refl} : \tau \leqslant \tau\\
&\quad \mathsf{trans} : \tau_1 \leqslant \tau_2 \to \tau_2 \leqslant \tau_3 \to \tau_1 \leqslant \tau_3\\
&\quad \ldots
\end{aligned}
$$