Parallel Parsing Processes Revisited

Michael Spivey
*University of Oxford*

Department of COMPUTER SCIENCE
Thompson [1968]

Compiles regexps into NFAs represented as machine code (for the IBM 7094).

Matching machine reads the input one character at a time, and dynamically maintains two lists of subroutine calls:

- **CLIST** – alternatives for the current character
- **NLIST** – alternatives for the next character.

[Makes a great student project!]

Translating regexps

- For $c$: if $\text{char} = c$ then add next to $\text{NLIST}$; goto $\text{FAIL}$
- For $\epsilon$: if $\text{char} = \Lambda$ then goto $\text{SUCCESS}$ else $\text{FAIL}$
- For $E_1 \ E_2$: code for $E_1$; code for $E_2$
- For $E_1 \mid E_2$: add $E_2$ to $\text{CLIST}$; code for $E_1$
- For $E_1^*$: add $\{ \ E_1; \text{goto } E_1^* \ \}$ to $\text{CLIST}$; goto next

$\text{FAIL}$:
   if $\text{CLIST} \neq [\ ]$ then pop and goto first element
   else $\{ \text{advance char}; \text{CLIST} = \text{NLIST}; \text{NLIST} = [\ ] \ \}$
Thompson lite

\[ \text{match} :: \]
\[ \text{Regexp} \rightarrow \{\text{Regexp}\} \rightarrow \{\text{Regexp}\} \rightarrow \text{String} \rightarrow \text{Bool} \]

\[ \text{match} \ (\text{Seq} \ (\text{Lit} \ c) \ e_k) \ \text{clist} \ \text{nlist} \ s \ | \ (\text{head} \ s == c) = \]
\[ \text{resume} \ \text{clist} \ (e_k : \text{nlist}) \ s \]

\[ \text{match} \ (\text{Seq} \ (\text{Alt} \ e_1 \ e_2) \ e_k) \ \text{clist} \ \text{nlist} \ s = \]
\[ \text{match} \ (\text{Seq} \ e_1 \ e_k) \ (\text{Seq} \ e_2 \ e_k : \text{clist}) \ \text{nlist} \ s \]

\[ \ldots \]

\[ \text{resume} \ (c : \text{clist}) \ \text{nlist} \ s = \text{match} \ c \ \text{clist} \ \text{nlist} \ s \]

\[ \text{resume} \ [\ ] \ (n : \text{nlist}) \ s = \text{match} \ n \ \text{nlist} \ [\ ] \ (\text{tail} \ s) \]
Parser combinators

\[ expr = \]
\[ \text{factor} \oplus \]
\[ (\text{do } a \leftarrow \text{factor}; \text{eat } '+'; b \leftarrow \text{expr}; \text{return} (\text{Plus } a \ b)) \]

\[ factor = \]
\[ (\text{do } x \leftarrow \text{ident}; \text{return} (\text{Var } x)) \oplus \]
\[ (\text{do } \text{eat } '('; a \leftarrow \text{expr}; \text{eat } ')'; \text{return} \ a) \]

\[ \text{eat } x = (\text{do } y \leftarrow \text{scan}; \text{if } x == y \text{ then return } () \text{ else fail}) \]

- Can be implemented with state and backtracking
- Or …
Claessen [2004]

'Parallel' parser combinators

```haskell
data Parser α =
    Scan (Token → Parser α)
  | Result α (Parser α)
  | Fail
```

- A parser can: say it wants to know the next token
- or produce a result (and provide alternatives)
- or just fail.

Alternation – the vital idea

\[ \text{Fail} \oplus q = q \]

\[ (\text{Result } x p') \oplus q = \text{Result } x (p' \oplus q) \]

\[ (\text{Scan } g) \oplus \text{Fail} = \text{Scan } g \]

\[ (\text{Scan } g) \oplus (\text{Result } x q') = \text{Result } x ((\text{Scan } g) \oplus q') \]

\[ (\text{Scan } g) \oplus (\text{Scan } h) = \text{Scan } (\lambda x \rightarrow g x \oplus h x) \]

- we delay \( p \oplus q \) from looking at the next token until both \( p \) and \( q \) are ready for it.
It's a monad and more

\[
\text{return } x = \text{Result } x \text{ fail}
\]

\[
(\text{Result } x \ p) \gg= f = \text{Result } x (\ p \gg= f)
\]
\[
(\text{Scan } g) \gg= f = \text{Scan } (\lambda \ x \to g \ x \gg= f)
\]
\[
\text{Fail} \gg= f = \text{Fail}
\]

\[
\text{scan} = \text{Scan return}
\]

\[
\text{fail} = \text{Fail}
\]

- These are the operations \((\text{MonadPlus plus scan})\) needed to write parsers.
Driving a parser

The main program marries the parser state with the stream of input tokens, looking for a result that consumes the whole input.

\[
\text{parse} :: \text{Parser } \alpha \rightarrow [\text{Token}] \rightarrow \alpha
\]

\[
\text{parse} (\text{Scan } g) \quad [\ ] \quad = \quad \text{error } "\text{unexpected EOF}"
\]

\[
\text{parse} (\text{Scan } g) \quad (t : ts) \quad = \quad \text{parse } (g \ t) \ ts
\]

\[
\text{parse} (\text{Result } x \ p) \quad [\ ] \quad = \quad x
\]

\[
\text{parse} (\text{Result } x \ p) \quad ts \quad = \quad \text{parse } p \ ts
\]

\[
\text{parse } \text{Fail} \quad _ \quad = \quad \text{error } "\text{syntax error}"
\]

- easy to track the latest token for error messages.
Benefits of PPP

- *No backtracking*, so cleans up non-viable alternatives early – simple grammars are usable without transformation or annotation.

- Reads the input *token by token*, so can be made interactive without relying on lazy streams. Example: prompting for each line of input.

- Will *report first token* that is not part of any legal sentence: one error message for free.

- *Fast enough* to use in practice.
Using continuations

An alternative implementation: each parser take one, two, three continuations.

\[
\text{type } KParser \alpha = VCont \alpha \rightarrow CCont \rightarrow NCont \rightarrow \text{Answer}
\]

\[
\text{type } VCont \alpha = \alpha \rightarrow CCont \rightarrow NCont \rightarrow \text{Answer}
\]

\[
\text{type } CCont = NCont \rightarrow \text{Answer}
\]

\[
\text{type } NCont = Token \rightarrow CCont \rightarrow \text{Answer}
\]

\[
\text{type } Answer = [Token] \rightarrow \text{Value}
\]

• \textbf{newtype} is needed all over the place.
A slew of one-liners

The same five operations now have direct definitions.

\( \text{return } x \cdot k = k \cdot x \)

\((p \gg f) \cdot k = p \cdot (\lambda x \rightarrow f \cdot x \cdot k)\)

\(\text{fail } k \cdot ck = ck\)

\((p \oplus q) \cdot k \cdot ck = (p \cdot k \cdot q \cdot k) \cdot ck = p \cdot k \cdot (\lambda nk \rightarrow q \cdot k \cdot ck \cdot nk)\)

\(\text{scan } k \cdot ck \cdot nk = ck \cdot (\lambda t \rightarrow nk \cdot t \cdot k \cdot t)\)
Where did that come from?

Define $rep :: Parser \alpha \rightarrow KParser \alpha$ by

\[
rep (Scan g) k ck nk = ck \ (\lambda t \rightarrow nk t \cdot k t)
\]

\[
rep (Result x p) k ck nk = k x (rep p k ck) nk
\]

\[
rep \ Fail k ck nk = ck nk
\]

Then all else follows!
Deriving bind and plus

In particular, we can prove inductively that

\[ \text{rep} (p ≫= f) k = \text{rep} p (\lambda x \to \text{rep} (f x) k) \]

and

\[ \text{rep} (p ⊕ q) k ck = \text{rep} p k (\text{rep} q k ck) \]

These justify the new definitions of ≫= and ⊕.
Driving the new parser

$kparse :: KParser Value \rightarrow [Token] \rightarrow Value$

$kparse p = p \ k_0 \ ck_0 \ nk_0$

where

$k_0 \ x \ ck \ nk \ ts =$

\textbf{if } ts == [ ] \ \textbf{then } x \ \textbf{else } ck \ nk \ ts$

$ck_0 \ nk \ [ ] = \text{error } "\text{unexpected EOF}"

$ck_0 \ nk \ (t:ts) = nk \ t \ ck_0 \ ts$

$nk_0 \ t \ ck = ck \ nk_0$
Defunctionalising

"Looking for the lambdas", we find that $N\text{Conts}$ are created only by the expression

$$(\lambda t \rightarrow nk t \cdot k t)$$

(with $k$ and $nk$ as free variables) and $C\text{Conts}$ only by the expression

$$(\lambda nk \rightarrow q k ck nk)$$

and by promoting $N\text{Conts}$ to $C\text{Conts}$ when scanning.

We can represent both by lists of (ordinary) continuations, with a suitable resume function.
Concrete continuations

\[
\text{scan } k \text{ clist nlist ts } = \\
\begin{align*}
\text{resume clist } (k \ (\text{head ts}) : \text{nlist})
\end{align*}
\]

\[
\text{fail } k \text{ clist nlist } = \text{resume clist nlist}
\]

\[
(p \oplus q) \ k \text{ clist nlist } = \ p \ k \ (q \ k \ : \text{clist}) \text{nlist}
\]

\[
\text{resume } (k : \text{clist}) \text{nlist ts } = \ k \text{ clist nlist ts}
\]

\[
\text{resume } [ ] \text{nlist } [ ] = \text{error } "\text{unexpected EOF}" \\
\text{resume } [ ] \text{nlist ts } = \text{resume } (\text{reverse nlist}) [ ] \ (\text{tail ts})
\]

- The \textit{reverse} is needed because sometimes we care about the order of results.
Focussing ...

\textbf{type} \textit{KParser} \alpha =
\quad \textit{VCont} \alpha \rightarrow [\textit{Cont}] \rightarrow [\textit{Cont}] \rightarrow [\textit{Token}] \rightarrow \textit{Value}

\textit{scan} \ k \ \textit{clist} \ \textit{nlist} \ \textit{ts} =
\quad \textit{resume} \ \textit{clist} \ (k \ (\textit{head} \ \textit{ts}) : \textit{nlist}) \ \textit{ts}

(p \oplus q) \ k \ \textit{clist} \ \textit{nlist} =
\quad p \ k \ (q \ k : \textit{clist}) \ \textit{nlist}

\textit{resume} \ (k : \textit{clist}) \ \textit{nlist} \ \textit{ts} = k \ \textit{clist} \ \textit{nlist} \ \textit{ts}

\textit{resume} \ [ ] \ (k : \textit{nlist}) \ \textit{ts} = k \ \textit{nlist} \ [ ] \ (\textit{tail} \ \textit{ts})
Comparing ...

\[
match :: \\
\quad \text{Regexp} \rightarrow [\text{Regexp}] \rightarrow [\text{Regexp}] \rightarrow \text{String} \rightarrow \text{Bool}
\]

\[
match (\text{Seq} (\text{Lit} c) \ e_k) \ \text{clist} \ \text{nlist} \ s \ | \ (\text{head} \ s = c) = \\\n\quad \text{resume} \ \text{clist} \ (e_k : \text{nlist}) \ s
\]

\[
match (\text{Seq} (\text{Alt} \ e_1 \ e_2) \ e_k) \ \text{clist} \ \text{nlist} \ s = \\\n\quad match (\text{Seq} \ e_1 \ e_k) \ (\text{Seq} \ e_2 \ e_k : \text{clist}) \ \text{nlist} \ s
\]

\[
\text{resume} (c : \text{clist}) \ \text{nlist} \ s = match \ c \ \text{clist} \ \text{nlist} \ s
\]

\[
\text{resume} [\ ] (n : \text{nlist}) \ s = match \ n \ \text{nlist} [\ ] (\text{tail} \ s)
\]
Remarks

Discovering this implementation seems to depend on the insight that a normal form for the context of a parser is

\[ \text{Scan} \left( p_1 \oplus \ldots \oplus p_k \right) \oplus \left( \text{?} \Rightarrow g \right) \oplus \left( q_m \oplus \ldots \oplus q_1 \right) \]

– so that \( p_1, \ldots, p_k \) and \( g \) and \( q_1, \ldots, q_m \) correspond to \( nlist \) and \( k \) and \( clist \) respectively.

Can this insight (in general) replaced by a formal calculation? Why does the (more complicated) free monad implementation seem easier to find?
A zoo of control constructs

Similar remarks apply to:

- Backtracking: [Spivey & Seres; Hinze; Wand & Vaillancourt].
- Coroutine pipelines: [ICFP'17].
- … and now parser combinators.
Some dreams

• A symbolic reasoning tool that makes higher-order calculations easier (like Mathematica or Alpha), not harder (like any verification tool you know).

• An automated defunctionaliser that helps us to control and visualise the results.