Parallel Parsing Processes *Revisited*

Michael Spivey
University of Oxford

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!]

[1] Ken Thompson, "Programming Techniques: Regular expression search algorithm," CACM 11, 6 (June 1968), pp. 419--22.

Translating regexps

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

FAIL:

if *CLIST* != [ ] then pop and goto first element
else { advance *char*; *CLIST* = *NLIST*; *NLIST* = [ ] }

Thompson lite

$$match :: \\ \quad Regexp \to [Regexp] \to [Regexp] \to String \to Bool$$

$$match\ (Seq\ (Lit\ c)\ e_k)\ clist\ nlist\ s \mid (head\ s == c) = \\ \quad resume\ clist\ (e_k : nlist)\ s$$

$$match\ (Seq\ (Alt\ e_1\ e_2)\ e_k)\ clist\ nlist\ s = \\ \quad match\ (Seq\ e_1\ e_k)\ (Seq\ e_2\ e_k : clist)\ nlist\ s$$

...

$$resume\ (c{:}clist)\ nlist\ s\ =\ match\ c\ clist\ nlist\ s$$

$$resume\ [\,]\ (n{:}nlist)\ s\ =\ match\ n\ nlist\ [\,]\ (tail\ s)$$

Parser combinators

$$
\begin{aligned}
&\mathit{expr} = \\
&\quad \mathit{factor} \mathbin{\underline{\oplus}} \\
&\quad (\mathbf{do}\ a \leftarrow \mathit{factor};\ \mathit{eat}\ \text{'+'};\ b \leftarrow \mathit{expr};\ \underline{\mathit{return}}\ (\mathit{Plus}\ a\ b))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{factor} = \\
&\quad (\mathbf{do}\ x \leftarrow \mathit{ident};\ \underline{\mathit{return}}\ (\mathit{Var}\ x)) \mathbin{\underline{\oplus}} \\
&\quad (\mathbf{do}\ \mathit{eat}\ \text{'('};\ a \leftarrow \mathit{expr};\ \mathit{eat}\ \text{')'};\ \underline{\mathit{return}}\ a)
\end{aligned}
$$

$$
\mathit{eat}\ x = (\mathbf{do}\ y \leftarrow \underline{\mathit{scan}};\ \mathbf{if}\ x == y\ \mathbf{then}\ \underline{\mathit{return}}\ (\,)\ \mathbf{else}\ \underline{\mathit{fail}})
$$

- Can be implemented with state and backtracking
- Or …

Claessen [2004]

'Parallel' parser combinators

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.

[2] Koen Claessen, "Functional Pearl: Parallel parsing processes," JFP 14, 6 (2004), pp. 741--57.

Alternation – the vital idea

$Fail \oplus q = q$

$(Result\ x\ p') \oplus q = Result\ x\ (p' \oplus q)$

$(Scan\ g) \oplus Fail = Scan\ g$

$(Scan\ g) \oplus (Result\ x\ q') = Result\ x\ ((Scan\ g) \oplus q')$

$(Scan\ g) \oplus (Scan\ h) = Scan\ (\lambda\, x \rightarrow g\ x \oplus h\ x)$

- 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

$$return\ x = Result\ x\ fail$$

$$\begin{aligned}
(Result\ x\ p) &\gg= f = Result\ \mathrm{x}\ (p \gg= f) \\
(Scan\ g) &\gg= f = Scan\ (\lambda\, x \rightarrow g\ x \gg= f) \\
Fail &\gg= f = Fail
\end{aligned}$$

$$scan = Scan\ return$$

$$fail = Fail$$

- These are the operations (*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.

$$parse :: Parser\ \alpha \rightarrow [Token] \rightarrow \alpha$$

$$\begin{array}{llcl}
parse\ (Scan\ g) & [\,] & = & error\ \text{"unexpected EOF"} \\
parse\ (Scan\ g) & (t : ts) & = & parse\ (g\ t)\ ts \\
parse\ (Result\ x\ p) & [\,] & = & x \\
parse\ (Result\ x\ p) & ts & = & parse\ p\ ts \\
parse\ Fail & _ & = & error\ \text{"syntax error"}
\end{array}$$

- 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.

type $KParser\ \alpha = VCont\ \alpha \rightarrow CCont \rightarrow NCont \rightarrow Answer$

type $VCont\ \alpha = \alpha \rightarrow CCont \rightarrow NCont \rightarrow Answer$

type $CCont = NCont \rightarrow Answer$

type $NCont = Token \rightarrow CCont \rightarrow Answer$

type $Answer = [Token] \rightarrow Value$

- **newtype** is needed all over the place.

A slew of one-liners

The same five operations now have direct definitions.

$$\mathit{return}\ x\ k\ =\ k\ x$$

$$(p \gg\!= f)\ k\ =\ p\ (\lambda\, x \rightarrow f\ x\ k)$$

$$\mathit{fail}\ k\ \mathit{ck}\ =\ \mathit{ck}$$

$$(p \oplus q)\ k\ \mathit{ck}\ =\ (p\ k \cdot q\ k)\ \mathit{ck}\ =\ p\ k\ \underline{(\lambda\, \mathit{nk} \rightarrow q\ k\ \mathit{ck}\ \mathit{nk})}$$

$$\mathit{scan}\ k\ \mathit{ck}\ \mathit{nk}\ =\ \mathit{ck}\ \underline{(\lambda\, t \rightarrow \mathit{nk}\ t \cdot k\ t)}$$

Where did that come from?

Define *rep* :: *Parser* $\alpha \to$ *KParser* α by

$$rep\ (Scan\ g)\ k\ ck\ nk \ = \ ck\ (\lambda\, t \ \to \ nk\ t \cdot k\ t)$$

$$rep\ (Result\ x\ p)\ k\ ck\ nk \ = \ k\ x\ (rep\ p\ k\ ck)\ nk$$

$$rep\ Fail\ k\ ck\ nk = \ ck\ nk$$

Then all else follows!

Deriving bind and plus

In particular, we can prove inductively that

$$rep\ (p \gg= f)\ k \;=\; rep\ p\ (\lambda\, x \rightarrow rep\ (f\, x)\ k)$$

and

$$rep\ (p \oplus q)\ k\ ck \;=\; rep\ p\ k\ (rep\ q\ k\ ck)$$

These justify the new definitions of $\gg=$ and $\oplus$.

Driving the new parser

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

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

where

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

if $ts == [\,]$ **then** x **else** $ck\ nk\ ts$

$ck_0\ nk\ [\,]\ =\ error$ "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 *NConts* are created only by the expression

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

(with *k* and *nk* as free variables) and *CConts* only by the expression

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

and by promoting *NConts* to *CConts* when scanning.

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

Concrete continuations

$$
\begin{aligned}
&\mathit{scan}\ k\ \mathit{clist}\ \mathit{nlist}\ \mathit{ts} \ = \\
&\quad \mathit{resume}\ \mathit{clist}\ (k\ (\mathit{head}\ \mathit{ts}) : \mathit{nlist})
\end{aligned}
$$

$$\mathit{fail}\ k\ \mathit{clist}\ \mathit{nlist} \ = \ \mathit{resume}\ \mathit{clist}\ \mathit{nlist}$$

$$(p \oplus q)\ k\ \mathit{clist}\ \mathit{nlist} \ = \ p\ k\ (q\ k : \mathit{clist})\ \mathit{nlist}$$

$$
\begin{aligned}
&\mathit{resume}\ (k : \mathit{clist})\ \mathit{nlist}\ \mathit{ts} \ = \ k\ \mathit{clist}\ \mathit{nlist}\ \mathit{ts} \\
&\mathit{resume}\ [\,]\ \mathit{nlist}\ [\,] \ = \ \mathit{error}\ \text{"unexpected EOF"} \\
&\mathit{resume}\ [\,]\ \mathit{nlist}\ \mathit{ts} \ = \ \mathit{resume}\ (\mathit{reverse}\ \mathit{nlist})\ [\,]\ (\mathit{tail}\ \mathit{ts})
\end{aligned}
$$

- The *reverse* is needed because sometimes we care about the order of results.

Focussing ...

$$\textbf{type}\ \mathit{KParser}\ \alpha = \mathit{VCont}\ \alpha \to [\mathit{Cont}] \to [\mathit{Cont}] \to [\mathit{Token}] \to \mathit{Value}$$

$$\mathit{scan}\ k\ \mathit{clist}\ \mathit{nlist}\ \mathit{ts} = \mathit{resume}\ \mathit{clist}\ (k\ (\mathit{head}\ \mathit{ts}) : \mathit{nlist})\ \mathit{ts}$$

$$(p \oplus q)\ k\ \mathit{clist}\ \mathit{nlist} = p\ k\ (q\ k : \mathit{clist})\ \mathit{nlist}$$

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

$$\mathit{resume}\ [\,]\ (k : \mathit{nlist})\ \mathit{ts} = k\ \mathit{nlist}\ [\,]\ (\mathit{tail}\ \mathit{ts})$$

Comparing ...

$$\mathit{match} :: \mathit{Regexp} \to [\mathit{Regexp}] \to [\mathit{Regexp}] \to \mathit{String} \to \mathit{Bool}$$

$$\mathit{match}\ (\mathit{Seq}\ (\mathit{Lit}\ c)\ e_k)\ \mathit{clist}\ \mathit{nlist}\ s \mid (\mathit{head}\ s == c) = \mathit{resume}\ \mathit{clist}\ (e_k : \mathit{nlist})\ s$$

$$\mathit{match}\ (\mathit{Seq}\ (\mathit{Alt}\ e_1\ e_2)\ e_k)\ \mathit{clist}\ \mathit{nlist}\ s = \mathit{match}\ (\mathit{Seq}\ e_1\ e_k)\ (\mathit{Seq}\ e_2\ e_k : \mathit{clist})\ \mathit{nlist}\ s$$

$$\mathit{resume}\ (c : \mathit{clist})\ \mathit{nlist}\ s \ = \ \mathit{match}\ c\ \mathit{clist}\ \mathit{nlist}\ s$$

$$\mathit{resume}\ [\,]\ (n : \mathit{nlist})\ s \ = \ \mathit{match}\ n\ \mathit{nlist}\ [\,]\ (\mathit{tail}\ s)$$

Remarks

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

$$Scan\ (p_1 \oplus\ \ldots \oplus p_k) \oplus (? \gg= g) \oplus (q_m \oplus \ldots \oplus q_1)$$

– 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.