Potential Fixedpoint Semantics for PEGREG

Jacob Salzberg

September 22, 2022

Progress on pegreg stalled because of concerns about the correctness of the algorithm that created an FST for possessive star.

Nonetheless, I believe it is still possible to define possessive star in finite state machines:

Parsing expression grammars are usually defined with respect to their input string, in effect embedding a "continuation" string into the semantics of a parsing expression grammar.

In the paper "Towards Typed Semantics for Parsing Expression Grammars" (2019), Rebeiro et. al. present an operational semantics for PEG that includes a left-recursive version of the star operator. I believe this semantics can be encoded as a fixpoint, viewing the strings "to be matched" as the set of all strings, then shown equivalent to the following rules:

$R_p[\![\langle \mathrm{ch}, \epsilon\rangle]\!] = \{\mathrm{ch}\}$	character literal
$R_p[\![\langle \mathrm{ch}, e_2\rangle]\!] = \{\mathrm{ch}.s \mid s \in R_p[\![\langle e_2, \epsilon\rangle]\!]\}$	character literal with continuation
$R_p[\![\langle e_1.e_2, e_3\rangle]\!] = R_p[\![\langle e_1, e_2.e_3\rangle]\!]$	concatenation
$R_p[\![\langle e_1/e_2, e_3\rangle]\!] = \{s_1s_3, s_2s_3 \mid s_1 \in R_p[\![\langle e_1, \epsilon\rangle]\!], s_3 \in R_p[\![\langle e_3, \epsilon\rangle]\!], s_2 \in R_p[\![\langle e_2, \epsilon\rangle]\!] \setminus R_p[\![\langle e_1, \epsilon\rangle]\!]\}$	ordered choice
$F_p[\![\langle e_1*, \epsilon\rangle]\!](X) = \{s_1 \mid s_1 \in R_p[\![\langle e_1, \epsilon\rangle]\!]\} \cup \{s_1s_2 \mid s_1 \in X \wedge s_2 \in X\}$	greedy repitition
$R_p[\![\langle e_1*, \epsilon\rangle]\!] = \mathrm{lfp}\, F_p$	$*\epsilon$-case
$R_p[\![\langle e_1*, e_2\rangle]\!] = \{s_1s_2 \mid s_1 \in R_p[\![\langle e_1*, \epsilon\rangle]\!], s_2 \in R_p[\![\langle e_2, \epsilon\rangle]\!] \setminus R_p[\![\langle e_1, \epsilon\rangle]\!]\}$	$*$general case

The set minus operation can then be encoded in a finite state machine: $L_1 \setminus L_2 = L_1 \cap \neg(L_2) = \neg(L_1 \cup \neg(L_2))$.

Update Sept. 22:

Perhaps something along the lines of the following could work:

From right to left, make the expression into an automaton, starting with the leftmost character. Explore the automaton m, from the start state q0 to all of the characters that can be reached from the PEG expression.

Because the automaton is finite, this will always terminate. From each complete run of the PEG expression on the automaton, collect a "frontier" of reached nodes. From this frontier, find all paths that follow; the automaton form the start state, thorugh the explored paths, then through the paths that follow, to give us a new automaton m'. Compute the automaton $L(m)$ $L(m')$.

Recurse on the remainder of the PEG expression. Concatenate the result of this recursion to your final automaton.

```
Algorithm:

PEGREG(PEG) -- outputs a FSM from a PEG
PEGREG_RECURSIVE(PEG, frontier, paths, automaton) -- Outputs an FSM from a PEG, recursing.

PEGREG() = empty language automaton
PEGREG(c) = the automaton matching {c}
PEGREG(x/y) =
  let y_automaton = PEGREG_RECURSIVE(y) in
    PEGREG(x)|PEGREG_RECURSIVE(x, {q0}, {q0}, y_automaton)
PEGREG(x*) =
  PEGREG_RECURSIVE(x, {q0}, {q0}, empty language automaton)
PEGREG(x.y) where y is the rightmost concatenated element =
  let y_automaton = PEGREG_RECURSIVE(y) in
    PEGREG(x).PEGREG_RECURSIVE(x, {q0}, {q0}, y_automaton)

PEGREG_RECURSIVE(, frontier, p, a) =
  from frontier explore all paths in a
  Concatenate these paths to p, resulting in an automaton aprefix
  Compute the automaton for the language L(a) \ L(aprefix), aout
  return aout
PEGREG_RECURSIVE(c, frontier, p, a) =
  for every transition (s0, c, s1) in a, add s1 to the frontier, and,
  if there is no transition (s0, c, s0) in a, where s0 is the one mentioned
  above, remove s0 from the frontier
  return PEGREG_RECURSIVE(, frontier, p, a)
PEGREG_RECURSIVE(x*, frontier, p, a)
  let a0 = PEGREG_RECURSIVE(x, frontier, p, a)
  until you reach a fixedpoint, an, compute
    ai + 1 = PEGREG_RECURSIVE(x, {q0}, {q0}, ai)
  return an
PEGREG_RECURSIVE(l/r, frontier, p, a)
  compute al = PEGREG_RECURSIVE(l, frontier, p, a)
  PEGREG_RECURSIVE(r, {q0}, {q0}, al)
PEGREG_RECURSIVE(x.y, frontier, p, a) where y is the rightmost concatenated expr. =
  a1 = PEGREG_RECURSIVE(x, frontier, p, a)
  PEGREG_RECURSIVE(y, {q0}, {q0}, a1)
```