Regular expression matching and operational semantics

Hayo Thielecke
joint work with Asiri Rathnayake

University of Birmingham, UK

Introduction: regular expression matching

- Regular expressions are everywhere
- In compiling: convert to Deterministic Finite Automaton (DFA) via tools like lex
- DFA matching is efficient, linear time
- Java, Perl etc do not build a DFA but use backtracking
- Naive backtracking:
  given $(e_1 \mid e_2)$, try to match e_1 and then if it fails try e_2
- functional programming classic: failure continuations
- But backtracking is inefficient: exponential
- In Java: matching $\underbrace{\texttt{a...a}}_{1000}$ `b` to `a**` takes forever

Thompson's lockstep matcher

- Thomson's lockstep matcher from 1968 is more efficient than backtracking
- Interpret the regular expression as a non-deterministic program
- Cox: "Name the most widely used bytecode interpreter or virtual machine."
- Lockstep interpretation avoids exponential blowup
- Thomson's construction can be seen as constructing an NFA (Non-deterministic Finite Automaton)
- But looking at it as merely a naive NFA misses key ideas
- Topic of this talk: use programming language technology, not so much automata theory

Preliminaries and background

- My background is in programming language theory (not automata theory)
- This talk is an extended version of our paper in Structural Operation Semantics 2011
- Operational semantics is much like writing an interpreter or `eval` in Lisp or Javascript
- Big-step semantics: $((1+2)+3) \Downarrow 6$
- Small step semantics $((1+2)+3) \to (3+3) \to 6$
- $\Downarrow$ and $\to$ are relations, not always functions: non-determinism
- Compare derivations in grammars

$$S \Rightarrow E + E \Rightarrow 1 + E \Rightarrow 1 + 2$$

Automata $\leftrightarrow$ operational semantics

Automata theory

- $p \xrightarrow{a} q$, where $a \in \Sigma$, $p, q \in Q$
- Typical questions: languages closed under complement
- p and q are atomic states, elements of a *finite* set Q without further structure

Programming language theory: operational semantics

- Structural: $\dfrac{P \to Q}{P\;K \to Q\;K}$ or $\langle P, K\rangle \to \langle Q, K\rangle$
- P, Q and K are defined by induction
- $\to$ is defined by induction following their structure (SOS)

Structure of the paper — not everything will be in the talk

Regular expression matching as big-step semantics

↓ Small step with continuations

EKW machine

↓ Pointer representation

PWπ machine

↓ Macro steps

Generic lockstep construction

Sequential scheduling → Sequential matcher

Parallel scheduling → Parallel matcher

↕ Processes

Implementation on Graphics Processor

Matching as big-step sematics

$$\frac{e_1 \downarrow w_1 \qquad e_2 \downarrow w_2}{(e_1\ e_2) \downarrow (w_1\ w_2)}\ (\textsc{Seq}) \qquad \frac{}{a \downarrow a}\ (\textsc{Match}) \qquad \frac{}{\varepsilon \downarrow \varepsilon}\ (\textsc{Epsilon})$$

$$\frac{e \downarrow w_1 \qquad e^* \downarrow w_2}{e^* \downarrow (w_1\ w_2)}\ (\textsc{Kleene1}) \qquad \frac{}{e^* \downarrow \varepsilon}\ (\textsc{Kleene2})$$

$$\frac{e_1 \downarrow w}{(e_1 \mid e_2) \downarrow w}\ (\textsc{Alt1}) \qquad \frac{e_2 \downarrow w}{(e_1 \mid e_2) \downarrow w}\ (\textsc{Alt2})$$

Like denotational sematics — but how do you match efficiently?

The EKW machine

$$\boxed{\langle e \,;\, k \,;\, w \rangle \to \langle e' \,;\, k' \,;\, w' \rangle}$$

$$\langle e_1 \mid e_2 \,;\, k \,;\, w \rangle \to \langle e_1 \,;\, k \,;\, w \rangle$$

$$\langle e_1 \mid e_2 \,;\, k \,;\, w \rangle \to \langle e_2 \,;\, k \,;\, w \rangle$$

$$\langle e_1\, e_2 \,;\, k \,;\, w \rangle \to \langle e_1 \,;\, e_2 :: k \,;\, w \rangle$$

$$\langle e^* \,;\, k \,;\, w \rangle \to \langle e \,;\, e^* :: k \,;\, w \rangle$$

$$\langle e^* \,;\, k \,;\, w \rangle \to \langle \varepsilon \,;\, k \,;\, w \rangle$$

$$\langle a \,;\, k \,;\, a\, w \rangle \to \langle \varepsilon \,;\, k \,;\, w \rangle$$

$$\langle \varepsilon \,;\, e :: k \,;\, w \rangle \to \langle e \,;\, k \,;\, w \rangle$$

Regular expressions as trees/graphs in memory

Expression as trees

We need to distinguish the two a positions in

$$(a\, b) \mid (a\, c)$$

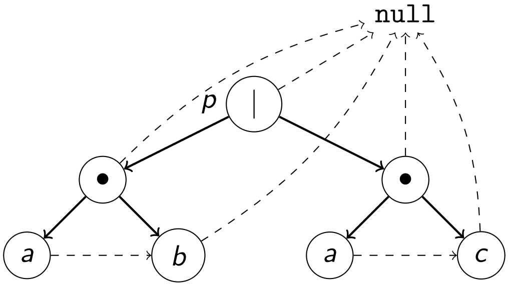
Represent syntax tree as heap with pointers, using ideas from Separation Logic.

Continuation pointers

The continuation (k in the EKW machine) is determined by the position in the tree.
Hardwire the continuation in the tree as a pointer.

Syntax tree in memory: $(a\,b) \mid (a\,c)$

$\longrightarrow$ child nodes in syntax tree
$\dashrightarrow$ continuation pointers



Processes

Interleaved communication for evolving pointers p (CCS)

$\boxed{M \longrightarrow M'}$

$$\frac{M_1 \longrightarrow M_2}{M_1 \parallel M_3 \longrightarrow M_2 \parallel M_3} \qquad \frac{}{((p \,.\, M) \parallel \overline{p}) \longrightarrow M}$$

Synchronous step for lockstep matching of symbol a (SCCS)

$\boxed{M \xrightarrow{a} M'}$

$$\frac{M' \not\equiv (\$\, a \,.\, M') \parallel M''' \qquad M' \not\longrightarrow}{(\$\, a \,.\, M_1 \parallel \ldots \parallel \$\, a \,.\, M_n \parallel M') \xrightarrow{a} (M_1 \parallel \ldots \parallel M_n)}$$

Process translation

Each node p in the syntax tree become a process $[\![p]\!]\,\pi$ for recognizing that expression.

$$
\begin{array}{ll}
[\![p]\!]\,\pi = p\,.\,(\overline{q_1} \parallel \overline{q_2}) & \text{if } \pi(p) = (q_1 \mid q_2) \\
[\![p]\!]\,\pi = p\,.\,\overline{q_1} & \text{if } \pi(p) = (q_1 \bullet q_2) \\
[\![p]\!]\,\pi = p\,.\,(\overline{q_1} \parallel \overline{q_2}) & \text{if } \pi(p) = {q_1}^{*} \text{ and } \mathtt{cont}\, p = q_2 \\
[\![p]\!]\,\pi = p\,.\,\overline{q} & \text{if } \pi(p) = \varepsilon \text{ and } \mathtt{cont}\, p = q \\
[\![p]\!]\,\pi = p\,.\,\$\,a\,.\,\overline{q} & \text{if } \pi(p) = a \text{ and } \mathtt{cont}\, p = q
\end{array}
$$

Example: $(a\,b) \mid (a\,c)$ with input $a\,b$

$$p.(\overline{p_1} \parallel \overline{p_2})$$

$p_1.\$a.\overline{p_3}$ $\quad$ $p_2.\$a.\overline{p_4}$

$p_3.\$b.\overline{p_5}$ $\quad$ $p_4.\$c.\overline{p_5}$

Remaining input: $a\,b$
p is sent

Example: $(a\,b) \mid (a\,c)$ with input $a\,b$

$$\overline{p_1} \parallel \overline{p_2}$$

$p_1.\$a.\overline{p_3}$ $\quad$ $p_2.\$a.\overline{p_4}$

$p_3.\$b.\overline{p_5}$ $\quad$ $p_4.\$c.\overline{p_5}$

Remaining input: $a\,b$

p_2 is sent

Example: $(a\, b) \mid (a\, c)$ with input $a\, b$

$\overline{p_1}$

$p_1.\$a.\overline{p_3}$ $\$a.\overline{p_4}$

$p_3.\$b.\overline{p_5}$ $p_4.\$c.\overline{p_5}$

Remaining input: $a\, b$

p_1 is sent

Example: $(a\, b) \mid (a\, c)$ with input $a\, b$

$\$a.\overline{p_3}$ $\quad$ $\$a.\overline{p_4}$

$p_3.\$b.\overline{p_5}$ $\quad$ $p_4.\$c.\overline{p_5}$

Remaining input: $a\, b$

a is matched synchronously

Example: $(a\,b) \mid (a\,c)$ with input $a\,b$

$\overline{p_3}$ $\overline{p_4}$

$p_3.\$b.\overline{p_5}$ $p_4.\$c.\overline{p_5}$

Remaining input: b

p_3 is sent

Example: $(a\,b) \mid (a\,c)$ with input $a\,b$

$\overline{p_4}$

$\$b.\overline{p_5}$ $\quad$ $p_4.\$c.\overline{p_5}$

Remaining input: b

p_4 is sent

Example: $(a\,b) \mid (a\,c)$ with input $a\,b$

$\$b.\overline{p_5}$ $\$c.\overline{p_5}$

Remaining input: b

b is matched synchronously

Example: $(a\, b) \mid (a\, c)$ with input $a\, b$

$\overline{p_5}$

Remaining input: empty

Successful match if p_5 is the final continuation

Main correctness result

- Theorem: our concurrent matcher is correct.
- Like computing the ε closure in automata theory
- But concurrently.
- Proof relies on invariant using

$$\Box\, P = \{ q \mid \exists p \in P . p \longrightarrow \cdots \longrightarrow q \wedge q \not\rightarrow \}$$

- $\Box\, P$ not quite a closure operator, as some elements are removed.
- $\Box\, P$ is a kind of tree modality

Summary

- We have formalized regular expression matching using abstract machines
- Thompson's technique from 1968 - more efficient than regex matchers in Perl or Java
- View expression as program and run it on an interpreter.
  - continuations: $(a\,b) \mid (a\,c)$ leads to stack holding b or c
  - pointers via separation logic: two copies of a at different addresses, intentionally $\neq$
  - processes: alternation $\mid$ becomes parallel composition, lockstep as in SCCS
- Bridge-building between programming language theory and automata theory

Further work 1: extended reg exp matching

- Machines can be extended to non-regular constructs used in practice
- Examples: submatching, back references: no longer regular
- `a(b)\1` and `(ab)\1` are not equivalent due to the backreference
- We try to adapt Thompson's technique to such constructs
- Compare: abstract machines for impure languages (e.g., SECD machine + assignment)
- Analogy: DFA $\cong$ λ, non-regular $\cong$ effects
- Some current and future work, perhaps Deep Packet Inspection for security

Further work 2: GPGPU programming and op sem

- GPGPU = General Purpose Graphics Processing Unit
- A leading example of non-numeric GPGPU is state machines
- We made a toy reg exp implementation in CUDA based on message passing
- But existing NFA matchers like iNFAnt are faster: optimized, ε transitions eliminated, data structure well suited to GPU memory
- We are working on abstract machine and operational semantics techniques for multi-core and GPGPU
- GPUs make both concurrent and interleaved transitions