Towards certified virtual machine-based regular expression parsing

Thales Delfino[1] **Rodrigo Ribeiro**[1]

[1]Departament of Computer Science
Universidade Federal de Ouro Preto

September 20, 2018

Introduction

- Parsing is pervasive in computing
  - String search tools, lexical analysers...
  - Binary data files like images, videos ...
- Our focus: Regular Languages (RLs)
  - Languages denoted by Regular Expressions (REs) and equivalent formalisms

Introduction

- Approaches for RE parsing:
  - Representation using FSM.
  - Derivatives for RE.
- Other approach: use of VM.
  - Pioneered by Knuth in the 70's for top-down parsing of CFG.
  - Revived by Cox in the context of REs.

Introduction

- RE VM by Cox.
  - RE are high-level programs executed by the VM.
  - RE are compiled to a sequence of VM instructions.
- Problems with Cox's VM:
  - Poorly specified, no correctness guarantees.
  - No disambiguation strategy specified.
- Our work:
  - A small-step operational semantics for RE parsing.
  - Semantics similar to abstract machines for λ-calculus (e.g. SECD and Krivine's machines).

Our contributions

- A small-step semantics for RE parsing inspired by Thompson's NFA construction.
- Prototype implementation of the semantics in Haskell.
- Use of property-based testing to verify it against a simple (and correct) implementation of RE parsing by Fisher et. al.
- Our semantics outputs bit-codes to represent parse trees for REs. We use Quickcheck to verify that produced codes correspond to valid parsing evidence

Background — RE Syntax

- RE Syntax

$$e ::= \emptyset \mid \epsilon \mid a \mid e\,e \mid e + e \mid e^\star$$

- Haskell Code

$$\textbf{data}\ \text{Regex} = \emptyset \mid \epsilon \mid \text{Chr Char} \mid \text{Regex} \bullet \text{Regex}$$
$$\mid \text{Regex} + \text{Regex} \mid \text{Star Regex}$$

Background - RE Semantics

$$\frac{}{\epsilon \in [\![\epsilon]\!]}\ \{Eps\} \qquad \frac{a \in \Sigma}{a \in [\![a]\!]}\ \{Chr\}$$

$$\frac{s \in [\![e]\!]}{s \in [\![e + e']\!]}\ \{Left\} \qquad \frac{s' \in [\![e']\!]}{s' \in [\![e + e']\!]}\ \{Right\}$$

$$\frac{}{\epsilon \in [\![e^\star]\!]}\ \{StarBase\} \qquad \frac{s \in [\![e]\!] \quad s' \in [\![e^\star]\!]}{ss' \in [\![e^\star]\!]}\ \{StarRec\}$$

$$\frac{s \in [\![e]\!] \quad s' \in [\![e']\!]}{ss' \in [\![ee']\!]}\ \{Cat\}$$

Parse trees for REs

- We interpret RE as types and parse tree as terms.
- Informally:
  - leafs: empty string and character.
  - concatenation: pair of parse trees.
  - choice: just the branch of chosen RE.
  - Kleene star: list of parse trees.
- In Haskell:

$$\textbf{data}\ \mathsf{Tree} = () \mid \mathsf{Chr}\ \mathsf{Char} \mid \mathsf{Tree} \bullet \mathsf{Tree} \mid \mathsf{InL}\ \mathsf{Tree} \mid \mathsf{InR}\ \mathsf{Tree} \mid \mathsf{List}\ [\mathsf{Tree}]$$

Parse trees for RE — Example

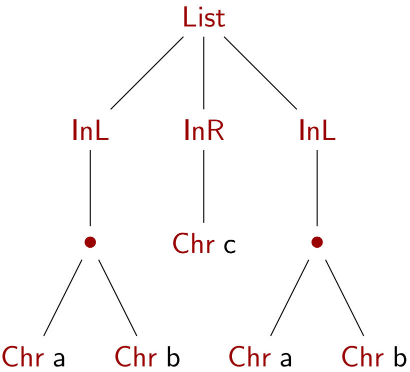


Figure: Parse tree for RE: $(ab + c)^{\star}$ and the string $w = abcab$.

Parse trees typing relation

$$\frac{}{\vdash () : \epsilon} \qquad \frac{}{\vdash \mathsf{Chr}\ \mathsf{a} : a} \qquad \frac{\vdash \mathsf{t} : e}{\vdash \mathsf{InL}\ \mathsf{t} : e + e'}$$

$$\frac{\vdash \mathsf{t}' : e'}{\vdash \mathsf{InR}\ \mathsf{t}' : e + e'} \qquad \frac{\vdash \mathsf{t} : e \quad \vdash \mathsf{t}' : e'}{\vdash \mathsf{t} \bullet \mathsf{t}' : ee'} \qquad \frac{\forall \mathsf{t}.\mathsf{t} \in \mathsf{ts} \rightarrow \vdash \mathsf{t} : e}{\vdash \mathsf{List}\ \mathsf{ts} : e^\star}$$

Relating parse trees and RE semantics

- Using function flat.
- Property: Let t be a parse tree for a RE e and a string s. Then, $flat(t) = s$ and $s \in [\![e]\!]$.

```
flat :: Tree → String
flat () = ""
flat (Chr c) = [c]
flat (t • t′) = flat t ++ flat t′
flat (InL t) = flat t
flat (InR t) = flat t
flat (List ts) = concatMap flat ts
```

Bit-codes for parse trees

- Instead of using parse trees...
  - We can use bit-codes in order to build memory efficient representations of evidence.
- Bit-codes mark...
  - which branch of choice was chosen during parsing: 0_b for left ; 1_b for right.
  - matchings done by the Kleene star operator: 0_b marks the beginning of a new match; 1_b finish the list of matchings.

Bit codes as parse trees for RE — Example

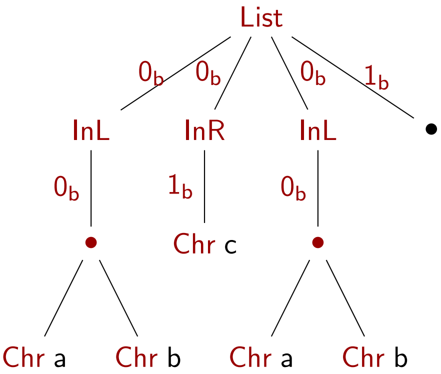


Figure: Parse tree for RE: $(ab + c)^{\star}$ and the string $w = abcab$.

Relating bit-codes and REs

- Typing relation for bit-codes.

$$\frac{}{[\,] \rhd \epsilon} \qquad \frac{}{[\,] \rhd a} \qquad \frac{\mathsf{bs} \rhd e}{0_b : \mathsf{bs} \rhd e + e'}$$

$$\frac{\mathsf{bs} \rhd e'}{1_b : \mathsf{bs} \rhd e + e'} \qquad \frac{\mathsf{bs} \rhd e \quad \mathsf{bs}' \rhd e'}{\mathsf{bs} +\!\!+ \mathsf{bs}' \rhd ee'} \qquad \frac{}{[1_b] \rhd e^\star}$$

$$\frac{\mathsf{bs} \rhd e \quad \mathsf{bss} \rhd e^\star}{0_b : \mathsf{bs} +\!\!+ \mathsf{bss} \rhd e^\star}$$

Relating bit-codes and parse trees

- Using functions code and decode.

  type Code = [Bit]

  code :: Regex → Tree → Code
  decode :: Regex → Code → Maybe Tree

- Correctness property:
  - if $\vdash t : e$ then (code e t) $\rhd$ e
  - decode e (code e t) $\equiv$ Just t

Proposed semantics — (I)

- We use evaluation contexts to represent how to reduce an input RE.
- Context syntax:

$$E[\,] \rightarrow E[\,] + e \mid e + E[\,] \mid E[\,]\,e \mid e\,E[\,] \mid \star$$

- We represent contexts using zippers (data type derivatives) for RE data type:

$$\textbf{data}\ \text{Hole} = \text{InChoiceL Regex} \mid \text{InChoiceR Regex} \mid \text{InCatL Regex} \mid \text{InCatR Regex} \mid \text{InStar}$$

Proposed semantics — (II)

- Semantics judgment express transitions between configurations: $c \to c'$
- Parse errors $\Rightarrow$ stuck states.

Proposed semantics — (III)

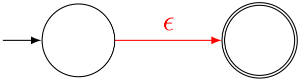
- Configurations of the form $\langle d, e, c, b, s\rangle$ are built from:
  - d is a direction, which specifies if the semantics is starting (denoted by B) or finishing (F) the processing of the current expression e.
  - e is the current expression being evaluated;
  - c is a context in which e occurs. Contexts are just a list of Hole type in our implementation.
  - b is a bit-code for the current parsing result, in reverse order.
  - s is the input string currently being processed.
- Acceptance configurations: $\langle F, e, [], b, \epsilon\rangle$

Proposed semantics — (III)

- Rule for Eps:

$$\frac{}{\langle B, \epsilon, c, b, s\rangle \rightarrow \langle F, \epsilon, c, b, s\rangle}\ (Eps)$$

- Corresponding NFA transition:

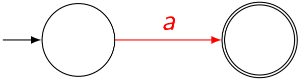


Proposed semantics — (IV)

- Rule for Chr:

$$\frac{}{\langle B, a, c, b, a : s\rangle \rightarrow \langle F, a, c, b, s\rangle}\ (Chr)$$

- Corresponding NFA transition:

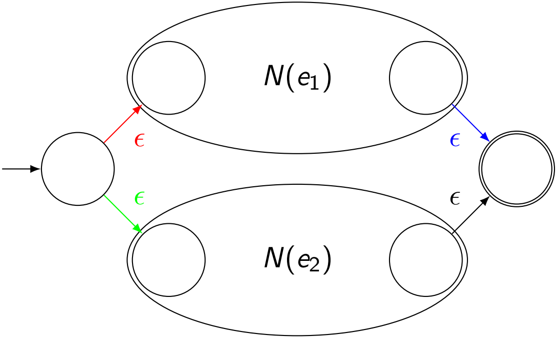


Proposed semantics — (V)

- Trying the left hand side of $e_1 + e_2$.

$$\frac{c' = E[\,] + e' : c}{\langle B, e + e', c, b, s\rangle \rightarrow \langle B, e, c', b, s\rangle} \; (Left_B)$$

- Transition in red.

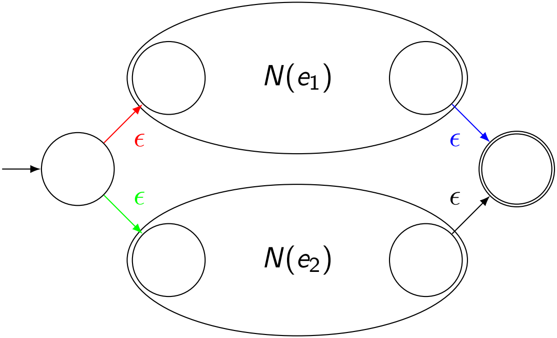


Proposed semantics — (VI)

- Finishing the left hand side of $e_1 + e_2$.

$$\frac{c = E[\,] + e' : c'}{\langle F, e, c, b, s\rangle \rightarrow \langle F, e + e', c', 0_b : b, s\rangle}\ (Left_E)$$

- Transition in blue.



Test suite

- We use Quickcheck to generate random non-problematic REs.
  - Problematic REs have the form $e^{\star}$ where $\epsilon \in [\![e]\!]$.
  - Our semantics can be extended to problematic REs straightforwardly.
- For a given RE, we have random generators for accepted and rejected strings.

Properties tested

- Our semantics accepts only and all the strings in the language described by the input RE.
  - Generating random strings that should be accepted.
  - Generating random strings that should be rejected.

Properties tested

- Our semantics generates valid parsing evidence:
  - the bit-codes can be parsed into a valid parse tree t for the random produced RE e, i.e. $\vdash t : e$ holds;
  - flat t = s and
  - code e t = bs.

Code coverage results

- 99% of code coverage by the test suite.

Top Level Definitions			Alternatives			Expressions		
%	covered / total		%	covered / total		%	covered / total	
100%	3/3		100%	10/10		100%	74/74	
100%	4/4		100%	18/18		97%	163/167	
-	0/0		-	0/0		-	0/0	
100%	7/7		100%	21/21		100%	173/173	
100%	7/7		100%	25/25		100%	142/142	
100%	21/21		100%	74/74		99%	552/556	

Current status

- We have a Coq formalization of a correct interpreter for this semantics.
- Current work:
  - On going formalization of the equivalence between the proposed semantics and the standard RE semantics.
  - Proof that the semantics follows the greedy disambiguation strategy.

Conclusion

- We developed a small-step semantics for RE parsing inspired by classical results of automata theory.
- We use property-based testing to check relevant properties of the semantics, before using a proof-assistant to mechanize the results.
- Next steps:
  - Finish Coq proofs and improve efficiency.