Certified Bit-Coded Regular Expression Parsing

Rodrigo Ribeiro[1] André Rauber Du Bois[2]

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

[2]Departament of Computer Science
Universidade Federal de Pelotas

September 21, 2017

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

- Is RE parsing a yes / no question?.
  - No! Better to produce evidence: parse trees.
- Why use bit-codes for parse trees?
  - Memory compact representation of parsing evidence.
  - Easy serialization of parsing results.

Contributions

- We provide fully certified proofs of a derivative based algorithm that produces a bit representation of a parse tree.
- We mechanize results about the relation between RE parse trees and bit-codes.
- We provide sound and complete decision procedures for prefix and substring matching of RE.
- Coded included in a RE search tool developed by us — verigrep.
- All results formalized in Agda version 2.5.2.

Regular Expression Syntax

- Definition of RE over a finite alphabet Σ.

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

- Agda code

```
data Regex : Set where
  ∅ : Regex
  ϵ : Regex
  $_ : Char → Regex
  _●_ : Regex → Regex → Regex
  _+_ : Regex → Regex → Regex
  _⋆ : Regex → Regex
```

Regular Expression Semantics

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

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

$$\frac{s' \in [\![e']\!]}{s' \in [\![e + e']\!]} \qquad \frac{s \in [\![\epsilon + ee^\star]\!]}{s \in [\![e^\star]\!]}$$

Regular Expression Semantics — Agda code

```
data _∈⟦_⟧ : List Char → Regex → Set where
  ε     : [] ∈⟦ ε ⟧
  $_    : (c : Char) → $ c ∈⟦ $ c ⟧
  _•_   : s ∈⟦ l ⟧ →
          s′ ∈⟦ r ⟧ →
          (s ++ s′) ∈⟦ l • r ⟧
  _+L_  : s ∈⟦ l ⟧ → s ∈⟦ l + r ⟧
  -- some code omitted...
```

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.

Parse trees for RE — Example

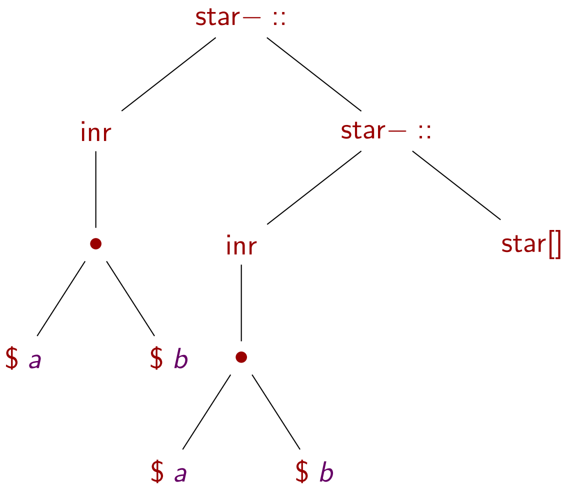

Figure: Parse tree for RE: $(c + ab)^\star$ and the string $w = abab$.

Parse trees for REs

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Tree} : \mathsf{Regex} \to \mathsf{Set}\ \textbf{where} \\
\quad \epsilon : \mathsf{Tree}\ \epsilon \\
\quad \$\_ : (c : \mathsf{Char}) \to \mathsf{Tree}\ (\$\ c) \\
\quad \mathsf{inl} : \mathsf{Tree}\ l \to \mathsf{Tree}\ (l + r) \\
\quad \mathsf{inr} : \mathsf{Tree}\ r \to \mathsf{Tree}\ (l + r) \\
\quad \_\bullet\_ : \mathsf{Tree}\ l \to \mathsf{Tree}\ r \to \mathsf{Tree}\ (l \bullet r) \\
\quad \mathsf{star[]} : \mathsf{Tree}\ (l \star) \\
\quad \mathsf{star}{-}{::} : \mathsf{Tree}\ l \to \mathsf{Tree}\ (l \star) \to \mathsf{Tree}\ (l \star)
\end{array}
$$

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, $\mathit{flat}(t) = s$ and $s \in [\![e]\!]$.

$$
\begin{array}{lcl}
\mathit{flat}(\epsilon) & = & [] \\
\mathit{flat}(\$\ a) & = & [a] \\
\mathit{flat}(\mathrm{inl}\ t) & = & \mathit{flat}(t) \\
\mathit{flat}(\mathrm{inr}\ t) & = & \mathit{flat}(t) \\
\mathit{flat}(t \bullet t') & = & \mathit{flat}(t) {+}{+} \mathit{flat}(t') \\
\mathit{flat}(\mathrm{star}[]) & = & [] \\
\mathit{flat}(\mathrm{star}- :: t\ ts) & = & \mathit{flat}(t) {+}{+} \mathit{flat}(ts)
\end{array}
$$

Relating parse trees and RE semantics

- flat type ensure its correctness property!

```
flat : Tree e → ∃ (λ xs → xs ∈ ⟦ e ⟧)
flat ϵ = [] , ϵ
flat ($ c) = [ c ] , ($ c)
flat (inl r t) with flat t
...| xs , prf = _ , r +L prf
flat (inr l t) with flat t
...| xs , prf = _ , l +R prf
flat (t • t′) with flat t | flat t′
...| xs , prf | ys , prf′ = _ , (prf • prf′)
-- some code omitted
```

Bit-codes for parse trees

- Bit-codes mark...
  - which branch of choice was chosen during parsing.
  - matchings done by the Kleene star operator.
- Predicate relating bit-codes to its RE.

data _IsCode_ : List Bit → Regex → Set **where**

ϵ : [ ] IsCode ϵ

$\$\_$: $(c$: Char$)$ → [ ] IsCode $(\$\ c)$

inl : xs IsCode l → $(0_b :: xs)$ IsCode $(l + r)$

inr : xs IsCode r → $(1_b :: xs)$ IsCode $(l + r)$

$\_\bullet\_$: xs IsCode l → ys IsCode r → $(xs \mathbin{++} ys)$ IsCode $(l \bullet r)$

star[] : $[\ 1_b\ ]$ IsCode $(l\ \star)$

star−:: : xs IsCode l → xss IsCode $(l\ \star)$ →
  $(0_b :: xs \mathbin{++} xss)$ IsCode $(l\ \star)$

How to relate bit-codes and parse trees?

- Function code builds bit-codes for parse trees.

```
code : Tree e → ∃ (λ bs → bs IsCode e)
code ($ c) = [] , ($ c)
code (inl r t) with code t
...| ys , pr = 0b :: ys , inl r pr
code (inr l t) with code t
...| ys , pr = 1b :: ys , inr l pr
code star[] = 1b :: [] , star[]
code (star- :: t ts) with code t | code ts
...| xs , pr | xss , prs = (0b :: xs ++ xss) , star- :: pr prs
-- some code omitted
```

How to relate bit-codes and parse-trees?

- Function decode parses a bit string for w.r.t. a RE.
- Property: forall t, decode (code t) $\equiv t$

$$
\begin{array}{l}
\mathsf{decode} : \exists\,(\lambda\ bs \rightarrow bs\ \mathsf{IsCode}\ e) \rightarrow \mathsf{Tree}\ e \\
\mathsf{decode}\ (\_\,,\ (\$\ c)) = \$\ c \\
\mathsf{decode}\ (\_\,,\ (\mathsf{inl}\ r\ pr)) = \mathsf{inl}\ r\ (\mathsf{decode}\ (\_\,,\ pr)) \\
\mathsf{decode}\ (\_\,,\ (\mathsf{inr}\ l\ pr)) = \mathsf{inr}\ l\ (\mathsf{decode}\ (\_\,,\ pr)) \\
\mathsf{decode}\ \mathsf{star}[] = (\_ +\mathsf{L}\ \epsilon)\star \\
\mathsf{decode}\ (\mathsf{star}{-} :: pr\ pr')\ \textbf{with}\ \mathsf{decode}\ (\_\,, pr) \mid \mathsf{decode}\ (\_\,, pr') \\
\ldots \mid pr1 \mid pr2 = (\_ +\mathsf{R}\ (pr1 \bullet pr2))\star
\end{array}
$$

```
-- some code omitted
```

Bit-codes for RE parse trees

- Building a parse tree for compute bit-codes is expansive.
- Better idea: build bit-codes during parsing, instead of computing parse trees.
- How? Just attach bit-codes to RE.

```
data BitRegex : Set where
  empty  : BitRegex
  eps    : List Bit → BitRegex
  char   : List Bit → Char → BitRegex
  choice : List Bit → BitRegex → BitRegex → BitRegex
  cat    : List Bit → BitRegex → BitRegex → BitRegex
  star   : List Bit → BitRegex → BitRegex
```

Relating REs and BREs

- Function internalize converts a RE into a BRE.

$$
\begin{aligned}
&\mathsf{internalize} \;:\; \mathsf{Regex} \;\rightarrow\; \mathsf{BitRegex}\\
&\mathsf{internalize}\;\emptyset \;=\; \mathsf{empty}\\
&\mathsf{internalize}\;\epsilon \;=\; \mathsf{eps}\;[\,]\\
&\mathsf{internalize}\;(\$\;x) \;=\; \mathsf{char}\;[\,]\;x\\
&\mathsf{internalize}\;(e \bullet e') \;=\; \mathsf{cat}\;[\,]\;(\mathsf{internalize}\;e)\;(\mathsf{internalize}\;e')\\
&\mathsf{internalize}\;(e + e')\\
&\quad=\; \mathsf{choice}\;[\,]\;(\mathsf{fuse}\;[\;0_b\;]\;(\mathsf{internalize}\;e))\\
&\qquad(\mathsf{fuse}\;[\;1_b\;]\;(\mathsf{internalize}\;e'))\\
&\mathsf{internalize}\;(e\,\star) \;=\; \mathsf{star}\;[\,]\;(\mathsf{internalize}\;e)
\end{aligned}
$$

Relating REs and BREs

- Function erase converts a BRE into a RE.
- Property: for all e, erase (internalize e) $\equiv e$.

$$
\begin{aligned}
&\text{erase} : \text{BitRegex} \to \text{Regex}\\
&\text{erase empty} = \emptyset\\
&\text{erase (eps } x) = \epsilon\\
&\text{erase (char } x\ c) = \$\ c\\
&\text{erase (choice } x\ e\ e') = \text{erase } e + (\text{erase } e')\\
&\text{erase (cat } x\ e\ e') = \text{erase } e \bullet (\text{erase } e')\\
&\text{erase (star } x\ e) = (\text{erase } e) \star
\end{aligned}
$$

Semantics of BREs

- Same as RE semantics, but includes the bit-codes.
- Property: $s \in [\![\, e \,]\!]$ iff $s \in \langle \, \text{internalize } e \, \rangle$.

$$
\begin{array}{l}
\textbf{data} \ \_ \in \langle \_ \rangle \ : \ \text{List Char} \ \to \ \text{BitRegex} \ \to \ \text{Set } \textbf{where} \\
\quad \text{eps} \ \ : \ [\,] \in \langle \, \text{eps } bs \, \rangle \\
\quad \text{char} \ : \ (c \ : \ \text{Char}) \ \to \ [\ c\ ] \in \langle \, \text{char } bs \ c \, \rangle \\
\quad \text{inl} \ \ \ : \ s \in \langle \, l \, \rangle \ \to \ s \in \langle \, \text{choice } bs \ l \ r \, \rangle \\
\quad \text{inr} \ \ : \ s \in \langle \, r \, \rangle \ \to \ s \in \langle \, \text{choice } bs \ l \ r \, \rangle \\
\quad \text{cat} \ \ : \ s \in \langle \, l \, \rangle \ \to \ s' \in \langle \, r \, \rangle \ \to \ (s +\!\!+ s') \in \langle \, \text{cat } bs \ l \ r \, \rangle \\
\quad \texttt{-- some code omitted}
\end{array}
$$

Derivatives in a nutshell

- What is the derivative of an (B)RE?
  - Derivatives are defined w.r.t. an alphabet symbol.
- The derivative of a (B)RE e w.r.t. a, $\partial(e, a)$, is another (B)RE that denotes all strings in e language with the leading a removed.

$$\partial_a(e) \quad = \quad \{s \mid as \in [\![e]\!]\}$$

- Property: $s \in \langle\ \partial[\ e\ ,\ x\ ]\ \rangle$ holds iff $(x :: s) \in \langle\ e\ \rangle$ holds.

Derivatives for BREs

- Follows the definition of Brzozowski's derivatives.

$$\partial[\_,\_] \;:\; \mathsf{BitRegex} \;\to\; \mathsf{Char} \;\to\; \mathsf{BitRegex}$$
$$\partial[\ \mathsf{eps}\ bs\ ,\ c\ ] \;=\; \mathsf{eps}\ bs$$
$$\partial[\ \mathsf{cat}\ bs\ e\ e'\ ,\ c\ ]\ \textbf{with}\ \nu[\ e\ ]$$
$$\partial[\ \mathsf{cat}\ bs\ e\ e'\ ,\ c\ ]\ \mid\ yes\ pr$$
$$\quad =\ \mathsf{choice}\ bs\ (\mathsf{cat}\ bs\ \partial[\ e\ ,\ c\ ]\ e')\ (\mathsf{fuse}\ (\mathsf{mkEps}\ pr)\ \partial[\ e'\ ,\ c\ ])$$
$$\partial[\ \mathsf{cat}\ bs\ e\ e'\ ,\ c\ ]\ \mid\ no\ pr\ \;=\; \mathsf{cat}\ bs\ \partial[\ e\ ,\ c\ ]\ e'$$
$$\partial[\ \mathsf{star}\ bs\ e\ ,\ c\ ]$$
$$\quad =\ \mathsf{cat}\ bs\ (\mathsf{fuse}\ [\ 0_{\mathsf{b}}\ ]\ \partial[\ e\ ,\ c\ ])\ (\mathsf{star}\ [\,]\ e)$$

```
-- some code omitted
```

Nullability test for BREs

- Checks if $\epsilon \in [\![e]\!]$, for some (B)RE e.
- Agda code: decision procedure for $[\,] \in \langle\ e\ \rangle$.

$$
\begin{array}{lcl}
\nu(\emptyset) & = & \emptyset \\
\nu(\epsilon) & = & \epsilon \\
\nu(a) & = & \emptyset \\
\nu(e\,e') & = & \left\{ \begin{array}{ll} \epsilon & \text{if } \nu(e) = \nu(e') = \epsilon \\ \emptyset & \text{otherwise} \end{array} \right. \\
\nu(e + e') & = & \left\{ \begin{array}{ll} \epsilon & \text{if } \nu(e) = \epsilon \text{ or } \nu(e') = \epsilon \\ \emptyset & \text{otherwise} \end{array} \right. \\
\nu(e^\star) & = & \epsilon
\end{array}
$$

Parsing with derivatives

- RE-based text search tools parse prefixes and substrings.
- Types IsPrefix and IsSubstr are proofs that a string is a prefix / substring of an input BRE.
- Parsing algorithms defined as proofs of decidability of IsPrefix and IsSubstr. Proof by induction on the input string, using properties of derivative operation.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{IsPrefix}\ (a\ :\ \mathsf{List\ Char})\ (e\ :\ \mathsf{BitRegex})\ :\ \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{Prefix}\ :\ a \equiv b +\!\!+\ c\ \rightarrow\ b \in \langle\ e\ \rangle\ \rightarrow\ \mathsf{IsPrefix}\ a\ e\\
&\textbf{data}\ \mathsf{IsSubstr}\ (a\ :\ \mathsf{List\ Char})\ (e\ :\ \mathsf{BitRegex})\ :\ \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{Substr}\ :\ a \equiv b +\!\!+\ c +\!\!+\ d\ \rightarrow\ c \in \langle\ e\ \rangle\ \rightarrow\ \mathsf{IsSubstr}\ a\ e
\end{aligned}
$$

Experimental Results

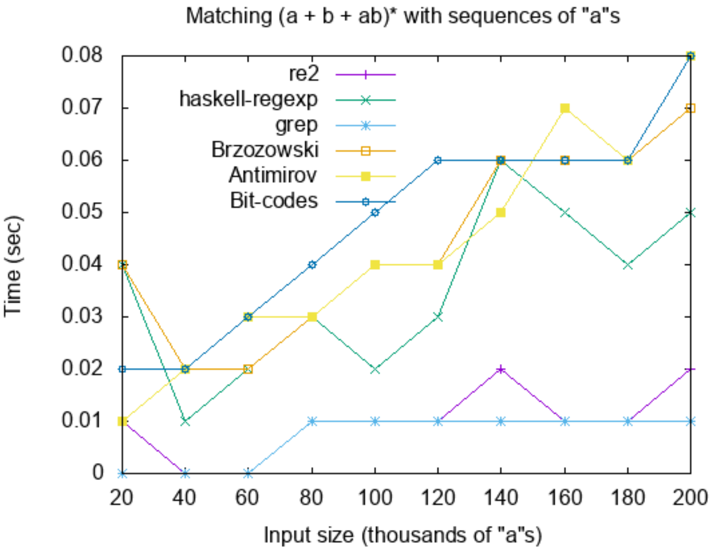

Figure: Results of experiment 1.

Future Work

- How to improve efficiency?
  - How intrinsic verification affects generated code efficiency?
  - Currently porting code to use extrinsic verification (proofs separated from program code).
  - Experiment with alternative formalization: BRE semantics defined by erase: $s \in \langle\, e \,\rangle \;=\; s \in [\![\, \text{erase}\ e \,]\!]$.
- How to measure memory consumption, without compiler support?
  - No profiling support in Agda compiler.
  - Agda compiles to Haskell, but there's no direct correspondence between Agda source code and Haskell generated code.

Conclusion

- We build a certified algorithm for BRE parsing in Agda.
- We certify several previous results about bit-coded parse trees and their relationship with RE semantics.
- Algorithm included in verigrep tool for RE text search.

Relating REs and BREs

- Function fuse attach a bit-string into a BRE.

$$
\begin{aligned}
&\mathsf{fuse} : \mathsf{List\ Bit} \to \mathsf{BitRegex} \to \mathsf{BitRegex}\\
&\mathsf{fuse}\ bs\ \mathsf{empty} = \mathsf{empty}\\
&\mathsf{fuse}\ bs\ (\mathsf{eps}\ x) = \mathsf{eps}\ (bs +\!\!+\ x)\\
&\mathsf{fuse}\ bs\ (\mathsf{char}\ x\ c) = \mathsf{char}\ (bs +\!\!+\ x)\ c\\
&\mathsf{fuse}\ bs\ (\mathsf{choice}\ x\ e\ e') = \mathsf{choice}\ (bs +\!\!+\ x)\ e\ e'\\
&\mathsf{fuse}\ bs\ (\mathsf{cat}\ x\ e\ e') = \mathsf{cat}\ (bs +\!\!+\ x)\ e\ e'\\
&\mathsf{fuse}\ bs\ (\mathsf{star}\ x\ e) = \mathsf{star}\ (bs +\!\!+\ x)\ e
\end{aligned}
$$

Building bit-codes for ϵ

$$
\begin{aligned}
&\mathsf{mkEps} : [\,] \in [\![\, t \,]\!] \to \mathsf{List\ Bit}\\
&\mathsf{mkEps}\,(\mathsf{eps}\ bs) = bs\\
&\mathsf{mkEps}\,(\mathsf{inl}\ br\ bs\ pr) = bs \mathbin{++} \mathsf{mkEps}\ pr\\
&\mathsf{mkEps}\,(\mathsf{inr}\ bl\ bs\ pr) = bs \mathbin{++} \mathsf{mkEps}\ pr\\
&\mathsf{mkEps}\,(\mathsf{cat}\ bs\ pr\ pr'\ eq) = bs \mathbin{++} \mathsf{mkEps}\ pr \mathbin{++} \mathsf{mkEps}\ pr'\\
&\mathsf{mkEps}\,(\mathsf{star}[]\ bs) = bs \mathbin{++} [\ 1_{\mathsf{b}}\ ]\\
&\mathsf{mkEps}\,(\mathsf{star}{-}::\ bs\ pr\ pr'\ x) = bs \mathbin{++} [\ 1_{\mathsf{b}}\ ]
\end{aligned}
$$

Relating parse trees and RE semantics

- unflat builds parse trees out of RE semantics evidence.
- Functions flat and unflat are inverses.

$$
\begin{aligned}
&\mathsf{unflat} : xs \in [\![\, e \,]\!] \rightarrow \mathsf{Tree}\ e\\
&\mathsf{unflat}\ \epsilon\ =\ \epsilon\\
&\mathsf{unflat}\ (\$\ c)\ =\ \$\ c\\
&\mathsf{unflat}\ (prf \bullet prf')\ =\ \mathsf{unflat}\ prf \bullet \mathsf{unflat}\ prf'\\
&\mathsf{unflat}\ (r\ +\mathrm{L}\ prf)\ =\ \mathsf{inl}\ r\ (\mathsf{unflat}\ prf)\\
&\mathsf{unflat}\ (l\ +\mathrm{R}\ prf)\ =\ \mathsf{inr}\ l\ (\mathsf{unflat}\ prf)
\end{aligned}
$$

```
-- some code omitted
```