Certified Derivative-based Parsing of Regular Expressions

Raul Lopes[1] Rodrigo Ribeiro [2] Carlos Camarão[3]

DECOM, Universidade Federal de Ouro Preto (UFOP), Ouro Preto

DECSI, Universidade Federal de Ouro Preto (UFOP), João Monlevade

DCC, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte

XX Brazilian Symposium on Programming Languages, 2016

Introduction

- Parsing is a fundamental activity in computing.
  - Used by compilers and string search tools
- In this work, we focus on parsing Regular Languages (RLs).
  - Languages accepted by (non)-deterministic finite automata and Regular Expressions (REs).

Introduction

- REs are an algebraic and compact way to specify RLs.
- RLs are used in lexical analyser generators and search tools.
- Our objective: Formalize an algorithm for RE based search using Dependently typed language Idris.

Introduction

- Our approach is to certify a search algorithm based on Brzozowski derivatives.
- Derivatives are an alternative method to compute a equivalent automaton from a given RE.
- Some recent works uses derivatives for parsing of both RE and even context free ones.

Contributions

- A complete formalization of derivative based parsing in Idris. The algorithm produces a parse tree that is an evidence that the input string is indeed on RE language or a proof that no such tree exists.
- A detailed formalization of the so-called "smart-constructors" used to quotient REs with respect to ACUI-axioms.

An Overview of Idris

- Idris is a dependently typed programming language which focus on programming rather than proving.
- Idris syntax is similar to Haskell syntax.
- Example: Natural numbers in Peano notation.

```
data Nat = Z | S Nat
```

An Overview of Idris

- Idris supports indexed type families using a Haskell GADT style syntax (similar to Agda's syntax).
- Example: Length-indexed lists (aka vectors) and safe head function.

```
data Vec : Nat -> Type -> Type where
    Nil : Vec Z a
    (::) : a -> Vec n a -> Vec (S n) a

head : Vec (S n) a -> a
head (x :: _) = x
```

Regular Expressions

- REs are defined with respect to a given alphabet, Σ, by the following context-free grammar:

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

Regular Expressions

- Idris representation

```
data RegExp : Type where
    Zero : RegExp
    Eps  : RegExp
    Chr  : Nat -> RegExp
    Cat  : RegExp -> RegExp -> RegExp
    Alt  : RegExp -> RegExp -> RegExp
    Star : RegExp -> RegExp
```

Regular Expression Semantics

- REs semantics as syntax directed judgement:

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

$$\frac{w \in [\![e]\!] \quad w' \in [\![e']\!]}{ww' \in [\![ee']\!]} \qquad \frac{w \in [\![e]\!]}{w \in [\![e + e']\!]}$$

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

Regular Expression Semantics

```
data InRegExp : Word -> RegExp -> Type where
    InEps : InRegExp [] Eps
    InChr : InRegExp [ a ] (Chr a)
    InCat : InRegExp xs l ->
            InRegExp ys r ->
            zs = xs ++ ys ->
            InRegExp zs (Cat l r)
    InAltL : InRegExp xs l ->
             InRegExp xs (Alt l r)
    -- some rules omitted...
```

Nullability Test

- Brzozowski definition uses an auxiliar function, $\nu(e)$, which returns ϵ if $\epsilon \in [\![e]\!]$ or $\emptyset$, otherwise.

$$
\begin{aligned}
\nu(\emptyset) &= \emptyset \\
\nu(\epsilon) &= \epsilon \\
\nu(a) &= \emptyset \\
\nu(e\,e') &= \begin{cases} \epsilon & \text{if } \nu(e) = \nu(e') = \epsilon \\ \emptyset & \text{otherwise} \end{cases} \\
\nu(e + e') &= \begin{cases} \epsilon & \text{if } \nu(e) = \epsilon \text{ or } \nu(e') = \epsilon \\ \emptyset & \text{otherwise} \end{cases} \\
\nu(e^\star) &= \epsilon
\end{aligned}
$$

Nullability Test

- Idris implementation of nullability tests shows its decidability.
- Theorem `hasEmptyDec` either returns a proof that `InRegExp Eps e` or an evidence that such construction is impossible.
  - Theorem follows by induction over RE structure.

Smart Constructors

- Smart constructors ensures the following RE equivalences

$$1)\ e + \emptyset \approx e \qquad 2)\ \emptyset + e \approx e$$
$$3)\ e\,\emptyset \approx \emptyset \qquad 4)\ e\,\epsilon \approx e$$
$$5)\ \emptyset\,e \approx \emptyset \qquad 6)\ \epsilon\,e \approx e$$
$$7)\ \emptyset^\star \approx \epsilon \qquad 8)\ \epsilon^\star \approx \epsilon$$

Smart Constructors

- Smart constructor for union.
  - Such smart constructor is sound and complete w.r.t. RE semantics.
  - Same structure for smart constructors for concatenation and star.

```
(.|.) : RegExp -> RegExp -> RegExp
Zero .|. e = e
e .|. Zero = e
e .|. e'   = Alt e e'
```

Derivative of Regular Expression

- Idris version uses nullability test function and smart constructors for concatenation, union and star.

$$
\begin{aligned}
\partial_a(\emptyset) &= \emptyset \\
\partial_a(\epsilon) &= \emptyset \\
\partial_a(b) &= \begin{cases} \epsilon & \text{if } b = a \\ \emptyset & \text{otherwise} \end{cases} \\
\partial_a(e\, e') &= \partial_a(e)\, e' + \nu(e)\, \partial_a(e') \\
\partial_a(e + e') &= \partial_a(e) + \partial_a(e') \\
\partial_a(e^\star) &= \partial_a(e)\, e^\star
\end{aligned}
$$

Derivative Properties

Theorem (Soundness of derivative operation)

For all RE `e`*, string* `xs` *and symbol* `x`*, if* `InRegExp xs (deriv x e)` *then* `InRegExp (x :: xs) e`.

Proof.

By induction on the structure of `e`, using the soundness lemmas for smart constructors and decidability of the emptiness test. □

Derivative Properties

Theorem (Completeness of derivative operation)

For all RE `e`*, string* `xs` *and symbol* `x`*, if* `InRegExp (x :: xs) e` *then* `InRegExp xs (deriv e x)`.

Proof.

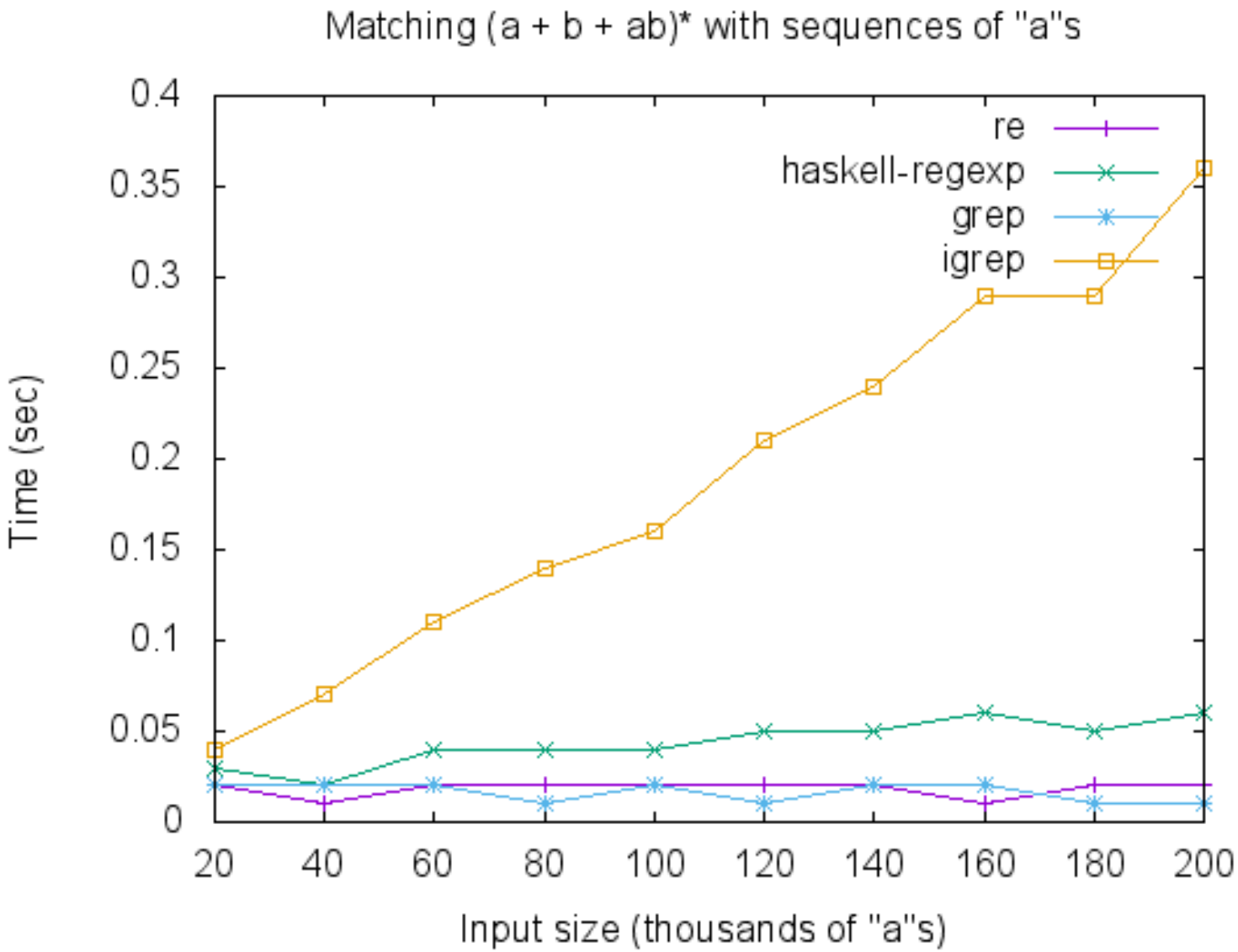
By induction on the structure of `e` using the completeness lemmas for smart constructors and decidability of the emptiness test. □

Parsing with Derivatives

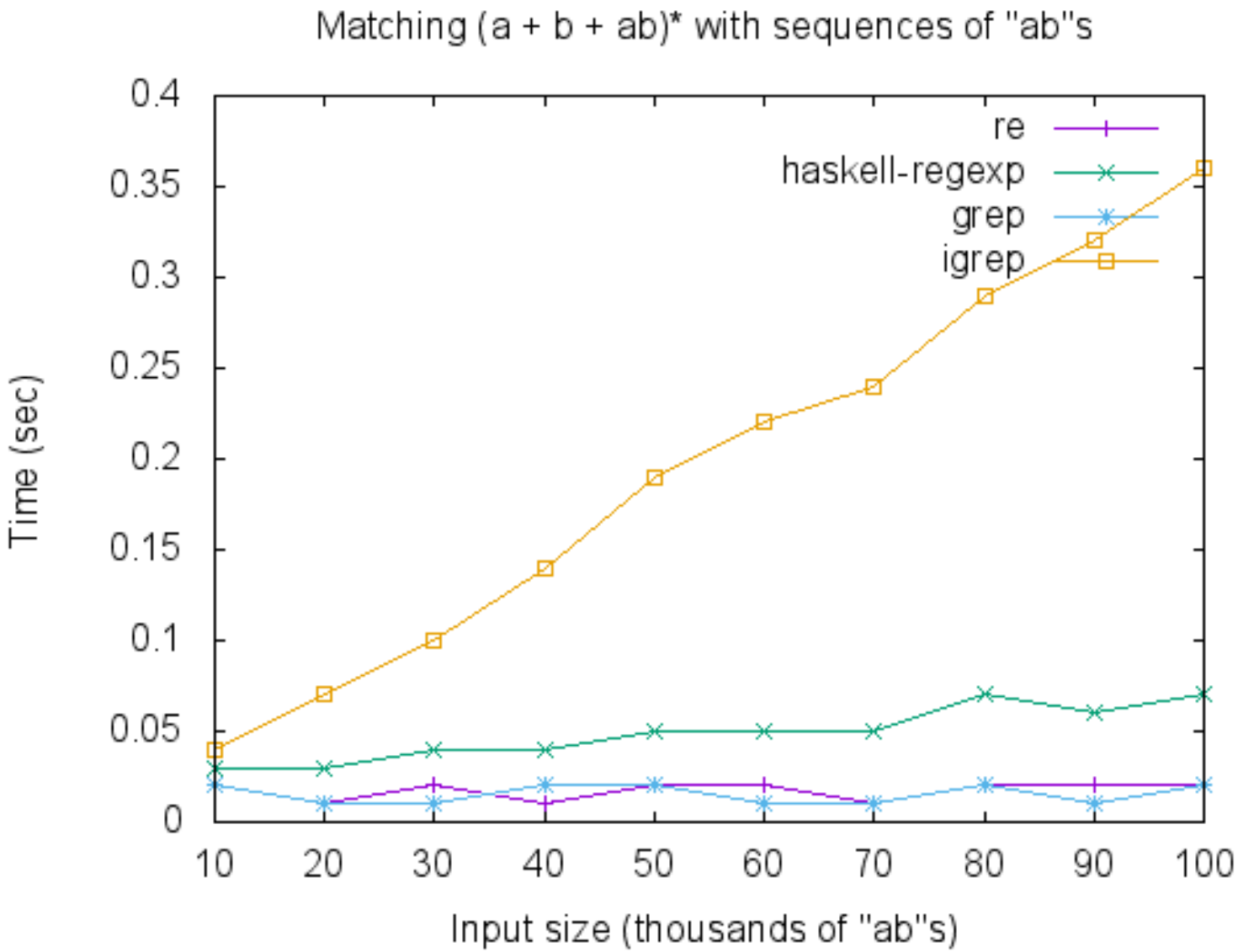
- Just apply the derivative operation recursively on each symbol of input.
- String accepted if resulting RE is nullable. Our implementation keeps track if a prefix or a substring of input is matched by input RE.

$$\begin{aligned} \partial^{\star}_{\epsilon}(e) &= e \\ \partial^{\star}_{a\,w}(e) &= \partial^{\star}_{w}(\partial_{a}(e)) \end{aligned}$$

Experiments



Experiments



Conclusion and Future Work

- Complete formalization of derivative based RE parsing algorithm in Idris.
- A tool, named iGrep, is produced from the formalization and experiments were conducted.
- Future work:
  - Improve efficient by using disambiguation strategies like greedy or POSIX.
  - Compile RE to an automaton and use it to parse input string and compare the efficency with derivative based algorithm.