Regular Expression Equivalence via Derivatives

Fabian Hellauer

Languages

- Words are lists.

Languages

- Words are lists.
- Languages are sets of words.

Languages

- Words are lists.
- Languages are sets of words.
- *Interesting* languages are the infinite ones.

Languages

- Words are lists.
- Languages are sets of words.
- *Interesting* languages are the infinite ones.
- Regular expressions (REs) are finite representations of languages

Regular Expressions

$$L(\emptyset) = \emptyset$$
$$L(\varepsilon) = \{[]\}$$
$$L(a) = \{[a]\}$$
$$L(r + s) = L(r) \cup L(s)$$
$$L(r \cdot s) = L(r)L(s)$$
$$L(r^*) = (L(r))^*$$

Regular Expressions

$$L(\emptyset) = \emptyset$$
$$L(\varepsilon) = \{[]\}$$
$$L(a) = \{[a]\}$$
$$L(r + s) = L(r) \cup L(s)$$
$$L(r \cdot s) = L(r)L(s)$$
$$L(r^*) = (L(r))^*$$

Equivalence problem:

Is $L(r_1) = L(r_2)$?

Goal

Equivalence checker for REs

Sample goal:

$$L\ ((\varepsilon + a \cdot b)^* \cdot (b + a)) = L\ ((a \cdot b + \varepsilon)^* \cdot (a + b))$$

→ *have the machine prove that for us*

The textbook method

A naive algorithm to decide RE equivalence:

The textbook method

A naive algorithm to decide RE equivalence:

1. construct NFAs from the REs

The textbook method

A naive algorithm to decide RE equivalence:

1. construct NFAs from the REs
2. convert the NFAs to DFAs

The textbook method

A naive algorithm to decide RE equivalence:

1. construct NFAs from the REs
2. convert the NFAs to DFAs
3. minimize the DFAs

The textbook method

A naive algorithm to decide RE equivalence:

1. construct NFAs from the REs
2. convert the NFAs to DFAs
3. minimize the DFAs

For each of these steps, we would have to

- express an algorithm

The textbook method

A naive algorithm to decide RE equivalence:

1. construct NFAs from the REs
2. convert the NFAs to DFAs
3. minimize the DFAs

For each of these steps, we would have to

- express an algorithm
- prove that this algorithm preserves the represented language

Bisimulation

Derivative-Language w.r.t. an atom :

$$D_x\ (A) := \{xs.\ x\#xs \in A\}$$

A relation “$\sim$“ with the following properties is called bisimulation:

for all A and B, if $A \sim B$, then

$$[] \in A \leftrightarrow [] \in B$$

and

$$\forall x.\ D_x(A) \sim D_x(B).$$

Bisimulation

Derivative-Language w.r.t. an atom :

$$D_x\ (A) := \{xs.\ x\#xs \in A\}$$

A relation "$\sim$" with the following properties is called bisimulation:

for all A and B, if $A \sim B$, then

$$[] \in A \leftrightarrow [] \in B$$

and

$$\forall x.\, D_x(A) \sim D_x(B).$$

Lemma:

If "$\sim$" is a bisimulation, then $A \sim B$ implies $A = B$. proof by list induction.

Derivatives of REs

- D is not useful, it works on the extensional representation
- use operation on REs instead: d

  goal: $L(d_a(r)) = D_a(L(r))$

  with $d :: 'a \Rightarrow 'a\ rexp \Rightarrow 'a\ rexp$ computable

Derivatives of REs

- D is not useful, it works on the extensional representation use operation on REs instead: d

  goal: $L(d_a(r)) = D_a(L(r))$

  with $d :: 'a \Rightarrow 'a\ rexp \Rightarrow 'a\ rexp$ computable

- These are the rules (Brzozowski 1964):

  $d_a\ (\emptyset) = \emptyset$
  $d_a\ (\varepsilon) = \emptyset$
  $d_a\ (\langle b \rangle) = (if\ a = b\ then\ \varepsilon\ else\ \emptyset)$
  $d_a\ (r + s) = d_a\ r + d_a\ s$

Derivatives of REs (cont.)

$$d_a(r \cdot s) =$$

$$(let\ drs = d_a\ (r) \cdot s$$

$$in\ if\ [\ ] \in L(r)\ then\ drs + d_a\ (s)\ else\ drs)$$

$$d_a\ (r^*) = d_a\ (r) \cdot r^*$$

Derivatives of REs (cont.)

$$d_a(r \cdot s) =$$
$$(let\ drs = d_a\ (r) \cdot s$$
$$in\ if\ [\,]\in L(r)\ then\ drs + d_a\ (s)\ else\ drs)$$

$$d_a\ (r^*) = d_a\ (r) \cdot r^*$$

$L(d_a(r)) = D_a(L(r))$ follows by structural induction.

Bisimulations (cont.)

We transpose the definition and lemma to the world of REs:

$$
\begin{aligned}
&\textit{is-bisimulation as ps} \longleftrightarrow \\
&\quad (\forall (r, s) \in ps. \\
&\qquad ([] \in L(r) \longleftrightarrow [] \in L(s)) \wedge \\
&\qquad (\forall a \in \textit{set as}.\ (d_a\ (r)\ , d_a\ (s)) \in ps) \wedge \\
&\qquad \textit{atoms } r \cup \textit{atoms } s \subseteq \textit{set as} \\
&\quad)
\end{aligned}
$$

$$\textit{is\_bisimulation as ps} \Longrightarrow (r, s) \in ps \Longrightarrow L\ (r) = L\ (s)$$

Algorithm

Assume we have a function that iterates a step s until a test t fails:

fun *while* where *while t s state =*
(if t s then while t s (s state) else state)

Algorithm

Assume we have a function that iterates a step s until a test t fails:

fun *while* where $while\ t\ s\ state =$
$$(if\ t\ s\ then\ while\ t\ s\ (s\ state)\ else\ state)$$

In our case, *state* has the type

$$(\alpha\ rexp \times \alpha\ rexp)\ list \times (\alpha\ rexp \times \alpha\ rexp)\ list$$

step

- A pair (r, s) from the work set is processed
- All pairs that are missing for the property

$$\forall a \in set\ as.\ (d_a(r), d_a(s)) \in R)$$

are added to the work set.

as will be the set of atoms in the original expressions (this does not change during execution).

step

```
fun step where step as (ws, ps) =
  (let
    new_p = hd ws;
    ps’ = new_p # ps;
    new_ws = [p ← succs as new_p . p ∉ set ps' ∪ set ws]
  in (new_ws @ tl ws, ps'))
```

…where succs as (r, s) = map (λa. (d_a (r) , d_a (s))) as

step

```
fun step where step as (ws, ps) =
  (let
    new_p = hd ws;
    ps’ = new_p # ps;
    new_ws = [p ← succs as new_p . p ∉ set ps' ∪ set ws]
  in (new_ws @ tl ws, ps'))
```

…where succs as (r, s) = map (λa. (d_a (r) , d_a (s))) as

We will iterate this step using the while function.

test

test (ws,_) $\leftrightarrow$ (case ws of

$\quad$ [] $\Rightarrow$ False |

$\quad$ (r, s)#_ $\Rightarrow$ *[] $\in$ L(r) $\leftrightarrow$ [] $\in$ L(s)*

)

The loop terminates if either

- the work set is empty (bisimulation constructed)
- a *definitely* nonequivalent pair of REs is to be processed (counterexample found)

Invariant

$$
\begin{aligned}
&\textit{pre-bisim as r s (ws, ps)} \leftrightarrow \\
&\qquad (r, s) \in ws \cup ps \land \\
&\qquad (\forall (p, q) \in ps. \\
&\qquad\qquad ([] \in L(p) \leftrightarrow [] \in L(q)) \land \\
&\qquad\qquad (\forall a \in as.\ (d_a(p), d_a(q)) \in ps \cup ws)) \land \\
&\qquad (\forall (p, q) \in ws \cup ps .\ atoms\ p \cup atoms\ q \subseteq set\ as)
\end{aligned}
$$

Towards termination

Towards termination

We now have soundness, but the execution often accumulates large REs of the form

$\emptyset \cdot (\ldots) + \emptyset \cdot (\ldots) + \emptyset \cdot (\ldots) + \ldots$ or

$\varepsilon \cdot \varepsilon \cdot \varepsilon \cdot \varepsilon \cdot (\ldots)$

Towards termination

We now have soundness, but the execution often accumulates large REEs of the form

$\emptyset \cdot (\dots) + \emptyset \cdot (\dots) + \emptyset \cdot (\dots) + \dots$ or

$\varepsilon \cdot \varepsilon \cdot \varepsilon \cdot \varepsilon \cdot (\dots)$

Using simplifications like

$\emptyset \cdot (\dots) \equiv \emptyset \qquad \emptyset + \mathrm{r} \equiv \mathrm{r} \qquad \varepsilon \cdot \mathrm{r} \equiv \mathrm{r}$

or their symmetric variants is no problem as long as

$L(d_a(r)) = D_a(L(r))$

Example goal

$$L\,((a \cdot b)^* \cdot a) = L\,(a \cdot (b \cdot a)^*)$$

Brzozowki's result about termination

ACI-equivalence

equality modulo *associativity*, *commutativity* and *idempotence* of *+*

Brzozowki's result about termination

ACI-equivalence

equality modulo *associativity*, *commutativity* and *idempotence* of *+*

In each step, we add the following to the work set:

$$\{(r,s) \leftarrow \text{succs as (hd ws)} \; . \; (r,s) \notin \text{set ps'} \cup \text{set ws}\}$$

Brzozowki’s result about termination

ACI-equivalence

equality modulo *associativity*, *commutativity* and *idempotence* of *+*

In each step, we add the following to the work set:

$$\{(r,s) \leftarrow \text{succs as (hd ws)} \,.\, (r,s) \notin \text{set ps}' \cup \text{set ws}\}$$

If the ∉-filter also considers ACI-equivalent REs to be equal,
then the computation terminates.

Decidability of ACI-equivalence (not verified)

equality modulo *associativity*, *commutativity* and *idempotence* of *+*

Decidability of ACI-equivalence (not verified)

equality modulo *associativity*, *commutativity* and *idempotence* of *+*

- ACI-equality of two REs can be reduced to equality by recursively sorting subterms of nested + terms, and eliminating duplicates

Decidability of ACI-equivalence (not verified)

equality modulo *associativity*, *commutativity* and *idempotence* of *+*

- ACI-equality of two REs can be reduced to equality by recursively sorting subterms of nested + terms, and eliminating duplicates
  - →Use some arbitrary order on the constructors: $\emptyset < \varepsilon < a <$ (_)* < (_ · _)

Decidability of ACI-equivalence (not verified)

equality modulo *associativity*, *commutativity* and *idempotence* of *+*

- ACI-equality of two REs can be reduced to equality by recursively sorting subterms of nested + terms, and eliminating duplicates
  - →Use some arbitrary order on the constructors: $\emptyset < \varepsilon < a <$ (_)* < (_ · _)
  - →The calls also make lists out of nested +'s.

Decidability of ACI-equivalence (not verified)

equality modulo *associativity*, *commutativity* and *idempotence* of *+*

- ACI-equality of two REs can be reduced to equality by recursively sorting subterms of nested + terms, and eliminating duplicates
  - →Use some arbitrary order on the constructors: $\emptyset < \varepsilon < a <$ (_)* < (_ · _)
  - →The calls also make lists out of nested +'s.
  - →Afterwards, check for equality.

Decidability of ACI-equivalence (not verified)

equality modulo *associativity*, *commutativity* and *idempotence* of *+*

- ACI-equality of two REs can be reduced to equality by recursively sorting subterms of nested + terms, and eliminating duplicates
  - →Use some arbitrary order on the constructors: $\emptyset < \varepsilon < a <$ (_)* < (_ · _)
  - →The calls also make lists out of nested +'s.
  - →Afterwards, check for equality.
- alternative: keep the REs in this normal form, as an invariant

Extensions

- "$\subseteq$" goals: Use the rule $A \subseteq B \longleftrightarrow A \cup B = B$

Extensions

- "⊆" goals: Use the rule $A \subseteq B \longleftrightarrow A \cup B = B$

- extended regular expressions:

  $\bar{r}$ (complement)

  $r\&s$ (intersection)

Extensions

- "⊆" goals: Use the rule A ⊆ B ⟷ A ∪ B = B

- extended regular expressions:

  $\overline{r}$ (complement)

  $r\&s$ (intersection)

  →We need derivative rules for these

  $$d_a\ (\overline{r}) = \overline{d_a\ (r)}$$

  $$d_a\ (r \,\&\, s) = d_a\ (r) \,\&\, d_a\ (s)$$

Questions