

# Regular Expression Equivalence via Derivatives

Fabian Hellauer

# Languages

- Words are lists.
- Languages are sets of words.
- Derivative-Language w.r.t. an atom :

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

- *Interesting* languages are the infinite ones.
  - represent them by regular expressions (REs)

# 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  $L(r_1) = L(r_2)$  ?*

# 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

# Goal

equivalence checker for regular expressions which is

- *automatic*: without user interaction
- *complete*: if  $L(r_1) = L(r_2)$ , the method *should* prove it
- elegant, i.e. easy to prove correct

# Bisimulation

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 computable
- use operation on REs instead:  $d$

$$\text{goal: } L(d_a(r)) = D_a(L(r))$$

with  $d :: 'a \Rightarrow 'a \text{ rexp} \Rightarrow 'a \text{ rexp}$  computable

- This is possible (Brzozowski 1964):

$$d_a(\emptyset) = \emptyset$$

$$d_a(\varepsilon) = \emptyset$$

$$d_a(<b>) = (\text{if } a = b \text{ then } \varepsilon \text{ else } \emptyset)$$

$$d_a(r + s) = d_a r + d_a s$$

# Derivatives of REs (cont.)

$$\begin{aligned} d_a(r \cdot s) = & \\ & (\text{let } drs = d_a(r) \cdot s \\ & \text{in if } [] \in L(r) \text{ then } drs + d_a(s) \text{ else } drs) \end{aligned}$$

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

---

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



# Bisimulations (cont.)

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

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

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

# 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*)

In our case, *state* has the type

$(\alpha \text{ rexp} \times \alpha \text{ rexp}) \text{ list} \times (\alpha \text{ rexp} \times \alpha \text{ rexp}) \text{ list}$

# step

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

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

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 ( $\lambda a. (d_a(r), d_a(s))$ ) as

We will iterate this step using the while function.

# test

test (ws,\_)  $\leftrightarrow$  (case ws of  
    []  $\Rightarrow$  False |  
    (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

*pre-bisim as r s (ws, ps)  $\leftrightarrow$*

*(r, s)  $\in$  ws  $\cup$  ps  $\wedge$*

*( $\forall (p, q) \in ps.$*

*( $[\ ] \in L(p) \leftrightarrow [\ ] \in L(q)$ )  $\wedge$*

*( $\forall a \in as. (d_a(p), d_a(q)) \in ps \cup ws$ ))  $\wedge$*

*( $\forall (p, q) \in ws \cup ps. atoms\ p \cup atoms\ q \subseteq set\ as$ )*

# Towards termination

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

$$\emptyset \cdot (...) + \emptyset \cdot (...) + \emptyset \cdot (...) + \dots \quad \text{or} \\ \varepsilon \cdot \varepsilon \cdot \varepsilon \cdot \varepsilon \cdot (...)$$

Using simplifications like

$$\emptyset \cdot (...) \equiv \emptyset \quad \emptyset + r \equiv r \quad \varepsilon \cdot r \equiv r$$

or their symmetric variants is no problem as long as

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

# Example goal

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



# Brzozowski'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  $\notin$ -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  $+$

- 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:  $\emptyset < \varepsilon < a < (\_)^* < (\_ \cdot \_)$
  - 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  $A \subseteq B \iff A \cup B = B$

- extended regular expressions:

$\bar{r}$  (complement)

$r \& s$  (intersection)

→ We need derivative rules for these

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

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

Questions