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.

  $\longrightarrow$ 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

Definition:

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.

Bisimulations (cont.)

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

$$
\begin{aligned}
&\text{is-bisimulation as ps} \longleftrightarrow \\
&\qquad (\forall (r, s) \in \text{ps}. \\
&\qquad\qquad (\text{final } r \longleftrightarrow \text{final } s) \wedge \\
&\qquad\qquad (\forall a \in \text{as}. (D\ a\ (r), D\ a\ (s)) \in \text{ps}) \wedge \\
&\qquad\qquad \text{atoms } r \cup \text{atoms } s \subseteq \text{as} \\
&\qquad)
\end{aligned}
$$

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

Derivatives of REs

- D is not computable
- use operation on REs instead: d

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

  with $d :: \text{‘}a \Rightarrow \text{‘}a\ rexp \Rightarrow \text{‘}a\ rexp$ computable
- This is possible (Brzozowski 1964):

  $d_a\ (\emptyset) = \emptyset$

  $d_a\ (\varepsilon) = \emptyset$

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

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

Derivatives of REs (cont.)

$$d_a(r \cdot s) =$$
$$(let\ drs = d_a\ (r) \cdot s$$
$$in\ if\ nullable\ r\ \ then\ drs + d_a\ (s)\ else\ drs)$$

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

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

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

We will iterate this step using the while function.

test

test (ws,_) ↔ (case ws of

  [] ⇒ False |

  (p, q)#_ ⇒ nullable p ↔ nullable q

)

The loop terminates if either

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

Example

<on the board>

result

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

Invariant

$$
\begin{aligned}
&\textit{pre-bisim as } r\ s\ (ws, ps) \leftrightarrow \\
&\qquad (r, s) \in ws \cup ps \wedge \\
&\qquad (\forall (r, s) \in ws \cup ps\ .\ atoms\ r \cup atoms\ s \subseteq as) \wedge \\
&\qquad (\forall (r, s) \in ps. \\
&\qquad\qquad (nullable\ r \leftrightarrow nullable\ s) \wedge \\
&\qquad\qquad (\forall a \in as.\ (d_a(r), d_a(s)) \in ps \cup ws))
\end{aligned}
$$

Choice operator: ACI

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

Brzozowki’s result about termination

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

[p ← succs as (hd ws) . p ∉ set ps' ∪ set ws]

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

The resulting relation will still be a bisimulation.

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