Regular Expression Equivalence via Derivatives

Fabian Hellauer

Goal

equivalence checker for regular expressions which is

- automatic: without user interaction
- complete
- elegant, i.e. easy to prove correct

… for an Isabelle proof method

Shortcuts

Avoid having to formalize and prove correct the automata

- construction (notation!)
- determinization
- minimization

Extension to relations (or other Kleene Algebras) without Kozen's theorem

Languages

- Words are lists.
- Languages are sets of words.
- *Interesting* languages are the infinite ones.
- Deriv x A := {xs. x # xs ∈ A}

Bisimulations

Definition

- for all A and B, $A \sim B$ ==> [] ∈ A <--> [] ∈ B
- for all A and B and x, $A \sim B$ ==> Deriv x A $\sim$ Deriv x B.

If "$\sim$" is a bisimulation, then $A \sim B$ implies A = B

proof by list induction.

Languages may be infinite

→represent them by regular expression (RE)

Regular Expressions

- L(0) = Ø
- L(1) = {[]}
- L(<a>) = {[a]}
- L(r + s) = L(r) ∪ L(s)
- L(r · s) = L(r)L(s)
- L(r*) = (L(r))*

deriv

- Deriv computes on infinitely many lists
- use operation on REs instead

  goal: L(deriv a r) = Deriv a (L(r))
- deriv :: ‘a => ‘a rexp => ‘a rexp computable
- Brzozowki formulatet the rules:

```
 "deriv _ Zero = Zero"
| "deriv _ One = Zero"
| "deriv a (Atom b) = (if a = b then One else Zero)"
| "deriv a (Plus r s) = Plus (deriv a r) (deriv a s)"
```

Times and Star

- deriv a (Times r s) =

  (let r's = Times (deriv a r) s

  in if nullable r then Plus r's (deriv a s) else r’s)

- deriv a (Star r) = Times (deriv a r) (Star r)

ACI

associativity, commutativity and idempotence of *+*

- the auxiliary function *norm* establishes a normal form

  ➔ ACI-equal terms are identified at that step already

- to verify this, one would have to state ACI-equivalence formally.

- the REs in the (emergent) bisimulation are kept in this normal form, as an invariant

nderiv

- keeps normed REEs normed
- "nderiv _ Zero = Zero"
- | "nderiv _ One = Zero"
- | "nderiv a (Atom b) = (if a = b then One else Zero)"
- | "nderiv a (Plus r s) = nPlus (nderiv a r) (nderiv a s)"
- | "nderiv a (Times r s) =

  (let r's = nTimes (nderiv a r) s

      in if nullable r then nPlus r's (nderiv a s) else r's)"
- | "nderiv a (Star r) = nTimes (nderiv a r) (Star r)"

Next steps of the algorithm

- to-do. Explain via condition, step and invariant or relate to general closure computation?
- <remember to stress why we want **simplicity**>

More Algorithm explanations

In each step

- A pair from the work set is processed
- All pairs are missing for the property

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

are added to the work set

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

More Algorithm explanations

More Algorithm explanations

Closure computation

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

Usage of functional Data Structures

- to-do?

Beyond equalities

- “⊆” can be solved easily

- ¬”≡” should really be stated differently, even though decidable

  e.g. $w \in A \setminus B$

- Relation algebras

Reflection due to Boyer and Moore

- one atom for every relation: <0>, <1>, …

- goal $(R^* \circ S^* \circ T)^* = (R \cup S \cup T)^*$

  $\leadsto$ goal $(<0>^* \cdot <1>^* \cdot <2>)^* = (<0> + <1> + <2>)^*$

Defining a usable proof method

- With *Eisbach*, one can use the usual method modifiers in a “proof method definition”:

```
method rexp = (unfold subset_eq_to_eq)?, (rule soundness, eval)+
```

- Example:

```
lemma "lang (Times (Star (Plus One AB)) A_or_B) ⊆ lang (Times (Star (Plus AB One)) A_or_B)“
    by rexp
```

References

- J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, Oct. 1964.
- A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. J. Automated Reasoning, 49:95–106, 2012. published online March 2011. http://www21.in.tum.de/~nipkow/pubs/jar12.html.
- Robert S. Boyer and J Strother Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In R. Boyer and J Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, 1981