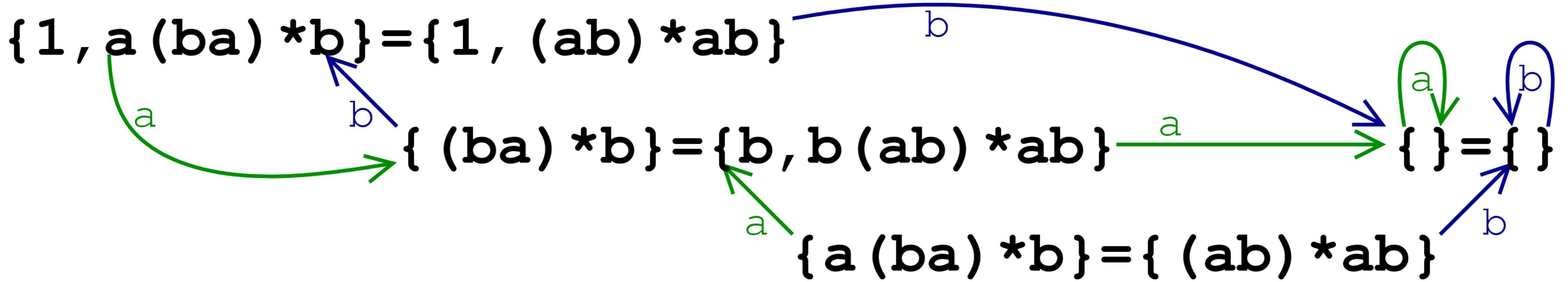


# Equality of Regular Expressions

## - A Coinductive Axiomatization



### Illustration of a Bisimulation



### What are Regular Expressions

- Describes a set of sequences of letters
- Are created from the following terms
  - 0 is a regular expression
  - 1 is a regular expression
  - $a, b$  are regular expressions
  - $e_1e_2$  is a regular expression if  $e_1$  and  $e_2$  are regular expressions
  - $e_1 + e_2$  is a regular expression if  $e_1$  and  $e_2$  are regular expressions
  - $e^*$  is a regular expression if  $e$  is a regular expression

### Language of an Expression

- The language of a regular expression is the set of sequences of letters that can be constructed from the expression.
- The language of a regular expression  $E$  is denoted  $\bar{\mathcal{L}}(E)$ .
- Examples
  - $\bar{\mathcal{L}}(0) = \{\}$
  - $\bar{\mathcal{L}}(1) = \{\varepsilon\}$
  - $\bar{\mathcal{L}}(a + b) = \{a, b\}$
  - $\bar{\mathcal{L}}(ab^*a) = \{aa, aba, abba, abbaa, \dots\}$
- The language of a set of regular expressions is the union of the languages of its elements.
- The language of a set of regular expressions  $A$  is denoted  $\mathcal{L}(A)$ .

### Equality of Expressions

- Two expressions are semantical equal if their languages are the same.
  - $\models E = F \Leftrightarrow \mathcal{L}(E) = \mathcal{L}(F)$ .
- Two sets of expressions are semantical equal if their languages are the same.
  - $\models A_1 = A_2 \Leftrightarrow \bigcup_{E \in A_1} \mathcal{L}(E) = \bigcup_{E \in A_2} \mathcal{L}(E)$ .
- Examples
  - $\models \{a, b\} = \{a + b\}$
  - $\models \{a(ba)^*b\} = \{(ab)^*ab\}$
  - $\models \{a\} \neq \{b\}$

### Empty Word Property

- A set of regular expressions has the empty word property if the empty sequence  $\varepsilon$  is in its language.
- $\text{ewp}(A) = \begin{cases} 1 & \text{if } \varepsilon \in \mathcal{L}(A) \\ 0 & \text{if } \varepsilon \notin \mathcal{L}(A) \end{cases}$
- The empty word property can easily be determined for any finite set of regular expressions.

### Derivatives

- $D_a(A)$  is a set of regular expressions.
- The language of  $D_a(A)$  is the sequences from the language of  $A$  starting with an  $a$ , but with the leading  $a$  removed.
  - $\mathcal{L}(D_a(A)) = \{w \mid aw \in \mathcal{L}(A)\}$ .
- Examples
  - $D_a(\{0\}) = D_a(\{1\}) = D_a(\{\}) = \{\}$
  - $D_a(\{a\}) = D_b(\{b\}) = \{1\}$
  - $D_a(\{b\}) = D_b(\{a\}) = \{\}$
  - $D_a(\{(ab)^*ab\}) = \{b(ab)^*ab, b\}$
- $D_a(A)$  and  $D_b(A)$  can easily be determined for any finite set of regular expressions  $A$ .

### Bisimulations

- A bisimulation is set of pairs of sets of regular expressions.
- A bisimulation  $R$  must fulfill the following properties
  - $A_1RA_2 \Rightarrow \text{ewp}(A_1) = \text{ewp}(A_2)$
  - $A_1RA_2 \Rightarrow D_a(A_1)RD_a(A_2)$
  - $A_1RA_2 \Rightarrow D_b(A_1)RD_b(A_2)$
- For all finite sets of regular expressions  $A_1$  and  $A_2$ , there is a finite bisimulation  $R$  such that  $A_1RA_2$  if and only if  $A_1$  and  $A_2$  are semantically equal.

### Problem

Given two sets of regular expressions, we want to determine if they represent the same language. We want to determine if  $\models A_1 = A_2$ .

### Result

Searching for a bisimulation in a structured way will either yield a proof of equality, or a word that is in one language but not the other.

### Applications

- Subtyping
- Translation of data
- DFA-generation without using NFA's

### Implementation (in SML)

```

(* The signature of a set *)
signature SetSig =
sig
  type element
  val empty : set
  val member : element -> set -> bool
  val subset : set -> set -> bool
  val equal : set * set -> bool
  val insert : element * set -> set
  val union : set -> set -> set
  val toList : set -> element list
  val fromList : element list -> set
  val toString : set -> string
end

(* A functor for setstructure generation *)
signature EQ =
sig
  type element
  val eq : element * element -> bool
  val elmToString : element -> string
end

structure SetPct(s : EQ) :> SetSig where type element = s.element =
struct
  type element = s.element
  type set = element list
  val empty = []
  fun member a = List.exists (fn y => s.eq(a,y)) xs
  fun subset xs ys = List.all (fn x => member x ys) xs
  fun equal (s1,s2) = subset s1 s2 andalso subset s2 s1
  fun insert (a,s) = if member a s then s else a::s
  val union = foldl insert empty
  val fromList = foldl insert empty
  val toList s = s
  fun toString xs = "[ " ^ foldl (fn (x,b) => s.elmToString x ^ ", " ^ b) "" xs ^ " ]"
end

(* Defining regular expressions *)
datatype reg =
  R0
  | Rletter of char
  | Rseq of reg * reg
  | Rsum of reg * reg
  | Rstar of reg

(* Pretty printing of regular expressions *)
fun regToString R0 = "0"
| regToString Rletter ch = Char.toString ch
| regToString Rseq (a,b) = "(" ^ regToString a ^ " " ^ regToString b ^ ")"
| regToString Rsum (a,b) = "(" ^ regToString a ^ " + " ^ regToString b ^ ")"
| regToString Rstar a = "(" ^ regToString a ^ "*" ^ ")"

(* Defining sets of regular expressions *)
structure RegEq : EQ =
struct
  type element = reg
  val eq = op =
  val elmToString = regToString
end

(* Set structure for regular expression *)
structure SetReg = SetPct(RegEq)
structure RegEnv : EQ =
struct
  type element = SetReg.set
  val eq = op =
  val elmToString = SetReg.toString
end

(* Set structure for environments, i.e. set structure for pairs of sets of regular expressions *)
structure SetEnv = SetPct(RegEnv)

(* The Empty Word Property *)
fun ewp R0 = true
| ewp Rletter _ = false
| ewp Rseq (a,b) = if ewp a = R1 andalso ewp b = R1 then R1 else R0
| ewp Rsum (a,b) = if ewp a = R1 orelse ewp b = R1 then R1 else R0
| ewp Rstar _ = R0

(* Combine a reg exp with a set of reg exps *)
fun SEQ1 R1 s = s
| SEQ1 R0 s = SetReg.empty
| SEQ1 Rseq (a,b) = foldl (fn (R0,A) => A | (R1,A) => SetReg.insert (E,A) | (E',A) => SetReg.insert (Rseq(E',E'),A)) (SetReg.empty (SetReg.toList s)) (SetReg.toList s)

(* Combine a set of reg exps with a reg exp *)
fun SEQ2 R1 s = s
| SEQ2 R0 s = SetReg.empty
| SEQ2 Rseq (a,b) = foldl (fn (R0,A) => A | (R1,A) => SetReg.insert (E,A) | (E',A) => SetReg.insert (Rseq(E',E'),A)) (SetReg.empty (SetReg.toList s)) (SetReg.toList s)

(* Differentiation *)
local
  fun diff - R0 = SetReg.empty
  | diff - Rletter b = if b = c then SetReg.insert (R1, SetReg.empty) else SetReg.empty
  | diff c (Rsum (a,b)) = SetReg.union (diff c a) (diff c b)
  | diff c (Rseq (a,b)) = SetReg.union (SEQ2 (diff c a) b) (SEQ1 (ewp a) (diff c b))
  | diff c (Rstar a) = SEQ2 (diff c a) (Rstar a)
in
  fun DIFF c s = foldl (fn (E,A) => SetReg.union (diff c E) A) (SetReg.empty (SetReg.toList s)) (SetReg.toList s)
end

exception NotEq of string
datatype proof = EqHyp | EqDiff of (char * proof) list

(* The algorithm *)
fun RegEq G A1 A2 w =
  if BWP A1 <> BWP A2 then raise NotEq w
  else if SetEnv.member (A1,A2) G then EqHyp
  else let val G' = SetEnv.insert ((A1,A2), G) in
    in
      let val Pa = RegEq G' (DIFF # "a" A1) (DIFF # "a" A2) (implode ((explode w) @ [# "a"]))
          val Pb = RegEq G' (DIFF # "b" A1) (DIFF # "b" A2) (implode ((explode w) @ [# "b"]))
      in EqDiff [(# "a", Pa), (# "b", Pb)]
      end
    end
  end

(* Defining constructors and equality-operator *)
infix 6 ++
fun A ++ B = Raum (A, B)
infix 7 ~
fun A ~ B = Rseq (A, B)
fun ++ A = Rstar A
infix 4 ==
fun A == B = let
  val x = RegEq SetEnv.empty
  in
    handle NotEq w => false
    end
  true
end
end

(* Defining the regular expressions *)
val a = (Rletter #"a") (* a *)
val ab = a ~ b (* ab *)
val ba = b ~ a (* ba *)
val E1 = a ++ a (* a+a *)
val E2 = a ++ b (* a+b *)
val E3 = b ++ a (* b+a *)
val E4 = a ~ b ~ a (* a ~ b ~ a *)
val E5 = a ~ a ~ b (* a ~ a ~ b *)
val E6 = a ~ a ~ b ~ a (* a ~ a ~ b ~ a *)
val bb = b ~ b
val bbb = b ~ b ~ b
val E7 = a ~ (b ++ bb ++ bbb) ~ a
val E8 = a ~ a ~ b ~ a (* a ~ a ~ b ~ a *)
val E9 = a ~ (bb ++ bbb) ~ a (* a ~ (bb ++ bbb) ~ a *)
val E10 = a ~ (R1 ++ b ~ b ~ b ~ b) ~ a (* a ~ (R1 ++ b ~ b ~ b ~ b) ~ a *)

(* Test execution *)
val test0 = (R0 == R1) = false (* w = "" *)
val test1 = (a == a) = true
val test2 = (b == b) = true
val test3 = (a == b) = false (* w = "a" *)
val test4 = (b == a) = false (* w = "a" *)
val test5 = (ab == ab) = true
val test6 = (ba == ab) = false (* w = "ab" *)
val test7 = (E1 == a) = true
val test8 = (E2 == E3) = true
val test9 = (E4 == E5) = true
val test10 = (R1 ++ E5 == E8) = true
val test11 = (E6 == E9) = false (* w = "" *)
val test12 = (E7 == E8) = false (* w = "abbbba" *)
val test13 = (E9 == E10) = true;
quit ();

```