

## 3.2: Equivalence and Simplification of Regular Expressions

In this section, we:

- say what it means for regular expressions to be equivalent;
- show a series of results about regular expression equivalence;
- describe two algorithms for the simplification of regular expressions, a weak, efficient one, and a stronger, but inefficient one, and show how these algorithms can be used in Forlan.

---

Copyright © 2003-9 Alley Stoughton

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

The L<sup>A</sup>T<sub>E</sub>X source of these slides, the associated book, and the distribution of the Forlan toolset are available on the WWW at <http://people.cis.ksu.edu/~stough/forlan/>.

1

### 3.2.1: Equivalence of Regular Expressions

We say that regular expressions  $\alpha$  and  $\beta$  are *equivalent* iff  $L(\alpha) = L(\beta)$ . In other words,  $\alpha$  and  $\beta$  are equivalent iff  $\alpha$  and  $\beta$  generate the same language. We define a relation  $\approx$  on **Reg** by:  $\alpha \approx \beta$  iff  $\alpha$  and  $\beta$  are equivalent.

For example,  $L((00)^* + \%) = L((00)^*)$ , and thus  $(00)^* + \% \approx (00)^*$ .

One approach to showing that  $\alpha \approx \beta$  is to show that  $L(\alpha) \subseteq L(\beta)$  and  $L(\beta) \subseteq L(\alpha)$ . The following proposition is useful for showing language inclusions, not just ones involving regular languages.

2

## (3.2.1) Language Inclusions

### Proposition 3.2.1

- (1) For all  $A_1, A_2, B_1, B_2 \in \mathbf{Lan}$ , if  $A_1 \subseteq B_1$  and  $A_2 \subseteq B_2$ , then  $A_1 \cup A_2 \subseteq B_1 \cup B_2$ .
- (2) For all  $A_1, A_2, B_1, B_2 \in \mathbf{Lan}$ , if  $A_1 \subseteq B_1$  and  $A_2 \subseteq B_2$ , then  $A_1 \cap A_2 \subseteq B_1 \cap B_2$ .
- (3) For all  $A_1, A_2, B_1, B_2 \in \mathbf{Lan}$ , if  $A_1 \subseteq B_1$  and  $B_2 \subseteq A_2$ , then  $A_1 - A_2 \subseteq B_1 - B_2$ .
- (4) For all  $A_1, A_2, B_1, B_2 \in \mathbf{Lan}$ , if  $A_1 \subseteq B_1$  and  $A_2 \subseteq B_2$ , then  $A_1 A_2 \subseteq B_1 B_2$ .
- (5) For all  $A, B \in \mathbf{Lan}$  and  $n \in \mathbb{N}$ , if  $A \subseteq B$ , then  $A^n \subseteq B^n$ .
- (6) For all  $A, B \in \mathbf{Lan}$ , if  $A \subseteq B$ , then  $A^* \subseteq B^*$ .

3

## (3.2.1) Language Inclusions (Cont.)

**Proof.** (1) and (2) are straightforward. We show (3) as an example, below. (4) is easy. (5) is proved by mathematical induction, using (4). (6) is proved using (5).

For (3), suppose that  $A_1, A_2, B_1, B_2 \in \mathbf{Lan}$ ,  $A_1 \subseteq B_1$  and  $B_2 \subseteq A_2$ . To show that  $A_1 - A_2 \subseteq B_1 - B_2$ , suppose  $w \in A_1 - A_2$ . We must show that  $w \in B_1 - B_2$ . It will suffice to show that  $w \in B_1$  and  $w \notin B_2$ .

Since  $w \in A_1 - A_2$ , we have that  $w \in A_1$  and  $w \notin A_2$ . Since  $A_1 \subseteq B_1$ , it follows that  $w \in B_1$ . Thus, it remains to show that  $w \notin B_2$ .

Suppose, toward a contradiction, that  $w \in B_2$ . Since  $B_2 \subseteq A_2$ , it follows that  $w \in A_2$ —contradiction. Thus we have that  $w \notin B_2$ .  $\square$

4

## (3.2.1) Basic Equivalences

### Proposition 3.2.2

- (1)  $\approx$  is reflexive on **Reg**, symmetric and transitive.
- (2) For all  $\alpha, \beta \in \mathbf{Reg}$ , if  $\alpha \approx \beta$ , then  $\alpha^* \approx \beta^*$ .
- (3) For all  $\alpha_1, \alpha_2, \beta_1, \beta_2 \in \mathbf{Reg}$ , if  $\alpha_1 \approx \beta_1$  and  $\alpha_2 \approx \beta_2$ , then  $\alpha_1\alpha_2 \approx \beta_1\beta_2$ .
- (4) For all  $\alpha_1, \alpha_2, \beta_1, \beta_2 \in \mathbf{Reg}$ , if  $\alpha_1 \approx \beta_1$  and  $\alpha_2 \approx \beta_2$ , then  $\alpha_1 + \alpha_2 \approx \beta_1 + \beta_2$ .

**Proof.** Follows from the properties of  $=$ . As an example, we show Part (4).

5

## (3.2.1) Basic Equivalences (Cont.)

**Proof (cont.).** Suppose  $\alpha_1, \alpha_2, \beta_1, \beta_2 \in \mathbf{Reg}$ , and assume that  $\alpha_1 \approx \beta_1$  and  $\alpha_2 \approx \beta_2$ . Then  $L(\alpha_1) = L(\beta_1)$  and  $L(\alpha_2) = L(\beta_2)$ , so that

$$\begin{aligned} L(\alpha_1 + \alpha_2) &= L(\alpha_1) \cup L(\alpha_2) = L(\beta_1) \cup L(\beta_2) \\ &= L(\beta_1 + \beta_2). \end{aligned}$$

Thus  $\alpha_1 + \alpha_2 \approx \beta_1 + \beta_2$ .  $\square$

6

## (3.2.1) Basic Equivalences (Cont.)

A consequence of Proposition 3.2.2 is the following proposition, which says that, if we replace a subtree of a regular expression  $\alpha$  by an equivalent regular expression, that the resulting regular expression is equivalent to  $\alpha$ .

### Proposition 3.2.3

Suppose  $\alpha, \beta, \beta' \in \mathbf{Reg}$ ,  $\beta \approx \beta'$ ,  $pat \in \mathbf{Path}$  is valid for  $\alpha$ , and  $\beta$  is the subtree of  $\alpha$  at position  $pat$ . Let  $\alpha'$  be the result of replacing the subtree at position  $pat$  in  $\alpha$  by  $\beta'$ . Then  $\alpha \approx \alpha'$ .

**Proof.** By induction on  $\alpha$ .  $\square$

7

## (3.2.1) Equivalences for Union

### Proposition 3.2.4

- (1) For all  $\alpha, \beta \in \mathbf{Reg}$ ,  $\alpha + \beta \approx \beta + \alpha$ .
- (2) For all  $\alpha, \beta, \gamma \in \mathbf{Reg}$ ,  $(\alpha + \beta) + \gamma \approx \alpha + (\beta + \gamma)$ .
- (3) For all  $\alpha \in \mathbf{Reg}$ ,  $\emptyset + \alpha \approx \alpha$ .
- (4) For all  $\alpha \in \mathbf{Reg}$ ,  $\alpha + \alpha \approx \alpha$ .
- (5) If  $L(\alpha) \subseteq L(\beta)$ , then  $\alpha + \beta \approx \beta$ .

**Proof.**

- (1) Follows from the commutativity of  $\cup$ .
- (2) Follows from the associativity of  $\cup$ .
- (3) Follows since  $\emptyset$  is the identity for  $\cup$ .
- (4) Follows since  $\cup$  is idempotent:  $A \cup A = A$ , for all sets  $A$ .
- (5) Follows since, if  $L_1 \subseteq L_2$ , then  $L_1 \cup L_2 = L_2$ .

$\square$

8

### (3.2.1) Equivalences for Concatenation

#### Proposition 3.2.5

- (1) For all  $\alpha, \beta, \gamma \in \mathbf{Reg}$ ,  $(\alpha\beta)\gamma \approx \alpha(\beta\gamma)$ .
- (2) For all  $\alpha \in \mathbf{Reg}$ ,  $\% \alpha \approx \alpha \approx \alpha \%$ .
- (3) For all  $\alpha \in \mathbf{Reg}$ ,  $\$ \alpha \approx \$ \approx \alpha \$$ .

#### Proof.

- (1) Follows from the associativity of language concatenation.
- (2) Follows since  $\{\%\}$  is the identity for language concatenation.
- (3) Follows since  $\emptyset$  is the zero for language concatenation.

□

9

### (3.2.1) Distributivity of Concatenation Over Union

#### Proposition 3.2.6

- (1) For all  $L_1, L_2, L_3 \in \mathbf{Lan}$ ,  $L_1(L_2 \cup L_3) = L_1L_2 \cup L_1L_3$ .
- (2) For all  $L_1, L_2, L_3 \in \mathbf{Lan}$ ,  $(L_1 \cup L_2)L_3 = L_1L_3 \cup L_2L_3$ .

**Proof.** We show the proof of Part (1); the proof of the other part is similar. Suppose  $L_1, L_2, L_3 \in \mathbf{Lan}$ . It will suffice to show that

$$L_1(L_2 \cup L_3) \subseteq L_1L_2 \cup L_1L_3 \subseteq L_1(L_2 \cup L_3).$$

### (3.2.1) Distributivity (Cont.)

**Proof (cont.).** To see that  $L_1(L_2 \cup L_3) \subseteq L_1L_2 \cup L_1L_3$ , suppose  $w \in L_1(L_2 \cup L_3)$ . We must show that  $w \in L_1L_2 \cup L_1L_3$ . By our assumption,  $w = xy$  for some  $x \in L_1$  and  $y \in L_2 \cup L_3$ . There are two cases to consider.

- Suppose  $y \in L_2$ . Then  $w = xy \in L_1L_2 \subseteq L_1L_2 \cup L_1L_3$ .
- Suppose  $y \in L_3$ . Then  $w = xy \in L_1L_3 \subseteq L_1L_2 \cup L_1L_3$ .

11

### (3.2.1) Distributivity (Cont.)

**Proof (cont.).** To see that  $L_1L_2 \cup L_1L_3 \subseteq L_1(L_2 \cup L_3)$ , suppose  $w \in L_1L_2 \cup L_1L_3$ . We must show that  $w \in L_1(L_2 \cup L_3)$ . There are two cases to consider.

- Suppose  $w \in L_1L_2$ . Then  $w = xy$  for some  $x \in L_1$  and  $y \in L_2$ . Thus  $y \in L_2 \cup L_3$ , so that  $w = xy \in L_1(L_2 \cup L_3)$ .
- Suppose  $w \in L_1L_3$ . Then  $w = xy$  for some  $x \in L_1$  and  $y \in L_3$ . Thus  $y \in L_2 \cup L_3$ , so that  $w = xy \in L_1(L_2 \cup L_3)$ .

□

12

## (3.2.1) Distributivity (Cont.)

### Proposition 3.2.7

(1) For all  $\alpha, \beta, \gamma \in \mathbf{Reg}$ ,  $\alpha(\beta + \gamma) \approx \alpha\beta + \alpha\gamma$ .

(2) For all  $\alpha, \beta, \gamma \in \mathbf{Reg}$ ,  $(\alpha + \beta)\gamma \approx \alpha\gamma + \beta\gamma$ .

**Proof.** Follows from Proposition 3.2.6. Consider, e.g., the proof of Part (1). By Proposition 3.2.6(1), we have that

$$\begin{aligned}L(\alpha(\beta + \gamma)) &= L(\alpha)L(\beta + \gamma) \\ &= L(\alpha)(L(\beta) \cup L(\gamma)) \\ &= L(\alpha)L(\beta) \cup L(\alpha)L(\gamma) \\ &= L(\alpha\beta) \cup L(\alpha\gamma) \\ &= L(\alpha\beta + \alpha\gamma)\end{aligned}$$

Thus  $\alpha(\beta + \gamma) \approx \alpha\beta + \alpha\gamma$ .  $\square$

13

## (3.2.1) Inclusions for Kleene Closure

### Proposition 3.2.8

- For all  $L \in \mathbf{Lan}$ ,  $LL^* \subseteq L^*$ .
- For all  $L \in \mathbf{Lan}$ ,  $L^*L \subseteq L^*$ .

**Proof.** E.g., to see that  $LL^* \subseteq L^*$ , suppose  $w \in LL^*$ . Then  $w = xy$  for some  $x \in L$  and  $y \in L^*$ . Hence  $y \in L^n$  for some  $n \in \mathbb{N}$ . Thus  $w = xy \in LL^n = L^{n+1} \subseteq L^*$ .  $\square$

14

## (3.2.1) Equivalences for Kleene Closure

### Proposition 3.2.9

- (1)  $\emptyset^* = \{\epsilon\}$ .
- (2)  $\{\epsilon\}^* = \{\epsilon\}$ .
- (3) For all  $L \in \mathbf{Lan}$ ,  $L^*L = LL^*$ .
- (4) For all  $L \in \mathbf{Lan}$ ,  $L^*L^* = L^*$ .
- (5) For all  $L \in \mathbf{Lan}$ ,  $(L^*)^* = L^*$ .
- (6) For all  $L_1L_2 \in \mathbf{Lan}$ ,  $(L_1L_2)^*L_1 = L_1(L_2L_1)^*$ .

**Proof.** The six parts can be proven in order using Proposition 3.2.1. All parts but (2), (5) and (6) can be proved without using induction.

As an example, we show the proof of Part (5). To show that  $(L^*)^* = L^*$ , it will suffice to show that  $(L^*)^* \subseteq L^* \subseteq (L^*)^*$ .

15

## (3.2.1) Equivalences for Kleene Closure (Cont.)

**Proof (cont.).** To see that  $(L^*)^* \subseteq L^*$ , we use mathematical induction to show that, for all  $n \in \mathbb{N}$ ,  $(L^*)^n \subseteq L^*$ .

**(Basis Step)** We have that  $(L^*)^0 = \{\epsilon\} = L^0 \subseteq L^*$ .

**(Inductive Step)** Suppose  $n \in \mathbb{N}$ , and assume the inductive hypothesis:  $(L^*)^n \subseteq L^*$ . We must show that  $(L^*)^{n+1} \subseteq L^*$ . By the inductive hypothesis, Proposition 3.2.1(4) and Part (4), we have that  $(L^*)^{n+1} = L^*(L^*)^n \subseteq L^*L^* = L^*$ .

Now, we use the result of the induction to prove that  $(L^*)^* \subseteq L^*$ . Suppose  $w \in (L^*)^*$ . We must show that  $w \in L^*$ . Since  $w \in (L^*)^*$ , we have that  $w \in (L^*)^n$  for some  $n \in \mathbb{N}$ . Thus, by the result of the induction,  $w \in (L^*)^n \subseteq L^*$ .

For the other inclusion, we have that  $L^* = (L^*)^1 \subseteq (L^*)^*$ .  $\square$

16

## (3.2.1) Equivalences for Kleene Closure (Cont.)

### Proposition 3.2.11

- (1)  $\$^* \approx \%$ .
- (2)  $\%^* \approx \%$ .
- (3) For all  $\alpha \in \mathbf{Reg}$ ,  $\alpha^* \alpha \approx \alpha \alpha^*$ .
- (4) For all  $\alpha \in \mathbf{Reg}$ ,  $\alpha^* \alpha^* \approx \alpha^*$ .
- (5) For all  $\alpha \in \mathbf{Reg}$ ,  $(\alpha^*)^* \approx \alpha^*$ .
- (6) For all  $\alpha, \beta \in \mathbf{Reg}$ ,  $(\alpha\beta)^* \alpha \approx \alpha(\beta\alpha)^*$ .

**Proof.** Follows from Proposition 3.2.9. Consider, e.g., the proof of Part (5). By Proposition 3.2.9(5), we have that

$$L((\alpha^*)^*) = L(\alpha^*)^* = (L(\alpha)^*)^* = L(\alpha)^* = L(\alpha^*).$$

Thus  $(\alpha^*)^* \approx \alpha^*$ .  $\square$

17

## 3.2.2: Regular Expression Simplification

In this subsection, we describe two algorithms for the simplification of regular expressions:

- one that is weak, but efficient; and
- one that is stronger, but inefficient.

## (3.2.2) Weakly Simplified Regular Expressions

We say that a regular expression  $\alpha$  is *weakly simplified* iff none of  $\alpha$ 's subtrees have any of the following forms:

- $\$ + \beta$  or  $\beta + \$$ ;
- $(\beta_1 + \beta_2) + \beta_3$ ;
- $\beta_1 + \beta_2$ , where  $\beta_1 \geq \beta_2$ , or  $\beta_1 + (\beta_2 + \beta_3)$ , where  $\beta_1 \geq \beta_2$ ;
- $\% \beta$  or  $\beta \%$ ;
- $\$ \beta$  or  $\beta \$$ ;
- $(\beta_1 \beta_2) \beta_3$ ;
- $\beta^* \beta$  or  $\beta^* (\beta \gamma)$ ;
- $(\beta_1 \beta_2)^* \beta_1$  or  $(\beta_1 \beta_2)^* \beta_1 \gamma$ ;
- $\%^*$  or  $\$^*$  or  $(\beta^*)^*$ .

Thus, if a regular expression  $\alpha$  is weakly simplified, then each of its subtrees will also be weakly simplified.

19

## (3.2.2) Properties of Weakly Simplified Expressions

### Proposition 3.2.18

For all  $\alpha \in \mathbf{Reg}$ :

- (1) if  $\alpha$  is weakly simplified and  $L(\alpha) = \{\%\}$ , then  $\alpha = \%$ ;
- (2) if  $\alpha$  is weakly simplified and  $L(\alpha) = \emptyset$ , then  $\alpha = \$$ ;
- (3) for all  $a \in \mathbf{Sym}$ , if  $\alpha$  is weakly simplified and  $L(\alpha) = \{a\}$ , then  $\alpha = a$ .

**Proof.** By simultaneous induction on regular expressions. We show part of the proof of the concatenation case. Suppose  $\alpha, \beta \in \mathbf{Reg}$  and assume the inductive hypothesis, that Parts (1)–(3) hold for  $\alpha$  and  $\beta$ . One must show that Parts (1)–(3) hold for  $\alpha\beta$ . We will show that Part (3) holds for  $\alpha\beta$ . Suppose  $a \in \mathbf{Sym}$ , and assume that  $\alpha\beta$  is weakly simplified and  $L(\alpha\beta) = \{a\}$ . We must show that  $\alpha\beta = a$ .

20

## (3.2.2) Properties of Weakly Simplified Expressions (Cont.)

**Proof (cont.).** Since  $L(\alpha)L(\beta) = L(\alpha\beta) = \{a\}$ , there are two cases to consider.

- Suppose  $L(\alpha) = \{a\}$  and  $L(\beta) = \{\%\}$ . Since  $\beta$  is weakly simplified and  $L(\beta) = \{\%\}$ , Part (1) of the inductive hypothesis on  $\beta$  tells us that  $\beta = \%$ . But this means that  $\alpha\beta = \alpha\%$  is not weakly simplified after all—contradiction. Thus we can conclude that  $\alpha\beta = a$ .
- Suppose  $L(\alpha) = \{\%\}$  and  $L(\beta) = \{a\}$ . The proof of this case is similar to that of the other one.

□

21

## (3.2.2) Properties of Weakly Simplified Expressions (Cont.)

### **Proposition 3.2.19**

*For all  $\alpha \in \mathbf{Reg}$ , if  $\alpha$  is weakly simplified and  $\alpha$  has one or more occurrences of  $\$$ , then  $\alpha = \$$ .*

**Proof.** An easy induction on regular expressions. □

### **Proposition 3.2.20**

*For all  $\alpha \in \mathbf{Reg}$ , if  $\alpha$  is weakly simplified and  $\alpha$  has one or more closures, then  $L(\alpha)$  is infinite.*

**Proof.** An easy induction on regular expressions. □

22

## (3.2.2) Definition of Weak Simplification

We define a function/algorithm  $\text{weakSimplify} \in \mathbf{Reg} \rightarrow \mathbf{Reg}$  by structural recursion:

- $\text{weakSimplify } \% = \%$ .
- $\text{weakSimplify } \$ = \$$ .
- For all  $a \in \mathbf{Sym}$ ,  $\text{weakSimplify } a = a$ .
- For all  $\alpha \in \mathbf{Reg}$ ,  $\text{weakSimplify}(\alpha^*)$  is formed as follows. Let  $\alpha' = \text{weakSimplify } \alpha$ .
  - If  $\alpha' = \%$  or  $\alpha' = \$$ , then  $\text{weakSimplify}(\alpha^*) = \%$ .
  - Otherwise, if  $\alpha'$  is a closure, then  $\text{weakSimplify}(\alpha^*) = \alpha'$ .
  - Otherwise,  $\text{weakSimplify}(\alpha^*) = \alpha'^*$ .

For example, if  $\alpha' = 0^*$ , then  $\text{weakSimplify}(\alpha^*) = 0^*$ .

23

## (3.2.2) Definition of Weak Simplification (Cont.)

- For all  $\alpha, \beta \in \mathbf{Reg}$ ,  $\text{weakSimplify}(\alpha\beta)$  is formed as follows. Let  $\alpha' = \text{weakSimplify } \alpha$  and  $\beta' = \text{weakSimplify } \beta$ .
  - If  $\alpha' = \$$  or  $\beta' = \$$ , then  $\text{weakSimplify}(\alpha\beta) = \$$ .
  - Otherwise, if  $\alpha' = \%$ , then  $\text{weakSimplify}(\alpha\beta) = \beta'$ .
  - Otherwise, if  $\beta' = \%$ , then  $\text{weakSimplify}(\alpha\beta) = \alpha'$ .
  - Otherwise, let  $n \in \mathbb{N}$  and  $\alpha'_1, \dots, \alpha'_n$  be such that  $\alpha' = \alpha'_1 \cdots \alpha'_n$  and  $\alpha'_n$  is not a concatenation. Then  $\text{weakSimplify}(\alpha\beta)$  is the result of repeatedly applying the rules

$$\begin{aligned} \alpha^* \alpha &\rightarrow \alpha \alpha^*, & \alpha^* \alpha \beta &\rightarrow \alpha \alpha^* \beta, \\ (\alpha \beta)^* \alpha &\rightarrow \alpha (\beta \alpha)^*, & (\alpha \beta)^* \alpha \gamma &\rightarrow \alpha (\beta \alpha)^* \gamma \end{aligned}$$

to  $\alpha'_1 \cdots \alpha'_n \beta'$ , as described in the book.

24

### (3.2.2) Definition of Weak Simplification (Cont.)

- (Definition of **weakSimplify**( $\alpha\beta$ ), cont.)
  - (Continuation of main case, processing of  $\alpha'_1 \cdots \alpha'_n \beta'$ .) In more detail, the rules

$$\begin{aligned} \alpha^* \alpha &\rightarrow \alpha \alpha^*, & \alpha^* \alpha \beta &\rightarrow \alpha \alpha^* \beta, \\ (\alpha\beta)^* \alpha &\rightarrow \alpha(\beta\alpha)^*, & (\alpha\beta)^* \alpha \gamma &\rightarrow \alpha(\beta\alpha)^* \gamma \end{aligned}$$

are applied down the rightmost path of  $\alpha'_1 \cdots \alpha'_n \beta'$ . And, in the third and fourth rules, if  $\beta = \beta_1 \cdots \beta_m$ , where  $\beta_m$  isn't a concatenation, then the result of recursively processing  $\beta_1 \cdots \beta_m \alpha$  is used instead of  $\beta\alpha$ .

For example, if  $\alpha' = 0^*(010^*)^*$  and  $\beta' = 0100^*$ , then **weakSimplify**( $\alpha\beta$ ) =  $00^*100^*(100^*)^*$ .

25

### (3.2.2) Definition of Weak Simplification (Cont.)

- For all  $\alpha, \beta \in \mathbf{Reg}$ , **weakSimplify**( $\alpha + \beta$ ) is formed as follows. Let  $\alpha' = \mathbf{weakSimplify} \alpha$  and  $\beta' = \mathbf{weakSimplify} \beta$ .
  - If  $\alpha' = \$$ , then **weakSimplify**( $\alpha + \beta$ ) =  $\beta'$ .
  - Otherwise, if  $\beta' = \$$ , then **weakSimplify**( $\alpha + \beta$ ) =  $\alpha'$ .
  - Otherwise, let  $n \in \mathbb{N}$  and  $\alpha'_1, \dots, \alpha'_n$  be such that  $\alpha' = \alpha'_1 + \cdots + \alpha'_n$  and  $\alpha'_n$  is not a union, and let  $m \in \mathbb{N}$  and  $\beta'_1, \dots, \beta'_m$  be such that  $\beta' = \beta'_1 + \cdots + \beta'_m$  and  $\beta'_m$  is not a union. Let  $\gamma_1, \dots, \gamma_l$ , for some  $l \in \mathbb{N}$ , be the result of putting the  $\alpha'_i$  and  $\beta'_i$  in order, without repetition. Then **weakSimplify**( $\alpha + \beta$ ) =  $\gamma_1 + \gamma_2 + \cdots + \gamma_l$ .

For example, if  $\alpha' = 0 + 1 + 2 = 0 + (1 + 2)$  and  $\beta' = 1 + 2 + 3 = 1 + (2 + 3)$ , then

$$\mathbf{weakSimplify}(\alpha + \beta) = 0 + (1 + (2 + 3)) = 0 + 1 + 2 + 3.$$

26

## (3.2.2) Properties of Weak Simplification

### Proposition 3.2.30

For all  $\alpha \in \mathbf{Reg}$ :

- (1)  $\mathbf{weakSimplify} \alpha \approx \alpha$ ;
- (2)  $\mathbf{weakSimplify} \alpha$  is weakly simplified;
- (3)  $\mathbf{alphabet}(\mathbf{weakSimplify}(\alpha)) \subseteq \mathbf{alphabet} \alpha$ ;
- (4)  $\mathbf{size}(\mathbf{weakSimplify} \alpha) \leq \mathbf{size} \alpha$ ;
- (5) the number of concatenations of  $\mathbf{weakSimplify} \alpha$  is less-than-or-equal-to the number of concatenations of  $\alpha$ ;
- (6) the number of symbols of  $\mathbf{weakSimplify} \alpha$  is less-than-or-equal-to the number of symbols of  $\alpha$ .

**Proof.** By induction on regular expressions.  $\square$

27

## (3.2.2) Calculating $L(\alpha)$ When Finite

Using our weak simplification algorithm, we can define an algorithm for calculating the language generated by a regular expression, when this language is finite, and for announcing that this language is infinite, otherwise.

First, we weakly simplify our regular expression,  $\alpha$ , and call the resulting regular expression  $\beta$ . If  $\beta$  contains no closures, then we compute its meaning in the usual way. But, if  $\beta$  contains one or more closures, then its language will be infinite, and thus we can output a message saying that  $L(\alpha)$  is infinite.

28

## (3.2.2) Toward a Stronger Simplification Algorithm

In preparation for the definition of our stronger simplification algorithm, we need two auxiliary functions (algorithms):

$$\begin{aligned}\mathbf{hasEmp} &\in \mathbf{Reg} \rightarrow \mathbf{Bool}, \\ \mathbf{weakSubset} &\in \mathbf{Reg} \times \mathbf{Reg} \rightarrow \mathbf{Bool}.\end{aligned}$$

The definitions of these functions are given in the book.

The function **hasEmp** meets the following specification: for all  $\alpha \in \mathbf{Reg}$ ,  $\mathbf{hasEmp} \alpha = \mathbf{true}$  iff  $\% \in L(\alpha)$ .

The function **weakSubset** is a *conservative approximation to subset testing*, i.e., for all  $\alpha, \beta \in \mathbf{Reg}$ , if  $\mathbf{weakSubset}(\alpha, \beta) = \mathbf{true}$ , then  $L(\alpha) \subseteq L(\beta)$ .

In Section 3.12, we will learn of a less efficient algorithm that will provide a complete test for  $L(\alpha) \subseteq L(\beta)$ .

29

## (3.2.2) A Stronger Simplification Algorithm

Now, we sketch the definition of our stronger simplification function (algorithm)

$$\mathbf{simplify} \in (\mathbf{Reg} \times \mathbf{Reg} \rightarrow \mathbf{Bool}) \rightarrow \mathbf{Reg} \rightarrow \mathbf{Reg},$$

which takes in a function *sub*, and returns a function that uses *sub* in order to simplify regular expressions. The function *sub* should be a conservative approximation to subset testing, like **weakSubset**.

## (3.2.2) Simplification Rules

Our simplification algorithm is based on the following 29 *simplification rules*, which may be applied to arbitrary subtrees of weakly simplified regular expressions, and never introduce new symbols.

Each simplification rule either:

- strictly decreases the size of a regular expression (this is true for all rules but 7–8 and 20–22, without the assumption about weak simplification); or
- preserves the size of a regular expression, but strictly decreases its number of concatenations (Rules 7–8 do this, without the assumption about weak simplification); or
- preserves the size and number of concatenations of a regular expression, but strictly decreases its number of symbols (this sometimes happens with Rules 20–22, and relies on the regular expression being weakly simplified; mostly, these rules strictly decrease the size of a regular expression).

31

## (3.2.2) Simplification Rules (Cont.)

- (1)  $\alpha^*(\beta\alpha^*)^* \rightarrow (\alpha + \beta)^*$ .
- (2)  $(\alpha^*\beta)^*\alpha^* \rightarrow (\alpha + \beta)^*$ .
- (3) If **hasEmp**( $\alpha$ ) and  $sub(\alpha, \beta^*)$ , then  $\alpha\beta^* \rightarrow \beta^*$ .
- (4) If **hasEmp**( $\beta$ ) and  $sub(\beta, \alpha^*)$ , then  $\alpha^*\beta \rightarrow \alpha^*$ .
- (5) If  $sub(\alpha, \beta^*)$ , then  $(\alpha + \beta)^* \rightarrow \beta^*$ .
- (6)  $(\alpha + \beta^*)^* \rightarrow (\alpha + \beta)^*$ .
- (7) (concatenations) If **hasEmp**( $\alpha$ ) and **hasEmp**( $\beta$ ), then  $(\alpha\beta)^* \rightarrow (\alpha + \beta)^*$ .
- (8) (concatenations) If **hasEmp**( $\alpha$ ) and **hasEmp**( $\beta$ ), then  $(\alpha\beta + \gamma)^* \rightarrow (\alpha + \beta + \gamma)^*$ .
- (9) If **hasEmp**( $\alpha$ ) and  $sub(\alpha, \beta^*)$ , then  $(\alpha\beta)^* \rightarrow \beta^*$ .
- (10) If **hasEmp**( $\beta$ ) and  $sub(\beta, \alpha^*)$ , then  $(\alpha\beta)^* \rightarrow \alpha^*$ .

32

### (3.2.2) Simplification Rules (Cont.)

- (11) If **hasEmp**( $\alpha$ ) and  $sub(\alpha, (\beta + \gamma)^*)$ , then  $(\alpha\beta + \gamma)^* \rightarrow (\beta + \gamma)^*$ .
- (12) If **hasEmp**( $\beta$ ) and  $sub(\beta, (\alpha + \gamma)^*)$ , then  $(\alpha\beta + \gamma)^* \rightarrow (\alpha + \gamma)^*$ .
- (13) If  $sub(\alpha, \beta)$ , then  $\alpha + \beta \rightarrow \beta$ .
- (14)  $\alpha\beta_1 + \alpha\beta_2 \rightarrow \alpha(\beta_1 + \beta_2)$ .
- (15)  $\alpha_1\beta + \alpha_2\beta \rightarrow (\alpha_1 + \alpha_2)\beta$ .
- (16) If  $sub(\alpha\beta_1, \alpha\beta_2)$ , then  $\alpha(\beta_1 + \beta_2) \rightarrow \alpha\beta_2$ .
- (17) If  $sub(\alpha_1\beta, \alpha_2\beta)$ , then  $(\alpha_1 + \alpha_2)\beta \rightarrow \alpha_2\beta$ .
- (18) If  $sub(\alpha\alpha^*, \beta)$ , then  $\alpha^* + \beta \rightarrow \%_0 + \beta$ .
- (19) If **hasEmp**( $\beta$ ) and  $sub(\alpha\alpha\alpha^*, \beta)$ , then  $\alpha^* + \beta \rightarrow \alpha + \beta$ .
- (20) (alphabet) If  $sub(\alpha^n, \beta)$ , then  $\alpha^{n+1}\alpha^* + \beta \rightarrow \alpha^n\alpha^* + \beta$ .

33

### (3.2.2) Simplification Rules (Cont.)

- (21) (alphabet)  $\alpha + \alpha\beta \rightarrow \alpha(\%_0 + \beta)$ .
- (22) (alphabet)  $\alpha + \beta\alpha \rightarrow (\%_0 + \beta)\alpha$ .
- (23) If  $sub((\alpha + \beta)^*, \alpha^* + \gamma)$ , then  $(\alpha + \beta)^* + \gamma \rightarrow \alpha^* + \gamma$ .
- (24) If  $sub(\gamma', \beta\alpha)$ , then  $(\alpha(\gamma + \gamma')^*\beta)^* \rightarrow (\alpha\gamma^*\beta)^*$ .
- (25)  $\%_0 + \alpha(\alpha + \beta)^* \rightarrow (\alpha\beta^*)^*$ .
- (26)  $\%_0 + (\alpha + \beta)^*\alpha \rightarrow (\beta^*\alpha)^*$ .
- (27)  $\%_0 + \alpha(\beta + \gamma\alpha)^*\gamma \rightarrow (\alpha\beta^*\gamma)^*$ .
- (28) If **size**(**allStr**(alphabet  $\alpha$ )) < **size**  $\alpha$ , and  $sub(\mathbf{allStr}(\mathbf{alphabet} \ \alpha), \alpha)$ , then  $\alpha \rightarrow \mathbf{allStr}(\mathbf{alphabet} \ \alpha)$ .
- (29) If **size**(**allStr**(alphabet  $\alpha$ )) < **size**  $\alpha$ , and  $sub(\mathbf{allStr}(\mathbf{alphabet} \ \alpha), \alpha + \beta)$ , then  $\alpha + \beta \rightarrow \mathbf{allStr}(\mathbf{alphabet} \ \alpha) + \beta$ .

34

## (3.2.2) Structural Rules

We also make use of the following nine *structural rules*, which may be applied to any subtree of a regular expression, and which preserve the alphabet, size, number of concatenations and number of symbols of a regular expression:

- (1)  $(\alpha + \beta) + \gamma \rightarrow \alpha + (\beta + \gamma)$ .
- (2)  $\alpha + (\beta + \gamma) \rightarrow (\alpha + \beta) + \gamma$ .
- (3)  $\alpha(\beta\gamma) \rightarrow (\alpha\beta)\gamma$ .
- (4)  $(\alpha\beta)\gamma \rightarrow \alpha(\beta\gamma)$ .
- (5)  $\alpha + \beta \rightarrow \beta + \alpha$ .
- (6)  $\alpha^*\alpha \rightarrow \alpha\alpha^*$ .
- (7)  $\alpha\alpha^* \rightarrow \alpha^*\alpha$ .
- (8)  $\alpha(\beta\alpha)^* \rightarrow (\alpha\beta)^*\alpha$ .
- (9)  $(\alpha\beta)^*\alpha \rightarrow \alpha(\beta\alpha)^*$ .

35

## (3.2.2) Simplified Regular Expressions

Because the structural rules preserve the size and alphabet of regular expressions, if we start with a regular expression  $\alpha$ , there are only finitely many regular expressions that we can transform  $\alpha$  into using structural rules (we can apply one of the rules to some subtree of  $\alpha$ , giving us  $\beta_1$ , apply a rule to one of the subtrees of  $\beta_1$ , giving us  $\beta_2$ , etc.).

Suppose *sub* is a conservative approximation to subset testing. We say that a regular expression  $\alpha$  is *sub-simplified* iff

- $\alpha$  is weakly simplified, and
- $\alpha$  can't be transformed by our structural rules into a regular expression to which one of our simplification rules (some of which use *sub*) applies.

Thus, if  $\alpha$  is *sub-simplified*, then every subtree of  $\alpha$  is also *sub-simplified*.

36

## (3.2.2) Stronger Simplification Algorithm

The main auxiliary function of our simplification function takes in a weakly simplified regular expression and returns a weakly simplified regular expression. It is defined by well-founded recursion on the relation that first considers regular expression size, and then considers number of concatenations, and then considers number of symbols.

- The function starts working its way through all of the finitely many regular expressions  $\beta$  that its argument  $\alpha$  can be transformed to using our structural rules.
- If one of our simplification rules applies to such a  $\beta$ , then the function applies the rule to  $\beta$ , yielding the result  $\gamma$ , and then calls itself recursively with **weakSimplify**  $\gamma$ .
- Otherwise, the function selects the next value of  $\beta$ , and continues this process.
- If the function exhausts all of the  $\beta$ 's, then it returns  $\alpha$  as its answer.

37

## (3.2.2) Stronger Simplification Algorithm (Cont.)

Our stronger simplification function **simplify** takes in  $sub \in \mathbf{Reg} \rightarrow \mathbf{Reg}$ , and returns the function that takes in  $\alpha \in \mathbf{Reg}$ , and calls the main auxiliary function with **weakSimplify**  $\alpha$ . The function  $sub$  should be a conservative approximation to subset testing.

## (3.2.2) Properties of Stronger Simplification

### Theorem 3.2.37

Suppose `sub` is a conservative approximation to subset testing.

For all  $\alpha \in \mathbf{Reg}$ :

- (1) `simplify sub  $\alpha$`   $\approx$   $\alpha$ ;
- (2) `simplify sub  $\alpha$`  is *sub-simplified*;
- (3) `alphabet(simplify sub  $\alpha$ )`  $\subseteq$  `alphabet  $\alpha$` ;
- (4) `size(simplify sub  $\alpha$ )`  $\leq$  `size  $\alpha$` .

## (3.2.2) Simplifying Regular Expressions in Forlan

The Forlan module `Reg` also defines the functions

```
val weakSimplify : reg -> reg
val fromStrSet   : str set -> reg
val toStrSet     : reg -> str set
val weakSubset   : reg * reg -> bool
```

The function `fromStrSet` converts a finite language into a regular expression generating that language in the most obvious way, and the function `toStrSet` returns the language generated by a regular expression, when that language is finite, and informs the user that the language is infinite, otherwise.

## (3.2.2) Simplifying Regular Expressions (Cont.)

Reg also contains:

```
val simplify      :
    int option * (reg * reg -> bool) -> reg -> reg
val simplifyTrace :
    int option * (reg * reg -> bool) -> reg -> reg
```

If the first argument to `simplify` is `NONE`, then the stronger simplification algorithm runs normally.

But if it's `SOME n`, then the algorithm limits itself to `n` structural rule closure steps at each recursive call. The result will still have all the expected properties, except that it won't necessarily be *sub*-simplified.

E.g., because there are many ways to reorganize `0001(001)*01(001)*` using structural rules, the simplification process takes a long time to terminate on this input.

`simplifyTrace` is like `simplify` except that it explains what it's doing.

41

## (3.2.2) Example Regular Expression Processing

Here are some example uses of these functions:

```
- val reg = Reg.input "";
@ (% + $0)(% + 0*0*0 + 0**)*
@ .
val reg = - : reg
- Reg.output("", Reg.weakSimplify reg);
(% + 0* + 00*0*)*
val it = () : unit
- Reg.output("", Reg.simplify (NONE, Reg.weakSubset) reg);
0*
val it = () : unit
- Reg.toStrSet reg;
language is infinite

uncaught exception Error
```

42

## (3.2.2) Examples (Cont.)

```
- val reg' = Reg.input "";
@ (1 + %)(2 + $)(3 + %*)(4 + $*)
@ .
val reg' = - : reg
- StrSet.output("", Reg.toStrSet reg');
2, 12, 23, 24, 123, 124, 234, 1234
val it = () : unit
- Reg.output("", Reg.weakSimplify reg');
(% + 1)2(% + 3)(% + 4)
val it = () : unit
- Reg.output("", Reg.fromStrSet(StrSet.input ""));
@ hello, there, again
@ .
again + hello + there
val it = () : unit
```

43

## (3.2.2) Examples (Cont.)

```
- val reg'' = Reg.input "";
@ 1 + (% + 0 + 2)(% + 0 + 2)*1 +
@ (1 + (% + 0 + 2)(% + 0 + 2)*1)
@ (% + 0 + 2 + 1(% + 0 + 2)*1)
@ (% + 0 + 2 + 1(% + 0 + 2)*1)*
@ .
val reg'' = - : reg
- Reg.size reg'';
val it = 68 : int
- Reg.size(Reg.weakSimplify reg'');
val it = 68 : int
- Reg.output("",
= Reg.simplify (NONE, Reg.weakSubset) reg'');
(0 + 2)*1(0 + 2 + 1(0 + 2)*1)*
val it = () : unit
```

44

## (3.2.2) Examples (Cont.)

```
- Reg.simplify (NONE, Reg.weakSubset) (Reg.input "");
@ % + 0*0(0 + 1)* + 1*1(0 + 1)*
@ .
val it = - : reg
- Reg.output("", it);
(0 + 1)*
val it = () : unit
- Reg.simplify (NONE, Reg.weakSubset) (Reg.input "");
@ % + 1*0(0 + 1)* + 0*1(0 + 1)*
@ .
val it = - : reg
- Reg.output("", it);
% + (0*1 + 1*0)(0 + 1)*
val it = () : unit
```

This last regular expression is actually equivalent to  $(0 + 1)^*$ . We'll be able to get this result with the perfect subset test of Section 3.12.

45

## (3.2.2) Examples (Cont.)

```
- Reg.simplifyTrace (NONE, Reg.weakSubset) (Reg.input "");
@ 1*(01*01)*
@ .
1*(01*01)*
weakly simplifies to
1*(01*01)*
is transformed by structural rules to
1*((01*0)1)*
is transformed by simplification rule 1 to
(1 + 01*0)*
weakly simplifies to
(1 + 01*0)*
is simplified
val it = - : reg
```

(Simplification Rule (1) is  $\alpha^*(\beta\alpha^*)^* \rightarrow (\alpha + \beta)^*$ .)

46

## (3.2.2) Examples (Cont.)

This takes about 15 seconds on a moderately fast laptop:

```
- Reg.simplifyTrace (SOME 3000, Reg.weakSubset)
=                               (Reg.input "");
@ 0001(001)*01(001)*
@ .
0001(001)*01(001)*
weakly simplifies to
00010(010)*1(001)*
structural rule closure aborted
val it = - : reg
```

On the same computer, without the limitation on structural rule closure steps, it takes about 30 minutes to terminate, with the same result.