3.3: Simplification of Regular Expressions

In this section, we give three algorithms—of increasing power, but decreasing efficiency—for regular expression simplification.

The first algorithm—weak simplification—is defined via a straightforward structural recursion, and is sufficient for many purposes.

The remaining two algorithms—local simplification and global simplification—are based on a set of simplification rules that is still incomplete and evolving.

Regular Expression Complexity

To begin with, let's consider how we might measure the complexity/simplicity of regular expressions. The most obvious criterion is size (remember that regular expressions are trees). But consider this pair of equivalent regular expressions:

$$\alpha = (00^*11^*)^*, \text{ and}$$
$$\beta = \% + 0(0 + 11^*0)^*11^*.$$

Regular Expression Complexity

To begin with, let's consider how we might measure the complexity/simplicity of regular expressions. The most obvious criterion is size (remember that regular expressions are trees). But consider this pair of equivalent regular expressions:

$$\alpha = (00^*11^*)^*, \text{ and}$$
$$\beta = \% + 0(0 + 11^*0)^*11^*.$$

The standard measure of the closure-related complexity of a regular expression is its *star-height*: the maximum number $n \in \mathbb{N}$ such that there is a path from the root of the regular expression to one of its leaves that passes through n closures.

α and β both have star-heights of 2.

Regular Expression Complexity

To begin with, let's consider how we might measure the complexity/simplicity of regular expressions. The most obvious criterion is size (remember that regular expressions are trees). But consider this pair of equivalent regular expressions:

$$\alpha = (00^*11^*)^*, \text{ and}$$
$$\beta = \% + 0(0 + 11^*0)^*11^*.$$

The standard measure of the closure-related complexity of a regular expression is its *star-height*: the maximum number $n \in \mathbb{N}$ such that there is a path from the root of the regular expression to one of its leaves that passes through n closures.

α and β both have star-heights of 2.

Star-height isn't respected by the ways of forming regular expressions: 0 has strictly lower star-height than 0^*, but

Regular Expression Complexity

To begin with, let's consider how we might measure the complexity/simplicity of regular expressions. The most obvious criterion is size (remember that regular expressions are trees). But consider this pair of equivalent regular expressions:

$$\alpha = (00^*11^*)^*, \text{ and}$$
$$\beta = \% + 0(0 + 11^*0)^*11^*.$$

The standard measure of the closure-related complexity of a regular expression is its *star-height*: the maximum number $n \in \mathbb{N}$ such that there is a path from the root of the regular expression to one of its leaves that passes through n closures.

α and β both have star-heights of 2.

Star-height isn't respected by the ways of forming regular expressions: 0 has strictly lower star-height than 0^*, but

Regular Expression Complexity

To begin with, let's consider how we might measure the complexity/simplicity of regular expressions. The most obvious criterion is size (remember that regular expressions are trees). But consider this pair of equivalent regular expressions:

$$\alpha = (00^*11^*)^*, \text{ and}$$
$$\beta = \% + 0(0 + 11^*0)^*11^*.$$

The standard measure of the closure-related complexity of a regular expression is its *star-height*: the maximum number $n \in \mathbb{N}$ such that there is a path from the root of the regular expression to one of its leaves that passes through n closures.

α and β both have star-heights of 2.

Star-height isn't respected by the ways of forming regular expressions: 0 has strictly lower star-height than 0^*, but 01^* has the same star-height as 0^*1^*.

Closure Complexity

Let's define a *closure complexity* to be a nonempty list *ns* of natural numbers that is (not-necessarily strictly) descending.

E.g., $[3, 2, 2, 1]$ is a closure complexity, but $[3, 2, 3]$ and $[\,]$ are not.

We write **CC** for the set of all closure complexities.

Closure Complexity

Let's define a *closure complexity* to be a nonempty list ns of natural numbers that is (not-necessarily strictly) descending.

E.g., $[3, 2, 2, 1]$ is a closure complexity, but $[3, 2, 3]$ and $[\,]$ are not.

We write **CC** for the set of all closure complexities.

For all $n \in \mathbb{N}$, $[n]$ is a *singleton* closure complexity.

The *union* of closure complexities ns and ms ($ns \cup ms$) is the closure complexity that results from putting ns @ ms in descending order, keeping any duplicate elements. E.g., $[3, 2, 2, 1] \cup [4, 2, 1, 0] = [4, 3, 2, 2, 2, 1, 1, 0]$.

The *successor* $\overline{ns}$ of a closure complexity ns is the closure complexity formed by adding one to each element of ns, maintaining the order of the elements. E.g., $\overline{[3, 2, 2, 1]} = [4, 3, 3, 2]$.

Closure Complexity

Proposition 3.3.1

(1) *For all* $ns, ms \in \mathbf{CC}$, $ns \cup ms = ms \cup ns$.

(2) *For all* $ns, ms, ls \in \mathbf{CC}$, $(ns \cup ms) \cup ls = ns \cup (ms \cup ls)$.

(3) *For all* $ns, ms \in \mathbf{CC}$, $\overline{ns \cup ms} = \overline{ns} \cup \overline{ms}$.

Proposition 3.3.2

(1) *For all* $ns, ms \in \mathbf{CC}$, $\overline{ns} = \overline{ms}$ *iff* $ns = ms$.

(2) *For all* $ns, ms, ls \in \mathbf{CC}$, $ns \cup ls = ms \cup ls$ *iff* $ns = ms$.

Closure Complexity

We define a relation $<_{cc}$ on **CC** by: for all $ns, ms \in$ **CC**, $ns <_{cc} ms$ iff either:

- $ms = ns \mathbin{@} ls$ for some $ls \in$ **CC**; or
- there is an $i \in \mathbb{N} - \{0\}$ such that
  - $i \leq |ns|$ and $i \leq |ms|$,
  - for all $j \in [1 : i - 1]$, $ns\,j = ms\,j$, and
  - $ns\,i < ms\,i$.

Closure Complexity

We define a relation $<_{cc}$ on **CC** by: for all $ns, ms \in$ **CC**, $ns <_{cc} ms$ iff either:

- $ms = ns$ @ ls for some $ls \in$ **CC**; or
- there is an $i \in \mathbb{N} - \{0\}$ such that
  - $i \leq |ns|$ and $i \leq |ms|$,
  - for all $j \in [1 : i - 1]$, $ns\, j = ms\, j$, and
  - $ns\, i < ms\, i$.

E.g., $[2, 2] <_{cc} [2, 2, 1]$ and $[2, 1, 1, 0, 0] <_{cc} [2, 2, 1]$.

Proposition 3.3.3

(1) *For all* $ns, ms \in \mathbf{CC}$, $\overline{ns} <_{cc} \overline{ms}$ *iff* $ns <_{cc} ms$.

(2) *For all* $ns, ms, ls \in \mathbf{CC}$, $ns \cup ls <_{cc} ms \cup ls$ *iff* $ns <_{cc} ms$.

(3) *For all* $ns, ms \in \mathbf{CC}$, $ns <_{cc} ns \cup ms$.

Proposition 3.3.4

$<_{cc}$ *is a strict total ordering on* $\mathbf{CC}$.

Proposition 3.3.5

$<_{cc}$ *is a well-founded relation on* $\mathbf{CC}$.

Closure Complexity

Now we can define the closure complexity of a regular expression. Define the function $\mathbf{cc} \in \mathbf{Reg} \rightarrow \mathbf{CC}$ by structural recursion:

$$\begin{aligned}
\mathbf{cc}\,\% &= [0]; \\
\mathbf{cc}\,\$ &= [0]; \\
\mathbf{cc}\,a &= [0], \text{ for all } a \in \mathbf{Sym}; \\
\mathbf{cc}(*(\alpha)) &= \overline{\mathbf{cc}\,\alpha}, \text{ for all } \alpha \in \mathbf{Reg}; \\
\mathbf{cc}(@(\alpha, \beta)) &= \mathbf{cc}\,\alpha \cup \mathbf{cc}\,\beta, \text{ for all } \alpha, \beta \in \mathbf{Reg}; \text{ and} \\
\mathbf{cc}(+(\alpha, \beta)) &= \mathbf{cc}\,\alpha \cup \mathbf{cc}\,\beta, \text{ for all } \alpha, \beta \in \mathbf{Reg}.
\end{aligned}$$

We say that $\mathbf{cc}\,\alpha$ is *the closure complexity of* α.

Closure Complexity

Now we can define the closure complexity of a regular expression. Define the function $\textbf{cc} \in \textbf{Reg} \to \textbf{CC}$ by structural recursion:

$$
\begin{aligned}
\textbf{cc}\,\% &= [0]; \\
\textbf{cc}\,\$ &= [0]; \\
\textbf{cc}\,a &= [0], \text{ for all } a \in \textbf{Sym}; \\
\textbf{cc}(*(\alpha)) &= \overline{\textbf{cc}\,\alpha}, \text{ for all } \alpha \in \textbf{Reg}; \\
\textbf{cc}(@(\alpha,\beta)) &= \textbf{cc}\,\alpha \cup \textbf{cc}\,\beta, \text{ for all } \alpha,\beta \in \textbf{Reg}; \text{ and} \\
\textbf{cc}(+(\alpha,\beta)) &= \textbf{cc}\,\alpha \cup \textbf{cc}\,\beta, \text{ for all } \alpha,\beta \in \textbf{Reg}.
\end{aligned}
$$

We say that $\textbf{cc}\,\alpha$ is *the closure complexity of* α.

E.g.,

$$
\begin{aligned}
\textbf{cc}((12^*)^*) &= \overline{\textbf{cc}(12^*)} = \overline{\textbf{cc}\,1 \cup \textbf{cc}(2^*)} = \overline{[0] \cup \overline{\textbf{cc}\,2}} \\
&= \overline{[0] \cup \overline{[0]}} = \overline{[0] \cup [1]} = \overline{[1,0]} = [2,1].
\end{aligned}
$$

Closure Complexity

Returning to our initial examples, we have that
$\mathbf{cc}((00^*11^*)^*) = [2, 2, 1, 1]$ and
$\mathbf{cc}(\% + 0(0 + 11^*0)^*11^*) = [2, 1, 1, 1, 1, 0, 0, 0]$.

Since $[2, 1, 1, 1, 1, 0, 0, 0] <_{cc} [2, 2, 1, 1]$, the closure complexity of $\% + 0(0 + 11^*0)^*11^*$ is strictly smaller than the closure complexity of $(00^*11^*)^*$.

Proposition 3.3.6
For all $\alpha \in \textbf{Reg}$, $|\textbf{cc}\,\alpha| =$.

Proof. An easy induction on regular expressions. □

Exercise 3.3.7
Find regular expressions α *and* β *such that* $\textbf{cc}\,\alpha = \textbf{cc}\,\beta$ *but* $\textbf{size}\,\alpha \neq \textbf{size}\,\beta$.

Proposition 3.3.9
Suppose $\alpha, \beta, \beta' \in \textbf{Reg}$, $\textbf{cc}\,\beta = \textbf{cc}\,\beta'$, $pat \in \textbf{Path}$ *is valid for* α, *and* β *is the subtree of* α *at position* pat. *Let* α' *be the result of replacing the subtree at position* pat *in* α *by* β'. *Then* $\textbf{cc}\,\alpha = \textbf{cc}\,\alpha'$.

Proof. By induction on α. □

Proposition 3.3.6
For all $\alpha \in \textbf{Reg}$, $|\textbf{cc}\,\alpha| = \textbf{numLeaves}\,\alpha$.

Proof. An easy induction on regular expressions. □

Exercise 3.3.7
Find regular expressions α *and* β *such that* $\textbf{cc}\,\alpha = \textbf{cc}\,\beta$ *but* $\textbf{size}\,\alpha \neq \textbf{size}\,\beta$.

Proposition 3.3.9
Suppose $\alpha, \beta, \beta' \in \textbf{Reg}$, $\textbf{cc}\,\beta = \textbf{cc}\,\beta'$, $\mathit{pat} \in \textbf{Path}$ *is valid for* α, *and* β *is the subtree of* α *at position* pat. *Let* α' *be the result of replacing the subtree at position* pat *in* α *by* β'. *Then* $\textbf{cc}\,\alpha = \textbf{cc}\,\alpha'$.

Proof. By induction on α. □

Proposition 3.3.11

*Suppose $\alpha, \beta, \beta' \in$ **Reg**, **cc** $\beta' <_{cc}$ **cc** β, pat $\in$ **Path** is valid for α, and β is the subtree of α at position pat. Let α' be the result of replacing the subtree at position pat in α by β'. Then* **cc** $\alpha' <_{cc}$ **cc** α.

Proof. By induction on α. □

Regular Expression Complexity

When judging the relative complexities of regular expressions α and β, we will first look at how their closure complexities are related.

And, when their closure complexities are equal, we will look at how their sizes are related. To finish explaining how we will judge the relative complexity of regular expressions, we need three definitions.

Numbers of Concatenations and Symbols

We write **numConcats** α and **numSyms** α for the number of concatenations and symbols, respectively, in α.

E.g., **numConcats**$(((01)^*(01))^*) = 3$. and **numSyms**$((0^*1) + 0) = 3$.

Standardization

We say that a regular expression α is *standardized* iff none of α's subtrees have any of the following forms:

- $(\beta_1 + \beta_2) + \beta_3$ (we can avoid needing parentheses, and make a regular expression easier to understand/process from left-to-right, by grouping unions to the right);

Standardization

We say that a regular expression α is *standardized* iff none of α's subtrees have any of the following forms:

- $(\beta_1 + \beta_2) + \beta_3$ (we can avoid needing parentheses, and make a regular expression easier to understand/process from left-to-right, by grouping unions to the right);
- $\beta_1 + \beta_2$, where $\beta_1 > \beta_2$, or $\beta_1 + (\beta_2 + \beta_3)$, where $\beta_1 > \beta_2$ (see Section 3.1 of book for our ordering on regular expressions—but unions are greater than all other kinds of regular expressions));

Standardization

We say that a regular expression α is *standardized* iff none of α's subtrees have any of the following forms:

- $(\beta_1 + \beta_2) + \beta_3$ (we can avoid needing parentheses, and make a regular expression easier to understand/process from left-to-right, by grouping unions to the right);
- $\beta_1 + \beta_2$, where $\beta_1 > \beta_2$, or $\beta_1 + (\beta_2 + \beta_3)$, where $\beta_1 > \beta_2$ (see Section 3.1 of book for our ordering on regular expressions—but unions are greater than all other kinds of regular expressions));
- $(\beta_1\beta_2)\beta_3$ (we can avoid needing parentheses, and make a regular expression easier to understand/process from left-to-right, by grouping concatenations to the right); and

Standardization

We say that a regular expression α is *standardized* iff none of α's subtrees have any of the following forms:

- $(\beta_1 + \beta_2) + \beta_3$ (we can avoid needing parentheses, and make a regular expression easier to understand/process from left-to-right, by grouping unions to the right);
- $\beta_1 + \beta_2$, where $\beta_1 > \beta_2$, or $\beta_1 + (\beta_2 + \beta_3)$, where $\beta_1 > \beta_2$ (see Section 3.1 of book for our ordering on regular expressions—but unions are greater than all other kinds of regular expressions));
- $(\beta_1\beta_2)\beta_3$ (we can avoid needing parentheses, and make a regular expression easier to understand/process from left-to-right, by grouping concatenations to the right); and
- $\beta^*\beta$, $\beta^*(\beta\gamma)$, $(\beta_1\beta_2)^*\beta_1$ or $(\beta_1\beta_2)^*\beta_1\gamma$ (moving closures to the right makes a regular expression easier to understand/process from left-to-right).

Judging Relative Complexity

Returning to our assessment of regular expression complexity, suppose that α and β are regular expressions generating %. Then $(\alpha\beta)^*$ and $(\alpha + \beta)^*$ are equivalent, and have the same closure complexity and size,

Judging Relative Complexity

Returning to our assessment of regular expression complexity, suppose that α and β are regular expressions generating %. Then $(\alpha\beta)^*$ and $(\alpha + \beta)^*$ are equivalent, and have the same closure complexity and size, but will will prefer the latter over the former, because unions are generally more amenable to understanding and processing than concatenations.

Judging Relative Complexity

Returning to our assessment of regular expression complexity, suppose that α and β are regular expressions generating %. Then $(\alpha\beta)^*$ and $(\alpha + \beta)^*$ are equivalent, and have the same closure complexity and size, but will will prefer the latter over the former, because unions are generally more amenable to understanding and processing than concatenations.

Consequently, when two regular expression have the same closure complexity and size, we will judge their relative complexity according to their numbers of concatenations.

Judging Relative Complexity

Next, consider the regular expressions $0 + 01$ and $0(\% + 1)$.

These regular expressions have the same closure complexity $[0, 0, 0]$, size (5) and number of concatenations (1).

Judging Relative Complexity

Next, consider the regular expressions $0 + 01$ and $0(\% + 1)$.

These regular expressions have the same closure complexity $[0, 0, 0]$, size (5) and number of concatenations (1).

We would like to consider the latter to be simpler than the former, since in general we would like to prefer $\alpha(\% + \beta)$ over $\alpha + \alpha\beta$.

Judging Relative Complexity

Next, consider the regular expressions $0 + 01$ and $0(\% + 1)$.

These regular expressions have the same closure complexity $[0, 0, 0]$, size (5) and number of concatenations (1).

We would like to consider the latter to be simpler than the former, since in general we would like to prefer $\alpha(\% + \beta)$ over $\alpha + \alpha\beta$.

And we can base this preference on the fact that the number of symbols of $0(\% + 1)$ (2) is one less than the number of symbols of $0 + 01$.

Judging Relative Complexity

Next, consider the regular expressions $0 + 01$ and $0(\% + 1)$.

These regular expressions have the same closure complexity $[0, 0, 0]$, size (5) and number of concatenations (1).

We would like to consider the latter to be simpler than the former, since in general we would like to prefer $\alpha(\% + \beta)$ over $\alpha + \alpha\beta$.

And we can base this preference on the fact that the number of symbols of $0(\% + 1)$ (2) is one less than the number of symbols of $0 + 01$.

Thus, when regular expressions have identical closure complexity, size and number of concatenations, we will use their relative numbers of symbols to judge their relative complexity.

Judging Relative Complexity

Finally, when regular expressions have the same closure complexity, size, number of concatenations, and number of symbols, we will judge their relative complexity according to whether they are standardized, thinking that a standardized regular expression is simpler than one that is not standardized.

Judging Relative Complexity

We define a relation $<_{\textsf{simp}}$ on **Reg** by, for all $\alpha, \beta \in$ **Reg**, $\alpha <_{\textsf{simp}} \beta$ iff:

- **cc** $\alpha <_{cc}$ **cc** β; or

We read $\alpha <_{\textsf{simp}} \beta$ as α is *simpler* (less *complex*) than β.

Judging Relative Complexity

We define a relation $<_{\mathsf{simp}}$ on **Reg** by, for all $\alpha, \beta \in$ **Reg**, $\alpha <_{\mathsf{simp}} \beta$ iff:

- **cc** $\alpha <_{cc}$ **cc** β; or
- **cc** $\alpha =$ **cc** β but **size** $\alpha <$ **size** β; or

We read $\alpha <_{\mathsf{simp}} \beta$ as α is *simpler* (less *complex*) than β.

Judging Relative Complexity

We define a relation $<_{\textsf{simp}}$ on **Reg** by, for all $\alpha, \beta \in$ **Reg**, $\alpha <_{\textsf{simp}} \beta$ iff:

- **cc** $\alpha <_{cc}$ **cc** β; or
- **cc** $\alpha =$ **cc** β but **size** $\alpha <$ **size** β; or
- **cc** $\alpha =$ **cc** β and **size** $\alpha =$ **size** β, but **numConcats** $\alpha <$ **numConcats** β; or

We read $\alpha <_{\textsf{simp}} \beta$ as α is *simpler* (less *complex*) than β.

Judging Relative Complexity

We define a relation $<_{\mathsf{simp}}$ on **Reg** by, for all $\alpha, \beta \in$ **Reg**, $\alpha <_{\mathsf{simp}} \beta$ iff:

- **cc** $\alpha <_{cc}$ **cc** β; or
- **cc** $\alpha =$ **cc** β but **size** $\alpha <$ **size** β; or
- **cc** $\alpha =$ **cc** β and **size** $\alpha =$ **size** β, but **numConcats** $\alpha <$ **numConcats** β; or
- **cc** $\alpha =$ **cc** β, **size** $\alpha =$ **size** β and **numConcats** $\alpha =$ **numConcats** β, but **numSyms** $\alpha <$ **numSyms** β; or

We read $\alpha <_{\mathsf{simp}} \beta$ as α is *simpler* (less *complex*) than β.

Judging Relative Complexity

We define a relation $<_{\textbf{simp}}$ on **Reg** by, for all $\alpha, \beta \in$ **Reg**, $\alpha <_{\textbf{simp}} \beta$ iff:

- **cc** $\alpha <_{cc}$ **cc** β; or
- **cc** $\alpha =$ **cc** β but **size** $\alpha <$ **size** β; or
- **cc** $\alpha =$ **cc** β and **size** $\alpha =$ **size** β, but **numConcats** $\alpha <$ **numConcats** β; or
- **cc** $\alpha =$ **cc** β, **size** $\alpha =$ **size** β and **numConcats** $\alpha =$ **numConcats** β, but **numSyms** $\alpha <$ **numSyms** β; or
- **cc** $\alpha =$ **cc** β, **size** $\alpha =$ **size** β, **numConcats** $\alpha =$ **numConcats** β and **numSyms** $\alpha =$ **numSyms** β, but α is standardized and β is not standardized.

We read $\alpha <_{\textbf{simp}} \beta$ as α is *simpler* (less *complex*) than β.

Judging Relative Complexity

We define a relation $\equiv_{\mathsf{simp}}$ on **Reg** by, for all $\alpha, \beta \in$ **Reg**, $\alpha \equiv_{\mathsf{simp}} \beta$ iff α and β have the same closure complexity, size, numbers of concatenations, numbers of symbols, and status of being (or not being) standardized.

We read $\alpha \equiv_{\mathsf{simp}} \beta$ as α and β have the *same complexity*.

Judging Relative Complexity

We define a relation $\equiv_{\mathsf{simp}}$ on **Reg** by, for all $\alpha, \beta \in$ **Reg**, $\alpha \equiv_{\mathsf{simp}} \beta$ iff α and β have the same closure complexity, size, numbers of concatenations, numbers of symbols, and status of being (or not being) standardized.

We read $\alpha \equiv_{\mathsf{simp}} \beta$ as α and β have the *same complexity*.

For example, the following regular expressions are equivalent and have the same complexity:

$$1(01 + 10) + (\% + 01)1 \quad \text{and} \quad 011 + 1(\% + 01 + 10).$$

Proposition 3.3.12

(1) $<_{\mathbf{simp}}$ *is transitive.*

(2) $\equiv_{\mathbf{simp}}$ *is reflexive on* **Reg**, *transitive and symmetric.*

(3) *For all* $\alpha, \beta \in$ **Reg**, *exactly one of the following holds:* $\alpha <_{\mathbf{simp}} \beta$, $\beta <_{\mathbf{simp}} \alpha$ *or* $\alpha \equiv_{\mathbf{simp}} \beta$.

Closure Complexity in Forlan

The Forlan module `Reg` defines the abstract type `cc` of closure complexities, along with these functions:

```
val ccToList  : cc -> int list
val singCC    : int -> cc
val unionCC   : cc * cc -> cc
val succCC    : cc -> cc
val cc        : reg -> cc
val compareCC : cc * cc -> order
```

Here are some examples of how these functions can be used:

```
- val ns =
=        Reg.succCC
=        (Reg.unionCC(Reg.singCC 1, Reg.singCC 1));
val ns = - : Reg.cc
- Reg.ccToList ns;
val it = [2,2] : int list
- val ms = Reg.unionCC(ns, Reg.succCC ns);
val ms = - : Reg.cc
- Reg.ccToList ms;
val it = [3,3,2,2] : int list
```

Closure Complexity in Forlan

```
- Reg.ccToList(Reg.cc(Reg.fromString "(00*11*)*"));
val it = [2,2,1,1] : int list
- Reg.ccToList
= (Reg.cc(Reg.fromString "% + 0(0 + 11*0)*11*"));
val it = [2,1,1,1,1,0,0,0] : int list
- Reg.compareCC
= (Reg.cc(Reg.fromString "(00*11*)*"),
=  Reg.cc(Reg.fromString "% + 0(0 + 11*0)*11*"));
val it = GREATER : order
- Reg.compareCC
= (Reg.cc(Reg.fromString "(00*11*)*"),
=  Reg.cc(Reg.fromString "(1*10*0)*"));
val it = EQUAL : order
```

Regular Expression Complexity in Forlan

The module Reg also includes these functions:

```
val numConcats        : reg -> int
val numSyms           : reg -> int
val standardized      : reg -> bool
val compareComplexity : reg * reg -> order
```

Regular Expression Complexity in Forlan

The module `Reg` also includes these functions:

```
val numConcats        : reg -> int
val numSyms           : reg -> int
val standardized      : reg -> bool
val compareComplexity : reg * reg -> order
```

Here are some examples of how these functions can be used:

```
- Reg.numConcats(Reg.fromString "(01)*(10)*");
val it = 3 : int
- Reg.numSyms(Reg.fromString "(01)*(10)*");
val it = 4 : int
- Reg.standardized(Reg.fromString "00*1");
val it = true : bool
- Reg.standardized(Reg.fromString "00*0");
val it = false : bool
```

Regular Expression Complexity in Forlan

```
- Reg.compareComplexity
= (Reg.fromString "(00*11*)*",
=  Reg.fromString "% + 0(0 + 11*0)*11*");
val it = GREATER : order
- Reg.compareComplexity
= (Reg.fromString "0**1**", Reg.fromString "(01)**");
val it = GREATER : order
- Reg.compareComplexity
= (Reg.fromString "(0*1*)*",
=  Reg.fromString "(0*+1*)*");
val it = GREATER : order
- Reg.compareComplexity
= (Reg.fromString "0+01", Reg.fromString "0(%+1)");
val it = GREATER : order
- Reg.compareComplexity
= (Reg.fromString "(01)2", Reg.fromString "012");
val it = GREATER : order
```

Regular Expression Complexity in Forlan

```
- Reg.compareComplexity
= (Reg.fromString "1(01+10)+(%+01)1",
=  Reg.fromString "011+1(%+01+10)");
val it = EQUAL : order
```

Weak Simplification

We say that a regular expression α is *weakly simplified* iff α is standardized and none of α's subtrees have any of the following forms:

- $\$ + \beta$ or $\beta + \$$ (the $\$$ is redundant);

Weak Simplification

We say that a regular expression α is *weakly simplified* iff α is standardized and none of α's subtrees have any of the following forms:

- $\$ + \beta$ or $\beta + \$$ (the $\$$ is redundant);
- $\beta + \beta$ or $\beta + (\beta + \gamma)$ (the duplicate occurrence of β is redundant);

Weak Simplification

We say that a regular expression α is *weakly simplified* iff α is standardized and none of α's subtrees have any of the following forms:

- $\$ + \beta$ or $\beta + \$$ (the $\$$ is redundant);
- $\beta + \beta$ or $\beta + (\beta + \gamma)$ (the duplicate occurrence of β is redundant);
- $\%\beta$ or $\beta\%$ (the $\%$ is redundant);

Weak Simplification

We say that a regular expression α is *weakly simplified* iff α is standardized and none of α's subtrees have any of the following forms:

- $\$ + \beta$ or $\beta + \$$ (the $\$$ is redundant);
- $\beta + \beta$ or $\beta + (\beta + \gamma)$ (the duplicate occurrence of β is redundant);
- $\%\beta$ or $\beta\%$ (the $\%$ is redundant);
- $\$\beta$ or $\beta\$$ (both are equivalent to $\$$); and

Weak Simplification

We say that a regular expression α is *weakly simplified* iff α is standardized and none of α's subtrees have any of the following forms:

- $\$ + \beta$ or $\beta + \$$ (the $\$$ is redundant);
- $\beta + \beta$ or $\beta + (\beta + \gamma)$ (the duplicate occurrence of β is redundant);
- $\%\beta$ or $\beta\%$ (the $\%$ is redundant);
- $\$\beta$ or $\beta\$$ (both are equivalent to $\$$); and
- $\%^*$ or $\$^*$ or $(\beta^*)^*$ (the first two can be replaced by $\%$, and the extra closure can be omitted in the third case).

Proposition 3.3.13

(1) *For all* $\alpha \in \mathbf{Reg}$*, if* α *is weakly simplified and* $L(\alpha) = \emptyset$*, then* $\alpha =$

Weak Simplification

Proposition 3.3.13

(1) *For all* $\alpha \in \mathbf{Reg}$, *if* α *is weakly simplified and* $L(\alpha) = \emptyset$, *then* $\alpha = \$$.

(2) *For all* $\alpha \in \mathbf{Reg}$, *if* α *is weakly simplified and* $L(\alpha) = \{\%\}$, *then* $\alpha =$

Proposition 3.3.13

(1) *For all* $\alpha \in$ **Reg**, *if* α *is weakly simplified and* $L(\alpha) = \emptyset$, *then* $\alpha = \$$.

(2) *For all* $\alpha \in$ **Reg**, *if* α *is weakly simplified and* $L(\alpha) = \{\%\}$, *then* $\alpha = \%$.

(3) *For all* $\alpha \in$ **Reg**, *for all* $a \in$ **Sym**, *if* α *is weakly simplified and* $L(\alpha) = \{a\}$, *then* $\alpha =$

Proposition 3.3.13

(1) *For all* $\alpha \in$ **Reg**, *if* α *is weakly simplified and* $L(\alpha) = \emptyset$, *then* $\alpha = \$$.

(2) *For all* $\alpha \in$ **Reg**, *if* α *is weakly simplified and* $L(\alpha) = \{\%\}$, *then* $\alpha = \%$.

(3) *For all* $\alpha \in$ **Reg**, *for all* $a \in$ **Sym**, *if* α *is weakly simplified and* $L(\alpha) = \{a\}$, *then* $\alpha = a$.

Proof.

Weak Simplification

Proposition 3.3.13

(1) *For all* $\alpha \in \mathbf{Reg}$*, if* α *is weakly simplified and* $L(\alpha) = \emptyset$*, then* $\alpha = \$$*.*

(2) *For all* $\alpha \in \mathbf{Reg}$*, if* α *is weakly simplified and* $L(\alpha) = \{\%\}$*, then* $\alpha = \%$*.*

(3) *For all* $\alpha \in \mathbf{Reg}$*, for all* $a \in \mathbf{Sym}$*, if* α *is weakly simplified and* $L(\alpha) = \{a\}$*, then* $\alpha = a$*.*

Proof. The three parts are proved in order, using induction on regular expressions.

Weak Simplification

Proposition 3.3.13

(1) *For all $\alpha \in$ **Reg**, if α is weakly simplified and $L(\alpha) = \emptyset$, then $\alpha = \$$.*

(2) *For all $\alpha \in$ **Reg**, if α is weakly simplified and $L(\alpha) = \{\%\}$, then $\alpha = \%$.*

(3) *For all $\alpha \in$ **Reg**, for all $a \in$ **Sym**, if α is weakly simplified and $L(\alpha) = \{a\}$, then $\alpha = a$.*

Proof. The three parts are proved in order, using induction on regular expressions. We will show the concatenation case of part (3). Suppose $\alpha, \beta \in$ **Reg** and assume the inductive hypothesis: for all $a \in$ **Sym**, if α is weakly simplified and $L(\alpha) = \{a\}$, then $\alpha = a$, and for all $a \in$ **Sym**, if β is weakly simplified and $L(\beta) = \{a\}$, then $\beta = a$. Suppose $a \in$ **Sym**, and assume that $\alpha\beta$ is weakly simplified and $L(\alpha\beta) = \{a\}$. We must show that $\alpha\beta = a$. We have that α and β are weakly simplified.

Weak Simplification

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

Weak Simplification

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

- Suppose $L(\alpha) = \{a\}$ and $L(\beta) = \{\%\}$.

- Suppose $L(\alpha) = \{\%\}$ and $L(\beta) = \{a\}$. The proof of this case is similar to that of the other one.

□

Weak Simplification

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 β is weakly simplified and $L(\beta) = \{\%\}$, part (2) tells us that $\beta = \%$.

- Suppose $L(\alpha) = \{\%\}$ and $L(\beta) = \{a\}$. The proof of this case is similar to that of the other one.

□

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 β is weakly simplified and $L(\beta) = \{\%\}$, part (2) 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.

□

Proposition 3.3.14

For all $\alpha \in$ **Reg**, *if* α *is weakly simplified, then* **alphabet**$(L(\alpha))$ **alphabet** α.

Proposition 3.3.14
For all $\alpha \in \textbf{Reg}$, *if* α *is weakly simplified, then* $\textbf{alphabet}(L(\alpha)) = \textbf{alphabet}\,\alpha$.

Proposition 3.3.14

For all $\alpha \in \mathbf{Reg}$, *if* α *is weakly simplified, then* $\mathbf{alphabet}(L(\alpha)) = \mathbf{alphabet}\,\alpha$.

Proposition 3.3.15

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

Weak Simplification

Proposition 3.3.14

For all $\alpha \in \textbf{Reg}$, *if* α *is weakly simplified, then* $\textbf{alphabet}(L(\alpha)) = \textbf{alphabet}\,\alpha$.

Proposition 3.3.15

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

Proposition 3.3.14

*For all $\alpha \in$ **Reg**, if α is weakly simplified, then* **alphabet**$(L(\alpha)) =$ **alphabet** α.

Proposition 3.3.15

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

Proposition 3.3.16

*For all $\alpha \in$ **Reg**, if α is weakly simplified and α has one or more closures, then $L(\alpha)$ is*

Weak Simplification

Proposition 3.3.14

For all $\alpha \in$ **Reg**, *if* α *is weakly simplified, then* **alphabet**$(L(\alpha)) =$ **alphabet** α.

Proposition 3.3.15

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

Proposition 3.3.16

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

Weak Simplification

Let

$$\mathbf{WS} = \{\, \alpha \in \mathbf{Reg} \mid \alpha \text{ is weakly simplified} \,\}.$$

Weak Simplification

Let

$$\textbf{WS} = \{\, \alpha \in \textbf{Reg} \mid \alpha \text{ is weakly simplified} \,\}.$$

Define a function **deepClosure** $\in$ **WS** $\to$ **WS** as follows. For all $\alpha \in$ **WS**:

$$\begin{aligned}
\textbf{deepClosure}\,\% &= \%, \\
\textbf{deepClosure}\,\$ &= \%, \\
\textbf{deepClosure}\,(*(\alpha)) &= \alpha^*, \text{ and} \\
\textbf{deepClosure}\,\alpha &= \alpha^*, \text{ if } \alpha \notin \{\%, \$\} \text{ and } \alpha \text{ is not a closure.}
\end{aligned}$$

Weak Simplification

Define a function **deepConcat** $\in$ **WS** $\times$ **WS** $\to$ **WS** as follows. For all $\alpha, \beta \in$ **WS**:

$$
\begin{aligned}
\textbf{deepConcat}(\alpha, \$) &= \$,\\
\textbf{deepConcat}(\$, \alpha) &= \$, \text{ if } \alpha \neq \$,\\
\textbf{deepConcat}(\alpha, \%) &= \alpha, \text{ if } \alpha \neq \$,\\
\textbf{deepConcat}(\%, \alpha) &= \alpha, \text{ if } \alpha \notin \{\$, \%\}, \text{ and}\\
\textbf{deepConcat}(\alpha, \beta) &= \textbf{shiftClosuresRight}(\textbf{rightConcat}(\alpha, \beta)),\\
&\qquad \text{if } \alpha, \beta \notin \{\$, \%\},
\end{aligned}
$$

If α_n is not a concatenation, then **rightConcat**$(\alpha_1 \cdots \alpha_n, \beta) = \alpha_1 \cdots \alpha_n \beta$. **shiftClosuresRight** repeatedly applies the following rules down the rightmost branch: $\beta^* \beta \to$ **rightConcat**(β, β^*), $\beta^* \beta \gamma \to \beta \beta^* \gamma$, $(\beta_1 \beta_2)^* \beta_1 \to \beta_1($**rightConcat**$(\beta_2, \beta_1))^*$ and $(\beta_1 \beta_2)^* \beta_1 \gamma \to \beta_1($**rightConcat**$(\beta_2, \beta_1))^* \gamma$.

Weak Simplification

Define a function **deepUnion** $\in$ **WS** $\times$ **WS** $\to$ **WS** as follows. For all $\alpha, \beta \in$ **WS**:

deepUnion$(\alpha, \$) = \alpha,$
deepUnion$(\$, \alpha) = \alpha,$ if $\alpha \neq \$,$ and
deepUnion$(\alpha, \beta) =$ **sortUnions**(**rightUnion**$(\alpha, \beta)),$ if $\alpha \neq \$$ and $\beta \neq \$.$

If α_n is not a union, then
rightUnion$(\alpha_1 + \cdots + \alpha_n, \beta) = \alpha_1 + \cdots + \alpha_n + \beta$.

sortUnions sorts the unions down the right branch using our total ordering on **Reg**, removing duplicates.

Weak Simplification

Define **weaklySimplify** $\in$ **Reg** $\to$ **WS** by structural recursion:

- **weaklySimplify** % = %;
- **weaklySimplify** \$ = \$;
- **weaklySimplify** $a = a$, for all $a \in$ **Sym**;
- **weaklySimplify**$(*(\alpha)) =$

  deepClosure(**weaklySimplify** α);

- **weaklySimplify**(@$(\alpha, \beta)) =$

  deepConcat(**weaklySimplify** α, **weaklySimplify** β); and

- **weaklySimplify**$(+(\alpha, \beta)) =$

  deepUnion(**weaklySimplify** α, **weaklySimplify** β).

Proposition 3.3.22
For all $\alpha \in \textbf{Reg}$*:*

(1) $\textbf{weaklySimplify}\,\alpha \approx \alpha$;

(2) $\textbf{alphabet}(\textbf{weaklySimplify}(\alpha)) \subseteq \textbf{alphabet}\,\alpha$;

(3) $\textbf{cc}(\textbf{weaklySimplify}\,\alpha) \leq_{cc} \textbf{cc}\,\beta$;

(4) $\textbf{size}(\textbf{weaklySimplify}\,\alpha) \leq \textbf{size}\,\alpha$;

(5) $\textbf{numSyms}(\textbf{weaklySimplify}\,\alpha) \leq \textbf{numSyms}\,\alpha$; *and*

(6) $\textbf{numConcats}(\textbf{weaklySimplify}\,\alpha) \leq \textbf{numConcats}\,\alpha$.

Proof. By induction on regular expressions. □

Weak Simplification

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.

Weak Simplification

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

Weak Simplification in Forlan

The Forlan module Reg defines the following functions relating to weak simplification:

```
val weaklySimplified : reg -> bool
val weaklySimplify   : reg -> reg
val toStrSet         : reg -> str set
```

Weak Simplification in Forlan

The Forlan module `Reg` defines the following functions relating to weak simplification:

```
val weaklySimplified : reg -> bool
val weaklySimplify   : reg -> reg
val toStrSet         : reg -> str set
```

Here are some examples of how these functions can be used:

```
- val reg = Reg.input "";
@ (% + $0)(% + 00*0 + 0**)*
@ .
val reg = - : reg
- Reg.output("", Reg.weaklySimplify reg);
(% + 0* + 000*)*
val it = () : unit
- Reg.toStrSet reg;
language is infinite

uncaught exception Error
```

Weak Simplification in Forlan

```
- 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.weaklySimplify reg');
(% + 1)2(% + 3)(% + 4)
val it = () : unit
- Reg.output
= ("",
=  Reg.weaklySimplify(Reg.fromString "(00*11*)*"));
(00*11*)*
val it = () : unit
```

Local and Global Simplification

We define a function **hasEmp** $\in$ **Reg** $\rightarrow$ **Bool** such that, for all $\alpha \in$ **Reg**, $\% \in L(\alpha)$ iff **hasEmp** $\alpha =$ **true**.

Local and Global Simplification

We define a function **hasEmp** $\in$ **Reg** $\to$ **Bool** such that, for all $\alpha \in$ **Reg**, $\% \in L(\alpha)$ iff **hasEmp** $\alpha =$ **true**.

We define a function **obviousSubset** $\in$ **Reg** $\times$ **Reg** $\to \{$**true**, **false**$\}$ that is a *conservative approximation to subset testing*: for all $\alpha, \beta \in$ **Reg**, if **obviousSubset**$(\alpha, \beta) =$ **true**, then $L(\alpha) \subseteq L(\beta)$.

Local and Global Simplification

We define a function **hasEmp** $\in$ **Reg** $\to$ **Bool** such that, for all $\alpha \in$ **Reg**, $\% \in L(\alpha)$ iff **hasEmp** $\alpha =$ **true**.

We define a function **obviousSubset** $\in$ **Reg** $\times$ **Reg** $\to \{$**true**, **false**$\}$ that is a *conservative approximation to subset testing*: for all $\alpha, \beta \in$ **Reg**, if **obviousSubset**$(\alpha, \beta) =$ **true**, then $L(\alpha) \subseteq L(\beta)$.

On the positive side, we have that, e.g., **obviousSubset**$(0^*011^*1, 0^*1^*) =$ **true**.

Local and Global Simplification

We define a function **hasEmp** $\in$ **Reg** $\to$ **Bool** such that, for all $\alpha \in$ **Reg**, $\% \in L(\alpha)$ iff **hasEmp** $\alpha =$ **true**.

We define a function **obviousSubset** $\in$ **Reg** $\times$ **Reg** $\to$ {**true**, **false**} that is a *conservative approximation to subset testing*: for all $\alpha, \beta \in$ **Reg**, if **obviousSubset**$(\alpha, \beta) =$ **true**, then $L(\alpha) \subseteq L(\beta)$.

On the positive side, we have that, e.g., **obviousSubset**$(0^*011^*1, 0^*1^*) =$ **true**.

On the other hand, **obviousSubset**$((01)^*, (\% + 0)(10)^*(\% + 1)) =$ **false**, even though $L((01)^*) \subseteq L((\% + 0)(10)^*(\% + 1))$.

Simplification Rules

We have three kinds of simplification rules, which may be applied on subtrees of regular expressions:

- structural rules,
- distributive rules,
- reduction rules.

Structural Rules

There are nine *structural rules*, which preserve the alphabet, closure complexity, size, number of concatenations and number of symbols of a regular expression:

(1) $(\alpha + \beta) + \gamma \to \alpha + (\beta + \gamma)$.

(2) $\alpha + (\beta + \gamma) \to (\alpha + \beta) + \gamma$.

(3) $\alpha(\beta\gamma) \to (\alpha\beta)\gamma$.

(4) $(\alpha\beta)\gamma \to \alpha(\beta\gamma)$.

(5) $\alpha + \beta \to \beta + \alpha$.

(6) $\alpha^*\alpha \to \alpha\alpha^*$.

(7) $\alpha\alpha^* \to \alpha^*\alpha$.

(8) $\alpha(\beta\alpha)^* \to (\alpha\beta)^*\alpha$.

(9) $(\alpha\beta)^*\alpha \to \alpha(\beta\alpha)^*$.

Distributive Rules

There are two *distributive rules*, which preserve the alphabet of a regular expression:

(1) $\alpha(\beta_1 + \beta_2) \to \alpha\beta_1 + \alpha\beta_2$.

(2) $(\alpha_1 + \alpha_2)\beta \to \alpha_1\beta + \alpha_2\beta$.

Reduction Rules

Finally, there are 26 *reduction rules*, some of which make use of a conservative approximation ***sub*** to subset testing.

When $\alpha \to \beta$ because of a reduction rule, we have that **alphabet** $\beta \subseteq$ **alphabet** α and β **simp** α, where **simp** is the well-founded relation on **Reg** defined below.

Most of the rules strictly decrease a regular expression's closure complexity *and* size. The exceptions are labeled "cc" (for when the closure complexity strictly decreases, but the size strictly increases), "concatenations" (for when the closure complexity and size are preserved, but the number of concatenations strictly decreases) or "symbols" (for when the closure complexity and size normally strictly decrease, but occasionally they and the number of concatenations stay they same, but the number of symbols strictly decreases).

Simplification Well-founded Relation

We define a relation **simp** on **Reg** by, for all $\alpha, \beta \in$ **Reg**, α **simp** β iff:

- **cc** $\alpha <_{cc}$ **cc** β; or
- **cc** $\alpha =$ **cc** β but **size** $\alpha <$ **size** β; or
- **cc** $\alpha =$ **cc** β and **size** $\alpha =$ **size** β, but **numConcats** $\alpha <$ **numConcats** β; or
- **cc** $\alpha =$ **cc** β, **size** $\alpha =$ **size** β and **numConcats** $\alpha =$ **numConcats** β, but **numSyms** $\alpha <$ **numSyms** β.

Proposition 3.3.29

*Suppose $\alpha, \beta, \beta' \in$ **Reg**, β' **simp** β, pat $\in$ **Path** is valid for α, and β is the subtree of α at position pat. Let α' be the result of replacing the subtree at position pat in α by β'. Then α' **simp** α.*

Proof. By induction on α. $\square$

Reduction Rules

(1) If *sub*(α, β), then $\alpha + \beta \to \beta$.

(2) $\alpha\beta_1 + \alpha\beta_2 \to \alpha(\beta_1 + \beta_2)$.

(3) $\alpha_1\beta + \alpha_2\beta \to (\alpha_1 + \alpha_2)\beta$.

(4) If **hasEmp** α and *sub*(α, β^*), then $\alpha\beta^* \to \beta^*$.

(5) If **hasEmp** β and *sub*(β, α^*), then $\alpha^*\beta \to \alpha^*$.

(6) If *sub*(α, β^*), then $(\alpha + \beta)^* \to \beta^*$.

(7) $(\alpha^* + \beta)^* \to (\alpha + \beta)^*$.

(8) (concatenations) If **hasEmp** α and **hasEmp** β, then $(\alpha\beta)^* \to (\alpha + \beta)^*$.

(9) (concatenations) If **hasEmp** α and **hasEmp** β, then $(\alpha\beta + \gamma)^* \to (\alpha + \beta + \gamma)^*$.

(10) If **hasEmp** α and *sub*(α, β^*), then $(\alpha\beta)^* \to \beta^*$.

(11) If **hasEmp** β and *sub*(β, α^*), then $(\alpha\beta)^* \to \alpha^*$.

Reduction Rules

(12) If **hasEmp** α and *sub*$(\alpha, (\beta + \gamma)^*)$, then $(\alpha\beta + \gamma)^* \to (\beta + \gamma)^*$.

(13) If **hasEmp** β and *sub*$(\beta, (\alpha + \gamma)^*)$, then $(\alpha\beta + \gamma)^* \to (\alpha + \gamma)^*$.

(14) (cc) If **not(hasEmp** α**)** and $\mathbf{cc}\,\alpha \cup \overline{\mathbf{cc}\,\beta} <_{cc} \overline{\overline{\mathbf{cc}\,\beta}}$, then $(\alpha\beta^*)^* \to \% + \alpha(\alpha + \beta)^*$.

(15) (cc) If **not(hasEmp** β**)** and $\overline{\mathbf{cc}\,\alpha} \cup \mathbf{cc}\,\beta <_{cc} \overline{\overline{\mathbf{cc}\,\alpha}}$, then $(\alpha^*\beta)^* \to \% + (\alpha + \beta)^*\beta$.

(16) (cc) If **not(hasEmp** α**)** or **not(hasEmp** γ**)**, and $\mathbf{cc}\,\alpha \cup \overline{\mathbf{cc}\,\beta} \cup \mathbf{cc}\,\gamma <_{cc} \overline{\overline{\mathbf{cc},\beta}}$, then $(\alpha\beta^*\gamma)^* \to \% + \alpha(\beta + \gamma\alpha)^*\gamma$.

(17) If *sub*$(\alpha\alpha^*, \beta)$, then $\alpha^* + \beta \to \% + \beta$.

(18) If **hasEmp** β and *sub*$(\alpha\alpha\alpha^*, \beta)$, then $\alpha^* + \beta \to \alpha + \beta$.

(19) (symbols) If $\alpha \notin \{\%, \$\}$ and *sub*(α^n, β), then $\alpha^{n+1}\alpha^* + \beta \to \alpha^n\alpha^* + \beta$.

Reduction Rules

(20) If $n \geq 2$, $l \geq 0$ and $2n - 1 < m_1 < \cdots < m_l$, then $(\alpha^n + \alpha^{n+1} + \cdots + \alpha^{2n-1} + \alpha^{m_1} + \cdots + \alpha^{m_l})^* \to \% + \alpha^n\alpha^*$.

(21) (symbols) If $\alpha \notin \{\%, \$\}$, then $\alpha + \alpha\beta \to \alpha(\% + \beta)$.

(22) (symbols) If $\alpha \notin \{\%, \$\}$, then $\alpha + \beta\alpha \to (\% + \beta)\alpha$.

(23) $\alpha^*(\% + \beta(\alpha + \beta)^*) \to (\alpha + \beta)^*$.

(24) $(\% + (\alpha + \beta)^*\alpha)\beta^* \to (\alpha + \beta)^*$.

(25) If $sub(\alpha, \beta^*)$ and $sub(\beta, \alpha)$, then $\% + \alpha\beta^* \to \beta^*$.

(26) If $sub(\beta, \alpha^*)$ and $sub(\alpha, \beta)$, then $\% + \alpha^*\beta \to \alpha^*$.

Local Simplification

Because the structural rules preserve the size and alphabet of regular expressions, if we start with a regular expression α, there are only finitely many regular expressions that we can transform α into using structural rules.

Local Simplification

Our local simplification algorithm/function is defined by well-founded recursion on **simp**. Given a regular expression α, it calls its main recursive function with the weak simplification, β, of α. The closure complexity, size, number of concatenations, and number of symbols of β are no bigger than those of α, and **alphabet** $\beta \subseteq$ **alphabet** α.

Local Simplification

Our local simplification algorithm/function is defined by well-founded recursion on **simp**. Given a regular expression α, it calls its main recursive function with the weak simplification, β, of α. The closure complexity, size, number of concatenations, and number of symbols of β are no bigger than those of α, and **alphabet** $\beta \subseteq$ **alphabet** α.

The recursive function works as follows, when called with a weakly simplified argument, α.

- It generates the set X of all regular expressions **weaklySimplify** γ, such that α can be reorganized using the structural rules into a regular expression β, which can be transformed by a single application of one of our reduction rules into γ.
- If X is empty, then it returns α.

Local Simplification

Our local simplification algorithm/function is defined by well-founded recursion on **simp**. Given a regular expression α, it calls its main recursive function with the weak simplification, β, of α. The closure complexity, size, number of concatenations, and number of symbols of β are no bigger than those of α, and **alphabet** $\beta \subseteq$ **alphabet** α.

The recursive function works as follows, when called with a weakly simplified argument, α.

- It generates the set X of all regular expressions **weaklySimplify** γ, such that α can be reorganized using the structural rules into a regular expression β, which can be transformed by a single application of one of our reduction rules into γ.
- If X is empty, then it returns α.
- Otherwise, it calls itself recursively on the simplest element, of X (ties broken by picking least element).

Local Simplification in Forlan

The Forlan module `Reg` provides the following functions relating to local simplification:

```
val locallySimplified     :
      (reg * reg -> bool) -> reg -> bool
val locallySimplify       :
      int option * (reg * reg -> bool) ->
      reg -> bool * reg
val locallySimplifyTrace :
      int option * (reg * reg -> bool) ->
      reg -> bool * reg
```

The argument of type `reg * reg -> bool` is a conservative approximation to subset testing. If the optional integer argument is `SOME` n, then at each recursive call of the principal function, only at most n structural reorganizations are considered. The returned boolean is `true` iff the returned regular expression is locally simplified.

Local Simplification in Forlan

```
- val locSimped =
=   Reg.locallySimplified Reg.obviousSubset;
val locSimped = fn : reg -> bool
- locSimped(Reg.fromString "(1 + 00*1)*00*");
val it = false : bool
- locSimped(Reg.fromString "(0 + 1)*0");
val it = true : bool
- fun locSimp nOpt =
=        Reg.locallySimplify(nOpt, Reg.obviousSubset);
val locSimp = fn : int option -> reg -> bool * reg
- locSimp
= NONE
= (Reg.fromString "% + 0*0(0 + 1)* + 1*1(0 + 1)*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
(0 + 1)*
val it = () : unit
```

Local Simplification in Forlan

```
- locSimp
= NONE
= (Reg.fromString "% + 1*0(0 + 1)* + 0*1(0 + 1)*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
(0 + 1)*
val it = () : unit
- locSimp NONE (Reg.fromString "(1 + 00*1)*00*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
(0 + 1)*0
val it = () : unit
```

Local Simplification in Forlan

```
- Reg.locallySimplifyTrace
= (NONE, Reg.obviousSubset)
= (Reg.fromString "0*(1 + 0*)*");
considered all 2 structural reorganizations of
0*(1 + 0*)*
0*(1 + 0*)* transformed by structural rule 5 at
position [2, 1] to 0*(0* + 1)* transformed by
reduction rule 7 at position [2] to 0*(0 + 1)*
considered all 2 structural reorganizations of
0*(0 + 1)*
0*(0 + 1)* transformed by reduction rule 4 at position
[] to (0 + 1)*
considered all 2 structural reorganizations of
(0 + 1)*
(0 + 1)* is locally simplified
val it = (true,-) : bool * reg
```

Local Simplification in Forlan

```
- 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.equal(Reg.weaklySimplify reg, reg);
val it = true : bool
- val (b', reg') = locSimp (SOME 10) reg;
val b' = false : bool
val reg' = - : reg
- Reg.output("", reg');
(0 + 2)*1(0 + 2 + 1(0 + 2)*1)*
val it = () : unit
```

Local Simplification in Forlan

```
- val (b'', reg'') = locSimp (SOME 1000) reg';
val b'' = true : bool
val reg'' = - : reg
- Reg.output("", reg'');
(0 + 2)*1(0 + 2 + 1(0 + 2)*1)*
val it = () : unit
```

Global Simplification

Given a regular expression α, global simplification consists of generating the set X of all regular expressions β that can formed from α by an arbitrary number of applications of weak simplification, the structural rules, reduction rules, and—in the case of the distributive variant—the distributive ones. The simplest element of X is then selected (ties broken by picking least element).

Global Simplification in Forlan

The Forlan module Reg provides the following functions relating to global simplification:

```
val globallySimplified    :
      bool * (reg * reg -> bool) -> reg -> bool
val globallySimplifyTrace :
      int option * bool * (reg * reg -> bool) ->
      reg -> bool * reg
val globallySimplify      :
      int option * bool * (reg * reg -> bool) ->
      reg -> bool * reg
```

The boolean argument specifies whether the distributive rules should be used. The argument of type `reg * reg -> bool` is a conservative approximation to subset testing. If the optional integer argument is `SOME` n, at most n candidates will be considered. The returned boolean is `true` iff all candidates were considered.

Global Simplification in Forlan

```
- fun globSimp(nOpt, dist) =
=         Reg.globallySimplify
=         (nOpt, dist, Reg.obviousSubset);
val globSimp = fn
  : int option * bool -> reg -> bool * reg
- fun globSimpTr(nOpt, dist) =
=         Reg.globallySimplifyTrace
=         (nOpt, dist, Reg.obviousSubset);
val globSimpTr = fn
  : int option * bool -> reg -> bool * reg
```

```
- globSimpTr (NONE, false) (Reg.fromString "(0*0)*");
considering candidates with explanations of length 0
simplest result now: (0*0)*
considering candidates with explanations of length 1
simplest result now: (0*0)* weakly simplifies to
(00*)*
simplest result now: (0*0)* transformed by reduction
rule 10 at position [] to 0*
considering candidates with explanations of length 2
considering candidates with explanations of length 3
considering candidates with explanations of length 4
considering candidates with explanations of length 5
considering candidates with explanations of length 6
search completed after considering 17 candidates with
maximum size 8
(0*0)* transformed by reduction rule 10 at position []
to 0* is globally simplified
val it = (true,-) : bool * reg
```

Global Simplification in Forlan

```
- locSimp NONE (Reg.fromString "(00*11*)*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
% + 00*1(% + (0 + 1)*1)
val it = () : unit
- globSimp (NONE, false) (Reg.fromString "(00*11*)*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
% + 0(0 + 1)*1
val it = () : unit
```

Global Simplification in Forlan

```
- globSimp
= (NONE, false)
= (Reg.fromString "% + 0*(0 + 1)");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
% + 0*(0 + 1)
val it = () : unit
- globSimp
= (NONE, true)
= (Reg.fromString "% + 0*(0 + 1)");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
0*(% + 1)
val it = () : unit
```

Global Simplification in Forlan

```
- globSimp
= (NONE, false)
= (Reg.fromString "(0(0(0 + 1))*)*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
% + 0(0(% + 0 + 1))*
val it = () : unit
- globSimp
= (NONE, true)
= (Reg.fromString "(0(0(0 + 1))*)*");
val it = (true,-) : bool * reg
- Reg.output("", #2 it);
% + 0(0(% + 1))*
val it = () : unit
```