Feasible Algorithms for Semantics — Employing Automata and Inference Systems

Dag Hovland

Dissertation for the degree of Philosophiae Doctor (PhD)

Department of Informatics
University of Bergen, Norway

December 2010

Scientific Environment

The research presented in this thesis was produced while the author was employed as a PhD student at the Department of Informatics, University of Bergen. The author was enrolled in the Research School in Information and Communication Technology, and was part of the research group for software development theory. The supervisor for this PhD work was Marc Bezem, and the co-supervisor was Khalid A. Mughal.

Acknowledgements

I am indebted to my supervisor Marc Bezem, without whom this thesis would not have been. I am grateful for his struggle to teach me the difference between what I know and what I believe. He has also been outstanding in taking care of practical matters. I thank my co-supervisor Khalid A. Mughal for asking the questions that led to much of this research, and for inspiration to go on. I must thank the whole group for software development theory for useful discussions and input, especially Andrew Polonsky and Federico Mancini.

Finally, I must thank Hanne, Øyvind, and Simien for staying with a busy PhD-student. I am also grateful for the wonderful coffee from the Zapatist cooperative Yachil Xojobal in Chiapas, Mexico.

Contents

List of Figures

List of Tables

1 Introduction

The main contribution of the thesis is three polynomial-time algorithms, each covering a semantic issue concerning a specific formal language. Hence the first part of the title, “Feasible Algorithms for Semantics”. One algorithm uses automata, and the two others use inference systems, hence the second part of the title, “Employing Automata and Inference Systems”.

The wider context of this thesis is that of theoretical computer science. Most parts fall within, or close to, the scope of *formal methods* for software engineering. The idea of formal methods is to support and even guide, the construction of information systems by fundamental knowledge developed in formal language and automata theory, process algebra, and type theory. Large parts of the thesis are also in the field of *formal languages*, or more specifically, *regular expressions*.

The thesis is based on three extended abstracts. Chapter 2 extends “The Inclusion Problem for Regular Expressions” [38], Chapter 3 extends “Regular Expressions with Numerical Constraints and Automata with Counters” [37], and Chapter 4 extends “A Type System for Usage of Software Components” [36].

All three chapters have a core in an algorithm for deciding a certain semantic issue concerning a formal language. In Chapters 2 and 3 the language is that of regular expressions with some extensions, and the semantics is the set of words denoted by a regular expression. The semantic issue in Chapter 2 is whether the sets denoted by two regular expressions are in an inclusion relation. In Chapter 3 we are interested in deciding whether a word is in the language of a a regular expression, and in using this to search a text for a word matching a regular expression. In the last chapter, the formal language is that of process expressions, and the (static) semantics are types of these expressions. The types give information about the bounds on the number of active component instances during execution of the process expression. The semantic issue is to infer the type of an expression. In all three chapters it has been a goal to find polynomial-time algorithms to decide the issues/questions. Chapters 2 and 4 have inference systems at the core of this algorithm, while Chapter 3 uses an automata-based approach.

Chapter 2 presents a polynomial-time algorithm for the inclusion problem for a subclass of the classical regular expressions. The input to the problem consists of two expressions, and the question is whether the language of the first expression is included in the language of the last expression. This problem was shown PSPACE-complete in Meyer & Stockmeyer [54], for arbitrary regular expressions. The classical algorithm involves the construction of a deterministic finite automaton for the second expression. A lower bound for this construction is given by the Myhill-Nerode theorem. Only for the subclass called the *1-unambiguous* regular expressions is there known a general polynomial-time construction of deterministic finite automata recognizing the same languages. The algorithm presented in Chapter 2 is not based on construction of automata, and can therefore be faster than the lower bound implied by the Myhill-Nerode theorem. The output of the algorithm is "Yes", "No" or "1-ambiguous". If it returns "Yes", the language of the first expression is included in that of the second expression. If it returns "No", this inclusion is not the case, and if it returns "1-ambiguous", the right-hand expression is not 1-unambiguous. The algorithm automatically discards irrelevant parts of the second expression. The irrelevant parts of the second expression might even be 1-ambiguous. For example, if r is a regular expression such that any deterministic finite automaton recognizing r is very large, the algorithm can still, in time independent of r, decide that the language of ab is included in that of $(a + r)b$. The algorithm is based on a syntax-directed inference system.

Chapter 3 is concerned with the class of regular expressions extended with operators for *unordered concatenation* and *numerical constraints*. Unordered concatenation is used in the Standard Generalized Markup Language (SGML), the precursor of XML. XML Schema uses a very limited form of unordered concatenation. Numerical constraints are an extension of regular expressions used in many applications, both in XML Schema and in the standard UNIX regular expressions. Regular expressions with unordered concatenation and numerical constraints denote the same languages as the classical regular expressions, but, in certain important cases, exponentially more succinct. The chapter contains a proof that the membership problem for regular expressions with unordered concatenation (without numerical constraints) is already NP-hard. The main contribution of Chapter 3 is a polynomial-time algorithm for the membership problem for regular expressions with numerical constraints and unordered concatenation, when restricted to a subclass called *strongly 1-unambiguous*.

The last chapter is concerned with counting the number of active component instances in a concurrent execution model allowing locking and freeing instances. The aim is to support component-based software engineering by modeling exclusive and inclusive usage of software components. Truong and Bezem describe in several papers abstract languages for component software with the aim to find bounds to the number of instances of components. Their languages include primitives for instantiating and deleting instances of components and operators for sequential, alternative and parallel composition, and a scope mechanism. The language is in Chapter 4 supplemented with primitives for *usage* of component instances. The main contribution is a type system which guarantees the safety of usage, in the following way: When a well-typed program executes a subexpression denoting usage of a component instance, it is guaranteed that an instance of that component is available. Type inference is shown to be polynomial. An alternative to using a type system is of course to run all possible executions of the program, and count the number of instances. However, the number of execution traces for any given program, is, in general, super-polynomial in the size of the program. Hence, this *brute force* approach is not feasible in this context.

The research questions that sparked all three papers came from the development of software for an e-learning environment at the Department of Informatics, University of Bergen. The questions concerning regular expressions were originally motivated by certain problems with the use of XML documents and XML Schema in the software. The work on the type system for component software (Chapter 4) elaborated the idea that one could support the developers in keeping track of the different components and Java packages used in a Java web application.

Since each chapter is based on a different extended abstract, each chapter can be read without reading the others. They are as self-contained as one can expect from a paper in the field. The only exception is some references in Chapter 3 to definitions in Chapter 2. Familiarity with the basic complexity classes, mathematical sets, operations on sets, and common notation for these is assumed and necessary to understand the content. Furthermore, knowledge of classical regular expressions is necessary to understand the introductions in Chapters 2 and 3.

"The Inclusion Problem for Regular Expressions" [38] won a prize for best student paper in LATA 2010. Implementations of the algorithms given in Chapters 2 and 4 are available from the the website `http://www.ii.uib.no/~dagh`. An implementation of the algorithm described in [37] is also available, but it has not been extended to include the unordered concatenation treated in Chapter 3. As a final remark, during the course of my PhD I also co-authored three papers that are not included in this thesis. The papers "Investigating the Limitations of Java Annotations for Input Validation" [48] and "The SHIP Validator: An Annotation-Based Content-Validation Framework for Java Applications" [49], are hard to reconcile with the other material, and have therefore been left out of the thesis. The paper "A Type System for Counting Instances of Software Components" [3] is presently under review by a journal. The latter paper has been the starting point and background for the material included in Chapter 4. I prefer to stick to the single-authored material, and therefore chose not to include [3].

2 The Inclusion Problem for Regular Expressions

This chapter presents a polynomial-time algorithm for the inclusion problem for a large class of regular expressions. The algorithm is not based on construction of finite automata, and can therefore be faster than the lower bound implied by the Myhill-Nerode theorem. The algorithm automatically discards irrelevant parts of the right-hand expression. The irrelevant parts of the right-hand expression might even be 1-ambiguous. For example, if r is a regular expression such that any DFA recognizing r is very large, the algorithm can still, in time independent of r, decide that the language of ab is included in that of $(a + r)b$. The algorithm is based on a syntax-directed inference system. It takes arbitrary regular expressions as input. If the 1-ambiguity of the right-hand expression becomes a problem, the algorithm will report this. Otherwise, it will decide the inclusion problem for the input.

2.1 Introduction

The inclusion problem for regular expressions was shown PSPACE-complete in Meyer & Stockmeyer [54]. The input to the problem consists of two expressions, the *left-hand* expression and the *right-hand* expression, respectively. The question is whether the language of the left-hand expression is included in the language of the right-hand expression. The classical algorithm starts with constructing non-deterministic finite automata (NFAs) for each of the expressions, then constructs a DFA from the NFA recognizing the language of the right-hand expression, and a DFA recognizing the complement of this language. Then an NFA recognizing the intersection of the language of the left-hand expression with the complement of the language of the right-hand expression is constructed. Finally, the algorithm checks that no final state is reachable in the latter NFA. The super-polynomial blowup occurs when constructing a DFA from the NFA recognizing the right-hand expression. A lower bound to this blowup is given by the Myhill-Nerode theorem [34, 57]. All the other steps, seen separately, are polynomial-time.

1-Unambiguous regular expressions were first used in SGML [1], and first formalized and studied by Brüggemann-Klein & Wood [10, 12]. The latter show a polynomial-time construction of DFAs from 1-unambiguous regular expressions. The classical algorithm can therefore be modified to solve the inclusion problem in polynomial time when the right-hand expression is 1-unambiguous. This chapter presents an alternative algorithm for inclusion of 1-unambiguous regular expressions. As in the modified classical algorithm, the left-hand expression can be an arbitrary regular expression. If the right-hand expression is 1-unambiguous, the algorithm is guaranteed to decide the inclusion problem, while if it is not 1-unambiguous (i.e., the expression is *1-ambiguous*), it might either decide the problem correctly, or report that the 1-ambiguity is a problem. An implementation of the algorithm is available from the website of the author. The algorithm can of course also be run twice to test whether the languages of two 1-unambiguous regular expressions are equal.

A consequence of the Myhill-Nerode theorem is that for many regular expressions, the minimal DFA recognizing this language, is of super-polynomial size. For example, there are no polynomial-size DFAs recognizing expressions of the form $(b+c)^*c(b+c)\cdots(b+c)$. An advantage of the algorithm presented in this chapter is that it only treats the parts of the right-hand expression which are necessary; it is therefore sufficient that these parts of the expression are 1-unambiguous. For some expressions, it can therefore be faster than the modified classical algorithm above. For example, the algorithm described in this chapter will (in polynomial time) decide that the language of ab is included in that of $(a+(b+c)^*c(b+c)\cdots(b+c))b$, and the sub-expression $(b+c)^*c(b+c)\cdots(b+c)$ will be discarded. The polynomial-time version of the classical algorithm cannot easily be modified to handle expressions like this, without adding complex and ad hoc pre-processing.

To summarize: Our algorithm always terminates in polynomial time. If the right-hand expression is 1-unambiguous, the algorithm will return a positive answer if and only if the expressions are in an inclusion relation, and a negative answer otherwise. If the right-hand expression is 1-ambiguous, three outcomes are possible: The algorithm might return a positive or negative answer, which is then guaranteed to be correct, or the algorithm might also decide that the 1-ambiguity of the right-hand expression is a problem, report this, and terminate.

Section 2.2 defines operations on regular expressions and properties of these. Section 2.3 describes the algorithm for inclusion, and Section 2.4 shows some important properties of the algorithm. The last section covers related work and a conclusion.

2.2 Regular Expressions

Fix an *alphabet* Σ of *letters*. Assume a, b, and c are members of Σ. $l, l_1, l_2, \ldots$ are used as variables for members of Σ.

Definition 2.2.1 (Regular Expressions)**.** *The* regular expressions *over the language* Σ *are denoted* R_Σ *and defined in the following inductive manner:*

$$R_\Sigma ::= R_\Sigma + R_\Sigma \mid R_\Sigma \cdot R_\Sigma \mid R_\Sigma^* \mid \Sigma \mid \epsilon$$

We use $r, r_1, r_2, \ldots$ *as variables for regular expressions. Concatenation is right-associative, such that, e.g.,* $r_1 \cdot r_2 \cdot r_3 = r_1 \cdot (r_2 \cdot r_3)$*. The sign for concatenation,* $\cdot$*, will often be omitted. A regular expression denoting the empty language is not included, as this is irrelevant to the results in this chapter. We denote the set of letters from* Σ *occurring in* r *by* $\mathsf{sym}(r)$*.*

The semantics of regular expressions is defined in terms of sets of words over the alphabet Σ. We lift concatenation of words to sets of words, such that if $L_1, L_2 \subseteq \Sigma^*$, then $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1,\ w_2 \in L_2\}$. ϵ denotes the *empty word* of zero length, such that for all $w \in \Sigma^*$, $\epsilon \cdot w = w \cdot \epsilon = w$. Integer exponents are short-hand for repeated concatenation of the same set, such that for a set L of words, e.g., $L^2 = L \cdot L$, and we define $L^0 = \{\epsilon\}$.

Definition 2.2.2 (Language of a Regular Expression)**.** *The* language *of a regular expression* r *is denoted* $\|r\|$ *and is defined by the following inductive rules:* $\|r_1 + r_2\| = \|r_1\| \cup \|r_2\|$, $\|r_1 \cdot r_2\| = \|r_1\| \cdot \|r_2\|$, $\|r^*\| = \bigcup_{0 \leq i} \|r\|^i$ *and for* $a \in \Sigma \cup \{\epsilon\}$, $\|a\| = \{a\}$*.*

All subexpressions of the forms $\epsilon \cdot r$, $\epsilon + \epsilon$ or ϵ^* can be simplified to r, ϵ, or ϵ respectively, in linear time, working bottom up. We will often tacitly assume there are no subexpressions of these forms. Furthermore, we use r^i as a short-hand for r concatenated with itself i times.

Definition 2.2.3 (Nullable Expressions)**.** *[29, 53] The* nullable *regular expressions are denoted $\mathfrak{N}_\Sigma$ and are defined inductively as follows:*

$$\mathfrak{N}_\Sigma ::= \mathfrak{N}_\Sigma + R_\Sigma \mid R_\Sigma + \mathfrak{N}_\Sigma \mid \mathfrak{N}_\Sigma \cdot \mathfrak{N}_\Sigma \mid {R_\Sigma}^* \mid \epsilon$$

We now show that the nullable expressions are exactly those denoting a language containing the empty word.

Lemma 2.2.4. *For all regular expressions $r \in R_\Sigma$, $\epsilon \in \|r\| \Leftrightarrow r \in \mathfrak{N}_\Sigma$.*

Proof. By induction on the regular expression r. The base cases $r = \epsilon$ and $r \in \Sigma$, and the induction case where r is of the form r_1^* are immediate from Definitions 2.2.2 and 2.2.3.

For the induction case where $r = r_1 + r_2$, we first treat the direction from left to right, that is, we assume $\epsilon \in \|r_1 + r_2\|$, and will prove that $r \in \mathfrak{N}_\Sigma$. From Definition 2.2.2 this implies that $\epsilon \in \|r_1\|$ or $\epsilon \in \|r_2\|$. Using the induction hypothesis we get that $r_1 \in \mathfrak{N}_\Sigma$ or $r_2 \in \mathfrak{N}_\Sigma$. From Definition 2.2.3 we then get that $r \in \mathfrak{N}_\Sigma$, as needed. For the other direction, assume $r_1 + r_2 \in \mathfrak{N}_\Sigma$. From Definition 2.2.3 we then get that $r_1 \in \mathfrak{N}_\Sigma$ or $r_2 \in \mathfrak{N}_\Sigma$. But then the induction hypothesis implies that $\epsilon \in \|r_1\|$ or $\epsilon \in \|r_2\|$. By using Definition 2.2.2 we then get that $\epsilon \in \|r\|$, as needed.

The induction case where $r = r_1 \cdot r_2$ can be shown by replacing "$+$" with "$\cdot$" and "or" with "and" in the previous paragraph. □

Intuitively, the first-set of a regular expression is the set of letters that can occur first in a word in the language. An inductive definition of the first-set is given in Table 2.1. Similar definitions have been given by many others, e.g., Glushkov [29] and Yamada & McNaughton [53].

Lemma 2.2.5 (first)**.** *[29, 53] For any regular expression r,* first$(r) = \{l \in \Sigma \mid \exists w : lw \in \|r\|\}$ *and* first(r) *can be calculated in time linear in r.*

The proof is by an easy induction on r, using Table 2.1 and Definition 2.2.2. The followLast-set of a regular expression is the set of letters which can follow a word in the language.

$\mathsf{first}(\epsilon) = \varnothing$,	$r \in \Sigma \Rightarrow \mathsf{first}(r) = \{r\}$
$r = r_1 + r_2 \Rightarrow \mathsf{first}(r) = \mathsf{first}(r_1) \cup \mathsf{first}(r_2)$	
$r = r_1 \cdot r_2 \wedge r_1 \in \mathfrak{N}_\Sigma \Rightarrow \mathsf{first}(r) = \mathsf{first}(r_1) \cup \mathsf{first}(r_2)$	
$r = r_1 \cdot r_2 \wedge r_1 \notin \mathfrak{N}_\Sigma \Rightarrow \mathsf{first}(r) = \mathsf{first}(r_1)$	
$r = r_1^* \Rightarrow \mathsf{first}(r) = \mathsf{first}(r_1)$	

Table 2.1: The first-set of a regular expression

Definition 2.2.6 (followLast). *[10]*

$$\mathsf{followLast}(r) = \{l \in \mathsf{sym}(r) \mid \exists u, v \in \mathsf{sym}(r)^* : (u \in \|r\| \wedge ulv \in \|r\|)\}$$

To limit the number of rules in the inference system explained in Section 2.3, we will put regular expressions on *header-form*.

Definition 2.2.7 (Header-form). *A regular expression is in header-form if it is of the form ϵ, $l \cdot r_1$, $(r_1 + r_2) \cdot r_3$ or $r_1^* \cdot r_2$, where $l \in \Sigma$ and $r_1, r_2, r_3 \in R_\Sigma$.*

A regular expression can in linear time be put in header-form without changing the denoted language by applying the mapping hdf. We need the auxiliary mapping header, which maps a pair of regular expressions to a single regular expression. It is defined by the following inductive rules:

$$\mathsf{header}(\epsilon, r) = r$$

$$\mathsf{header}(r_1, r_2) = \begin{cases} \text{(write } r_1 \text{ as } r'_1 \cdots r'_n \text{ for } n \geq 1\text{, where } r'_n \text{ is not a concatenation)} \\ r_1 \cdot r_2 & \text{if } n = 1 \\ \mathsf{header}(r'_1, r_2) & \text{if } n = 2, r'_2 = \epsilon \\ \mathsf{header}(r'_1, r'_2 \cdots r'_{n-1} \cdot r_2) & \text{if } n > 2, r'_n = \epsilon \\ \mathsf{header}(r'_1, r'_2 \cdots r'_n \cdot r_2) & \text{if } n \geq 2, r'_n \neq \epsilon \end{cases}$$

We can now define $\mathsf{hdf}(r) = \mathsf{header}(r, \epsilon)$. We summarize the basic properties of these mappings in the following lemma:

Lemma 2.2.8. *For any regular expression r:*

1. $\mathsf{hdf}(r)$ *is in header-form,*

2. $\|\mathsf{hdf}(r)\| = \|r\|$,

3. $\exists n \geq 0, r_1, \ldots, r_n \in R_\Sigma - \{\epsilon\} : \mathsf{hdf}(r) = r_1 \cdots r_n \cdot \epsilon$.

4. $\mathsf{hdf}(\mathsf{hdf}(r)) = \mathsf{hdf}(r)$.

Proof. 1. We first show that for any $r_1 \in R_\Sigma - \{\epsilon\}$ and any $r_2 \in R_\Sigma$, $\mathsf{header}(r_1, r_2)$ is in header form by induction on r_1. If r_1 is not a concatenation, then, since, $r_1 \neq \epsilon$, we get directly from the definitions that $\mathsf{header}(r_1, r_2) = r_1 \cdot r_2$ is in header form. Otherwise, if r_1 is of the form $r'_1 \cdot r'_2$, we get that the result is a new call to header where the first argument is r'_1. Since we have assumed that ϵ prefixes are removed, $r'_1 \neq \epsilon$. Thus we can apply the induction hypothesis to r'_1 and get that the result is in header form.

Now, by definition $\mathsf{hdf}(r) = \mathsf{header}(r, \epsilon)$. If $r = \epsilon$, $\mathsf{hdf}(r) = \epsilon$, which is in header form. Otherwise, we can use the result above to get that $\mathsf{header}(r, \epsilon)$ is in header form.

2. Since $\mathsf{hdf}(r) = \mathsf{header}(r, \epsilon)$ we only need to show that for any r_1, r_2 $\|\mathsf{header}(r_1, r_2)\| = \|r_1 r_2\|$. The latter follows almost directly from associativity of concatenation and neutrality of concatenation with ϵ.

3. If $r = \epsilon$, $\mathsf{hdf}(r) = \epsilon$ so we are done. Otherwise, since $\mathsf{hdf}(r) = \mathsf{header}(r, \epsilon)$, it is sufficient to prove by induction on $r \in R_\Sigma - \{\epsilon\}$ that for any $r'_1, \ldots, r'_m \in R_\Sigma - \{\epsilon\}$ there are $r_1, \ldots, r_n \in R_\Sigma - \{\epsilon\}$ such that

$$\mathsf{header}(r, r'_1 \cdots r'_m \cdot \epsilon) = r_1 \cdots r_n \cdot \epsilon$$

If r is not a concatenation, then we get

$$\mathsf{header}(r, r'_1 \cdots r'_m \cdot \epsilon) = r \cdot r'_1 \cdots r'_m \cdot \epsilon$$

Otherwise, if $r = r''_1 \cdot r''_2$ is a concatenation, we get a new call to header, where the first argument is r''_1 and the second argument is of the form required by the induction hypothesis. (Recall that ϵ prefixes have been removed). Therefore we can apply the induction hypothesis to get the result.

4. From the previous item, there are $r_1, \ldots, r_n \in R_\Sigma - \{\epsilon\}$ such that $\mathsf{hdf}(r) = r_1 \cdots r_n \cdot \epsilon$. If $n = 0$, the lemma holds since $\mathsf{hdf}(\epsilon) = \mathsf{header}(\epsilon, \epsilon) = \epsilon$. Otherwise, we get from the definitions of hdf and

header

$$\mathsf{hdf}(\mathsf{hdf}(r)) =$$

$$\mathsf{hdf}(r_1 \cdots r_n \cdot \epsilon) =$$
$$\mathsf{header}(r_1 \cdots r_n \cdot \epsilon, \epsilon) =$$
$$\mathsf{header}(r_1, r_2 \cdots r_n \cdot \epsilon) =$$
$$r_1 \cdots r_n \cdot \epsilon =$$

$$\mathsf{hdf}(r)$$

□

2.2.1 Term Trees and Positions

Given a regular expression r, we follow Terese [4] and define the term tree of r as the tree where the root is labeled with the main operator (choice, concatenation, or star) and the subtrees are the term trees of the subexpression(s). If $a \in \Sigma \cup \{\epsilon\}$ the term tree is a single root-node with a as label.

We use $\langle n_1, \ldots, n_k \rangle$, a possibly empty sequence of natural numbers, to denote a position in a term tree. We let p, q, including subscripted variants, be variables for such possibly empty sequences of natural numbers. The position of the root is $\langle\rangle$. If $r = r_1 \cdot r_2$ or $r = r_1 + r_2$, and $n_1 \in \{1, 2\}$, the position $\langle n_1, \ldots, n_k \rangle$ in r is the position $\langle n_2, \ldots, n_k \rangle$ in the subtree of child n_1, that is, in the term tree of r_{n_1}. If $r = {r_1}^*$, the position $\langle 1, n_1, \ldots, n_k \rangle$ in r is the position $\langle n_1, \ldots, n_k \rangle$ in the term tree of r_1. Let $\mathsf{pos}(r)$ be the set of positions in r.

For two positions $p = \langle m_1, \ldots, m_k \rangle$ and $q = \langle n_1, \ldots, n_l \rangle$, the notation $p \odot q$ will be used for the concatenated position $\langle m_1, \ldots, m_k, n_1, \ldots, n_l \rangle$. We will also use this notation for lists of positions, so if $p, p_1, \ldots, p_n$ are positions, then $p \odot (p_1 \cdot \cdots \cdot p_n) = (p \odot p_1) \cdot \cdots \cdot (p \odot p_n)$. Further, we use the notation for concatenating a position with each elements of a set consisting of lists of positions, such that if p is a position, and S is a set of lists of positions, then $p \odot S = \{p \odot q \mid q \in S\}$.

Below we will encounter regular expressions whose alphabet are sets of positions. Concatenating a position with such an expression is defined by concatenating the position with all the positions occurring in the expression. Note that the language of such a regular expression is a set of lists of positions. Hence, for p a position, $r_1 \in R_\Sigma$, and $r \in R_{\mathsf{pos}(r_1)}$, $\|p \odot r\| = p \odot \|r\|$.

Whenever concatenating with a position of length one, we will often omit the sign $\odot$ and abbreviate, such that for example $p1 = p \odot \langle 1 \rangle$, $2S = \langle 2 \rangle \odot S$, $ir = \langle i \rangle \odot r$, etc.

For a position p in r we will denote the subexpression rooted at this position by $r[p]$.Note that $r[\langle\rangle] = r$. We also set $r[\epsilon] = \epsilon$. Furthermore, given $p_1, \ldots, p_n$ in $\mathsf{pos}(r) \cup \{\epsilon\}$, put $r[p_1 \cdot \cdots \cdot p_n] = r[p_1] \cdot \cdots \cdot r[p_n]$. Lastly, we lift $r[]$ to sets of string, such that if $S \subseteq \mathsf{pos}(r)^*$, then $r[S] = \{r[w] \mid w \in S\}$.

Note that for $r \in R_\Sigma$, $p \in \mathsf{pos}(r)$, and $q \in \mathsf{pos}(r[p])$, we have $r[p \odot q] = r[p][q]$. This can be shown by induction on $r[p]$ (see, e.g., Terese [4]). For example in the case of $r[p] = r_1 \cdot r_2$, we have that q is a position in either r_1 or r_2. Assume it is in r_1, then $q = 1q'$ for some $q' \in \mathsf{pos}(r_1)$. As $r[p][\langle 1 \rangle] = r_1 = r[p1]$ we get that $r[p][1q'] = r[p1][q']$, and by the induction hypothesis $r[p1][q'] = r[p1 \odot q']$.

The concept of *marked expressions* will be important in this chapter. It was first used in a similar context by Brüggemann-Klein & Wood [12]. The intuition is that the marked expression is the expression where every instance of any symbol from Σ is substituted by its position in the expression.

Definition 2.2.9 (Marked Expressions). *If $r \in R_\Sigma$ is a regular expression, $\mu(r) \in R_{\mathsf{pos}(r)}$ is the marked expression, defined in the following inductive manner:*

- $\mu(\epsilon) = \epsilon$
- *for $l \in \Sigma$, $\mu(l) = \langle\rangle$*
- $\mu(r_1 + r_2) = 1\mu(r_1) + 2\mu(r_2)$
- $\mu(r_1 \cdot r_2) = 1\mu(r_1) \cdot 2\mu(r_2)$
- $\mu(r_1^*) = (1\mu(r_1))^*$

Note that, e.g., $\mu(b) = \mu(a) = \langle\rangle$, which shows that marking is not injective. Furthermore $\|\mu(r_1 \cdot r_2)\| = 1\|\mu(r_1)\| \cdot 2\|\mu(r_2)\|$, $\|\mu(r_1 + r_2)\| = 1\|\mu(r_1)\| \cup 2\|\mu(r_2)\|$, and $\|\mu(r^*)\| = 1\|\mu(r)^*\|$. The following lemma will often be used tacitly.

Lemma 2.2.10. *For any regular expression r,*

1. $\|r\| = r[\|\mu(r)\|]$

2. *For any* $p \in \mathsf{sym}(\mu(r))$*,* $\mu(r)[p] = p$*.*

3. *For any* $p \in \mathsf{pos}(r)$*,* $r[p] \in \Sigma$ *iff* $p \in \mathsf{sym}(\mu(r))$*.*

Proof.

1. By induction on r. The base cases $r = \epsilon$ and $r \in \Sigma$ are immediate from Definitions 2.2.2 and 2.2.9 and the definition of $r[]$.

For the inductive case where $r = r_1 + r_2$, by Definition 2.2.9, $\mu(r_1 + r_2) = 1\mu(r_1) + 2\mu(r_2)$. Applying Definition 2.2.2 to the latter we get $\|\mu(r_1 + r_2)\| = \|1\mu(r_1)\| \cup \|2\mu(r_2)\|$. Hence, by definition of concatenating a position with a regular expression, $\|\mu(r_1 + r_2)\| = 1\|\mu(r_1)\| \cup 2\|\mu(r_2)\|$. By applying distributivity of $r[]$ over concatenation we get $r[\|\mu(r_1 + r_2)\|] = r[1\|\mu(r_1)\|] \cup r[2\|\mu(r_2)\|]$. Note now that for any $i \in \{1,2\}$ and any $q \in \mathsf{pos}(r_i)$, we have $r[iq] = r[\langle i \rangle][q] = r_i[q]$. Applying this we get $r[\|\mu(r_1 + r_2)\|] = r_1[\|\mu(r_1)\|] \cup r_2[\|\mu(r_2)\|]$. By applying the induction hypothesis we get $r[\|\mu(r_1 + r_2)\|] = \|r_1\| \cup \|r_2\|$. Hence, by Definition 2.2.2, $r[\|\mu(r_1 + r_2)\|] = \|r_1 + r_2\|$.

The inductive case where $r = r_1 \cdot r_2$, can be shown by replacing "$+$" and "$\cup$" with "$\cdot$" in the previous paragraph.

For the inductive case where $r = r_1^*$, by Definition 2.2.9, $\mu(r_1^*) = (1\mu(r_1))^*$. Applying Definition 2.2.2 to the latter we get $\|\mu(r_1^*)\| = \bigcup_{0 \leq i} \|1\mu(r_1)\|^i$. Hence, by definition of concatenating a position with a regular expression, $\|\mu(r_1^*)\| = \bigcup_{0 \leq i}(1\|\mu(r_1)\|)^i$. By applying distributivity of $r[]$ over union and concatenation we get $r[\|\mu(r_1^*)\|] = \bigcup_{0 \leq i}(r[1\|\mu(r_1)\|])^i$. Note now that for any $q \in \mathsf{pos}(r_1)$, we have $r[1q] = r[\langle 1 \rangle][q] = r_1[q]$. Applying this we get $r[\|\mu(r_1^*)\|] = \bigcup_{0 \leq i}(r_1[\|\mu(r_1)\|])^i$. By applying the induction hypothesis we get $r[\|\mu(r_1^*)\|] = \bigcup_{0 \leq i} \|r_1\|^i$. Hence, by Definition 2.2.2, $r[\|\mu(r_1^*)\|] = \|r_1^*\|$.

2. By induction on r. The base case $r = \epsilon$ holds vacuously. The base case $r \in \Sigma$ holds immediately from Definition 2.2.9. For the inductive cases where $r = r_1 \cdot r_2$ or $r = r_1 + r_2$, we assume some $p \in \mathsf{sym}(\mu(r))$ and proceed to show that $\mu(r)[p] = p$. By Definition 2.2.9, $\mathsf{sym}(\mu(r)) = 1\mathsf{sym}(\mu(r_1)) \cup 2\mathsf{sym}(\mu(r_2))$. Hence, there is $i \in \{1,2\}$ and $p' \in \mathsf{sym}(\mu(r_i))$ such that $p = ip'$. By the induction hypothesis for r_i, $\mu(r_i)[p'] = p'$, hence $(i\mu(r_i))[p'] = p$. Since

$\mu(r)[p] = (1\mu(r_1) \cdot 2\mu(r_2))[ip'] = (i\mu(r_i))[p']$ we get $\mu(r)[p] = p$. The inductive case where $r = r_1^*$ is similar to the previous case, but easier.

3. By induction on r. The base cases, and the cases where $p = \langle\rangle$, hold directly from Definition 2.2.9.

For the inductive cases where $r = r_1 \cdot r_2$ or $r = r_1 + r_2$, and $p \neq \langle\rangle$, there is $i \in \{1,2\}$ and $p' \in \mathsf{pos}(r_i)$ such that $p = ip'$. We have $r[p] = r_i[p']$ and that $p \in \mathsf{sym}(\mu(r))$ iff $p' \in \mathsf{sym}(\mu(r_i))$. Hence, the lemma holds by applying the induction hypothesis for r_i.

For the inductive case where $r = r_1^*$ and $p \neq \langle\rangle$, we can use the same argument as in the previous case, except that i is set to 1.

□

2.2.2 1-Unambiguous Regular Expressions

Definition 2.2.11 (Star Normal Form). *[10, 12]: A regular expression is in* star normal form *iff for all subexpressions r^*: $r \notin \mathfrak{N}_\Sigma$ and $\mathsf{first}(\mu(r)) \cap \mathsf{followLast}(\mu(r)) = \varnothing$.*

Brüggemann-Klein & Wood described also in [10, 12] a linear time algorithm mapping a regular expression to an equivalent expression in star normal form. We will therefore often tacitly assume that all regular expressions are in star normal form.

It is almost immediate that hdf preserves star normal form, as starred subexpressions are not altered.

Definition 2.2.12. *[10, 12] A regular expression r is 1-unambiguous if for any two $upv, uqw \in \|\mu(r)\|$, where $p, q \in \mathsf{sym}(\mu(r))$ (i.e., $r[p], r[q] \in \Sigma$) and $u, v, w \in \mathsf{sym}(\mu(r))^*$ such that $r[p] = r[q]$, we have $p = q$.*

Examples of 1-unambiguous regular expressions are aa^* and $b^*a(b^*a)^*$, while $(\epsilon + a)a$ and $(a + b)^*a$ are not 1-unambiguous. The languages denoted by 1-unambiguous regular expressions will be called *1-unambiguous regular languages*. An expression which is not 1-unambiguous is called 1-ambiguous. Brüggemann-Klein & Wood [12] showed that there exist regular languages that are not 1-unambiguous regular languages, e.g. $\|(a + b)^*(ac + bd)\|$. However, the reverse of $(a + b)^*(ac + bd)$, namely $(ca + db)(a + b)^*$ is 1-unambiguous. There are of course also expressions like $(a + b)^*(ac + bd)(c + d)^*$, which denotes a 1-ambiguous language, read both backwards and forwards.

Brüggemann-Klein & Wood characterized the 1-unambiguous regular expressions in [12, Lemma 3.2]. The latter lemma implies that all subexpressions of a 1-unambiguous regular expression (in star normal form) are 1-unambiguous. Another important consequence is that if r_1 and r_2 are 1-unambiguous, and $\mathsf{first}(r_1) \cap \mathsf{first}(r_2) = \varnothing$, then $r_1 + r_2$ is 1-unambiguous. Lastly, $r_1 \cdot r_2$ is 1-unambiguous if r_1 and r_2 are 1-unambiguous, $r_1 \in \mathfrak{N}_\Sigma \Rightarrow \mathsf{first}(r_1) \cap \mathsf{first}(r_2) = \varnothing$, and $\mathsf{followLast}(r_1) \cap \mathsf{first}(r_2) = \varnothing$. The latter three facts will be used several times.

Lemma 2.2.13. *For a 1-unambiguous regular expression r, $\mathsf{hdf}(r)$ is also 1-unambiguous.*

Proof. First we prove by induction on r_1, where $r_1 \neq \epsilon$, that if $r_1 \cdot r_2$ is 1-unambiguous, then $\mathsf{header}(r_1, r_2)$ is 1-unambiguous. The cases where $r_1 \neq \epsilon$ is not a concatenation hold immediately, as $\mathsf{header}(r_1, r_2) = r_1 \cdot r_2$. For the remaining cases, there are $n \geq 1$ and $r'_1, \ldots, r'_n \in R_\Sigma - \{\epsilon\}$ such that either $n \geq 2$, $r_1 = r'_1 \cdots r'_n$, and r'_n is not a concatenation, or $r_1 = r'_1 \cdots r'_n \cdot \epsilon$. We first show that $r'_1 \cdots r'_n \cdot r_2$ is 1-unambiguous. Let u, p, q, v, w as in Definition 2.2.12 such that $u \cdot p \cdot v, u \cdot q \cdot w \in \|\mu(r'_1 \cdots r'_n \cdot r_2)\|$ and $(r'_1 \cdots r'_n \cdot r_2)[p] = (r'_1 \cdots r'_n \cdot r_2)[q] \in \mathsf{sym}(\mu(r'_1 \cdots r'_n \cdot r_2))$. Note now that the u, p, q, v, w can easily be modified to get u', p', q', v', w' such that $p = q \Leftrightarrow p' = q'$, $u' \cdot p' \cdot v', u' \cdot q' \cdot w' \in \|\mu(r_1 \cdot r_2)\|$, and $(r_1 \cdot r_2)[p'] = (r_1 \cdot r_2)[q']$. But since $r_1 \cdot r_2$ 1-unambiguous by assumption, we get from Definition 2.2.12 that $p' = q'$. Therefore $p = q$. Thus $r'_1 \cdots r'_n \cdot r_2$ is 1-unambiguous. By the induction hypothesis on r'_1 this implies that $\mathsf{header}(r'_1, r'_2 \cdots r'_n \cdot r_2)$ is 1-unambiguous. Hence, $\mathsf{header}(r_1, r_2) = \mathsf{header}(r'_1, r'_2 \cdots r'_n \cdot r_2)$ is 1-unambiguous.

Secondly, we prove that if $r \neq \epsilon$ and r is 1-unambiguous, then also $r \cdot \epsilon$ is 1-unambiguous. Take any u, p, q, v, w as in Definition 2.2.12 for $r \cdot \epsilon$ such that $u \cdot p \cdot v, u \cdot q \cdot w \in \|\mu(r \cdot \epsilon)\|$ and $(r \cdot \epsilon)[p] = (r \cdot \epsilon)[q]$. It is easy to see that there are u', p', q', v', w' such that $u = 1u'$, $p = 1p'$, $q = 1q'$, $v = 1v'$, and $w = 1w'$. This implies that $u' \cdot p' \cdot v', u' \cdot q' \cdot w' \in \|\mu(r)\|$ and $r[p'] = r[q']$. We can use Definition 2.2.12 for r to get $p' = q'$. Therefore $p = q$ and $r \cdot \epsilon$ is 1-unambiguous.

Finally, if $r = \epsilon$, $\mathsf{hdf}(r) = \epsilon$ is 1-unambiguous. Otherwise, if $r \neq \epsilon$, we have by the previous paragraph that $r \cdot \epsilon$ is 1-unambiguous. By the paragraph above, this implies that $\mathsf{header}(r, \epsilon)$ is 1-unambiguous. Since $\mathsf{hdf}(r) = \mathsf{header}(r, \epsilon)$ we get that $\mathsf{hdf}(r)$ is 1-unambiguous. $\square$

1-unambiguity is different from, though related with, *unambiguity*, as used to classify grammars in language theory, and studied for regular expressions by Book et al. [6]. From [6]: "A regular expression is called unambiguous if every tape in the event can be generated from the expression in one way only"[1] It is not hard to see that the class of 1-unambiguous regular expressions is included in the class of unambiguous regular expressions.

Proof. We first prove by induction on r that if r is in star normal form, and r is ambiguous, then there are $u, u' \in \|\mu(r)\|$ such that $u \neq u'$ but $r[u] = r[u']$. The base cases hold vacuously.

For the induction case where $r = r_1 + r_2$, there must be a word w which is either generated in two ways by r_1 or by r_2, or which is generated by both r_1 and r_2. In the former case, it suffices to use the induction hypothesis for r_1 or r_2. In the latter case, we get $u \in \|\mu(r_1)\|$ and $u' \in \|\mu(r_2)\|$ such that $r_1[u] = w = r_2[u']$. Hence, $r[1u] = w = r[2u']$ and $1u \neq 2u'$.

For the induction case where $r = r_1 \cdot r_2$, let w be a witness that r is ambiguous. There must be $w_1 \in \|r_1\|$ and $w_2 \in \|r_2\|$ such that $w = w_1 \cdot w_2$. For $i \in \{1, 2\}$, if w_i can be generated in two ways by r_i, we get the result by the induction hypothesis for r_i. Otherwise, we get $w'_1 \in \|r_1\|$ and $w'_2 \in \|r_2\|$ such that $w_1 \neq w'_1$, $w_2 \neq w'_2$ and $w = w'_1 \cdot w'_2$. Furthermore, there are $u_1, u'_1 \in \|\mu(r_1)\|$ and $u_2, u'_2 \in \|\mu(r_2)\|$ such that $w_1 = r_1[u_1]$, $w'_1 = r_1[u'_1]$, $w_2 = r_2[u_2]$, and $w'_2 = r_2[u'_2]$. Hence, $r[1u_1 \cdot 2u_2] = r[1u'_1 \cdot 2u'_2]$ and $1u_1 \cdot 2u_2 \neq 1u'_1 \cdot 2u'_2$.

For the induction case where $r = r_1^*$, let w be a witness that r_1^* is ambiguous. Note that ϵ can only be generated in one way, since r is in star normal form and $r_1 \notin \mathfrak{N}_\Sigma$. Hence, $w \neq \epsilon$, and there must be $w_1, \ldots, w_n \in \|r_1\|$ such that $w = w_1 \cdots w_n$. If one of the w_i's is generated in two ways by r_1 we get the result from the induction hypothesis for r_1. Otherwise, there must be $w'_1, \ldots, w'_m \in \|r_1\|$ different from $w_1, \ldots, w_n$ such that $w = w'_1 \cdots w'_m$. Then there is i such that $w_i \neq w'_i$, but for $0 < j < i$, $w_j = w'_j$. Hence, there is $l \in \Sigma$ and $w' \in \Sigma^*$ such that either $w_i = w'_i l w'$ or $w_i l w' = w'_i$. The cases are symmetric, so we treat only $w_i = w'_i l w'$. Then there are $u_1, \ldots, u_n, u'_1, \ldots, u'_m \in \|\mu(r_1)\|$ such that $\forall j \in \{1, \ldots, n\} : w_j = r_1[u_j]$ and $\forall j \in \{1, \ldots, m\} : w'_j = r_1[u'_j]$. Hence, $r[1(u_1 \cdots u_n)] = w = r[1(u'_1 \cdots u'_m)]$. There are also

[1]In modern language, "tape" is "word" and "event" is "language".

$p, p' \in \mathsf{sym}(\mu(r_1))$ and $u', u'', u''' \in \mathsf{sym}(\mu(r_1))^*$ such that $u_i = u'pu''$, $u'_{i+1} = p'u'''$, $l = r_1[p] = r_1[p']$, $w' = r_1[u'']$, and $w'_i = r_1[u']$. Since $p'u''' \in \|\mu(r_1)\|$, $p' \in \mathsf{first}(\mu(r_1))$. If $u'_i \neq u'$ we get immediately $u'_1 \cdots u'_m \neq u_1 \cdots u_n$ and we are done. Otherwise, since $u'_i, u'_i pu'' \in \|\mu(r_1)\|$, we get $p \in \mathsf{followLast}(\mu(r_1))$. Since r is in star normal form $p \neq p'$, hence $u'_1 \cdots u'_m \neq u_1 \cdots u_n$.

We now proceed to show that the class of 1-unambiguous regular expressions is included in the class of unambiguous regular expressions. We prove the contra-positive statement. We assume that r is ambiguous and proceed to show that r is 1-ambiguous. If r is not in star normal form, we get that r is 1-ambiguous from Definitions 2.2.12 and 2.2.11. Otherwise, we get from the arguments above $u, u' \in \|\mu(r)\|$ such that $u \neq u'$ but $r[u] = r[u']$. Let u_1 be the longest common prefix of u and u'. Then there are p, q, v, w such that $u_1pv, u_1qw \in \|\mu(r)\|$, $p \neq q$ and $r[p] = r[q]$. By Definition 2.2.12 this means r is 1-ambiguous. $\square$

The inclusion is strict, as for example the expression $(a + b)^*a$ is both unambiguous and 1-ambiguous. See also [10, 12] for comparisons of unambiguity and 1-unambiguity.

2.3 Rules for Inclusion

The algorithm is based on an inference system described in Tables 2.2 and 2.3, inductively defining a binary relation $\sqsubseteq$ between regular expressions. The core of the algorithm is a goal-directed, depth-first search using this inference system. We will show later that a pair of regular expressions is in the relation $\sqsubseteq$ if and only if their languages are in the inclusion relation.

We will say that $r_1 \sqsubseteq r_2$ *is sound*, if $\|r_1\| \subseteq \|r_2\|$. Each rule consists of a horizontal line with a conclusion below it, and zero, one, or two premises above the line. All rules but one also have *side-conditions* in square brackets. We only allow rule instances where the side-conditions are satisfied. This means that *matching* the conclusion of a rule implies satisfying the side-conditions. Note that (StarChoice1) and (LetterChoice) each have only one premise.

Figure 2.1 describes the algorithm for deciding inclusion of regular expressions. The algorithm takes a pair of regular expressions as

```
Input: Two regular expressions $r_1$ and $r_2$
Output: "Yes", "No" or "1-ambiguous"
Initialize stack T and set S as empty ;
push $(\mathsf{hdf}(r_1), \mathsf{hdf}(r_2))$ on T;
while T not empty do
    pop $(r_3, r_4)$ from T;
    if $(r_3, r_4) \notin \mathsf{S}$ then
        if $\mathsf{first}(r_3) \not\subseteq \mathsf{first}(r_4)$ or $r_3 \in \mathfrak{N}_\Sigma \wedge r_4 \notin \mathfrak{N}_\Sigma$ or $r_4 = \epsilon \wedge r_3 \neq \epsilon$ then
            return "No";
        end
        if $r_3 \sqsubseteq r_4$ matches conclusion of more than one rule instance
        then
            return "1-ambiguous";
        end
        add $(r_3, r_4)$ to S;
        for all premises $r_5 \sqsubseteq r_6$ of the rule instance where $r_3 \sqsubseteq r_4$
        matches the conclusion do
            push $(\mathsf{hdf}(r_5), \mathsf{hdf}(r_6))$ on T;
        end
    end
end
return "Yes";
```

Figure 2.1: Algorithm for inclusion of regular expressions

input, and if it returns "Yes" they are in an inclusion relation, if it returns "No" they are not, and if it returns "1-ambiguous", the right-hand expression is 1-ambiguous. The stack T is used for a depth-first search, while the set S keeps track of already treated pairs of regular expressions. Both S and T consist of pairs of regular expressions.

Figures 2.2, 2.3, and 2.4 show examples of how to use the inference rules. The example noted in the introduction, deciding whether $\|ab\| \subseteq \|(a + (b + c)^*c(b + c) \cdots (b + c))b\|$ is shown in Fig. 2.4. Note that branches end either in an instance of the rule (Axm), usage of the store of already treated relations, or a failure. In addition to correctness of the algorithm, termination is of course of paramount importance. It is natural to ask how the algorithm possibly can terminate, when the rules (LetterStar), (LeftStar), and (StarChoice2) have more complex premises than conclusions. This will be answered in the next section.

Table 2.2: The rules for the relation $\sqsubseteq$ (Continued in Table 2.3).

(Axm)
$$\frac{}{\epsilon \sqsubseteq r} \quad [r \in \mathfrak{N}_\Sigma]$$

(Letter)
$$\frac{r_1 \sqsubseteq r_2}{l \cdot r_1 \sqsubseteq l \cdot r_2}$$

(LetterStar)
$$\frac{l \cdot r_1 \sqsubseteq r_2 r_2^* r_3}{l \cdot r_1 \sqsubseteq r_2^* r_3} \quad [l \in \mathsf{first}(r_2)]$$

(LetterChoice)
$$\frac{l \cdot r_1 \sqsubseteq r_i r_4}{l \cdot r_1 \sqsubseteq (r_2 + r_3) r_4} \quad \left[\begin{array}{c} i \in \{2,3\} \\ l \in \mathsf{first}(r_i) \end{array}\right]$$

(LeftChoice)
$$\frac{\begin{array}{c} r_1 r_3 \sqsubseteq r_4 \\ r_2 r_3 \sqsubseteq r_4 \end{array}}{(r_1 + r_2) r_3 \sqsubseteq r_4}$$

(LeftStar)
$$\frac{\begin{array}{c} r_1 r_1^* r_2 \sqsubseteq r_3 r_4 \\ r_2 \sqsubseteq r_3 r_4 \end{array}}{r_1^* r_2 \sqsubseteq r_3 r_4} \quad \left[\begin{array}{c} \mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_3) \neq \varnothing \\ \exists l, r_5 : r_3 = l \vee r_3 = r_5^* \end{array}\right]$$

2.4 Properties of the Algorithm

To help understanding the algorithm and the rules, Table 2.4 shows which rules might apply for each combination of header-forms of the left-hand and right-hand expressions. The following lemma implies

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathit{Store}(1)}{(\mathsf{Letter})\ 5 : a^*b^* \sqsubseteq (a+b)^*}}{(\mathsf{LetterChoice})\ 4 : aa^*b^* \sqsubseteq a(a+b)^*}}{(\mathsf{LetterStar})\ 3 : aa^*b^* \sqsubseteq (a+b)(a+b)^*}}{(\mathsf{LeftStar})\ 2 : aa^*b^* \sqsubseteq (a+b)^*} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathit{Store}(6)}{(\mathsf{Letter})\ 10 : b^* \sqsubseteq (a+b)^*}}{(\mathsf{LetterChoice})\ 9 : bb^* \sqsubseteq b(a+b)^*}}{(\mathsf{LetterStar})\ 8 : bb^* \sqsubseteq (a+b)(a+b)^*}}{(\mathsf{LeftStar})\ 7 : bb^* \sqsubseteq (a+b)^*} \quad \dfrac{(\mathsf{Axm})}{11 : \epsilon \sqsubseteq (a+b)^*}}{6 : b^* \sqsubseteq (a+b)^*}}{1 : a^*b^* \sqsubseteq (a+b)^*}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathit{Store}(3)}{(\mathsf{Letter})\ 7 : b(ab)^*a \sqsubseteq (ba)^*}}{(\mathsf{LeftStar})\ 6 : ab(ab)^*a \sqsubseteq a(ba)^*} \quad \dfrac{\dfrac{(\mathsf{Axm})}{(\mathsf{Letter})\ 9 : \epsilon \sqsubseteq (ba)^*}}{8 : a \sqsubseteq a(ba)^*}}{(\mathsf{Letter})\ 5 : (ab)^*a \sqsubseteq a(ba)^*}}{(\mathsf{LetterStar})\ 4 : b(ab)^*a \sqsubseteq ba(ba)^*}}{(\mathsf{Letter})\ 3 : b(ab)^*a \sqsubseteq (ba)^*}}{(\mathsf{LeftStar})\ 2 : ab(ab)^*a \sqsubseteq a(ba)^*} \quad \dfrac{\dfrac{\mathit{Store}(9)}{(\mathsf{Letter})\ 12 : \epsilon \sqsubseteq (ba)^*}}{11 : a \sqsubseteq a(ba)^*}}{1 : (ab)^*a \sqsubseteq a(ba)^*}$$

Figure 2.2: Example usage of the inference rules to decide $a^*b^* \sqsubseteq (a+b)^*$ and $(ab)^*a \sqsubseteq a(ba)*$

Table 2.3: The rules for the relation $\sqsubseteq$ (Continued from Table 2.2).

(StarChoice1)

$$\frac{r_1^* r_2 \sqsubseteq r_i r_5}{r_1^* r_2 \sqsubseteq (r_3+r_4) r_5} \quad \left[\begin{array}{c} i \in \{3,4\} \\ \mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_i) \neq \varnothing \\ \mathsf{first}(r_1^* r_2) \subseteq \mathsf{first}(r_i r_5) \\ r_2 \notin \mathfrak{N}_\Sigma \vee r_i \in \mathfrak{N}_\Sigma \end{array}\right]$$

(StarChoice2)

$$\frac{\begin{array}{c} r_1 r_1^* r_2 \sqsubseteq (r_3+r_4) r_5 \\ r_2 \sqsubseteq (r_3+r_4) r_5 \end{array}}{r_1^* r_2 \sqsubseteq (r_3+r_4) r_5} \quad \left[\begin{array}{c} \mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_3+r_4) \neq \varnothing \\ \left(\begin{array}{l} (r_4 \notin \mathfrak{N}_\Sigma \wedge \mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_3 r_5) \neq \varnothing \\ \vee\ \mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_3) \neq \varnothing \\ \vee (r_2 \in \mathfrak{N}_\Sigma \wedge r_4 \notin \mathfrak{N}_\Sigma) \end{array}\right) \\ \left(\begin{array}{l} (r_3 \notin \mathfrak{N}_\Sigma \wedge \mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_4 r_5) \neq \varnothing \\ \vee\ \mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_4) \neq \varnothing \\ \vee (r_2 \in \mathfrak{N}_\Sigma \wedge r_3 \notin \mathfrak{N}_\Sigma) \end{array}\right) \end{array}\right]$$

(ElimCat)

$$\frac{r_1 \sqsubseteq r_3}{r_1 \sqsubseteq r_2 r_3} \quad \left[\begin{array}{c} \exists l, r_4, r_5 : r_1 = l \cdot r_4 \ \vee \ r_1 = r_4^* r_5 \\ r_2 \in \mathfrak{N}_\Sigma \\ \mathsf{first}(r_1) \subseteq \mathsf{first}(r_3) \end{array}\right]$$

that if the second "if" inside the main loop of the algorithm fails, then there is always at least one rule matching the pair. Note also that the conditions in the lemma hold for all pairs which are in the inclusion relation.

Lemma 2.4.1. *For any regular expressions r_1 and r_2 in header normal form, where* $\mathsf{first}(r_1) \subseteq \mathsf{first}(r_2)$, *$r_1 \notin \mathfrak{N}_\Sigma \vee r_2 \in \mathfrak{N}_\Sigma$, and $r_1 = \epsilon \vee r_2 \neq \epsilon$, there is at least one rule instance with conclusion $r_1 \sqsubseteq r_2$.*

Proof. By a case distinction on r_1 and r_2, using Tables 2.2, 2.3, and 2.4, Definition 2.2.2, and Lemma 2.2.5. The only combinations that are never matched are when the right-hand expression is ϵ while the left-hand expression is not (5, 9, and 13 in Table 2.4), and the combinations where the left-hand expression is ϵ while the right-hand is of the form $l \cdot r$ (2 in Table 2.4). The former cannot occur under the assumptions

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{Fail because } \mathsf{first}((ab)^*) \not\subseteq \mathsf{first}(b^*)}{(\mathsf{Letter})\ \ 7 : (ab)^* \sqsubseteq b^*}}{(\mathsf{LetterStar})\ \ 6 : b(ab)^* \sqsubseteq bb^*}}{(\mathsf{ElimCat})\ \ 5 : b(ab)^* \sqsubseteq b^*}}{(\mathsf{Letter})\ \ 4 : b(ab)^* \sqsubseteq a^*b^*}}{(\mathsf{LetterStar})\ \ 3 : ab(ab)^* \sqsubseteq aa^*b^*}}{(\mathsf{LeftStar})\ \ 2 : ab(ab)^* \sqsubseteq a^*b^*} \quad \dfrac{(\mathsf{Axm})}{8 : \epsilon \sqsubseteq a^*b^*}}{1 : (ab)^* \sqsubseteq a^*b^*}$$

Figure 2.3: Example usage of the inference rules to decide that $(ab)^* \sqsubseteq a^*b^*$ is not sound

of the lemma since subexpressions of the forms $\epsilon \cdot r'$, $\epsilon + \epsilon$ and ϵ^* are assumed removed, while the latter combinations follow from that $l \cdot r \notin \mathfrak{N}_\Sigma$.

The cases when $r_1 = \epsilon$ (1, 2, 3, and 4 in Table 2.4) the pair matches (Axm), as the only side condition, $r_2 \in \mathfrak{N}_\Sigma$, is true by assumption. When both expressions start with a letter (6 in Table 2.4), the pair matches (Letter), which has no side-conditions.

In the cases where $r_1 = lr'$ and $r_2 = (r_3 + r_4)r_5$ (7 in Table 2.4) we have by assumption either that $l \in \mathsf{first}(r_3 + r_4)$, such that (LetterChoice) matches, or we have $r_3 + r_4 \in \mathfrak{N}_\Sigma$ and $l \in \mathsf{first}(r_5)$ such that (ElimCat) matches the pair.

In the cases where $r_1 = lr'$ and $r_2 = r_3^* r_4$ (8 in Table 2.4) we have by assumption either that $l \in \mathsf{first}(r_3)$, such that (LetterStar) matches, or we have $l \in \mathsf{first}(r_4)$ such that (ElimCat) matches the pair.

The cases where $r_1 = (r_3 + r_4)r_5$ (9, 10, 11, and 12 in Table 2.4) match (LeftChoice) which has no side-conditions.

The cases where $r_1 = r_3^* r_4$ and $r_2 = l \cdot r_5$ (14 in Table 2.4) are matched by (LeftStar). The first side-condition holds by the assumptions in the lemma, and the second by the form of r_2.

For the cases where $r_1 = r_3^* r_4$ and $r_2 = r_5^* r_6$ (16 in Table 2.4), note that from the assumptions in the lemma, $\mathsf{first}(r_1) \subseteq \mathsf{first}(r_5^* r_6)$. There are two cases to treat. Firstly, we can have $\mathsf{first}(r_1) \subseteq \mathsf{first}(r_6)$ such that the pair matches (ElimCat). Otherwise, we have $\mathsf{first}(r_1) \cap \mathsf{first}(r_5) \neq \varnothing$ which implies that the pair matches (LeftStar).

We now treat the hardest case (15 in Table 2.4). For expository reasons we stick to the notation in (StarChoice2) and take the left hand side ("r_1") to be $r_1^* r_2$ and the right hand side ("r_2") to be $(r_3 + r_4)r_5$. The pair can possibly match (ElimCat), (StarChoice1), or (StarChoice2). We will treat this case by assuming that the pair does not match (ElimCat) or (StarChoice1), and proceeding to show that it is then matched by (StarChoice2). We only need to show that all the side-conditions of (StarChoice2) hold. For the first side-condition, note that one assumption in the lemma is that $\mathsf{first}(r_1^* r_2) \subseteq \mathsf{first}((r_3 + r_4)r_5)$. If $r_3 + r_4 \notin \mathfrak{N}_\Sigma$, we have $\mathsf{first}((r_3 + r_4)r_5) = \mathsf{first}(r_3 + r_4)$ and therefore get $\mathsf{first}(r_1^* r_2) \subseteq \mathsf{first}(r_3 + r_4)$, so the first side-condition holds. Otherwise, if $r_3 + r_4 \in \mathfrak{N}_\Sigma$, we get from the fact that (ElimCat) does not match the conclusion that $\mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_5)$, so we must also have the first side-condition. For the two remaining side-conditions of (StarChoice2), note first that since (StarChoice1) does not match, we get the following two facts:

$$\mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_3) = \varnothing \lor \mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_3 r_5) \lor (r_2 \in \mathfrak{N}_\Sigma \land r_3 \notin \mathfrak{N}_\Sigma) \quad (2.1)$$

$$\mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_4) = \varnothing \lor \mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_4 r_5) \lor (r_2 \in \mathfrak{N}_\Sigma \land r_4 \notin \mathfrak{N}_\Sigma) \quad (2.2)$$

From the fact that (ElimCat) did not match, we get that $r_3 \in \mathfrak{N}_\Sigma \Rightarrow \mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_5)$ and that $r_4 \in \mathfrak{N}_\Sigma \Rightarrow \mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_5)$. Therefore, $\mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_3) = \varnothing \Rightarrow \mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_3 r_5)$ and $\mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_4) = \varnothing \Rightarrow \mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_4 r_5)$. Hence (2.1) and (2.2) can be simplified to

$$\mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_3 r_5) \lor (r_2 \in \mathfrak{N}_\Sigma \land r_3 \notin \mathfrak{N}_\Sigma) \quad (2.3)$$

$$\mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_4 r_5) \lor (r_2 \in \mathfrak{N}_\Sigma \land r_4 \notin \mathfrak{N}_\Sigma) \quad (2.4)$$

Applying standard operations in propositional logic to (2.3) and (2.4) we get

$$\left(\begin{array}{l} (r_3 \notin \mathfrak{N}_\Sigma \land \mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_3 r_5)) \\ \lor\ (r_3 \in \mathfrak{N}_\Sigma \land \mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_3 r_5)) \\ \lor\ (r_2 \in \mathfrak{N}_\Sigma \land r_3 \notin \mathfrak{N}_\Sigma) \end{array}\right) \quad (2.5)$$

$$\left(\begin{array}{l} (r_4 \notin \mathfrak{N}_\Sigma \land \mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_4 r_5)) \\ \lor\ (r_4 \in \mathfrak{N}_\Sigma \land \mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_4 r_5)) \\ \lor\ (r_2 \in \mathfrak{N}_\Sigma \land r_4 \notin \mathfrak{N}_\Sigma) \end{array}\right) \quad (2.6)$$

We use again that $\mathsf{first}(r_1^* r_2) \subseteq \mathsf{first}((r_3 + r_4)r_5) = \mathsf{first}(r_3 r_5) \cup \mathsf{first}(r_4 r_5)$ to get the following implications: $\mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_3 r_5) \Rightarrow \mathsf{first}(r_1^* r_2) \cap$

$\mathsf{first}(r_4r_5) \neq \varnothing$, $\mathsf{first}(r_1^*r_2) \not\subseteq \mathsf{first}(r_4r_5) \Rightarrow \mathsf{first}(r_1^*r_2) \cap \mathsf{first}(r_3r_5) \neq \varnothing$, $(r_3 \in \mathfrak{N}_\Sigma \wedge \mathsf{first}(r_1^*r_2) \not\subseteq \mathsf{first}(r_3r_5)) \Rightarrow \mathsf{first}(r_1^*r_2) \cap \mathsf{first}(r_4) \neq \varnothing$, and $(r_4 \in \mathfrak{N}_\Sigma \wedge \mathsf{first}(r_1^*r_2) \not\subseteq \mathsf{first}(r_4r_5)) \Rightarrow \mathsf{first}(r_1^*r_2) \cap \mathsf{first}(r_3) \neq \varnothing$. Applying these implications to (2.5) and (2.6) gives exactly the two last side-conditions of (StarChoice2). □

2.4.1 1-Unambiguity and the Rules

We must make sure that the rules given in Tables 2.2 and 2.3 preserve 1-unambiguity for the *right-hand* expressions.

Lemma 2.4.2 (Preservation of 1-unambiguity). *For any rule instance, if the right-hand expression in the conclusion is 1-unambiguous, then also the right-hand expressions in all the premises are 1-unambiguous.*

Proof. For most rules we either have that the right-hand expression is the same in the premise and the conclusion, or we can use the fact that all subexpressions of a 1-unambiguous regular expression are 1-unambiguous. The latter fact was shown by Brüggemann-Klein & Wood [12, Lemma 3.2]. The remaining cases can also be shown by using [12, Lemma 3.2]. For the convenience of the reader, we show it in an alternative way using Definition 2.2.12.

For the rule (LetterStar), the right-hand expression of the premise is of the form $r_1r_1^*r_2$ and we know that $r_1^*r_2$ is 1-unambiguous. We will prove that $r_1r_1^*r_2$ is 1-unambiguous given that $r_1^*r_2$ is 1-unambiguous. We must use the fact that all expressions are in star normal form (see Definition 2.2.11), thus $r_1 \notin \mathfrak{N}_\Sigma$, and $\mathsf{first}(\mu(r_1)) \cap \mathsf{followLast}(\mu(r_1)) = \varnothing$. Take $u, v, w \in \mathsf{sym}(\mu(r_1r_1^*r_2))^*$ and $p, q \in \mathsf{sym}(\mu(r_1r_1^*r_2))$ as in Definition 2.2.12, such that $upv, uqw \in \|\mu(r_1r_1^*r_2)\|$ and $(r_1r_1^*r_2)[p] = (r_1r_1^*r_2)[q]$. To prove 1-unambiguity of $r_1r_1^*r_2$ we must show that $p = q$. For each of the words upv and uqw there are two possibilities to con-

Table 2.4: The rules that might apply for any combination of header-forms of the left-hand and right-hand expressions

Left \ Right	ϵ	$l \cdot r$	$(r_1 + r_2) \cdot r_3$	$r_1^* \cdot r_2$
ϵ	1 : (Axm)	2 : $\not\subset$	3 : (Axm)	4 : (Axm)
$l \cdot r$	5 : $\not\subset$	6 : (Letter)	7 : (ElimCat) (LetterChoice)	8 : (ElimCat) (LetterStar)
$(r_1 + r_2) \cdot r_3$	9 : $\not\subset$	10 : (LeftChoice)	11 : (LeftChoice)	12 : (LeftChoice)
$r_1^* \cdot r_2$	13 : $\not\subset$	14 : (LeftStar)	15 : (ElimCat) (StarChoice1) (StarChoice2)	16 : (ElimCat) (LeftStar)

sider:

$$\exists! u_1, u_2, p_1, v_1 : \begin{array}{l} u{=}(1u_1)\cdot(2u_2) \wedge v{=}2v_1 \wedge p{=}2p_1 \\ \wedge u_1 \in \|\mu(r_1)\| \wedge u_2p_1v_1 \in \|\mu(r_1^*r_2)\| \end{array} \tag{2.7}$$

$$\exists! u_1, p_1, v_1, v_2 : \begin{array}{l} u{=}1u_1 \wedge p{=}1p_1 \wedge v{=}(1v_1)\cdot(2v_2) \\ \wedge u_1p_1v_1 \in \|\mu(r_1)\| \wedge v_2 \in \|\mu(r_1^*r_2)\| \end{array} \tag{2.8}$$

$$\exists! u_1, u_2, q_1, w_1 : \begin{array}{l} u{=}(1u_1)\cdot(2u_2) \wedge w{=}2w_1 \wedge q{=}2q_1 \\ \wedge u_1 \in \|\mu(r_1)\| \wedge u_2q_1w_1 \in \|\mu(r_1^*r_2)\| \end{array} \tag{2.9}$$

$$\exists! u_1, q_1w_1, w_2 : \begin{array}{l} u{=}1u_1 \wedge q{=}1q_1 \wedge w{=}(1w_1)\cdot(2w_2) \\ \wedge u_1q_1w_1 \in \|\mu(r_1)\| \wedge w_2 \in \|\mu(r_1^*r_2)\| \end{array} \tag{2.10}$$

Exactly one of (2.7) or (2.8) must hold, and exactly one of (2.9) or (2.10) must hold. Firstly, if both (2.8) and (2.10) hold, then $p_1 = q_1$ follows from 1-unambiguity of r_1, and thus $p = 1p_1 = 1q_1 = q$. Secondly, if both (2.7) and (2.9) hold, the u_1 and u_2 chosen must be the same in both cases, and therefore 1-unambiguity of $r_1^*r_2$ can be used to get $p_1 = q_1$. Thus $p = 2p_1 = 2q_1 = q$. We now show that the two remaining combinations cannot hold. By symmetry, we only treat one case, and assume (by contradiction) that (2.8) and (2.9) hold. This implies that $u_2 = \epsilon$, thus $p_1 \in \mathsf{followLast}(\mu(r_1))$ and $q_1 \in \mathsf{first}(\mu(r_1^* \cdot r_2))$. Now we can use $(\langle 1,1\rangle \odot (u_1 \cdot p_1 \cdot v_1)) \cdot v_2 \in \|\mu(r_1^*r_2)\|$ and $(\langle 1,1\rangle \odot u_1) \cdot q_1 \cdot w_1 \in \|\mu(r_1^*r_2)\|$ together with the fact that $(r_1^*r_2)[q] = (r_1^*r_2)[\langle 1,1\rangle \odot p_1]$ in Definition 2.2.12 to show that $\langle 1,1\rangle \odot p_1 = q_1$. Combined with $q_1 \in \mathsf{first}(\mu(r_1^* \cdot r_2))$ we get that $p_1 \in \mathsf{first}(\mu(r_1))$. But then $p_1 \in \mathsf{first}(\mu(r_1)) \cap \mathsf{followLast}(\mu(r_1))$, which contradicts with the fact that $r_1^*r_2$ is in star normal form.

For (LetterChoice) and (StarChoice1), the right-hand expression in the conclusion is of the form $(r_1 + r_2)r_3$. We can, by symmetry, assume the right-hand expression in the premise is r_1r_3. We assume the right-hand expressions in the conclusion is 1-unambiguous and show that r_1r_3 also is 1-unambiguous. Note now that $\|\mu(r_1 \cdot r_3)\| = 1\|\mu(r_1)\| \cdot 2\|\mu(r_3)\|$, and $\|\mu((r_1 + r_2)r_3)\| = 1\|\mu(r_1 + r_2)\| \cdot 2\|\mu(r_3)\| = (\langle 1,1\rangle \odot \|\mu(r_1)\|) \cdot 2\|\mu(r_3)\| \cup (\langle 1,2\rangle \odot \|\mu(r_1)\|) \cdot 2\|\mu(r_3)\|$. For any $upv, uqw \in \|\mu(r_1r_3)\|$ as in Definition 2.2.12 concerning r_1r_3 we therefore have corresponding u', p', q', v', w' concerning $(r_1 + r_2)r_3$ which are obtained by prefixing the positions in u, p, q, v, w starting in 1 with one more 1. Furthermore, $p' = q' \Rightarrow p = q$. Since $(r_1 + r_2)r_3$ is 1-unambiguous we have that $(r_1 + r_2)r_3[p'] = (r_1 + r_2)r_3[q'] \Rightarrow p' = q'$

and therefore also that $r_1 r_3[p] = r_1 r_3[q] \Rightarrow p = q$, such that $r_1 r_3$ is also 1-unambiguous.

□

We must now substantiate the claim that if the side-conditions of more than one applicable rule hold, the right-hand expression is 1-ambiguous.

Lemma 2.4.3. *For any two regular expressions r_1 and r_2, where r_2 is 1-unambiguous, there is at most one rule instance with $r_1 \sqsubseteq r_2$ in the conclusion.*

Proof. This is proved by comparing each pair of rule instances of rules occurring in Table 2.4 and using Definition 2.2.12. For each case, we show that the existence of several rule instances with the same conclusion implies that the right-hand expression is 1-ambiguous.

- We first consider the case that one rule has several instances matching the same conclusion. The only rules that can have more than one instance with the same conclusion are (StarChoice1) and (LetterChoice). For (LetterChoice), the conclusion is of the form $l \cdot r_1 \sqsubseteq (r_2 + r_3) \cdot r_4$, and the existence of two instances implies that $l \in \mathsf{first}(r_2) \cap \mathsf{first}(r_3)$. This can only be the case if the right-hand expression is 1-ambiguous. For (StarChoice1), the conclusion is of the form $r_1^* r_2 \sqsubseteq (r_3 + r_4) r_5$, and the existence of two instances of this rule would imply that $\mathsf{first}(r_1^* r_2)$ and $\mathsf{first}(r_4)$ have a non-empty intersection, which furthermore is included in $\mathsf{first}(r_3 r_5)$. The expression $(r_3 + r_4) r_5$ is therefore 1-ambiguous.

- If instances of both (ElimCat) and either (LetterStar) or (LetterChoice) match the pair of expressions (see 7 and 8 in Table 2.4), then the right-hand expression is of the form $r_2 r_3$. From (LetterStar) and (LetterChoice) the left-hand expression is of the form $l r_1$ and $l \in \mathsf{first}(r_2)$. From (ElimCat) we get that $r_2 \in \mathfrak{N}_\Sigma$ and $l \in \mathsf{first}(r_3)$. But this means that $r_2 r_3$ is 1-ambiguous.

- If instances of both (ElimCat) and either (StarChoice1) or (StarChoice2) had the same conclusion (see 15 in Table 2.4), then the the conclusion is of the form $r_1^* r_2 \sqsubseteq (r_3 + r_4) r_5$. From (ElimCat), we get that $r_3 + r_4 \in \mathfrak{N}_\Sigma$ and $\mathsf{first}(r_1^* r_2) \subseteq \mathsf{first}(r_5)$. From the second side-condition of (StarChoice1) or the first side-condition of (StarChoice2) we get that $\mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_3 + r_4) \neq \varnothing$. This means that $\mathsf{first}(r_3 + r_4) \cap \mathsf{first}(r_5) \neq$

$$\cfrac{\cfrac{\cfrac{\cfrac{(\mathsf{Axm})}{(\mathsf{Letter})\ \ 4 : \epsilon \sqsubseteq \epsilon}}{(\mathsf{Letter})\ \ 3 : b \sqsubseteq b}}{(\mathsf{LetterChoice})\ \ 2 : ab \sqsubseteq ab}}{1 : ab \sqsubseteq (a + (b+c)^*c(b+c)\cdots(b+c))b}$$

Figure 2.4: Example usage of the inference rules

$\varnothing$. We combine the latter with $r_3 + r_4 \in \mathfrak{N}_\Sigma$ to get that the right-hand expression $(r_3 + r_4)r_5$ is 1-ambiguous.

• For the cases where both an instance of $(\mathsf{StarChoice1})$ and one of $(\mathsf{StarChoice2})$ match the conclusion, we can by symmetry assume the instance of $(\mathsf{StarChoice1})$ has $i = 3$. The last side-condition of $(\mathsf{StarChoice1})$ is then $r_2 \notin \mathfrak{N}_\Sigma \vee r_3 \in \mathfrak{N}_\Sigma$. This is exactly the negation of the third disjunct of the third side condition of $(\mathsf{StarChoice2})$. We must therefore have that the remaining disjunction holds, that is,

$$(r_3 \notin \mathfrak{N}_\Sigma \wedge \mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_4 r_5) \neq \varnothing) \vee \mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_4) \neq \varnothing \tag{2.11}$$

Assume first that the left disjunct of (2.11) holds. Then since $r_3 \notin \mathfrak{N}_\Sigma$, we get $\mathsf{first}(r_3 r_5) = \mathsf{first}(r_3)$. Combined with the third side-condition of $(\mathsf{StarChoice1})$ this implies that $\mathsf{first}(r_4 r_5) \cap \mathsf{first}(r_3) \neq \varnothing$, which means that $(r_3 + r_4)r_5$ is 1-ambiguous. Otherwise, if the right disjunct of (2.11) holds, we can use the third side-condition of $(\mathsf{StarChoice1})$ to get that $\mathsf{first}(r_4) \cap \mathsf{first}(r_3 r_5) \neq \varnothing$, which also means that $(r_3 + r_4)r_5$ is 1-ambiguous.

• If instances of both $(\mathsf{ElimCat})$ and $(\mathsf{LeftStar})$ match the pair of expressions (see 16 in Table 2.4), then the conclusion is of the form $r_1^* r_2 \sqsubseteq r_3 r_4$, where $r_3 \in \mathfrak{N}_\Sigma$ and both $\mathsf{first}(r_1^* r_2) \subseteq \mathsf{first}(r_4)$ and $\mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_3) \neq \varnothing$. This can only hold if $r_3 r_4$ is 1-ambiguous.

□

2.4.2 Invertibility of the Rules

We shall now prove that the rules given in Tables 2.2 and 2.3 are *invertible*. By this we mean that, for each rule instance, assuming that no other rule instance matches the conclusion, then the conclusion is sound if and only if all premises are sound.

Proof. By a case distinction on the rules. For all rules, the fact that the premise(s) implies the conclusion follows almost directly from Definition 2.2.2. We only treat the converse:

- For (Axm), we only note that the side-condition is that the right-hand expression is nullable, and then $\{\epsilon\}$ is of course a subset of the language.

- For (Letter) we are just removing a single letter prefix from both languages, and this preserves the inclusion relation.

- For (LetterStar), the conclusion is of the form $lr_1 \sqsubseteq r_2^* r_3$. Note that $\|r_2^* r_3\| = \|r_2 r_2^* r_3\| \cup \|r_3\|$. Since (ElimCat) does not match the conclusion, and $r_2^* \in \mathfrak{N}_\Sigma$, we must have that $\mathsf{first}(lr_1) \not\subseteq \mathsf{first}(r_3)$, that is, $l \notin \mathsf{first}(r_3)$. Therefore $\|lr_1\| \cap \|r_3\| = \emptyset$, and thus $\|lr_1\| \subseteq \|r_2 r_2^* r_3\|$ and the premise is sound

- For (LetterChoice), the conclusion is of the form $lr_1 \sqsubseteq (r_2 + r_3)r_4$. Again we depend on the fact that no other instance of (LetterChoice) nor (ElimCat) match the conclusion. We can assume by symmetry that $i = 2$ and the premise is of the form $lr_1 \sqsubseteq r_2 r_4$. Since $i = 3$ does not match we get that $l \notin \mathsf{first}(r_3)$. Note that $\|(r_2 + r_3)r_4\| = \|r_2 r_4\| \cup \|r_3 r_4\|$. Since (ElimCat) does not match the conclusion we get that $(r_2 + r_3) \in \mathfrak{N}_\Sigma \Rightarrow l \notin \mathsf{first}(r_4)$. This implies that $\|lr_1\| \cap \|r_3 r_4\| = \emptyset$, so $\|lr_1\| \subseteq \|r_2 r_4\|$ and we have the premise.

- For (LeftChoice), the implication follows from Definition 2.2.2.

- (LeftStar) and (StarChoice2) hold by Definition 2.2.2, as $\|r_1^* r_2\| = \|r_1 r_1^* r_2\| \cup \|r_2\|$.

- For (StarChoice1), the conclusion is of the form $r_1^* r_2 \sqsubseteq (r_3 + r_4)r_5$. Note again that $\|(r_3 + r_4)r_5\| = \|r_3 r_5\| \cup \|r_4 r_5\|$. We can, by symmetry, assume $i = 3$. The second side-condition is then that $\mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_3) \neq \emptyset$. Note that this implies the first side-condition and the middle disjunct of the second side-condition in (StarChoice2). Since (StarChoice2) does not match, we must have the negation of the third side-condition of (StarChoice2). Hence

$$\mathsf{first}(r_4) \cap \mathsf{first}(r_1^* r_2) = \emptyset \wedge (r_3 \in \mathfrak{N}_\Sigma \vee \mathsf{first}(r_4 r_5) \cap \mathsf{first}(r_1^* r_2) = \emptyset) \quad (2.12)$$

Now, if $r_3 \in \mathfrak{N}_\Sigma$, we get that $\|r_5\| \subseteq \|r_3 r_5\|$, which implies that $\|(r_3 + r_4)r_5\| = \|r_3 r_5\| \cup (\|r_4 r_5\| - \|r_5\|)$. From (2.12) we have that $\mathsf{first}(r_4) \cap \mathsf{first}(r_1^* r_2) = \emptyset$, which implies that $(\|r_4 r_5\| - \|r_5\|) \cap \|r_1^* r_2\| = \emptyset$. Therefore the premise $r_1^* r_2 \sqsubseteq r_3 r_5$ is sound. On the other hand, if $r_3 \notin \mathfrak{N}_\Sigma$, we get from (2.12) that $\mathsf{first}(r_4 r_5) \cap \mathsf{first}(r_1^* r_2) = \emptyset$. This implies that $\|r_1^* r_2\| \cap \|r_4 r_5\| = \emptyset$, which implies that the premise $r_1^* r_2 \sqsubseteq r_3 r_5$ is sound.

- For (ElimCat), we have $\|r_2 r_3\| = (\|r_2 r_3\| - \|r_3\|) \cup \|r_3\|$. Therefore it is sufficient to show that $\mathsf{first}(r_1) \cap \mathsf{first}(r_2) = \emptyset$. Note that the left-hand expression is constrained by the first side-condition to be of the form $l \cdot r_4$ or $r_4^* r_5$. The right-hand expression must from Definition 2.2.3, and the definition of header-form be of the form $r_6^* r_3$ or $(r_6 + r_7) r_3$. We do a case distinction on these forms. If r_1 is of the form $l \cdot r_4$, then since neither (LetterStar) or (LetterChoice) matches the conclusion, we get that $l \notin \mathsf{first}(r_2)$, and the premise $r_1 \sqsubseteq r_3$ must be sound. If the conclusion is of the form $r_4^* r_5 \sqsubseteq r_6^* r_7$, then we must have that the first side-condition of (LeftStar) fails. Thus $\mathsf{first}(r_4^* r_5) \cap \mathsf{first}(r_6^*) = \emptyset$. Lastly, if the conclusion is of the form $r_4^* r_5 \sqsubseteq (r_6 + r_7) r_3$, note that from the second side-condition of (ElimCat) we have $r_6 + r_7 \in \mathfrak{N}_\Sigma$, so we can by symmetry assume $r_6 \in \mathfrak{N}_\Sigma$. We will use that (StarChoice1) with $i = 3$ does not match, and that the third and fourth side-conditions of this instance of (StarChoice1) hold by the assumption that the side-conditions of (ElimCat) hold. This implies that the second side-condition of (StarChoice1) with $i = 3$ does not hold, so we get $\mathsf{first}(r_4^* r_5) \cap \mathsf{first}(r_6) = \emptyset$. If $r_7 \in \mathfrak{N}_\Sigma$, we can use a similar argument to also get $\mathsf{first}(r_4^* r_5) \cap \mathsf{first}(r_7) = \emptyset$. Otherwise, if $r_7 \notin \mathfrak{N}_\Sigma$, we use the fact that (StarChoice2) does not match. The first disjunct of the second side-condition of (StarChoice2) holds, since we have assumed $r_7 \notin \mathfrak{N}_\Sigma$, and we argued above that $\mathsf{first}(r_4^* r_5) \subseteq \mathsf{first}(r_6 r_3)$. Therefore either the first or the third side-condition of (StarChoice2) must fail. Either case implies that $\mathsf{first}(r_4^* r_5) \cap \mathsf{first}(r_7) = \emptyset$, so we are done.

□

Invertibility implies that, at any point during an execution of the algorithm, the pair originally given as input is in the inclusion relation if and only if all the pairs in both the store S and the stack T are in the inclusion relation. These properties are used in the proofs of soundness and completeness below.

2.4.3 Termination and Polynomial Run-time

To prove that the algorithm always terminates in polynomial time, we will prove that the number of iterations of the main loop where at least one new pair is pushed onto the stack T, has an upper bound in the product of the number of positions in the two regular expressions given as input. This implies that the whole algorithm runs in polynomial time, by the following three observations.

- The number of positions in a regular expression is linear in the length of the regular expression.

- The number of iterations where no new pair is pushed to the stack T, cannot be more than one more than half the total of all iterations. Note that the iterations where no pairs are pushed are those where the first "if"-test "$(r_1, r_2) \in \mathsf{S}$" succeeds, those where the second "if"-test fails, and those where the pair matches (Axm). That these are not more than one more than the half follows from that the other rules never push more than two pairs, and standard arguments on binary trees.

- The time used in each iteration of the loop is polynomial in the size of the regular expressions given as input.

Assume that the algorithm is given r_l and r_r as input. We will prove that there is an injective mapping from each r' occurring on the left-hand or right-hand of a pair in the stack T during the run of the algorithm, to a p in $\mathsf{pos}(r_l)$ or $\mathsf{pos}(r_r)$, respectively. If r is the corresponding input expression, then $r[p]$ is the first factor of r'.

For the purposes of this section, let $\between$ be a special *undefined* position, and let $\mathsf{pos}(r)^{\between} = \mathsf{pos}(r) \cup \{\between\}$. We proceed to define a mapping next_r, which will be used to describe the expressions occurring in a run of the algorithm in terms of subexpressions of the corresponding expression given as input.

Definition 2.4.4 (next_r)**.** *For a regular expression r, let the mapping $\mathsf{next}_r : \mathsf{pos}(r) \to \mathsf{pos}(r)^{\between}$ be defined in the following top-down inductive manner:*

- *Put $\mathsf{next}_r(\langle\rangle) = \between$.*

- *If $r[p] = r_1 \cdot r_2$, put $\mathsf{next}_r(p1) = p2$ and put $\mathsf{next}_r(p2) = \mathsf{next}_r(p)$.*

- *If* $r[p] = r_1 + r_2$*, put* $\mathsf{next}_r(p1) = \mathsf{next}_r(p2) = \mathsf{next}_r(p)$.
- *If* $r[p] = r_1^*$*, put* $\mathsf{next}_r(p1) = p$.

We extend next_r *to* next_r^* *which maps a position in* r *to a list of positions in* r*:*

$$\mathsf{next}_r^*(p) = \begin{cases} \epsilon & \textit{if } \mathsf{next}_r(p) = \text{\char"2573} \\ \mathsf{next}_r(p) \cdot \mathsf{next}_r^*(\mathsf{next}_r(p)) & \textit{otherwise} \end{cases}$$

Example 2.4.5. *Let* $\Sigma = \{a, b, c, d\}$ *and* $r = ((a \cdot b) \cdot c^*) \cdot d$*. Then* $\mathsf{pos}(r) = \{\langle\rangle, \langle 1\rangle, \langle 1,1\rangle, \langle 1,1,1\rangle, \langle 1,1,2\rangle, \langle 1,2\rangle, \langle 1,2,1\rangle, \langle 2\rangle\}$*, and* next_r *and* next_r^* *have the following values:*

$\mathsf{pos}(r)$	next_r	next_r^*
$\langle\rangle$	╳	ϵ
$\langle 1\rangle$	$\langle 2\rangle$	$\langle 2\rangle$
$\langle 1,1\rangle$	$\langle 1,2\rangle$	$\langle 1,2\rangle \cdot \langle 2\rangle$
$\langle 1,1,1\rangle$	$\langle 1,1,2\rangle$	$\langle 1,1,2\rangle \cdot \langle 1,2\rangle \cdot \langle 2\rangle$
$\langle 1,1,2\rangle$	$\langle 1,2\rangle$	$\langle 1,2\rangle \cdot \langle 2\rangle$
$\langle 1,2\rangle$	$\langle 2\rangle$	$\langle 2\rangle$
$\langle 1,2,1\rangle$	$\langle 1,2\rangle$	$\langle 1,2\rangle \cdot \langle 2\rangle$
$\langle 2\rangle$	╳	ϵ

We now need an auxiliary lemma concerning header.

Lemma 2.4.6. *For any regular expressions* r*,* r_1*, …,* r_n*,* r'_1*, …,* r'_m*, and* r'*, if* $\mathsf{header}(r, r_1 \cdots r_n \cdot \epsilon) = r'_1 \cdots r'_m \cdot \epsilon$*, then* $\mathsf{header}(r, r_1 \cdots r_n \cdot r') = r'_1 \cdots r'_m \cdot r'$*.*

Proof. By induction on r. For the base case $r = \epsilon$ we have by definition that $\mathsf{header}(r, r_1 \cdots r_n \cdot \epsilon) = r_1 \cdots r_n \cdot \epsilon$, and $\mathsf{header}(r, r_1 \cdots r_n \cdot r') = r_1 \cdots r_n \cdot r'$, which means the lemma holds. For the cases where r is a letter, choice, or a starred expression, we get by definition that $\mathsf{header}(r, r_1 \cdots r_n \cdot \epsilon) = r \cdot r_1 \cdots r_n$, and $\mathsf{header}(r, r_1 \cdots r_n \cdot r') = r \cdot r_1 \cdots r_n \cdot r'$, which also means the lemma holds. For the induction cases when $r = r''_1 \cdot \epsilon$ we get $\mathsf{header}(r, r_1 \cdots r_n \cdot \epsilon) = \mathsf{header}(r''_1, r_1 \cdots r_n \cdot \epsilon)$ and $\mathsf{header}(r, r_1 \cdots r_n \cdot r') = \mathsf{header}(r''_1, r_1 \cdots r_n \cdot r')$. We now get the result by applying the induction hypothesis for r''_1 to $\mathsf{header}(r''_1, r_1 \cdots r_n \cdot \epsilon) = r'_1 \cdots r'_m \cdot \epsilon$. Lastly, we treat the induction cases where there are $p \geq 2$ and $r''_1, \ldots, r''_p \in R_\Sigma - \{\epsilon\}$ such that either $r = r''_1 \cdots r''_p$ and r''_p is not a concatenation, or $r = r''_1 \cdots r''_p \cdot \epsilon$. We then get

$$\mathsf{header}(r, r_1 \cdots r_n \cdot \epsilon) = \mathsf{header}(r''_1, r''_2 \cdots r''_p \cdot r_1 \cdots r_n \cdot \epsilon)$$
$$\mathsf{header}(r, r_1 \cdots r_n \cdot r') = \mathsf{header}(r''_1, r''_2 \cdots r''_p \cdot r_1 \cdots r_n \cdot r')$$

We get the result by applying the induction hypothesis for r''_1 to $\mathsf{header}(r''_1, r''_2 \cdots r''_p \cdot r_1 \cdots r_n \cdot \epsilon) = r'_1 \cdots r'_m \cdot \epsilon$. □

Corollary 2.4.7. *For $m > 0$, and regular expressions r, r', r'', $r'_1, \dots, r'_m$, if $\mathsf{hdf}(r) = r'' \cdot \mathsf{hdf}(r'_1 \cdots r'_m \cdot \epsilon)$, then $\mathsf{hdf}(r \cdot r') = r'' \cdot \mathsf{hdf}(r'_1 \cdots r'_m \cdot r')$.*

Proof. If $r = \epsilon$, the corollary holds vacuously. Otherwise, we can assume there are $r''_1, \dots, r''_n \in R_\Sigma - \{\epsilon\}$ and $n \geq 0$ such that either $r' = r''_1 \cdots r''_n$ where $n \geq 1$ and r''_n is not a concatenation, or that $r' = r''_1 \cdots r''_n \cdot \epsilon$. Put $r''' = r''_1 \cdots r''_n \cdot \epsilon$. Then $\mathsf{hdf}(r \cdot r') = \mathsf{header}(r, r''')$, $\mathsf{hdf}(r'_1 \cdots r'_m \cdot r') = \mathsf{header}(r'_1, r'_2 \cdots r'_m \cdot r''')$, and $\mathsf{header}(r, \epsilon) = r'' \cdot \mathsf{header}(r'_1, r'_2 \cdots r'_m \cdot \epsilon)$. From Lemmas 2.2.8 and 2.4.6 we therefore get $\mathsf{header}(r, r''') = r'' \cdot \mathsf{header}(r'_1, r'_2 \cdots r'_m \cdot r''')$. Hence, $\mathsf{hdf}(r \cdot r') = r'' \cdot \mathsf{hdf}(r'_1 \cdots r'_m \cdot r')$. □

We now need an auxiliary lemma concerning the mapping next^*.

Lemma 2.4.8. *For any regular expressions r, r_1, r_2, and any position $p \in \mathsf{pos}(r)$ such that $\mathsf{hdf}(r[p]) = r_1 \cdot r_2$, there is an $n \geq 0$ and positions q, $p_1, \dots, p_n \in \mathsf{pos}(r)$ such that $p \leq q$, $\mathsf{next}^*_r(q) = p_1 \cdot \cdots \cdot p_n \cdot \mathsf{next}^*_r(p)$, and $r_1 \cdot r_2 = r[q] \cdot \mathsf{hdf}(r[p_1] \cdots r[p_n] \cdot \epsilon)$.*

Proof. By induction on the expression $r[p]$. The base case when $r[p] = \epsilon$ holds vacuously. The base cases when $r[p] \in \Sigma$, and the induction cases where $r[p]$ is of the forms $r_1 + r_2$ or r_1^* hold immediately (without using the induction hypothesis), as $\mathsf{hdf}(r[p]) = r[p] \cdot \epsilon = r[p] \cdot \mathsf{hdf}(\epsilon)$ and we can use $q = p$ and $n = 0$. The remaining induction cases are when $r[p]$ is of the form $r[p1] \cdot r[p2]$. By the note after Definition 2.2.2 $r[p1] \neq \epsilon$ and we have the induction hypothesis for $r[p1]$. Combining this with Definition 2.4.4, we get that there are $q, p_1, \dots, p_n$, such that $p1 \leq q$, $\mathsf{next}^*_r(q) = p_1 \cdots p_n \cdot \mathsf{next}^*_r(p1) = p_1 \cdots p_n \cdot p2 \cdot \mathsf{next}^*_r(p)$, and $\mathsf{hdf}(r[p1]) = r[q] \cdot \mathsf{hdf}(r[p_1] \cdots r[p_n] \cdot \epsilon)$. The latter fact applied to Corollary 2.4.7 implies that $\mathsf{hdf}(r[p]) = r[q] \cdot \mathsf{hdf}(r[p_1] \cdots r[p_n] \cdot r[p2])$, so we have proved the lemma. □

We can now formulate the main lemma of this section, defining the mapping $\mathsf{iterPos}$.

Lemma 2.4.9. *For each regular expression r given as input to any execution of the algorithm there exists a mapping $\mathsf{iterPos}_r$ with the following properties.*

- *The domain of $\mathsf{iterPos}_r$ is the set of non-ϵ expressions occurring on the same side as r in any pair on the stack during the execution of the algorithm.*
- *The codomain of $\mathsf{iterPos}_r$ is $\mathsf{pos}(r)$.*
- *If $p = \mathsf{iterPos}_r(r')$, then $r' = r[p] \cdot \mathsf{hdf}(r[\mathsf{next}^*_r(p)])$.*

Proof. By induction on the number of iterations of the main loop in an execution of the algorithm. We can assume that the lemma holds for the expressions in the pair that is popped from the stack, and show that it holds for the expressions being pushed onto the stack. Remember that hdf is applied to the expressions before they are pushed onto the stack.

The base case is the expressions r_l and r_r given as input. By symmetry we treat only r_l. We can apply Lemma 2.4.8 to r_l and $\langle\rangle$ to get $q, p_1, \ldots, p_n$ such that $\mathsf{hdf}(r_l) = r_l[q] \cdot \mathsf{hdf}(r_l[p_1] \cdots r_l[p_n] \cdot \epsilon)$, and $\mathsf{next}^*_{r_l}(q) = p_1 \cdots p_n \cdot \mathsf{next}^*_{r_l}(\langle\rangle)$. By Definition 2.4.4 $\mathsf{next}^*_{r_l}(\langle\rangle) = \epsilon$, so we get $\mathsf{hdf}(r_l) = r_l[q] \cdot \mathsf{hdf}(r_l[\mathsf{next}^*_{r_l}(q)])$, and we can let $\mathsf{iterPos}_{r_l}(r_l) = q$.

The induction case for (Axm), the induction cases where the first if-test "$(r_1, r_2) \notin \mathsf{S}$" fails, and the cases where the second if-test holds (such that "No" is returned) all hold directly by using the induction hypothesis, since the stack is not changed.

In the remaining induction cases, the expressions put on the stack follow five patterns: 1: that $\mathsf{hdf}(r)$ is pushed after popping r, 2: that $\mathsf{hdf}(r)$ is pushed after popping $l \cdot r$, 3: that $\mathsf{hdf}(r_1 r_3)$ is pushed after popping $(r_1 + r_2)r_3$, 4: that $\mathsf{hdf}(r_1 r_1^* r_2)$ is pushed after popping $r_1^* r_2$, and lastly, 5: that r_2 is pushed after popping $r_1 r_2$ where $r_1 \in \mathfrak{N}_\Sigma$. We treat these cases separately.

1. If we push $\mathsf{hdf}(r)$ on the stack after popping r, we get from Lemma 2.2.8 that $\mathsf{hdf}(r) = r$, so we get the result from the induction hypothesis.

2. The first interesting case is where $l \cdot r_1$ is a member of the pair popped from the stack T, and $\mathsf{hdf}(r_1)$ is a member of the pair pushed. Assume r is the corresponding input expression. By the induction hypothesis we know that there is a p such that $\mathsf{iterPos}_r(lr_1) = p$, $r[p] = l$, and $\mathsf{hdf}(r[\mathsf{next}^*_r(p)]) = r_1$. If $r_1 = \epsilon$, the lemma holds vacuously. Otherwise, we must now calculate the value of $\mathsf{iterPos}_r(r_1)$, that is, a $p' \in \mathsf{pos}(r)$, and show that it has the required properties. We have $r[\mathsf{next}^*_r(p)] \neq \epsilon$, so we get $\mathsf{next}^*_r(p) \neq \epsilon$, thus $\mathsf{next}_r(p) \neq \chi$, and $r_1 = \mathsf{hdf}(r[\mathsf{next}_r(p)] \cdot r[\mathsf{next}^*_r(\mathsf{next}_r(p))])$. We can now apply Lemma 2.4.8 to r and $\mathsf{next}_r(p)$, and get $p', p_1, \ldots, p_n$ such that $\mathsf{hdf}(r[\mathsf{next}_r(p)]) = r[p'] \cdot \mathsf{hdf}(r[p_1] \cdots r[p_n] \cdot \epsilon)$ and $\mathsf{next}^*_r(p') = p_1 \cdots p_n \cdot \mathsf{next}^*_{r_l}(\mathsf{next}_r(p))$. By applying this to Corollary 2.4.7 we get that

$$\begin{aligned}&\mathsf{hdf}(r[\mathsf{next}_r(p)] \cdot r[\mathsf{next}^*_r(\mathsf{next}_r(p))])\\ &= r[p'] \cdot \mathsf{hdf}(r[\mathsf{next}^*_r(p')])\end{aligned}$$

Applying Lemma 2.2.8 we therefore get

$$\mathsf{hdf}(r_1) = r_1 = r[p'] \cdot \mathsf{hdf}(r[\mathsf{next}_r(p')]),$$

so the lemma holds.

3. We next treat the case when we push a pair containing an expression of the form $\mathsf{hdf}(r_1 r_3)$ on the stack after popping a pair containing $(r_1 + r_2)r_3$. Assume r is the corresponding input expression. By the induction hypothesis there is a p such that $\mathsf{iterPos}_r((r_1 + r_2)r_3) = p$, $r[p] = (r_1 + r_2)$, and $\mathsf{hdf}(r[\mathsf{next}^*_r(p)]) = r_3$. If $\mathsf{hdf}(r_1 r_3) = \epsilon$ the lemma holds vacuously. Otherwise, since $r_1 = r[p1]$ we get from Lemma 2.4.8 for r and $p1$ that there are $p', p_1, \ldots, p_n$ such that:

$$\mathsf{hdf}(r_1) = r[p'] \cdot \mathsf{hdf}(r[p_1] \cdots r[p_n] \cdot \epsilon) \tag{2.13}$$

$$\mathsf{next}^*_r(p') = p_1 \cdots p_n \cdot \mathsf{next}^*_r(p1) \tag{2.14}$$

Applying Corollary 2.4.7 to (2.13) we get

$$\mathsf{hdf}(r_1 \cdot r_3) = r[p'] \cdot \mathsf{hdf}(r[p_1] \cdots r[p_n] \cdot r_3)$$

Since $r_3 = r[\mathsf{next}^*_r(p)]$ we get $\mathsf{hdf}(r_1 \cdot r_3) = r[p'] \cdot \mathsf{hdf}(r[p_1] \cdots r[p_n] \cdot r[\mathsf{next}^*_r(p)])$. Furthermore, from Definition 2.4.4 $\mathsf{next}_r(p1) = \mathsf{next}_r(p)$, and therefore $\mathsf{next}^*_r(p1) = \mathsf{next}^*_r(p)$. Combining the latter with (2.14) we get $\mathsf{next}^*_r(p') = p_1 \cdots p_n \cdot \mathsf{next}^*_{r_l}(p)$. Finally, we therefore get $\mathsf{hdf}(r_1 \cdot r_3) = r[p'] \cdot \mathsf{hdf}(r[\mathsf{next}^*_r(p')])$, and we can put $\mathsf{iterPos}_r(r_3) = p'$.

4. We treat the case where $r_1^* r_2$ is a member of the pair popped from the stack, and $\mathsf{hdf}(r_1 r_1^* r_2)$ is a member of the pair pushed. Assume r is the corresponding input expression. By the induction hypothesis we have a p such that $\mathsf{iterPos}_r(r_1^* r_2) = p$, where $r[p] = r_1^*$ and $\mathsf{hdf}(r[\mathsf{next}_r^*(p)]) = r_2$. Since $r_1 = r[p1]$, we can apply Lemma 2.4.8 to r and $p1$, to get $p', p_1, \ldots, p_n$, such that $\mathsf{hdf}(r_1) = r[p'] \cdot \mathsf{hdf}(r[p_1 \cdots p_n])$ and $\mathsf{next}_r^*(p') = p_1 \cdots p_n \cdot \mathsf{next}_r^*(p1)$. By Definition 2.4.4, we get

$$\mathsf{next}_r^*(p') = p_1 \cdots p_n \cdot p \cdot \mathsf{next}_r^*(p)$$

Thus, applying Corollary 2.4.7 we get

$$\mathsf{hdf}(r_1 r_1^* r_2) = r[p'] \cdot \mathsf{hdf}(r[\mathsf{next}_r^*(p')])$$

So we can set $\mathsf{iterPos}_r(r_1 r_1^* r_2) = p'$.

5. For the case where $r_1 r_2$ is popped from the stack, $r_1 \in \mathfrak{N}_\Sigma$, and $\mathsf{hdf}(r_2)$ is pushed, assume again that r is the corresponding input expression. From the induction hypothesis there is a p such that $\mathsf{iterPos}_r(r_1 r_2) = p$, $r_1 = r[p]$, and $r_2 = \mathsf{hdf}(r[\mathsf{next}_r^*(p)])$. If $\mathsf{next}_r(p) = \chi$, then $r_2 = \epsilon$ and the lemma holds vacuously for $\mathsf{hdf}(r_2) = \epsilon$. Otherwise, $r_2 = \mathsf{hdf}(r[\mathsf{next}_r(p)] \cdot r[\mathsf{next}_r^*(\mathsf{next}_r(p))])$. Applying Lemma 2.4.8 to r and $\mathsf{next}_r(p)$ gives $q, p_1, \ldots, p_n$ such that $\mathsf{hdf}(r[\mathsf{next}_r(p)]) = r[q] \cdot \mathsf{hdf}(r[p_1] \cdots r[p_n] \cdot \epsilon)$ and $\mathsf{next}_r^*(q) = p_1 \cdots p_n \cdot \mathsf{next}_r^*(\mathsf{next}_r(p))$. Applying Corollary 2.4.7 to this we get $\mathsf{hdf}(r[\mathsf{next}_r(p)] \cdot r[\mathsf{next}_r^*(\mathsf{next}_r(p))]) = r[q] \cdot \mathsf{hdf}(r[p_1] \cdots r[p_n] \cdot r[\mathsf{next}_r^*(\mathsf{next}_r(p))]) = r[q] \cdot \mathsf{hdf}(r[\mathsf{next}_r^*(q)])$. Thus $\mathsf{hdf}(r_2) = r[q] \cdot \mathsf{hdf}(r[\mathsf{next}_r^*(q)])$, and we can set $\mathsf{iterPos}_r(r_2) = q$.

□

Note now that the mappings $\mathsf{iterPos}_{r_l}$ and $\mathsf{iterPos}_{r_r}$ are injective. We show this by letting $r \in \{r_l, r_r\}$, and assuming that for two regular expressions r_1 and r_2 occurring on the same side as r in two pairs in the stack, we have $\mathsf{iterPos}_r(r_1) = \mathsf{iterPos}_r(r_2) = p$ for some p. But from Lemma 2.4.9 we then have $r_1 = r_2 = r[p] \cdot \mathsf{hdf}(r[\mathsf{next}_r^*(p)])$. So the mapping $\mathsf{iterPos}_r$ is injective.

We are now done with showing termination and polynomial runtime, since Lemma 2.4.9 implies that the product of the number of positions in the two regular expressions given as input is an upper bound to the number of pairs of members from $R_\Sigma - \{\epsilon\}$ occurring in the stack. The latter number is exactly the number of iterations of the main loop where new pairs are pushed to the stack, since an ϵ on the left-hand side can only be matched by (Axm) and if only the right-hand side is ϵ, this leads to a "No" answer. As argued in the beginning of this section, this means the run-time of the whole algorithm is polynomial.

2.5 Soundness and Completeness

We need some auxiliary definitions and lemmas before we can prove soundness.

Definition 2.5.1 (Execution graph)**.** *An execution graph is a directed graph representing a successful run of the algorithm. The nodes correspond to the iterations of the main loop in the algorithm where the test "$(r_1, r_2) \in \mathsf{S}$" fails. Each node is labeled by the name and conclusion of the rule instance matching the pair popped from the stack in the corresponding iteration. There is an edge from each node to the node(s) labeled with the premise(s) of the rule instance applied in the corresponding iteration.*

Every usage of the store corresponds to a loop in the graph. Note that the only nodes without outgoing edges in an execution graph, are those labeled (Axm).

Example 2.5.2. *Figure 2.5 shows the execution graph corresponding to a run of the algorithm with input $a^*b^*, (a+b)^*$.*

Let the *size* of a regular expression be the sum of the number of letters and operators $*$ and $+$ occurring in the expression. Note that the concatenation operator and ϵ are not counted. We will label an edge in an execution graph as *left-increasing* or *right-increasing*, respectively, if the left-hand or right-hand expression labeling the start node has smaller size than the corresponding expression in the end node. *Left-decreasing* and *right-decreasing* labels are defined similarly.

Nodes labeled (StarChoice2) and (LeftStar) have one left-increasing and left-decreasing outgoing edge. An edge is right-increasing if and only if it starts in a node labeled (LetterStar). Outgoing edges from all other rules are left-decreasing, right-decreasing, or both. If an edge is neither left-increasing nor left-decreasing then the expression on the

$$\begin{tikzcd}
(\mathsf{LeftStar}), a^*b^* \sqsubseteq (a+b)^* \arrow[r] \arrow[d] & (\mathsf{LetterStar}), aa^*b^* \sqsubseteq (a+b)^* \arrow[r] & (\mathsf{LetterChoice}), aa^*b^* \sqsubseteq (a+b)(a+b)^* \arrow[r] & (\mathsf{Letter}), aa^*b^* \sqsubseteq a(a+b)^* \arrow[lll, bend left] \\
(\mathsf{LeftStar}), b^* \sqsubseteq (a+b)^* \arrow[r] \arrow[d] & (\mathsf{LetterStar}), bb^* \sqsubseteq (a+b)^* \arrow[r] & (\mathsf{LetterChoice}), bb^* \sqsubseteq (a+b)(a+b)^* \arrow[r] & (\mathsf{Letter}), bb^* \sqsubseteq b(a+b)^* \arrow[lll, bend right] \\
(\mathsf{Axm}), \epsilon \sqsubseteq (a+b)^* & & &
\end{tikzcd}$$

Figure 2.5: The execution graph corresponding to input a^*b^*, $(a+b)^*$. (cf. Fig. 2.2).

left-hand side in the start and end node are the same. The similar statement holds for the right-hand side. The edges corresponding to usage of the store S have no labels, since by construction the expressions in the start and end node are the same.

Lemma 2.5.3. *If there is a left-increasing edge in a loop, then there is also a node labeled* (Letter) *in the loop.*

Proof. We prove a stronger statement, which implies that any path starting with a left-increasing edge, and not containing a node labeled (Letter) cannot be a loop: We show that in a path where there is no node labeled (Letter) and which starts with a left-increasing edge, the left-hand expressions in all nodes except the first node are of the form $r'_1 \cdots r'_n \cdot r_1^* r_2$ for some $r'_1 \cdots r'_n \notin \mathfrak{N}_\Sigma$. This is proved by induction on the length of the path. For the base case, only one edge in the path, note that the start node must be labeled (StarChoice2) or (LeftStar), so the left-hand expression in the first node is of the form $r_1^* r_2$ and the left-hand expression in the last node is $r_1 r_1^* r_2$. Now, $r_1 \notin \mathfrak{N}_\Sigma$ follows from the fact that the expressions are in star normal form.

There is an induction case for each premise of each rule. Note that (Axm) cannot be applied because of the induction hypothesis. For the premises corresponding to edges which are neither left-decreasing nor left-increasing, the left-hand expression is unchanged, and we can just use the induction hypothesis. For the premises corresponding to left-increasing edges, note that by the induction hypothesis the start node is of the form $r'_1 \cdots r'_n \cdot r_1^* r_2$, and the left-hand expression in the last node is $r' \cdot r'_1 \cdots r'_n \cdot r_1^* r_2$ for some r', and $r'_1 \cdots r'_n \notin \mathfrak{N}_\Sigma \Rightarrow r' \cdot r'_1 \cdots r'_n \notin \mathfrak{N}_\Sigma$.

The interesting cases are the premises corresponding to left-decreasing edges. For (LeftChoice), we can apply the induction hypothesis to get that the left-hand expression in the start node is $(r'_1 + r'_2) \cdot r'_3 \cdots r'_n \cdot r_1^* r_2$ where $(r'_1 + r'_2) \cdot r'_3 \cdots r'_n \notin \mathfrak{N}_\Sigma$. The last node has left-hand expression $r'_i \cdot r'_3 \cdots r'_n \cdot r_1^* r_2$ for $i \in \{1, 2\}$. But $(r'_1 + r'_2) \cdot r'_3 \cdots r'_n \notin \mathfrak{N}_\Sigma \Rightarrow r'_i \cdot r'_3 \cdots r'_n \notin \mathfrak{N}_\Sigma$, so the lemma holds also for the new last node.

For a left-decreasing edge, corresponding to a premise of (StarChoice2) or (LeftStar), we get from the induction hypothesis and Definition 2.2.3 that the left-hand expression in the starting node is of the form ${r'_1}^* \cdot r'_2 \cdots r'_n \cdot r_1^* r_2$ where $r'_2 \cdots r'_n \notin \mathfrak{N}_\Sigma$. The left-hand expression in the last node is $r'_2 \cdots r'_n \cdot r_1^* r_2$, so the lemma holds also for this case. □

Lemma 2.5.4. *If there is a right-increasing edge in a loop, then there is also an instance of* (Letter) *in the loop.*

Proof. We prove a stronger statement which implies that any path starting with a right-increasing edge, and not containing a node labeled (Letter) cannot be a loop: In a path where there is no node labeled (Letter) and which starts with a right-increasing edge, all nodes except the first are of the form $l \cdot r_1 \sqsubseteq r'_1 \cdots r'_n \cdot r_2^* r_3$ for some $r'_1 \cdots r'_n \notin \mathfrak{N}_\Sigma$ where $l \in \mathsf{first}(r'_1 \cdots r'_n)$. This is proved by induction on the length of the path.

For the base case, only one edge in the path, the first node must be labeled with (LetterStar) and $l \cdot r_1 \sqsubseteq r_2^* r_3$. The last node is then labeled $l \cdot r_1 \sqsubseteq r_2 r_2^* r_3$. That $r_2 \notin \mathfrak{N}_\Sigma$ follows from that the expressions are in star normal form. $l \in \mathsf{first}(r_2)$ is the side conditions on (LetterStar).

There are induction cases for all rules, but the only rules that can match the relation are (LetterStar), (LetterChoice) and (ElimCat).

For (LetterStar), the conclusion must be of the form

$$l \cdot r_1 \sqsubseteq {r'_1}^* r'_2 \cdots r'_n \cdot r_2^* r_3$$

where $r'_2 \cdots r'_n \notin \mathfrak{N}_\Sigma$. From the side-condition we get $l \in \mathsf{first}(r'_1)$. Thus the lemma holds for the premise $l \cdot r_1 \sqsubseteq r'_1 {r'_1}^* r'_2 \cdots r'_n r_2^* r_3$.

For the cases matching (LetterChoice) the conclusion is of the form $l \cdot r_1 \sqsubseteq (r'_1 + r'_2) r'_3 \cdots r'_n r_2^* r_3$, and the premise is $l \cdot r_1 \sqsubseteq r'_i \cdot r'_3 \cdots r'_n r_2^* r_3$ where $i \in \{1, 2\}$, $l \in \mathsf{first}(r'_1)$ and $r'_i \cdot r'_3 \cdots r'_n \notin \mathfrak{N}_\Sigma$.

For (ElimCat), the conclusion is of the form $l \cdot r_1 \sqsubseteq r'_1 \cdots r'_n \cdot r_2^* r_3$ for some $r'_1 \in \mathfrak{N}_\Sigma$, where $r'_1 \cdots r'_n \notin \mathfrak{N}_\Sigma$. The latter implies that $r'_2 \cdots r'_n \notin \mathfrak{N}_\Sigma$, and the side-conditions ensure that $l \in \mathsf{first}(r'_2 \cdots r'_n \cdot r_2^* r_3)$, which imply that $l \in \mathsf{first}(r'_2 \cdots r'_n)$. So the lemma also holds for the premise $l \cdot r_1 \sqsubseteq r'_2 \cdots r'_n \cdot r_2^* r_3$. □

Lemma 2.5.5. *In any loop, there is at least one instance of* (Letter)

Proof. At least one rule instance in a loop is right- or left-increasing or -decreasing. This implies there must be at least one left- or right-increasing instance, and the result follows immediately from Lemmas 2.5.3 and 2.5.4. □

Definition 2.5.6 (Letter-path)**.** *A* letter-path *is a path in an execution graph of the algorithm where the last node is labeled* (Letter) *and there are no other nodes labeled* (Letter),

Lemma 2.5.7 (Letter-path language conservation)**.** *In every letter-path, if the last node is labeled $lr_1 \sqsubseteq lr_2$ and the first node is labeled $r_3 \sqsubseteq r_4$, then $\|r_2\| \subseteq \{w \mid lw \in \|r_4\|\}$*

Proof. By induction on the length of the letter-path.

The base case is a path consisting of a single node labeled (Letter). This case is immediate, as we get $r_4 = l \cdot r_2$.

There are induction cases for each of the rules shown in Tables 2.2 and 2.3, except (Axm) and (Letter). The cases where the right-hand expression is unchanged ((LeftChoice), (LeftStar), and (StarChoice2)) hold immediately from the induction hypothesis.

For (LetterStar), the right-hand expression in the label of the first node is of the form $r_5^* r_6$ and the induction hypothesis is that $\|r_2\| = \{w \mid lw \in \|r_5 r_5^* r_6\|\}$. The inclusion $\|r_5 r_5^* r_6\| \subseteq \|r_5^* r_6\|$ follows from Definition 2.2.2, thus we also get that $\{w \mid lw \in \|r_5 r_5^* r_6\|\} \subseteq \{w \mid lw \in \|r_5^* r_6\|\}$, so the lemma holds.

For (LetterChoice) and (StarChoice1), the right-hand expression in the conclusion is of the form $(r_5 + r_6)r_7$, and by symmetry we can assume the right-hand expression in the premise is $r_5 r_7$, so the induction hypothesis is that $\|r_2\| \subseteq \{w \mid lw \in \|r_5 r_7\|\}$. But since $\|r_5 r_7\| \subseteq \|(r_5 + r_6)r_7\|$ follows from Definition 2.2.2 we also get that $\{w \mid lw \in \|r_5 r_7\|\} \subseteq \{w \mid lw \in \|(r_5 + r_6)r_7\|\}$, so the lemma holds.

For (ElimCat), the right-hand expression in the conclusion is of the form $r_5 r_6$ where $r_5 \in \mathfrak{N}_\Sigma$, and the induction hypothesis is that $\|r_2\| \subseteq \{w \mid lw \in \|r_6\|\}$. But since $\|r_6\| \subseteq \|r_5 r_6\|$ follows from Definition 2.2.2 and $r_5 \in \mathfrak{N}_\Sigma$, we also get that $\{w \mid lw \in \|r_6\|\} \subseteq \{w \mid lw \in \|r_5 r_6\|\}$, so the lemma holds. □

Lemma 2.5.8. *For any node $r_1 \sqsubseteq r_2$ in an execution graph, and for any $w \in \|r_1\|$, $w \neq \epsilon$, there is a letter-path from this node to an instance of* (Letter) *such that w is in the language of the left-hand expression in the conclusion of this instance of* (Letter)*.*

Proof. For all rules, except (Letter), the union of the languages of the left-hand expressions in the premise(s) equals the language of the left-hand expression in the conclusion. We can therefore construct the letter-path by repeatedly choosing the next node corresponding to a premise where the left-hand expression matches w. This process will terminate in an instance of (Letter) by the following arguments. Instances of (Axm) will not occur as $w \notin \|\epsilon\|$, and Lemma 2.5.5 assures that all loops contain at least one instance of (Letter). □

Lemma 2.5.9. *For any $r_1 \sqsubseteq r_2$ in an execution graph of the algorithm, $\|r_1\| \subseteq \|r_2\|$.*

Proof. The lemma can be reformulated, stating that for all $w \in \Sigma^*$, and all $r_1 \sqsubseteq r_2$ in the execution graph, $w \in \|r_1\|$ implies $w \in \|r_2\|$. We prove this simultaneously for all nodes in the execution graph, by induction on the length of w. The base case is that $w = \epsilon$. In this case $r_1 \in \mathfrak{N}_\Sigma$, and the algorithm guarantees that also $r_2 \in \mathfrak{N}_\Sigma$. The induction case is that $w = lw'$ for some $l \in \Sigma$ and $w' \in \Sigma^*$. Assume some $r_1 \sqsubseteq r_2$ in the execution graph, where $lw' \in \|r_1\|$. We must prove that $lw' \in \|r_2\|$. From Lemma 2.5.8 there is a letter-path starting with the instance with conclusion $r_1 \sqsubseteq r_2$, and ending in an instance of (Letter) with conclusion $lr_3 \sqsubseteq lr_4$ such that $w' \in \|r_3\|$. From using the induction hypothesis on w' and the premiss $r_3 \sqsubseteq r_4$ of this instance of (Letter) we then get that $w' \in \|r_4\|$, and therefore $w = l \cdot w' \in \|lr_4\|$. Lemma 2.5.7 now states that $\|r_4\| \subseteq \{v \mid lv \in \|r_2\|\}$, so we get that $w \in \|r_2\|$. □

Soundness is now an immediate corollary of the previous lemma.

Theorem 2.5.10 (Soundness)**.** *Let r_1, r_2 be regular expressions. If the algorithm is run with r_1 and r_2 as input, and returns "Yes", then $\|r_1\| \subseteq \|r_2\|$.*

Proof. Since the input is r_1 and r_2 we know that $r_1 \sqsubseteq r_2$ occurs in the corresponding execution graph. From Lemma 2.5.9 we then get that $\|r_1\| \subseteq \|r_2\|$. □

Since the rules are invertible, and, as seen above, the algorithm always terminates, we get completeness almost for free.

Theorem 2.5.11 (Completeness)**.** *If $\|r_1\| \subseteq \|r_2\|$, the algorithm will either accept $r_1 \sqsubseteq r_2$, or it will report that the 1-ambiguity of r_2 is a problem.*

Proof. Since the rules are invertible, and the algorithm always terminates, all that remains is to show that for all regular expressions r_1 and r_2, where their languages are in an inclusion relation, there is at least one rule instance with conclusion $r_1 \sqsubseteq r_2$. But this follows directly from Lemma 2.4.1. □

2.6 Related Work and Conclusion

This chapter is an extension of work in [38]. Martens, Neven & Schwentick study in [50] the complexity of the inclusion problem for several sub-classes of the regular expressions. Colazzo, Ghelli & Sartiani, describe in [18] and [26] asymmetric polynomial-time algorithms for inclusion of a subclass of regular expressions called collision-free. The collision-free regular expressions have at most one occurrence of each symbol from Σ, and the Kleene star can only be applied to disjunctions of letters. The latter class is strictly included in the class of 1-unambiguous regular expressions. The main focus of Colazzo, Ghelli and Sartiani is on the extensions of regular expressions used in XML Schemas. These extensions are not covered by the algorithm presented here. Hosoya et al. [35] study the inclusion problem for XML Schemas. They also use a syntax-directed inference system, but the algorithm is not polynomial-time. Salomaa [63] presents two axiom systems for equality of regular expressions, but does not treat the run-time. The inference system used by our algorithm has some inspiration from the concept of derivatives of regular expressions, first defined by Brzozowski [13]. The first use of derivatives for the inclusion problem is by Brzozowski in [14]. Antimirov reinvents and details this approach in [2], as a term rewriting system for inequalities of regular expressions. Chen & Chen [16] adopt Antimirov's algorithm to the inclusion problem for 1-unambiguous regular expressions. They do not treat the left-hand and right-hand together in the way the rules of the algorithm in this chapter do. The analysis of their algorithm depends on both the left-hand and the right-hand regular expressions being 1-unambiguous.

2.6.1 Conclusion

We have described a polynomial-time algorithm for language inclusion of regular expressions. The algorithm is based on a syntax-directed inference system, and is guaranteed to give the correct answer if the right-hand expression is 1-unambiguous. If the right-hand expression is 1-ambiguous the algorithm either reports an error or gives the correct answer. In certain cases, irrelevant parts of the right-hand expression are automatically discarded. This is the main advantage over the classical algorithms for inclusion. An implementation of the algorithm is available on the author's website.

3 The Membership Problem for Regular Expressions with Numerical Constraints and Unordered Concatenation

We study the membership problem for regular expressions extended with operators for *unordered concatenation* and *numerical constraints*. Unordered concatenation is used in the ISO standard for the Standard Generalized Markup Language (SGML), the precursor of XML. XML Schema uses a very limited form of unordered concatenation. Numerical constraints are an extension of regular expressions used in many applications, e.g. text search (e.g., UNIX grep), document formats (e.g. XML Schema). Regular expressions with unordered concatenation and numerical constraints denote the same languages as the classical regular expressions, but, in certain important cases, exponentially more succinct. We show that the membership problem for regular expressions with unordered concatenation (without numerical constraints) is already NP-hard. Kilpeläinen & Tuhkanen have in [42] shown that the membership problem for regular expressions with numerical constraints is in P. We also show a polynomial-time algorithm for the membership problem for regular expressions with numerical constraints and unordered concatenation, when restricted to a subclass called *strongly 1-unambiguous*.

3.1 Introduction

In the ISO standard for the Standard Generalized Markup Language (SGML) [1], the precursor of XML, the operator "&" is used for what in this chapter is called *unordered concatenation*, that is, the languages are concatenated, but in any order. For example, $\&(ya, basta)$ denotes $\{yabasta, bastaya\}$. In SGML "&" is infix, but because it is not associative and not binary, we find it more clear to write it prefix. Brüggemann-Klein [11, 15] investigates unambiguity of regular expressions extended with such an unordered concatenation operator. XML Schema [21] uses a restricted form of unordered concatenation, called `all`.

Numerical constraints allow expressing that a subexpression must be matched a number of times specified by a lower and a upper bound. For example, $(a+b)^{2..3}$ denotes the words of length 2 or 3 consisting only of a's and b's. Numerical constraints are used in XML Schema, and also in applications for text search, e.g. GNU `grep`. The Single UNIX Specification [59] requires this as a standard part of regular expressions. In the GNU version of the UNIX program grep [30] and in the programming language Perl they are included as standard and in XML Schema [21] the 1-unambiguous subclass is allowed. In GNU grep you can, for example, write `([0-9]{1,3}\.){3}[0-9]{1,3}` to match any IPv4 address in dotted-decimal notation. Regular expressions with numerical constraints has been studied by, among others, Meyer & Stockmeyer [54], Sperberg-McQueen [64], Kilpeläinen & Tuhkanen [42, 43], Gelade et al. [24, 25], Gelade [22], Ghelli et al. [27], Gelade et al.[23], and Hovland [37]

Common uses of regular expressions with numerical constraints and/or unordered concatenation are matching and searching. With matching we mean the membership problem, the problem of deciding whether a given word is in the language defined by the regular expression. Searching means to decide whether one or more sub-strings of a given text match the regular expression. For an overview of searching and matching with classical regular expressions, see Navarro & Raffinot [56, Chapter 5]. Kilpeläinen & Tuhkanen [42] showed that for the regular expressions with numerical constraints, matching can be done with a polynomial-time dynamic programming algorithm. Using this algorithm, one can also search in polynomial time. Unfortunately, their algorithm uses space quadratic in the length of the word being matched. For many real-world applications the word is very long, and quadratic space in the word can be too much.

Many programs that search using regular expressions with numerical constraints use algorithms with super-polynomial behavior in the length of the regular expression. These programs typically have as input one short regular expression and many, long, texts to be searched.

It is therefore common to construct a deterministic finite automaton (DFA) for matching or searching. By using the string $\Sigma^* r \Sigma^*$, a DFA can be used to search in time linear in the length of the text. A quadratic algorithm is usually preferred, as it is faster in most practical cases. Meyer & Stockmeyer [54] show that the inclusion problem for regular expressions with squaring is EXPSPACE-complete. Furthermore, the problem of deciding whether the languages of two arbitrary DFAs are in the inclusion relation is in P. Therefore, any general method for constructing a DFA recognizing the language of a regular expression with numerical constraints must use superpolynomial space in the worst case.

As an example, consider an experiment lasting 100 hours, where we need to record the moments at which some (unspecified) events take place. We will use one string to describe each 100-hour experiment. For each hour when there is an event, the hour is given, followed by "h", followed by a string describing the events occurring that hour. This string is formatted in the following way: for each minute when there is an event, the minute is given, followed by "m", followed by the second and "s" for each second at which there was an event during that minute. If there were, e.g., a total of three events during one experiment, at 3:12:22, 3:12:43 and 20:45:01, then the string describing the experiment is `3h12m22s43s20h45m1s`. For testing the strings we decide to use the regular expression $((0+\cdots+9)^{1..2}h((1+\cdots+5)^{0..1}(0+\cdots+9)m((1+\cdots+5)^{0..1}(0+\cdots+9)s)^{1..60})^{1..60})^{0..100}$ by executing the command in Fig. 3.1 (See next section for syntax and semantics of the regular expressions). However, this command turns out to use over 2 gigabytes of memory[1], independent of the length of the text to be matched.

```
grep -E \
"([0-9]{1,2}h([1-5]?[0-9]m([1-5]?[0-9]s){1,60}){1,60}){0,100}"
```

Figure 3.1: Example invocation of grep

An algorithm for the matching problem will be called a *fast-matcher*, if there is a constant c such that the algorithm runs in time $O(|r|^c \cdot |w|)$

[1]Measurements done with procps version 3.2.7 running GNU grep version 2.5.3 compiled with GNU cc version 4.1.2 on a machine with four 2,0 GHz 32-bit CPUs running CentOS-5.2 with Linux 2.6.18 and GNU C library version 2.5.

(where r is the regular expression and w is the word to be matched). There exists a fast-matcher for the classical regular expressions (without numerical constraints). The algorithm constructs a non-deterministic finite automaton (NFA) recognizing the regular expression, and runs the NFA on the word by maintaining the set of possible states. The latter set is limited by the size of the NFA, and the number of steps is exactly the length of the word. Construction of an NFA recognizing the language of a regular expression is possible in polynomial time. The downside of this algorithm is that each step in the matching is not constant-time. Brüggemann-Klein [10] describes a different fast-matcher for a subset of the regular expressions, called 1-unambiguous regular expressions. Their algorithm constructs in polynomial time a deterministic finite automaton from a 1-unambiguous regular expression. However, no polynomial-time construction of an NFA is known for 1-unambiguous regular expressions with numerical constraints. Furthermore, a polynomial-time construction of DFAs from 1-unambiguous regular expressions with numerical constraints would imply P=NP, since Kilpeläinen [41] has shown that the inclusion problem is NP-hard for these expressions.

In this article we describe the *finite automata with counters*, and a fast-matcher for a subset of the regular expressions with numerical constraints and unordered concatenation, called *strongly 1-unambiguous regular expressions*. The algorithm works by constructing deterministic finite automata with counters from these expressions. The algorithm, but without support for unordered concatenation, has been implemented[2] in `C` in a manner inspired by grep. The command in Fig. 3.1 executed with our implementation on the same machine uses less memory by three orders of magnitude.

Concerning unordered concatenation, this chapter has a theoretical and a more practical motivation. The theoretical motivation is curiosity about the properties of the SGML-style unordered concatenation operator. The operator is intuitive and seems useful for searches and definitions in natural language text. The practical motivation concerns the `all` operator from XML Schema. The operator can be seen as unordered concatenation where the arguments are restricted to being either a single letter or a choice between the empty word and a single letter. In light of the results from this chapter, some of the restrictions

[2]Available from `http://www.ii.uib.no/~dagh/fac`

on the use of this operator seem unnecessary. More specifically, *membership* is shown to be tractable for the *strongly 1-unambiguous* regular expressions with unordered concatenation *and* numerical constraints.

It is also possible to see the `all` operator as a restricted form of what is usually called *shuffling* or *interleaving*[3]. Ghelli et al. [27] have studied the membership problem for this extension together with numerical constraints. They obtain linear runtime, but they must restrict their scope to a very limited subclass, where for example the starred subexpressions must be disjunctions of letters. Shuffling has been studied by Mayer & Stockmeyer [51] and Ogden et al. [58], and membership shown to be intractable when unrestricted use of the parallel interleaving operator is allowed together with choice and Kleene star.

In this chapter we will study the regular expressions with unordered concatenation and numerical constraints, and the membership problem for these expressions. In the next section we give a definition of these expressions and their languages, and in Section 3.3 we show that the membership problem is NP-complete already without numerical constraints. In Section 3.4 we define the finite automata with counters. In Section 3.5 we define subscripting, some auxiliary lemmas, and strong 1-unambiguity. In Section 3.6 we define the mappings first, last, and follow, which are central to the construction of finite automata with counters given in Section 3.7. In Section 3.7 we show that finite automata with counters can be constructed in polynomial time from regular expressions with unordered concatenation and numerical constraints. If the expression is strongly 1-unambiguous, the construction leads to a deterministic automaton. The last section presents some related work and a conclusion.

3.2 Regular Expressions with Unordered Concatenation and Numerical Constraints

Fix an *alphabet* Σ of *letters*. Assume a, b, and c are members of Σ. $l, l_1, l_2, \ldots$ are used as variables for members of Σ. Let $\mathbb{N} = \{1, 2, \ldots\}$, $\mathbb{N}_1 = \{2, 3, 4, \ldots\} \cup \{\infty\}$, and $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$. We define that $n < \infty$ for all numbers n.

[3]Email communication with C. M. Sperberg-McQueen: `http://lists.w3.org/Archives/Public/xmlschema-dev/2009May/0063.html`

Definition 3.2.1. *Given an alphabet Σ, R_Σ is the set of* regular expressions with unordered concatenation and numerical constraints *over Σ, defined by the following grammar:*

$$R_\Sigma ::= R_\Sigma + R_\Sigma \mid R_\Sigma \cdot R_\Sigma \mid R_\Sigma^{\mathbb{N}..\mathbb{N}_1} \mid \&(R_\Sigma, \ldots, R_\Sigma) \mid \Sigma \mid \epsilon$$

We only allow $r^{l..u}$ for $l \leq u$. Parentheses are used, when necessary, to group sub-expressions. We use $r, r_1, r_2, \ldots$ as variables for regular expressions. The sign for concatenation, $\cdot$, will often be omitted. A regular expression denoting the empty language is not included, as this is irrelevant to the results in this chapter. We use $r^{l..}$ as shorthand for $r^{l..\infty}$, $r^{0..n}$ as shorthand for $r^{1..n} + \epsilon$, r^+ as shorthand for $r^{1..\infty}$, r^* as shorthand for $r^{0..}$, and r^n for $r^{n..n}$. We denote the set of letters from Σ occurring in r by $\mathsf{sym}(r)$.

The reason that the unordered concatenation operator is not binary infix, is that, as we will see below, it is not associative. The *star-free regular expressions with unordered concatenation* are the subset of R_Σ with no numerical constraints, that is, no subexpressions of the form $r^{l..u}$.

We use similar definitions of positions in term trees and concatenation of positions as in Section 2.2.1. The only difference is that the term trees have some new operators.

Definition 3.2.2 (Marked Expressions). *If $r \in R_\Sigma$ is a regular expression, $\mu(r) \in R_{\mathsf{pos}(r)}$ is the marked expression, defined in the following inductive manner:*

- $\mu(\epsilon) = \epsilon$
- *for $l \in \Sigma$, $\mu(l) = \langle\rangle$*
- $\mu(r_1 + r_2) = 1\mu(r_1) + 2\mu(r_2)$
- $\mu(r_1 \cdot r_2) = 1\mu(r_1) \cdot 2\mu(r_2)$
- $\mu(r_1^{l..u}) = (1\mu(r_1))^{l..u}$
- $\mu(\&(r_1, \ldots, r_n)) = \&(1\mu(r_1), \ldots, n\mu(r_n))$

Example 3.2.3. *Consider $\Sigma = \{a, b\}$ and $r = (\&(a^2, b))^{3..4}$. Then $\mu(r) = (\&(\langle 1,1,1\rangle^2, \langle 1,2\rangle))^{3..4}$. The term trees of r and $\mu(r)$ are shown in Figs. 3.2 and 3.3, respectively.*

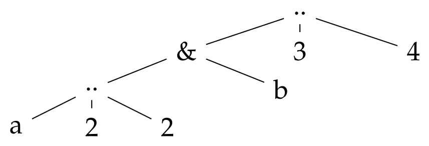


Figure 3.2: Term tree for $(\&(a^2, b))^{3..4}$

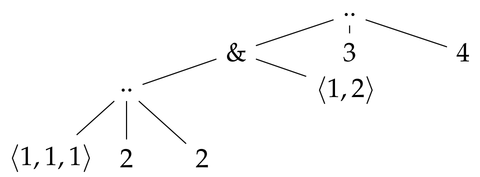


Figure 3.3: Term tree for $\mu((\&(a^2, b))^{3..4})$

We lift concatenation of words to sets of words, such that if $L_1, L_2 \subseteq \Sigma^*$, then

$$L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}.$$

Moreover, ϵ denotes the *empty word* of zero length, such that for all $w \in \Sigma^*$, $\epsilon \cdot w = w \cdot \epsilon = w$. Further, we allow non-negative integers as exponents meaning repeated concatenation, such that for an $L \subseteq \Sigma^*$, we have $L^n = L^{n-1} \cdot L$ for $n > 0$ and $L^0 = \{\epsilon\}$. The semantics of unordered concatenation is defined in terms of permutations. By $\mathsf{Perm}(\{1, \ldots, n\})$ we mean the set of permutations of $\{1, \ldots, n\}$. If $\sigma \in \mathsf{Perm}(\{1, \ldots, n\})$, we assume $\sigma = \sigma_1, \ldots, \sigma_n$. For convenience, we recall in Definition 3.2.4 the language denoted by a regular expression, and extend it to unordered concatenation and numerical constraints.

Definition 3.2.4 (Language)**.** *The language $\|r\|$ denoted by a regular ex-*

pression $r \in R_\Sigma$, is defined in the following inductive way:

$$\begin{aligned}
\|r_1 + r_2\| &= \|r_1\| \cup \|r_2\| \\
\|r_1 \cdot r_2\| &= \|r_1\| \cdot \|r_2\| \\
\|\&(r_1, \dots, r_n)\| &= \bigcup_{\sigma \in \mathsf{Perm}(\{1,\dots,n\})} \|r_{\sigma_1}\| \cdots \|r_{\sigma_n}\| \\
\|r^{l..u}\| &= \bigcup_{i=l}^{u} \|r\|^i \\
\textit{for } a \in \Sigma \cup \{\epsilon\}, \|a\| &= \{a\}
\end{aligned}$$

All subexpressions of the forms $\epsilon \cdot r$, $r \cdot \epsilon$, $\epsilon + \epsilon$, $\epsilon^{l..u}$, or

$$\&(r_1, \dots, r_i, \epsilon, r_{i+1}, \dots, r_n)$$

can be simplified to r, r, ϵ, ϵ, or $\&(r_1, \dots, r_n)$ respectively, in linear time, working bottom up. We will often tacitly assume there are no subexpressions of these forms.

Some examples of regular expressions and their languages are: $\|\&(ab, c)\| = \{abc, cab\}$ and $\|\&(a, b, c)\| = \{abc, bac, acb, bca, cab, cba\}$. Note that unordered concatenation is not associative, for example: $\|\&(\&(a, b), c)\| = \{abc, bac, cab, cba\} \neq \{abc, acb, bca, cba\} = \|\&(a, \&(b, c))\|$.

Remark 3.2.5. *The language of any regular expression with unordered concatenation is also the language of some regular expression without unordered concatenation, since unordered concatenation can be translated to choice by exploiting the following equality: $\&(r_1, \dots, r_n) = r_1 \cdot \&(r_2, \dots, r_n) + \cdots + r_n \cdot \&(r_1, \dots, r_{n-1})$. This translation gives a super-polynomial blowup of the expression. The existence of a translation without a super-polynomial increase in size would imply that NP=P, since the membership problem for the classical regular expressions is in P, while we will show that the problem is NP-complete for regular expressions with unordered concatenation.*

The above translation does not preserve 1-unambiguity, and in fact, Brüggemann-Klein noted in [15] that there are 1-unambiguous regular expressions with unordered concatenation which denote languages that are not denoted by any 1-unambiguous regular expressions without unordered concatenation.

3.3 Complexity of Membership under Unordered Concatenation

For regular expressions with numerical constraints (without unordered concatenation), the membership problem is known to be in P [42]. In this section we treat only the star-free regular expressions with unordered concatenation. For the remainder of this section, "regular expression" will mean "star-free regular expression with unordered concatenation". The usage of the exponents in the expressions and words in this section is only a short-hand for repeated concatenation. The *membership*-problem is to decide, given a regular expression with unordered concatenation $r \in R_\Sigma$, and a word $w \in \Sigma^*$, whether $w \in \|r\|$. This is also called *matching*.

The fact that the above membership is in NP is not hard to see. The certificate for an instance of the problem, consists in making all the necessary choices in the regular expression, such that one can see that the word is in the language. The size of the certificate is polynomial in the lengths of the word and the regular expression.

3.3.1 Membership is in NP

In this section we give the construction of the polynomial-size certificate for membership of star-free regular expressions with unordered concatenation. The principles of this construction seem to be folklore. This certificate is needed for the standard proof of a problem being in NP. Given a word w and a regular expression r, the polynomial-size certificate $\mathsf{cert}(r,w)$ certifying that $w \in \|r\|$ is a list of numbers. We use the same notations for lists of numbers as for positions. $\mathsf{cert}(r,w)$ is defined by induction on header-form of r

$$\begin{aligned}
\mathsf{cert}(\epsilon,\epsilon) &= \langle\rangle. \\
\mathsf{cert}(l \cdot r, lw) &= \mathsf{cert}(r,w), \text{ where } l \in \Sigma. \\
\mathsf{cert}((r_1+r_2)r_3,w) &= \begin{cases} 1\mathsf{cert}(r_1 \cdot r_3, w), & \text{if } w \in \|r_1 \cdot r_3\| \\ 2\mathsf{cert}(r_2 \cdot r_3, w), & \text{if } w \in \|r_2 \cdot r_3\|. \end{cases} \\
\mathsf{cert}(\&(r_1,\dots,r_n)\cdot r', w) &= i\mathsf{cert}(r_i \cdot \&(r_1,\dots,r_{i-1},r_{i+1},\dots,r_n)\cdot r', w), \\
&\quad \text{where } 1 \le i \le n \text{ and} \\
&\quad w \in \|r_i \cdot \&(r_1,\dots,r_{i-1},r_{i+1},\dots,r_n)\cdot r'\|.
\end{aligned}$$

That $\mathsf{cert}(r, w)$ is defined for all w and r whenever $w \in \|r\|$ is proved by an easy induction on r and using Definition 3.2.4.

The list $\mathsf{cert}(r, w)$ consists of numbers which are no larger than the logarithm of the largest number of parameters to any unordered concatenation in r. The number of elements in $\mathsf{cert}(r, w)$ is at most the product of the length of w and the sum of $+$'s and &'s in r.

To use the certificate $\mathsf{cert}(r, w)$ to check that $w \in \|r\|$ we proceed in the following inductive manner, assuming that r is in header form:

- If $r = \epsilon$ then we must have $w = \epsilon$ which is easy to check.

- If $r = l \cdot r_1$ for some $l \in \Sigma$, then we must have $w = l \cdot w_1$ and $\mathsf{cert}(r, w) = \mathsf{cert}(r_1, w_1)$. We can use the method recursively to check that $\mathsf{cert}(r_1, w_1)$ certifies that $w_1 \in \|r_1\|$.

- If $r = (r_1 + r_2)r_3$ then $\mathsf{cert}(r, w) = i\mathsf{cert}(r_i \cdot r_3, w)$ for $i \in \{1, 2\}$, so we can use the method recursively to check that $\mathsf{cert}(r_i \cdot r_3, w)$ certifies that $w \in \|r_i r_3\|$.

- If $r = \&(r_1, \ldots, r_n)r'$, then there is an $i \in \{1, \ldots, n\}$ such that $w \in \|r_i \cdot \&(r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n)r'\|$ and

$$\mathsf{cert}(r, w) = i\mathsf{cert}(r_i \cdot \&(r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n)r', w).$$

We can therefore use the method recursively to check that

$$\mathsf{cert}(r_i \cdot \&(r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n)r', w)$$

certifies that $w \in \|r_i\&(r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n)r'\|$.

3.3.2 Membership is NP-hard

To show that membership is NP-hard, we use a reduction from satisfiability of propositional formulas in conjunctive normal form. Satisfiability of propositional formulas was shown to be NP-hard by Cook [19], though the name "NP" is of newer origin. A polynomial-time translation to conjunctive normal form is given by Tseitin [69]. To make the construction readable, we will use exponents as a shorthand for repeated concatenation. The alphabet consists of the names of the Boolean variables. Given a formula with c clauses and v variables, we construct a regular expression r which is a unordered concatenation of $c + v$ expressions. The first c expressions in the unordered concatenation each represent a clause. In these *clause-expressions*, disjunction is represented by choice ($+$), a positive literal is represented by itself, and a negated literal is represented by concatenating the respective letter with itself $c + 1$ times. The last v expressions in the unordered concatenation, one for each variable x, are of the following form $((x + \epsilon)^c x^{c^2}) + (x^{c+1} + \epsilon)^c$. The word w that we will check for membership, is ${x_1}^{c^2+c} \cdots {x_v}^{c^2+c}$, assuming the variables are $x_1, \ldots, x_v$.

Example 3.3.1. *Let the formula be*

$$(x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3 \vee \neg x_6)$$

Then $v = 6$, $c = 3$ *and* $\Sigma = \{x_1, x_2, x_3, x_4, x_5, x_6\}$. *The regular expression becomes* $\&((x_1 + x_2^4 + x_3^4 + x_4), (x_3 + x_5^4 + x_6), (x_3 + x_6^4), r_1, r_2, r_3, r_4, r_5, r_6)$, *where each* r_i *is* $((x_i + \epsilon)^3 x_i^9) + (x_i^4 + \epsilon)^3$. *The word to check membership in the language of this regular expression becomes* $x_1^{12} \cdots {x_6}^{12}$.

Note first that the formula is satisfiable, e.g. let x_1 *and* x_3 *be true and the others false. We should also get that the word is a member of the constructed expression. This is done as follows: In the first three parts of the unordered concatenation we choose* x_1, x_3 *and* x_3. *In* r_1 *we match* x_1^{11}, *in* r_3 *we match* x_3^{10} *and the other* r_i*'s match* x_i^{12}.

We must show that the problem instance of the membership problem is only polynomially larger than the propositional formula. Let the length of the formula be n, and the number of clauses be c and variables v as above. The length of the word we must check membership for, is $v(c^2 + c)$. The length of the first part of the regular expression representing the clauses, is not more than $(c + 1)n$, and the last part, representing the choices of variables, is $v(2c^2 + 9c)$.

Finally, we must show that the word is in the language of the regular expression if and only if the propositional formula is satisfiable. The main idea is that in the choices in the last v parts of the regular expressions, the left choice can be used if the corresponding variable can be true, and the right choice if it can be false, and that in the subexpressions representing the clauses, the chosen subexpression must be true in the formula.

We first show that if $w \in \|r\|$, then the formula is satisfiable. A satisfiable assignment can be extracted from the matching process in the following manner: For each variable, define its truth value according to which of the choices were used for the matching of the parts of the unordered concatenation corresponding to the variables; If the left hand choice was used, the variable is true, otherwise false. We will show that this assignment satisfies the formula by showing that for each clause-expression, the subexpression used in the matching corresponds to a true literal. More specifically, we show that no subexpression corresponding to a false literal under the assignment can have been used in the matching. For a variable x and the subexpression of r representing it, if the left-hand choice is used when matching w (so the assignment of the variable is to true), then this covers between c^2 and $c^2 + c$ of the x in the word w. This means that in none of the subexpressions representing the clauses, can the subexpression x^{c+1}, representing a negated literal, be chosen, as that would give more x than there are in the word w. On the other hand, if the right hand choice in the sub-expression representing the variable x is chosen, that is, $(x^{c+1} + \epsilon)^c$, then there can be no single x chosen in any of the clause-subexpressions, as we would then get $(x^{c+1})^j x^k$ for some $k < c + 1$ and $j \geq 0$, and this cannot match the $(x^{c+1})^c$ in the word w.

We now show the other direction, that is, that if the formula is satisfiable, then the word w above is in the language of the regular expression r described above. Since the formula is satisfiable, there is at least one satisfying assignment of the variables to truth values. Take such an assignment. Now, for the first c parts of the unordered concatenation, we can use the choice that corresponds to a true literal under this assignment. For the last v parts we use the left-hand choice if the corresponding variable is true in the assignment, and the right-hand choice otherwise. Thus, for variables that are assigned true, the parts of the unordered concatenation corresponding to the clauses match between 0 and c occurrences of the corresponding letter, while the part of the unordered concatenation corresponding to the variable can be used to match between c^2 and $c^2 + c$ of the corresponding letter, and thus the $c^2 + c$ repeated letters in the word w can be matched. For the variables that are assigned false, the clause parts will match $(c + 1)^i$ of the corresponding letters for some i, $0 \leq i \leq c$. But the variable-part of the unordered concatenation can be used to match the remaining $(c + 1)^{(c+1-i)}$ occurrences of the letter in the word.

Some remarks on the above proof are in order. Note that it is enough for NP-hardness with one single top-level unordered concatenation. It is also interesting to note that the proof can easily be adapted to show that the membership problem for regular expressions extended with shuffling is NP-hard.

3.4 Finite Automata with Counters

In this section we describe the finite automata with counters (FAC). The earliest reference having resemblances to FACs are the *multicounter* automata found in Greibach [31]. Sperberg-McQueen [64], Gelade et al. [23], Gelade et al. [24, 25], and Hovland [37] describe the use of FACs to decide the membership problem for regular expressions with numerical constraints. For subexpressions with numerical constraints we use the counters to keep track of the number of times the subexpression has been matched, and use this to control that the numerical constraints are not violated. For regular expressions with unordered concatenation we will use the counters to keep track of which parts of a unordered concatenation have been matched. We keep a counter for every argument in every unordered concatenation. All counters are initially 0. A part of an unordered concatenation can only be used for matching if the corresponding counter is 0, the counter will then be increased to 1. The matching process is only allowed to leave the unordered concatenation when all parts not in $\mathfrak{N}_\Sigma$ have been matched. The counters are then reset to 0.

The FACs described here are a variant of those described by the author in [37], modified to fit unordered concatenation. In general, the counters map to integers not smaller than 0. In the case of unordered concatenation, only the values 0 and 1 are used.

3.4.1 Counter States and Update Instructions

We define *counter states*, which will be used both to keep track of which sub-expressions of an unordered concatenation have been matched, and also to keep track of how many times subexpressions with numerical constraints have been matched. Let $\mathcal{C}$ be the set of positions of subexpressions we need to keep track of. We model counter states as mappings $\gamma : \mathcal{C} \to \mathbb{N}_0$. Let γ_0 be the counter state in which all counters are 0. We define an *update instruction* ψ as a partial mapping from $\mathcal{C}$ to $\{\mathsf{inc}, \mathsf{res}, \mathsf{one}\}$ (inc for *increment*, res for *reset*, one for setting to 1). Update instructions ψ define mappings f_ψ between counter states in the following way: If $\psi(p) = \mathsf{inc}$, then $f_\psi(\gamma)(p) = \gamma(p) + 1$, if $\psi(p) = \mathsf{res}$ then $f_\psi(\gamma)(p) = 0$, if $\psi(p) = \mathsf{one}$ then $f_\psi(\gamma)(p) = 1$, and otherwise $f_\psi(\gamma)(p) = \gamma(p)$.

Definition 3.4.1 (Satisfaction of update instructions). *We define a satisfaction relation between update instructions and counter states. Given* $\gamma : \mathcal{C} \to \mathbb{N}_0$, $\psi : \mathcal{C} \to \{\mathsf{inc}, \mathsf{res}, \mathsf{one}\}$, $\mathsf{min} : \mathcal{C} \to \mathbb{N}_0$, *and* $\mathsf{max} : \mathcal{C} \to \mathbb{N}_1$, $\gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi$ *is defined by the following inductive rules*

$$\begin{array}{lcl}
\gamma \models^{\mathsf{max}}_{\mathsf{min}} \varnothing & & \\
\gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi_1 \cup \{p \mapsto \mathsf{inc}\} & \Leftrightarrow & \gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi_1 \wedge \gamma(p) < \mathsf{max}(p) \\
\gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi_1 \cup \{p \mapsto \mathsf{res}\} & \Leftrightarrow & \gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi_1 \wedge \gamma(p) \geq \mathsf{min}(p) \\
\gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi_1 \cup \{p \mapsto \mathsf{one}\} & \Leftrightarrow & \gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi_1 \wedge \gamma(p) \geq \mathsf{min}(p)
\end{array}$$

The intuition of Definition 3.4.1 is that a counter can only be increased if the value is smaller than the maximum, while a value can only be reset if it's value is at least as large as the minimum.

Example 3.4.2. *Let* $C = \{p_1, p_2\}$, $\mathsf{min}(p_1) = \mathsf{max}(p_1) = 2$, $\mathsf{min}(p_2) = 1$, $\mathsf{max}(p_2) = \infty$ *and* $\gamma = \{p_1 \mapsto 2, p_2 \mapsto 1\}$, *and let* $\psi_1 = \{p_1 \mapsto \mathsf{inc}\}, \psi_2 = \{p_1 \mapsto \mathsf{res}, p_2 \mapsto \mathsf{inc}\}$ *and* $\psi_3 = \{p_1 \mapsto \mathsf{one}, p_2 \mapsto \mathsf{res}\}$. *Then* $f_{\psi_1}(\gamma) = \{p_1 \mapsto 3, p_2 \mapsto 1\}$, $f_{\psi_2}(\gamma) = \{p_1 \mapsto 0, p_2 \mapsto 2\}$ *and* $f_{\psi_3}(\gamma) = \{p_1 \mapsto 1, p_2 \mapsto 0\}$. *Furthermore,* $\gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi_2$ *and* $\gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi_3$ *hold, while it does not hold that* $\gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi_1$.

3.4.2 Overlapping Update Instructions

Given mappings max and min, two update instructions are called *overlapping*, if there is a counter state that satisfies both of the update instructions.

Definition 3.4.3 (Overlapping Update Instructions). *Given mappings* max *and* min, *update instructions* ψ_1 *and* ψ_2 *are overlapping, if and only if there is a counter state* γ, *such that both* $\gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi_1$ *and* $\gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi_2$ *hold.*

Whether two update instructions are overlapping can be decided in linear time, relative to the size of $\mathcal{C}$, by the algorithm presented in the following lemma.

Lemma 3.4.4. *Given mappings* max *and* min*, two update instructions are overlapping if and only if: for every* p *such that it is mapped to* inc *by one of the update instructions, and it is mapped to either* res *or* one *by the other update instruction, it must hold that* $\mathsf{min}(p) < \mathsf{max}(p)$.

Proof. "If-part": Assume that for every p which is mapped to inc by one update instruction, and to res or one by the other, $\mathsf{min}(p) < \mathsf{max}(p)$. We must show that the update instructions are overlapping. A counter state γ satisfying both update instructions can be constructed as follows. Let $\gamma(p) = \mathsf{min}(p)$ if p is mapped to res or one by at least one of the update instructions, otherwise let $\gamma(p) = 0$.

"Only if-part": Assume the update instructions are overlapping. Thus there is at least one counter state γ which satisfies both update instructions ψ_1 and ψ_2. Now, for every p such that $\psi_i(p) = \mathsf{inc}$ and $\psi_{3-i}(p) \in \{\mathsf{res}, \mathsf{one}\}$ for $i \in \{1, 2\}$, we get that $\mathsf{min}(p) \leq \gamma(p) < \mathsf{max}(p)$ from Definition 3.4.1, such that $\mathsf{min}(p) < \mathsf{max}(p)$. □

3.4.3 Finite Automata with Counters

Definition 3.4.5 (Finite Automata with Counters)**.** *A finite automaton with counters (FAC) is a tuple* $(\Sigma, Q, \mathcal{C}, \mathcal{A}, \Phi, \mathsf{min}, \mathsf{max}, q^I, \mathcal{F})$*. The members of the tuple are described below:*

- Σ *is a finite, non-empty set (the alphabet).*
- Q *and* $\mathcal{C}$ *are finite sets of* states *and* counters*, respectively.*
- $q^I \in Q$ *is the* initial state*.*
- $\mathcal{A} : Q - \{q^I\} \to \Sigma$ *maps each non-initial state to the letter which is matched when entering the state.*
- Φ *maps each state to a set of pairs of a state and an update instruction.*

$$\Phi : Q \to \wp(Q \times (\mathcal{C} \to \{\mathsf{inc}, \mathsf{res}, \mathsf{one}\})) .$$

- $\mathsf{min} : \mathcal{C} \to \mathbb{N}_0$ *and* $\mathsf{max} : \mathcal{C} \to \mathbb{N}_1$ *are the counter-conditions.*

- $\mathcal{F} \subset Q \times (\mathcal{C} \to \{\mathsf{res}\})$ *describes the* final configurations *(See Definition 3.4.6).*

Running or executing an FAC is defined in terms of *transitions* between *configurations*. The configurations of an FAC are pairs, where the first element is a member of Q, and the second element is a counter state.

Definition 3.4.6 (Configuration of an FAC)**.** *A* configuration *of an FAC is a pair (q, γ), where $q \in Q$ is the current state and $\gamma : \mathcal{C} \to \mathbb{N}_0$ is the counter state. A configuration (q, γ) is final, if there is $(q, \psi) \in \mathcal{F}$ such that $\gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi$.*

Intuitively, the first member of each of the pairs mapped to by Φ, is the state that can be entered, and the second member describes the changes to the current configuration of the automaton in this step. Thus, Φ and $\mathcal{A}$ together describe the possible transitions of the automaton. This is formalized as the transition function δ.

Definition 3.4.7 (Transition function of an FAC)**.** *For an FAC $(\Sigma, Q, \mathcal{C}, \mathcal{A}, \Phi, q^I, \mathcal{F})$, the transition function δ is defined for any configuration (q, γ) and letter l by*

$$\delta((q, \gamma), l) = \{(p, f_\psi(\gamma)) \mid \mathcal{A}(p) = l, (p, \psi) \in \Phi(q), \gamma \models^{\mathsf{max}}_{\mathsf{min}} \psi\}.$$

Definition 3.4.8 (Deterministic FAC)**.** *An FAC $(\Sigma, Q, \mathcal{C}, \mathcal{A}, \Phi, q^I, \mathcal{F})$ is deterministic if and only if $|\delta((q, \gamma), l)| \leq 1$ for all $q \in Q, l \in \Sigma$ and $\gamma : \mathcal{C} \to \mathbb{N}_0$.*

Deciding whether an FAC is deterministic can be done in polynomial time as follows: For each state p, for each two different (p_1, ψ_1), (p_2, ψ_2) both in $\Phi(p)$, verify that either $\mathcal{A}(p_1) \neq \mathcal{A}(p_2)$ or that ψ_1 and ψ_2 are not overlapping. That this test is sound and complete follows using the definition of δ and the properties of overlapping update instructions.

3.4.4 Word recognition.

An FAC either *accepts* or *rejects* a given input. A deterministic FAC recognizes a word by treating letters in the word one by one. It starts in the *initial configuration* (q^I, γ_0). An FAC in configuration (q, γ), with letter $l \in \Sigma$ next in the word, will reject the word if $\delta((q,\gamma),l)$ is empty. Otherwise it enters the unique configuration $(q',\gamma') \in \delta((q,\gamma),l)$. If the whole word has been read, a deterministic FAC accepts the word if and only if it is in a final configuration. The subset of Σ^* consisting of words being accepted by an FAC A is denoted $\|A\|$. A deterministic FAC accepts or rejects a word in time linear in the length of the word.

Example 3.4.9. *Let*

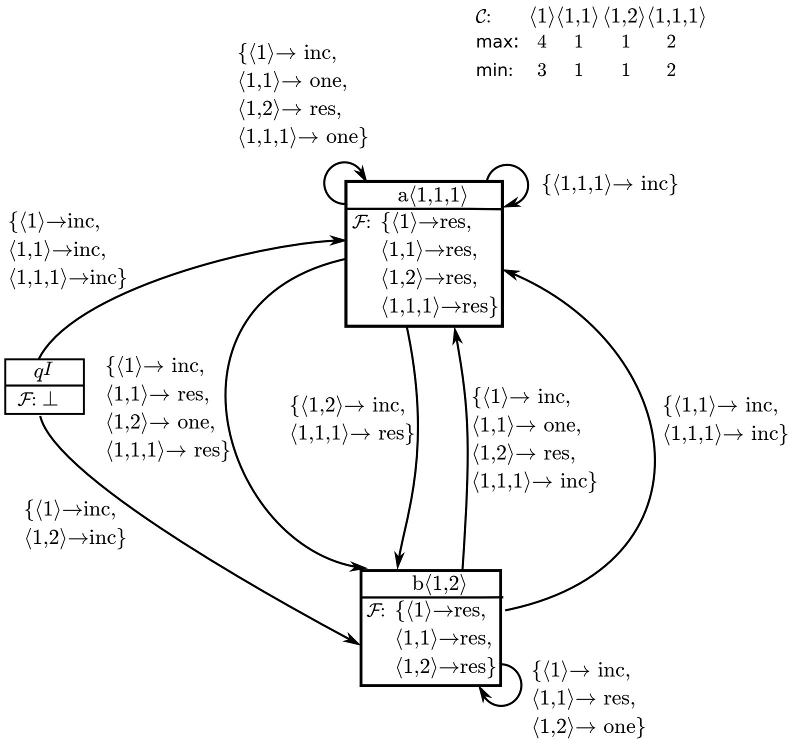
$$\begin{aligned}\Sigma &= \{a,b\}\\ Q &= \{q^I, a\langle 1,1,1\rangle, b\langle 1,2\rangle\}\\ \mathcal{C} &= \{\langle 1\rangle, \langle 1,1\rangle, \langle 1,1,1\rangle, \langle 1,2\rangle\}\end{aligned}$$

Figure 3.4 illustrates a deterministic FAC $(\Sigma, Q, \mathcal{C}, \mathcal{A}, \Phi, \mathsf{min}, \mathsf{max}, q^I, \mathcal{F})$ which recognizes $\|(\&(a^2,b))^{3..4}\|$. Note that the names of the non-initial states are decorated with the values of $\mathcal{A}$. Every state is depicted as a rectangle with the name of the state, and with $\mathcal{F}$ described by the reset instructions. Every member of Φ is shown as an arrow, annotated with the corresponding update instruction. $\mathcal{C}$, min, *and* max *are shown in the top of the figure. The sequence of configurations of this FAC while recognizing aabbaabaa is :*

$$\begin{array}{ll}
(q^I, & \gamma_0)\\
(a\langle 1,1,1\rangle, & \{\langle 1\rangle \mapsto 1, \langle 1,1\rangle \mapsto 1, \langle 1,1,1\rangle \mapsto 1, \langle 1,2\rangle \mapsto 0\})\\
(a\langle 1,1,1\rangle, & \{\langle 1\rangle \mapsto 1, \langle 1,1\rangle \mapsto 1, \langle 1,1,1\rangle \mapsto 2, \langle 1,2\rangle \mapsto 0\})\\
(b\langle 1,2\rangle, & \{\langle 1\rangle \mapsto 1, \langle 1,1\rangle \mapsto 1, \langle 1,1,1\rangle \mapsto 0, \langle 1,2\rangle \mapsto 1\})\\
(b\langle 1,2\rangle, & \{\langle 1\rangle \mapsto 2, \langle 1,1\rangle \mapsto 0, \langle 1,1,1\rangle \mapsto 0, \langle 1,2\rangle \mapsto 1\})\\
(a\langle 1,1,1\rangle, & \{\langle 1\rangle \mapsto 2, \langle 1,1\rangle \mapsto 1, \langle 1,1,1\rangle \mapsto 1, \langle 1,2\rangle \mapsto 1\})\\
(a\langle 1,1,1\rangle, & \{\langle 1\rangle \mapsto 2, \langle 1,1\rangle \mapsto 1, \langle 1,1,1\rangle \mapsto 2, \langle 1,2\rangle \mapsto 1\})\\
(b\langle 1,2\rangle, & \{\langle 1\rangle \mapsto 3, \langle 1,1\rangle \mapsto 0, \langle 1,1,1\rangle \mapsto 0, \langle 1,2\rangle \mapsto 1\})\\
(a\langle 1,1,1\rangle, & \{\langle 1\rangle \mapsto 3, \langle 1,1\rangle \mapsto 1, \langle 1,1,1\rangle \mapsto 1, \langle 1,2\rangle \mapsto 1\})\\
(a\langle 1,1,1\rangle, & \{\langle 1\rangle \mapsto 3, \langle 1,1\rangle \mapsto 1, \langle 1,1,1\rangle \mapsto 2, \langle 1,2\rangle \mapsto 1\})
\end{array}$$

The last configuration is final, since $\mathsf{min}(\langle 1\rangle) \leq 3$, $\mathsf{min}(\langle 1,1\rangle) \leq 1$, *and* $\mathsf{min}(\langle 1,1,1\rangle) \leq 2$.

Lemma 3.4.10 (Linear-time recognition)**.** *For any deterministic FAC $A = (\Sigma, Q, \mathcal{C}, \mathcal{A}, \Phi,$ min, max, $q^I, \mathcal{F})$, if $\sigma(A)$ is the size of the textual representation of A, then for any word $w \in \Sigma^*$, the FAC A accepts or rejects w in time $O(|w|\sigma(A)^2)$.*



$\mathcal{C}$: ⟨1⟩⟨1,1⟩⟨1,2⟩⟨1,1,1⟩
max: 4 1 1 2
min: 3 1 1 2
{⟨1⟩→ inc, ⟨1,1⟩→ one, ⟨1,2⟩→ res, ⟨1,1,1⟩→ one}
{⟨1,1,1⟩→ inc}
a⟨1,1,1⟩
$\mathcal{F}$: {⟨1⟩→res, ⟨1,1⟩→res, ⟨1,2⟩→res, ⟨1,1,1⟩→res}
{⟨1⟩→inc, ⟨1,1⟩→inc, ⟨1,1,1⟩→inc}
qI
$\mathcal{F}$: ⊥
{⟨1⟩→ inc, ⟨1,1⟩→ res, ⟨1,2⟩→ one, ⟨1,1,1⟩→ res}
{⟨1,2⟩→ inc, ⟨1,1,1⟩→ res}
{⟨1⟩→ inc, ⟨1,1⟩→ one, ⟨1,2⟩→ res, ⟨1,1,1⟩→ inc}
{⟨1,1⟩→ inc, ⟨1,1,1⟩→ inc}
{⟨1⟩→inc, ⟨1,2⟩→inc}
b⟨1,2⟩
$\mathcal{F}$: {⟨1⟩→res, ⟨1,1⟩→res, ⟨1,2⟩→res}
{⟨1⟩→ inc, ⟨1,1⟩→ res, ⟨1,2⟩→ one}

Figure 3.4: Illustration of FAC recognizing $\|(\&(a^2, b))^{3..4}\|$.

Proof. The FAC makes at most $|w|$ steps in the recognition, and at each step, there can be at most $\max\{|\Phi(q)| \mid q \in Q\}$ outgoing edges, and for each of these we might have to check the counter state γ against at most $|\mathcal{C}|$ constraints. Testing whether the last configuration is accepting, takes time $O(|\mathcal{C}| \cdot \max\{|\mathcal{F}(q)| \mid q \in Q\})$. Thus we get the result, as $|\mathcal{C}|$, $\max\{|\mathcal{F}(q)| \mid q \in Q\}$ and $\max\{|\Phi(q)| \mid q \in Q\}$ are all $O(\sigma(A))$. □

3.4.5 Searching with FACs

We formalize the problems called matching and searching as the binary predicates $\mathsf{m}, \mathsf{s} \subseteq R_\Sigma \times \Sigma^*$, defined as follows: $\mathsf{m}(r, w) \Leftrightarrow w \in \|r\|$

and $\mathsf{s}(r,w) \Leftrightarrow \exists u,v,v' : (w = u \cdot v \cdot v' \wedge v \in \|r\|)$. A deterministic FAC recognizing $\|r\|$ can decide $\mathsf{m}(r,w)$ in time linear in $|w|$. If the alphabet (Σ) is fixed, an approach for solving $\mathsf{s}(r,w)$ is to solve $\mathsf{m}(\Sigma^* \cdot r \cdot \Sigma^*)$, where Σ here denotes the disjunction of all the letters. For our purposes, this approach is not usable, as we cannot in general construct a deterministic FAC recognizing $\Sigma^* \cdot r \cdot \Sigma^*$. Also, in practical cases, the size of Σ can be prohibitively large. Another option is therefore to decide $\mathsf{s}(r,w)$ by executing $\mathsf{m}(r,w')$ for every subword w' of w. This leads to $O(|w|^2)$ executions of the algorithm for m. However, a deterministic FAC can also decide in linear time the *prefix problem*. The latter is also formalized as a binary predicate, namely $\mathsf{p} \subseteq R_\Sigma \times \Sigma^*$, where $\mathsf{p}(r,w) \Leftrightarrow \exists u,v : (w = u \cdot v \wedge u \in \|r\|)$. $O(|w|)$ executions of an algorithm for p is sufficient to decide s. Thus, deterministic FACs can be used to search in time quadratic in the length of the text.

3.5 Subscripting and Unambiguity

In this section we will introduce notation and lemmas that are necessary for the construction of FACs from regular expressions.

Definition 3.5.1 (Concatenating positions). *For $p \in \mathbb{N}^*$ and $S, S' \subseteq \mathbb{N}^*$*

- *Put $p \odot S = \{p \odot q \mid q \in S\}$*
- *Put $S \odot S' = \{p \odot q \mid p \in S, q \in S'\}$*
- *For $\psi : (\mathbb{N}^* \to \{\mathsf{inc}, \mathsf{res}, \mathsf{one}\})$, put $p \odot \psi = \{(p \odot q) \mapsto \psi(q) \mid q \in \mathsf{dom}(\psi)\}$.*
- *For $S \subseteq \mathbb{N}^* \times (\mathbb{N}^* \to \{\mathsf{inc}, \mathsf{res}, \mathsf{one}\})$ put $p \odot S = \{(p \odot q, p \odot \psi) \mid (q, \psi) \in S\}$.*

Definition 3.5.2 (Subposition). *We use the notation $p \leq q$ for p a* prefix *or* subposition *of q, that is, $p \leq q \Leftrightarrow \exists p_1 : q = p \odot p_1$. We write $p < q$ iff $p \leq q$ and $p \neq q$.*

Definition 3.5.3 (Subexpression). *Assume a regular expression r.*

- *For any position $p \in \mathsf{pos}(r)$ we will denote the subexpression rooted at this position by $r[p]$.*

• *For any string of positions $p_1 \cdots p_n$, where $p_1, \ldots, p_n \in \mathsf{pos}(r)$, put $r[p_1 \cdots p_n] = r[p_1] \cdots r[p_n]$.*

• *For a set S of strings of positions in r, that is, $S \subseteq (\mathsf{pos}(r))^*$, put $r[S] = \{r[w] \mid w \in S\}$.*

Note that $r[\langle\rangle] = r$. Note that for any regular expression r, and any position p, $\|p \odot \mu(r)\| = p \odot \|\mu(r)\|$. The following lemma, corresponding to Lemma 2.2.10, will often be used tacitly.

Lemma 3.5.4. *For any regular expression r,*

1. $\|r\| = r[\|\mu(r)\|]$
2. *For any $p \in \mathsf{sym}(\mu(r))$, $\mu(r)[p] = p$*
3. *For any $p \in \mathsf{pos}(r)$, $r[p] \in \Sigma$ iff $p \in \mathsf{sym}(\mu(r))$.*

Proof.

1. By induction on r, similar to the proof of the corresponding part of Lemma 2.2.10. We treat only the induction case where $r = \&(r_1, \ldots, r_n)$ in more detail. By Definition 3.2.2 and Definition 3.2.4,

$$\|\mu(r)\| = \bigcup_{\sigma \in \mathsf{Perm}(\{1,\ldots,n\})} (\|\sigma_1 \mu(r_{\sigma_1})\| \cdots \|\sigma_n \mu(r_{\sigma_n})\|) \tag{3.1}$$

For each i, $\|\sigma_i \mu(r_{\sigma_i})\| = \sigma_i \|\mu(r_{\sigma_i})\|$. We now apply distributivity of $r[]$ over union and concatenation to (3.1) and get

$$r[\|\mu(r)\|] = \bigcup_{\sigma \in \mathsf{Perm}(\{1,\ldots,n\})} (r[\|\sigma_1 \mu(r_{\sigma_1})\|] \cdots r[\|\sigma_n \mu(r_{\sigma_n})\|]) \tag{3.2}$$

By the properties of concatenating positions, and Definition 3.5.3

$$\forall i \in \{1, \ldots, n\} : r[\|\sigma_i \mu(r_{\sigma_i})\|] = r[\langle\sigma_i\rangle][\|\mu(r_{\sigma_i})\|] = r_{\sigma_i}[\|\mu(r_{\sigma_i})\|] \tag{3.3}$$

We apply (3.3) and the induction hypothesis to (3.2) and get

$$r[\|\mu(r)\|] = \bigcup_{\sigma \in \mathsf{Perm}(\{1,\ldots,n\})} (\|r_{\sigma_1}\| \cdots \|r_{\sigma_n}\|) \tag{3.4}$$

Hence, by Definition 3.2.4 $r[\|\mu(\&(r_1, \ldots, r_n))\|] = \|\&(r_1, \ldots, r_n)\|$.

2. By induction on r, similar to the proof of the corresponding part of Lemma 2.2.10. We treat only the induction case where $r = \&(r_1, \ldots, r_n)$ in more detail. $p \in \mathsf{sym}(\mu(r))$ implies there is $i \in \{1, \ldots, n\}$ and $p' \in \mathsf{sym}(\mu(r_i))$ such that $p = ip'$. By the induction hypothesis for r_i, $\mu(r_i)[p'] = p'$, hence $i(\mu(r_i)[p']) = p$. By Definitions 3.2.2 and 3.5.3, $r[ip'] = (i\mu(r_i))[p'] = i(\mu(r_i)[p'])$. Thus $r[p] = p$.

3. By induction on r, similar to the proof of the corresponding part of Lemma 2.2.10. We treat only the induction case where $r = \&(r_1, \ldots, r_n)$ and $p \neq \langle\rangle$. There is $i \in \{1, \ldots, n\}$ and $p' \in \mathsf{pos}(r_i)$ such that $p = ip'$. We have $r[p] = r_i[p']$ and that $p \in \mathsf{sym}(\mu(r))$ iff $p' \in \mathsf{sym}(\mu(r_i))$. Hence, the lemma holds by applying the induction hypothesis for r_i.

□

Definition 3.5.5. *Let $r \in R_\Sigma$ and $p \in \mathsf{pos}(r)$.*

1. *Put $\langle\sharp\rangle(r) \subseteq \mathsf{pos}(r)$ to be the positions of all subexpressions r_1 occurring in a subexpression $r_1^{l..u}$ of r. Expressed formally, $\langle\sharp\rangle(r) =$*

$$\{q \odot \langle 1\rangle \in \mathsf{pos}(r) \mid \exists n \in \mathbb{N}, m \in \mathbb{N}_1, r_1 \in R_\Sigma : r[q] = r_1^{n..m}\}.$$

2. *Put $\langle\&\rangle(r) \subseteq \mathsf{pos}(r)$ to be the positions of all arguments of all unordered subexpressions in r. Expressed formally, $\langle\&\rangle(r) =$*

$$\{q \odot \langle i\rangle \in \mathsf{pos}(r) \mid \exists r_1, \ldots, r_n \in R_\Sigma : r[q] = \&(r_1, \ldots, r_n))\}.$$

3. *Put $\langle\&\rangle(r,p) \subseteq \mathsf{pos}(r)$ to be all positions in r which correspond to the argument of any unordered subexpressions above position p in r. Expressed formally, $\langle\&\rangle(r,p) =$*

$$\{q \odot \langle i\rangle \in \mathsf{pos}(r) \mid q \leq p \ \wedge \ \exists r_1, \ldots, r_n \in R_\Sigma : r[q] = \&(r_1, \ldots, r_n))\}.$$

4. *Put $\langle\sharp\rangle(r,p) \subseteq \langle\sharp\rangle(r)$ to be the left children of all positions of all numerical constraints in r above p. Expressed formally,*

$$\langle\sharp\rangle(r,p) = \{q \in \langle\sharp\rangle(r) \mid q \leq p\}.$$

5. *Put $\langle\&\sharp\rangle(r) = \langle\&\rangle(r) \cup \langle\sharp\rangle(r)$ and $\langle\&\sharp\rangle(r,p) = \langle\&\rangle(r,p) \cup \langle\sharp\rangle(r,p)$.*

In the sequel we need to express the set of regular expressions whose language contains the empty word. The set of *nullable expressions*, $\mathfrak{N}_\Sigma = \{r \in R_\Sigma \mid \epsilon \in \|r\|\}$, has an easy inductive definition extending Definition 2.2.3.

Definition 3.5.6 (Nullable Expressions)**.** *Given an alphabet Σ, the set of nullable expressions, $\mathfrak{N}_\Sigma$, is defined in the following inductive manner*

$$\mathfrak{N}_\Sigma ::= \mathfrak{N}_\Sigma \cdot \mathfrak{N}_\Sigma \mid \mathfrak{N}_\Sigma + R_\Sigma \mid R_\Sigma + \mathfrak{N}_\Sigma \mid \mathfrak{N}_\Sigma^{\mathbb{N}..\mathbb{N}_1} \mid \&(\mathfrak{N}_\Sigma, \ldots, \mathfrak{N}_\Sigma) \mid \epsilon$$

3.5.1 Constraint Normal Form

We will below define the right unambiguity we need for constructing deterministic automata. However, the construction of FACs can be applied to any regular expression, but it must first be put in *constraint normal form*. The construction of an FAC from an expression in this class can also be done in polynomial time, but the FAC might not be deterministic. Furthermore, we can in polynomial time put an expression in constraint normal form. An expression is in constraint normal form if, for every subexpression of the form $r^{n..m}$, r is not nullable.

Definition 3.5.7. *A regular expression r is in constraint normal form if it has no subexpression of the form $r_1^{n..m}$ with $r_1 \in \mathfrak{N}_\Sigma$.*

For example, $(a^*a)^{2..3}$ is in constraint normal form, while $(a^*)^{2..3}$ is not.

There is a mapping from arbitrary regular expressions to regular expressions in constraint normal form, called cnf, which preserves the language of the regular expression.

Let $|r| \in \mathbb{N}$ be the size of the regular expression r, that is, the sum of the number of letter-, ϵ-, and operator- occurrences in r.

Definition 3.5.8. *We define simultaneously the mappings $\mathsf{cnf} : R_\Sigma \to R_\Sigma$ and $\mathsf{cnf}^> : \mathfrak{N}_\Sigma - \{\epsilon\} \to R_\Sigma$ with the following inductive rules:*

- $\mathsf{cnf}(\epsilon) = \epsilon$ *(and $\mathsf{cnf}^>(\epsilon)$ undefined).*

- $\mathsf{cnf}(r) = r$ *(and $\mathsf{cnf}^>(r)$ undefined) if $r \in \Sigma$.*

- $\mathsf{cnf}(r) = \mathsf{cnf}(r_1) + \epsilon$ *and* $\mathsf{cnf}^>(r) = \mathsf{cnf}(r_1)$ *when* $r = r_1 + \epsilon$ *or* $r = \epsilon + r_1$*, where* $r_1 \notin \mathfrak{N}_\Sigma$.

- $\mathsf{cnf}(r) = \mathsf{cnf}(r_1) + \epsilon$ *and* $\mathsf{cnf}^>(r) = \mathsf{cnf}^>(r_1)$*, when* $r = r_1 + \epsilon$ *or* $r = \epsilon + r_1$*, where* $r_1 \in \mathfrak{N}_\Sigma$*.*

- $\mathsf{cnf}(r_1 + r_2) = \mathsf{cnf}(r_1) + \mathsf{cnf}(r_2)$ *and* $\mathsf{cnf}^>(r_1 + r_2) = r_1' + r_2'$*, when* $r_1, r_2 \neq \epsilon$ *and for* $i \in \{1, 2\}$*: if* $r_i \in \mathfrak{N}_\Sigma$*,* $r_i' = \mathsf{cnf}^>(r_i)$*, else* $r_i' = \mathsf{cnf}(r_i)$*.*

- $\mathsf{cnf}(r_1 \cdot r_2) = \mathsf{cnf}(r_1) \cdot \mathsf{cnf}(r_2)$ *and* $\mathsf{cnf}^>(r_1 \cdot r_2) = \mathsf{cnf}^>(r_1) \cdot \mathsf{cnf}(r_2) + \mathsf{cnf}^>(r_2)$*, when* $|r_1| > |r_2|$*.*

- $\mathsf{cnf}(r_1 \cdot r_2) = \mathsf{cnf}(r_1) \cdot \mathsf{cnf}(r_2)$ *and* $\mathsf{cnf}^>(r_1 \cdot r_2) = \mathsf{cnf}(r_1) \cdot \mathsf{cnf}^>(r_2) + \mathsf{cnf}^>(r_1)$*, when* $|r_1| \leq |r_2|$*.*

- $\mathsf{cnf}(r_1^{n..m}) = \mathsf{cnf}(r_1)^{n..m}$ *(and* $\mathsf{cnf}^>(r)$ *undefined), when* $r_1 \notin \mathfrak{N}_\Sigma$*.*

- $\mathsf{cnf}(r_1^{n..m}) = \mathsf{cnf}^>(r_1)^{1..m} + \epsilon$ *and* $\mathsf{cnf}^>(r_1^{n..m}) = \mathsf{cnf}^>(r_1)^{1..m}$*, when* $r_1 \in \mathfrak{N}_\Sigma$*.*

- $\mathsf{cnf}(\&(r_1)) = \mathsf{cnf}(r_1)$ *and* $\mathsf{cnf}^>(\&(r_1)) = \mathsf{cnf}^>(r_1)$*.*

- $\mathsf{cnf}(\&(r_1, \ldots, r_n)) = \&(\mathsf{cnf}(r_1), \ldots, \mathsf{cnf}(r_n))$ *and* $\mathsf{cnf}^>(\&(r_1, \ldots, r_n)) =$

$$\begin{aligned}\&(\mathsf{cnf}(r_1), \ldots, \mathsf{cnf}(r_{i-1}), \mathsf{cnf}^>(r_i), \mathsf{cnf}(r_{i+1}), \ldots, \mathsf{cnf}(r_n))\\ +\mathsf{cnf}^>(\&(r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n))\end{aligned}$$

when $n \geq 2$ *and* r_i *is the largest of the expressions* $r_1, \ldots, r_n$*.*

Before we can show the correctness of the definition of cnf, we need some new notation. For a set $L \subseteq \Sigma^*$ put $L^> = L - \{\epsilon\}$ and $\epsilon^L = L \cap \{\epsilon\}$.

Lemma 3.5.9. *For any regular expression r:*

1. $\mathsf{cnf}(r)$ *is defined, and if* $r \in \mathfrak{N}_\Sigma - \{\epsilon\}$*,* $\mathsf{cnf}^>(r)$ *is also defined.*

2. $\|r\| = \|\mathsf{cnf}(r)\|$*, and if* $r \in \mathfrak{N}_\Sigma - \{\epsilon\}$*,* $\|\mathsf{cnf}^>(r)\| = \|r\|^>$*.*

3. $\mathsf{cnf}(r)$ *is in constraint normal form.*

4. $|\mathsf{cnf}(r)| \leq |r|^2$*, and if* $r \in \mathfrak{N}_\Sigma - \{\epsilon\}$*,* $|\mathsf{cnf}^>(r)| \leq |r|^2$*.*

Proof. All parts are proved in a simultaneous induction on the size of r. The base cases $r = \epsilon$ and $r \in \Sigma$ hold easily, as $\mathsf{cnf}(r) = r$ is in constraint normal form, and r is not in the domain of $\mathsf{cnf}^>$.

For the induction cases where $r = r_1 + \epsilon$ or $r = \epsilon + r_1$ note that $r_1 \neq \epsilon$, as subexpressions of the form $\epsilon + \epsilon$ are simplified. All parts then hold by using the induction hypothesis for r_1.

The induction case where $r = r_1 + r_2$ and $r_1, r_2 \neq \epsilon$ holds by applying the induction hypothesis for r_1 and r_2.

For the induction case where $r = r_1 \cdot r_2$ and $|r_1| > |r_2|$, all parts hold for cnf directly from the induction hypothesis. For $\mathsf{cnf}^>$ we need some more details. For part 1 note that since $r_1 \cdot r_2 \in \mathfrak{N}_\Sigma$ then also $r_1, r_2 \in \mathfrak{N}_\Sigma$. Furthermore, $r_1, r_2 \neq \epsilon$ since subexpressions of the forms $r\epsilon$ and ϵr are simplified. For part 2, note that since $r_1, r_2 \in \mathfrak{N}_\Sigma$, $\|r_1 \cdot r_2\|^> = (\|r_1\|^> \cdot \|r_2\|^>) \cup \|r_1\|^> \cup \|r_2\|^>$. Furthermore, since $\|r_1\|^> \cdot \|r_2\|^> \cup \|r_1\|^> = \|r_1\|^> \cdot \|r_2\|$, we get that $(\|r_1\|^> \cdot \|r_2\|^>) \cup \|r_1\|^> \cup \|r_2\|^> = (\|r_1\|^> \cdot \|r_2\|) \cup \|r_2\|^>$. By applying the induction hypothesis and Definition 3.2.4 we get $(\|r_1\|^> \cdot \|r_2\|) \cup \|r_2\|^> = \|\mathsf{cnf}^>(r_1) \cdot r_2 + \mathsf{cnf}^>(r_2)\|$, hence $\|r_1 \cdot r_2\|^> = \|\mathsf{cnf}^>(r_1 \cdot r_2)\|$. Part 3 holds directly from the induction hypothesis. For part 4, by the definition, $|\mathsf{cnf}^>(r)| = |r_1|^2 + 2|r_2|^2$, which since $|r_2| < |r_1|$ is smaller than $(|r_1| + |r_2|)^2 = |r_1|^2 + 2|r_1||r_2| + |r_2|^2 > |r_1|^2 + 3|r_2|^2$.

The induction case where $r = r_1 \cdot r_2$ and $|r_1| \leq |r_2|$ is similar to the previous case.

For the induction case where $r = r_1^{l..u}$, where $r_1 \notin \mathfrak{N}_\Sigma$, all parts hold for cnf directly from the induction hypothesis. Since $r_1 \notin \mathfrak{N}_\Sigma$, $r \notin \mathfrak{N}_\Sigma$, so r is not in the domain of $\mathsf{cnf}^>$.

We next treat the induction case where $r = r_1^{l..u}$ for $r_1 \in \mathfrak{N}_\Sigma$. Part 1 holds from the induction hypothesis, the assumption $r_1 \in \mathfrak{N}_\Sigma$, and the fact that subexpressions of the form $\epsilon^{l..u}$ are simplified. For part 2 it is sufficient to show that $\|(\mathsf{cnf}^>(r_1))^{1..u}\| = \|r_1^{l..u}\|^>$. By Definition 3.2.4 and the induction hypothesis for r_1 we get

$$\|(\mathsf{cnf}^>(r_1))^{1..u}\| = \bigcup_{i=1}^{u} (\|r_1\|^>)^i \tag{3.5}$$

By Definition 3.2.4 $\|r_1^{l..u}\| = \bigcup_{i=l}^{u} \|r_1\|^i$, hence it only remains to show that

$$\bigcup_{i=1}^{u} (\|r_1\|^>)^i = (\bigcup_{i=l}^{u} \|r_1\|^i)^> \tag{3.6}$$

We split the equality (3.6) in two inclusions. Firstly, if a word w is in the left-hand set, then w is a concatenation of j words from $\|r_1\|^>$, where $1 \leq j \leq u$. If $j \geq l$, we can choose $i = j$ and the same words as in the left-hand side, hence w is also in the right-hand set. Otherwise, if $j < l$, we let $i = l$, and choose the same words as on the left-hand side for the first j words, and choose ϵ from the last $l - j$ parts of the concatenation. Hence w is also in the right-hand set. Secondly, if a word w is in the right-hand set, it is a concatenation of $l \leq i \leq u$ words from $\|r_1\|$, where at least one word is not ϵ. Put i on the left-hand side union the number of non-epsilon words chosen on the right-hand side. All these words are of course in $\|r_1\|^>$, and they number at most u and at least 1. Hence w is also in the left-hand set. For part 3 we use parts 2 and 3 of the induction hypothesis on r_1. Part 4 holds from the induction hypothesis.

For the induction case where $r = \&(r_1)$ all parts hold directly from the induction hypothesis.

For the induction case where $r = \&(r_1, \ldots, r_n)$, $n \geq 2$, and r_i is the largest of the expressions, all parts hold for cnf directly from the induction hypothesis. For $\mathsf{cnf}^>$ note that we have $r_1, \ldots, r_n \in \mathfrak{N}_\Sigma - \{\epsilon\}$, so part 1 holds by using the induction hypothesis. For part 2 we use Definition 3.2.4, and distributivity of $^>$ over union to get

$$\|\&(r_1, \ldots, r_n)\|^> = \bigcup_{\sigma \in \mathsf{Perm}(\{1,\ldots,n\})} (\|r_{\sigma_1}\| \cdots \|r_{\sigma_n}\|)^> \tag{3.7}$$

For any $\sigma \in \mathsf{Perm}(\{1, \ldots, n\})$, put $\sigma^{-1}(i)$ such that $\sigma_{\sigma^{-1}(i)} = i$. We get that for any $\sigma \in \mathsf{Perm}(\{1, \ldots, n\})$

$$\begin{aligned}
&(\|r_{\sigma_1}\| \cdots \|r_{\sigma_n}\|)^> \\
&= \left(\begin{array}{l} \|r_{\sigma_1}\| \cdots \|r_{\sigma_{\sigma^{-1}(i)-1}}\| \cdot \|r_i\|^> \cdot \|r_{\sigma_{\sigma^{-1}(i)+1}}\| \cdots \|r_{\sigma_n}\| \\ \cup(\|r_{\sigma_1}\| \cdots \|r_{\sigma_{\sigma^{-1}(i)-1}}\| \cdot \|r_{\sigma_{\sigma^{-1}(i)+1}}\| \cdots \|r_{\sigma_n}\|)^> \end{array} \right)
\end{aligned} \tag{3.8}$$

A word is a member of the set on either side of the equality in (3.8) if it is of the form $w_1 \cdots w_n$, where for each w_j, $w_j \in \|r_{\sigma_j}\|$, and there is at least one w_j such that $w_j \neq \epsilon$.

By applying the induction hypothesis and (3.8) to (3.7), we get

$$
\begin{aligned}
&\|\&(r_1,\ldots,r_n)\|^{>} = \\
&\bigcup_{\sigma\in\mathsf{Perm}(\{1,\ldots,n\})}\left(\begin{array}{l}\|\mathsf{cnf}(r_{\sigma_1})\|\cdots\|\mathsf{cnf}(r_{\sigma_{\sigma^{-1}(i)-1}})\|\cdot\\ \cdot\|\mathsf{cnf}^{>}(r_i)\|\cdot\|\mathsf{cnf}(r_{\sigma_{\sigma^{-1}(i)+1}})\|\cdots\|\mathsf{cnf}(r_{\sigma_n})\|\\ \cup\|r_{\sigma_1}\|\cdots\|r_{\sigma_{\sigma^{-1}(i)-1}}\|\cdot\|r_{\sigma_{\sigma^{-1}(i)+1}}\|\cdots\|r_{\sigma_n}\|)^{>}\end{array}\right)\\
&=\left(\begin{array}{l}\bigcup_{\sigma\in\mathsf{Perm}(\{1,\ldots,n\})}\left(\begin{array}{l}\|\mathsf{cnf}(r_{\sigma_1})\|\cdots\|\mathsf{cnf}(r_{\sigma_{\sigma^{-1}(i)-1}})\|\cdot\\ \cdot\|\mathsf{cnf}^{>}(r_i)\|\cdot\|\mathsf{cnf}(r_{\sigma_{\sigma^{-1}(i)+1}})\|\cdots\|\mathsf{cnf}(r_{\sigma_n})\|\end{array}\right)\\ \cup(\bigcup_{\sigma\in\mathsf{Perm}(\{1,\ldots,n\})-\{i\}}\|r_{\sigma_1}\|\cdots\|r_{\sigma_n}\|)^{>}\end{array}\right)
\end{aligned}
\tag{3.9}
$$

Applying Definition 3.2.4 and the induction hypothesis once more to (3.9) we get

$$
\begin{aligned}
&\|\&(r_1,\ldots,r_n)\|^{>}\\
&=\|\&(\mathsf{cnf}(r_1),\ldots,\mathsf{cnf}(r_{i-1}),\mathsf{cnf}^{>}(r_i),\mathsf{cnf}(r_{i+1}),\ldots,\mathsf{cnf}(r_n))\\
&\quad+\mathsf{cnf}^{>}(\&(r_1,\ldots,r_{i-1},r_{i+1},\ldots,r_n))\|
\end{aligned}
$$

Hence, $\|\&(r_1,\ldots,r_n)\|^{>}=\|\mathsf{cnf}^{>}(\&(r_1,\ldots,r_n))\|$.

Part 3 holds from the induction hypothesis. For part 4, recall that the size of an unordered concatenation is the sum of the size of the arguments plus one. We therefore get

$$
\begin{aligned}
&|\&(r_1,\ldots,r_n)|^2\\
&=(|r_1|+\cdots+|r_n|+1)^2\\
&=\Sigma_{j=1}^{n}(|r_j|^2+|r_j|+|r_j|\Sigma_{k\in\{1,\ldots,n\}-\{j\}}|r_k|\\
&=\Sigma_{j=1}^{n}|r_j|^2+\Sigma_{j=1}^{n}|r_j|+\Sigma_{j=1}^{n}(|r_j|\Sigma_{k\in\{1,\ldots,n\}-\{j\}}|r_k|)
\end{aligned}
\tag{3.10}
$$

Since r_i is the largest argument to the unordered concatenation,

$$
|r_i|\Sigma_{j\in\{1,\ldots,n\}-\{i\}}|r_j|\geq\Sigma_{j\in\{1,\ldots,n\}-\{i\}}|r_j|^2 \tag{3.11}
$$

By the induction hypothesis, Definition 3.5.8, and (3.11) we get

$$
\begin{aligned}
&|\mathsf{cnf}^{>}(\&(r_1,\ldots,r_n))| \\
&\leq \Sigma_{j=1}^{n}|r_j|^2 + (|r_1| + \cdots + |r_{i-1}| + |r_{i+1}| + \cdots + |r_n| + 1)^2 \\
&= \Sigma_{j=1}^{n}|r_j|^2 + \Sigma_{j\in\{1,\ldots,n\}-\{i\}}(|r_j|^2 + |r_j| + |r_j|(\Sigma_{k\in\{1,\ldots,n\}-\{j,i\}}|r_k|)) \\
&= \left(\begin{array}{l} \Sigma_{j=1}^{n}|r_j|^2 + \Sigma_{j\in\{1,\ldots,n\}-\{i\}}|r_j|^2 + \Sigma_{j\in\{1,\ldots,n\}-\{i\}}|r_j| \\ +\Sigma_{j\in\{1,\ldots,n\}-\{i\}}(|r_j|\Sigma_{k\in\{1,\ldots,n\}-\{j,i\}}|r_k|) \end{array}\right) \\
&\leq \left(\begin{array}{l} \Sigma_{j=1}^{n}|r_j|^2 + |r_i|\Sigma_{j\in\{1,\ldots,n\}-\{i\}}|r_j| + \Sigma_{j=1}^{n}|r_j| \\ +\Sigma_{j\in\{1,\ldots,n\}-\{i\}}(|r_j|\Sigma_{k\in\{1,\ldots,n\}-\{j\}}|r_k|) \end{array}\right) \\
&= \left(\begin{array}{l} \Sigma_{j=1}^{n}|r_j|^2 + \Sigma_{j=1}^{n}|r_j| \\ +|r_i|\Sigma_{j\in\{1,\ldots,n\}-\{i\}}|r_j| + \Sigma_{j\in\{1,\ldots,n\}-\{i\}}(|r_j|\Sigma_{k\in\{1,\ldots,n\}-\{j\}}|r_k|)) \end{array}\right) \\
&= \Sigma_{j=1}^{n}|r_j|^2 + \Sigma_{j=1}^{n}|r_j| + \Sigma_{j=1}^{n}(|r_j|\Sigma_{k\in\{1,\ldots,n\}-\{j\}}|r_k|)
\end{aligned}
\tag{3.12}
$$

Hence, combining (3.10) and (3.12) we get

$$|\mathsf{cnf}^{>}(\&(r_1,\ldots,r_n))| \leq |\&(r_1,\ldots,r_n)|^2$$

□

The following corollary is mainly a summary of Lemma 3.5.9.

Corollary 3.5.10. *For any regular expression r, $\mathsf{cnf}(r)$ can be calculated in polynomial time, is in constraint normal form, and recognizes the same language as r.*

Proof. All properties are immediate from Lemma 3.5.9, except the polynomial runtime. To show that the runtime is polynomial, we will require some pre-processing before applying the rules of Definition 3.5.8. For each subexpression of r, we calculate the size, and whether it is nullable. This can be done in time polynomial in r in a bottom-up manner. We can then prove by induction on the size of r that calculating $\mathsf{cnf}(r)$ takes time $O(|r|^2)$ and if $r \in \mathfrak{N}_\Sigma - \{\epsilon\}$, calculating $\mathsf{cnf}^{>}(r)$ takes time $O(|r|^2)$. The base cases are immediate. The inductive cases for choice and star hold by applying the induction hypothesis. The cases for concatenation and unordered concatenation are similar to part 4 of the corresponding cases in the proof of Lemma 3.5.9. □

The mappings cnf and $\mathsf{cnf}^{>}$ are inspired by the mappings $^{\circ}$ and $^{\bullet}$ defined in [10] and $^{-}$ defined in [12]

3.5.2 Subscripted Expressions

Subscripting is inspired by the *bracketing* used by Koch & Scherzinger [47] and Gelade et al. [23]. The intuition is that for a regular expression r, the subscripted regular expression $\mathsf{ss}(r)$, is such that all subexpressions of the form $r_1^{l..u}$ or $\&(r_1, \dots, r_n)$ are subscripted with their position in the term tree.

Definition 3.5.11. *For any regular expression r, the subscripted regular expression $\mathsf{ss}(r)$ is defined by the following inductive rules:*

- $\mathsf{ss}(\epsilon) = \epsilon$.
- *If $l \in \Sigma$, then* $\mathsf{ss}(l) = l$.
- $\mathsf{ss}(r_1 + r_2) = \odot_1 \mathsf{ss}(r_1) + \odot_2 \mathsf{ss}(r_2)$
- $\mathsf{ss}(r_1 \cdot r_2) = \odot_1 \mathsf{ss}(r_1) \cdot \odot_2 \mathsf{ss}(r_2)$
- $\mathsf{ss}(r_1^{l..u}) = (\odot_1 \mathsf{ss}(r_1))_{\langle\rangle}^{l..u}$
- $\mathsf{ss}(\&(r_1, \dots, r_n)) = \&(\odot_1 \mathsf{ss}(r_1), \dots, \odot_n \mathsf{ss}(r_n))_{\langle\rangle}$

where for $i \in \mathbb{N}$ and r a subscripted regular expression, $\odot_i r$ denotes the expression where all positions in subscripts in r are prefixed with i.

For example, $\mathsf{ss}((\&(a^2, b))^{3..4}) = (\&(a^2_{\langle 1,1\rangle}, b)_{\langle 1\rangle})^{3..4}_{\langle\rangle}$.

Lemma 3.5.12. *For any regular expression r, any $p \in \mathsf{pos}(r)$, and any subscripted regular expression r'_q, if $\mathsf{ss}(r)[p] = r'_q$, then $p = q$.*

Proof. By induction on r. The base cases hold vacuously. For the inductive cases where $r = r_1 \cdot r_2$ or $r = r_1 + r_2$ we must have $p \neq \langle\rangle$. Hence, there are $i \in \{1,2\}$, $p_i \in \mathsf{pos}(r_i)$, and r''_{q_i} such that $p = ip_i$, $q = iq_i$ and $\mathsf{ss}(r_i)[p_i] = r''_{q_i}$. By the induction hypothesis on r_i, $p_i = q_i$, hence $p = q$. For the inductive cases where $r = r_1^{l..u}$ and $r = \&(r_1, \dots, r_n)$, if $p = \langle\rangle$, the lemma holds immediately, as $r'_q = \mathsf{ss}(r)$. The case where $p \neq \langle\rangle$ is similar to the cases for concatenation and choice. $\square$

For a regular expression r, let $\Gamma_r = \bigcup_{p \in \langle \& \sharp \rangle (r)} \{\uparrow_p, \downarrow_p\}$. The language of a subscripted expression r is a set of strings over $\mathsf{sym}(r) \cup \Gamma_r$. For the not subscripted parts we use the same rules as in Definition 3.2.4, while

$$\|r_p^{l..u}\| = (\bigcup_{i=l}^{u} (\{\uparrow_{p1}\} \cdot \|r\|)^i) \cdot \{\downarrow_{p1}\},$$

and $\|\&(r_1, \ldots, r_n)_p\| =$

$$\epsilon^{\|\&(r_1,\ldots,r_n)\|} \cup \left(\bigcup_{\sigma \in \mathsf{Perm}(\{1,\ldots,n\})} \left(\begin{array}{l} \epsilon^{\|r_{\sigma_1}\|} \cup \{\uparrow_{p\sigma_1}\} \cdot \|r_{\sigma_1}\|^{>} \\ \cdots \\ \epsilon^{\|r_{\sigma_n}\|} \cup \{\uparrow_{p\sigma_n}\} \cdot \|r_{\sigma_n}\|^{>} \end{array} \right)^{>} \right) \cdot \{\downarrow_{p1} \cdots \downarrow_{pn}\}$$

Example 3.5.13. *The word*

$$\begin{array}{l} \uparrow_{\langle 1\rangle}\uparrow_{\langle 1,1\rangle}\uparrow_{\langle 1,1,1\rangle}\langle 1,1,1\rangle\uparrow_{\langle 1,1,1\rangle}\langle 1,1,1\rangle\downarrow_{\langle 1,1,1\rangle}\uparrow_{\langle 1,2\rangle}\langle 1,2\rangle\downarrow_{\langle 1,1\rangle}\downarrow_{\langle 1,2\rangle} \\ \uparrow_{\langle 1\rangle}\uparrow_{\langle 1,2\rangle}\langle 1,2\rangle\uparrow_{\langle 1,1\rangle}\uparrow_{\langle 1,1,1\rangle}\langle 1,1,1\rangle\uparrow_{\langle 1,1,1\rangle}\langle 1,1,1\rangle\downarrow_{\langle 1,1,1\rangle}\downarrow_{\langle 1,1\rangle}\downarrow_{\langle 1,2\rangle} \\ \uparrow_{\langle 1\rangle}\uparrow_{\langle 1,1\rangle}\uparrow_{\langle 1,1,1\rangle}\langle 1,1,1\rangle\uparrow_{\langle 1,1,1\rangle}\langle 1,1,1\rangle\downarrow_{\langle 1,1,1\rangle}\uparrow_{\langle 1,2\rangle}\langle 1,2\rangle\downarrow_{\langle 1,1\rangle}\downarrow_{\langle 1,2\rangle} \\ \downarrow_{\langle 1\rangle} \end{array}$$

is in $\|\mathsf{ss}(\mu((\&(a^2, b))^{3..4}))\|$.

For a word $w \in (\Sigma \cup \Gamma_r)^*$, put $\flat(w) \in \Sigma^*$ the same as w except that all symbols from Γ_r are removed. Furthermore, for $S \subseteq (\Sigma \cup \Gamma_r)^*$, put $\flat(S) = \{\flat(w) \mid w \in S\}$.

Lemma 3.5.14. *For any regular expression r in constraint normal form*

1. $\epsilon^{\|\mathsf{ss}(r)\|} = \epsilon^{\|r\|}$.

2. $\flat(\|\mathsf{ss}(r)\|) = \|r\|$.

Proof. We will use repeatedly that for $L_1, L_2 \subseteq (\Sigma \cup \Gamma_r)^*$, $\epsilon^{L_1} \cup \epsilon^{L_2} = (L_1 \cap \{\epsilon\}) \cup (L_2 \cap \{\epsilon\}) = (L_1 \cup L_2) \cap \{\epsilon\} = \epsilon^{L_1 \cup L_2}$, and $\epsilon^{L_1 \cdot L_2} = (L_1 \cap \{\epsilon\}) \cap (L_2 \cap \{\epsilon\}) = (L_1 \cap L_2) \cap \{\epsilon\} = \epsilon^{L_1 \cap L_2}$. We will also often use tacitly that $\flat()$ is neutral for the empty set and ϵ, and distributes over union, intersection, and concatenation.

1. By induction on r. The base cases are easy, since by Definition 3.5.11 $\mathsf{ss}(r) = r$.

For the induction case where $r = r_1 + r_2$, by Definition 3.5.11 and the definition of the language of a subscripted expression, $\epsilon^{\|\mathsf{ss}(r_1+r_2)\|} = \epsilon^{\|\odot_1\mathsf{ss}(r_1)\|\cup\|\odot_2\mathsf{ss}(r_2)\|} = \epsilon^{\|\odot_1\mathsf{ss}(r_1)\|} \cup \epsilon^{\|\odot_2\mathsf{ss}(r_2)\|}$. Since $\epsilon^{\|\odot_i\mathsf{ss}(r_i)\|} = \epsilon^{\|\mathsf{ss}(r_i)\|}$ for $i \in \{1,2\}$, we can apply the induction hypothesis for r_1 and r_2 to get $\epsilon^{\|\mathsf{ss}(r_1+r_2)\|} = \epsilon^{\|r_1\|} \cup \epsilon^{\|r_2\|} = \epsilon^{\|r_1\|\cup\|r_2\|}$. By applying Definition 3.2.4, we get $\epsilon^{\|\mathsf{ss}(r_1+r_2)\|} = \epsilon^{\|r_1+r_2\|}$.

For the induction case where $r = r_1 \cdot r_2$, by definition $\epsilon^{\|\mathsf{ss}(r_1\cdot r_2)\|} = \epsilon^{\|\odot_1\mathsf{ss}(r_1)\|\cdot\|\odot_2\mathsf{ss}(r_2)\|} = \epsilon^{\|\mathsf{ss}(r_1)\|} \cap \epsilon^{\|\mathsf{ss}(r_2)\|}$. We apply the induction hypothesis for r_1 and r_2 to get $\epsilon^{\|\mathsf{ss}(r_1\cdot r_2)\|} = \epsilon^{\|r_1\|} \cap \epsilon^{\|r_2\|} = \epsilon^{\|r_1\|\cap\|r_2\|}$. By applying Definition 3.2.4, we get $\epsilon^{\|\mathsf{ss}(r_1\cdot r_2)\|} = \epsilon^{\|r_1\cdot r_2\|}$.

For the induction case where $r = r_1^{l..u}$, by Definition 3.5.11 and the definition of the language of a subscripted expression $\epsilon^{\|\mathsf{ss}(r_1^{l..u})\|} = \epsilon^{(\bigcup_{i=l}^{u}(\{\uparrow_{\langle 1\rangle}\}\cdot\|\odot_1\mathsf{ss}(r_1)\|)^i)\cdot\{\downarrow_{\langle 1\rangle}\}} = \epsilon^{\bigcup_{i=l}^{u}\|\mathsf{ss}(r_1)\|^i}$. By Definition 3.2.1, $l \geq 1$, hence, $\epsilon^{\bigcup_{i=l}^{u}\|\mathsf{ss}(r_1)\|^i} = \epsilon^{\|\mathsf{ss}(r_1)\|}$. Since r is in constraint normal form, $r_1 \notin \mathfrak{N}_\Sigma$, thus $\epsilon^{\|r_1\|} = \varnothing$. By applying the induction hypothesis for r_1 we therefore get $\epsilon^{\|\mathsf{ss}(r_1^{l..u})\|} = \varnothing$. Furthermore, since $l \geq 1$ and $\epsilon^{\|r_1\|} = \varnothing$, we get by Definition 3.2.4 that $\epsilon^{\|r_1^{l..u}\|} = \varnothing = \epsilon^{\|\mathsf{ss}(r_1^{l..u})\|}$.

For the induction case where $r = \&(r_1, \ldots, r_n)$, by Definition 3.5.11 and the definition of the language of a subscripted expression

$$\epsilon^{\|\mathsf{ss}(\&(r_1,\ldots,r_n))\|} = \epsilon^{\|\&(\odot_1\mathsf{ss}(r_1),\ldots,\odot_n\mathsf{ss}(n)r_n)\|} = \bigcap_{i=1}^{n} \epsilon^{\|\mathsf{ss}(r_i)\|}$$

By applying the induction hypothesis and Definition 3.2.4, we therefore get

$$\epsilon^{\|\mathsf{ss}(\&(r_1,\ldots,r_n))\|} = \bigcap_{i=1}^{n} \epsilon^{\|r_i\|} = \epsilon^{\|\&(r_1,\ldots,r_n)\|}$$

2. By induction on r. For each case we use the corresponding part of Definition 3.5.11. The base cases are easy, since $\mathsf{ss}(r) = r$.

For the induction case where $r = r_1 + r_2$, by Definition 3.5.11 and the definition of the language of a subscripted expression $\|\mathsf{ss}(r_1 + r_2)\| = \|\odot_1\mathsf{ss}(r_1)\| \cup \|\odot_2\mathsf{ss}(r_2)\|$. Furthermore, $\flat(\|\odot_1\mathsf{ss}(r_1)\| \cup \|\odot_2\mathsf{ss}(r_2)\|) = \flat(\|\mathsf{ss}(r_1)\|) \cup \flat(\|\mathsf{ss}(r_2)\|)$, and by the induction hypothesis $\flat(\|\mathsf{ss}(r_1)\|) \cup$

$\flat(\|\mathsf{ss}(r_2)\|) = \|r_1\| \cup \|r_2\|$. By Definition 3.2.4, $\|r_1\| \cup \|r_2\| = \|r_1 + r_2\|$, hence, $\flat(\|\mathsf{ss}(r_1 + r_2)\|) = \|r_1 + r_2\|$.

The induction case for $r = r_1 \cdot r_2$ can be shown by replacing $+$ and $\cup$ with $\cdot$ in the preceding paragraph.

For the induction case where $r = r_1^{l..u}$, by Definition 3.5.11 and the definition of the language of a subscripted expression, we get $\|\mathsf{ss}(r)\| = (\bigcup_{i=l}^{u}(\{\uparrow_{\langle 1\rangle}\} \cdot \|\odot_1\mathsf{ss}(r_1)\|)^i) \cdot \{\downarrow_{\langle 1\rangle}\}$. Hence, $\flat(\|\mathsf{ss}(r)\|) = \bigcup_{i=l}^{u}(\flat(\|\mathsf{ss}(r_1)\|))^i$. By applying the induction hypothesis for r_1 and Definition 3.2.4 we therefore get $\flat(\|\mathsf{ss}(r_1^{l..u})\|) = \bigcup_{i=l}^{u}\|r_1\|^i = \|r_1^{l..u}\|$

For the induction case where $r = \&(r_1, \ldots, r_n)$, by Definition 3.5.11 and the definition of the language of a subscripted expression, we get $\|\mathsf{ss}(r)\| =$

$$\begin{array}{l} \epsilon^{\|\&(\odot_1\mathsf{ss}(r_1),\ldots,\odot_n\mathsf{ss}(r_n))\|} \cup \\ \left(\bigcup_{\sigma\in\mathsf{Perm}(\{1,\ldots,n\})} \left(\begin{array}{l} \epsilon^{\|\odot_{\sigma_1}\mathsf{ss}(r_{\sigma_1})\|} \cup \{\uparrow_{p\sigma_1}\} \cdot \|\odot_{\sigma_1}\mathsf{ss}(r_{\sigma_1})\|^{>} \\ \cdots \\ \epsilon^{\|\odot_{\sigma_n}\mathsf{ss}(r_{\sigma_n})\|} \cup \{\uparrow_{p\sigma_n}\} \cdot \|\odot_{\sigma_n}\mathsf{ss}(r_{\sigma_n})\|^{>} \end{array} \right)^{>} \right. \\ \left. \cdot\{\downarrow_{p1} \cdots \downarrow_{pn}\} \right. \end{array} \tag{3.13}$$

We apply that $\flat()$ is neutral for the empty set and ϵ, and distributes over union, intersection, and concatenation to (3.13), and get $\flat(\|\mathsf{ss}(r)\|) =$

$$\begin{array}{l} \epsilon^{\|\&(\odot_1\mathsf{ss}(r_1),\ldots,\odot_n\mathsf{ss}(r_n))\|} \cup \\ \left(\bigcup_{\sigma\in\mathsf{Perm}(\{1,\ldots,n\})} \left(\begin{array}{l} \flat(\epsilon^{\|\odot_{\sigma_1}\mathsf{ss}(r_{\sigma_1})\|} \cup \|\odot_{\sigma_1}\mathsf{ss}(r_{\sigma_1})\|^{>}) \\ \cdots \\ \flat(\epsilon^{\|\odot_{\sigma_n}\mathsf{ss}(r_{\sigma_n})\|} \cup \|\odot_{\sigma_n}\mathsf{ss}(r_{\sigma_n})\|^{>}) \end{array} \right)^{>} \right) \end{array} \tag{3.14}$$

By Definition 3.2.4, the definition of the language of a subscripted expression, and part 1 of the present lemma, we get

$$\begin{aligned} &\epsilon^{\|\&(\odot_1\mathsf{ss}(r_1),\ldots,\odot_n\mathsf{ss}(r_n))\|} \\ &= \epsilon^{\|\mathsf{ss}(r_1)\|} \cap \cdots \cap \epsilon^{\|\mathsf{ss}(r_n)\|} \\ &= \epsilon^{\|r_1\|} \cap \cdots \cap \epsilon^{\|r_n\|} \\ &= \epsilon^{\|\&(r_1,\ldots,r_n)\|} \end{aligned} \tag{3.15}$$

By applying (3.15) and that that for any set S, $S^{>} \cup \epsilon^{S} = S$ to (3.14) we get $\flat(\|\mathsf{ss}(r)\|) =$

$$\epsilon^{\|\&(r_1,\ldots,r_n)\|} \cup \left(\bigcup_{\sigma \in \mathsf{Perm}(\{1,\ldots,n\})} \begin{pmatrix} \flat(\|\odot_{\sigma_1}\mathsf{ss}(r_{\sigma_1})\|) \\ \cdots \\ \flat(\|\odot_{\sigma_n}\mathsf{ss}(r_{\sigma_n})\|) \end{pmatrix}^{>} \right) \tag{3.16}$$

By applying the induction hypothesis for $r_1, \ldots, r_n$ to (3.16) we get $\flat(\|\mathsf{ss}(r)\|) =$

$$\epsilon^{\|\&(r_1,\ldots,r_n)\|} \cup \left(\bigcup_{\sigma \in \mathsf{Perm}(\{1,\ldots,n\})} (\|r_{\sigma_1}\| \cdots \|r_{\sigma_n}\|) \right)^{>} \tag{3.17}$$

By applying Definition 3.2.4 and (again) that for any set S, $S^{>} \cup \epsilon^{S} = S$ to (3.17), we get $\flat(\|\mathsf{ss}(r)\|) = \bigcup_{\sigma \in \mathsf{Perm}(\{1,\ldots,n\})}(\|r_{\sigma_1}\| \cdots \|r_{\sigma_n}\|)$. Hence, by Definition 3.2.4 $\flat(\|\mathsf{ss}(r)\|) = \|r\|$.

□

For positions p, q, put $p \odot \uparrow_q = \uparrow_{p\odot q}$ and $p \odot \downarrow_q = \downarrow_{p\odot q}$. For positions p, q, and a subscripted regular expression r put $p \odot r_q = (p \odot r)_{p\odot q}$. We extend the rules in Definitions 3.5.1 and 3.5.3 to include subscripts and arrows.

Lemma 3.5.15. *For any regular expression r in constraint normal form $r[\flat(\|\mathsf{ss}(\mu(r))\|)] = \|r\|$.*

Proof. By Lemma 3.5.14 $r[\flat(\|\mathsf{ss}(\mu(r))\|)] = r[\|\mu(r)\|]$. By Lemma 3.5.4 $r[\|\mu(r)\|] = \|r\|$. Hence, $r[\flat(\|\mathsf{ss}(\mu(r))\|)] = \|r\|$. □

We summarize some properties of the languages of subscripted expressions in the following lemma.

Lemma 3.5.16. *For any regular expression r in constraint normal form, any $u, v, w \in (\Sigma \cup \mathsf{pos}(r) \cup \Gamma_r)^*$, any $p, q \in \mathsf{pos}(r)$, any $l, m \in \Sigma$, and any $\alpha, \beta \in \Gamma_r^*$:*

1. $\|\mathsf{ss}(r)\| \cap (\Gamma_r^*)^{>} = \varnothing$

2. *If $\alpha l u \in \|\mathsf{ss}(r)\|$, then there are $p_1, \ldots, p_i \in \mathsf{pos}(r)$ such that $\alpha = \uparrow_{p_1} \cdots \uparrow_{p_i}$.*

3. *If $ul\alpha \in \|\mathsf{ss}(r)\|$, then there are $p_1, \ldots, p_i \in \mathsf{pos}(r)$ such that $\alpha = \downarrow_{p_1} \cdots \downarrow_{p_i}$.*

4. *If $\alpha pu, \beta pv \in \|\mathsf{ss}(\mu(r))\|$, then $\alpha = \beta$.*

5. *If $up\alpha, vp\beta \in \|\mathsf{ss}(\mu(r))\|$, then $\alpha = \beta$.*

6. *If $ul\alpha mv \in \|\mathsf{ss}(r)\|$, then there are $p_1, \ldots, p_i, \ldots, p_{i+j}$ such that $\alpha = \downarrow_{p_1} \cdots \downarrow_{p_i} \cdot \uparrow_{p_{i+1}} \cdots \uparrow_{p_{i+j}}$.*

Proof.

1. By induction on r. The base cases are immediate. The induction cases where $r = r_1 \cdot r_2$ or $r = r_1 + r_2$ follow from the induction hypothesis.

For the induction case where $r = r_1^{l..u}$ we must use that r is in constraint normal form, thus $r_1 \notin \mathfrak{N}_\Sigma$. Therefore any word in $\|\mathsf{ss}(r)\|$ must contain letters from r_1.

For the induction case where $r = \&(r_1, \ldots, r_n)$, note that from the definition, for any word $w \in \|\mathsf{ss}(r)\|^{>}$ there is a i, $1 \leq i \leq n$, a $u \in \|\mathsf{ss}(r_i)\|^{>}$ and a v such that $w = \uparrow_{\langle i \rangle} \cdot iu \cdot v$. By the induction hypothesis $u \notin \Gamma_{r_i}^*$, so the lemma holds.

2. By induction on r. The base case $r = \epsilon$ holds vacuously. The base case where $r \in \Sigma$ hold since $\alpha = \epsilon$. The induction case for choice hold directly from the induction hypothesis. The induction case for concatenation hold by applying part 1 and the induction hypothesis.

For the induction case where $r = r_1^{l..u}$, we get from part 1 that there are α_1, u_1, and u_2 such that $\alpha \cdot l \cdot u = \uparrow_{\langle 1 \rangle} \cdot 1\alpha_1 \cdot l \cdot 1u_1 \cdot u_2$ and $\alpha_1 \cdot l \cdot u_1 \in \|\mathsf{ss}(r_1)\|$. We can apply the induction hypothesis for r_1 to get the result.

For the induction case where $r = \&(r_1, \ldots, r_n)$, we can again use part 1 to get that there is a j, $1 \leq j \leq n$, and there are α_1, u_1, and u_2 such that $\alpha \cdot l \cdot u = \uparrow_{\langle j \rangle} \cdot j\alpha_1 \cdot l \cdot ju_1 \cdot u_2$ and $\alpha_1 \cdot l \cdot u_1 \in \|\mathsf{ss}(r_j)\|$. We can now apply the induction hypothesis for r_j to get the result.

3. Analogous to the previous part.

4. By induction on r. The base cases hold immediately. The induction case for $r = r_1 + r_2$ holds by applying the induction hypothesis for r_1 and r_2. The induction case for $r = r_1 \cdot r_2$ holds by applying part 1 together with the induction hypothesis for r_1 and r_2.

For the induction case where $r = r_1^{l..n}$ note that by part 1 and by definition there must be $\alpha_1, \beta_1 \in \Gamma_{r_1}^*$, $p_1 \in \mathsf{pos}(r_1)$, $u_1, v_1 \in (\mathsf{pos}(r_1) \cup \Gamma_{r_1})^*$, and $u_2, v_2 \in (\mathsf{pos}(r) \cup \Gamma_r)^*$ such that $\alpha p u = \uparrow_{\langle 1 \rangle} \cdot 1(\alpha_1 \cdot p_1 \cdot u_1) \cdot u_2$, $\beta p v = \uparrow_{\langle 1 \rangle} \cdot 1(\beta_1 \cdot p_1 \cdot v_1) \cdot v_2$, and $\alpha_1 p_1 u_1, \beta_1 p_1 v_1 \in \|\mathsf{ss}(\mu(r_1))\|$. By applying the induction hypothesis for r_1 we get $\alpha_1 = \beta_1$, hence $\alpha = \beta$.

For the induction case where $r = \&(r_1, \ldots, r_n)$ note that by part 1 and by definition there must be $i \in \{1, \ldots, n\}, \alpha_1, \beta_1 \in \Gamma_{r_i}^*$, $u_1, v_1 \in (\mathsf{pos}(r_i) \cup \Gamma_{r_i})^*$, $u_2, v_2 \in (\mathsf{pos}(r) \cup \Gamma_r)^*$, and $p_1 \in \mathsf{sym}(\mu(r_i))$ such that $p = i p_1$, $\alpha p u = \uparrow_{\langle i \rangle} \cdot i(\alpha_1 p_1 u_1) \cdot u_2$, $\beta p u = \uparrow_{\langle i \rangle} \cdot i(\beta_1 p_1 v_1) \cdot v_2$, and $\alpha_1 p_1 u_1, \beta_1 p_1 v_1 \in \|\mathsf{ss}(\mu(r_i))\|$. By applying the induction hypothesis for r_i we get $\alpha_1 = \beta_1$, hence $\alpha = \beta$.

5. By induction on r. The base cases hold immediately. The induction case for $r = r_1 + r_2$ holds by applying the induction hypothesis for r_1 and r_2. The induction case for $r = r_1 \cdot r_2$ holds by applying 1 together with the induction hypothesis for r_1 and r_2.

For the induction case where $r = r_1^{l..u}$ note that by 1 and by definition there must be $\alpha_1, \beta_1 \in \Gamma_{r_1}^*$ $p_1 \in \mathsf{sym}(\mu(r_1))$, $u_1, v_1 \in (\mathsf{pos}(r) \cup \Gamma_r)^*$, and $u_2, v_2 \in (\mathsf{pos}(r_1) \cup \Gamma_{r_1})^*$ such that $u p \alpha = u_1 \cdot \uparrow_{\langle 1 \rangle} \cdot 1(u_2 p_1 \alpha_1) \cdot \downarrow_{\langle 1 \rangle}$ and $v p \beta = v_1 \cdot \uparrow_{\langle 1 \rangle} \cdot 1(v_2 p_1 \beta_1) \cdot \downarrow_{\langle 1 \rangle}$. By applying the induction hypothesis for r_1 we get $\alpha_1 = \beta_1$, hence $\alpha = \beta$.

For the induction case where $r = \&(r_1, \ldots, r_n)$ note that by 1 and by definition there must be $i \in \{1, \ldots, n\}, \alpha_1, \beta_1 \in \Gamma_{r_i}^*$, $p_1 \in \mathsf{sym}(\mu(r_i))$, $u_1, v_1 \in (\mathsf{pos}(r) \cup \Gamma_r)^*$, and $u_2, v_2 \in (\mathsf{pos}(r_i) \cup \Gamma_{r_i})^*$ such that $u p \alpha = u_1 \cdot i(u_2 p_1 \alpha_1) \cdot \downarrow_{\langle 1 \rangle} \cdots \downarrow_{\langle n \rangle}$, $v p \beta = v_1 \cdot \uparrow_{\langle i \rangle} \cdot i(v_2 p_1 \beta_1) \cdot \downarrow_{\langle 1 \rangle} \cdots \downarrow_{\langle n \rangle}$, and $u_2 p_1 \alpha_1, v_2 p_1 \beta_1 \in \|\mathsf{ss}(\mu(r_i))\|$. By applying the induction hypothesis for r_i we get $\alpha_1 = \beta_1$, hence $\alpha = \beta$.

6. By induction on r. The base cases hold vacuously.

For the induction case where $r = r_1 \cdot r_2$, note that $\|\mathsf{ss}(r_1 \cdot r_2)\| = \odot_1 \|\mathsf{ss}(r_1)\| \cdot \odot_2 \|\mathsf{ss}(r_2)\|$. The cases where $u \cdot l \cdot \alpha \cdot m \cdot v = u \cdot l \cdot \alpha \cdot m \cdot v_1 \cdot v_2$ and $u \cdot l \cdot \alpha \cdot m \cdot v_1 \in \odot_1 \|\mathsf{ss}(r_1)\|$, and the cases where $u \cdot l \cdot \alpha \cdot m \cdot v = u_1 \cdot u_2 \cdot l \cdot \alpha \cdot m \cdot v_1 \cdot v_2$ and $u_2 \cdot l \cdot \alpha \cdot m \cdot v \in \odot_1 \|\mathsf{ss}(r_1)\|$, hold by applying the induction hypothesis for r_1 and r_2, respectively. The remaining cases hold by applying parts 2 and 3.

The induction case where $r = r_1 + r_2$ holds by applying the induction hypothesis for r_1 and r_2.

For the induction case where $r = r_1^{l..u}$, note that from definition

$$\|\mathsf{ss}(r)\| = (\bigcup_{i=l}^{u}(\{\uparrow_{\langle 1\rangle}\} \cdot \|\odot_1\mathsf{ss}(r_1)\|)^i) \cdot \{\downarrow_{\langle 1\rangle}\}$$

From 1 we have $\|\odot_1\mathsf{ss}(r_1)\| \cap ({}^{>}\Gamma_r^*) = \varnothing$. Thus, either $ul\alpha mv = u_1u_2l\alpha mv_1v_2$, where $u_2l\alpha mv_1 \in \|\odot_1\mathsf{ss}(r_1)\|$, or $ul\alpha mv = u_1u_2l\alpha_1\alpha_2m1v_1v_2$, where $u_2l\alpha_1, \alpha_2mv_1 \in \|\odot_1\mathsf{ss}(r_1)\|$. In the former case we can apply the induction hypothesis for r_1 to get the result. In the latter case we apply parts 2 and 3 of this lemma.

The induction case where $r = \&(r_1, \ldots, r_n)$ holds either by applying the induction hypothesis for one of the r_i, or by applying 2 and 3 together with Lemma 3.5.15 and the definition of the languages of subscripted expressions.

□

Subscripted expressions are central in the construction of FACs. The next definition describes how a word in Γ_r^* defines an update instruction.

Definition 3.5.17. *Given a regular expression r, and a word α over Γ_r, we define the update instruction $\mathsf{update}_r(\alpha) : \langle\&\sharp\rangle(r) \to \{\mathsf{inc}, \mathsf{res}, \mathsf{one}\}$ in the following way.*

$$(\mathsf{update}_r(\alpha))(c) = \begin{cases} \mathsf{one} & \textit{if both } \downarrow_c \textit{ and } \uparrow_c \textit{ are in } \alpha \\ \mathsf{res} & \textit{if } \downarrow_c \textit{ but not } \uparrow_c \textit{ is in } \alpha \\ \mathsf{inc} & \textit{if } \uparrow_c \textit{ but not } \downarrow_c \textit{ is in } \alpha \end{cases}$$

If α is empty, $\mathsf{update}_r(\alpha)$ is the empty update instruction.

We will further define a mapping $\mathsf{s}()$ with the purpose of describing the connection between words in the subscripted language and configurations of the FAC during matching of a word.

Definition 3.5.18. *For any regular expression r and any word $w \in (\mathsf{pos}(r) \cup \Gamma_r)^*$, $\mathsf{s}_r(w)$ is an update instruction over $\langle\&\sharp\rangle(r)$, defined in the following inductive manner: $\mathsf{s}_r(\epsilon) = \gamma_0$, $\mathsf{s}_r(\alpha) = \mathsf{update}_r(\alpha)(\gamma_0)$ for $\alpha \in (\Gamma_r^*)^{>}$, and $\mathsf{s}_r(w \cdot p \cdot \alpha) = \mathsf{update}_r(\alpha)(\mathsf{s}_r(w))$ for $\alpha \in \Gamma_r^*$, $p \in \mathsf{pos}(r)$.*

Definition 3.5.19. *For any regular expression r, $\mathsf{min}(r)$ and $\mathsf{max}(r)$ are mappings with domain $\langle \& \sharp \rangle(r)$ and codomains $\mathbb{N}_0$ and $\mathbb{N}_1$, respectively. For any $q1 \in \langle \sharp \rangle(r)$ we have $r[q] = r[q1]^{l..u}$ and put $\mathsf{min}(r)(q1) = l$ and $\mathsf{max}(r)(q1) = u$. For any $q \in \langle \& \rangle(r)$, put $\mathsf{max}(r)(q) = 1$, and if $r[q] \in \mathfrak{N}_\Sigma$, put $\mathsf{min}(r)(q) = 0$, otherwise let $\mathsf{min}(r)(q) = 1$.*

Lemma 3.5.20 formulates some easy properties of $\mathsf{update}()$, $\mathsf{s}()$, and $\models$. The lemma will often be used tacitly.

Lemma 3.5.20. *For any regular expression r, any $q \in \mathsf{pos}(r)$, any $w \in (\mathsf{pos}(r[q]) \cup \Gamma_{r[q]})^*$, and any $\alpha \in \Gamma^*_{r[q]}$, the following holds:*

1. $\mathsf{update}_r(q \odot \alpha) = q \odot \mathsf{update}_{r[q]}(\alpha)$.

2. $q \odot \mathsf{s}_{r[q]}(w) = \mathsf{s}_r(q \odot w)$.

3. $\mathsf{s}_r(q \odot w) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(q \odot \alpha)$ *if and only if* $\mathsf{s}_{r[q]}(w) \models^{\mathsf{max}(r[q])}_{\mathsf{min}(r[q])} \mathsf{update}_{r[q]}(\alpha)$.

Proof.

1. Immediate from Definition 3.5.17.

2. By strong induction on the word w. The cases when $w = \epsilon$ and when $r = \alpha$ hold immediately, and by using part 1, respectively. The induction cases where $w = w'p\alpha$ hold by applying the induction hypothesis for w' and part 1 to α.

3. From parts 1 and 2 and Definitions 3.5.19 and 3.4.1.

□

Lemma 3.5.21. *For any regular expression r in constraint normal form, the following holds:*

1. *For any $w \in \|\mathsf{ss}(r)\|$, $\mathsf{s}_r(w) = \gamma_0$.*

2. *If there are $u, v \in (\mathsf{pos}(r) \cup \Gamma_r)^*$, $\alpha \in \Gamma^*_r$, and $p, q \in \mathsf{pos}(r)$ such that $up\alpha qv \in \|\mathsf{ss}(\mu(r))\|$ or $up\alpha \in \|\mathsf{ss}(\mu(r))\|$, then $\mathsf{s}_r(u) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(\alpha)$.*

Proof.

1. By induction on the regular expression r. The base cases hold immediately. The induction case where $r = r_1 + r_2$ holds by applying the induction hypothesis for r_1 and r_2, and part 2 of Lemma 3.5.20.

For the induction case where $r = r_1 r_2$, there are $w_1 \in \|\mathsf{ss}(r_1)\|$ and $w_2 \in \|\mathsf{ss}(r_2)\|$ such that $w = 1w_1 \cdot 2w_2$. If $w_1 = \epsilon$ or $w_2 = \epsilon$, we get from the induction hypothesis for r_2 or r_1, respectively, combined with part 2 of Lemma 3.5.20 that $\mathsf{s}_r(w) = \gamma_0$. Otherwise, we get from Lemma 3.5.16 that there are $i, j \geq 0$, $k \geq 1$, $p, p_1, \ldots, p_i \in \mathsf{pos}(r_1)$, $q_1, \ldots, q_j, q_{j+1}, \ldots, q_{j+k} \in \mathsf{pos}(r_2)$, $\alpha_1, \ldots, \alpha_k \in \Gamma_{r_2}^*$, and $u, v \in (\mathsf{pos}(r) \cup \Gamma_r)^*$ such that $w_1 = up{\downarrow_{p_1}} \cdots {\downarrow_{p_i}}$ and $w_2 = {\uparrow_{q_1}} \cdots {\uparrow_{q_j}} q_{j+1}\alpha_1 \cdots q_{j+k}\alpha_k$. Note that the positions in $1\mathsf{pos}(r_1)$ and $2\mathsf{pos}(r_2)$ do not overlap. We use the latter fact, together with the induction hypothesis for r_1, Definition 3.5.17, and Definition 3.5.18 to get that

$$\begin{aligned}
&\mathsf{update}_r(1({\downarrow_{p_1}} \cdots {\downarrow_{p_i}}) \cdot 2({\uparrow_{q_1}} \cdots {\uparrow_{q_j}}))(\mathsf{s}_r(1u)) \\
=&\mathsf{update}_r(2({\uparrow_{q_1}} \cdots {\uparrow_{q_j}}))(\mathsf{update}_r(1({\downarrow_{p_1}} \cdots {\downarrow_{p_i}}))(\mathsf{s}_r(1u))) \\
=&\mathsf{update}_r(2({\uparrow_{q_1}} \cdots {\uparrow_{q_j}}))(\mathsf{s}_r(1w_1)) \\
=&\mathsf{update}_r(2({\uparrow_{q_1}} \cdots {\uparrow_{q_j}}))(\gamma_0).
\end{aligned}$$

Using the latter equation, the induction hypothesis for r_2 and Definition 3.5.18 we get

$$\begin{aligned}
&\mathsf{s}_r(w) \\
=\ &\mathsf{update}_r(2\alpha_k)(\cdots(\mathsf{update}_r(2\alpha_1)(\mathsf{update}_r(1({\downarrow_{p_1}} \cdots {\downarrow_{p_i}}) \\
&\quad \cdot 2({\uparrow_{q_1}} \cdots {\uparrow_{q_j}}))(\mathsf{s}_r(1u)))) \cdots) \\
=\ &\mathsf{update}_r(2\alpha_k)(\cdots(\mathsf{update}_r(2\alpha_1)(\mathsf{update}_r(2({\uparrow_{q_1}} \cdots {\uparrow_{q_j}}))(\gamma_0))) \cdots) \\
=\ &\mathsf{update}_r(2w_2) \\
=\ &\gamma_0
\end{aligned}$$

For the induction case when $r = r_1^{l..u}$, by definition all words end in ${\downarrow_{\langle 1 \rangle}}$. Combined with the induction hypothesis and a calculation analogous to the case for concatenation, we get the lemma.

For the induction case where $r = \&(r_1, \ldots, r_n)$, by definition all words end in ${\downarrow_{\langle 1 \rangle}} \cdots {\downarrow_{\langle n \rangle}}$. Combined with the induction hypothesis and a calculation analogous to the case for concatenation, we get the lemma.

2. By induction on the regular expression r. The base cases hold immediately. The induction case where $r = r_1 + r_2$ hold by applying the induction hypothesis for r_1 or for r_2.

For the induction case where $r = r_1 \cdot r_2$ we treat first the case where $up\alpha \in \|\mathsf{ss}(\mu(r_1 \cdot r_2))\|$. If $\langle 1\rangle \leq p$, note that from part 1 of Lemma 3.5.16 we must have $up\alpha \in 1\|\mathsf{ss}(\mu(r_1 \cdot r_2))\|$, and we get from the induction hypothesis for r_1 that $\mathsf{s}_r(u) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(\alpha)$. Otherwise, if $\langle 2\rangle \leq p$, we must have u_1, u_2 such that $u = u_1 \cdot u_2$, $u_1 \in 1\|\mathsf{ss}(\mu(r_1))\|$, and $u_2 p\alpha \in 2\|\mathsf{ss}(\mu(r_2))\|$. We get from the induction hypothesis for r_2 that $\mathsf{s}_r(u_2) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(\alpha)$. Combined with part 1 of the current lemma we get $\mathsf{s}_r(u_1 \cdot u_2) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(\alpha)$.

If $up\alpha qv \in \|\mathsf{ss}(\mu(r_1 \cdot r_2))\|$, we only treat the case where there are $p' \in \mathsf{pos}(r_1)$ and $q' \in \mathsf{pos}(r_2)$ such that $p = 1p'$ and $q = 2q'$. (The other cases follow from using the induction hypothesis, part 1 of the current lemma, and part 1 of Lemma 3.5.16.) Then we also have $\alpha_1 \in {\Gamma_{r_1}}^*$, $\alpha_2 \in {\Gamma_{r_2}}^*$, $u' \in (\mathsf{pos}(r_1) \cup \Gamma_{r_1})^*$, and $v' \in (\mathsf{pos}(r_2) \cup \Gamma_{r_2})^*$ such that $up\alpha qv = 1(u' \cdot p' \cdot \alpha_1) \cdot 2(\alpha_2 \cdot q' \cdot v')$, $u' \cdot p' \cdot \alpha_1 \in \|\mathsf{ss}(\mu(r_1))\|$, and $\alpha_2 \cdot q' \cdot v' \in \|\mathsf{ss}(\mu(r_2))\|$. We can apply the induction hypothesis for r_1 to get that $\mathsf{s}_{r_1}(u') \models^{\mathsf{max}(r_1)}_{\mathsf{min}(r_1)} \mathsf{update}_{r_1}(\alpha_1)$. This implies that $\mathsf{s}_{r_1 \cdot r_2}(u) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(1\alpha_1)$. Furthermore, from part 2 of Lemma 3.5.16, there are $p_1, \ldots, p_i \in \mathsf{pos}(r_2)$ such that $\alpha_2 = \uparrow_{p_1} \cdots \uparrow_{p_i}$. Combined with part 1 of the current lemma, and the fact that for each $p_k \in \{p_1, \ldots, p_i\}$, $\mathsf{max}(r)(p_i) \geq 1$, we get that $\mathsf{s}_{r_1 \cdot r_2}(u) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(1\alpha_1 \cdot 2\alpha_2)$.

For the induction case where $r = r_1^{l..n}$ and $up\alpha \in \|\mathsf{ss}(\mu(r_1^{l..n}))\|$, we have, by definition, that there are u_1, u_2, p', α such that $u = u_1 \cdot 1u_2$, $p = 1p'$, $\alpha = 1\alpha_1 \cdot \downarrow_{\langle 1\rangle}$, and $u_2 p' \alpha_1 \in \|\mathsf{ss}(\mu(r_1))\|$. From the induction hypothesis for r_1, $\mathsf{s}_{r_1}(u_2) \models^{\mathsf{max}(r_1)}_{\mathsf{min}(r_1)} \mathsf{update}_{r_1}(\alpha_1)$. Combined with part 1, this implies that $\mathsf{s}_{r_1^{l..n}}(u) \models^{\mathsf{max}(r_1^{l..n})}_{\mathsf{min}(r_1^{l..n})} \mathsf{update}_{r_1^{l..n}}(1\alpha_1)$. Lastly, we must assure that there are at least l instances of $\uparrow_{\langle 1\rangle}$ in u. Since this follows from the definition, the lemma holds. Hence

$$\mathsf{s}_{r_1^{l..n}}(u) \models^{\mathsf{max}(r_1^{l..n})}_{\mathsf{min}(r_1^{l..n})} \mathsf{update}_{r_1^{l..n}}(1\alpha_1 \cdot \downarrow_{\langle 1\rangle}).$$

For the case where $up\alpha qv \in \|\mathsf{ss}(\mu(r_1^{l..n}))\|$, we first treat the case where there are no instances of $\uparrow_{\langle 1\rangle}$ in α. Intuitively, this implies that p and q are from the same iteration of r_1. Formally, there are $w_1, \ldots, w_i$, p_1, q_1, α_1, u_1, v_1, v_2 such that $w_1, \ldots, w_i, u_1 p_1 \alpha_1 q_1 v_1 \in \|\mathsf{ss}(\mu(r_1))\|$, $v = 1v_1 \cdot v_2$, and $u = \uparrow_{\langle 1\rangle} \cdot 1w_1 \cdots \uparrow_{\langle 1\rangle} \cdot 1w_i \cdot \uparrow_{\langle 1\rangle} \cdot 1u_1$. We can apply the induction hypothesis for r_1 to get

$$\mathsf{s}_{r_1}(u_1) \models^{\mathsf{max}(r_1)}_{\mathsf{min}(r_1)} \mathsf{update}_{r_1}(\alpha_1).$$

From part 1 of the current lemma, for any p such that $\uparrow_p \in \Gamma_{r_1}$ or $\downarrow_p \in \Gamma_{r_1}$, $\mathsf{s}_r(\uparrow_{\langle 1\rangle} \cdot 1w_1 \cdots \uparrow_{\langle 1\rangle} \cdot 1w_i \cdot \uparrow_{\langle 1\rangle})(1p) = 0$, hence

$$\mathsf{s}_{(1r_1)}(u) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(1\alpha_1),$$

which implies the lemma holds.

If there is an instance of $\uparrow_{\langle 1\rangle}$ in α, we get from parts 3 and 2 of Lemma 3.5.16 that there are $p_1, \ldots, p_i, \alpha_1, u_1, u_2$ such that $\alpha = \alpha_1 \cdot \uparrow_{\langle 1\rangle} \cdot \uparrow_{p_1} \cdots \uparrow_{p_i}$, $u = u_1 \cdot \uparrow_{\langle 1\rangle} \cdot u_2$, and $u_2 p \alpha_1 \in 1\|\mathsf{ss}(\mu(r_1))\|$. By the induction hypothesis for r_1, and part 1 of the current lemma

$$\mathsf{s}_{r_1^{l..n}}(u) \models^{\mathsf{max}(r_1^{l..n})}_{\mathsf{min}(r_1^{l..n})} \mathsf{update}_{r_1^{l..n}}(\alpha_1 \uparrow_{p_1} \cdots \uparrow_{p_i}).$$

By definition, there are less than n instances of $\uparrow_{\langle 1\rangle}$ in u, so we also have

$$\mathsf{s}_{r_1^{l..n}}(u) \models^{\mathsf{max}(r_1^{l..n})}_{\mathsf{min}(r_1^{l..n})} \mathsf{update}_{r_1^{l..n}}(\alpha).$$

For the induction case where $r = \&(r_1, \ldots, r_n)$ we treat first the case where $up\alpha \in \|\mathsf{ss}(\mu(r))\|$. By definition and part 3 of Lemma 3.5.16 there are then $i, u_1, u_2, p_1, \alpha_1$ such that $i \in \{1, \ldots, n\}$, $u = u_1 \cdot \uparrow_{\langle i\rangle} \cdot iu_2$, $u_2 p_1 \alpha_1 \in \|\mathsf{ss}(\mu(r_i))\|$, and $\alpha = i\alpha_1 \cdot \downarrow_{\langle 1\rangle} \cdots \downarrow_{\langle n\rangle}$. From the induction hypothesis for r_i we get that

$$\mathsf{s}_{r_i}(u_2) \models^{\mathsf{max}(r_i)}_{\mathsf{min}(r_i)} \mathsf{update}_{r_i}(\alpha_1).$$

Note now that the set of positions occurring on the arrows in $u_1 \cdot \uparrow_{\langle i\rangle}$ does not overlap with the set of positions occurring on the arrows in α_1. Hence,

$$\mathsf{s}_r(u_1 \cdot \uparrow_{\langle i\rangle} \cdot iu_2) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(i\alpha_1).$$

Recall from Definition 3.5.17 that

$$\mathsf{update}_r(\downarrow_{\langle 1\rangle}\cdots\downarrow_{\langle n\rangle}) = \{\langle 1\rangle \mapsto \mathsf{res}, \ldots, \langle n\rangle \mapsto \mathsf{res}\}$$

From Definition 3.4.1 and Definition 3.5.18, we now only need to check that for each $j \in \{1,\ldots,n\}$, the number of $\uparrow_{\langle j\rangle}$ in u is at least $\mathsf{min}(r)(\langle j\rangle)$. By definition, for all $j \in \{1,\ldots,n\}$ if $r_j \notin \mathfrak{N}_\Sigma$, there is exactly one $\uparrow_{\langle j\rangle}$ in $u_1 \cdot \uparrow_{\langle i\rangle}$. Furthermore, from Definition 3.5.19 we have that $\mathsf{min}(r)(\langle j\rangle)$ is 1 when $r_j \notin \mathfrak{N}_\Sigma$, and otherwise it is 0. Thus we have the lemma:

$$\mathsf{s}_r(u_1 \cdot \uparrow_{\langle i\rangle} \cdot iu_2) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(i\alpha_1 \cdot \downarrow_{\langle 1\rangle}\cdots\downarrow_{\langle n\rangle}).$$

If $up\alpha qv \in \|\mathsf{ss}(\mu(\&(r_1,\ldots,r_n)))\|$ we treat first the case where there is no $i \in \{1,\ldots,n\}$ such that $\uparrow_{\langle i\rangle}$ is in α. By definition there are then $i, u_1, u_2, v_1, v_2, p_1, q_1, \alpha_1$ such that $i \in \{1,\ldots,n\}$, $u = u_1 \cdot \uparrow_{\langle i\rangle} \cdot iu_2$, $v = iv_1 \cdot v_2$, $u_2 p_1 \alpha_1 q_1 v_1 \in \|\mathsf{ss}(\mu(r_i))\|$, and $\alpha = i\alpha_1$. From the induction hypothesis for r_i we get that

$$\mathsf{s}_{r_i}(u_2) \models^{\mathsf{max}(r_i)}_{\mathsf{min}(r_i)} \mathsf{update}_{r_1}(\alpha_1).$$

Note now that the set of positions occurring on the arrows in $u_1 \cdot \uparrow_{\langle i\rangle}$ does not overlap with the set of positions occurring on the arrows in $i\alpha_1$. Hence,

$$\mathsf{s}_r(u_1 \cdot \uparrow_{\langle i\rangle} \cdot iu_2) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(i\alpha_1).$$

Lastly, we treat the case where $up\alpha qv \in \|\mathsf{ss}(\mu(\&(r_1,\ldots,r_n)))\|$ and there is $i \in \{1,\ldots,n\}$ such that $\uparrow_{\langle i\rangle}$ is in α. By definition and part 3 of Lemma 3.5.16 there are then $i, j, u_1, u_2, p_1, \alpha_1$ such that $i, j \in \{1,\ldots,n\}$, $i \neq j$, $u = u_1 \cdot \uparrow_{\langle i\rangle} \cdot iu_2$, $u_2 p_1 \alpha_1 \in \|\mathsf{ss}(\mu(r_i))\|$, $\alpha_2 q_1 v_1 \in \|\mathsf{ss}(\mu(r_j))\|$, and $\alpha = i\alpha_1 \cdot \uparrow_{\langle j\rangle} j\alpha_2$. The induction hypothesis for r_i then implies that that

$$\mathsf{s}_r(iu_2) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(i\alpha_1).$$

From part 2 of Lemma 3.5.16 there are $p_2,\ldots,p_k \in \mathsf{pos}(r_j)$ such that $\alpha_2 = \uparrow_{p_2}\cdots\uparrow_{p_k}$. From Definition 3.5.17 we get then that $\mathsf{update}_{r_j}(\alpha_2) = \{p_2 \mapsto \mathsf{inc}, \ldots, p_k \mapsto \mathsf{inc}\}$. Since the positions $ip_2,\ldots,ip_k$ do not occur on any arrows in $u_1 \cdot \uparrow_{\langle i\rangle} \cdot iu_2$, we get that for $m \in \{2,\ldots,k\}$: $\mathsf{s}_r(u)(p_m) = 0$. Since $\mathsf{max}(r)$ has $\mathbb{N}_1$ as co-domain, we get

$$\mathsf{s}_r(u_1 \cdot iu_2) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(i\alpha_1 \cdot j\alpha_2).$$

Lastly, there can by definition not be any $\uparrow_{\langle j\rangle}$ in u, hence also

$$\mathsf{s}_r(u_1 \cdot iu_2) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(i\alpha_1 \cdot \uparrow_{\langle j\rangle} \cdot j\alpha_2).$$

□

3.5.3 Unambiguity

We can now define the right unambiguity we need for constructing deterministic automata. *Strongly 1-unambiguous* regular expressions were first defined by Koch & Scherzinger [47], but the definitions used here also bear on Gelade et al. [23]. Section 3.7 describes how a deterministic FAC can be constructed in polynomial time from such expressions.

We recall the definition of 1-unambiguity such that the difference with strong 1-unambiguity becomes clear.

Definition 3.5.22 (1-unambiguity[10, 12]). *A regular expression r is 1-unambiguous if for all $upv, uqw \in \|\mu(r)\|$, where $u, v \in (\mathsf{pos}(r))^*$ and $p, q \in \mathsf{pos}(r)$, $r[p] = r[q]$ implies $p = q$.*

Strong 1-unambiguity is needed to prevent unambiguities related to the numerical constraints. For example, $(a^{3..4})^2$ is 1-unambiguous, but there is an ambiguity related to which of the two numerical constraints should be increased when seeing the fourth a in a word. This corresponds to the fact that there are u, v, w such that both $u \cdot a \cdot \uparrow_{\langle 1,1\rangle} \cdot a \cdot v$ and $u \cdot a \cdot \downarrow_{\langle 1,1\rangle} \cdot \uparrow_{\langle 1\rangle} \cdot \uparrow_{\langle 1,1\rangle} \cdot a \cdot w$ are words in $\|\mathsf{ss}((a^{3..4})^2)\|$.

Definition 3.5.23 (Strong 1-unambiguity[23, 47]). *A regular expression r is strongly 1-unambiguous if it is 1-unambiguous, and for all $u\alpha av, u\beta bw \in \|\mathsf{ss}(r)\|$, where $a, b \in \mathsf{sym}(r)$, $\alpha, \beta \in \Gamma_r^*$ and $u, v, w \in (\Sigma \cup \Gamma_r)^*$, $a = b$ implies $\alpha = \beta$.*

Examples of expressions that are not strongly 1-unambiguous are $(a^{1..2})^{1..2}$, $(a^*a)^{2..3}$ and $\&(a^{1..2}, b)^{1..2}$, while $(a+b)^{1..4}$ is strongly 1-unambiguous. For some of the expressions that are not strongly 1-unambiguous, we can multiply the numerical constraints to possibly get strongly 1-unambiguous expressions. In general, for regular expressions of the form $(r^{l_1..u_1})^{l_2..u_2}$, if $l_2 \geq \frac{l_1-1}{u_1-l_1}$, then

$$\|r^{l_1 \cdot l_2..u_1 \cdot u_2}\| = \|(r^{l_1..u_1})^{l_2..u_2}\|$$

For example, $\|(a^{1..2})^{1..2}\| = \|a^{1..4}\|$.

Lemma 3.5.24. *Any strongly 1-unambiguous regular expression r is in constraint normal form.*

Proof. By induction on r. All cases are easy, except where $r = r_1^{l..n}$. For the latter case, we will prove the contrapositive, that is, by assuming $r_1^{l..n}$ is not in constraint normal form, we will prove that $r_1^{l..n}$ is not strongly 1-unambiguous. By the induction hypothesis r_1 is in constraint normal form, so we must have $r_1 \in \mathfrak{N}_\Sigma$. Recall that subexpressions of the forms $\epsilon \cdot \epsilon$, $\epsilon + \epsilon$, $\&(\epsilon, \ldots, \epsilon)$, and $\epsilon^{l..n}$ are not allowed, so there must be $a \in \mathsf{sym}(r_1)$, $\alpha \in \Gamma_{r_1}^*$, and $u \in (\mathsf{sym}(r_1) \cup \Gamma_{r_1})^*$ such that $\alpha a u \in \|\mathsf{ss}(r_1)\|$, hence there is a v such that $\uparrow_{\langle 1 \rangle} \cdot 1\alpha \cdot a \cdot v \in \|\mathsf{ss}(r_1^{l..u})\|$. By Lemma 3.5.14 applied to $r_1 \in \mathfrak{N}_\Sigma$, we get $\epsilon \in \|\mathsf{ss}(r_1)\|$. Using the definition of the language of a subscripted expression we get a v' such that $\uparrow_{\langle 1 \rangle} \cdot \uparrow_{\langle 1 \rangle} \cdot 1\alpha \cdot 1p \cdot v' \in \|\mathsf{ss}(r_1^{l..u})\|$. Hence, by Definition 3.5.23, r is not strongly 1-unambiguous. $\square$

We do not provide a direct algorithm for testing that a regular expression is strongly 1-unambiguous. However, the polynomial-time construction of FACs given in Definition 3.7.1 will by Lemma 3.7.2 result in a deterministic FAC if and only if the input is a strongly 1-unambiguous regular expression. Furthermore, in the paragraph below Definition 3.4.8 we sketch a polynomial-time procedure for testing that an FAC is deterministic. Hence, we can in polynomial time test whether a regular expression is strongly 1-unambiguous.

3.6 First, Last, and Follow

Following Brüggemann-Klein & Wood [12] and Glushkov [29], we define three mappings, $\mathsf{first}, \mathsf{last}$ and follow. They are central in the construction of FACs from regular expressions. first and last both take a regular expression r as argument, while follow takes both a regular expression r and a position $p \in \mathsf{sym}(\mu(r))$ as input. All three mappings return a set of pairs. The first member in each pair is a position $p \in \mathsf{sym}(\mu(r))$, while the second member is an update instruction with domain $\langle \& \sharp \rangle(r)$. The algorithm to calculate the three mappings is adapted from the algorithm for regular expressions with numerical constraints [64, 37, 23], and follows the same pattern as for classical regular expressions [29, 53]. The algorithm uses time polynomial in the size of the regular expression.

Definition 3.6.1 (First, Last, and Follow). *Given any $r \in R_\Sigma$ in constraint normal form, and any $p \in \mathsf{sym}(\mu(r))$:*

$$\mathsf{first}(r) = \left\{ (q, \mathsf{update}_r(\alpha)) \;\middle|\; \begin{array}{l} q \in \mathsf{sym}(\mu(r)),\ \alpha \in \Gamma_r^*, \\ \exists u : \alpha q u \in \|\mathsf{ss}(\mu(r))\| \end{array} \right\}$$

$$\mathsf{last}(r) = \left\{ (q, \mathsf{update}_r(\alpha)) \;\middle|\; \begin{array}{l} q \in \mathsf{sym}(\mu(r)),\ \alpha \in \Gamma_r^*, \\ \exists u : u q \alpha \in \|\mathsf{ss}(\mu(r))\| \end{array} \right\}$$

$$\mathsf{follow}(r, p) = \left\{ (q, \mathsf{update}_r(\alpha)) \;\middle|\; \begin{array}{l} q \in \mathsf{sym}(\mu(r)),\ \alpha \in \Gamma_r^*, \\ \exists u, v : u p \alpha q v \in \|\mathsf{ss}(\mu(r))\| \end{array} \right\}$$

Example 3.6.2. *Put $r = (\&(a^2, b))^{3..4}$ as used above. Then*

$$\mathsf{first}(r) = \left\{ \begin{array}{l} (\langle 1,1,1\rangle, \{\langle 1\rangle \mapsto \mathsf{inc}, \langle 1,1\rangle \mapsto \mathsf{inc}, \langle 1,1,1\rangle \mapsto \mathsf{inc}\}), \\ (\langle 1,2\rangle, \{\langle 1\rangle \mapsto \mathsf{inc}, \langle 1,2\rangle \mapsto \mathsf{inc}\}) \end{array} \right\}$$

because there are non-empty $S, S' \subseteq (\mathsf{pos}(r) \cup \Gamma_r)^*$ such that $\|\mathsf{ss}(\mu(r))\| = (\uparrow_{\langle 1\rangle}\uparrow_{\langle 1,1\rangle}\uparrow_{\langle 1,1,1\rangle}\langle 1,1,1\rangle S) \cup (\uparrow_{\langle 1\rangle}\uparrow_{\langle 1,2\rangle}\langle 1,2\rangle \cdot S')$.

$$\mathsf{last}(r) = \left\{ \begin{array}{l} \left(\langle 1,1,1\rangle, \left\{ \begin{array}{l} \langle 1\rangle \mapsto \mathsf{res}, \langle 1,1\rangle \mapsto \mathsf{res}, \\ \langle 1,2\rangle \mapsto \mathsf{res}, \langle 1,1,1\rangle \mapsto \mathsf{res} \end{array} \right\}\right), \\ (\langle 1,2\rangle, \{\langle 1\rangle \mapsto \mathsf{res}, \langle 1,1\rangle \mapsto \mathsf{res}, \langle 1,2\rangle \mapsto \mathsf{res}\}) \end{array} \right\}$$

because there are non-empty $S, S' \subseteq (\mathsf{pos}(r) \cup \Gamma_r)^*$ such that $\|\mathsf{ss}(\mu(r))\| = (S \cdot \langle 1,1,1\rangle\downarrow_{\langle 1,1,1\rangle}\downarrow_{\langle 1,1\rangle}\downarrow_{\langle 1,2\rangle}\downarrow_{\langle 1\rangle}) \cup (S' \cdot \langle 1,2\rangle\downarrow_{\langle 1,1\rangle}\downarrow_{\langle 1,2\rangle}\downarrow_{\langle 1\rangle})$.

$$\mathsf{follow}(r, \langle 1,1,1\rangle) =$$
$$\left\{ \begin{array}{l} (\langle 1,2\rangle, \{\langle 1\rangle \mapsto \mathsf{inc}, \langle 1,2\rangle \mapsto \mathsf{one}, \langle 1,1\rangle \mapsto \mathsf{res}, \langle 1,1,1\rangle \mapsto \mathsf{res}\}) \\ (\langle 1,2\rangle, \{\langle 1,2\rangle \mapsto \mathsf{inc}, \langle 1,1,1\rangle \mapsto \mathsf{res}\}) \\ (\langle 1,1,1\rangle, \{\langle 1\rangle \mapsto \mathsf{inc}, \langle 1,1\rangle \mapsto \mathsf{one}, \langle 1,1,1\rangle \mapsto \mathsf{one}, \langle 1,2\rangle \mapsto \mathsf{res}\}) \\ (\langle 1,1,1\rangle, \{\langle 1,1,1\rangle \mapsto \mathsf{inc}\}) \end{array} \right\}$$

because there are non-empty $S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8 \subseteq (\mathsf{pos}(r) \cup \Gamma_r)^*$ such that $\|\mathsf{ss}(\mu(r))\| \cap ((\mathsf{pos}(r) \cup \Gamma_r)^* \cdot \langle 1,1,1\rangle \cdot (\mathsf{pos}(r) \cup \Gamma_r)^*) =$

$$\begin{aligned} & S_1\langle 1,1,1\rangle\downarrow_{\langle 1,1,1\rangle}\downarrow_{\langle 1,1\rangle}\downarrow_{\langle 1,2\rangle}\uparrow_{\langle 1\rangle}\uparrow_{\langle 1,2\rangle}\langle 1,2\rangle S_2 \\ \cup\ & S_3\langle 1,1,1\rangle\downarrow_{\langle 1,1,1\rangle}\uparrow_{\langle 1,2\rangle}\langle 1,2\rangle S_4 \\ \cup\ & S_5\langle 1,1,1\rangle\downarrow_{\langle 1,1,1\rangle}\downarrow_{\langle 1,1\rangle}\downarrow_{\langle 1,2\rangle}\uparrow_{\langle 1\rangle}\uparrow_{\langle 1,1\rangle}\uparrow_{\langle 1,1,1\rangle}\langle 1,1,1\rangle S_6 \\ \cup\ & S_7\langle 1,1,1\rangle\uparrow_{\langle 1,1,1\rangle}\langle 1,1,1\rangle S_8. \end{aligned}$$

$$\mathsf{follow}(r,\langle 1,2\rangle) = \left\{\begin{array}{l} (\langle 1,2\rangle, \{\langle 1\rangle \mapsto \mathsf{inc}, \langle 1,1\rangle \mapsto \mathsf{res}, \langle 1,2\rangle \mapsto \mathsf{one}\}) \\ (\langle 1,1,1\rangle, \{\langle 1\rangle \mapsto \mathsf{inc}, \langle 1,1\rangle \mapsto \mathsf{one}, \langle 1,2\rangle \mapsto \mathsf{res}, \langle 1,1,1\rangle \mapsto \mathsf{inc}\}) \\ (\langle 1,1,1\rangle, \{\langle 1,1,1\rangle \mapsto \mathsf{inc}, \langle 1,1\rangle \mapsto \mathsf{inc}\}) \end{array}\right\}$$

because there are non-empty $S_1, S_2, S_3, S_4, S_5, S_6 \subseteq (\mathsf{pos}(r) \cup \Gamma_r)^*$ *such that* $\|\mathsf{ss}(\mu(r))\| \cap ((\mathsf{pos}(r) \cup \Gamma_r)^* \cdot \langle 1,2\rangle \cdot (\mathsf{pos}(r) \cup \Gamma_r)^*) =$

$$\begin{array}{l} \ S_1\langle 1,2\rangle{\downarrow_{\langle 1,1\rangle}}{\downarrow_{\langle 1,2\rangle}}{\uparrow_{\langle 1\rangle}}{\uparrow_{\langle 1,2\rangle}}\langle 1,2\rangle S_2 \\ \cup\ S_3\langle 1,2\rangle{\downarrow_{\langle 1,1\rangle}}{\downarrow_{\langle 1,2\rangle}}{\uparrow_{\langle 1\rangle}}{\uparrow_{\langle 1,1\rangle}}{\uparrow_{\langle 1,1,1\rangle}}\langle 1,1,1\rangle S_4 \\ \cup\ S_5\langle 1,2\rangle{\uparrow_{\langle 1,1\rangle}}{\uparrow_{\langle 1,1,1\rangle}}\langle 1,1,1\rangle S_6. \end{array}$$

3.6.1 Calculating first, last, and follow

We show here an inductive definition of first, last, and follow. The inductive definition follows the pattern first used by Glushkov [29] for the classical regular expressions, but here the situation is much more complicated. Note that the construction only works for regular expressions in constraint normal form.

We define in Definitions 3.6.3 and 3.6.4 three auxiliary mappings first^μ, last^μ, and follow^μ. first^μ and last^μ take a regular expression and a position in the term tree of this expression as input. follow^μ takes in addition a second position in the subtree below the first position. Then, for any regular expression r, we calculate $\mathsf{first}(r)$ by $\mathsf{first}^\mu(r,\langle\rangle)$, $\mathsf{last}(r)$ by $\mathsf{last}^\mu(r,\langle\rangle)$, and for $p \in \mathsf{sym}(\mu(r))$, we calculate $\mathsf{follow}(r,p)$ by $\mathsf{follow}^\mu(r,\langle\rangle,p)$. Note that $\mathsf{first}^\mu(r,p)$ and $\mathsf{last}^\mu(r,p)$ are only defined for $p \in \mathsf{pos}(r)$, and $\mathsf{follow}^\mu(r,p,q)$ is only defined when $q \in \mathsf{sym}(\mu(r[p]))$, and $p \in \mathsf{pos}(r)$.

Definition 3.6.3 (first$^\mu$ and last$^\mu$). *For any regular expression r in constraint normal form, and any $p \in \mathsf{pos}(r)$:*

$$r[p] = \epsilon \;\Rightarrow\; \mathsf{first}^\mu(r,p) = \mathsf{last}^\mu(r,p) = \varnothing$$

$$r[p] \in \Sigma \Rightarrow \mathsf{first}^\mu(r,p) = \mathsf{last}^\mu(r,p) = \{(p,\varnothing)\}$$

$$\begin{aligned} r[p] = r_1 + r_2 \Rightarrow \\ \mathsf{first}^\mu(r,p) = \mathsf{first}^\mu(r,p1) \cup \mathsf{first}^\mu(r,p2) \wedge \\ \mathsf{last}^\mu(r,p) = \mathsf{last}^\mu(r,p1) \cup \mathsf{last}^\mu(r,p2) \end{aligned}$$

$$\begin{aligned} r[p] = r_1 \cdot r_2 \wedge r_1 \in \mathfrak{N}_\Sigma \;\Rightarrow \\ \mathsf{first}^\mu(r,p) = \mathsf{first}^\mu(r,p1) \cup \mathsf{first}^\mu(r,p2) \end{aligned}$$

$$\begin{aligned} r[p] = r_1 \cdot r_2 \wedge r_2 \in \mathfrak{N}_\Sigma \;\Rightarrow \\ \mathsf{last}^\mu(r,p) = \mathsf{last}^\mu(r,p1) \cup \mathsf{last}^\mu(r,p2) \end{aligned}$$

$$\begin{aligned} r[p] = r_1 \cdot r_2 \wedge r_1 \notin \mathfrak{N}_\Sigma \;\Rightarrow\; \mathsf{first}^\mu(r,p) = \mathsf{first}^\mu(r,p1) \\ r[p] = r_1 \cdot r_2 \wedge r_2 \notin \mathfrak{N}_\Sigma \;\Rightarrow\; \mathsf{last}^\mu(r,p) = \mathsf{last}^\mu(r,p2) \end{aligned}$$

$$\begin{aligned} r[p] = r_1^{n..m} \;\Rightarrow \\ \mathsf{first}^\mu(r,p) = \{(q, \psi \cup \{p1 \mapsto \mathsf{inc}\}) \mid (q,\psi) \in \mathsf{first}^\mu(r,p1)\} \wedge \\ \mathsf{last}^\mu(r,p) = \{(q, \psi \cup \{p1 \mapsto \mathsf{res}\}) \mid (q,\psi) \in \mathsf{last}^\mu(r,p1)\} \end{aligned}$$

$$\begin{aligned} r[p] = \&(r_1,\ldots,r_n) \;\Rightarrow \\ \mathsf{first}^\mu(r,p) = \{(q, \psi \cup \{\, pi \mapsto \mathsf{inc}\}) \mid \exists i : 1 \leq i \leq n, (q,\psi) \in \mathsf{first}^\mu(r,pi)\} \wedge \\ \mathsf{last}^\mu(r,p) = \{(q, \psi \cup \{pj{\mapsto}\mathsf{res} \mid 1{\leq}j{\leq}n\}) \mid \exists i : 1{\leq}i{\leq}n, (q,\psi){\in}\mathsf{last}^\mu(r,pi)\} \end{aligned}$$

Definition 3.6.4 (follow$^\mu$). *For any regular expression r in constraint normal form, and any $p \in \mathsf{pos}(r)$:*

$$r[p] \in \Sigma \cup \{\epsilon\} \;\Rightarrow\; \mathsf{follow}^\mu(r, p, \langle\rangle) = \varnothing$$

$$r[p] = r_1 + r_2 \;\Rightarrow\; \mathsf{follow}^\mu(r, p, 1q) = \mathsf{follow}^\mu(r, p1, q)$$
$$r[p] = r_1 + r_2 \;\Rightarrow\; \mathsf{follow}^\mu(r, p, 2q) = \mathsf{follow}^\mu(r, p2, q)$$

$$r[p] = r_1 \cdot r_2 \;\Rightarrow\; \mathsf{follow}^\mu(r, p, 2q) = \mathsf{follow}^\mu(r, p2, q)$$

$$\begin{aligned} r[p] = r_1 \cdot r_2 \;\wedge\; (p \odot 1q, \psi) \in \mathsf{last}^\mu(r, p1) \;\Rightarrow& \\ \mathsf{follow}^\mu(r, p, 1q) =& \\ \mathsf{follow}^\mu(r, p1, q) \,\cup\, \{(q', \psi \cup \psi') \mid (q', \psi') \in \mathsf{first}^\mu(r, p2)\}& \end{aligned}$$

$$\begin{aligned} r[p] = r_1 \cdot r_2 \;\wedge\; (\nexists \psi : (p \odot 1q, \psi) \in \mathsf{last}^\mu(r, q1)) \;\Rightarrow& \\ \mathsf{follow}^\mu(r, p, 1q) = \mathsf{follow}^\mu(r, p1, q)& \end{aligned}$$

$$\begin{aligned} r[p]{=}\&(r_1, \ldots, r_n) \;\wedge\; i{\in}\{1, \ldots, n\} \;\wedge\; (\nexists \psi : (p \odot iq, \psi){\in}\mathsf{last}^\mu(r, pi)) \;\Rightarrow& \\ \mathsf{follow}^\mu(r, p, iq) = \mathsf{follow}^\mu(r, pi, q)& \end{aligned}$$

$$\begin{aligned} r[p] = \&(r_1, \ldots, r_n) \;\wedge\; i \in \{1, \ldots, n\} \;\wedge\; (p \odot iq, \psi) \in \mathsf{last}^\mu(r, pi) \;\Rightarrow& \\ \mathsf{follow}^\mu(r, p, iq) =& \\ \mathsf{follow}^\mu(r, pi, q) \,\cup\, \{(q', \psi \cup \psi') \mid (q', \psi') \in \mathsf{first}^\mu(r, p), pi \not\leq q'\}& \end{aligned}$$

$$\begin{aligned} r[p] = r_1^{n..m} \;\Rightarrow& \\ \mathsf{follow}^\mu(r, p, 1q) =& \\ \mathsf{follow}^\mu(r, p1, q) \cup \{\psi \uplus \psi' \mid (p \odot 1q, \psi){\in}\mathsf{last}^\mu(r, p1), (q', \psi') \in \mathsf{first}^\mu(r, p)\}& \end{aligned}$$
$$\textit{where } \psi \uplus \psi' = \begin{cases} c \mapsto \mathsf{one} & \textit{if } \psi(c) = \mathsf{res} \wedge \psi'(c) = \mathsf{inc} \\ c \mapsto \mathsf{res} & \textit{if } \psi(c) = \mathsf{res} \wedge c \notin \mathsf{dom}(\psi') \\ c \mapsto \mathsf{inc} & \textit{if } \psi'(c) = \mathsf{inc} \wedge c \notin \mathsf{dom}(\psi) \end{cases}$$

Example 3.6.5. *Put $r = (\&(a^2, b))^{3..4}$ like in Example 3.6.2*

$$\begin{aligned}
\mathsf{last}^{\mu}(r, \langle 1,2\rangle) &= \{(\langle 1,2\rangle, \varnothing)\}\\
\mathsf{last}^{\mu}(r, \langle 1,1,1\rangle) &= \{(\langle 1,1,1\rangle, \varnothing)\}\\
\mathsf{last}^{\mu}(r, \langle 1,1\rangle) &= \{(\langle 1,1,1\rangle, \{\langle 1,1,1\rangle \mapsto \mathsf{res}\})\}\\
\mathsf{last}^{\mu}(r, \langle 1\rangle) &= \left\{ \begin{array}{l} \left(\langle 1,1,1\rangle, \left\{ \begin{array}{l} \langle 1,1\rangle \mapsto \mathsf{res},\\ \langle 1,2\rangle \mapsto \mathsf{res},\\ \langle 1,1,1\rangle \mapsto \mathsf{res} \end{array}\right\}\right),\\ (\langle 1,2\rangle, \{\langle 1,1\rangle \mapsto \mathsf{res}, \langle 1,2\rangle \mapsto \mathsf{res}\}) \end{array} \right\}\\
\mathsf{last}^{\mu}(r, \langle\rangle) &= \mathsf{last}(r)
\end{aligned}$$

$$\begin{aligned}
\mathsf{first}^{\mu}(r, \langle 1,2\rangle) &= \{(\langle 1,2\rangle, \varnothing)\}\\
\mathsf{first}^{\mu}(r, \langle 1,1,1\rangle) &= \{(\langle 1,1,1\rangle, \varnothing)\}\\
\mathsf{first}^{\mu}(r, \langle 1,1\rangle) &= \{(\langle 1,1,1\rangle, \{\langle 1,1,1\rangle \mapsto \mathsf{inc}\})\}\\
\mathsf{first}^{\mu}(r, \langle 1\rangle) &= \left\{ \begin{array}{l} (\langle 1,1,1\rangle, \{\langle 1,1\rangle \mapsto \mathsf{inc}, \langle 1,1,1\rangle \mapsto \mathsf{inc}\}),\\ (\langle 1,2\rangle, \{\langle 1,2\rangle \mapsto \mathsf{inc}\}) \end{array} \right\}\\
\mathsf{first}^{\mu}(r, \langle\rangle) &= \mathsf{first}(r)
\end{aligned}$$

$$\begin{aligned}
\mathsf{follow}^{\mu}(r, \langle 1,1\rangle, \langle 1\rangle) &= \{(\langle 1,1,1\rangle, \{\langle 1,1,1\rangle \mapsto \mathsf{inc}\})\}\\
\mathsf{follow}^{\mu}(r, \langle 1\rangle, \langle 1,1\rangle) &= \left\{ \begin{array}{l} (\langle 1,1,1\rangle, \{\langle 1,1,1\rangle \mapsto \mathsf{inc}\}),\\ (\langle 1,2\rangle, \{\langle 1,2\rangle \mapsto \mathsf{inc}, \langle 1,1,1\rangle \mapsto \mathsf{res}\}) \end{array} \right\}\\
\mathsf{follow}^{\mu}(r, \langle\rangle, \langle 1,1,1\rangle) &= \mathsf{follow}(r, \langle 1,1,1\rangle)
\end{aligned}$$

$$\begin{aligned}
\mathsf{follow}^{\mu}(r, \langle 1\rangle, \langle 2\rangle) &= \{(\langle 1,1,1\rangle, \{\langle 1,1\rangle \mapsto \mathsf{inc}, \langle 1,1,1\rangle \mapsto \mathsf{inc}\})\}\\
\mathsf{follow}^{\mu}(r, \langle\rangle, \langle 1,2\rangle) &= \mathsf{follow}(r, \langle 1,2\rangle)
\end{aligned}$$

That $\mathsf{first}^{\mu}(r,p)$, $\mathsf{last}^{\mu}(r,p)$ and $\mathsf{follow}^{\mu}(r,p,q)$ are defined for their whole domains is shown by an easy induction on $r[p]$. We must prove that these calculations satisfy the specifications in Definition 3.6.1.

Lemma 3.6.6. *For any regular expression r, and any positions $q \in \mathsf{pos}(r)$, and $p \in \mathsf{sym}(\mu(r[q]))$, $\mathsf{first}^{\mu}(r,q) = q \odot \mathsf{first}(r[q])$, $\mathsf{last}^{\mu}(r,q) = q \odot \mathsf{last}(r[q])$, and $\mathsf{follow}^{\mu}(r,q,p) = q \odot \mathsf{follow}(r[q],p)$.*

Proof. By a simultaneous induction on $r[q]$.

For the base case $r[q] = \epsilon$, all calculations and definitions give the empty set.

For the base case $r[q] \in \Sigma$, note that $p = \langle\rangle$ and $\|\mathsf{ss}(\mu(r[q]))\| = \|\mu(r[q])\| = \{\langle\rangle\}$. first^{μ} and last^{μ} return a singleton set containing $(q, \varnothing)$, while $q \odot \mathsf{first}(r[q]) = q \odot \mathsf{last}(r[q]) = q \odot \{(\langle\rangle, \varnothing)\} = \{(q, \varnothing)\}$. Lastly, $\mathsf{follow}^{\mu}(r, q, \langle\rangle) = \mathsf{follow}(r[q], \langle\rangle) = \varnothing$.

For the induction case where $r[q] = r_1 + r_2$, we get first by Definition 3.6.3 that $\mathsf{first}^{\mu}(r, q) = \mathsf{first}^{\mu}(r, q1) \cup \mathsf{first}^{\mu}(r, q2)$, and this is by the induction hypothesis equal to $q1 \odot \mathsf{first}(r[q1]) \cup q2 \odot \mathsf{first}(r[q2])$. That the latter equals $q \odot \mathsf{first}(r[q])$ follows from

$$\|\mathsf{ss}(\mu(r[q]))\| = 1\|\mathsf{ss}(\mu(r[q1]))\| \cup 2\|\mathsf{ss}(\mu(r[q2]))\|$$

A similar argument holds for last^{μ} and last. For follow, note first that $\exists i \in \{1, 2\}, p' \in \mathsf{sym}(\mu(r[qi])) : p = ip'$. Assume some such i. From Definition 3.6.4, $\mathsf{follow}^{\mu}(r, q, ip') = \mathsf{follow}^{\mu}(r, qi, p')$. From the induction hypothesis, $\mathsf{follow}^{\mu}(r, qi, p') = qi \odot \mathsf{follow}(r[qi], p')$. We must show that $i\mathsf{follow}(r[qi], p') = \mathsf{follow}(r[q], p)$. Note now that since $\|\mathsf{ss}(\mu(r_1 + r_2))\| = 1\|\mathsf{ss}(\mu(r_1))\| \cup 2\|\mathsf{ss}(\mu(r_2))\|$, we get the needed results from the induction hypothesis and Definition 3.6.1.

For the induction case where $r[q] = r_1 \cdot r_2$, we first show that $\mathsf{first}^{\mu}(r, q) = q \odot \mathsf{first}(r_1 \cdot r_2)$. If $r_1 \notin \mathfrak{N}_{\Sigma}$, by Definition 3.6.3, $\mathsf{first}^{\mu}(r, q) = \mathsf{first}^{\mu}(r, q1)$, and from Definition 3.6.1, $\mathsf{first}(r_1 r_2) = 1\mathsf{first}(r_1)$. By applying the induction hypothesis for r_1, we therefore get $q \odot \mathsf{first}(r_1 r_2) = \mathsf{first}^{\mu}(r, q)$. On the other hand, if $r_1 \in \mathfrak{N}_{\Sigma}$, by Definition 3.6.3 $\mathsf{first}^{\mu}(r, q)$ is the same as when $r[q] = r_1 + r_2$. Furthermore, using Definition 3.6.1 and part 1 of Lemma 3.5.16, we get $\mathsf{first}(r_1 r_2) = \mathsf{first}(r_1 + r_2)$. Therefore we can use the same arguments as in the case for choice. The arguments that $\mathsf{last}^{\mu}(r, q) = q \odot \mathsf{last}(r_1 r_2)$ are similar: If $r_2 \in \mathfrak{N}_{\Sigma}$, we can use the same arguments as in the case when $r[q] = r_1 + r_2$, and if $r_2 \notin \mathfrak{N}_{\Sigma}$, we can use the induction hypothesis for r_2.

We proceed to show that when $r[q] = r_1 \cdot r_2$, $\mathsf{follow}^{\mu}(r, q, p) = q \odot \mathsf{follow}(r_1 \cdot r_2, p)$. If $q2 \leq p$, we can apply the induction hypothesis for r_2, and if $\forall \psi : (q \odot p, \psi) \notin \mathsf{last}^{\mu}(r, q1)$ we can apply the induction hypothesis for r_1. Consider the case where $\exists \psi : (q \odot p, \psi) \in \mathsf{last}^{\mu}(r, q1)$. Then there is a $p_1 \in \mathsf{pos}(r_1)$ such that $p = 1p_1$. By the induction hypothesis on r_1, $\mathsf{last}^{\mu}(r, q1) = q1 \odot \mathsf{last}(r_1)$. Hence, there is a ψ_1 such that $\psi = q1 \odot \psi_1$ and $(q1 \odot p_1, \psi_1) \in \mathsf{last}(r_1)$. By Definition 3.6.1 this implies that there are $\alpha_1 \in \Gamma^*_{r_1}$ and $u_1 \in (\mathsf{pos}(r_1) \cup \Gamma_{r_1})^*$ such that

$$\psi_1 = \mathsf{update}_{r_1}(\alpha_1) \ \wedge \ u_1 p_1 \alpha_1 \in \|\mathsf{ss}(\mu(r_1))\| \tag{3.18}$$

By Definition 3.6.4 $\mathsf{follow}^\mu(r,q,p) = \mathsf{follow}^\mu(r,q1,p_1) \cup \{(p',\psi \cup \psi') \mid (p',\psi') \in \mathsf{first}^\mu(r,q2)\}$. Applying the induction hypothesis for r_1 and r_2 to the latter equation we get

$$\begin{aligned} &\mathsf{follow}^\mu(r,q,p) = \\ (q1 \odot \mathsf{follow}(r_1,p_1)) \cup \{(p',\psi \cup \psi') \mid (p',\psi') &\in q2 \odot \mathsf{first}(r_2)\} \end{aligned} \tag{3.19}$$

By Definition 3.6.1 and (3.18)

$$\{(p',\psi \cup \psi') \mid (p',\psi') \in q2 \odot \mathsf{first}(r_2)\} = \left\{ q \odot (p', \mathsf{update}_r(\alpha_1 \cdot \alpha_2)) \;\middle|\; \begin{array}{l} p' \in 2\mathsf{pos}(r_2), \\ \alpha_1 \in 1\Gamma^*_{r_1}, \\ \alpha_2 \in 2\Gamma^*_{r_2}, \\ \exists u : up\alpha_1 \in 1\|\mathsf{ss}(\mu(r_1))\|, \\ \exists u : \alpha_2 p' u \in 2\|\mathsf{ss}(\mu(r_2))\| \end{array} \right\} \tag{3.20}$$

By Definition 3.6.1

$$q1 \odot \mathsf{follow}(r_1,p_1) = q \odot \left\{ (p', \mathsf{update}_{r_1 \cdot r_2}(\alpha')) \;\middle|\; \begin{array}{l} p' \in 1\mathsf{pos}(r_1), \\ \alpha' \in 1\Gamma^*_{r_1}, \\ \exists u,v : up\alpha' p' v \in 1\|\mathsf{ss}(\mu(r_1))\| \end{array} \right\} \tag{3.21}$$

and

$$q \odot \mathsf{follow}(r_1 \cdot r_2, p) = q \odot \left\{ (p', \mathsf{update}_{r_1 \cdot r_2}(\alpha')) \;\middle|\; \begin{array}{l} p' \in \mathsf{pos}(r_1 \cdot r_2), \\ \alpha' \in \Gamma^*_{r_1 \cdot r_2}, \\ \exists u,v : up_1\alpha' p' v \in \|\mathsf{ss}(\mu(r_1 \cdot r_2))\| \end{array} \right\} \tag{3.22}$$

Since the p' in (3.22) can originate in r_1 or r_2 we use (3.22) and (3.18) to get

$$
\begin{aligned}
&q \odot \mathsf{follow}(r[q], p) = \\
&q \odot \left\{ (p', \mathsf{update}_r(\alpha')) \;\middle|\; \begin{array}{l} \left(\begin{array}{l} p' \in 1\mathsf{pos}(r_1), \alpha' \in 1\Gamma^*_{r_1}, \\ \exists u, v : u p_1 \alpha' p' v \in 1\|\mathsf{ss}(\mu(r_1))\| \end{array} \right) \vee \\ \left(\begin{array}{l} p' \in 2\mathsf{pos}(r_2), \alpha_1 \in 1\Gamma^*_{r_1}, \\ \alpha_2 \in 2\Gamma^*_{r_2}, \alpha' = \alpha_1 \cdot \alpha_2, \\ \exists u : u p_1 \alpha_1 \in 1\|\mathsf{ss}(\mu(r_1))\|, \\ \exists v : \alpha_2 p' v \in 2\|\mathsf{ss}(\mu(r_2))\| \end{array} \right) \end{array} \right\} = \\
&q \odot \left\{ (p', \mathsf{update}_r(\alpha')) \;\middle|\; \begin{array}{l} p' \in 1\mathsf{pos}(r_1), \alpha' \in 1\Gamma^*_{r_1}, \\ \exists u, v : u p \alpha' p' v \in 1\|\mathsf{ss}(\mu(r_1))\| \end{array} \right\} \bigcup \\
&q \odot \left\{ (p', \mathsf{update}_r(\alpha_1 \cdot \alpha_2)) \;\middle|\; \begin{array}{l} p' \in 2\mathsf{pos}(r_2), \\ \alpha_1 \in 1\Gamma^*_{r_1}, \alpha_2 \in 2\Gamma^*_{r_2}, \\ \exists u : u p \alpha_1 \in 1\|\mathsf{ss}(\mu(r_1))\|, \\ \exists v : \alpha_2 p' v \in 2\|\mathsf{ss}(\mu(r_2))\| \end{array} \right\}
\end{aligned}
\tag{3.23}
$$

By applying (3.20) and (3.21) to (3.23), and then applying (3.19) we get $q \odot \mathsf{follow}(r[q], p) = \mathsf{follow}^{\mu}(r, q, p)$.

For the induction case where $r[q] = r_1^{l..u}$ we treat first and follow in more detail. Note that

$$
\begin{aligned}
&\|\mathsf{ss}(\mu(r[q]))\| \\
&\quad =\text{(by assumption)} \\
&\quad\quad \|\mathsf{ss}(\mu(r_1^{l..u}))\| \\
&\quad =(\text{ since } \mu(r_1^{l..u}) = (1\mu(r_1))^{l..u}) \\
&\quad\quad \|\mathsf{ss}((1\mu(r_1))^{l..u})\| \\
&\quad =\text{(by Definition 3.5.11)} \\
&\quad\quad \|(\odot_1 \mathsf{ss}(1\mu(r_1)))^{l..u}_{\langle\rangle}\| \\
&\quad =\text{(by definition of concatenating with positions)} \\
&\quad\quad \|(1\mathsf{ss}(\mu(r_1)))^{l..u}_{\langle\rangle}\| \\
&\quad =\text{(by definition of the language of a subscripted expression)} \\
&\quad\quad (\bigcup_{i=l}^{u}(\{\uparrow_{\langle 1\rangle}\} \cdot \|1\mathsf{ss}(\mu(r_1))\|)^i) \cdot \{\downarrow_{\langle 1\rangle}\} \\
&\quad =\text{(by definition of concatenating with positions)} \\
&\quad\quad (\bigcup_{i=l}^{u}(\{\uparrow_{\langle 1\rangle}\} \cdot (1\|\mathsf{ss}(\mu(r_1))\|))^i) \cdot \{\downarrow_{\langle 1\rangle}\} \\
&\quad =\text{(by Definition 3.5.1)} \\
&\quad\quad 1((\bigcup_{l\le i\le u}(\{\uparrow_{\langle\rangle}\} \cdot \|\mathsf{ss}(\mu(r_1))\|)^i) \cdot \{\downarrow_{\langle\rangle}\})
\end{aligned}
\tag{3.24}
$$

We use (3.24) taking into account that $r_1 \notin \mathfrak{N}_\Sigma$, and we get

$$
\begin{gathered}
\{(p,\alpha) \mid p \in \mathsf{sym}(\mu(r_1^{l..u})), \alpha \in \Gamma_{r_1^{l..u}}{}^*, \exists u : \alpha p u \in \|\mathsf{ss}(\mu(r_1^{l..u}))\|\} = \\
\{(1p, \uparrow_{\langle 1\rangle} 1\alpha) \mid p \in \mathsf{sym}(\mu(r_1)), \alpha \in \Gamma_{r_1}{}^*, \exists u : \alpha p u \in \|\mathsf{ss}(\mu(r_1))\|\}
\end{gathered}
\tag{3.25}
$$

We use (3.25) together with Definitions 3.5.17 and 3.6.1 to get

$$\begin{aligned}
&\mathsf{first}(r[q])\\
=&(\text{by Definition 3.6.1})\\
&\left\{\begin{array}{l}(p, \mathsf{update}_{r_1^{l..u}}(\alpha))\\ \mid\ p \in \mathsf{sym}(\mu(r_1^{l..u})), \alpha \in \Gamma^*_{r_1^{l..u}}, \exists u : \alpha p u \in \|\mathsf{ss}(\mu(r_1^{l..u}))\|\end{array}\right\}\\
=&(\text{by (3.25)})\\
&\left\{\begin{array}{l}(1p, \{\langle 1\rangle \mapsto \mathsf{inc}\} \cup 1\mathsf{update}_{r_1}(\alpha))\\ \mid\ p \in \mathsf{sym}(\mu(r_1)), \alpha \in \Gamma_{r_1}{}^*, \exists u : \alpha p u \in \|\mathsf{ss}(\mu(r_1))\|\end{array}\right\} \qquad (3.26)\\
=&(\text{by Definition 3.5.1})\\
&1\left\{\begin{array}{l}(p, \{\langle\rangle \mapsto \mathsf{inc}\} \cup \mathsf{update}_{r_1}(\alpha))\\ \mid\ p \in \mathsf{sym}(\mu(r_1)), \alpha \in \Gamma^*_{r_1}, \exists u : \alpha p u \in \|\mathsf{ss}(\mu(r_1))\|\end{array}\right\}\\
=&(\text{by Definition 3.6.1})\\
&1\{(p, \psi \cup \{\langle\rangle \mapsto \mathsf{inc}\}) \mid (p, \psi) \in \mathsf{first}(r_1)\}
\end{aligned}$$

Finally, we then get

$$\begin{aligned}
&\mathsf{first}^{\mu}(r, q)\\
=&(\text{by Definition 3.6.3})\\
&\{(p, \psi \cup \{q1 \mapsto \mathsf{inc}\}) \mid (p, \psi) \in \mathsf{first}^{\mu}(r, q1)\}\\
=&(\text{by the induction hypothesis})\\
&\{(p, \psi \cup \{q1 \mapsto \mathsf{inc}\}) \mid (p, \psi) \in q1 \odot \mathsf{first}(r_1)\} \qquad (3.27)\\
=&(\text{by Definition 3.5.1})\\
&q1 \odot \{(p, \psi \cup \{\langle\rangle \mapsto \mathsf{inc}\}) \mid (p, \psi) \in \mathsf{first}(r_1)\}\\
=&(\text{by (3.26)})\\
&q \odot \mathsf{first}(r[q])
\end{aligned}$$

The case for last is analogous and will therefore not be treated.

For follow, note first that since $p \in \mathsf{sym}(\mu(r_1^{l..u}))$, there is a $p_1 \in \mathsf{sym}(\mu(r_1))$ such that $p = 1p_1$. Again we use (3.24) together with $r_1 \notin \mathfrak{N}_\Sigma$ (since r in constraint normal form), to get

$$\left\{ (p',\alpha) \;\middle|\; \begin{array}{l} p' \in \mathsf{sym}(\mu(r_1^{l..u})), \alpha \in {\Gamma_{r_1^{l..u}}}^*, \\ \exists u,v : up\alpha p'v \in \|\mathsf{ss}(\mu(r_1^{l..u}))\| \end{array} \right\}$$

$$= \left\{ (1p', 1\alpha) \;\middle|\; \begin{array}{l} p' \in \mathsf{sym}(\mu(r_1)), \alpha \in {\Gamma_{r_1}}^*, \\ \exists u,v : up_1\alpha p'v \in \|\mathsf{ss}(\mu(r_1))\| \end{array} \right\}$$

$$\cup \left\{ (p',\alpha_1\alpha_2) \;\middle|\; \begin{array}{l} p' {\in} \mathsf{sym}(\mu(r_1^{l..u})),\ \alpha_1,\alpha_2 \in {\Gamma_{r_1^{l..u}}}^*, \\ \exists u,v : \left(\begin{array}{l} up_1\alpha_1 {\in} 1\|\mathsf{ss}(\mu(r_1))\| \\ \wedge\ \alpha_2 p'v {\in} \|\mathsf{ss}(\mu(r_1^{l..u}))\| \end{array} \right) \end{array} \right\} \tag{3.28}$$

$$=1 \left\{ (p',\alpha) \;\middle|\; \begin{array}{l} p' \in \mathsf{sym}(\mu(r_1)), \alpha \in \Gamma_{r_1}^*, \\ \exists u,v : up_1\alpha p'v \in \|\mathsf{ss}(\mu(r_1))\| \end{array} \right\}$$

$$\cup \left\{ (p',\alpha_1\alpha_2) \;\middle|\; \begin{array}{l} p' {\in} \mathsf{sym}(\mu(r_1^{l..u})),\ \alpha_1,\alpha_2 \in {\Gamma_{r_1^{l..u}}}^*, \\ \exists u,v : \left(\begin{array}{l} up_1\alpha_1 {\in} 1\|\mathsf{ss}(\mu(r_1))\| \\ \wedge\ \alpha_2 p'v {\in} \|\mathsf{ss}(\mu(r_1^{l..u}))\| \end{array} \right) \end{array} \right\}$$

Finally, we can use Definition 3.6.4, (3.28), the induction hypothesis, and the already proved parts about first to show that:

$$\begin{aligned}
& \mathsf{follow}^{\mu}(r,q,p) = \mathsf{follow}^{\mu}(r,q,1p_1) \\
= \;& \text{(by Definition 3.6.4)} \\
& \mathsf{follow}^{\mu}(r,q1,p_1) \\
& \cup \{(q',\psi \uplus \psi') \mid (q \odot 1p_1,\psi) \in \mathsf{last}^{\mu}(r,q1), (q',\psi') \in \mathsf{first}^{\mu}(r,q)\} \\
= \;& \text{(by the induction hypothesis and (3.27))} \\
& q1 \odot \mathsf{follow}(r_1,p_1) \\
& \cup \;\left\{(q',\psi \uplus \psi') \mid (q \odot 1p_1,\psi) \in q1 \odot \mathsf{last}(r_1), (q',\psi') \in q \odot \mathsf{first}(r_1^{l..u})\right\} \\
= \;& \text{(by Definition 3.5.1)} \\
q\odot & \begin{pmatrix} 1\mathsf{follow}(r_1,p_1) \cup \\ \left\{(q',(1\psi) \uplus \psi') \mid (p_1,\psi) \in \mathsf{last}(r_1), (q',\psi') \in \mathsf{first}(r_1^{l..u})\right\} \end{pmatrix} \\
= \;& \text{(by Definition 3.6.1)} \\
q\odot & \left(\begin{array}{l} 1\left\{(p',\mathsf{update}_{r_1}(\alpha)) \;\middle|\; \begin{array}{l} p' \in \mathsf{sym}(\mu(r_1)), \alpha \in \Gamma^*_{r_1}, \\ \exists u,v : up_1\alpha p'v {\in} \|\mathsf{ss}(\mu(r_1))\| \end{array}\right\} \cup \\ \left\{ \left(q', \begin{pmatrix} 1\mathsf{update}_{r_1}(\alpha) \\ \uplus \; \mathsf{update}_{r_1^{l..u}}(\alpha_2) \end{pmatrix}\right) \;\middle|\; \begin{array}{l} p_1 \in \mathsf{sym}(\mu(r_1)), \\ \alpha \in \Gamma^*_{r_1}, \\ \exists u : up_1\alpha {\in} \|\mathsf{ss}(\mu(r_1))\|, \\ q' \in \mathsf{sym}(\mu(r_1^{l..u})), \\ \alpha_2 \in \Gamma^*_{r_1^{l..u}}, \\ \exists u : \alpha_2 q' u \in \|\mathsf{ss}(\mu(r_1^{l..u}))\| \end{array} \right\} \end{array} \right) \\
= \;& \text{(by Definition 3.5.17)} \\
q\odot & \left(\begin{array}{l} 1\left\{(p',\mathsf{update}_{r_1}(\alpha)) \;\middle|\; \begin{array}{l} p' \in \mathsf{sym}(\mu(r_1)), \alpha \in \Gamma^*_{r_1}, \\ \exists u,v : up_1\alpha p'v \in \|\mathsf{ss}(\mu(r_1))\| \end{array}\right\} \cup \\ \left\{ (p',\mathsf{update}_{r_1^{l..u}}(\alpha_1\alpha_2)) \;\middle|\; \begin{array}{l} \alpha_1 \in \Gamma_{r_1^{l..u}}{}^*, \\ \exists u : up\alpha_1 \in 1\|\mathsf{ss}(\mu(r_1))\|, \\ p' \in \mathsf{sym}(\mu(r_1^{l..u})), \\ \alpha_2 \in \Gamma_{r_1^{l..u}}{}^*, \\ \exists u : \alpha_2 p' u \in \|\mathsf{ss}(\mu(r_1^{l..u}))\| \end{array} \right\} \end{array} \right) \\
= \;& \text{(by (3.28))} \\
& q \odot \left(\left\{ (p',\mathsf{update}_{r_1^{l..u}}(\alpha)) \;\middle|\; \begin{array}{l} p' \in \mathsf{sym}(\mu(r_1^{l..u})), \alpha \in \Gamma^*_{r_1^{l..u}}, \\ \exists u,v : up\alpha p'v \in \|\mathsf{ss}(\mu(r_1^{l..u}))\| \end{array} \right\} \right) \\
= \;& \text{(by Definition 3.6.1)} \\
& q \odot \mathsf{follow}(r[q],p)
\end{aligned}$$

For the induction case where $r[q] = \&(r_1, \ldots, r_n)$, $\mathsf{first}^{\mu}(r, q) = q \odot \mathsf{first}(r[q])$ follows by unfolding the definitions, applying the induction hypothesis and using that $\{p' \in \mathsf{pos}(r[q]) \mid \exists w : p'w \in \|\mu(r[q])\|\} = \bigcup_{i\in\{1,\ldots,n\}} \langle i \rangle \odot \{p' \in \mathsf{pos}(r_i) \mid \exists w : p'w \in \|\mu(r_i)\|\}$. Similarly, to show that $\mathsf{last}^{\mu}(r, q) = q \odot \mathsf{last}(r[q])$, we unfold the definitions, use the induction hypothesis and that $\{p' \in \mathsf{pos}(r[q]) \mid \exists w : wp' \in \|\mu(r[q])\|\} = \bigcup_{i\in\{1,\ldots,n\}} \langle i \rangle \odot \{p' \in \mathsf{pos}(r_i) \mid \exists : wp' \in \|\mu(r_i)\|\}$. For $\mathsf{follow}^{\mu}(r, q, ip)$, the case where there is no $(q \odot ip, \psi) \in \mathsf{last}^{\mu}(r, qi)$, is easy. The case where there is a $(q \odot ip, \psi) \in \mathsf{last}^{\mu}(r, qi)$ follows by a similar argument as for the corresponding case when $r[q] = r_1 \cdot r_2$.

□

Theorem 3.6.7 (Polynomial runtime). *For any regular expression $r \in R_{\Sigma}$ in constraint normal form, and all positions $q \in \mathsf{sym}(\mu(r))$:*

1. *Computing $\mathsf{first}^{\mu}(r, \langle\rangle)$ and $\mathsf{last}^{\mu}(r, \langle\rangle)$ takes time $O(|r|^2)$.*

2. *For any $q \in \mathsf{pos}(r)$, computing $\mathsf{follow}^{\mu}(r, \langle\rangle, q)$ takes time $O(|r|^3)$.*

In the proof we will use that there are a linear number of subexpressions of a regular expression. This is a consequence of the fact that every subexpression is uniquely determined by an instance of a letter or of an operator in the regular expression. Therefore we also have that the size of $\mathsf{pos}(r)$ is linear in r.

Proof.

1. Note first that $|\mathsf{first}^{\mu}(r, \langle\rangle)| = O(|r|)$ and $|\mathsf{last}^{\mu}(r, \langle\rangle)| = O(|r|)$ follow from Definition 3.6.1, Lemma 3.6.6, and parts 4 and 5 of Lemma 3.5.16. Furthermore, since all unions are between disjoint sets, we can represent the sets as lists. Thus union can be implemented in constant time. Prefixing a number to a position (as in $1p$) can also be done in constant time. Since there are a linear number of subexpressions, there are at most a linear number of *calls* to first^{μ} or last^{μ}. The work in each call (excluding recursive calls) is at most linear, thus the total run-time is $O(|r|^2)$.

2. Start with computing first and last for all subexpressions of r. We calculate this bottom-up, saving all values of calls. Since there are a linear number of subexpressions, this process can be done in time $O(|r|^2)$. Computing $\mathsf{follow}^{\mu}(r, \langle\rangle, q)$ will then mean a linear number

of *calls* to follow^{μ}, each of which takes $O(|r|^2)$ time in addition to the recursive call to follow^{μ}.

□

3.7 Constructing FACs

Constructions of Finite Automata with Counters from regular expressions with other extensions (not unordered concatenation) have been given by Sperberg-McQueen [64], Gelade et al. [25], Gelade et al. [23], and Hovland [37].

Definition 3.7.1. *[FAC Construction] Given any regular expression r in constraint normal form we construct the* $\mathsf{FAC}(r)$*,*

$$(\Sigma, Q, \mathcal{C}, \mathcal{A}, \Phi, \mathsf{min}(r), \mathsf{max}(r), q^I, \mathcal{F}),$$

where $Q = \mathsf{sym}(\mu(r)) \cup \{q^I\}$ and $\mathcal{C} = \langle\&\sharp\rangle(r)$. For all $q \in \mathsf{sym}(\mu(r))$, put $\mathcal{A}(q) = r[q]$ and for $q \in \mathsf{sym}(\mu(r))$, put $\Phi(q) = \mathsf{follow}(r,q)$. Put $\Phi(q^I) = \mathsf{first}(r)$. The initial configuration is final if and only if r is nullable. Therefore, the set $\mathcal{F}$ is defined as follows. If $r \in \mathfrak{N}_\Sigma$ put $\mathcal{F} = \mathsf{last}(r) \cup \{(q^I, \varnothing)\}$, and otherwise put $\mathcal{F} = \mathsf{last}(r)$.

The result of applying this algorithm to $r = (\&(a^2, b))^{3..4}$ from Example 3.2.3 is the FAC in Example 3.4.9. (The only difference is that the non-initial states are decorated with the value of $\mathcal{A}$.)

Lemma 3.7.2 (Deterministic FACs)**.** *The FAC constructed from a regular expression is deterministic if and only if the regular expression is strongly 1-unambiguous.*

Proof. Let r be a regular expression, and the $\mathsf{FAC}(r)$ constructed as above.

Firstly, we will assume r is not strongly 1-unambiguous and proceed to prove that the constructed FAC is not deterministic. We first treat the case where r is not 1-unambiguous. There are by Definition 3.5.22 $upv, uqw \in \|\mu(r)\|$, where $u, v \in (\mathsf{pos}(r))^*$, $p, q \in \mathsf{pos}(r)$, $r[p] = r[q]$ and $p \neq q$. If $u = \epsilon$, this implies by Lemma 3.5.14 and Definitions 3.6.1 and 3.7.1 that there are $(p, \psi_1), (q, \psi_2) \in \Phi(q^I)$ where $\mathcal{A}(p) = \mathcal{A}(q)$. Furthermore, also applying part 2 of Lemma 3.5.16, we get $\psi_1, \psi_2 \in (\mathcal{C} \to \{\mathsf{inc}\})$, hence, $\gamma_0 \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \psi_1$ and $\gamma_0 \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \psi_2$, and by Definitions 3.4.7 and 3.4.8 the FAC is not deterministic. Otherwise, if $u \neq \epsilon$, assume $u = p_1 \cdots p_n$ for $p_1, \ldots, p_n \in \mathsf{pos}(r)$. By Lemma 3.5.14 there are $u', v', w' \in (\mathsf{pos}(r) \cup \Gamma_r)^*$ and $\alpha, \beta \in \Gamma_r^*$ such that $u' p_n \alpha p v', u' p_n \beta q w' \in \|\mathsf{ss}(\mu(r))\|$. We get from Definition 3.6.1 and 3.7.1 that $(p, \mathsf{update}_r(\alpha)), (q, \mathsf{update}_r(\beta)) \in \Phi(p_n)$ and $\mathcal{A}(p) = \mathcal{A}(q)$. Further, applying part 2 of Lemma 3.5.21, we get that

$$\mathsf{s}_r(u') \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \mathsf{update}_r(\alpha)$$

and

$$\mathsf{s}_r(u') \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \mathsf{update}_r(\beta)$$

hence by Definitions 3.4.7 and 3.4.8 the FAC is not deterministic. Next, we treat the case where r is 1-unambiguous. There are by Definition 3.5.23 $u\alpha a v, u\beta b w \in \|\mathsf{ss}(r)\|$, where $a, b \in \mathsf{sym}(r)$, $\alpha, \beta \in \Gamma_r^*$, $u, v, w \in (\Sigma \cup \Gamma_r)^*$, $a = b$, and $\alpha \neq \beta$. By Definition 3.5.22 there are therefore either $\alpha p v, \beta p w \in \|\mathsf{ss}(\mu(r))\|$ or $uq\alpha p v, uq\beta p w \in \|\mathsf{ss}(\mu(r))\|$, where $p, q \in \mathsf{sym}(\mu(r))$, $\alpha, \beta \in \Gamma_r^*$, $u, v, w \in (\mathsf{pos}(r) \cup \Gamma_r)^*$, and $\alpha \neq \beta$. This implies by Definition 3.6.1 and 3.7.1 that there is $q \in Q$ and $(p, \mathsf{update}_r(\alpha)), (p, \mathsf{update}_r(\beta)) \in \Phi(q)$. Furthermore, from part 2 of Lemma 3.5.21, we get that $\mathsf{update}_r(\alpha)$ and $\mathsf{update}_r(\beta)$ are overlapping, hence the FAC is not deterministic.

Secondly, we will assume r is strongly 1-unambiguous, and proceed to prove that the automaton is deterministic. That is, for any configuration (q, γ) of the $\mathsf{FAC}(r)$ there are no two different pairs (p_1, ψ_1), $(p_2, \psi_2) \in \Phi(q)$ such that $\mathcal{A}(p_1) = \mathcal{A}(p_2)$, $\gamma \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \psi_1$, and $\gamma \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \psi_2$. We prove the contrapositive, that is, assuming $(p_1, \psi_1), (p_2, \psi_2) \in \Phi(q)$, $\mathcal{A}(p_1) = \mathcal{A}(p_2)$, $\gamma \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \psi_1$, and $\gamma \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \psi_2$ we prove that $(p_1, \psi_1) = (p_2, \psi_2)$.

For the initial configuration (q^I, γ_0), we have $\Phi(q^I) = \mathsf{first}(r)$. Thus from Definition 3.6.1 there are $\alpha_1, \alpha_2 \in \Gamma_r^*$ and $u, v \in (\mathsf{pos}(r) \cup \Gamma_r)^*$ such that $\psi_1 = \mathsf{update}_r(\alpha_1)$, $\psi_2 = \mathsf{update}_r(\alpha_2)$, and $\alpha_1 p_1 u, \alpha_2 p_2 v \in \|\mathsf{ss}(\mu(r))\|$. Since r is 1-unambiguous, and we can take $u = \epsilon$ in Definition 3.5.22, we get $p_1 = p_2$. Furthermore, from part 4 of Lemma 3.5.16 and Definition 3.6.1 we also get $\alpha_1 = \alpha_2$, hence $(p_1, \psi_1) = (p_2, \psi_2)$.

For a non-initial configuration (q, γ) assume $(p_1, \psi_1), (p_2, \psi_2) \in \Phi(q) = \mathsf{follow}(r, q)$, where $\mathcal{A}(p_1) = \mathcal{A}(p_2)$, $\gamma \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \psi_1$, and $\gamma \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \psi_2$. From Definition 3.6.1, we get that there are $u, v, w \in (\Gamma_r \cup \mathsf{pos}(r))^*$

and $\alpha, \beta \in \Gamma_r^*$ such that $uq\alpha p_1 v$ and $uq\beta p_2 w$ are in $\|\mathsf{ss}(\mu(r))\|$, and $\psi_1 = \mathsf{update}_r(\alpha)$ and $\psi_2 = \mathsf{update}_r(\beta)$. Since r is strongly 1-unambiguous and $r[p_1] = \mathcal{A}(p_1) = \mathcal{A}(p_2) = r[p_2]$ we get from Definition 3.5.22 that $p_1 = p_2$ and from Definition 3.5.23 that $\alpha = \beta$ which implies that $\psi_1 = \psi_2$. So we get $(p_1, \psi_1) = (p_2, \psi_2)$ which means that the automaton is deterministic. □

Theorem 3.7.5 states that the construction of FACs is correct. To prove the theorem we need some auxiliary lemmas.

Lemma 3.7.3. *Let r be any regular expression in constraint normal form, $q, p \in \mathsf{pos}(r)$, $\alpha \in \Gamma_r^*$ and $u, v, v', w \in (\mathsf{pos}(r) \cup \Gamma_r)^*$.*

1. If $up\alpha, wpv \in \|\mathsf{ss}(\mu(r))\|$ and $\mathsf{s}_r(w) \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \mathsf{update}_r(\alpha)$, then $wp\alpha \in \|\mathsf{ss}(\mu(r))\|$.

2. If $up\alpha qv, wpv' \in \|\mathsf{ss}(\mu(r))\|$ and $\mathsf{s}_r(w) \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \mathsf{update}_r(\alpha)$, then there is a v'' such that $wp\alpha qv'' \in \|\mathsf{ss}(\mu(r))\|$.

Proof.

1. By induction on r. The base case $r = \epsilon$ holds vacuously. The base case where $r \in \Sigma$ holds immediately, since then $\|\mathsf{ss}(\mu(r))\| = \{\langle\rangle\}$, and thus u, α, w, and v are all ϵ.

The induction case where $r = r_1 + r_2$ holds by applying the induction hypothesis for r_1 and r_2.

For the induction case where $r = r_1 \cdot r_2$, we have $up\alpha, wpv \in \|\mathsf{ss}(\mu(r_1 \cdot r_2))\|$, and $\mathsf{s}_{r_1 r_2}(w) \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \mathsf{update}_r(\alpha)$, and want to prove that $wp\alpha \in \|\mathsf{ss}(\mu(r_1 \cdot r_2))\|$. We have either $\langle 1\rangle \leq p$, or $\langle 2\rangle \leq p$. In the former case, note that $r_2 \in \mathfrak{N}_\Sigma$, and by part 1 of Lemma 3.5.16 we therefore get that there are v_1, v_2 such that $v = v_1 v_2$ and $up\alpha, wpv_1 \in 1\|\mathsf{ss}(\mu(r_1))\|$. Therefore we can apply the induction hypothesis for r_1 to get that $wp\alpha \in 1\|\mathsf{ss}(\mu(r_1))\|$, and thus $wp\alpha \in \|\mathsf{ss}(\mu(r_1 \cdot r_2))\|$. The case where $\langle 2\rangle \leq p$ follows from using the induction hypothesis for r_2 and that $1\Gamma_{r_1} \cap 2\Gamma_{r_2} = \varnothing$.

For the induction case where $r = r_1^{l..n}$, we have $up\alpha, wpv \in \|\mathsf{ss}(\mu(r_1^{l..n}))\|$, and $\mathsf{s}_{r_1^{l..n}}(w) \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \mathsf{update}_r(\alpha)$, and want to prove that also $wp\alpha \in \|\mathsf{ss}(\mu(r_1^{l..n}))\|$. By the definition of subscripted language, we get there is j such that $l \leq j \leq n$ and there are $u_1, \dots, u_j$ such that $up\alpha = \uparrow_{\langle 1 \rangle} u_1 \cdots \uparrow_{\langle 1 \rangle} u_j p \alpha_1 \downarrow_{\langle 1 \rangle}$ and $u_1, \dots, u_{j-1}, u_j p \alpha_1 \in 1 \| \mathsf{ss}(\mu(r_1)) \|$. There is also m such that $l \leq m \leq n$, and there are $w_1, \dots, w_m, v_1, v_2$ such that $wpv = \uparrow_{\langle 1 \rangle} w_1 \cdots \uparrow_{\langle 1 \rangle} w_m p v_1 v_2$ and $w_1, \dots, w_{m-1}, w_m p v_1 \in 1 \| \mathsf{ss}(\mu(r_1)) \|$. From Lemma 3.5.16 $\alpha_1 = \downarrow_{x_1} \cdots \downarrow_{x_j}$ for $x_1, \dots, x_j \in 1\mathsf{pos}(r_1)$. Note now that from part 1 of Lemma 3.5.21, for $p' \neq \langle 1 \rangle$, $\mathsf{s}_{r_1^{l..n}}(w)(p') = \mathsf{s}_{r_1^{l..n}}(w_m)(p')$. Therefore $\mathsf{s}_{r_1^{l..n}}(w_m) \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \mathsf{update}_r(\alpha_1)$, and we can apply the induction hypothesis for r_1 to get that $w_m p \alpha_1 \in 1 \| \mathsf{ss}(\mu(r_1)) \|$. Since $l \leq m \leq u$, we can now use the definition of the language of a subscripted expression to get that $wp\alpha \in \| \mathsf{ss}(\mu(r_1^{l..n})) \|$.

For the induction case where $r = \&(r_1, \dots, r_n)$, we have $up\alpha, wpv \in \| \mathsf{ss}(\mu(\&(r_1, \dots, r_n))) \|$, and $\mathsf{s}_{\&(r_1, \dots, r_n)}(w) \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \mathsf{update}_r(\alpha)$, and want to prove that also $wp\alpha \in \| \mathsf{ss}(\mu(\&(r_1, \dots, r_n))) \|$. We can by symmetry assume that $\langle n \rangle \leq p$, such that there is a $p' \in \mathsf{pos}(r_n)$ such that $p = np'$. By the definitions and Lemma 3.5.16, there are $u_1, w_1, v_2 \in (\mathsf{pos}(r) \cup \Gamma_r)^*$, $u_2, v_1, w_2 \in (\mathsf{pos}(r_n) \cup \Gamma_{r_n})^*$ and $\alpha_1 \in \Gamma_{r_n}^*$, such that

$$u \cdot p \cdot \alpha = u_1 \cdot \uparrow_{\langle n \rangle} \cdot n(u_2 \cdot p' \cdot \alpha_1) \cdot \downarrow_{\langle 1 \rangle} \cdots \downarrow_{\langle n \rangle} \tag{3.29}$$

$$u_2 \cdot p' \cdot \alpha_1 \in \| \mathsf{ss}(\mu(r_n)) \| \tag{3.30}$$

$$w \cdot p \cdot v = w_1 \cdot \uparrow_{\langle n \rangle} \cdot n(w_2 \cdot p' \cdot v_1) \cdot v_2 \cdot \downarrow_{\langle 1 \rangle} \cdots \downarrow_{\langle n \rangle} \tag{3.31}$$

$$w_2 \cdot p' \cdot v_1 \in \| \mathsf{ss}(\mu(r_n)) \| \tag{3.32}$$

Since the positions on the arrows in $n\alpha_1$ are non-overlapping with the rest of α, and since $\mathsf{s}_{\&(r_1, \dots, r_n)}(w) \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \mathsf{update}_r(\alpha)$ we get that $\mathsf{s}_{r_n}(w_2) \models_{\mathsf{min}(r_n)}^{\mathsf{max}(r_n)} \mathsf{update}_{r_n}(\alpha_1)$. We can apply the induction hypothesis for r_n to (3.32) and (3.30) to get that $w_2 \cdot p' \cdot \alpha_1 \in \| \mathsf{ss}(\mu(r_n)) \|$. It is now immediate from the definitions that $w_1 \cdot \uparrow_{\langle n \rangle} \cdot n(w_2 \cdot p' \cdot \alpha_1) \cdot v_2 \cdot \downarrow_{\langle 1 \rangle} \cdots \downarrow_{\langle n \rangle} \in \| \mathsf{ss}(\mu(\&(r_1, \dots, r_{n-1}))) \|$. We need to show the previous expression with v_2 removed. But since $\mathsf{s}_{\&(r_1, \dots, r_n)}(w) \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \mathsf{update}_r(\alpha)$ we know that there is a $\uparrow_{\langle i \rangle}$ in w for all $r_i \notin \mathfrak{N}_\Sigma$. Thus it is OK to remove v_2, and we get that $wp\alpha \in \| \mathsf{ss}(\mu(r)) \|$.

2. By induction on the regular expression r. The base cases where $r = \epsilon$ or $r \in \Sigma$ hold vacuously.

The induction case where $r = r_1 + r_2$ holds by applying the induction hypothesis for r_1 and r_2.

For the induction case where $r = r_1 \cdot r_2$, we have $up\alpha qv, wpv' \in \|\mathsf{ss}(\mu(r_1 \cdot r_2))\|$ and $\mathsf{s}_r(w) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(\alpha)$, and wish to prove that there is a v'' such that $wp\alpha qv'' \in \|\mathsf{ss}(\mu(r_1 \cdot r_2))\|$. Note first that if $\langle 1\rangle \leq p \wedge \langle 1\rangle \leq q$ or $\langle 2\rangle \leq p \wedge \langle 2\rangle \leq q$, we get the result by applying Lemma 3.5.21 and applying the induction hypothesis for r_1 or r_2, respectively. Otherwise, we have $p' \in \mathsf{pos}(r_1)$ and $q' \in \mathsf{pos}(r_2)$ such that $p = 1p'$ and $q = 2q'$. We then get from Lemma 3.5.16 that there are $\alpha_1 \in {\Gamma_{r_1}}^*$ and $\alpha_2 \in {\Gamma_{r_2}}^*$ such that $up\alpha_1 \in 1\|\mathsf{ss}(\mu(r_1))\|$, $up\alpha qv = up\alpha_1\alpha_2 qv$, and $\alpha_2 qv \in 2\|\mathsf{ss}(\mu(r_2))\|$. We have $\mathsf{s}_r(w) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(\alpha_1)$, and can apply part 1 of the lemma to r_1 and get that $wp\alpha_1 \in 1\|\mathsf{ss}(\mu(r_1))\|$. Therefore we also have $wp\alpha_1\alpha_2 qv \in \|\mathsf{ss}(\mu(r_1 \cdot r_2))\|$.

For the induction case where $r = r_1^{l..n}$, we have $up\alpha qv, wpv' \in \|\mathsf{ss}(\mu(r_1^{l..n}))\|$ and $\mathsf{s}_{r_1^{l..n}}(w) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(\alpha)$, and wish to prove that there is a v'' such that $wp\alpha qv'' \in \|\mathsf{ss}(\mu(r_1^{l..n}))\|$. If there is no $\uparrow_{\langle 1\rangle}$ in α, we get the result from the induction hypothesis for r_1 and Lemma 3.5.21. Otherwise, note that since r is in constraint normal form, $r_1 \notin \mathfrak{N}_\Sigma$, so there is exactly one $\uparrow_{\langle 1\rangle}$ in α. Using the definition of the language of a subscripted expression, and Lemma 3.5.21 we get that there are $p', q' \in \mathsf{pos}(r_1)$, $u_1, v_2, w_1, v_2' \in (\mathsf{pos}(r) \cup \Gamma_r)^*$, $u_2, v_1, w_2, v_1' \in (\mathsf{pos}(r_1) \cup \Gamma_{r_1})^*$, and $\alpha_1, \alpha_2 \in \Gamma_{r_1}^*$ such that

$$p = 1p' \wedge q = 1q' \tag{3.33}$$

$$up\alpha qv = u_1 \cdot \uparrow_{\langle 1\rangle} \cdot 1(u_2 \cdot p' \cdot \alpha_1) \cdot \uparrow_{\langle 1\rangle} \cdot 1(\alpha_2 \cdot q' \cdot v_1) \cdot v_2 \tag{3.34}$$

$$u_2 \cdot p' \cdot \alpha_1 \in \|\mathsf{ss}(\mu(r_1))\| \tag{3.35}$$

$$\alpha_2 \cdot q' \cdot v_1 \in \|\mathsf{ss}(\mu(r_1))\| \tag{3.36}$$

$$w \cdot p \cdot v' = w_1 \cdot \uparrow_{\langle 1\rangle} \cdot 1(w_2 \cdot p' \cdot v_1') \cdot v_2' \tag{3.37}$$

$$w_2 \cdot p' \cdot v_1' \in \|\mathsf{ss}(\mu(r_1))\| \tag{3.38}$$

$$\mathsf{s}_{r_1}(w_2) \models^{\mathsf{max}(r_1)}_{\mathsf{min}(r_1)} \mathsf{update}_{r_1}(\alpha_1) \tag{3.39}$$

By applying part 1 of the lemma for r_1 to (3.35), (3.38), and (3.39) we get $w_2 \cdot p' \cdot \alpha_1 \in \|\mathsf{ss}(\mu(r_1))\|$. But then we can use $\mathsf{s}_{r_1^{l..n}}(w) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(\alpha)$ to get that the number of $\uparrow_{\langle 1\rangle}$ in w_1 plus two is at most n. Furthermore, if v_2' is $\downarrow_{\langle 1\rangle}$, put $v_2'' = v_2'$, otherwise, put v_2'' such that

$v_2' \in \uparrow_{\langle 1\rangle} \cdot 1\|\mathsf{ss}(\mu(r_1))\| \cdot v_2''$. Thus $w_1 \cdot \uparrow_{\langle 1\rangle} \cdot 1(w_2 \cdot p' \cdot \alpha_1) \cdot \uparrow_{\langle 1\rangle} \cdot 1(\alpha_2 \cdot q' \cdot v_1) \cdot v_2'' = w \cdot p \cdot \alpha \cdot q \cdot 1v_1 \cdot v_2'' \in \|\mathsf{ss}(\mu(r))\|$.

For the induction case where $r = \&(r_1, \ldots, r_n)$, we have $up\alpha qv, wpv' \in \|\mathsf{ss}(\mu(\&(r_1, \ldots, r_n)))\|$ and $\mathsf{s}_{\&(r_1,\ldots,r_n)}(w) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(\alpha)$, and wish to prove that there is a v'' such that $wp\alpha qv'' \in \|\mathsf{ss}(\mu(\&(r_1, \ldots, r_n)))\|$. If there is an i such that $\langle i\rangle \leq p$ and $\langle i\rangle \leq q$, then the result follows by applying the induction hypothesis for r_i. Otherwise, there are i, j, p', q' such that $1 \leq i, j \leq n$, $i \neq j$, $p = ip'$, and $q = jq'$. Furthermore, there are $u_1, u_2, \alpha_1, \alpha_2, v_1, v_2$ such that

$$up\alpha qv = u_1 \cdot \uparrow_{\langle i\rangle} \cdot i(u_2 \cdot p' \cdot \alpha_1) \cdot \uparrow_{\langle j\rangle} \cdot j(\alpha_2 \cdot q' \cdot v_1) \cdot v_2 \tag{3.40}$$

$$u_2 \cdot p' \cdot \alpha_1 \in \|\mathsf{ss}(\mu(r_i))\| \tag{3.41}$$

$$\alpha_2 \cdot q' \cdot v_1 \in \|\mathsf{ss}(\mu(r_j))\| \tag{3.42}$$

$$w \cdot p \cdot v' = w_1 \cdot \uparrow_{\langle i\rangle} \cdot i(w_2 \cdot p' \cdot v_1') \cdot v_2' \tag{3.43}$$

$$w_2 \cdot p' \cdot v_1' \in \|\mathsf{ss}(\mu(r_i))\| \tag{3.44}$$

$$\mathsf{s}_{r_i}(w_2) \models^{\mathsf{max}(r_i)}_{\mathsf{min}(r_i)} \mathsf{update}_{r_i}(\alpha_1) \tag{3.45}$$

We can apply (3.41), (3.44), and (3.45) to part 1 of the lemma for r_i to get $w_2 \cdot p' \cdot \alpha_1 \in \|\mathsf{ss}(\mu(r_i))\|$. Since $w_1 \cdot \uparrow_{\langle i\rangle} \cdot iw_2 \cdot p \cdot v' \in \|\mathsf{ss}(\mu(\&(r_1, \ldots, r_n)))\|$ there cannot be any $\uparrow_{\langle i\rangle}$ in w_1. Since $\mathsf{s}_{\&(r_1,\ldots,r_n)}(w) \models^{\mathsf{max}(r)}_{\mathsf{min}(r)} \mathsf{update}_r(\alpha)$ there cannot be any $\uparrow_{\langle j\rangle}$ in w_1. We now distinguish two cases. If v_2' does not contain $\uparrow_{\langle j\rangle}$ (this implies that $r_j \in \mathfrak{N}_\Sigma$), then we already have $w_1 \cdot \uparrow_{\langle i\rangle} \cdot i(w_2 \cdot p' \cdot \alpha_1) \cdot \uparrow_{\langle j\rangle} \cdot j(\alpha_2 \cdot q' \cdot v_1) \cdot v_2' = w \cdot p \cdot \alpha \cdot q \cdot jv_1 \cdot v_2' \in \|\mathsf{ss}(\mu(\&(r_1, \ldots, r_n)))\|$. That means the lemma holds for $v'' = jv_1 \cdot v_2'$. Otherwise, if there is a $\uparrow_{\langle j\rangle}$ in v_2', then there are $v_a, v_b, v_c \in (\mathsf{pos}(r) \cup \Gamma_r)$ $k \in \{1, \ldots, n\} - \{j, i\}$ and $v_j \in j\|\mathsf{ss}(\mu(r_j))\|$ such that $v_2' = v_a \uparrow_{\langle j\rangle} v_j v_b$ and either $v_b = \uparrow_{\langle k\rangle} v_c$ or $v_b = \downarrow_{\langle 1\rangle} \cdots \downarrow_{\langle n\rangle}$. Thus $w_1 \cdot \uparrow_{\langle i\rangle} \cdot i(w_2 \cdot p' \cdot \alpha_1) \cdot \uparrow_{\langle j\rangle} \cdot j(\alpha_2 \cdot q' \cdot v_1) \cdot v_a' \cdot v_b = w \cdot p \cdot \alpha \cdot q \cdot jv_1 \cdot v_a \cdot v_b \in \|\mathsf{ss}(\mu(\&(r_1, \ldots, r_n)))\|$, and the lemma holds for $v'' = jv_1 \cdot v_a \cdot v_b$.

□

Lemma 3.7.4. *For any regular expression r in constraint normal form and any word w, if* $\mathsf{FAC}(r)$ *recognizes w, then there is a $w' \in \|\mathsf{ss}(\mu(r))\|$ such that $w = r[\flat(w')]$.*

Proof. If $w = \epsilon$, then by Definition 3.4.5 and the definition of word recognition by FAC the initial configuration is final, and by Definition 3.7.1 $r \in \mathfrak{N}_\Sigma$. Hence, we can use $w' = \epsilon$ as $\epsilon \in \|\mathsf{ss}(\mu(r))\|$ and $\epsilon = r[\flat(\epsilon)]$.

For the case where $w \neq \epsilon$ is recognized by FAC(r), there is $n \geq 1$ and $l_1, \ldots, l_n \in \Sigma$ such that $w = l_1 \cdots l_n$. We prove in the next paragraph that if the sequence of configurations used to match $w = l_1 \cdots l_n$ is $(q^I, \gamma_0), (q_1, \gamma_1), \ldots, (q_n, \gamma_n)$, then there are $\alpha_1, \ldots, \alpha_n \in \Gamma_r^*$ and v such that $\alpha_1 q_1 \cdots \alpha_n q_n v \in \|\mathsf{ss}(\mu(r))\|$, $w = r[q_1 \cdots q_n]$, and for $i \in \{1, \ldots, n\}$, $\gamma_i = \mathsf{s}_r(\alpha_1 q_1 \cdots \alpha_{i-1} q_{i-1} \alpha_i)$. Since the configuration (q_n, γ_n) is final, there are by Definitions 3.7.1, 3.6.1, and 3.4.6 $u \in (\mathsf{pos}(r) \cup \Gamma_r)^*$ and $\alpha_{n+1} \in \Gamma_r^*$ such that $u q_n \alpha_{n+1} \in \|\mathsf{ss}(\mu(r))\|$ and $\gamma_n \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \mathsf{update}_r(\alpha_{n+1})$. We can now apply part 1 of Lemma 3.7.3 to get $w' = \alpha_1 q_1 \cdots \alpha_n q_n \alpha_{n+1} \in \|\mathsf{ss}(\mu(r))\|$.

We prove by induction on m that for any prefix (q^I, γ_0), (q_1, γ_1), $\ldots$, (q_m, γ_m) of length $m + 1$ of any sequence of configurations used to match the word $w = l_1 \cdots l_n$, there are $\alpha_1, \ldots, \alpha_m \in \Gamma_r^*$, $w' \in (\mathsf{pos}(r) \cup \Gamma_r)^*$ such that $\alpha_1 q_1 \cdots \alpha_m \cdot q_m w' \in \|\mathsf{ss}(\mu(r))\|$, and for all $1 \leq i \leq m$, $l_i = r[q_i]$ and $\gamma_i = \mathsf{s}_r(\alpha_1 q_1 \cdots \alpha_{i-1} q_{i-1} \alpha_i)$.

The base case is the prefix of length 1 containing only the initial configuration, in which case $w' = \epsilon$, $m = 0$, and $\mathsf{s}_r(\epsilon) = \gamma_0$.

For the inductive case, we assume a prefix of length $m + 1$, and that the next configuration in the sequence is (q_{m+1}, γ_{m+1}). This implies that $\delta((q_m, \gamma_m), l_{m+1}) = (q_{m+1}, \gamma_{m+1})$. By construction of the FAC and Definition 3.6.1 this implies that $r[q_{m+1}] = l_{m+1}$ and that there are $u, v \in (\mathsf{pos}(r) \cup \Gamma_r)^*$ and an $\alpha_{m+1} \in \Gamma_r^*$ such that $u q_m \alpha_{m+1} q_{m+1} v \in \|\mathsf{ss}(\mu(r))\|$, $\gamma_{m+1} = f_{\mathsf{update}_r(\alpha_{m+1})}(\gamma_m)$, and $\gamma_m \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \mathsf{update}_r(\alpha_{m+1})$. The induction hypothesis gives us words $\alpha_1, \ldots, \alpha_m \in \Gamma_r^*$, $w' \in (\mathsf{pos}(r) \cup \Gamma_r)^*$ such that $\alpha_1 q_1 \cdots \alpha_m q_m w' \in \|\mathsf{ss}(\mu(r))\|$ and that $l_i = r[q_i]$ and $\gamma_i = \mathsf{s}_r(\alpha_1 q_1 \cdots \alpha_{i-1} q_{i-1} \alpha_i)$ for all $1 \leq i \leq m$. Therefore $\gamma_{m+1} = \mathsf{s}_r(\alpha_1 \cdots \alpha_{m+1})$. Furthermore, since $\gamma_m = \mathsf{s}_r(\alpha_1 \cdots \alpha_m)$, and

$$\alpha_1 q_1 \cdots \alpha_m \cdot q_m w', u q_m \alpha_{m+1} q_{m+1} v \in \|\mathsf{ss}(\mu(r))\|,$$

we can apply Lemma 3.7.3 to get $\alpha_1 q_1 \cdots \alpha_m q_m \alpha_{m+1} q_{m+1} w'' \in \|\mathsf{ss}(\mu(r))\|$ for some w'', and the lemma holds. □

Theorem 3.7.5. *For any regular expression r in constraint normal form,* $\mathsf{FAC}(r)$ *constructed as in Definition 3.7.1 recognizes exactly $\|r\|$.*

Proof. First, assume a regular expression r in constraint normal form, and a word w which is recognized by $\mathsf{FAC}(r)$. We must show that $w \in \|r\|$. By Lemma 3.7.4 we get a word $w' \in \|\mathsf{ss}(\mu(r))\|$ such that $w = r[\flat(w')]$. By Lemma 3.5.15, $r[\flat(w')] \in \|r\|$, thus $w \in \|r\|$.

Conversely, assume $w = l_1 \cdots l_n \in \|r\|$, where n is the length of w. We must show that w is recognized by $\mathsf{FAC}(r)$. Since $w \in \|r\|$, we get from Lemma 3.5.15 that there is a $w' \in \|\mathsf{ss}(\mu(r))\|$ such that $r[\flat(w')] = w$. Furthermore, there are $\alpha_1, \ldots, \alpha_{n+1} \in \Gamma_r^*$ and $p_1, \ldots, p_n \in \mathsf{sym}(\mu(r))$ such that $\flat(w') = p_1 \cdots p_n$ and $w' = \alpha_1 p_1 \cdots \alpha_n p_n \alpha_{n+1}$. By Definitions 3.7.1 and 3.6.1 this implies that $(p_1, \mathsf{update}_r(\alpha_1)) \in \Phi(q^I)$, $(p_n, \mathsf{update}_r(\alpha_{n+1})) \in \mathcal{F}$, and furthermore, for $1 \leq i < n$,

$$(p_{i+1}, \mathsf{update}_r(\alpha_{i+1})) \in \Phi(p_i)$$

Recall that $\gamma_0 = \mathsf{s}_r(\epsilon)$, put $\gamma_1 = \mathsf{update}_r(\alpha_1)$, and for each $i > 1$, let $\gamma_i = \mathsf{s}_r(\alpha_1 q_1 \cdots \alpha_{i-1} q_{i-1} \alpha_i)$. It follows from Definition 3.5.18 that for each $i \geq 1$, $\gamma_i = \mathsf{update}_r(\alpha_i)(\gamma_{i-1})$. Consider the sequence of configurations $(q^I, \gamma_0), (q_1, \gamma_1), \ldots, (q_n, \gamma_n)$. To prove that this is a run of $\mathsf{FAC}(r)$ which recognizes w it suffices to check that for any $i, 1 \leq i < n$,

$$\gamma_i \models_{\mathsf{min}(r)}^{\mathsf{max}(r)} \mathsf{update}_r(\alpha_{i+1})$$

But this is immediate from Lemma 3.5.21. □

Corollary 3.7.6. *For any regular expression r, we can in polynomial time construct an FAC recognizing exactly $\|r\|$. For any word w, and any strongly 1-unambiguous regular expression r, we can in polynomial time decide whether $w \in \|r\|$.*

Proof. From Theorem 3.7.5, Corollary 3.5.9, and Lemmas 3.6.7, 3.7.2, and 3.4.10. □

3.8 Related Work and Conclusion

3.8.1 Related Work

Sperberg-McQueen [64] has studied regular expressions with numerical constraints and a translation to finite automata with counters, though no proofs are given. Gelade et al. [24, 25] and Gelade et al. [23] also wrote about this, including full proofs. The latter was published simultaneously with the paper [37]. The present chapter is based on [37], but also incorporates ideas from [23], most notably the bracketing, which was inspiration for the subscripted expressions. Section 6 from [23], including the proofs for Section 6 in the Appendix of [23] has inspired some of the content in Sections 3.5.2, 3.5.3, and 3.7.

Kilpeläinen & Tuhkanen [42, 43, 44], Gelade [22], Gelade et al. [24, 25], and Gelade et al. [23] also investigated properties of the regular expressions with numerical constraints, and give algorithms for membership. Stockmeyer & Meyer [54] study the regular expressions with squaring, a subclass of the regular expressions with numerical constraints. Colazzo, Ghelli & Sartiani, describe in [28] an algorithm for linear-time membership in a subclass of regular expressions called collision-free. The collision-free regular expressions have at most one occurrence of each symbol from Σ, and the counters (and the Kleene star) can only be applied directly to letters or disjunctions of letters. The latter class is strictly included in the class of strongly 1-unambiguous regular expressions. The results by Brüggemann-Klein & Wood in [9, 10, 12] concerning 1-unambiguous regular expressions, are in some ways what the present chapter attempts to extend to the regular expressions with numerical constraints and unordered concatenation.

Extensions of finite automata similar to Finite Automata with Counters have been studied by many authors. The earliest is the treatment of multicounter automata by Greibach [31]. The definitions used in this chapter are adapted from Hovland [37]. Brüggemann-Klein [11, 15] gives an algorithm for deciding 1-unambiguity of regular expressions with unordered concatenation. Unordered concatenation is also mentioned in [12, 10]. Strong 1-unambiguity has also been mentioned by Brüggemann-Klein & Wood [12, 10] and Sperberg-McQueen [64], and Gelade et al. [23]. The first in-depth study of strong 1-unambiguity was by Koch & Scherzinger [47].

3.8.2 Conclusion

We have studied the membership problem for regular expressions extended with numerical constraints and with an operator similar to "interleaving" in SGML. The membership problem was shown to be NP-complete already without the numerical constraints. We defined *Finite Automata with Counters* (FAC), and a polynomial-time translation from the regular expressions with numerical constraints and unordered concatenation to these automata. Further we defined *strongly 1-unambiguous regular expressions*, a subset of the regular expressions with numerical constraints and unordered concatenation in constraint normal form, and for which the FAC resulting from the translation is deterministic. The deterministic FAC can recognize the language of the given regular expression in time linear in the size of word to be tested. Testing whether an FAC is deterministic can be done in polynomial time. This implies that the restrictions put on unordered concatenation in XML Schema might be stronger than necessary.

4 A Type System for Usage of Software Components

The aim of this chapter is to support component-based software engineering by modeling exclusive and inclusive usage of software components. Truong and Bezem describe in several papers abstract languages for component software with the aim to find bounds to the number of instances of components. Their languages include primitives for instantiating and deleting instances of components and operators for sequential, alternative and parallel composition, and a scope mechanism. The language is here supplemented with the primitives `use`, `lock` and `free`. The main contribution is a type system which guarantees the safety of usage, in the following way: When a well-typed program executes a subexpression `use`$[x]$ or `lock`$[x]$, it is guaranteed that an instance of x is available. Type inference is shown to be polynomial.

4.1 Introduction

The idea of "Mass produced software components" was first formulated by McIlroy [52] in an attempt to encourage the production of software routines in much the same way industry manufactures ordinary, tangible products. The last two decades "component" has got the more general meaning of a highly reusable piece of software. According to Szyperski [66] (p. 3), "(...) software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system". We will model software that is constructed of such components, and assume that during the execution of such a program, instances of the components can be created, used and deleted.

Efficient component software engineering is not compatible with programmers having to acquire detailed knowledge of the internal structure of components that are being used. Components can also be constructed to use other components, such that instantiating one component, could lead to several instances of other components. This lack of knowledge in combination with nested dependencies weakens the control over resource usage in the composed software.

The goal of this chapter is to guarantee the safe usage of components, such that one can specify that some instances must be available, possibly exclusively to the current thread of execution. In [5, 67, 68], Truong and Bezem describe abstract languages for component software with the aim of finding bounds of the number of instances of components existing during and remaining after execution of a component program. Their languages include primitives for instantiating and deleting instances of components and have operators for sequential, alternative and parallel composition and a scope mechanism. The first three operators are well-known, and have been treated by for example Milner [55] (where alternative composition is called *summation*). The scope mechanism works like this: Any component instantiated in a scope has a lifetime limited to the scope. Furthermore, inside a scope, only instances in the local store of the same scope can be deleted. The types count the maximum number of active component instances during execution and remaining after execution of a component program.

The languages described by Truong and Bezem lack a direct way of specifying that one or more instances of a component must exist at some point in the execution. In this chapter we have added the primitives `use`, `lock` and `free` in order to study the usage of components. The first (`use`) is used for "inclusive usage", that is, when a set of instances must be available, but these instances may be shared between threads. The other form (`lock` and `free`) is used when the instances must be exclusively available for this execution thread. The difference between exclusive and inclusive usage can be seen by comparing the expressions $\texttt{new}\,x(\texttt{use}\,[x] \parallel \texttt{use}\,[x])$ and $\texttt{new}\,x(\texttt{lock}\,[x]\texttt{free}\,[x] \parallel \texttt{use}\,[x])$. The first expression is safe to execute, while executing the latter expression can lead to an error if x is locked, but not freed, by the left thread before it is used by the right thread. Instances of the same component cannot be distinguished, such that locking and freeing is not applied to specific instances, but to the number of instances of each component.

The type system must guarantee that the instances that are to be used are available. The system will not test whether the deletion of instances in local stores is safe, as this can be tested by the type systems in [3, 5, 67, 68] while ignoring `lock`, `free`, and `use`. Only non-recursive programs are treated, but an extension with loops and simple recursion, described in [3], can also be applied to this system. An alternative to using a type system is of course to run all possible executions of the program, and count the number of instances. However, the number of execution traces for any given program, is, in general, super-polynomial in the size of the program. Hence, this *brute force* approach is not feasible in this context.

Section 4.2 introduces an example using C++, to which our type system is applied in Section 4.6. The language of component programs is defined in Section 4.3, and the operational semantics is defined in Section 4.4. The types and the type system are explained in Section 4.5. Important properties of the type system are formulated in Section 4.7, while the main results concerning correctness are collected in Section 4.8. The chapter ends with a section on related work and a conclusion.

4.2 Example: Objects on the Free Store in C++

We will introduce an example with dynamically allocated memory in C++ [65]. In Section 4.6 we will apply the type system to the example. The example is inspired by a similar one in [3].

In the program fragment in Fig. 4.1, so-called POSIX threads [39] are used for parallelism. After creating an instance of the class `C`, the function `pthread_create` launches a new thread calling the function which is third in the parameter list with the argument which is fourth. This function call, either `P1(C_instance)` or `P2(C_instance)`, is executed in parallel to `P3(C_instance)`, and the two threads are joined in `pthread_join` before the instance of `C` is deleted.

The dynamic data type `C` and the functions `P1, P2, P3` are left abstract. We will assume the latter three functions use the instance of `C` in some way, and that `P2` needs exclusive access to the instance.

The question in this example is whether we can guarantee that `P2` gets exclusive access to the instance of `C`. In this small example it is possible to see that this is not the case. After the grammar is explained in the next section we will model the program in the language as shown in Fig. 4.2, and use the type system to answer the question and correct

```
void EX(int choice) {
  pthread_t pth;
  C* C_instance = new C();
  pthread_create(&pth, NULL, choice ? P1 : P2 , C_instance);
  P3(C_instance);
  pthread_join(pth, NULL);
  delete C_instance;
}
```

Figure 4.1: C++ code using threads and objects on the free store.

the program.

4.3 Syntax

The language for components is parametrized by an arbitrary set $\mathbb{C} = \{a, b, c, \ldots\}$ of *component names*. We let variables x, y, z range over $\mathbb{C}$. Bags and multisets are used frequently in this chapter, and will therefore be explained here.

4.3.1 Bags and Multisets

Bags are like sets but allow multiple occurrences of elements. Formally, a *bag* with underlying set of elements $\mathbb{C}$ is a mapping $M : \mathbb{C} \to \mathbb{N}$. Bags are often also called multisets, but we reserve the term multiset for a concept which allows one to express a deficit of certain elements as well. Formally, a *multiset* with underlying set of elements $\mathbb{C}$ is a mapping $M : \mathbb{C} \to \mathbb{Z}$. We shall use the operations $\cup, \cap, +, -$ defined on multisets, as well as relations $\subseteq$ and $\in$ between multisets and between an element and a multiset, respectively. We recall briefly their definitions: $(M \cup M')(x) = \mathsf{max}(M(x), M'(x))$, $(M \cap M')(x) = \mathsf{min}(M(x), M'(x))$, $(M + M')(x) = M(x) + M'(x)$, $(M - M')(x) = M(x) - M'(x)$, $M \subseteq M'$ iff $M(x) \leq M'(x)$ for all $x \in \mathbb{C}$. The operation $+$ is sometimes called *additive union*. Bags are closed under all operations above with the exception of $-$. Note that the operation $\cup$ returns a bag if at least one of its operands is a bag. For convenience, multisets with a limited number of elements are sometimes denoted as, for example, $M = [2x, -y]$, instead of $M(x) = 2$, $M(y) = -1$, $M(z) = 0$ for all $z \neq x, y$. In this notation, $[\,]$ stands for the *empty* multiset, i.e., $[\,](x) = 0$ for all $x \in \mathbb{C}$. We further abbreviate $M + [x]$ by $M + x$ and $M - [x]$ by $M - x$. Both multisets and bags will be denoted by M or N (with primes and subscripts), it will always be clear from the context when a bag is meant. For any bag, let $\mathsf{set}(M)$ denote its set of elements, that is, $M = \{x \in \mathbb{C} \mid M(x) > 0\}$. Note that a bag is also a multiset, while a multiset is also a bag only if it maps all elements to non-negative numbers.

4.3.2 Grammar

Table 4.1: Syntax

Expr	$::=$	*Factor* $\mid$ *Expr* $\cdot$ *Expr*
Factor	$::=$	`new`x $\mid$ `del`x $\mid$ `lock`M $\mid$ `free`M $\mid$ `use`M $\mid$ `nop`
		$\mid$ (*Expr* + *Expr*) $\mid$ (*Expr* $\Vert$ *Expr*) $\mid$ *ScExp*
ScExp	$::=$	$\{M, \mathit{Expr}\}$
M	$::=$	bag of elements from $\mathbb{C}$
Prog	$::=$	`nil` $\mid$ *Prog*, $x \multimap$ *Expr*

Component expressions are given by the syntax in Table 4.1. We let capital letters $A, \ldots, E$ (with primes and subscripts) range over *Expr*. A *component program* P is a comma-separated list starting with `nil` and followed by zero or more *component declarations*, which are of the form $x \multimap \mathit{Expr}$, with $x \in \mathbb{C}$ (`nil` will usually be omitted, except in the case of a program containing no declarations). $\mathsf{dom}(P)$ denotes the set of component names declared in P (so $\mathsf{dom}(\texttt{nil}) = \varnothing$). Declarations of the form $x \multimap$ `nop` are used for *primitive* components, i.e., components that do not use *subcomponents*.

We have two primitives `new` and `del` for creating and deleting instances of a component, three primitives `free`,`lock` and `use` for specifying usage of instances of components and four primitives for composition: sequential composition denoted by juxtaposition, $+$ for choice (also called sum), $\Vert$ for parallel and $\{\ldots\}$ for scope. Note that instances of the same component cannot be distinguished. The effect of `lock` is therefore to decrease the number of instances available for usage, while `free` increases this number.

Executing the sum $E_1 + E_2$ means choosing either one of the expressions E_1 or E_2 and executing that one. Executing E_1 and E_2 in parallel, that is, executing $(E_1 \parallel E_2)$, means executing both expressions in some arbitrary interleaved order. Executing an expression inside a scope, $\{[\,], E\}$ means executing E, while only allowing deletion of instances inside the same scope, and after the execution of E, deleting all instances inside the scope.

The grammatical ambiguity in the rule for *Expr* is unproblematic. Like in process algebra, sequential composition can be viewed as an associative multiplication operation and products may be denoted as $E\,E'$ instead of $E \cdot E'$. The operations $+$ and $\parallel$ are also associative and we only parenthesize if necessary to prevent ambiguity. Sequential composition has the highest precedence, followed by $\parallel$ and then $+$. The primitive `nop` models zero or more operations that do not involve component instantiation or deallocation.

In the third clause of the grammar we define *scope expressions*, used to limit the lifetime of instances and the scope of deletion. A scope expression is a pair of a bag, called the local store, and an expression. Scope expressions appearing in a *component declaration* in a program are required to have an empty local store. Non-empty local stores only appear *during execution* of a program.

Definition 4.3.1. *By* $\mathsf{var}(E)$ *we denote the set of component names occurring in E, formally defined by* $\mathsf{var}(\mathtt{nop}) = \varnothing$, $\mathsf{var}(\mathtt{new}\,x) = \mathsf{var}(\mathtt{del}\,x) = \{x\}$,$\mathsf{var}(\mathtt{use}\,M) = \mathsf{var}(\mathtt{free}\,M) = \mathsf{var}(\mathtt{lock}\,M) = \mathsf{set}(M)$, $\mathsf{var}(E_1 + E_2) = \mathsf{var}(E_1 \parallel E_2) = \mathsf{var}(E_1\,E_2) = \mathsf{var}(E_1) \cup \mathsf{var}(E_2)$ *and* $\mathsf{var}(\{M, E\}) = \mathsf{set}(M) \cup \mathsf{var}(E)$.

Definition 4.3.2. *The* size *of an expression E, denoted* $\sigma(E)$*, is defined by* $\sigma(\mathtt{new}\,x) = \sigma(\mathtt{del}\,x) = \sigma(\mathtt{use}\,N) = \sigma(\mathtt{lock}\,N) = \sigma(\mathtt{free}\,N) = \sigma(\mathtt{nop}) = 1$, $\sigma(\{M, E\}) = \sigma(E) + 1$ *and* $\sigma(A + B) = \sigma(AB) = \sigma(A||B) = \sigma(A) + \sigma(B) + 1$. *The size of a program P, denoted* $\sigma(P)$*, is defined by* $\sigma(P, x \multimap A) = \sigma(P) + 1 + \sigma(A)$ *and* $\sigma(\mathtt{nil}) = 1$.

4.3.3 Examples

We assume that a program is executed by executing $\texttt{new}\,x$, where x is the last component declared in the program, starting with empty stores of component instances. Examples of programs that will execute properly and will be well-typed are

Example 4.3.3.

$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\texttt{use}\,[x]\,\texttt{lock}\,[x]\,\texttt{free}\,[x]$$
$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\texttt{new}\,x\,\{[\,],(\texttt{use}\,[x] \parallel \texttt{lock}\,[x])\}\,\texttt{free}\,[x]$$

Examples of programs that can, for some reason, produce an error are:

Example 4.3.4.

$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\texttt{new}\,x\,\{[\,],(\texttt{use}\,[x] \parallel \texttt{lock}\,[x])\}$$
$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\texttt{lock}\,[x]\,\texttt{use}\,[x]\,\texttt{free}\,[x]$$
$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\{[\,],(\texttt{use}\,[x] \parallel \texttt{lock}\,[x])\}\,\texttt{free}\,[x]$$
$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\texttt{free}\,[x]\,\texttt{lock}\,[x]$$
$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\{[\,],(\texttt{use}\,[x] + \texttt{lock}\,[x])\}\,\texttt{free}\,[x]$$

The first program leaves one instance of x locked after execution. The second will get stuck as no instance of x will be available for use by the use-statement. The third might also get stuck. Note that there exists an error-free execution of the third program, where the left branch of $(\texttt{use}\,[x] \parallel \texttt{lock}\,[x])$ is executed before the right one. But as we do not wish to make any assumptions about the scheduling of the parallel execution, we consider this an error. The fourth program tries to free a component instance that is not locked. The fifth program has a run in which $\texttt{free}\,[x]$ is executed, but no instance of x has been locked.

C++ Example

We now describe the model of the example program in Fig. 4.1 (page 113). Functions (such as EX) as well as objects on the free store (such as C_instance) are modeled as components. We let $\texttt{call}\,f$ abbreviate $\texttt{new}\,f\,\texttt{del}\,f$ and use this expression to model a function call. Note that f is deleted automatically by $\texttt{call}\,f$, which models the (automatic) deallocation of stack objects created by f. However, the subcomponents

$$\begin{aligned}
&c \;\;\; \prec\; \texttt{nop},\\
&p_1 \prec \texttt{use}\,[c],\\
&p_2 \prec \texttt{lock}\,[c]\,\texttt{free}\,[c],\\
&p_3 \prec \texttt{use}\,[c],\\
&ex \;\prec\; \texttt{new}\,c\,((\texttt{call}\,p_1 + \texttt{call}\,p_2) \parallel \texttt{call}\,p_3)\,\texttt{del}\,c
\end{aligned}$$

Figure 4.2: Program P, a model of the C++ program in Fig. 4.1.

of f are not deleted by $\texttt{del}\,f$. We use small letters for the component names and model functions as components, where the function body is given by the right hand side of the declaration. Since `P2` needs exclusive access to an instance of `C` we add $\texttt{lock}\,[c]\,\texttt{free}\,[c]$ to the declaration of p_2. For p_1 and p_3 we indicate the non-exclusive usage by $\texttt{use}\,[c]$. Collecting all declarations we get the program in Fig. 4.2.

4.4 Operational Semantics

A *state*, or state expression, is a pair $(M_u, \{M, E\})$ consisting of a bag M_u (called the global store) with underlying set of elements $\mathbb{C}$, and a scope expression $\{M, E\}$. The store M in this scope expression is called the local store of the expression. An *initial state* is of the form $([\,], \{[\,], \texttt{new}\,x\})$, and a *terminal state* is of the form $(M_u, \{M, \texttt{nop}\})$.

A state $(M_u, \{M, E\})$ expresses that we execute E with a local bag M and a global bag M_u of instances of components. The local stores keep track of the instances allocated inside the corresponding scopes. The instances in the local store are deleted when the corresponding scope is exited. The global store keeps track of all instances available for `lock` or `free`. The intuition is that in any state during the execution a well-typed program, the global store should be a subset of the sum of all the local stores. (See also Corollary 4.8.4, page 140.)

The operational semantics is given in Tables 4.2 and 4.3 as a state transition system in the style of structural operational semantics [61]. The inductive rules are osPar1, osPar2, osScp and osSeq. The other rules are not inductive, but osNew, osDel, osLock, osUse and osPop are conditional with the condition specified as a premiss of the rule. The transition relation with respect to a program P is denoted by $\rightsquigarrow_P$, with transitive and reflexive closure by $\rightsquigarrow_P^*$.

Remark 4.4.1. *The type systems in [3, 5, 67, 68] can be used to test whether the deletion of instances is safe, by first translating* `use`, `lock`, *and* `free` *to* `nop`. *We need therefore only consider programs where deletion of instances from the local store is safe. That is, whenever* `del`x *is executed, we can assume there is an x in the corresponding local store.*

4.4.1 Unsafe States

A *stuck state* is usually defined as a state which is not terminal, and where there is no possible next transition. We wish to use a different condition, because we want to assure that all possible runs are error-free. This means that we do not assume anything about the interleaving used in parallel executions. This is more in line with how parallelism works by default in many environments, for example with `pthreads` and C++ without mutex locking. Informally, we call a state *unsafe* if there is at least one transition which cannot be used in this state, but which would be possible with a larger global store. For example, $([\,], \{[x], \texttt{lock}[x] \parallel \texttt{free}[x]\})$ is an unsafe state, because using osPar1 is possible with global store containing x.

Definition 4.4.2 (Unsafe states). *Given a component program P, a state $(M_u, \{M, E\})$ is called unsafe if and only if there exist bags M'_u, M' and N and an expression E' such that $(M_u + N, \{M, E\}) \rightsquigarrow_P (M'_u + N, \{M', E'\})$, but not $(M_u, \{M, E\}) \rightsquigarrow_P (M'_u, \{M', E'\})$*

It is also possible to characterize the unsafe states with the following inductive rules parametrized by a program P and bags M_u and M. The base cases are that for all x and N, where $x \notin M_u$ and $N \not\subseteq M_u$, $(M_u, \{M, \texttt{lock}\,N\})$, $(M_u, \{M, \texttt{use}\,N\})$, $(M_u, \{M, \texttt{del}\,x\})$ and $(M_u, \{M, \{N, \texttt{nop}\}\})$ are unsafe. The induction cases are that for all expressions E and F, if $(M_u, \{M, E\})$ is unsafe then for all bags N, also $(M_u, \{N, \{M, E\}\})$, $(M_u, \{M, EF\})$, $(M_u, \{M, E \parallel F\})$, and $(M_u, \{M, F \parallel E\})$ are unsafe. There is no induction case for $E + F$, since $(M_u, \{M, E + F\})$ is safe. That is, osAlt1 or osAlt2 can be applied independent of the size of the global store. Proving equality of the inductive characterization and Definition 4.4.2 is done by induction on the derivation of the step $(M_u + N, \{M, E\}) \rightsquigarrow_P (M'_u + N, \{M', E'\})$. Recall also that deletion of component instances in the local store is assumed to always be safe, as this can be assured by the system in [3]

Table 4.2: Transition rules for a component program P (continued in Table 4.3)

(osNop)
$$\frac{}{(M_u, \{M, \texttt{nop}\, E\}) \rightsquigarrow_P (M_u, \{M, E\})}$$

(osNew)
$$\frac{x \multimap\!\!\prec A \in P}{(M_u, \{M, \texttt{new}\, x\}) \rightsquigarrow_P (M_u + x, \{M + x, A\})}$$

(osDel)
$$\frac{x \in (M \cap M_u)}{(M_u, \{M, \texttt{del}\, x\}) \rightsquigarrow_P (M_u - x, \{M - x, \texttt{nop}\})}$$

(osLock)
$$\frac{N \subseteq M_u}{(M_u, \{M, \texttt{lock}\, N\}) \rightsquigarrow_P (M_u - N, \{M, \texttt{nop}\})}$$

(osFree)
$$\frac{}{(M_u, \{M, \texttt{free}\, N\}) \rightsquigarrow_P (M_u + N, \{M, \texttt{nop}\})}$$

(osUse)
$$\frac{N \subseteq M_u}{(M_u, \{M, \texttt{use}\, N\}) \rightsquigarrow_P (M_u, \{M, \texttt{nop}\})}$$

(osScp)
$$\frac{(M_u, \{N, A\}) \rightsquigarrow_P (M'_u, \{N', A'\})}{(M_u, \{M, \{N, A\}\}) \rightsquigarrow_P (M'_u, \{M, \{N', A'\}\})}$$

Table 4.3: Transition rules for a component program P (continued from Table 4.2)

(osPop)
$$\frac{N \subseteq M_u}{(M_u, \{M, \{N, \mathtt{nop}\}\}) \rightsquigarrow_P (M_u - N, \{M, \mathtt{nop}\})}$$

(osAlt$i$)
$$\frac{i \in \{1,2\}}{(M_u, \{M, (E_1 + E_2)\}) \rightsquigarrow_P (M_u, \{M, E_i\})}$$

(osSeq)
$$\frac{(M_u, \{M, A\}) \rightsquigarrow_P (M'_u, \{M', A'\})}{(M_u, \{M, A\, E\}) \rightsquigarrow_P (M'_u, \{M', A'\, E\})}$$

(osParEnd)
$$\frac{}{(M_u, \{M, (\mathtt{nop} \parallel \mathtt{nop})\}) \rightsquigarrow_P (M_u, \{M, \mathtt{nop}\})}$$

(osPar1)
$$\frac{(M_u, \{M, E_1\}) \rightsquigarrow_P (M'_u, \{M', E'_1\})}{(M_u, \{M, (E_1 \parallel E_2)\}) \rightsquigarrow_P (M'_u, \{M', (E'_1 \parallel E_2)\})}$$

(osPar2)
$$\frac{(M_u, \{M, E_2\}) \rightsquigarrow_P (M'_u, \{M', E'_2\})}{(M_u, \{M, (E_1 \parallel E_2)\}) \rightsquigarrow_P (M'_u, \{M', (E_1 \parallel E'_2)\})}$$

while ignoring `lock`, `free`, and `use`. A state which is not unsafe is called *safe*.

4.4.2 Valid States

For some state $(M_u, \{M, E\})$ in a run, M_u models all component instances available for usage. We must therefore have M_u not larger than the sum of N in all subexpressions $\{N, A\}$ of E. For example $([x], \{[\,], \mathtt{nop}\})$ should not appear in a run because $M_u \supset [\,]$. Conditions for this to be true will be stated later. However, there are transitions where the states in the transition fulfill this condition, while the derivation of the transition contains states which do not fulfill the condition. An example is the transition $([x], \{[x], \{[], \texttt{use}\,[x]\}\}) \rightsquigarrow_P ([x], \{[x], \{[], \texttt{nop}\}\})$, in which both states fulfill this condition, while it is the result of applying osScp to the premiss $([x], \{[], \texttt{use}\,[x]\}) \rightsquigarrow_P ([x], \{[], \texttt{nop}\})$, where none of the two states fulfill the condition.

To express this property more formally we need a way to sum all the local stores in an expression. In doing so, however, one would count in instances that will never coexist, such as in $\{M_1, E_1\} + \{M_2, E_2\}$ and $\{M_1, E_1\}\,\{M_2, E_2\}$. Therefore we also define the notion of a valid expression, in which irrelevant bags are empty.

Definition 4.4.3 (Sum of local stores)**.** *For any expression E, let ΣE be the sum of all N in subexpressions $\{N, A\}$ of E. More formally: $\Sigma\{M, E\} = M + \Sigma E$ and $\Sigma(E_1 \parallel E_2) = \Sigma(E_1\,E_2) = \Sigma(E_1 + E_2) = \Sigma E_1 + \Sigma E_2$ and $\Sigma\texttt{del}\,x = \Sigma\texttt{new}\,x = \Sigma\texttt{use}\,N = \Sigma\texttt{lock}\,N = \Sigma\texttt{free}\,N = \Sigma\texttt{nop} = [\,]$. An expression E is valid if for all subexpressions of the form $(E_1 + E_2)$ we have $\Sigma(E_1 + E_2) = [\,]$, and for all subexpressions of the form $F\,E'$, F a factor, we have $\Sigma E' = [\,]$.*

Note that an expression is valid if and only if all its subexpressions are valid. We will say that a state $(M_u, \{M, E\})$ is valid if and only if E is valid. The initial state is valid by definition. In any declaration $x \multimap E$, since only empty bags are allowed to occur in E, E is obviously valid and $\Sigma E = [\,]$.

4.5 Type System

4.5.1 Types

A *type* of a component expression is a tuple

$$X = \langle X^u, X^n, X^l, X^d, X^p, X^h \rangle$$

where X^n, X^u and X^p are bags and X^l, X^d and X^h are multisets. We use $U, \ldots, Z$ to denote types. The properties of the different parts of the types are summarized in Table 4.4, and will be explained below. The bag X^u (u for "usage") contains the minimum size the global store must have for an expression to be safely executed.

Because of sequential composition, we also need a multiset X^l. To run the expression $E_1\,E_2$, we must not only know the minimum safe

Table 4.4: The parts of the types

X^u:	Minimum size of the global store for safe execution.
X^n:	Largest decrease of the global store during execution.
X^l:	Lower bound of the net effect on the global store.
X^d:	Net change in the difference between the local and the global store.
X^p:	Maximum increase, during execution, of the difference between the global store and the sum of all local stores.
X^h:	Maximum net effect on the difference between the global store and the sum of all the local stores.

sizes for executing E_1 and E_2 separately, but also how much E_1 decreases or increases the global store. The multiset X^l therefore contains, for each $x \in \mathbb{C}$, the *lowest* net increase in the number of instances in the global store after the execution of the expression. (Where a decrease is negative increase.) This implies that, if the type of E is X and if $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', \texttt{nop}\})$, then $X^l \subseteq M'_u - M_u$.

The scope operator makes necessary the component X^d. When a scope is *popped* with the rule osPop, the remaining bag in the scope is subtracted from the global store. The difference between these two bags must therefore be controlled by X^d. In addition, concerning the two alternatives in a choice expression, the net effect on the difference between the global store and the local store are required to be equal. This corresponds to the requirement "$X_1^d = X_2^d$" on the rule Alt. An example of an invalid expression excluded by this rule is $(\texttt{lock}\, x + \texttt{use}\, x)$. If the latter expression was allowed in a program, it would not be possible to give the guarantees needed for osPop to the number of instances of x locked after execution. The multiset X^d therefore contains the exact change in the difference between the local store and the global store made by execution of the expression. This difference is required to be independent of how the expression is executed. This implies that, if the type of E is X and if $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', \texttt{nop}\})$, then $X^d = (M'_u - M') - (M_u - M)$.

Parallel composition necessitates the bag X^n. The minimum safe size for executing $(E_1 \parallel E_2)$ depends not only on the minimum safe size for executing each of E_1 and E_2, but also on how much each of them decreases the global store. For example, both $\texttt{use}\,x$ and $\texttt{lock}\,x\,\texttt{free}\,x$ need one instance of x, but $\texttt{use}\,x \parallel \texttt{use}\,x$ also needs only one, whereas $\texttt{lock}\,x\,\texttt{free}\,x \parallel \texttt{lock}\,x\,\texttt{free}\,x$ needs two instances of x. X^n contains, for each $x \in \mathbb{C}$, the highest *negative* net change in the number of instances in the global store during the execution of the expression. This implies that, if the type of E is X and if $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', E'\})$, then $-X^n \subseteq M'_u - M_u$.

As seen in Example 4.3.4 in Section 4.3.3, there are grammatically correct programs that "free" instances that are not locked. So far, we have not distinguished between $\texttt{free}\,[x]\,\texttt{lock}\,[x]$ and $\texttt{lock}\,[x]\,\texttt{free}\,[x]$. Obviously, these expressions cannot be assigned the same type. For example, the program

$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\texttt{free}\,[x]\,\texttt{lock}\,[x]$$

is wrong, and should not be well-typed, while the program

$$x \multimap \texttt{nop}, y \multimap \texttt{new}\,x\,\texttt{lock}\,[x]\,\texttt{free}\,[x]$$

is correct and should be well-typed. There is a need for types concerned with the difference between the number of instances in the sum of all local stores and the number of instances in the global store. If $(M_u, \{M, E\})$ is a state during the execution of a component program, then the value of $(M_u - \Sigma\{M, E\})(x)$ for a component x is negative if an instance of x is locked, but not yet freed, and positive if it has been freed without being locked. The latter is seen as an error and should not occur in the run of a well-typed program. The bag X^p and multiset X^h are used for keeping track of the set $M_u - \Sigma\{M, E\}$, and contain, the highest *positive* change during execution and the *highest* net increase of this bag after execution. This implies that if the type of E is X, then if $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', E'\})$ then $X^p \supseteq (M'_u - \Sigma\{M', E'\}) - (M_u - \Sigma\{M, E\})$. And, if $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', \texttt{nop}\})$, we get $X^h \supseteq (M'_u - M') - (M_u - \Sigma\{M, E\})$. In the type of a well-typed program these parts must be empty bags. These parts of the type can also be seen to give guarantees about the maximum decrease in the difference between the sum of the local stores and the global store, that is, the maximum decrease in the number of locked instances.

4.5.2 Typing Rules

The typing rules in Table 4.5 and Table 4.6 must be understood with the above interpretation in mind. They define a ternary typing relation $\Gamma \vdash E : X$ and a binary typing relation $\vdash P : \Gamma$ in the usual inductive way. Here Γ is usually called a *basis*, mapping component names to the type of the expression in its declaration. In the relation $\vdash P : \Gamma$, Γ can be viewed as a type of P. An expression of the form $\Gamma \vdash E : X$ or $\vdash P : \Gamma$ will be called a *typing* and will also be phrased as 'expression E has type X in Γ' or 'program P has type Γ', respectively.

A basis Γ is a partial mapping of components $x \in \mathbb{C}$ to types. By $\mathtt{dom}(\Gamma)$ we denote the domain of Γ, and for any $x \in \mathtt{dom}(\Gamma)$, $\Gamma(x)$ denotes its type in Γ. For a set $S \subseteq \mathtt{dom}(\Gamma)$, $\Gamma|_S$ is Γ restricted to the domain S. For any $x \in \mathbb{C}$ and type X, $\{x \mapsto X\}$ denotes a basis with domain $\{x\}$ and which maps x to X. An expression E is called *typable* in Γ if $\Gamma \vdash E : X$ for some type X. The latter type X will be proved to be unique and will sometimes be denoted by $\Gamma(E)$.

Definition 4.5.1 (Well-typed program). *A program P with at least one declaration is* well-typed *if there are Γ and X such that $\vdash P : \Gamma$, $\Gamma \vdash \mathtt{new}\, x : X$ and $X^d = X^u = X^p = X^h = []$, where x is the last component declared in P.*

The condition in Definition 4.5.1 that parts X^d, X^u, X^p, and X^h be empty deserves an explanation. X^d must be empty, because the global and local store must be equal in the final state, that is, no instances are still locked when the program ends. X^u is the minimum safe size of the global store, and we assume the program is executed starting with an empty global store, so X^u must be empty. X^p must be empty, because this is the only way to guarantee that, during execution, no instance is freed, unless there already is a locked instance of the same component. X^h must be empty, because we must guarantee that there does not remain any locked instances after execution.

Type inference in this system is similar to [3, 5, 67, 68]. In particular, the type inference algorithm has quadratic runtime. An implementation of the type system can be downloaded from the author's website.

Table 4.5: Typing Rules (continued in Table 4.6))

(AxmP)
$$\frac{}{\vdash \texttt{nil}: \varnothing}$$

(Axm)
$$\frac{}{\Gamma \vdash \texttt{nop}: \langle [\,], [\,], [\,], [\,], [\,], [\,] \rangle}$$

(New)
$$\frac{\Gamma(x) = X}{\Gamma \vdash \texttt{new}\, x: \langle X^u, X^n, X^l + x, X^d, X^p, X^h \rangle}$$

(Del)
$$\frac{\Gamma(x) = X}{\Gamma \vdash \texttt{del}\, x: \langle [x], [x], [-x], [\,], [\,], [\,] \rangle}$$

(Lock)
$$\frac{\mathsf{set}(N) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash \texttt{lock}\, N: \langle N, N, -N, -N, [\,], -N \rangle}$$

(Use)
$$\frac{\mathsf{set}(N) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash \texttt{use}\, N: \langle N, [\,], [\,], [\,], [\,], [\,] \rangle}$$

(Free)
$$\frac{\mathsf{set}(N) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash \texttt{free}\, N: \langle [\,], [\,], N, N, N, N \rangle}$$

(Prog)
$$\frac{\Gamma \vdash E: X, \ \vdash P: \Gamma, \ x \notin \mathsf{dom}(\Gamma)}{\vdash P, x \multimap E: \Gamma \cup \{x \mapsto X\}}$$

4.6 C++ Example Continued

Recall the C++ program in Fig. 4.1 and the component program in Fig. 4.2. Type inference gives the following results:

$$\begin{aligned}
&\texttt{call}\, p_1: \langle [c], [], [], [], [], [] \rangle,\\
&\texttt{call}\, p_2: \langle [c], [c], [], [], [], [] \rangle,\\
&\texttt{call}\, p_3: \langle [c], [], [], [], [], [] \rangle,\\
&\texttt{call}\, ex: \langle [c], [], [], [], [], [] \rangle
\end{aligned}$$

Table 4.6: Typing Rules (continued from Table 4.5))

(Par)

$$\frac{\Gamma \vdash E_1 : X_1,\ \Gamma \vdash E_2 : X_2}{\Gamma \vdash E_1 \parallel E_2 : \left\langle \begin{array}{l} (X_1^u + X_2^n) \cup (X_2^u + X_1^n), X_1^n + X_2^n, \\ X_1^l + X_2^l, X_1^d + X_2^d, X_1^p + X_2^p, X_1^h + X_2^h \end{array} \right\rangle}$$

(Alt)

$$\frac{\Gamma \vdash E_1 : X_1,\ \Gamma \vdash E_2 : X_2,\ X_1^d = X_2^d}{\Gamma \vdash E_1 + E_2 : \langle X_1^u \cup X_2^u, X_1^n \cup X_2^n, X_1^l \cap X_2^l, X_1^d, X_1^p \cup X_2^p, X_1^h \cup X_2^h \rangle}$$

(Seq)

$$\frac{\Gamma \vdash E_1 : X_1,\ \Gamma \vdash E_2 : X_2}{\Gamma \vdash E_1\, E_2 : \left\langle \begin{array}{l} X_1^u \cup (X_2^u - X_1^l), X_1^n \cup (X_2^n - X_1^l), \\ X_1^l + X_2^l, X_1^d + X_2^d, X_1^p \cup (X_2^p + X_1^h), X_1^h + X_2^h \end{array} \right\rangle}$$

(Scope)

$$\frac{\Gamma \vdash E : X,\ \mathsf{set}(M) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash \{M, E\} : \langle X^u \cup (M - X^d), X^n \cup (M - X^d), X^d - M, X^d - M, X^p, X^h \rangle}$$

This signals in the first multiset ($\cdot^u$) of the type of $\texttt{call}\,ex$ that one instance of c is needed before execution of $\texttt{call}\,ex$. This is caused by the possible choice of $\texttt{call}\,p_2$ instead of $\texttt{call}\,p_1$ by ex, whereby there could be parallel calls to p_2 and p_3. One way to fix this is to instantiate two instances of C instead of just one. Then one instance could be passed to P1 or P2 and the second to P3. This means that P is changed by changing ex into $ex' \multimap \texttt{new}\,c\,\texttt{new}\,c\,((\texttt{call}\,p_1 + \texttt{call}\,p_2) \parallel \texttt{call}\,p_3)\,\texttt{del}\,c$. The type of $\texttt{call}\,ex'$ is $\langle [], [], [c], [], [], [] \rangle$ which signals that the expression now can be executed starting with an empty store. But the third multiset ($\cdot^l$) signals that there is one instance of c left after execution. This can be fixed by deleting one more instance, that is, changing ex' to $ex'' \multimap \texttt{new}\,c\,\texttt{new}\,c\,((\texttt{call}\,p_1 + \texttt{call}\,p_2) \parallel \texttt{call}\,p_3\,\texttt{del}\,c)\,\texttt{del}\,c$. The type of $\texttt{call}\,ex''$ is $\langle [], [], [], [], [], [] \rangle$.

Another way of solving the original problem is to remove the parallelism from the program, such that ex is changed to, e.g.,

$$ex''' \multimap \texttt{new}\,c\,(\texttt{call}\,p_1 + \texttt{call}\,p_2)\,\texttt{call}\,p_3\,\texttt{del}\,c$$

The type of callex''' is also $\langle [], [], [], [], [], [] \rangle$.

4.7 Properties of the Type System

This section contains several basic lemmas about the type system.

We will use quite often the fact that if a multiset is not a superset of a union, then it is not a superset of both the multisets joined by the union. That is, for any multisets A, B and C, $A \not\supseteq B \cup C$ implies $A \not\supseteq B \vee A \not\supseteq C$. This can be shown by using the definitions of union and $\supseteq$ for multisets, or via the, perhaps more intuitive, contrapositive statement, that is, if both $A \supseteq B$ and $A \supseteq C$, then also $A \supseteq B \cup C$.

Lemma 4.7.1 (Basics). *1. If* $\Gamma \vdash E : X$, *then* $\mathsf{var}(E) \subseteq \mathsf{dom}(\Gamma)$.

2. If $\vdash P : \Gamma$ *and* $\Gamma \vdash E : X$, *then* $\mathsf{dom}(P) = \mathsf{dom}(\Gamma)$ *and* $-X^u \subseteq -X^n \subseteq X^l$ *and* $X^h \subseteq X^p$.

Proof. 1. By structural induction on the derivation of $\Gamma \vdash E : X$.

2. By induction on $\vdash P : \Gamma$ one proves $\mathsf{dom}(P) = \mathsf{dom}(\Gamma)$. The two last parts require a double induction, the primary induction on the length of Γ and a secondary induction on the derivation $\Gamma \vdash E : X$. The primary base case, $\Gamma = \emptyset$ and $E = \texttt{nop}$ is trivial. Now let $\Gamma \vdash E : X$ for some non-empty Γ and assume the result has been proved for all shorter bases. We prove $X^n \subseteq X^u$, $-X^n \subseteq X^l$ and $X^h \subseteq X^p$ by a secondary induction on the derivation of $\Gamma \vdash E : X$. The secondary base cases $E = \texttt{nop}$, $E = \texttt{del}\,x$, $E = \texttt{free}\,N$, $E = \texttt{lock}\,N$ and $E = \texttt{use}\,N$ are trivial. Consider the base case $E = \texttt{new}\,x$ with $\Gamma(x) = X'$ for some X'. Then $\Gamma' \vdash E' : X'$ for some $\Gamma' \subset \Gamma$ with $x \multimap\!\!\prec E' \in P$. Now we can apply the induction hypothesis to Γ' and the result for X follows from that of X'. Consider the secondary induction case where $\Gamma \vdash E : X$ is inferred from the following application of the rule Par:

$$\text{(Par)}\quad \frac{\Gamma \vdash E_1 : X_1,\ \Gamma \vdash E_2 : X_2}{\Gamma \vdash E_1 \parallel E_2 : \left\langle \begin{array}{l} (X_1^u + X_2^n) \cup (X_2^u + X_1^n), X_1^n + X_2^n, \\ X_1^l + X_2^l, X_1^d + X_2^d, X_1^p + X_2^p, X_1^h + X_2^h \end{array} \right\rangle}$$

Then $X^u = (X_1^u + X_2^n) \cup (X_2^u + X_1^n) \supseteq X_1^u + X_2^n \supseteq X_1^n + X_2^n = X^n$. In the second last step we apply the induction hypothesis. That $-X^n \subseteq X^l$

and $X^h \subseteq X^p$ follows by applying the induction hypothesis. The secondary induction cases Seq, Scp and Alt follow by similar calculations. $\square$

Lemma 4.7.2 (Associativity). *If $\Gamma \vdash A : X$, $\Gamma \vdash B : Y$ and $\Gamma \vdash C : Z$, then the two ways of typing the expression $A\,B\,C$ by the rule Seq, corresponding to the different parses $(A\,B)\,C$ and $A\,(B\,C)$, lead to the same type.*

Proof. By applying Seq to $\Gamma \vdash A : X$ and $\Gamma \vdash B : Y$ we get

$$\Gamma \vdash A\,B : \left\langle \begin{array}{l} X^u \cup (Y^u - X^l), X^n \cup (Y^n - X^l), \\ X^l + Y^l, X^d + Y^d, X^p \cup (Y^p + X^h), X^h + Y^h \end{array} \right\rangle$$

and combining this with $\Gamma \vdash C : Z$ we get

$$\Gamma \vdash A\,B\,C : \left\langle \begin{array}{l} (X^u \cup (Y^u - X^l)) \cup (Z^u - (X^l + Y^l)), \\ (X^n \cup (Y^n - X^l)) \cup (Z^n - (X^l + Y^l)), \\ (X^l + Y^l) + Z^l, (X^d + Y^d) + Z^d, \\ (X^p \cup (Y^p + X^h)) + Z^p, (X^h + Y^h) + Z^h \end{array} \right\rangle$$

By applying Seq to $\Gamma \vdash B : Y$ and $\Gamma \vdash C : Z$ we get

$$\Gamma \vdash B\,C : \left\langle \begin{array}{l} Y^u \cup (Z^u - Y^l), Y^n \cup (Z^n - Y^l), \\ Y^l + Z^l, Y^d + Z^d, Y^p \cup (Z^p + Y^h), Y^h + Z^h \end{array} \right\rangle$$

and combining this with $\Gamma \vdash A : X$ we get $\Gamma \vdash A\,B\,C : \langle X^u \cup ((Y^u \cup (Z^u - Y^l)) - X^l), X^n \cup ((Y^n \cup (Z^n - Y^l)) - X^l), X^l + (Y^l + Z^l), X^d + (Y^d + Z^d), X^p \cup (Y^p \cup (Z^p + Y^h) + X^h), X^h + (Y^h + Z^h)\rangle$. It remains to prove that the two types resulting from the combination are equal. For the parts ^{l}, ^{d} and ^{h} of the tuples this trivially follows from the associativity of $+$ for multisets. For the remaining parts this follows from the associativity of $\cup$ and the distributivity of $+$ and $-$ over $\cup$. $\square$

The following lemma is necessary since the typing rules are not fully syntax-directed. If, e.g., $E_1 = A \cdot B$, then the type of $E_1 \cdot E_2$ could have been inferred by an application of the rule Seq to A and $B\,E_2$. In that case we apply the previous lemma.

Lemma 4.7.3 (Inversion).

1. *If* $\vdash P : \Gamma$ *and* $\Gamma(x) = X$, *then there exists a program* P' *and an expression* A *such that* $P', x \multimap A$ *is the initial segment of* P *and* $\vdash P' : \Gamma|_{\mathsf{dom}(P')}$ *and* $\Gamma|_{\mathsf{dom}(P')} \vdash A : X$.

2. *If* $\Gamma \vdash \mathtt{new}\, x : X$, *then* $X = \langle \Gamma(x)^u, \Gamma(x)^n, \Gamma(x)^l + x, \Gamma(x)^d, \Gamma(x)^p, \Gamma(x)^h \rangle$.

3. *If* $\Gamma \vdash \mathtt{del}\, x : X$, *then* $X = \langle [x], [x], [-x], [\,], [\,], [\,] \rangle$.

4. *If* $\Gamma \vdash \mathtt{lock}\, N : X$, *then* $X = \langle N, N, -N, -N, [\,], -N \rangle$.

5. *If* $\Gamma \vdash \mathtt{free}\, N : X$, *then* $X = \langle [\,], [\,], N, N, N, N \rangle$.

6. *If* $\Gamma \vdash \mathtt{use}\, N : X$, *then* $X = \langle N, [\,], [\,], [\,], [\,], [\,] \rangle$.

7. *If* $\Gamma \vdash \mathtt{nop} : X$, *then* $X = \langle [\,], [\,], [\,], [\,], [\,], [\,] \rangle$.

8. *For* $\circ \in \{+, \|, \cdot\}$, *if* $\Gamma \vdash (E_1 \circ E_2) : X$, *then there exists* X_i *such that* $\Gamma \vdash E_i : X_i$ *for* $i = 1, 2$. *Moreover,*
$X = \langle X_1^u \cup X_2^u, X_1^n \cup X_2^n, X_1^l \cap X_2^l, X_1^d, X_1^p \cup X_2^p, X_1^h \cup X_2^h \rangle$
and $X_1^d = X_2^d$ *if* $\circ = +$,
$$X = \left\langle \begin{array}{l} (X_1^u + X_2^n) \cup (X_2^u + X_1^n), X_1^n + X_2^n, \\ X_1^l + X_2^l, X_1^d + X_2^d, X_1^p + X_2^p, X_1^h + X_2^h \end{array} \right\rangle \text{ if } \circ = \|, \text{ and}$$
$$X = \left\langle \begin{array}{l} X_1^u \cup (X_2^u - X_1^l), X_1^n \cup (X_2^n - X_1^l), \\ X_1^l + X_2^l, X_1^d + X_2^d, X_1^p \cup (X_2^p + X_1^h), X_1^h + X_2^h \end{array} \right\rangle \text{ if } \circ = \cdot.$$

9. *If* $\Gamma \vdash \{M, A\} : X$, *then there exists a type* Y, *such that* $\Gamma \vdash A : Y$ *and* $X = \langle Y^u \cup (M - Y^d), Y^n \cup (M - Y^d), Y^d - M, Y^d - M, Y^p, Y^h \rangle$.

Proof. We first prove the first part by an easy induction on $\vdash P : \Gamma$. The base case AxmP is trivial, and in the induction case we have the following application of the rule Prog:

$$\frac{\Gamma' \vdash B : Z, \ \vdash P' : \Gamma', \ y \notin \mathsf{dom}(\Gamma')}{\vdash P', y \multimap B : \Gamma' \cup \{y \mapsto Z\}}$$

If $x = y$ we have the result from the rule application. Otherwise we can apply the induction hypothesis to the premiss $\vdash P' : \Gamma'$.

The other parts are proved by structural induction on the derivation of $\Gamma \vdash E : X$. The base cases Axm, Lock, Free, Use, Del and New and the induction cases Alt, Scp and Par are obvious (no need for the induction hypothesis). The only interesting case is the rule Seq, which has three sub-cases. Consider the conclusion $\Gamma \vdash E_1\, E_2 : X$. If this has been inferred by an application of Seq with premises $\Gamma \vdash E_i : X_i$ for $i = 1,2$ we are done (no need for the induction hypothesis). However, it is possible that $E_1 = A\,B$ and that Seq is applied to A and $B\,E_2$. The third case, $E_2 = B\,C$ and Seq applied to $E_1\,B$ and C, follows by symmetry. So let $E_1 = A\,B$ and consider the following application of the rule Seq.

$$\frac{\Gamma \vdash A : Y_1, \Gamma \vdash B\,E_2 : Y_2}{\Gamma \vdash E_1\,E_2 : \left\langle \begin{array}{l} Y_1^u \cup (Y_2^u - Y_1^l), Y_1^n \cup (Y_2^n - Y_1^l), \\ Y_1^l + Y_2^l, Y_1^d + Y_2^d, Y_1^p \cup (Y_2^p + Y_1^h), Y_1^h + Y_2^h \end{array} \right\rangle}$$

The type in the conclusion is the type X for which we have to find types X_i such that $\Gamma \vdash E_i : X_i$ for $i = 1,2$, and $X = \langle X_1^u \cup (X_2^u - X_1^l), X_1^u \cup (X_2^u - X_1^l), X_1^l + X_2^l, X_1^d + X_2^d, X_1^p \cup (X_2^p + X_1^h), X_1^h + X_2^h \rangle$. By the induction hypothesis applied to $\Gamma \vdash B\,E_2 : Y_2$ we get types Z and X_2 such that $\Gamma \vdash B : Z$ and $\Gamma \vdash E_2 : X_2$. By applying Seq to $\Gamma \vdash A : Y_1$ and $\Gamma \vdash B : Z$ we get a type X_1 such that $\Gamma \vdash E_1 : X_1$. It follows by Lemma 4.7.2 (Associativity) that $X = \langle Y_1^u \cup (Y_2^u - Y_1^l), Y_1^u \cup (Y_2^u - Y_1^l), Y_1^l + Y_2^l, Y_1^d + Y_2^d, Y_1^p \cup (Y_2^p + Y_1^h), Y_1^h + Y_2^h \rangle$. □

The last lemma in this section is concerned with three forms of uniqueness of the types inferred in the type system. This is necessary in some of the proofs, and for an algorithm for type inference.

Lemma 4.7.4 (Uniqueness of types).

1. *If* $\Gamma_1 \vdash E : X$, $\Gamma_2 \vdash E : Y$ *and* $\Gamma_1|_{\mathsf{var}(E)} = \Gamma_2|_{\mathsf{var}(E)}$, *then* $X = Y$.

2. *If* $\vdash P : \Gamma$ *and* $\vdash P : \Gamma'$, *then* $\Gamma = \Gamma'$.

3. *If* $\vdash P_1 : \Gamma_1$ *and* $\vdash P_2 : \Gamma_2$ *and* P_2 *is a reordering of* a subset of P_1, *then* $\Gamma_1|_{\mathsf{dom}(P_2)} = \Gamma_2$.

Proof. 1. By structural induction on the derivation of $\Gamma_1 \vdash E : X$. In the cases of the rules Axm, Lock, Free, Use, Del and New we have that $E = \texttt{nop}$, $E = \texttt{lock}\,N$, $E = \texttt{free}\,N$, $E = \texttt{use}\,N$, $E = \texttt{del}\,x$ and $E = \texttt{new}\,x$, for some bag N and component x. In all three cases $X = Y$ follows by applying the Inversion Lemma 4.7.3 to $\Gamma_2 \vdash E : Y$.

Assume $\Gamma_1 \vdash E : X$ is inferred by the following application of the rule Par:

$$\frac{\Gamma_1 \vdash E_1 : X_1,\ \Gamma_1 \vdash E_2 : X_2}{\Gamma_1 \vdash E_1 \parallel E_2 : \left\langle \begin{array}{l} (X_1^u + X_2^n) \cup (X_2^u + X_1^n), X_1^n + X_2^n, \\ X_1^l + X_2^l, X_1^d + X_2^d, X_1^p + X_2^p, X_1^h + X_2^h \end{array} \right\rangle}$$

Applying the Inversion Lemma to $\Gamma_2 \vdash E_1 \parallel E_2 : Y$ gives types Y_i such that $\Gamma_2 \vdash E_i : Y_i$ and $Y = \left\langle \begin{array}{c} (Y_1^u + Y_2^n) \cup (Y_2^u + Y_1^n), Y_1^n + Y_2^n, \\ Y_1^l + Y_2^l, Y_1^d + Y_2^d, Y_1^p + Y_2^p, Y_1^h + Y_2^h \end{array} \right\rangle$. Now we apply the induction hypothesis to the premises $\Gamma_2 \vdash E_i : X_i$ and get $X_i = Y_i$ for $i = 1, 2$. It follows that $X = Y$.

The cases of the rules Alt, Scp and Seq are analogous to the case of Par.

2. The base case AxmP is trivial. Assume therefore $\vdash P : \Gamma$ is inferred by the following application of the rule Prog:

$$\frac{\Gamma_1 \vdash E : X, \ \vdash P_1 : \Gamma_1, \ x \notin \mathsf{dom}(\Gamma_1)}{\vdash P_1, x \multimap E : \Gamma_1 \cup \{x \mapsto X\}}$$

From applying Lemma 4.7.3 to $\vdash P : \Gamma'$ and then the induction hypothesis and part 1 of this lemma we get $\Gamma = \Gamma'$.

3. Let conditions be as above. We use induction on the derivation of $\vdash P_2 : \Gamma_2$. The base case is $P_2 = \texttt{nil}$, in which case $\Gamma_2 = \Gamma_1|_\varnothing = \varnothing$. For the induction case assume $\vdash P_2 : \Gamma_2$ has been inferred by the following application of the rule Prog:

$$\frac{\vdash P_2' : \Gamma_2', \ \Gamma_2' \vdash E : \Gamma_2(x), \ x \notin \mathsf{dom}(\Gamma_2')}{\vdash P_2', x \multimap E : \Gamma_2' \cup \{x \mapsto \Gamma_2(x)\}}$$

Since $x \in \mathsf{dom}(\Gamma_1)$ we get by the Inversion Lemma 4.7.3 that there is P_1' such that $P_1', x \multimap E$ is an initial segment of P_1 and for $\Gamma_1' = \Gamma_1|_{\mathsf{dom}(P_1')}$ we have $\vdash P_1' : \Gamma_1'$ and $\Gamma_1' \vdash E : \Gamma_1(x)$. Since $\Gamma_2' \vdash E : \Gamma_2(x)$ the Basics Lemma 4.7.1 implies $\mathsf{var}(E) \subseteq \mathsf{dom}(P_2')$. Since $\mathsf{dom}(P_1) \supset \mathsf{dom}(P_2')$ we have from the Basics Lemma 4.7.1 and the Uniqueness Lemma 4.7.4 that $\Gamma_1|_{\mathsf{dom}(P_2')} \vdash E : \Gamma_1(x)$. From the induction hypothesis $\Gamma_1|_{\mathsf{dom}(P_2')} = \Gamma_2'$ so from the Uniqueness Lemma 4.7.4 we get $\Gamma_1(x) = \Gamma_2(x)$. This combined with the induction hypothesis, $\Gamma_1|_{\mathsf{dom}(P_2')} = \Gamma_2'$, implies that $\Gamma_1|_{\mathsf{dom}(P_2)} = \Gamma_2$.

□

4.7.1 Type Inference

Type inference means to compute, for a given component program P and expression E, types Γ and X such that $\vdash P : \Gamma$ and $\Gamma \vdash E :$

X if there are such types, and to report failure otherwise. This may require reordering P, a task that should not burden the programmer. We should then prove that the type, if it exists, is independent of the specific reordering used. We prepare reordering with a lemma.

Lemma 4.7.5. *For any program P, the following are equivalent:*

1. $\vdash P : \Gamma$ for some Γ;

2. Every x is declared at most once in P and for every initial segment $P', x \multimap A$ of P there is Γ such that $\vdash P' : \Gamma$ and there is X such that $\Gamma \vdash A : X$.

Proof. For proving that 1 implies 2, assume 1 and let $P', x \multimap A$ be an initial segment of P. At some point in the derivation of $\vdash P : \Gamma$, P' is extended to $P', x \multimap A$ by the following application of Prog:

$$\frac{\vdash P' : \Gamma',\ \Gamma' \vdash A : X,\ x \notin \mathsf{dom}(\Gamma')}{\vdash P', x \multimap A : \Gamma' \cup \{x \mapsto X\}}$$

for some Γ' and X. By the premiss we get the result.

It remains to prove that 2 implies 1. This will be done by induction on the length of P. The base case `nil` is typed by AxmP. Assume $P = P', x \multimap A$ satisfies 2. Therefore $x \notin \mathsf{dom}(P')$, and there is Γ' and X such that $\vdash P' : \Gamma'$ and $\Gamma' \vdash A : X$. By Lemma 4.7.1 $x \notin \mathsf{dom}(\Gamma')$. By Prog we conclude that $\vdash P', x \multimap A : \Gamma' \cup \{x \mapsto X\}$. □

Part 2 of the above lemma partially specifies the ordering in P. For example, if P is $\texttt{nil}, x \multimap \texttt{new}\, z, y \multimap \texttt{new}\, z, z \multimap \texttt{nop}$ then both

$$P_1 = \texttt{nil}, z \multimap \texttt{nop}, x \multimap \texttt{new}\, z, y \multimap \texttt{new}\, z$$

and

$$P_2 = \texttt{nil}, z \multimap \texttt{nop}, y \multimap \texttt{new}\, z, x \multimap \texttt{new}\, z$$

satisfy 2. The Lemma 4.7.4, part 3 proves that in general types do not depend on the ordering chosen.

Theorem 4.7.6 (Type inference). *There exists an algorithm that, given a component program P and an expression E, does the following:*

1. First program P is reordered to satisfy part 2 in Lemma 4.7.5. If P cannot be reordered in such a way, or if $\mathsf{var}(E) \not\subseteq \mathsf{dom}(P)$, the algorithm reports a failure.

2. In the second phase a basis Γ and a type X such that $\vdash P : \Gamma$ and $\Gamma \vdash E : X$ are computed, if they exist. If there is no such Γ and X, the algorithm reports a failure.

The algorithm works in time $O(\sigma(P)^2 + \sigma(E)^2)$. The types X and Γ in phase 2 are unique if they exist.

Proof. After assuring there are at most one declaration of each component, phase 1 can easily be done by a topological sorting [45] of the directed graph defined by: the nodes are $\mathsf{dom}(P)$ and there is an edge from y to x if and only if there exists a declaration $x \multimap A$ in P such that y occurs in A.

For phase 2, we first use the type system to either find a Γ such that $\vdash P : \Gamma$, or decide that there is no such Γ. In the former case, we can also use the type system to either find an X such that $\Gamma \vdash E : X$, or decide that there is no such X. The failure to find Γ or X can only be caused by the condition "$X_1^d = X_2^d$" of the rule $\mathsf{Alt}i$ to fail. The inference trees have size linear in P and E, respectively. As the multiset operations are in linear time the whole phase takes quadratic time.

The algorithm reports failure if P cannot be reordered or if the types X and Γ cannot be found. Γ and X are independent of the particular reordering of P by Lemma 4.7.4. $\square$

It should be noted that we can type programs and expressions that might not safe to execute. We only prove that it is safe to run well-typed programs, starting with empty local and global stores. (See also Corollary 4.8.4, page 140.)

4.8 Correctness

This section contains lemmas and theorems connecting the type system and the operational semantics. Included are theorems comparable to what is often called preservation and progress, for example in [60]. The following lemma implies that all states in sequences representing the execution of a well-typed program are valid, as defined in Definition 4.4.3.

Lemma 4.8.1. *If $\vdash P : \Gamma$, $\Gamma \vdash E : X$, E is valid and $(M_u, \{M, E\}) \rightsquigarrow_P (M'_u, \{M', E'\})$ is a step in the operational semantics, then also E' is valid.*

Proof. By induction on the definition of $\rightsquigarrow_P$. Assume E is valid. In the cases of osDel, osLock, osFree, osUse, osPop and osParEnd, $E' = \texttt{nop}$ and hence valid. In the cases of osNop, E' is a subexpression of E and hence valid. In the case osNew, note that $\Sigma A = [\,]$ for any declaration $x \multimap A$. In the case of osAlt$i$, $E = (E_1 + E_2)$ and hence $\Sigma E = [\,]$, so also $\Sigma E' = [\,]$. In the cases of osPar1 and osPar2 we use the induction hypothesis, and that $(E_1||E_2)$ is valid if and only if both E_1 and E_2 are valid. In the case of osScp we use the induction hypothesis, and that $\{M, E\}$ is valid if and only if E. In the case of osSeq we also use the induction hypothesis, and that AE is valid if and only if A is valid, E is valid, and $\Sigma E = [\,]$. □

The next lemma fixes several properties of two states connected by a single step in the operational semantics. This is used heavily in the main theorems below. The first part expresses that typability is preserved. The remaining parts reflect the fact that every step reduces the set of reachable states, and the correct accounting of instances in each step. Hence maxima do not increase and minima do not decrease.

Lemma 4.8.2 (Invariants). *Let P be a component program, E a valid expression, Γ a basis and U a type such that $\vdash P : \Gamma$, $\Gamma \vdash E : U$, and $(M_u, \{M, E\}) \rightsquigarrow_P (M'_u, \{M', E'\})$ is a step in the operational semantics. Then we have for some type V:*

1. $\Gamma \vdash E' : V$.

2. $M'_u - V^u \supseteq M_u - U^u$, *i.e., the safety margin of the global store does not decrease.*

3. $M'_u - V^n \supseteq M_u - U^n$, *i.e., the lower bound on the global store in all reachable states does not decrease.*

4. $M'_u + V^l \supseteq M_u + U^l$, *i.e., the lower bound on the global store after execution does not decrease.*

5. $M'_u - M' + V^d = M_u - M + U^d$, *i.e., the difference between the local and the global store after execution does not change.*

6. $M'_u - \Sigma\{M', E'\} + V^p \subseteq M_u - \Sigma\{M, E\} + U^p$, *i.e., the upper bound on the difference, in any reachable state, between the global store and the sum of the local stores, does not increase.*

7. $M'_u - \Sigma\{M', E'\} + V^h \subseteq M_u - \Sigma\{M, E\} + U^h$, i.e., the upper bound on the net effect on the difference between the global store and the sum of the local stores does not increase.

Proof. All parts are proved by simultaneous induction on the definition of $\rightsquigarrow_P$. Part 1 uses the Inversion Lemma 4.7.3 to break down the typing $\Gamma \vdash E : U$. Thereafter a type for E' can be inferred in all cases.

For the base case osNew, let $\Gamma \vdash \texttt{new}\, x : U$ and consider a step $(M_u, \{M, \texttt{new}\, x\}) \rightsquigarrow_P (M_u + x, \{M + x, A\})$. By applying the Inversion Lemma 4.7.3 and the Uniqueness Lemma 4.7.4 we get that $U = \langle V^u, V^n, V^l + x, V^d, V^p, V^h \rangle$, where $V = \Gamma(A)$. Parts 4 to 7 becomes equalities, for 6 and 7 remember that $\Sigma A = [\,]$ from the restriction on programs. Parts 2 and 3 follow from $M'_u - V^\diamond = M_u - U^\diamond + x$, where $\diamond \in \{n, u\}$.

For osDel, let $\Gamma \vdash \texttt{del}\, x : U$ and consider a step $(M_u, \{M, \texttt{del}\, x\}) \rightsquigarrow_P (M_u - x, \{M - x, \texttt{nop}\})$. By applying the Inversion Lemma 4.7.3 we get $U = \langle [x], [x], [-x], [\,], [\,], [\,] \rangle$ and $V = \langle [\,], [\,], [\,], [\,], [\,], [\,] \rangle$. This makes parts 2 to 7 equalities.

osLock is treated similar to osDel. All parts except 6 become equalities, and 6 holds since $M'_u = M_u - N \subseteq M_u$.

For osFree, let $\Gamma \vdash \texttt{free}\, N : U$ and consider a step

$$(M_u, \{M, \texttt{free}\, N\}) \rightsquigarrow_P (M_u + N, \{M, \texttt{nop}\})$$

By applying the Inversion Lemma 4.7.3 we get $U = \langle [\,], [\,], N, N, N, N \rangle$ and $V = \langle [\,], [\,], [\,], [\,], [\,], [\,] \rangle$. This makes parts 2 and 3 hold from N being a bag and $M'_u = M_u + N$. Parts 4 to 7 become equalities.

For osUse, let $\Gamma \vdash \texttt{use}\, N : U$ and consider a step $(M_u, \{M, \texttt{use}\, N\}) \rightsquigarrow_P (M_u, \{M, \texttt{nop}\})$ inferred from $N \subseteq M_u$. By applying the Inversion Lemma 4.7.3 we get $U = \langle N, [\,], [\,], [\,], [\,], [\,] \rangle$ and $V = \langle [\,], [\,], [\,], [\,], [\,], [\,] \rangle$. This makes part 2 hold from N being a bag and $M'_u = M_u$. Parts 3 to 7 become equalities

For the base case osNop, let $\Gamma \vdash \texttt{nop}\, E' : U$ and consider a step

$$(M_u, \{M, \texttt{nop}\, E'\}) \rightsquigarrow_P (M_u, \{M, E'\})$$

By applying the Inversion Lemma 4.7.3 we get a type V such that $\Gamma \vdash E' : V$ and $V = U$. Parts 2 to 7 become equalities. The case osParEnd is similar.

For osAlt$i$, let $\Gamma \vdash (E_1 + E_2) : U$, and consider $(M_u, \{M, (E_1 + E_2)\}) \rightsquigarrow_P (M_u, \{M, E_i\})$. From the Inversion Lemma 4.7.3 we have X_1 and X_2 such that the typings $E_1 : X_1$ and $E_2 : X_2$, where $X_1^d = X_2^d$, hold in Γ. We have $U = \langle X_1^u \cup X_2^u, X_1^n \cup X_2^n, X_1^l \cap X_2^l, X_1^d, X_1^p \cup X_2^p, X_1^h \cup X_2^h \rangle$ and $V = X_i$. The calculations for parts 2 to 7 of the lemma are done by using $U^\diamond \supseteq X_i^\diamond$ for $\diamond \in \{u, n, p, h\}$ and $U^l \subseteq X_i^l$ using mono/antitonicity properties of $\cup$, $\cap$, $+$, and $-$, and finally that $X_i^d = U^d$. For parts 6 and 7 we also need that $\Sigma E_1 = \Sigma E_2 = [\,]$ from E valid.

For the induction case osPar1, let $\Gamma \vdash (E_1 \parallel E_2) : U$ and consider the step $(M_u, \{M, (E_1 \parallel E_2)\}) \rightsquigarrow_P (M'_u, \{M', (E'_1 \parallel E_2)\})$, inferred from the step $(M_u, \{M, E_1\}) \rightsquigarrow_P (M'_u, \{M', E'_1\})$. From the Inversion Lemma we have types X and X_2 such that typings $E_1 : X$ and $E_2 : X_2$ hold in Γ where $U^u = (X^u + X_2^n) \cup (X_2^u + X^n)$ and $U^\diamond = X^\diamond + X_2^\diamond$ for $\diamond \in \{n, l, d, p, h\}$. We get from the induction hypothesis a type Y such that $\Gamma \vdash E'_1 : Y$ and all parts of the lemma hold with X for U and Y for V. We get $V^u = (Y^u + X_2^n) \cup (X_2^u + Y^n)$ and $V^\diamond = Y^\diamond + X_2^\diamond$ for $\diamond \in \{n, l, d, p, h\}$ by applying the typing rule Par. Parts 3 to 7 carry over from the induction hypothesis for X, Y. Part 2 follows from $M'_u - V^u = M'_u - (Y^u + X_2^n) \cup (X_2^u + Y^n) = ((M'_u - Y^u) - X_2^n) \cap ((M'_u - Y^n) - X_2^u) \supseteq ((M_u - X^u) - X_2^n) \cap ((M_u - X^n) - X_2^u) = M_u - (X^u + X_2^n) \cup (X_2^u + X^n) = M_u - U^u$. The case of osPar2 follows by symmetry.

For the induction case osSeq, let $\Gamma \vdash A\,E'' : U$ and consider a step

$$(M_u, \{M, A\,E''\}) \rightsquigarrow_P (M'_u, \{M', B\,E''\})$$

inferred from a step $(M_u, \{M, A\}) \rightsquigarrow_P (M'_u, \{M', B\})$. By applying the Inversion Lemma (Lemma 4.7.3), part 8, and the induction hypothesis we get types X, Y, Z such that the typings $A : X$, $B : Y$ and $E'' : Z$ hold in Γ, such that $U = \langle X^u \cup (Z^u - X^l), X^n \cup (Z^n - X^l), X^l + Z^l, X^d + Z^d, X^p \cup (Z^p + X^h), X^h + Z^h \rangle$. By applying Seq to $\Gamma \vdash B : Y$ and $\Gamma \vdash E'' : Z$, we get V the same as U, only with Y substituted for X. Parts 4, 5 and 7 of the lemma carry over from the induction hypothesis for X, Y. Parts 2, 3 follow from applying parts 2 and 4, and parts 3 and 4, respectively, of the induction hypothesis to get $M'_u - V^\diamond = M'_u - (Y^\diamond \cup (Z^\diamond - Y^l)) = (M'_u - Y^\diamond) \cap ((M'_u + Y^l) - Z^\diamond) \supseteq (M_u - X^\diamond) \cap ((M_u + X^l) - Z^\diamond) = M_u - (X^\diamond \cup (Z^\diamond - X^l)) = M_u - U^\diamond$, where $\diamond \in \{u, n\}$. Part 6 follows from a similar argument applying parts 6 and 7 of the induction hypothesis, and that since $A\,E''$ is valid,

$\Sigma E'' = [\,]$, hence $\Sigma A\, E'' = \Sigma A$ and $\Sigma B E'' = \Sigma B$.

For the base case osPop, let $\Gamma \vdash \{N, \mathtt{nop}\} : U$ and consider

$$(M_u, \{M, \{N, \mathtt{nop}\}\}) \rightsquigarrow_P (M_u - N, \{M, \mathtt{nop}\})$$

By the Inversion Lemma 4.7.3, $U = \langle N, N, -N, -N, [\,], [\,] \rangle$ and further $\Gamma \vdash \mathtt{nop} : V$ where $V = \langle [\,], [\,], [\,], [\,], [\,], [\,] \rangle$. Hence, parts 2 to 7 become equalities.

For the inductive case osScp, let $\Gamma \vdash \{N, A\} : U$ and consider a step

$$(M_u, \{M, \{N, A\}\}) \rightsquigarrow_P (M'_u, \{M, \{N', A'\}\})$$

inferred from a step

$$(M_u, \{N, A\}) \rightsquigarrow_P (M'_u, \{N', A'\})$$

By the Inversion Lemma 4.7.3 we have X such that $\Gamma \vdash A : X$ and $U = \langle X^u \cup (N - X^d), X^n \cup (N - X^d), X^d - N, X^d - N, X^p, X^h \rangle$. We get from the induction hypothesis that $\Gamma \vdash A' : Y$, so we can apply the typing rule Scp to get $\Gamma \vdash \{N', A'\} : V$, where $V = \langle Y^u \cup (N' - Y^d), Y^n \cup (N' - Y^d), Y^d - N', Y^d - N', Y^p, Y^h \rangle$. Parts 6 and 7 hold by the induction hypothesis. Parts 4 and 5 become equalities by the induction hypothesis for A, A' part 5. Parts 2 and 3 follow from $M'_u - V^\diamond = M'_u - (Y^\diamond \cup (N' - Y^d)) = (M'_u - Y^\diamond) \cap (M'_u - N' + Y^d) \supseteq (M_u - X^\diamond) \cap (M_u - N + X^d) = M_u - (X^\diamond \cup (N - X^d)) = M_u - U^\diamond$, where $\diamond \in \{u, n\}$.

$\square$

The following Theorem 4.8.3 is a combination of several statements which in combination are often called *soundness* or *safety*. Parts 1, 2 and 3 are similar to the properties often called *preservation*, *progress* and *termination*, respectively. (See for example [60]). Parts 1, 4 and 5 assert that the parts of the types have the meanings given in 4.5.1.

Theorem 4.8.3 (Soundness). *If $\vdash P : \Gamma$, $\Gamma \vdash E : X$, E is valid and $X^u \subseteq M_u$, then the following holds:*

1. If $(M_u, \{M, E\}) \rightsquigarrow_P (M'_u, \{M', E'\})$ and $\Sigma\{M, E\} - M_u \supseteq X^p$, then there is Y such that $\Gamma \vdash E' : Y$, $M'_u \supseteq Y^u$ and $\Sigma\{M', E'\} - M'_u \supseteq Y^p$.

2. If E is not `nop`*, we have $(M_u, \{M, E\}) \rightsquigarrow_P (M'_u, \{M', E'\})$ for some $(M'_u, \{M', E'\})$.*

3. *All $\rightsquigarrow_P$-sequences starting in state $(M_u, \{M, E\})$ are finite.*

4. *If $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', \texttt{nop}\})$, then $X^l \subseteq M'_u - M_u$, $X^d = (M'_u - M') - (M_u - M)$ and $X^h \supseteq (M'_u - M') - (M_u - \Sigma\{M, E\})$.*

5. *If $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', E'\})$ then $-X^n \subseteq M'_u - M_u$ and $X^p \supseteq (M'_u - \Sigma\{M', E'\}) - (M_u - \Sigma\{M, E\})$.*

6. *All states reachable from $(M_u, \{M, E\})$ are safe.*

Proof. Let $\vdash P : \Gamma$, $\Gamma \vdash E : X$ and $X^u \subseteq M_u$.

1. Assume $(M_u, \{M, E\}) \rightsquigarrow_P (M'_u, \{M', E'\})$. E' is typable in Γ by Lemma 4.8.2, part 1. Moreover, $\Gamma(E')^u \subseteq M'_u$ and $\Sigma\{M', E'\} - M'_u \supseteq \Gamma(E')^p$ follows immediately from parts 2 and 6 of the same lemma.

2. By induction on the size of E. Any E can be written in one of the following forms: $\texttt{new}\,x$, $\texttt{del}\,x$, $\texttt{lock}\,N$, $\texttt{free}\,N$, $\texttt{use}\,N$, $\texttt{nop}$, $E_1\,E_2$, $(E_1 + E_2)$, $(E_1 \parallel E_2)$, $\{N, E_1\}$. For each of these forms we check that part 2 of the theorem holds. In case $\texttt{new}\,x$ we have a declaration for x in P by Lemma 4.7.1 so that we can apply osNew. In case $\texttt{del}\,x$ we have $x \in M$ by Remark 4.4.1, and since from the Inversion Lemma $[x] = X^u$, and from assumption $X^u \subseteq M_u$, we have $x \in M_u$, so that we can apply osDel. In the cases $\texttt{lock}\,N$ and $\texttt{use}\,N$ we have $N = X^u \subseteq M_u$ so that we can apply osLock and osUse, respectively. In case $\texttt{free}\,N$ we can apply osFree. The case $\texttt{nop}$ holds vacuously. In case $E_1\,E_2$, if $E_1 = \texttt{nop}$ we can apply rule osNop, otherwise we have from the Inversion Lemma 4.7.3 a type X_1 such that $\Gamma \vdash E_1 : X_1$ and $X_1^u \subseteq X^u$. We can then apply the induction hypothesis for the smaller E_1 and use this step as premiss for an application of osSeq. In case $(E_1 \parallel E_2)$, if $E_1 = E_2 = \texttt{nop}$ we can apply osParEnd. Otherwise we can use the induction hypothesis for at least one of the smaller E_1 or E_2 so that we can apply osPar1 or osPar2. In case $(E_1 + E_2)$ we can always apply osAlt$i$. In case $\{N, E_1\}$, if $E_1 = \texttt{nop}$, we get from the Inversion Lemma that $N = X^u$, so $N \subseteq M_u$ and we can apply osPop. Otherwise we have from the Inversion Lemma that $\Gamma \vdash E_1 : Y$ for some Y and $Y^u \subseteq M_u$, so we can apply the induction hypothesis for the smaller E_1 and use osScp on this step.

3. Assume $\vdash P : \Gamma$, and let $\mathbb{E}_P$ be the set of terms that can be typed in Γ. For every $E \in \mathbb{E}_P$, define $|E|$ in the following recursive way: $|\mathtt{del}\,x| = |\mathtt{lock}\,N| = |\mathtt{free}\,N| = |\mathtt{use}\,N| = |\mathtt{nop}| = 1$, $|\mathtt{new}\,x| = 1 + |A|$ if $x \prec A \in P$, $|(E_1 + E_2)| = 1 + \mathsf{max}(|E_1|, |E_2|)$, $|\{N, E_1\}| = 1 + |E_1|$ and $|E_1 \cdot E_2| = |(E_1 \parallel E_2)| = |E_1| + |E_2|$. By structural induction on the derivation of $\Gamma \vdash E : X$ one easily sees that $|E|$ is well-defined and gives an upper bound to the number of steps in the operational semantics.

4. By induction on the number of steps, using Lemma 4.8.2. The base case $(M_u, \{M, E\}) = (M'_u, \{M', \mathtt{nop}\})$ follows from applying the Inversion Lemma to $\Gamma \vdash \mathtt{nop} : X$. For the induction step, consider

$$(M_u, \{M, E\}) \rightsquigarrow_P (M''_u, \{M'', E'\}) \rightsquigarrow^*_P (M'_u, \{M', \mathtt{nop}\})$$

and assume $\Gamma(E) = X$. From Lemma 4.8.2, we get $\Gamma(E') = Y$ such that we can apply the induction hypothesis and get $Y^l \subseteq M'_u - M''_u$, $Y^d = (M'_u - M') - (M''_u - M'')$ and $Y^h \supseteq (M'_u - M') - (M''_u - \Sigma\{M'', E'\})$. By Lemma 4.8.2, part 4, we have $X^l + M_u \subseteq Y^l + M''_u$, so we get $X^l \subseteq Y^l + M''_u - M_u \subseteq M'_u - M_u$. By part 5 of the same lemma we have $Y^d + M''_u - M'' = X^d + M_u - M$, so we get $(M'_u - M') - (M_u - M) = X^d$. Finally, by part 7 of Lemma 4.8.2 we have $Y^h + M''_u - \Sigma\{M'', E'\} \subseteq X^h + M_u - \Sigma\{M, E\}$, so we get $(M'_u - M') - (M_u - \Sigma\{M, E\}) \subseteq X^h$.

5. By induction on the number of steps, using Lemma 4.8.2. The base case (zero steps) is trivial. For the induction step, consider

$$(M_u, \{M, E\}) \rightsquigarrow_P (M''_u, \{M'', E''\}) \rightsquigarrow^*_P (M'_u, \{M', E'\})$$

and assume $\Gamma(E) = X$. From Lemma 4.8.2 we get $\Gamma(E'') = Y$ such that we can apply the induction hypothesis and get $-Y^n \subseteq M'_u - M''_u$ and $(M'_u - \Sigma\{M', E'\}) - (M''_u - \Sigma\{M'', E''\}) \subseteq Y^p$ By Lemma 4.8.2, part 3, we have $M_u - X^n \subseteq M''_u - Y^n$ so we get $-X^n \subseteq M'_u - M_u$. By part 6 of the same lemma, we have $Y^p + M''_u - \Sigma\{M'', E''\} \subseteq X^p + M_u - \Sigma\{M, E\}$, so we get $(M'_u - \Sigma\{M', E'\}) - (M_u - \Sigma\{M, E\}) \subseteq X^p$.

6. It is enough to show that $(M_u, \{M, E\})$ is safe, as part 1 together with Lemma 4.8.1 guarantee that the theorem can be applied in all reachable states. Recall the definition of unsafe states in Definition 4.4.2. Assume there exists a bag N and a transition $(M_u + N, \{M, E\}) \rightsquigarrow_P$

$(M'_u + N, \{M', E'\})$, and use induction on the derivation of this transition to show that there also exists a transition $(M_u, \{M, E\}) \rightsquigarrow_P (M'_u, \{M', E'\})$:

The base cases osFree, osAlt1, osAlt2, osNop and osParEnd are easy, as there are no conditions or restrictions on the transition. The base case osNew holds since P does not change. In the base cases osDel, osLock, osUse and osPop we need to use that $X^u \subseteq M_u$ and that the smallest safe size of M_u, as seen from the operational semantics, in each case is exactly X^u, as seen from the type rules. For the induction cases osScp, osPar1, osPar2 and osSeq, assume that the premiss from which the transition is inferred is $(M_u + N, \{N', F\}) \rightsquigarrow_P (M'_u + N, \{N'', F'\})$, where the Inversion Lemma 4.7.3 implies that $\Gamma \vdash F : Y$ for some Y, where $Y^u \subseteq X^u$, such that $Y^u \subseteq M_u$. We can therefore apply the induction hypothesis to this premiss, to get the existence of $(M_u, \{N', F\}) \rightsquigarrow_P (M'_u, \{N'', F'\})$ from which we can use the same rule to infer $(M_u, \{M, E\}) \rightsquigarrow_P (M'_u, \{M', E'\})$. □

Finally, we summarize the properties of the type system for well-typed programs, as defined in Definition 4.5.1 on page 124. The reader is referred to the paragraph following Definition 4.5.1 for an explanation of the four bags required to be empty, to Section 4.4.1 and Definition 4.4.2 for an explanation of safe states, and to Section 4.4.2 for an explanation of why it is important that $M'_u \subseteq \Sigma\{M', E'\}$.

Corollary 4.8.4. *If* $\vdash P : \Gamma$ *and* $\Gamma \vdash \texttt{new}\, x : X$, *where x is the last component declared in P and* $X^d = X^u = X^p = X^h = [\,]$, *then*

- *All maximal transition sequences starting with* $([\,], \{[\,], \texttt{new}\, x\})$ *end with* $(M, \{M, \texttt{nop}\})$ *for some bag M.*

- *All states* $(M'_u, \{M', E'\})$ *reachable from* $([\,], \{[\,], \texttt{new}\, x\})$ *are safe, and such that* $M'_u \subseteq \Sigma\{M', E'\}$.

The following theorem states that the types are *sharp*. Informally, this means, they are as small as they can be, while still guaranteeing safety of execution. The part X^d is not included as it is already stated in Theorem 4.8.3 to be exact. The property is formulated differently for the part X^u because of its nature — the other parts contain information about how some of the bags or the difference between them change, while X^u only states the minimum safe size of the bag M_u.

Theorem 4.8.5 (Sharpness). *Assume some program P, bags M and M_u and valid expression E such that $\vdash P\!:\!\Gamma$ and $\Gamma \vdash E\!:\!X$ and $M_u \subseteq \Sigma\{M, E\}$*

1. *If $M_u \not\supseteq X^u$, then an unsafe state is reachable from $(M_u, \{M, E\})$.*
2. *If $M_u \supseteq X^u$:*

   n For every $y \in \mathbb{C}$ there exists a state $(M'_u, \{M', E'\})$ such that $(M_u, \{M, E\}) \rightsquigarrow^*_P (M'_u, \{M', E'\})$ and $(M'_u - M_u)(y) = -X^n(y)$.

   l For every $y \in \mathbb{C}$ there exists a terminal state $(M'_u, \{M', \texttt{nop}\})$ such that $(M_u, \{M, E\}) \rightsquigarrow^*_P (M'_u, \{M', \texttt{nop}\})$ and $(M'_u - M_u)(y) = X^l(y)$.

   p For every $y \in \mathbb{C}$ there exists a state $(M'_u, \{M', E'\})$ such that $(M_u, \{M, E\}) \rightsquigarrow^*_P (M'_u, \{M', E'\})$ and $(M'_u - \Sigma\{M', E'\}) - (M_u - \Sigma\{M, E\})(y) = X^p(y)$.

   h For every $y \in \mathbb{C}$ there exists a terminal state $(M'_u, \{M', \texttt{nop}\})$ such that $(M_u, \{M, E\}) \rightsquigarrow^*_P (M'_u, \{M', \texttt{nop}\})$ and $(M'_u - M') - (M_u - \Sigma\{M, E\})(y) = X^h(y)$.

Proof. By primary induction on the length of P and secondary induction on the derivation of $\Gamma \vdash E\!:\!X$. If the length of P is zero the result is trivial. Otherwise, $\vdash P\!:\!\Gamma$ has been proved by the following application of Prog:

$$\frac{\Gamma' \vdash A\!:\!Y,\ \vdash P'\!:\!\Gamma',\ x \notin \mathsf{dom}(\Gamma')}{\vdash P', x \multimap A\!:\!\Gamma' \cup \{x \mapsto Y\}}$$

Assume the result has been proved for all programs with fewer declarations than P. We now prove that X is sharp whenever $\Gamma \vdash E : X$ by induction on the derivation of the latter.

For X^u note that $M_u \not\supseteq X^u$ implies that there is $y \in \mathbb{C}$ such that $M_u(y) < X^u(y)$ and $X^u(y) \geq 1$. Let $y \in \mathbb{C}$. Note that, for $\diamond \in \{n, p\}$, if some $X^\diamond(y) = 0$, one can take $(M_u, \{M', E'\}) = (M_u, \{M, E\})$ to get the desired result. With this in mind the base cases Axm, Lock, Free, Use and Del are easy.

The base case New is more interesting since it uses the primary induction hypothesis. Assume $x \multimap A \in P$ and $\Gamma \vdash E\!:\!X$ is inferred by the following application of the rule New:

$$\frac{\Gamma(x) = Y}{\Gamma \vdash \texttt{new}\,x\!:\!\langle Y^u, Y^n, Y^l + x, Y^d, Y^p, Y^h\rangle}$$

If x is not the last variable declared in P, then the sharpness of $X = \langle Y^u, Y^n, Y^l + x, Y^d, Y^p, Y^h \rangle$ follows from the primary induction hypothesis (in combination with Uniqueness). Otherwise, we have that P is $P', x \multimap A$ as in the application of Prog above, with $\vdash P' : \Gamma'$, $\Gamma' \vdash A : Y$. Concerning part 1, note that $X^u = Y^u$ and from the Basics Lemma $x \notin Y^u$, such that if $M_u \not\supseteq X^u$ then also $M_u + x \not\supseteq X^u$. We can therefore use the primary induction hypothesis for A, Y on the state obtained by one application of the rule osNew. For the second part, and for any $y \in \mathbb{C}$ different from x we can use the primary induction hypothesis for A, Y and prefix the sequences obtained by a step using the rule osNew as we have $M(y) = (M + x)(y)$, $M_u(y) = (M_u + x)(y)$ and $Y^l(y) = (Y^l + x)(y)$. So for x, we only need to note that $Y^\diamond(x) = 0$ for $\diamond \in \{n, p, h\}$ and take $(M'_u, \{M', E'\}) = (M_u, \{M, E\})$ to get $-X^\diamond(x) = Y^\diamond(x) = 0 = (M' - M)(x) = (M'_u - M_u)(x)$. Finally $Y^l(x) = 1$ and we can take $(M'_u, \{M', E'\}) = (M_u + x, \{M + x, A\})$.

In the induction case Alt the induction hypothesis can be applied to the premises $\Gamma \vdash E_i : X_i$ $(i = 1, 2)$. For part 1, note that if $M_u \not\supseteq X^u$, then there is an $i \in \{1, 2\}$ such that $M_u \not\supseteq X_i^u$. The induction hypothesis can then be used after an application of osAlt$i$. For the second part, for each y and $\diamond \in \{n, l, d, p, h\}$, if $X^\diamond(y) = X_i^\diamond(y)$, then one uses the induction hypothesis for E_i (i may vary with $y, \diamond$).

In the case of the rule Seq we also apply the induction hypothesis to the premises $\Gamma \vdash E_i : X_i$ $(i = 1, 2)$. For part 1, note first that if $M_u \not\supseteq X^u$, then either $M_u \not\supseteq X_1^u$ or $M_u \not\supseteq X_1^u - X_1^l$. In the first case, we can apply the induction hypothesis to E_1, and postfix all expressions with E_2 in the obtained sequence. In the second case, where for some y, $M_u(y) < X_2^u(y) - X_1^l(y)$, we take the sequence $(M_u, \{M, E_1\}) \rightsquigarrow_P^* (M'_u, \{M', \texttt{nop}\})$ with $M'_u(y) = M_u(y) - X_1^l(y) < X_2^u(y)$ such that $M'_u \not\supseteq X_2^u$, postfix its expressions with E_2, and then proceed with an instance of osNop and the sequence $(M'_u, \{M', E_2\}) \rightsquigarrow_P^* (M''_u, \{M'', E'_2\})$ with $(M''_u, \{M'', E'_2\})$ being an unsafe state. Concerning X^l and X^h we can concatenate the two sequences obtained from the induction hypothesis. For X^n and component y we distinguish between $X^n(y) = X_1^n(y)$ and $X^n(y) = X_2^n(y) - X_1^l(y)$. In the first case we take the sequence for $(M_u, \{M, E_1\})$ and postfix all expressions with E_2 to obtain a sequence for $(M_u, \{M, E_1\, E_2\})$ with the desired property. In the second case we take the sequence $(M_u, \{M, E_1\}) \rightsquigarrow_P^* (M'_u, \{M', \texttt{nop}\})$ with $(M'_u - M_u)(y) = X_1^l(y)$, postfix its expressions with E_2, and then proceed with an instance of osNop and then the sequence

$$(M'_u, \{M', E_2\}) \rightsquigarrow^*_P (M''_u, \{M'', E'_2\})$$

with $(M''_u - M'_u)(y) = -X^n_2(y)$. The total sequence

$$(M_u, \{M, E_1\, E_2\}) \rightsquigarrow^*_P (M'_u, \{M', E_2\}) \rightsquigarrow^*_P (M''_u, \{M'', E'_2\})$$

enjoys $(M''_u - M_u)(y) = X^l_1(y) - X^n_2(y) = -X^n(y)$. The remaining case X^p, can be dealt with in a way very similar to $X^n(y)$. Note that $\Sigma E_2 = [\,]$ because E is valid.

For the induction case of the rule Par assume $E = (E_1 \parallel E_2)$ so the induction hypothesis can be applied to the premises $\Gamma \vdash E_i : X_i$ $(i = 1, 2)$. For part 1, where $X^u = (X^u_1 + X^n_2) \cup (X^u_2 + X^n_1)$, note that if $M_u \not\supseteq X^u$, then either $M_u \not\supseteq X^u_1 + X^n_2$ or $M_u \not\supseteq X^u_2 + X^n_1$. In the former case there is y such that $M_u(y) < X^u_1(y) + X^n_2(y)$. We can use the induction hypothesis for X^n_2 and y and apply osPar2 to each step in the sequence obtained. Assume $(M'_u, \{M', (E_1 \parallel E'_2)\})$ is the last state in this sequence, such that $M'_u(y) = M_u(y) - X^n_2(y) < X^u_2(y)$. So we have $M'_u \not\supseteq X^u_1(y)$ and we can apply the induction hypothesis to get an unsafe state reachable from $(M'_u, \{M', E_1\})$. Apply osPar1 to each step in the sequence to the latter unsafe state, and we get the reachable unsafe state needed for part 1. The other case follows by symmetry.

For part 2 we apply the induction hypothesis to both premises and concatenate both sequences using the inductive rules osPar1 and osPar2. By additivity this gives the desired results. (Any interleaving of the two sequences would amount to the same.)

For the induction case Scope, assume $E = \{M_s, E_s\}$ so the induction hypothesis can be applied to the premiss $\Gamma \vdash E_s : X_s$. For X^n we do a case distinction on whether $X^n(y) = X^n_s(y)$ or $X^n(y) = M_s(y) - X^d_s(y)$. In the first case we can apply the induction hypothesis to E_s and M_s to get the sequence $(M_u, \{M_s, E_s\}) \rightsquigarrow^*_P (M'_u, \{M'_s, E'_s\})$, where $(M'_u - M_u)(y) = -X^n(y)$. Applying osScp to every step, we get a sequence $(M_u, \{M, \{M_s, E_s\}\}) \rightsquigarrow^*_P (M'_u, \{M, \{M'_s, E'_s\}\})$. In the second case, where $X^n(y) = M_s(y) - X^d_s(y)$, we have from Theorem 4.8.3, that for any sequence $(M_u, \{M, \{M_s, E_s\}\}) \rightsquigarrow^*_P (M'_u, \{M, \{M'_s, \texttt{nop}\}\})$ it is true that $X^d_s = (M'_u - M'_s) - (M_u - M_s)$. Postfix this sequence with a transition osPop: $(M'_u, \{M, \{M'_s, \texttt{nop}\}\}) \rightsquigarrow_P (M'_u - M'_s, \{M, \texttt{nop}\})$. We can now reorder $M'_u(y) = X^d_s(y) + M'_s(y) - M_s(y) + M_u(y)$ to get $(M'_u - M'_s)(y) - M_u(y) = X^d_s(y) - M_s(y) = X^n(y)$.

The case for X^u follows from a similar argument as for X^n. The case for $X^l(y) = X_s^d(y) - M_s(y)$ is similar to the second case for X^n. We have from Theorem 4.8.3, that for any sequence $(M_u, \{M, \{M_s, E_s\}\}) \rightsquigarrow_P^*$ $(M_u', \{M, \{M_s', \texttt{nop}\}\})$ it is true that $X_s^d = (M_u' - M_s') - (M_u - M_s)$. Postfix this sequence with a transition osPop: $(M_u', \{M, \{M_s', \texttt{nop}\}\}) \rightsquigarrow_P$ $(M_u' - M_s', \{M, \texttt{nop}\})$. We can now reorder $M_u'(y) = X_s^d(y) + M_s'(y) - M_s(y) + M_u(y)$ to get $(M_u' - M_s')(y) - M_u(y) = X_s^d(y) - M_s(y) = X^l(y)$. For X^d, X^p and X^h apply the induction hypothesis to E_s and M_s and apply osScp to each step in the obtained sequence.

□

4.9 Related Work and Conclusion

There is a large amount of work related to similar problems. Most approaches differ from this chapter by using super-polynomial algorithms, by assuming more on the runtime scheduling of parallel executions, or by treating only memory consumption. For the functional languages, see e.g. [20, 32, 46, 70]. Popea and Chin in [62] also discuss usage in a related way. Their algorithm depends on solving constraints in Presburger arithmetic, which in the worst case uses doubly exponential time. Igarashi & Kobayashi in [40], analyze the *resource usage problem* for an extension of simply typed lambda calculus including resource usage. The algorithm extracts the set of possible traces of usage from the program, and then decides whether all these traces are allowed by the specification. This latter problem is still computationally hard to solve and undecidable in the worst case. Parallel composition is not considered. For the imperative paradigm, which is closer to the system described here, e.g. [8, 17, 33] treat memory usage. The problem of component usage in a parallel setting is related to prevention of deadlocks and race conditions. Boyapati et al. describe in [7] an explicitly typed system for verifying there are no deadlocks or race conditions in Java programs. In addition to the higher level of detail, the main difference from the system described in this chapter is the assumptions on the scheduling of parallel executions, namely the ability of a thread to wait until another thread frees/releases a lock. This scheduling has of course a cost in terms of added runtime and of complexity of the implementation.

We have defined a component language with a small-step operational semantics and a type system. The type system combined with the system in [3] or the system in [67] guarantees that the execution of a well-typed program will terminate and cannot reach an unsafe state. The language described in this chapter is an extension of the language first described in [68], and uses the results from [68, 3]. The properties proved in the current chapter are new, though, and in some ways orthogonal to those shown in [68, 3]. The language we introduced is inspired by CCS [55], with the atomic actions interpreted as component instantiation, deallocation and usage. The basic operators are sequential, alternative and parallel composition and a scope operator. The operational semantics is SOS-style [61], with the approach to soundness similar in spirit to [71]. We have presented a type system for this language which predicts sharp bounds of the number of instances of components necessary for safe execution. The type inference algorithm has quadratic runtime.

Bibliography

[1] ISO 8879. Information processing — text and office systems — standard generalized markup language (SGML), October 1986.

[2] Valentin M. Antimirov. Rewriting regular inequalities (extended abstract). In Horst Reichel, editor, *FCT*, volume 965 of *Lecture Notes in Computer Science*, pages 116–125. Springer, 1995.

[3] Marc Bezem, Dag Hovland, and Hoang Truong. A type system for counting instances of software components. Technical Report 363, Department of Informatics, The University of Bergen, P.O. Box 7800, N-5020 Bergen, Norway, October 2007.

[4] Marc Bezem, Jan Willem Klop, and Roel de Vrijer, editors. *Term Rewriting Systems*. Cambridge University Press, 2003.

[5] Marc Bezem and Hoang Truong. A type system for the safe instantiation of components. *Electronic Notes in Theoretical Computer Science*, 97:197–217, 2004.

[6] Ronald Book, Shimon Even, Sheila Greibach, and Gene Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, c-20(2):149–153, 1971.

[7] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[8] Víctor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, June 2006.

[9] Anne Brüggemann-Klein. Regular expressions into finite automata. In Imre Simon, editor, *LATIN*, volume 583 of *Lecture Notes in Computer Science*, pages 87–98. Springer, 1992.

[10] Anne Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[11] Anne Brüggemann-Klein. Unambiguity of extended regular expressions in SGML document grammars. In Thomas Lengauer, editor, *ESA*, volume 726 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 1993.

[12] Anne Brüggemann-Klein and Derick Wood. One-unambiguous regular languages. *Information and Computation*, 140(2):229–253, 1998.

[13] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[14] Janusz A. Brzozowski. Roots of star events. *J. ACM*, 14(3):466–477, 1967.

[15] Anne Brüggemann-Klein. Compiler-construction tools and techniques for SGML parsers: Difficulties and solutions, May 1994.

[16] Haiming Chen and Lei Chen. Inclusion test algorithms for one-unambiguous regular expressions. In John S. Fitzgerald, Anne Elisabeth Haxthausen, and Hüsnü Yenigün, editors, *ICTAC*, volume 5160 of *LNCS*, pages 96–110. Springer, 2008.

[17] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin C. Rinard. Memory usage verification for OO programs. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 70–86. Springer, 2005.

[18] Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Efficient asymmetric inclusion between regular expression types. In Ronald Fagin, editor, *ICDT*, volume 361 of *ACM International Conference Proceeding Series*, pages 174–182. ACM, 2009.

[19] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

[20] Karl Crary and Stephanie Weirich. Resource bound certification. In *POPL '00: Proceedings of the 27th ACM SIGPLAN–SIGACT symposium on Principles of programming languages*, pages 184–198, New York, NY, USA, 2000. ACM Press.

[21] Thompson et al. XML Schema part 1: Structures second edition, W3C recommendation, 2004.

[22] Wouter Gelade. Succinctness of regular expressions with interleaving, intersection and counting. In Edward Ochmanski and Jerzy Tyszkiewicz, editors, *MFCS*, volume 5162 of *Lecture Notes in Computer Science*, pages 363–374. Springer, 2008.

[23] Wouter Gelade, Marc Gyssens, and Wim Martens. Regular expressions with counting: Weak versus strong determinism. `http://lrb.cs.uni-dortmund.de/~martens/data/mfcs09-appendix.pdf`. In Rastislav Královic and Damian Niwinski, editors, *MFCS*, volume 5734 of *Lecture Notes in Computer Science*, pages 369–381. Springer, 2009.

[24] Wouter Gelade, Wim Martens, and Frank Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. In Thomas Schwentick and Dan Suciu, editors, *Proceedings of ICDT*, volume 4353 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2007.

[25] Wouter Gelade, Wim Martens, and Frank Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. *SIAM J. Comput.*, 38(5):2021–2043, 2009.

[26] Giorgio Ghelli, Dario Colazzo, and Carlo Sartiani. Efficient inclusion for a class of xml types with interleaving and counting. In Marcelo Arenas and Michael I. Schwartzbach, editors, *DBPL*, volume 4797 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2007.

[27] Giorgio Ghelli, Dario Colazzo, and Carlo Sartiani. Linear time membership for a class of XML types with interleaving and counting. In *PLAN-X*, 2008.

[28] Giorgio Ghelli, Dario Colazzo, and Carlo Sartiani. Linear time membership in a class of regular expressions with interleaving and counting. In James G. Shanahan, Sihem Amer-Yahia, Ioana Manolescu, Yi Zhang, David A. Evans, Aleksander Kolcz, Key-Sun Choi, and Abdur Chowdhury, editors, *CIKM*, pages 389–398. ACM, 2008.

[29] V M Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961.

[30] GNU. *GNU grep manual.*

[31] Sheila A. Greibach. Remarks on blind and partially blind one-way multicounter machines. *Theor. Comput. Sci.*, 7:311–324, 1978.

[32] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 185–197, New York, NY, USA, 2003. ACM.

[33] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis (for an object-oriented language). In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems*, volume 3924 of *LNCS*, pages 22–37. Springer, 2006.

[34] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[35] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[36] Dag Hovland. A type system for usage of software components. In Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro, editors, *TYPES*, volume 5497 of *Lecture Notes in Computer Science*, pages 186–202. Springer, 2008.

[37] Dag Hovland. Regular expressions with numerical constraints and automata with counters. In Martin Leucker and Carroll Morgan, editors, *ICTAC*, volume 5684 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2009.

[38] Dag Hovland. The inclusion problem for regular expressions. In Adrian Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *LATA*, volume 6031 of *Lecture Notes in Computer Science*, pages 309–320. Springer, 2010.

[39] IEEE. The open group base specifications issue 6, 2004.

[40] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. *ACM Trans. Program. Lang. Syst.*, 27(2):264–313, 2005.

[41] Pekka Kilpeläinen. Inclusion of unambiguous ♯REs is NP-hard. Technical report, University of Kuopio, May 2004.

[42] Pekka Kilpeläinen and Rauno Tuhkanen. Regular expressions with numerical occurrence indicators - preliminary results. In Pekka Kilpeläinen and Niina Päivinen, editors, *SPLST*, pages 163–173. University of Kuopio, Department of Computer Science, 2003.

[43] Pekka Kilpeläinen and Rauno Tuhkanen. Towards efficient implementation of XML schema content models. In Ethan V. Munson and Jean-Yves Vion-Dury, editors, *ACM Symposium on Document Engineering*, pages 239–241. ACM, 2004.

[44] Pekka Kilpeläinen and Rauno Tuhkanen. One-unambiguity of regular expressions with numeric occurrence indicators. *Information and Computation*, 205(6):890–916, 2007.

[45] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, Third Edition*. Addison-Wesley, 1997.

[46] Naoki Kobayashi, Kohei Suenaga, and Lucian Wischik. Resource usage analysis for the π-calculus. *Logical Methods in Computer Science*, 2(3), 2006.

[47] Christoph Koch and Stefanie Scherzinger. Attribute grammars for scalable query processing on XML streams. *VLDB J.*, 16(3):317–342, 2007.

[48] Federico Mancini, Dag Hovland, and Khalid A. Mughal. Investigating the limitations of Java annotations for input validation. In *ARES*, pages 513–518. IEEE Computer Society, 2010.

[49] Federico Mancini, Dag Hovland, and Khalid A. Mughal. The SHIP validator: An annotation-based content-validation framework for Java applications. In *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*, pages 122 – 128, 9-15 2010.

[50] Wim Martens, Frank Neven, and Thomas Schwentick. Complexity of decision problems for simple regular expressions. In Jirí Fiala, Václav Koubek, and Jan Kratochvíl, editors, *MFCS*, volume 3153 of *Lecture Notes in Computer Science*, pages 889–900. Springer, 2004.

[51] Alain J. Mayer and Larry J. Stockmeyer. Word problems-this time with interleaving. *Inf. Comput.*, 115(2):293–311, 1994.

[52] Douglas M. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, pages 79–87. Scientific Affairs Division, NATO, October 1968.

[53] Robert McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, 9:39–47, 1960.

[54] Albert R. Meyer and Larry J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of FOCS*, pages 125–129. IEEE, 1972.

[55] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[56] Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.

[57] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

[58] William F. Ogden, William E. Riddle, and William C. Rounds. Complexity of expressions allowing concurrency. In *POPL*, pages 185–194, 1978.

[59] The Open Group. *The Open Group Base Specifications Issue 6, IEEE Std 1003.1*, 2 edition, 1997.

[60] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[61] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, July-December 2004.

[62] Corneliu Popeea and Wei-Ngan Chin. A type system for resource protocol verification and its correctness proof. In Nevin Heintze and Peter Sestoft, editors, *PEPM*, pages 135–146. ACM, 2004.

[63] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, 1966.

[64] C. M. Sperberg-McQueen. Notes on finite state automata with counters, 2004.

[65] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 2000.

[66] Clemens Szyperski. *Component Software—Beyond Object–Oriented Programming*. Addison–Wesley / ACM Press, 2nd edition, 2002.

[67] Hoang Truong. Guaranteeing resource bounds for component software. In Martin Steffen and Gianluigi Zavattaro, editors, *FMOODS*, volume 3535 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2005.

[68] Hoang Truong and Marc Bezem. Finding resource bounds in the presence of explicit deallocation. In Dang Van Hung and Martin Wirsing, editors, *Proceedings ICTAC*, volume 3722 of *Lecture Notes in Computer Science*, pages 227–241. Springer, 2005.

[69] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, part 2*, pages 115–225, 1968.

[70] Leena Unnikrishnan, Scott D. Stoller, and Yanhong A. Liu. Optimized live heap bound analysis. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 70–85, London, UK, 2003. Springer-Verlag.

[71] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.