TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
17/2020

Operational Semantics of Weak Sequential Composition

HENDRIK MAARAND

TAL TECH PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

The dissertation was accepted for the defence of the degree of Doctor of Philosophy (Informatics) on 29 March 2020

Supervisor: Prof. Tarmo Uustalu
Tallinn University of Technology / Reykjavik University

Opponents: Dr. Brijesh Dongol
University of Surrey

Prof. Peter Thiemann
Albert-Ludwigs-Universität Freiburg

Defence of the thesis: 26 June 2020, Tallinn

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology, has not been submitted for any academic degree elsewhere.

Hendrik Maarand _____
signature

European Union
European Regional
Development Fund

Investing
in your future

ISSN 2585-6898 (publication)
ISBN 978-9949-83-560-7 (publication)
ISSN 2585-6901 (PDF)
ISBN 978-9949-83-561-4 (PDF)
Printed by Auratrükk

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
17/2020

Nõrga jadakompositsiooni operatsioonsemantika

HENDRIK MAARAND

TAL TECH
KIRJASTUS

Contents

List of Publications

I H. Maarand and T. Uustalu. Reordering derivatives of trace closures of regular languages. In W. J. Fokkink and R. van Glabbeek, editors, *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*, volume 140 of *LIPIcs*, pages 40:1–40:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019

II H. Maarand and T. Uustalu. Certified Foata normalization for generalized traces. In A. Dutle, C. A. Muñoz, and A. Narkawicz, editors, *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, volume 10811 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2018

III H. Maarand and T. Uustalu. Certified normalization of generalized traces. *Innovations in Systems and Software Engineering*, 15(3-4):253–265, 2019
(This is the journal version of Publication II.)

IV H. Maarand and T. Uustalu. Generating representative executions [extended abstract]. In V. T. Vasconcelos and P. Haller, editors, *Proceedings of the Tenth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2017, Uppsala, Sweden, 29th April 2017*, volume 246 of *EPTCS*, pages 39–48. Open Publishing Association, 2017

V H. Maarand and T. Uustalu. Operational semantics with semicommutations. *Accepted for publication in J. Log. Algebr. Methods Program.*
(This is the journal version of Publication IV and the material has been significantly elaborated here.)

Author's Contributions to the Publications

In Publication I, I proposed the idea of developing reordering derivatives with the goal of developing an operational semantics for relaxed memory out of it. Many of the results were obtained together with my supervisor. I wrote the first draft of the manuscript and also presented the results at the conference.

In Publications (II–III), I proposed the idea of using normal forms of generalised traces to allow a more precise description of some relaxed memory models which in turn led to this work. I carried out the formalisation and wrote the first drafts of the manuscripts. I presented the results of Publication II at the conference.

In Publications (IV–V), I proposed the idea of using the same independence relation both for generating the program executions and also for discarding those executions that are not in normal form. I implemented the prototypes and wrote the first drafts of the manuscripts. I presented the results of Publication IV at the workshop.

1 Introduction

This dissertation considers the execution of concurrent programs in a manner that is not sequentially consistent. Intuitively, this means that we consider program executions that are not justifiable by simply interleaving the program-order instruction sequences of the individual threads. The sequentially consistent model is intuitive, but real world CPUs and compilers do not adhere to it as it forbids many common optimisations. Instead we have consistency models that are weaker than sequential consistency in the sense that they allow more program executions. These weaker models, however, are less intuitive than sequential consistency and may lead to unexpected results at runtime.

A mechanism that is commonly used in programming languages to introduce ordering constraints between instructions is sequential composition. If we take p and q to be programs, then their sequential composition $p;q$ expresses that the program q is to be executed after the program p has been executed. In this dissertation we develop an operational semantics that allows to weaken (or relax) some of the ordering constraints introduced by sequential composition.

Context

A memory consistency model (or, more generally, a memory model) describes the meaning of memory operations such as reads and writes in a shared memory system. For example, it should specify the set of values that a processor is allowed to read when executing an instruction that reads the content of memory location x. Intuitively, this set of values should be determined based on all the write instructions to memory location x that have occurred so far.

It is very intuitive to consider the shared memory to be a mapping from locations to values that is updated accordingly whenever a write to a location x occurs. As an example we now consider something different. We say that the processors agree on variable x if the last value written to x has also been at some point written to location x by all of the processors and no other values have been written to location x in between these writes. We say that the current value of location x is the last value that the processors agreed upon (and we assume that the processors agree on the initial values of the memory locations). For example, if during the execution of a program the processors never agree on any location, then every memory location still holds the initial value also in the final state. If we are not aware of the fact that the shared memory of some machine behaves as described above, then we might be very surprised when we begin to execute programs on that machine. We can see that, with this unconventional memory behaviour, we may have to write our programs differently and we also must reason about the programs differently. In other words, we have to take the behaviour of the memory system into account in both of these tasks.

Perhaps the simplest memory model is Sequential Consistency (SC) which was defined by Lamport [44] for a system consisting of processors and memory modules where the processors communicate with each other only by sending *fetch* and *store* requests to the memory modules. He defined a system to be sequentially consistent if the following holds: *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.* In the same paper Lamport also notes that the requirements for sequential consistency rule out some techniques that can be used to speed up individual processors and thus, for some applications, achieving sequential consistency might not be worth the price of slowing down the processors.

Modern CPUs and compilers indeed include optimisations that make them weaker than sequential consistency. It is common, however, for weaker consistency models to still satisfy the requirement that each processor (or thread) is *individually sequentially consistent* by which we mean that a single processor executing in isolation is sequentially consistent. It is also a very reasonable requirement for compiler optimisations that, for single-threaded programs, the optimisations do not introduce any new behaviours (in which case we also say that the optimisation is individually sequentially consistent). As Lamport notes in [44], the fact that each processor is individually sequentially consistent is not sufficient for the whole system to be sequentially consistent. In other words, the effects of optimisations that are individually sequentially consistent might become visible in a concurrent setting.

We now continue with a small example to explain how optimisations can violate sequential consistency. The following concurrent program represents the *message-passing* pattern.

$$x := 41; y := 1 \parallel r_1 := y; r_2 := x$$

Here the first thread stores to x the result of some computation (which happens to be 41) and then sets y to 1. The intended meaning of the second instruction is that the variable y having value 1 indicates that the first thread has finished its computation (and the result is now available in variable x). The second thread first reads the variable y and then the variable x. If we follow the requirements of sequential consistency, then no execution of this program, starting from an initial state where everything is set to 0, can result in a final state where $r_1 = 1$ and $r_2 = 0$. In other words, when the variable y holds the value 1 (indicating that the first thread has finished), then the variable x must hold the value 41.

We can observe, however, that in both threads reordering the two instructions is individually sequentially consistent, i.e., when executing a thread in isolation, we cannot distinguish (based on the final state) whether we executed the instructions in the given order or in the reverse order. This is so because the instructions in a single thread use different variables. After an optimisation like this we could end up with the following program where the instructions in the second thread have been reordered.

$$x := 41; y := 1 \parallel r_2 := x; r_1 := y$$

This optimised program has the interleaving

$$r_2 := x; x := 41; y := 1; r_1 := y$$

which results in a final state where $r_1 = 1$ and $r_2 = 0$. Thus we can see that applying an optimisation that is individually sequentially consistent to a concurrent program (either in advance or at runtime) might introduce behaviour that is not possible under sequential consistency. In other words, this optimisation is not valid under sequential consistency.

Systems with a memory consistency model that allows more behaviours than sequential consistency are often said to have a *weak* or *relaxed* memory model. The above example demonstrates that under a relaxed memory model a program can have additional final states compared to those that it has under sequential consistency. As a consequence, when reasoning about programs we have to be precise about the particular memory model as the same program might satisfy certain properties on some memory models but not on others.

Problem Statement

In the example above we interpreted the message passing program in two different ways. First, assuming sequential consistency, we argued that the program does not allow a cer-

tain undesirable final state. Then we argued that, if we would first apply a seemingly harmless optimisation to the program, then the final state in question is allowed. In other words, the optimised program can reach the undesirable final state. The optimisation necessary for this just reorders a single pair of instructions. This raises the question: how to describe or specify such relaxed memory models?

A successful approach to describing memory models has been the use of axiomatic models or axiomatic style descriptions. An example of this is the generic framework for weak memory models developed by Alglave in her PhD thesis [6]. A characteristic feature of the axiomatic approach is the predicate on (complete) candidate executions which defines those (candidate) executions that are allowed by the memory model. In other words, the memory model definition itself does not specify how to construct an execution, it only specifies when an execution is allowed by the memory model.

Our goal in this dissertation is somewhat opposite: we pursue a framework for defining relaxed memory models via *small-step operational semantics*. Thus our goal is to describe how to execute a program in a relaxed manner. In other words, given a configuration consisting of a program and a machine state, the framework should specify what are the possible next steps in the execution that are allowed by the memory model. The executions allowed by the memory model are then precisely those executions that can be constructed in such a step-by-step manner and which take us from the initial state to a final state.

Contributions

We develop an operational semantics that has reordering capabilities (such as those necessary for the example optimisation above) built in. This means that to obtain the undesirable final state for the example program we do not have to consider a set of optimised versions of the given program as we did above. Instead, we can set up our operational semantics so that it can produce an execution that goes from the initial state to this undesirable final state. Altogether, we make the following contributions.

1. **Reordering derivatives**: We use regular expressions over an alphabet of instructions as the syntax with which we describe programs. For regular expressions, the idea corresponding to operational semantics is the notion of (syntactic) *derivatives*. By this we mean that derivatives describe how to construct (in a letter-by-letter manner) a word in the language of a regular expression.

   Thus as a first step towards our goal we investigate derivatives of regular expressions in the presence of an independence relation on the alphabet. The independence relation (as in Mazurkiewicz traces [52]) specifies the pairs of letters that commute and we use this as the mechanism to describe the optimisations that we include in the semantics.

   As a specification of what the (syntactic) reordering derivatives should compute, we develop a non-standard interpretation of regular expressions as trace-closed languages. This means that if a word belongs to the interpretation, then so does any word that is equivalent to it (according to the equivalence relation induced by the independence relation). We then generalise the Brzozowski [20] and Antimirov [10] (syntactic) derivative operations to match this non-standard interpretation.

   We also investigate questions regarding the finiteness of the set syntactic reordering derivatives. This part is not directly related to relaxed memory models.

   Our generalisations of Brzozowski and Antimirov derivative operations can also be

used for constructing an automaton from a regular expression. In general, these automata cannot be finite for the simple reason that, in general, they must accept non-regular languages. We show that, for a class of regular expressions called *star-connected* expressions, the set of Antimirov reordering derivatives is finite modulo certain equations.

We also develop a refinement of the Antimirov reordering derivative operation that allows us to more precisely keep track of how an expression is derived along a word. With this refinement in mind, we define a stronger version of *(scattering) rank* [32] which we call *uniform (scattering) rank* and show that languages defined by star-connected expressions have finite uniform scattering rank. We then show that, if the language of an expression has finite uniform scattering rank, then the refined Antimirov reordering derivative operation can be used to construct a finite automaton from the expression. This construction does not require any quotienting, instead it relies on truncation to make the state set finite.

2. **Normalisation of generalised traces**: In the previous item we mentioned that the non-standard interpretation of regular expressions produces trace-closed languages. In other words, the non-standard interpretation wrt. independence relation I produces a language that can be partitioned into equivalence classes (traces) according to the equivalence relation induced by I. In some applications it might be desirable to work with concrete words representing these equivalence classes. For Mazurkiewicz traces there are two well-known normal forms that can be used to specify a canonical representative of a trace (equivalence class): the Foata normal form and the lexicographic normal form.

   Our development here is motivated by the fact that, in some cases, an independence relation (as in Mazurkiewicz traces) might not be expressive enough. More precisely, in Mazurkiewicz traces, the independence relation is a binary relation on the alphabet and thus it is static. By this we mean that we cannot have a pair of letters independent in some configuration of the system, but not in others.

   We develop Foata and lexicographic normal forms and corresponding normalisation algorithms for a generalisation of Mazurkiewicz traces introduced by Sassone, Winskel and Nielsen [74]. We also formalise this development in the dependently typed programming language Agda.

   In this generalisation the independence relation is replaced by a family of independence relations, i.e., for any word u we have an independence relation I_u associated with u. In this setting, $a\, I_u\, b$ (only) means that a and b are independent in the context u (the prefix of the letters a and b). In other words, if $a\, I_u\, b$, then $uabv$ and $ubav$ are considered equivalent, but it is not necessarily the case that the same holds for $u'abv$ and $u'bav$. This word parameter u can be seen as the current configuration of a system and thus allows us to say that a pair of letters are independent in some configuration but not in others.

   We are interested in generalised traces since, in some relaxed memory models, it may be that two instructions commute in some machine state but not in others. This could be because one of the instructions accesses a shared resource only under certain conditions. For example, a processor executing a read instruction might access the shared memory to determine the result of the read, but also it might not, when the result is determined based on locally available information.

3. **Operational semantics with semicommutations**: We further extend the Antimirov reordering derivatives to obtain an operational semantics that, according to a given independence relation, produces all possible executions that are justified by the reorderings described by the independence relation.

   The basic changes needed to go from Antimirov reordering derivatives to operational semantics are the following. First, we interpret the alphabet as the set of possible instructions. By this we mean that we can interpret the letters of the alphabet as (partial) state transformers. A program then is just a regular expression over this alphabet. Intuitively, the Antimirov reordering derivatives of a program tell us what the instructions are that we are allowed to execute next and what the corresponding residual programs are that still need to be executed afterwards. Here we also let go of the requirement that an independence relation must be symmetric. Thus we are working with a generalisation of Mazurkiewicz traces called *semicommutations* that was introduced by Clerbout and Latteux [24]. To describe parallel programs we also add parallel composition to the syntax.

   Note that, if we take the set of machine states to be a singleton set and we interpret the letters as the identity state transformer, then we essentially get back to Antimirov reordering derivatives.

   To allow more intricate relaxed behaviour we also describe a couple of extensions of this framework. For example, we allow a semicommutation relation to be context-dependent so that we are justified to reorder a pair of instructions in some state but not in others. We also allow for the possibility that reordering a pair of instructions can modify them. This is described using reordering actions, i.e., we have left and right actions of the alphabet (acting) on itself. We also allow for instructions to be executed in multiple steps.

   Relaxed memory models are often specified in the axiomatic style. To test the capabilities of the operational framework described above, we take the axiomatically specified multicopy-atomic ARMv8 [67] memory model and translate a fragment of it into our operational framework. The main contribution here is an example of how the extensions mentioned in the previous paragraph can be used to give an operational translation of an axiomatic model.

Outline

In Chapter 2 we go over some background material relevant to this work.

In Chapter 3 we describe the non-standard interpretation of regular expressions as trace-closed languages, the I-reordering language derivatives and the corresponding generalisations of the Brzozowski and Antimirov derivative operations. We then investigate the question of finiteness of the set of these syntactic derivatives. We also develop a refinement of the Antimirov reordering derivative. This is based on Publication I.

In Chapter 4 we cover the Foata and lexicographic normal forms for generalised traces together with the corresponding normalisation algorithms and correctness proofs. This is based on Publications (II–III).

In Chapter 5 we further generalise the Antimirov reordering derivative operation to develop an operational semantics for relaxed memory where the semantics is parameterised by an independence relation that controls the relaxedness of the semantics. This is based on Publication V. (The early ideas for this work are from Publication IV, but they were developed significantly further in Publication V.)

In Chapter 6 we give an example of how the framework from Chapter 5 could be used to describe a memory model. More precisely, we take the multicopy-atomic ARMv8 [67] memory model, which is axiomatically specified, and translate a fragment of it into the operational framework from Chapter 5. This is based on Publication V.

2 Preliminaries

In this chapter we briefly go over some background material that is necessary for or relevant to our later developments.

2.1 Word Languages

An *alphabet* is a finite non-empty set of *letters* (also called *symbols*). A *word* (or *string*) over an alphabet Σ is a finite sequence of letters from Σ. The empty word (the sequence consisting of zero letters) is denoted by ε. The concatenation of words u and v (denoted by $u \cdot v$ but $\cdot$ may be omitted and then we just write uv) is the sequence that consists of the letters of the sequence u followed by the letters of the sequence v.

The set Σ^* of all words over Σ is the free monoid on Σ with the empty word ε as the unit and concatenation of words as the multiplication. Thus we have that $\varepsilon u = u = u\varepsilon$ and $s(tu) = (st)u$ for any $s,t,u \in \Sigma^*$.

We write $|u|$ for the length of the word u (i.e., the length of the sequence u). We have $|\varepsilon| = 0$ (i.e., the length of the empty sequence is 0) and $|uv| = |u| + |v|$. For a set X, we write $|X|$ for the cardinality of the set X. By $\pi_X(u)$ we mean the projection of a word u to a subalphabet $X \subseteq \Sigma$. Thus $\pi_X(u)$ is a subword (or subsequence) of u obtained by discarding from u all letters that are not in X. By $\Sigma(u) \subseteq \Sigma$ we denote the set of letters that occur in u. As a shorthand we write $|u|_X$ for $|\pi_X(u)|$ which is the number of occurrences of letters from X in u.

For two words $u, v \in \Sigma^*$, we say that u is a prefix of v when there exists $t \in \Sigma^*$ such that $ut = v$. Similarly, v is a suffix of u when there exists t such that $tv = u$.

A *(word) language* over Σ is a set of words over Σ. The empty language is the empty set $\emptyset$ and the language consisting of all possible words over Σ, the universal language, is Σ^*. Thus a (word) language is a subset of Σ^*. The empty word and the concatenation of words lift to word languages via $\mathbf{1} =_{\mathrm{df}} \{\varepsilon\}$ and $L \cdot L' =_{\mathrm{df}} \{uv \mid u \in L \wedge v \in L'\}$.

The *shuffle (product)* of $u, v \in \Sigma^*$, denoted by $u \sqcup\!\sqcup v$, is the set of all valid interleavings of u and v.

$$\begin{array}{rcl} \varepsilon \sqcup\!\sqcup v & =_{\mathrm{df}} & \{v\} \\ u \sqcup\!\sqcup \varepsilon & =_{\mathrm{df}} & \{u\} \\ au \sqcup\!\sqcup bv & =_{\mathrm{df}} & \{a\} \cdot (u \sqcup\!\sqcup bv) \cup \{b\} \cdot (au \sqcup\!\sqcup v) \end{array}$$

This lifts to word languages via $L \sqcup\!\sqcup L' =_{\mathrm{df}} \{u \sqcup\!\sqcup v \mid u \in L \wedge v \in L'\}$.

2.2 Regular Languages

The set RE of *regular expressions* over an alphabet Σ is given by the grammar

$$E ::= a \mid 0 \mid E + E \mid 1 \mid EE \mid E^*$$

where a ranges over Σ. We write $\mathsf{RE}(\Sigma)$ when we need to explicitly specify the alphabet.

The *word-language semantics* of regular expressions is given by the function $[\![\_]\!] : \mathsf{RE} \to \mathscr{P}(\Sigma^*)$ defined recursively by

$$\begin{array}{rcl} [\![a]\!] & =_{\mathrm{df}} & \{a\} \\ [\![0]\!] & =_{\mathrm{df}} & \emptyset \\ [\![E+F]\!] & =_{\mathrm{df}} & [\![E]\!] \cup [\![F]\!] \\ [\![1]\!] & =_{\mathrm{df}} & \mathbf{1} \\ [\![EF]\!] & =_{\mathrm{df}} & [\![E]\!] \cdot [\![F]\!] \\ [\![E^*]\!] & =_{\mathrm{df}} & \mu X.\, \mathbf{1} \cup [\![E]\!] \cdot X = \bigcup_{n \in \mathbb{N}} [\![E]\!]^n \end{array}$$

A word language L is said to be *regular* (or *rational*) if $L = [\![E]\!]$ for some regular expression $E \in \mathsf{RE}$. We also say that $[\![\_]\!]$ is the standard interpretation of regular expressions as (regular) word languages.

A *deterministic finite automaton* (DFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set (of states), Σ is the (input) alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of final states. We write $(q)a$ for $\delta(q,a)$ and extend it to words as $(q)\varepsilon =_{\mathrm{df}} q$ and $(q)au =_{\mathrm{df}} ((q)a)u$. A deterministic automaton is said to accept a word u when $(q_0)u \in F$, i.e., when the automaton transitions according to δ from the initial state q_0 as prescribed by the word u to a state that is final (or accepting). The language $\{u \in \Sigma^* \mid (q_0)u \in F\}$ of all words accepted by the automaton is the language recognised by the automaton. The transitions of the automaton can also be given relationally in terms of δ as $\{(q,a,(q)a) \mid q \in Q \wedge a \in \Sigma\}$.

A *nondeterministic finite automaton* (NFA) is otherwise just like a deterministic finite automaton, except that the transition function becomes $\delta : Q \times \Sigma \to \mathscr{P}(Q)$, i.e., in state $q \in Q$ with input letter $a \in \Sigma$ we may have zero or more states we can transition to. We write $(q)a$ for the set $\delta(q,a)$ and extend it to words as $(q)\varepsilon =_{\mathrm{df}} \{q\}$ and $(q)au =_{\mathrm{df}} \bigcup\{(q')u \mid q' \in (q)a\}$. A nondeterministic automaton is said to accept a word u when u can take the automaton from the initial state q_0 to some accepting state $q \in F$. In other words, u is accepted when $(q_0)u \cap F \neq \emptyset$. The transitions of the automaton can also be given relationally in terms of δ as $\{(q,a,q') \mid q \in Q \wedge a \in \Sigma \wedge q' \in (q)a\}$.

Kleene's theorem [40] says that a word language is rational iff it is *recognisable*, i.e., accepted by a deterministic finite automaton (acceptance by a nondeterministic finite automaton is an equivalent condition because of determinisability [69]).

A Kleene algebra is an idempotent semiring with an additional operation $(\_)^*$ (the Kleene star) together with some additional axioms governing this operation. It was shown by Kozen [42] that the set $\{[\![E]\!] \mid E \in \mathsf{RE}\}$ of all regular languages together with the language operations $\emptyset, \cup, \mathbf{1}, \cdot, (\_)^*$ is the free Kleene algebra on Σ. An important consequence of this is that the equational theory of Kleene algebra is sound and complete for $[\![\_]\!]$. In other words, we have that $E \doteq F$ iff $[\![E]\!] = [\![F]\!]$ where $\doteq$ refers to valid equations in the Kleene algebra theory.

2.3 Mazurkiewicz Traces

An *independence relation* on an alphabet Σ is an irreflexive and symmetric binary relation $I \subseteq \Sigma \times \Sigma$. Its complement $D = \Sigma \times \Sigma \setminus I$, which is reflexive and symmetric, is called the *dependence relation*. An independence (or concurrency) alphabet (Σ, I) is just an alphabet Σ together with an independence relation I on it. We extend the independence relation to words by saying that two words u and v are independent, denoted by $u\,I\,v$, if $a\,I\,b$ for all $a,b \in \Sigma$ such that $a \in \Sigma(u)$ and $b \in \Sigma(v)$. Two words u and v are dependent, denoted by $u\,D\,v$, when they are not independent, i.e., there exist $a,b \in \Sigma$ such that $a \in \Sigma(u)$, $b \in \Sigma(v)$ and $a\,D\,b$.

Intuitively, the independence relation reflects the meaning we have attached to the symbols in the alphabet Σ. If $a\,I\,b$, then we say that the letters a and b are independent and by this we mean that the ordering of the letters a and b in a word does not matter, i.e., the words $uabv$ and $ubav$ represent the same thing (we consider them equivalent).

We define $\sim^I \subseteq \Sigma^* \times \Sigma^*$ to be the least relation such that $a\,I\,b$ implies $uabv \sim^I ubav$, i.e., it relates words that differ only by the ordering of a pair of adjacent independent letters. We define *(Mazurkiewicz) equivalence* $\sim^{I*}$ to be its reflexive-transitive closure. A *(Mazurkiewicz) trace* is an equivalence class of words wrt. $\sim^{I*}$. The equivalence class of a word w is denoted by $[w]^I$. Equivalently, we could define $\sim^{I*}$ as the least congruence

(wrt. $\cdot$) such that $a\,I\,b$ implies $ab \sim^{I*} ba$. A consequence of the definition of the equivalence relation is that the ordering of dependent letters is fixed in an equivalence class, i.e., if $a\,D\,b$ and $u \sim^{I*} v$, then $\pi_{\{a,b\}}(u) = \pi_{\{a,b\}}(v)$.

The set $\Sigma^*/\sim^{I*}$ of all traces is the free partially commutative monoid on (Σ, I). If $I = \emptyset$, then $\Sigma^*/\sim^{I*} \cong \Sigma^*$, the set of words, i.e., we recover the free monoid. On the other hand, if $I = \{(a,b) \mid a \neq b\}$, then $\Sigma^*/\sim^{I*} \cong \mathscr{M}_{\mathrm{f}}(\Sigma)$, the set of finite multisets over Σ, i.e., the free commutative monoid.

A *trace language* is a subset of $\Sigma^*/\sim^{I*}$. Trace languages are in bijection with word languages L that are *(trace) closed* in the sense that, if $z \in L$ and $z \sim^{I*} w$, then also $w \in L$. If T is a trace language, then its flattening $L =_{\mathrm{df}} \bigcup T = \{u \in \Sigma^* \mid [u]^I \in T\}$ is a closed word language. On the other hand, the trace language corresponding to a closed word language L is $T =_{\mathrm{df}} \{t \in \Sigma^*/\sim^{I*} \mid \exists z \in t.\, z \in L\} = \{t \in \Sigma^*/\sim^{I*} \mid \forall z \in t.\, z \in L\}$.

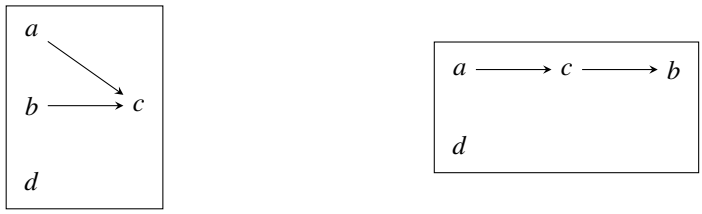
Given a general (not necessarily closed) word language L, we define its *(trace) closure* $[L]^I$ as the least closed word language containing L. Clearly $[L]^I = \{w \in \Sigma^* \mid \exists z \in L.\, w \sim^{I*} z\}$ and also $[L]^I = \bigcup\{t \in \Sigma^*/\sim^{I*} \mid \exists z \in t.\, z \in L\}$. The trace closure operator $[\_]^I$ is indeed a closure operator as, for any L, we have $L \subseteq [L]^I$ and $[[L]^I]^I = [L]^I$. Furthermore, we have $[\emptyset]^I = \emptyset$ and, for any L and L', $[L \cup L']^I = [L]^I \cup [L']^I$. We also have that $[\mathbf{1}]^I = \mathbf{1}$ and, for any $a \in \Sigma$, $[\{a\}]^I = \{a\}$. A language L is closed iff $[L]^I = L$.

A word $w = a_1 \ldots a_n$ where $a_i \in \Sigma$ yields a directed node-labelled acyclic graph as follows. We take the vertex set to be $V =_{\mathrm{df}} \{1, \ldots, n\}$ and we label vertex i with a_i. We take the edge set to be $E =_{\mathrm{df}} \{(i,j) \mid i < j \wedge a_i D a_j\}$. This graph (V, E) for a word w is called the *dependence graph* of w and is denoted by $\langle w \rangle_D$. If $w \sim^{I*} z$, then the dependence graphs of w and z are isomorphic, i.e., traces can be identified with dependence graphs up to isomorphism. If $w \sim^{I*} z$, then w is a linearisation of $\langle z \rangle_D$.

We say that a letter a is minimal in the word z when there exist v' and v'' such that $z = v'av''$ and $v'\,I\,a$. An equivalent condition is that a is the first letter of an equivalent word, i.e., there exists v such that $z \sim^{I*} av$. In the dependence graph of z the node corresponding to this a has no incoming edges. If $I = \emptyset$, then the word av has exactly one minimal letter and this is the letter a.

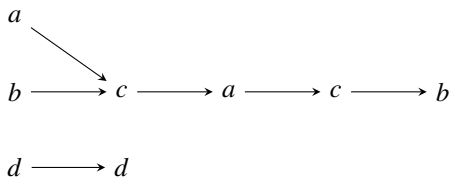
As an example independence alphabet we take $\Sigma =_{\mathrm{df}} \{a,b,c,d\}$ and I to be the least symmetric relation satisfying $a\,I\,b$, $a\,I\,d$, $b\,I\,d$, $c\,I\,d$. Then the words $abcd$ and $bdac$ are equivalent, since $abcd \sim^I bacd \sim^I badc \sim^I bdac$, but $acbd$ is not equivalent to them. The words $abcd$, $abdc$, $adbc$, $bacd$, $badc$, $bdac$, $dabc$, $dbac$ form one equivalence class of words or a trace. Another is $\{acbd, acdb, adcb, dacb\}$. Altogether, there are four traces containing each letter from Σ exactly once.

Continuing with the above example, we have seen that the words $abcd$ and $acbd$ are not equivalent and thus should have distinct dependence graphs. Indeed, we can draw the dependence graphs of the words $abcd$ and $acbd$ as follows (we have drawn rectangles to separate the two dependence graphs).



The concatenation of two dependence graphs is obtained by adding the necessary edges between dependent vertices of the two graphs. The concatenation of the two depen-

dence graphs above ($\langle abcd \rangle_D$ and $\langle acbd \rangle_D$) can be drawn as follows.



Just to emphasise, we draw the transitive reduct of the dependence graph, i.e., we have omitted transitive edges like the one between the two a's. We can check that this is indeed the dependence graph of $abcdacbd$.

2.3.1 Normal Forms

Traces are equivalence classes of words. We have seen that a trace can be identified with its dependence graph (up to isomorphism) and thus a dependence graph can be used to represent a trace. Here we describe two well-known normal forms that specify a word representing a trace as a canonical representative of that trace. For this we require a strict total order (i.e., a transitive and asymmetric relation) $\prec$ on the alphabet. We take the strict total order on the example alphabet Σ to be $a \prec b \prec c \prec d$.

The Foata normal form [21] is a well-formed sequence of well-formed steps. A well-formed step is a $\prec$-sorted word where all the letters are pairwise independent. We think of a step as a set of independent letters, but add the requirement of $\prec$-sortedness to pick a representative word for that step. A sequence of steps is well-formed when every letter in a step has a dependent letter in the previous step (the first step is excepted). This leads to the maximally parallel representation of the trace, with every letter occurrence in the earliest possible step.

The Foata normal form of $abcd$ in our example is $(abd)(c)$. Here abd is the first step and c is the second step of the normal form. It is a normal form since the letters are pairwise independent in both of the steps and c has a dependent letter in the previous step. It is the normal form of $abcd$ since, if we turn the normal form $(abd)(c)$ into a word by flattening the steps, we get $abdc$, which is equivalent to $abcd$. Similarly, the normal form of $acbd$ is $(ad)(c)(b)$.

The Foata normal form of a trace can be read off of its dependence graph. The first step of the normal form is the set of minimal letters (vertices with no incoming edges) in its dependence graph. The rest of the normal form is obtained recursively from the residual dependence graph, i.e., the dependence graph with its minimal letters removed. From the dependence graph of $abcd$ above we can see that the first step in the normal form of $abcd$ is indeed (abd). Similarly, the first step in the normal form of $acbd$ is (ad). From the dependence graph of $abcdacbd$ we can see that the first step in the normal form is (abd).

A word is said to be in lexicographic normal form if it is the least element in its equivalence class according to the lexicographic ordering induced by $\prec$. An equivalent characterisation in terms of a forbidden pattern was given by Anisimov and Knuth [9]: a word s is in lexicographic normal form if and only if, for every factorisation $tbuav$ of s where $b\,I\,a$ and $a \prec b$, there is a letter d in u such that $d\,D\,a$ (in other words, we are not able to commute a to the left past b).

The word $abcd$ is the lexicographic normal form in its equivalence class. Similarly, the word $acbd$ is the lexicographic normal form in its equivalence class. The only potentially forbidden pattern in this word could be formed by c and b as they occur in the wrong order, but the two letters are dependent and hence there are no forbidden patterns. The

equivalent word $acdb$ has a forbidden pattern: the letters d and b are in the wrong order, independent, and the subword between them (the empty word) does not contain a dependent letter. We denote the set of lexicographic normal forms for the independence alphabet (Σ, I) by $Lex(\Sigma, I)$.

Similarly to the Foata normal form, the lexicographic normal form of a trace can also be read off of its dependence graph. The first letter in the normal form is the least letter according to $\prec$ among the minimal letters in the dependence graph of the trace. The rest of the normal form is obtained recursively from the residual dependence graph, i.e., the dependence graph with the first letter removed.

2.4 Properties of Trace Closures of Regular Languages

Trace closures of regular languages are theoretically interesting due to their intricate properties and have therefore been studied in a number of works, e.g., [13, 59, 3, 71, 32, 41]. For a thorough survey, see Ochmański's handbook chapter [60].

An important property for us is that the trace closure of a regular language is not necessarily regular.

Proposition 2.1. *There exists a regular language L such that $[L]^I$ is not regular.*

Proof. Take $\Sigma =_{\mathrm{df}} \{a, b\}$, $a \mathrel{I} b$ and let the regular language be $L =_{\mathrm{df}} [\![(ab)^*]\!]$. The language $[L]^I = \{u \in \Sigma^* \mid |u|_a = |u|_b\}$ is not regular. □

The class of trace closures of regular languages behaves quite differently from the class of regular languages. Here are some results demonstrating this.

Theorem 2.2 (Bertoni et al. [14], Aalbersberg and Welzl [3], Sakarovitch [71])**.** *(cf. [60, Thm. 6.2.5]) The class of trace closures (wrt. I) of regular languages (over Σ) is closed under complement iff I is quasi-transitive (i.e., its reflexive closure is transitive).*

Theorem 2.3 (Bertoni et al. [13], Aalbersberg and Welzl [3] ("if" part); Aalbersberg and Hoogeboom [1])**.** *(cf. [60, Thm. 6.2.5]) The problem of whether the trace closures (wrt. I) of two regular languages (over Σ) are equal is decidable iff I is quasi-transitive.*

Theorem 2.4 (Sakarovitch [72])**.** *(cf. [60, Thm. 6.2.7]) The problem of whether the trace closure (wrt. I) of the language of an expression over Σ is regular is decidable iff I is quasi-transitive.*

A closed language is regular iff the corresponding trace language is accepted by a finite asynchronous (a.k.a. Zielonka) automaton [80, 81]. In Section 2.6 we will see further characterisations of regular closed languages based on star-connected expressions.

2.5 Rational and Recognisable Languages of Monoids

Trace languages are a special case of languages of monoids. A subset T of a monoid M is called an M-language. An M-language T is called *rational* if $T = [\![E]\!]^M$ for some regular expression E over M. Here $[\![\_]\!]^M : \mathrm{RE}(M) \to \mathscr{P}(M)$ interprets any element m of M as $\{m\}$, the $0, +$ constructors of regular expressions by $\emptyset$ and $\cup$, the $1, \cdot$ constructors as mandated by the monoid structure, and $(\_)^*$ as the appropriate least fixpoint.

An M-language T is *recognised* by an action $\delta : Q \times M \to Q$ if there exist $q_0 \in Q$ and $F \subseteq Q$ such that $T = \{m \in M \mid (q_0)m \in F\}$ where we write $(q)m$ for $\delta(q, m)$. In other words, T is the set of elements of M that take q_0 to F. An M-language T is *recognisable* if it is recognised by an action of M on a finite set. If an M-language T is recognised by an

action $\delta : Q \times M \to Q$, then δ can be seen as an automaton by taking Q to be the state set, q_0 to be the initial state, F to be the final states and the set of transitions is given by $\{(q, m, (q)m) \mid q \in Q \wedge m \in M\}$. If M is finitely generated with G as the generators, then we can obtain an equivalent automaton by restricting the set of transitions to be $\{(q, m, (q)m) \mid q \in Q \wedge m \in G\}$. If Q is finite, then the resulting automaton is also finite.

Kleene's celebrated theorem says that, for languages of free monoids on finite sets (i.e., word languages over finite alphabets), rationality and recognisability are equivalent conditions (and we can thus just speak about regularity). For a general monoid, however, the two notions are different.

Theorem 2.5 (Kleene [40])**.** *Let M be the free monoid Σ^* on a finite set Σ. An M-language T is rational iff T is recognisable.*

Theorem 2.6 (McKnight [53])**.** *Let M be finitely generated. If an M-language T is recognisable, then T is rational.*

Given a monoid M and a congruence $\equiv$ on M, the set $M/\equiv$ is a monoid too. We view $M/\equiv$-languages as sets of equivalence classes wrt. $\equiv$.

Proposition 2.7. *Given a monoid M and a congruence $\equiv$ on it.*

1. *Given a regular expression E, its $M/\equiv$-language $[\![E]\!]^{M/\equiv}$ is expressible via its M-language $[\![E]\!]^M$ by $[\![E]\!]^{M/\equiv} = \{t \in M/\equiv \mid \exists u \in t.\, u \in [\![E]\!]^M\}$.*
2. *A $M/\equiv$-language T is recognisable iff its flattening $\bigcup T$ into an M-language is recognisable.*

In the free partially commutative monoid the classes of rational and recognisable languages are different: the class of recognisable languages is a proper subclass of that of rational languages. In view of Proposition 2.7, a trace language T is rational if and only if $T = \{t \in \Sigma^*/\sim^{I*} \mid \exists u \in t.\, u \in L\}$ or, equivalently, $\bigcup T = [L]^I$ for some regular word language L (in the terminology of Aalbersberg and Welzl [3], such a trace language T is called *existentially regular*), and recognisable iff $\bigcup T = L$ for some regular word language L (such a trace language is called *consistently regular*).

The question of when a rational trace language is recognisable is nontrivial. We have just seen that, reformulated in terms of word languages, it becomes: given a regular language L, when is its trace closure $[L]^I$ regular?

2.6 Star-Connected Expressions

Star-connected expressions are important as they characterise regular closed languages. A corollary of that is a further characterisation of such languages in terms of a "concurrent" semantics of regular expressions that interprets the Kleene star non-standardly as "concurrent star".

Definition 2.8. A word $w \in \Sigma^*$ is connected if its dependence graph $\langle w \rangle_D$ is connected. A language $L \subseteq \Sigma^*$ is connected if every word $w \in L$ is connected.

Definition 2.9.

1. *Star-connected expressions* are a subset of the set of all regular expressions defined inductively by: 0, 1 and $a \in \Sigma$ are star-connected. If E and F are star-connected, then so are $E + F$ and EF. If E is star-connected and $[\![E]\!]$ is a connected language, then E^* is star-connected.

2. A language L is said to be *star-connected* if $L = [\![E]\!]$ for some star-connected expression E.

Ochmański [59] proved that a closed language is regular iff it is the closure of a star-connected language. This means that, for any expression E, the language $[[\![E]\!]]^I$ is regular iff there exists a star-connected expression E' such that $[[\![E]\!]]^I = [[\![E']\!]]^I$. It is important to realise that generally $E \neq E'$ and also $[\![E]\!] \neq [\![E']\!]$. Ochmański's proof was as follows (recall that we write $Lex(\Sigma, I)$ for the set of all lexicographic normal forms over the independence alphabet (Σ, I)).

Lemma 2.10. *(cf. [60, Props. 6.3.4, 6.3.10])*

1. *$Lex(\Sigma, I)$ is regular.*
2. *For any regular language L, if $L \subseteq Lex(\Sigma, I)$, then L is star-connected.*

Theorem 2.11 (Ochmański [59])**.** *(cf. [60, Thm. 6.3.13]) For any closed language L (i.e., $L = [L]^I$), the following are equivalent:*

1. *L is regular;*
2. *$L \cap Lex(\Sigma, I)$ is regular;*
3. *there exists a star-connected L' such that $L = [L']^I$.*

Proof. (1) $\Longrightarrow$ (2) is a consequence of Lemma 2.10(1) as the intersection of regular languages is regular. (2) $\Longrightarrow$ (3) follows from Lemma 2.10(2) as $L = [L \cap Lex(\Sigma, I)]^I$. For the step (3) $\Longrightarrow$ (1), Ochmański employed Hachiguchi's notion of rank of a language and Hachiguchi's lemma, which we will study in Definition 3.43 and Proposition 3.44 below, and proved that, if L is closed and connected, then L^* has rank. □

The non-standard *concurrent-star trace-language semantics* of regular expressions, denoted by $[\![\_]\!]^{\mathrm{con}} : \mathrm{RE}(\Sigma) \to \mathscr{P}(\Sigma^*)$, is like $[\![\_]\!]$ except that the Kleene star is interpreted non-standardly as the *concurrent star* operation. Informally, the concurrent star of a language iterates not the given language but the language of connected components of its words.

The concurrent star of a connected language coincides with its Kleene star. The idea of this non-standard semantics is to make non-star-connected regular expressions harmless, so as to obtain the following replacement for Kleene's theorem.

Theorem 2.12 (Ochmański [59])**.** *(cf. [60, Thm. 6.3.16]) A closed language L is regular iff $L = [[\![E]\!]^{\mathrm{con}}]^I$ for some regular expression E.*

2.7 Derivatives of a Language

A word language L is said to be *nullable* (or that it has the *empty word property*), denoted by $L\!\downarrow$, if $\varepsilon \in L$. The *derivative* (or *left quotient*)[1] of L along a word u is defined by

$$\mathsf{D}_u L =_{\mathrm{df}} \{v \in \Sigma^* \mid uv \in L\}.$$

For any L, we have $\mathsf{D}_\varepsilon L = L$ as well as $\mathsf{D}_{uv} L = \mathsf{D}_v(\mathsf{D}_u L)$ for any $u, v \in \Sigma^*$. Thus the operation $\mathsf{D} : \mathscr{P}(\Sigma^*) \times \Sigma^* \to \mathscr{P}(\Sigma^*)$ is a right action of Σ^* on $\mathscr{P}(\Sigma^*)$. We have

$$L = \{\varepsilon \mid L\!\downarrow\} \cup \bigcup \{\{a\} \cdot \mathsf{D}_a L \mid a \in \Sigma\}$$

[1] We use the word 'derivative' both for languages and expressions, reserving the word 'quotient' for quotients of sets by equivalence relations.

and, for any $u \in \Sigma^*$, we have

$$u \in L \iff (\mathsf{D}_u L)\!\downarrow .$$

Example 2.13. Let $\Sigma =_{\mathrm{df}} \{a,b\}$, $E =_{\mathrm{df}} 1+a(1+b(1+a))+b(1+a(1+b))$ and $L =_{\mathrm{df}} [\![E]\!]$. We have

$$L = \{\varepsilon, a, ab, aba, b, ba, bab\}$$

and thus

$$\mathsf{D}_a L = \{\varepsilon, b, ba\}.$$

Since $\varepsilon \in \mathsf{D}_a L$, we have $(\mathsf{D}_a L)\!\downarrow$ and thus we know it must be that $a \in L$.

Similarly, we have $\mathsf{D}_b L = \{\varepsilon, a, ab\}$, $\mathsf{D}_{ab} L = \{\varepsilon, a\}$ and $\mathsf{D}_{aba} L = \{\varepsilon\}$. On the other hand, $\mathsf{D}_{aa} L = \emptyset$ and thus we know that $aa \notin L$.

Derivatives of regular languages are regular. A remarkable fact is that they can be computed syntactically, on the level of regular expressions. There are two constructions for this, due to Brzozowski [20] and Antimirov [10]. We now continue with a brief overview of these two constructions.

2.7.1 Brzozowski Derivative

Nullability and derivative are semantic notions, defined on languages. However, Brzozowski [20] noticed that for regular languages, one can compute nullability and the derivatives syntactically, on the level of regular expressions.

Definition 2.14. The *syntactic nullability* (or *empty word property*) and the *Brzozowski derivative* of a regular expression are given by functions $(\_)\!\downarrow : \mathsf{RE} \to \mathbb{B}$, $\mathsf{D} : \mathsf{RE} \times \Sigma \to \mathsf{RE}$ and $\mathsf{D} : \mathsf{RE} \times \Sigma^* \to \mathsf{RE}$ defined recursively by

$$\begin{array}{rcl@{\qquad}rcl}
b\!\downarrow & =_{\mathrm{df}} & \mathsf{ff} & \mathsf{D}_a b & =_{\mathrm{df}} & \text{if } a = b \text{ then } 1 \text{ else } 0 \\
0\!\downarrow & =_{\mathrm{df}} & \mathsf{ff} & \mathsf{D}_a 0 & =_{\mathrm{df}} & 0 \\
(E+F)\!\downarrow & =_{\mathrm{df}} & E\!\downarrow \vee F\!\downarrow & \mathsf{D}_a(E+F) & =_{\mathrm{df}} & \mathsf{D}_a E + \mathsf{D}_a F \\
1\!\downarrow & =_{\mathrm{df}} & \mathsf{tt} & \mathsf{D}_a 1 & =_{\mathrm{df}} & 0 \\
(EF)\!\downarrow & =_{\mathrm{df}} & E\!\downarrow \wedge F\!\downarrow & \mathsf{D}_a(EF) & =_{\mathrm{df}} & \text{if } E\!\downarrow \text{ then } (\mathsf{D}_a E)F + \mathsf{D}_a F \text{ else } (\mathsf{D}_a E)F \\
(E^*)\!\downarrow & =_{\mathrm{df}} & \mathsf{tt} & \mathsf{D}_a(E^*) & =_{\mathrm{df}} & (\mathsf{D}_a E)E^*
\end{array}$$

$$\begin{array}{rcl}
\mathsf{D}_\varepsilon E & =_{\mathrm{df}} & E \\
\mathsf{D}_{ua} E & =_{\mathrm{df}} & \mathsf{D}_a(\mathsf{D}_u E)
\end{array}$$

The important property is that the syntactic nullability and derivative agree with their semantic counterparts as shown by the following proposition.

Proposition 2.15. *For any E,*

1. $[\![E]\!]\!\downarrow = E\!\downarrow$;
2. *for any $a \in \Sigma$,* $\mathsf{D}_a[\![E]\!] = [\![\mathsf{D}_a E]\!]$;
3. *for any $u \in \Sigma^*$,* $\mathsf{D}_u[\![E]\!] = [\![\mathsf{D}_u E]\!]$.

Corollary 2.16. *For any E,*

1. $[\![E]\!] = \{\varepsilon \mid E\!\downarrow\} \cup \bigcup\{\{a\} \cdot [\![\mathsf{D}_a E]\!] \mid a \in \Sigma\}$;
2. *for any $a \in \Sigma$ and $v \in \Sigma^*$, $av \in [\![E]\!]$ iff $v \in [\![\mathsf{D}_a E]\!]$;*
3. *for any $u, v \in \Sigma^*$, $uv \in [\![E]\!]$ iff $v \in [\![\mathsf{D}_u E]\!]$;*

4. *for any* $u \in \Sigma^*$, $u \in [\![E]\!]$ *iff* $(\mathsf{D}_u E)\not\downarrow$.

The Brzozowski derivative operation gives a method for turning a regular expression into a deterministic automaton. Given an expression E, the set of states is $Q^E = \{\mathsf{D}_u E \mid u \in \Sigma^*\}$, the initial state is $q_0^E = E$, the final states are $F^E = \{E' \in Q^E \mid E'\not\downarrow\}$ and the transition function δ^E is defined by D restricted to Q^E. This automaton is generally not finite, but becomes finite when quotiented by associativity, commutativity and idempotence of $+$. Identified up to the Kleene algebra theory, the states of the Brzozowski automaton correspond to the derivatives of the language $[\![E]\!]$. Note that regular languages can be characterised as languages with finitely many derivatives.

Example 2.17. Let $\Sigma =_{\mathrm{df}} \{a,b\}$, $E =_{\mathrm{df}} 1+a(1+b(1+a))+b(1+a(1+b))$ and $L =_{\mathrm{df}} [\![E]\!]$ as in Example 2.13. Before (in 2.13) we calculated that $\mathsf{D}_a L = \{\varepsilon, b, ba\}$. Syntactically we have the following.

$$\begin{array}{rcl}
\mathsf{D}_a E & = & \mathsf{D}_a 1 + \mathsf{D}_a(a(1+b(1+a))) + \mathsf{D}_a(b(1+a(1+b))) \\
& = & 0 + (\mathsf{D}_a a)(1+b(1+a)) + (\mathsf{D}_a b)(1+a(1+b)) \\
& = & 0 + 1(1+b(1+a)) + 0(1+a(1+b)) \\
& \doteq & 1 + b + ba
\end{array}$$

We can see that $\mathsf{D}_a L = [\![\mathsf{D}_a E]\!]$. Before we had $\mathsf{D}_{ab} L = \{\varepsilon, a\}$. Thus we should be able to show that $\mathsf{D}_{ab} E \doteq 1 + a$.

$$\begin{array}{rcl}
\mathsf{D}_{ab} E & = & \mathsf{D}_b(\mathsf{D}_a E) \\
& \doteq & \mathsf{D}_b(1 + b + ba) \\
& = & \mathsf{D}_b 1 + \mathsf{D}_b b + \mathsf{D}_b(ba) \\
& = & 0 + 1 + (\mathsf{D}_b b)a \\
& = & 0 + 1 + 1a \\
& \doteq & 1 + a
\end{array}$$

Before we had that $\mathsf{D}_{aa} L = \emptyset$. Thus we should be able to show that $\mathsf{D}_{aa} E \doteq 0$.

$$\begin{array}{rcl}
\mathsf{D}_{aa} E & = & \mathsf{D}_a(\mathsf{D}_a E) \\
& \doteq & \mathsf{D}_a(1 + b + ba) \\
& = & \mathsf{D}_a 1 + \mathsf{D}_a b + \mathsf{D}_a(ba) \\
& = & 0 + 0 + (\mathsf{D}_a b)a \\
& = & 0 + 0 + 0a \\
& \doteq & 0
\end{array}$$

2.7.2 Antimirov Derivative

Antimirov [10] optimised Brzozowski's construction essentially constructing a nondeterministic finite automaton instead of a deterministic one, with a smaller number of states and, crucially, without having to identify states up to equations.

Antimirov's syntactic derivative operation is a multivalued function, in other words, a relation. Antimirov spoke of "partial derivatives", we prefer to use the term "parts-of-derivative". The relational definition below corresponds to the equational characterisation given in [10].

Definition 2.18. The *Antimirov parts-of-derivative* of a regular expression along a letter and a word are given by the relations $\to \subseteq \mathsf{RE} \times \Sigma \times \mathsf{RE}$ and $\to^* \subseteq \mathsf{RE} \times \Sigma^* \times \mathsf{RE}$ defined

inductively by

$$\frac{}{a \to (a,1)} \quad \frac{E \to (a,E')}{E+F \to (a,E')} \quad \frac{F \to (a,F')}{E+F \to (a,F')}$$

$$\frac{E \to (a,E')}{EF \to (a,E'F)} \quad \frac{E \not\to \quad F \to (a,F')}{EF \to (a,F')} \quad \frac{E \to (a,E')}{E^* \to (a,E'E^*)}$$

$$\frac{}{E \to^* (\varepsilon,E)} \quad \frac{E \to^* (u,E') \quad E' \to (a,E'')}{E \to^* (ua,E'')}$$

The following proposition tells us that the Antimirov parts-of-derivative indeed compute parts of the semantic derivative (though the parts are not necessarily disjoint). Put another way, the Antimirov parts-of-derivative collectively compute the semantic derivative.

Proposition 2.19. *For any E,*

1. *for any $a \in \Sigma$, $\mathsf{D}_a[\![E]\!] = \bigcup\{[\![E']\!] \mid E \to (a,E')\}$;*
2. *for any $u \in \Sigma^*$, $\mathsf{D}_u[\![E]\!] = \bigcup\{[\![E']\!] \mid E \to^* (u,E')\}$.*

Corollary 2.20. *For any E,*

1. *for any $a \in \Sigma$ and $v \in \Sigma^*$, $av \in [\![E]\!] \iff \exists E'.\, E \to (a,E') \wedge v \in [\![E']\!]$;*
2. *for any $u,v \in \Sigma^*$, $uv \in [\![E]\!] \iff \exists E'.\, E \to^* (u,E') \wedge v \in [\![E']\!]$;*
3. *for any $u \in \Sigma^*$, $u \in [\![E]\!] \iff \exists E'.\, E \to^* (u,E') \wedge E' \not\to$.*

The last item tells us that a word u belongs to the language $[\![E]\!]$ if and only if there exists a derivation for $E \to^* (u,E')$ such that $E' \not\to$. Thus each such derivation gives us a justification why the word u belongs to $[\![E]\!]$. Importantly, there can be many such derivations for a given u.

If we took languages to be multisets of words (i.e., introduced the notion of a word occurring in a language some number of times) and adopted the obvious multisets-of-words semantics of regular expressions, then the Antimirov parts-of-derivative would also compute the semantic derivative, but in a partitioning manner. In the sets-of-words semantics, however, overlaps are possible, so we do not get a partition.

The parts-of-derivative of an expression E induce a nondeterministic automaton. The state set is $Q^E =_{\mathrm{df}} \{E' \mid \exists u \in \Sigma^*.\, E \to^* (u,E')\}$. The initial state is $q_0^E =_{\mathrm{df}} E$. The set of final states is $F^E =_{\mathrm{df}} \{E' \in Q^E \mid E' \not\to\}$. Finally, the transition relation is defined by $E' \to^E (a,E'') =_{\mathrm{df}} E' \to (a,E'')$ for $E',E'' \in Q^E$.

The state set Q^E is shown finite by proving it to be a subset of another set that is straightforwardly seen to be finite.

Definition 2.21. For any E, the set $E^{\to^*}$ of regular expressions is defined recursively by

$$\begin{aligned}
E^{\to^*} &=_{\mathrm{df}} \{E\} \cup E^{\to^+} \\
\\
a^{\to^+} &=_{\mathrm{df}} \{1\} \\
0^{\to^+} &=_{\mathrm{df}} \emptyset \\
(E+F)^{\to^+} &=_{\mathrm{df}} E^{\to^+} \cup F^{\to^+} \\
1^{\to^+} &=_{\mathrm{df}} \emptyset \\
(EF)^{\to^+} &=_{\mathrm{df}} E^{\to^+} \cdot \{F\} \cup F^{\to^+} \\
(E^*)^{\to^+} &=_{\mathrm{df}} E^{\to^+} \cdot \{E^*\}
\end{aligned}$$

Proposition 2.22. *For any E,*

1. *$E^{\to^*}$ is finite, in fact, of cardinality linear in the size of E;*
2. *$Q^E \subseteq E^{\to^*}$.*

Corollary 2.23. *For any E, the Antimirov automaton is finite.*

We note that the Antimirov automaton, constructed as above, while canonical, is generally not trim: every state is reachable, but not every state is generally coreachable (i.e., not every state needs to have a path to some final state). A state E' is not coreachable if and only if $[\![E']\!] = \emptyset$. This is the case precisely when E' equals 0 in the theory of idempotence of $+$ and the left and right zero laws of 0 wrt. $\cdot$. The Antimirov automaton is trimmed by removing the states that are not coreachable.

Now we can also show that a suitable sound quotient of the Brzozowski automaton is finite. (We consider a quotient to be sound if the resulting automaton is equivalent to the original automaton.) For this we prove a syntactic version of Proposition 2.19 relating the Brzozowski derivative and the Antimirov parts-of-derivative.

Proposition 2.24. *For any E,*

1. *for any $a \in \Sigma$, $\mathsf{D}_a E \doteq \sum\{E' \mid E \to (a, E')\}$;*
2. *for any $u \in \Sigma^*$, $\mathsf{D}_u E \doteq \sum\{E' \mid E \to^* (u, E')\}$.*

(using the semilattice equations for 0 and $+$, the left zero law, and distributivity of $\cdot$ over $+$ from the right).

Corollary 2.25. *For any E, the Brzozowski automaton, suitably quotiented, is finite.*

Proof. Just notice that the powerset of a finite set is finite too. □

This quotient does not give the minimal deterministic automaton (given by semantic derivatives of $[\![E]\!]$). The minimal deterministic automaton is obtained from the Brzozowski automaton by quotienting it by the full Kleene algebra theory.

Example 2.26. Let $\Sigma =_{\mathrm{df}} \{a, b\}$, $E =_{\mathrm{df}} 1 + a(1 + b(1 + a)) + b(1 + a(1 + b))$ and $L =_{\mathrm{df}} [\![E]\!]$ as in Examples 2.13 and 2.17. For the Brzozowski derivative we had that $\mathsf{D}_a E \doteq 1 + b + ba$. With Antimirov parts-of-derivative we have the following.

$$\dfrac{\dfrac{\overline{a \to (a, 1)}}{a(1 + b(1 + a)) \to (a, 1(1 + b(1 + a))}}{1 + a(1 + b(1 + a)) + b(1 + a(1 + b)) \to (a, 1(1 + b(1 + a)))}$$

Of course $1(1 + b(1 + a)) \doteq 1 + b + ba$. Note that $a(1 + b(1 + a))$ is the part of E that we can derive along a with Antimirov parts-of-derivative, i.e., there is no derivation for $1 \to (a, E')$ nor $b(1 + a(1 + b)) \to (a, E'')$ for any E' and E''.

Before we had that $\mathsf{D}_{ab} E \doteq 1 + a$ and similarly we consider here deriving the expression $1 + b + ba$ along b. The difference is that here we now have a choice, i.e., we have two ways to derive it along b.

$$\dfrac{\dfrac{\overline{b \to (b, 1)}}{b + ba \to (b, 1)}}{1 + b + ba \to (b, 1)} \qquad \dfrac{\dfrac{\overline{b \to (b, 1)}}{ba \to (b, 1a)}}{\dfrac{b + ba \to (b, 1a)}{1 + b + ba \to (b, 1a)}}$$

We can see that in this case the Antimirov parts-of-derivative collectively deliver the Brzozowski derivative: $1 + 1a \doteq 1 + a$.

2.8 Small-Step Operational Semantics

The characteristic feature of small-step operational semantics is that the focus is on describing the individual small steps that are taken during an execution. These small steps represent the execution of instructions (like assignments) but also the evaluation of conditionals. An introduction to the subject can be found in [56].

The meaning of a statement S in state σ is described in terms of a transition system with configurations either of the form $\langle \sigma, S \rangle$ (the statement S is to be executed from state σ) or σ (a terminal configuration). The transitions are given by a relation $\langle \sigma, S \rangle \Rightarrow \gamma$ where γ is either of the form $\langle \sigma', S' \rangle$ or σ'. The transition $\langle \sigma, S \rangle \Rightarrow \gamma$ describes the first step of the execution of S from σ. If $\gamma = \langle \sigma', S' \rangle$, then the execution of S from the state σ has not yet completed and the residual computation is represented by the configuration $\langle \sigma', S' \rangle$. If $\gamma = \sigma'$, then the execution of S from σ has terminated with σ' as the final state.

The definition of $\Rightarrow$ for a simple While language is given by the following rules. We have arithmetic expressions (ranged over by a), Boolean expressions (ranged over by b) and variables (ranged over by x). A state σ is just a mapping of variables to values.

$$\frac{}{\langle \sigma, x := a \rangle \Rightarrow \sigma[x \mapsto \mathscr{A}[\![a]\!]\sigma]}$$

$$\frac{}{\langle \sigma, \mathtt{skip} \rangle \Rightarrow \sigma}$$

$$\frac{\langle \sigma, S_1 \rangle \Rightarrow \langle \sigma', S_1' \rangle}{\langle \sigma, S_1; S_2 \rangle \Rightarrow \langle \sigma', S_1'; S_2 \rangle} \qquad \frac{\langle \sigma, S_1 \rangle \Rightarrow \sigma'}{\langle \sigma, S_1; S_2 \rangle \Rightarrow \langle \sigma', S_2 \rangle}$$

$$\frac{}{\langle \sigma, \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2 \rangle \Rightarrow \langle \sigma, S_1 \rangle}\ \mathscr{B}[\![b]\!]\sigma = \mathsf{tt}$$

$$\frac{}{\langle \sigma, \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2 \rangle \Rightarrow \langle \sigma, S_2 \rangle}\ \mathscr{B}[\![b]\!]\sigma = \mathsf{ff}$$

$$\frac{}{\langle \sigma, \mathtt{while}\ b\ \mathtt{do}\ S \rangle \Rightarrow \langle \sigma, \mathtt{if}\ b\ \mathtt{then}\ (S; \mathtt{while}\ b\ \mathtt{do}\ S)\ \mathtt{else}\ \mathtt{skip} \rangle}$$

For example, the rule for assignment says that executing $x := a$ in state σ terminates in a state which is otherwise like σ except the value of variable x has been updated to be $\mathscr{A}[\![a]\!]\sigma$ (which is the value of the arithmetic expression a in the state σ).

The interesting part for us are the two rules for sequential composition $S_1; S_2$. These say that to execute $S_1; S_2$ from state σ, we first must execute a step of S_1 from state σ. This has two possible outcomes: either $\langle \sigma, S_1 \rangle \Rightarrow \langle \sigma', S_1' \rangle$ or $\langle \sigma, S_1 \rangle \Rightarrow \sigma'$. In the first case we still have the residual program S_1' to execute, but in the second case the execution of S_1 terminated in state σ'. Since these are the only rules for $S_1; S_2$, we cannot execute anything from S_2 until the execution of S_1 has terminated. What we will develop later is precisely about relaxing this restriction, i.e., we will allow (under certain conditions) to execute something from S_2 even when the execution of S_1 has not yet terminated.

A derivation sequence of statement S from state σ is either a finite sequence $\gamma_0, \ldots, \gamma_k$ or an infinite sequence $\gamma_0, \gamma_1, \ldots$ such that $\gamma_0 = \langle \sigma, S \rangle$, γ_k is a terminal configuration, and $\gamma_i \Rightarrow \gamma_{i+1}$. Thus a derivation sequence either describes a path in the transition system from configuration $\langle \sigma, S \rangle$ to a terminal configuration γ_k or it describes an infinite path from the configuration $\langle \sigma, S \rangle$.

We will represent programs in the style of Kleene algebra with tests (KAT) [43], i.e., with regular expressions over some alphabet of instructions. A Kleene algebra with tests is a Kleene algebra where the tests b (a subset of the carrier of the Kleene algebra) form a Boolean algebra with $\bar{b}$ as the complement. Thus for us an assignment $x := a$ will just be

a letter of the alphabet. The statement `skip` will be the expression 1. Sequential composition $S_1;S_2$ will be multiplication E_1E_2 where E_i is the expression representing S_i. Conditionals are represented as a determinised choice. In other words, `if` b `then` S_1 `else` S_2 is represented by $bE_1 + \bar{b}E_2$. (Note that the expression $E_1 + E_2$ is a nondeterministic choice between E_1 and E_2; we take our alphabet to also include tests b together with their complements.) While loops `while` b `do` S are represented as $(bE)^*\bar{b}$ where E is the expression representing S. The expression 0 will represent the program that is stuck or aborted: something that does not lead to a terminal configuration.

Although we said that we will represent programs in the style of Kleene algebra with tests, our intention is not to include upfront all axioms of KAT into our operational semantics (that we will develop in Chapter 5). For any test b, $bb \doteq b$ holds in KAT as conjunction is idempotent in Boolean algebra. Similarly, for any tests b and c, $bc \doteq cb$ holds in KAT. In terms of operational semantics, we view these as potential optimisations that can be applied to programs. Furthermore, the effects of such optimisations may become visible in a concurrent context. For describing relaxed memory models we leave these open as we may want to allow only some of these for a particular memory model.

2.9 Axiomatic Models

The axiomatic style of describing memory models specifies when a given (complete) execution is allowed by the memory model. Basically, the memory model is a predicate on candidate executions which defines those that are allowed (or valid) on that model. Typically this predicate is given in terms of certain relations on memory accesses that occurred during the candidate execution. We now give a brief introduction to this approach. A more thorough introduction can be found in [8].

The instructions that are executed during the execution of a program are represented as abstract events. For example, a write instruction like $x := 1$ could be represented by the event $W(x,1)$ saying that this is a write instruction (W) that writes the value 1 to variable x. A read instruction like $r_1 := y$ would be represented (if y happens to hold the value 2) by the event $R(y,2)$ saying that this event represents a read instruction (R) that reads the value 2 from variable y. These events may also hold some extra information like a processor or event identifiers.

The relations that are defined on these events basically describe two things: control flow and data flow.

The relations for the control flow relate these events to the program that we are considering. For example, there is a relation named *program order* (denoted by po) which relates pairs of events that are from the same thread and it records the order in which the corresponding instructions occur in the program. Thus $(a,b) \in po$ when the instruction represented by a occurs before the instruction represented by b in the program text. Another such would be the *control dependency* relation (denoted by $ctrl$) for which $(a,b) \in ctrl$ when b corresponds to an instruction in a conditional branch where the condition depends on the outcome of the instruction a.

The data flow relations are used to describe how memory events are related in terms of the values that they read and write. For example, in a concurrent program we may have two threads that both write to variable x. The *coherence order* relation (denoted by co) relates write events to the same variable, i.e., $(a,b) \in co$ says that a and b are both write instructions to the same variable and a reaches the memory first. The coherence order relation can be partitioned into coi and coe for internal (events from same thread) and external (events from different threads) coherence order.

Since there may be several write events to a variable, we also need to specify for a

read event where the value (that it reads) came from. The *read-from* relation (denoted by rf) relates a write event and a read event (to the same variable) when the read event reads the value that was written by the write event, i.e., $(a,b) \in rf$ when a is a write event to variable x, b is a read event from variable x and b reads the value that was written by a (meaning that there was no other write to x in between). The rf relation can also be partitioned into *rfi* and *rfe*.

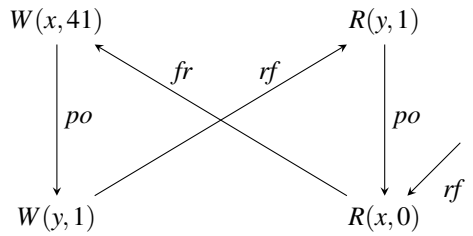
A candidate execution is a set of events together with program order, dependency, read-from and coherence relations. Whether this execution is allowed on a particular memory model is determined by the constraints that the memory model requires from a valid execution. A typical example how this is achieved is by constructing from the basic relations described above a relation called *happens-before* which represents certain invariants of the memory model. A candidate execution is allowed when this happens-before relation does not introduce a cycle on the set of memory events, i.e., the transitive closure of the happens-before relation is irreflexive. Intuitively, a cycle would basically say that some memory event has to happen before itself for this execution to be valid on this memory model.

To determine whether a candidate execution is a valid execution, we just need to check whether the predicate holds on it. If we want to find all possible valid executions of a program on a memory model, then we first have to construct a set of candidate executions that is complete (i.e., contains at least all valid executions) and then filter by the predicate that the memory model requires. A candidate execution is often visualised as a graph where memory events are the vertices and a relation is given by the edges with a certain label. It is then possible to visually check whether the candidate execution is valid by checking whether the happens-before relation forms a cycle.

For the example program we considered in the introduction,

$$x := 41; y := 1 \parallel r_1 := y; r_2 := x$$

one possible candidate execution (as a graph) is the following.



Here we have also used the relation fr. This is derived from rf and co as follows: $(a,b) \in fr$ if there exists a c such that $(c,a) \in rf$ and $(c,b) \in co$. Hence $(a,b) \in fr$ says that a is a read event and it reads its value from a write event that is before b in the coherence order. The dangling rf edge to $R(x,0)$ just says that it reads its value from the initial state. Since the initial state can be seen as a $W(x,0)$ event that is before $W(x,41)$, we then get the fr edge.

Is this candidate execution allowed? If we take the happens-before relation to be $po \cup rf \cup fr \cup co$ and we require it to be acyclic, then it is not a valid execution since there is a cycle. If we take the happens-before relation to be $rf \cup fr \cup co$ (i.e., how the two threads communicate), then there is no cycle and the execution is valid. Excluding program order from the happens-before relation corresponds to allowing the reordering of instructions we considered before and we can see that this is the execution where $r_1 = 1$ and $r_2 = 0$.

3 Reordering Derivatives

In this chapter we introduce *reordering derivatives*. If we consider our overall goal of operational semantics, then the Antimirov reordering derivatives that we define here can be seen as a very limited form of operational semantics. The main limitation or difference compared to usual operational semantics is that here we do not include machine states. Therefore, considering regular expressions to be programs over an alphabet of instructions and the Antimirov reordering derivatives to be operational semantics, in this chapter we do not know *what* a given program computes. Instead, we know *how* the program computes. By this we mean that the Antimirov reordering derivatives give us the possible ways (sequences of instructions; words) how the program may execute. But this precisely tells us how instructions may be reordered during execution, and, as we saw in Chapter 1, such reordering of instructions can be responsible for the relaxed behaviour in some memory models.

Intuitively, the notion of language derivatives that we covered in Section 2.7 is about (strict) prefixes and suffixes of words in a language. What we do here is just change the notion of prefix and suffix—we are interested in prefixes and suffixes of words when considered as representatives of traces.

We define a non-standard interpretation of regular expressions as trace-closed languages. Then we define the reordering language derivatives and generalise the Brzozowski and Antimirov (syntactic) derivative operations to match the non-standard interpretation of regular expressions. As was the case before, these generalisations of Brzozowski and Antimirov derivatives can also be used for constructing an automaton from a regular expression. It is not the case, in general, that the resulting automaton is finite as it must accept a non-regular language in general. We show that, for a class of regular expressions, called *star-connected* expressions, the set of Antimirov reordering derivatives is finite modulo certain equations. This also tells us, as expected, that the non-standard interpretation of star-connected expressions is a regular language.

We also develop a refinement of the Antimirov reordering derivative operation to allows us to more precisely keep track of how an expression is derived along a word. With this refinement in mind, we define a stronger version of *(scattering) rank* [32] which we call *uniform (scattering) rank* and show that languages defined by star-connected expressions have finite uniform scattering rank. Finally, we show that, if an expression defines a language with finite uniform scattering rank, then the refined Antimirov reordering derivative operation can be used to construct a finite automaton from the expression. This construction does not require any quotienting, instead it relies on truncation to make the state set finite.

3.1 Prefixes and Suffixes of Representatives of Traces

As mentioned in our introduction to Mazurkiewicz traces in Section 2.3, when $I = \emptyset$, then the free partially commutative monoid $\Sigma^*/{\sim^{I*}}$ is isomorphic to the free monoid Σ^*. Thus the language derivatives described in Section 2.7 are (implicitly) for the case where $I = \emptyset$. The derivative of a language L along a word u is the set of words v such that $uv \in L$, i.e., u is a prefix and v is a suffix of a word in L (or, some word in L can be factored as uv). Here we are interested in what the (word) prefixes and suffixes of a trace, represented as a word, should be.

We start with a small example. Let $\Sigma =_{\mathrm{df}} \{a, b, c\}$ and $I =_{\mathrm{df}} \emptyset$. We will now look at the word abc, the trace $[abc]^I$ and the corresponding dependence graph $\langle abc \rangle_D$. Since the independence relation is empty, the equivalence class of abc is the singleton set $\{abc\}$.

Equivalently, the dependence graph $\langle abc \rangle_D$ of the trace $[abc]^I$ is a linear order with abc as its sole linearisation. Omitting the transitive edge from a to c, we can draw the graph as follows.

$$a \longrightarrow b \longrightarrow c$$

The word abc can be factored as $abc = uv$ where $u = a$ and $v = bc$. Hence we have $[abc]^I = [a]^I \bullet [bc]^I$ where $\bullet$ denotes multiplication in $\Sigma^*/\sim^{I^*}$. Now, the dependence graphs of u and v can be drawn as follows (where we have drawn boxes around both dependence graph to separate them).

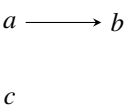
By adding the missing edges from $\langle a \rangle_D$ to $\langle bc \rangle_D$ between the dependent letters, we indeed get back $\langle abc \rangle_D$. The fact that abc can be factored as uv tells us that $\mathsf{D}_a\{abc\} = \{bc\}$. Since equivalence classes are singletons here, we can identify $[abc]^I$ with $\{abc\}$ and $[bc]^I$ with $\{bc\}$. Thus we could also say that $\mathsf{D}_a[abc]^I = [bc]^I$.

Similarly, the word abc can be factored as $abc = uv$ where $u = ab$ and $v = c$. Hence we have $[abc]^I = [ab]^I \bullet [c]^I$ and the dependence graphs of $\langle ab \rangle_D$ and $\langle c \rangle_D$ can be drawn as follows.

Again, we also have that $[abc]^I = [ab]^I \bullet [c]^I$. Since the word abc can be factored as uv, we have that $\mathsf{D}_{ab}\{abc\} = \{c\}$. We could also say that $\mathsf{D}_{ab}[abc]^I = [c]^I$.

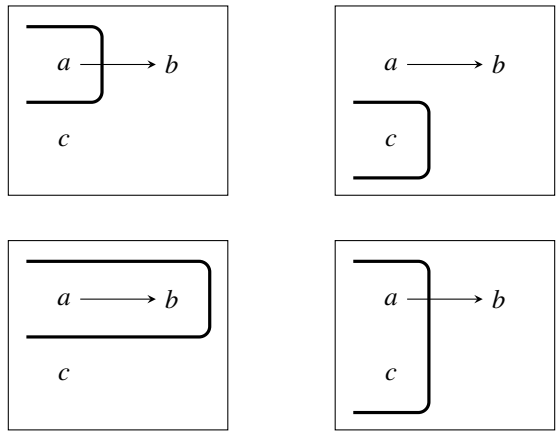
Note that, if $u = b$, then there is no word v such that $uv = abc$, i.e., b is not a prefix of abc. Thus $\mathsf{D}_b\{abc\} = \emptyset$. The same applies for $u = c$.

We now take I to be the least independence relation such that $a\,I\,c$ and $b\,I\,c$. The dependence graph $\langle abc \rangle_D$ according to the new independence relation is the following.



With the new independence relation we also have that $[abc]^I = [a]^I \bullet [bc]^I$, but now the equivalence class corresponding to $[bc]^I$ is $\{bc, cb\}$. We also have $[abc]^I = [c]^I \bullet [ab]^I = [ab]^I \bullet [c]^I$ where $[ab]^I = \{ab\}$. Yet another factorisation is $[abc]^I = [ac]^I \bullet [b]^I$ where $[ac]^I = \{ac, ca\}$. The last factorisation can be drawn as dependence graphs in the following way.

We can see that in all of the above factorisations of $[abc]^I$ as $[u]^I \bullet [v]^I$ the prefix $[u]^I$ corresponds to a downwards-closed subgraph of $\langle abc \rangle_D$, i.e., if a letter a is in the prefix, then so is any D-predecessor of it. This is true in general—a prefix of a trace is a down-set. Here are some examples of prefixes (shown by the thick line) corresponding to the above factorisations.



Corresponding to these pictures, we would like to say, for example, that $\mathsf{D}_a[abc]^I = [bc]^I$ and $\mathsf{D}_c[abc]^I = [ab]^I$. In this chapter, we develop the machinery to do so, operating with words and word languages (representing traces) instead of working with traces or equivalence classes directly. For this reason, we consider the idea of prefixes (and suffixes) up to reordering. In other words, we consider u to be a reordering prefix of z when $[u]^I \bullet [v]^I = [z]^I$ for some (reordering suffix) v. We now develop notation to describe such prefixes on words rather than traces.

For a word vav' such that $v\,I\,a$, we know that a is a minimal letter of vav' and we consider a to be a reordering prefix of vav' with vv' as the corresponding suffix since $vav' \sim^{I*} avv'$ with a as a (strict) prefix and vv' as the suffix. For the same reason we consider u to be a reordering prefix of vuv' when $v\,I\,u$. If $u' \sim^{I*} u$ and $v\,I\,u$, then $vuv' \sim^{I*} uvv' \sim^{I*} u'vv'$. Thus we also consider u' to be a reordering prefix of vuv'. Note that, if a is a reordering prefix of z, then, by irreflexivity of I, this a is the first a of z. We can also scale this idea further to the case $vuv'u'v''$ where $v\,I\,u$ and $vv'\,I\,u'$. We then have $vuv'u'v'' \sim^{I*} uu'vv'v''$.

We now make precise the idea described in the previous paragraph. We call this I-scattering. This is just a way to describe a selection of letter occurrences of u in z subject to certain constraints.

Definition 3.1. For all $n \in \mathbb{N}, u_1, \ldots, u_n \in \Sigma^+, v_0 \in \Sigma^*, v_1, \ldots, v_{n-1} \in \Sigma^+, v_n \in \Sigma^*, z \in \Sigma^*$,

$$u_1, \ldots, u_n \lhd z \rhd v_0, \ldots, v_n \quad =_{\mathrm{df}} \quad z = v_0 u_1 v_1 \ldots u_n v_n \wedge \forall i. \forall j < i. v_j\,I\,u_i.$$

We also say that the word $u_1 \ldots u_n$ can be (strictly) scattered in z (according to I and with degree n) as $z = v_0 u_1 v_1 \ldots u_n v_n$. Note that only v_0 and v_n are allowed to be the empty word. An important consequence of the above definition, reflecting its prefix-suffix aspect, is the following lemma.

Lemma 3.2. *For all $n \in \mathbb{N}, u_1, \ldots, u_n \in \Sigma^+, v_0 \in \Sigma^*, v_1, \ldots, v_{n-1} \in \Sigma^+, v_n \in \Sigma^*, z \in \Sigma^*$, if $u_1, \ldots, u_n \lhd z \rhd v_0, \ldots, v_n$, then $z \sim^{I*} u_1 \ldots u_n v_0 \ldots v_n$.*

Thus, if we have $u_1, \ldots, u_n \lhd z \rhd v_0, \ldots, v_n$, then we know that the word $u_1 \ldots u_n$ is a prefix of z when considered as traces.

We often use underlined letters as the notation to visualise the scattering of u in z. More precisely, we underline the factors u_i in z. Continuing with the independence alphabet from above where $\Sigma =_{\mathrm{df}} \{a, b, c\}$ and $a\,I\,c$, $b\,I\,c$, we have that $c\underline{a}c\underline{b}c$ is a valid scattering (with $u_1 = a$, $u_2 = b$ and $v_0 = v_1 = v_2 = c$) since $c\,I\,a$ and $cc\,I\,b$. The scattering $c\underline{acbc}$ is not valid since here $v_0 = c$ and $u_1 = acb$, but $c\,D\,c$. Similarly, the scattering $cac\underline{b}c$ is not valid because here $v_0 = cac$ and $u_1 = b$, but $a\,D\,b$.

Definition 3.3. For all $u, v, z \in \Sigma^*$,

1. $u \lhd z \rhd v \quad =_{\text{df}} \quad \exists n \in \mathbb{N}, u_1, \ldots, u_n, v_0, \ldots, v_n . u = u_1 \ldots u_n \wedge v = v_0 \ldots v_n \wedge u_1, \ldots, u_n \lhd z \rhd v_0, \ldots, v_n;$
2. $u \sim \lhd z \rhd v \quad =_{\text{df}} \quad \exists u' . u \sim^{I*} u' \wedge u' \lhd z \rhd v;$
3. $u \sim \lhd z \rhd \sim v \quad =_{\text{df}} \quad \exists u', v' . u \sim^{I*} u' \wedge u' \lhd z \rhd v' \wedge v' \sim^{I*} v.$

In all three cases, we talk about u being a prefix and v being a suffix of z, up to reordering. We also say that u is scattered in z with the residual (unselected letters) v. The difference of the three lies in the reordering of the letters of u and v. In the first case, the letters of u (resp. v) must appear in the same order in z as they do in u (resp. v). In the second case, we allow for the letters of u to be scattered (or to occur) in z according to an equivalent word u', i.e., the letters of u can be scattered in a reordered fashion. In the third case, we also allow to reorder v.

Continuing with the same independence alphabet from above, take $z =_{\text{df}} ccabac$. Then we have $ca \lhd z \rhd cbac$ since $c, a \lhd z \rhd \varepsilon, c, bac$, i.e., $z = \varepsilon \underline{c} c \underline{a} bac$. We also have $ac \sim \lhd z \rhd cbac$, justified by the same scattering, since $ac \sim^{I*} ca$. (We do not have $ac \lhd z \rhd cbac$ as we have to preserve the ordering of letters in the prefix part in this case and thus the only candidates are $cc\underline{a}ba\underline{c}$ and $ccaba\underline{ac}$, but neither of those is a valid scattering and the suffix is not $cbac$.) By reordering the letters of the suffix, we also have $ac \sim \lhd z \rhd \sim bacc$, witnessed by the same scattering, since $cbac \sim^{I*} bacc$.

A useful property of scatterings is that (a prefix) u and (a suffix) v can be scattered in z in at most one way. In other words, if $u \lhd z \rhd v$, then the number n and the words u_i and v_j are uniquely determined. Furthermore, equivalent words result in the same scattering.

Lemma 3.4. *For any $u, v, z \in \Sigma^*$,*

1. $u \lhd z \rhd v \quad \iff \quad \exists! n \in \mathbb{N}, u_1, \ldots, u_n, v_0, \ldots, v_n . u = u_1 \ldots u_n \wedge v = v_0 \ldots v_n \wedge u_1, \ldots, u_n \lhd z \rhd v_0, \ldots, v_n;$
2. $u \sim \lhd z \rhd v \quad \iff \quad \exists! u' . u \sim^{I*} u' \wedge u' \lhd z \rhd v;$
3. $u \sim \lhd z \rhd \sim v \quad \iff \quad \exists! u', v' . u \sim^{I*} u' \wedge u' \lhd z \rhd v' \wedge v' \sim^{I*} v.$

We also mention that, if u is a reordering prefix of z, i.e., $u \sim \lhd z \rhd v$ for some v, then the corresponding scattering is obtained by underlining (one by one) the minimal occurrences of the letters of u in z.

Later, we will be interested in the degree n of scattering (the number of u_i blocks in z). Thus we also define degree-bounded versions of scattering, i.e., we allow at most N underlined subwords in z. These become relevant in Section 3.4.

Definition 3.5. For all $u, v, z \in \Sigma^*$ and $N \in \mathbb{N}$,

1. $u \lhd_N z \rhd v \quad =_{\text{df}} \quad \exists n \leq N, u_1, \ldots, u_n, v_0, \ldots, v_n . u_1, \ldots, u_n \lhd z \rhd v_0, \ldots, v_n;$
2. $u \sim \lhd_N z \rhd v \quad =_{\text{df}} \quad \exists u' . u \sim^{I*} u' \wedge u' \lhd_N z \rhd v;$
3. $u \sim \lhd_N z \rhd \sim v \quad =_{\text{df}} \quad \exists u', v' . u \sim^{I*} u' \wedge u' \lhd_N z \rhd v' \wedge v' \sim^{I*} v.$

Finally, we have that scattering (as described above) together with reordering both in the prefix and suffix corresponds to trace-prefix and trace-suffix.

Proposition 3.6. *For all* $u, v, z \in \Sigma^*, uv \sim^{I*} z \iff u \sim\!\lhd z \rhd\!\sim v$.

Proof.

- $\Longleftarrow$: Since $u \sim\!\lhd z \rhd\!\sim v$, there exist u' and v' such that $u \sim^{I*} u'$, $v' \sim^{I*} v$ and $u' \lhd z \rhd v'$. By Lemma 3.2, we have $u'v' \sim^{I*} z$ and thus $uv \sim^{I*} z$.
- $\Longrightarrow$: By induction on z.
  - Case $z = \varepsilon$: It must be that both $u = \varepsilon$ and $v = \varepsilon$. We have $\varepsilon \sim\!\lhd \varepsilon \rhd\!\sim \varepsilon$.
  - Case $z = az'$: We have $uv \sim^{I*} az'$. The first (occurrence of) a in uv is either in u or in v.
    If $a \in u$, then exist u_l, u_r such that $u = u_l a u_r$ and $u_l \, I \, a$. Thus $(u_l u_r)v \sim^{I*} z'$ and by i.h. we have $u_l u_r \sim\!\lhd z' \rhd\!\sim v$. Thus $a(u_l u_r) \sim\!\lhd az' \rhd\!\sim v$ is a valid scattering and we get $u \sim\!\lhd z \rhd\!\sim v$.
    If $a \in v$, then exist v_l, v_r such that $v = v_l a v_r$ and $uv_l \, I \, a$. Thus $u(v_l v_r) \sim^{I*} z'$ and by i.h. we have $u \sim\!\lhd z' \rhd\!\sim v_l v_r$. Since $u \, I \, a$, we have that $u \sim\!\lhd az' \rhd\!\sim a(v_l v_r)$ is a valid scattering and thus $u \sim\!\lhd z \rhd\!\sim v$. □

To emphasise, the $\Longleftarrow$ direction also holds when we consider reordering only in the prefix ($u \sim\!\lhd z \rhd v$) or no reordering at all ($u \lhd z \rhd v$).

3.2 Trace-Closing Semantics of Regular Expressions

We now define a non-standard word-language semantics of regular expressions that directly interprets an expression E as the trace closure $[\![E]\!]^I$ of its standard word-language interpretation $[\![E]\!]$.

We have already noted that $[\{a\}]^I = \{a\}$, $[\emptyset]^I = \emptyset$, $[L \cup L']^I = [L]^I \cup [L']^I$ and $[\mathbf{1}]^I = \mathbf{1}$. Crucially, for general I, we do not have $[L \cdot L']^I = [L]^I \cdot [L']^I$. For example, with $\Sigma =_{\text{df}} \{a, b\}$ and $a \, I \, b$, we thus have $[\{a\}]^I = \{a\}$, $[\{b\}]^I = \{b\}$ whereas $[\{ab\}]^I = \{ab, ba\} \neq \{ab\} = [\{a\}]^I \cdot [\{b\}]^I$. Hence we need a different concatenation operation, one that would concatenate $\{a\}$ and $\{b\}$ as $\{ab, ba\}$ when $a \, I \, b$.

Definition 3.7.

1. The *I-reordering concatenation of words* $\cdot^I : \Sigma^* \times \Sigma^* \to \mathscr{P}(\Sigma^*)$ is defined by

$$\begin{array}{rcl} \varepsilon \cdot^I v & =_{\text{df}} & \{v\} \\ u \cdot^I \varepsilon & =_{\text{df}} & \{u\} \\ au \cdot^I bv & =_{\text{df}} & \{a\} \cdot (u \cdot^I bv) \cup \{b \mid au \, I \, b\} \cdot (au \cdot^I v) \end{array}$$

2. The lifting of I-reordering concatenation to *languages* is defined by

$$L \cdot^I L' =_{\text{df}} \bigcup \{u \cdot^I v \mid u \in L \wedge v \in L'\}$$

Note that $\{b \mid au \, I \, b\}$ acts as a test: it is either $\emptyset$ or $\{b\}$. (We can also consider $\{a\} \cdot (u \cdot^I bv)$ to include a trivial test, i.e., $\{a \mid \mathrm{tt}\} \cdot (u \cdot^I bv)$.) This makes the definition of $au \cdot^I bv$ biased towards its left argument: a can always occur as the first letter of a word in $au \cdot^I bv$, but for b we require that $au \, I \, b$.

Example 3.8. Let $\Sigma =_{\text{df}} \{a, b\}$ and $a \, I \, b$. Then $a \cdot^I b = \{ab, ba\}$, $aa \cdot^I b = \{aab, aba, baa\}$, $a \cdot^I bb = \{abb, bab, bba\}$ and $ab \cdot^I ba = \{abba\}$. The last example shows that although I-reordering concatenation is defined quite similarly to the shuffle product of words, it is different. For example, we have $baba \in ab \sqcup\!\sqcup ba$.

An important property of the I-reordering concatenation $u \cdot^I v$, as demonstrated by the following proposition, is that the reordering occurs at the "boundary" between u and v but not within the words u and v, i.e., the ordering of letters (letter occurrences) in u and v is preserved.

Proposition 3.9. *For any* $u, v, z \in \Sigma^*$, $z \in u \cdot^I v \iff u \lhd z \rhd v$.

Proof.

- $\Longrightarrow$: By induction on z.
  - Case $z = \varepsilon$: Then it must be that $u = \varepsilon$ and $v = \varepsilon$. We have $\varepsilon \lhd \varepsilon \rhd \varepsilon$.
  - Case $z = az'$: If $u = au'$, then $z' \in u' \cdot^I v$, and, by i.h. we have $u' \lhd z' \rhd v$. Thus $au' \lhd az' \rhd v$ is a valid scattering.

    Otherwise, it must be that $v = av'$ and $u \, I \, a$. We have $z' \in u \cdot^I v'$, and, by i.h. we have $u \lhd z' \rhd v'$. Since $u \, I \, a$, we have that $u \lhd az' \rhd av'$ is a valid scattering.
- $\Longleftarrow$: By induction on z.
  - Case $z = \varepsilon$: Then it must be that $u = \varepsilon$ and $v = \varepsilon$. We have $\varepsilon \in \varepsilon \cdot^I \varepsilon$.
  - Case $z = az'$: If $u = au'$, then $u' \lhd z' \rhd v$, and, by i.h. we have $z' \in u' \cdot^I v$. Thus also $az' \in au' \cdot^I v$.

    Otherwise, it must be that $v = av'$ and $u \, I \, a$. We have $u \lhd z' \rhd v'$, and, by i.h. we have $z' \in u \cdot^I v'$. Since $u \, I \, a$, we also have $az' \in u \cdot^I av'$. □

The I-reordering concatenation of closures of languages computes the closure of the ordinary concatenation of the languages. In comparison to the previous proposition, here we first reorder "inside" u and v and then do the reordering concatenation.

Proposition 3.10. *For any languages* L *and* L', $[L \cdot L']^I = [L]^I \cdot^I [L']^I$.

Proof. For any $w \in \Sigma^*$,

$$\begin{aligned}
w \in [L \cdot L']^I &\iff \exists u \in L, v \in L' . \, w \sim^{I*} uv \\
&\iff \exists u \in L, v \in L' . \, u \sim \lhd \, w \, \rhd \sim v \\
&\iff \exists u \in L, v \in L', u', v' \in \Sigma^* . \, u \sim^{I*} u' \wedge v \sim^{I*} v' \wedge u' \lhd w \rhd v' \\
&\iff \exists u \in L, v \in L', u', v' \in \Sigma^* . \, u \sim^{I*} u' \wedge v \sim^{I*} v' \wedge w \in u' \cdot^I v' \\
&\iff \exists u' \in [L]^I, v' \in [L']^I . \, w \in u' \cdot^I v' \\
&\iff w \in [L]^I \cdot^I [L']^I .
\end{aligned}$$

□

The above proposition also tells us that the closure $[\_]^I$ is a monoid morphism from $(\mathscr{P}(\Sigma^*), \cdot, \mathbf{1})$ to $(\{L \subseteq \Sigma^* \mid L = [L]^I\}, \cdot^I, \mathbf{1})$, i.e., from word languages to trace-closed word languages.

Evidently, if $I = \emptyset$, then the reordering concatenation coincides with the ordinary concatenation: $u \cdot^\emptyset v = \{uv\}$ and $L \cdot^\emptyset L' = L \cdot L'$. For $I = \Sigma \times \Sigma$, which is forbidden in independence alphabets, as I is required to be irreflexive, it is shuffle: $u \cdot^{\Sigma \times \Sigma} v = u \sqcup\!\sqcup v$. For general

I, it has properties similar to concatenation. In particular, we have the following.

$$\begin{array}{rcl}(L \cdot^I L') \cdot^I L'' & = & L \cdot^I (L' \cdot^I L'') \\ \mathbf{1} \cdot^I L & = & L \\ L \cdot^I \mathbf{1} & = & L \\ \emptyset \cdot^I L & = & \emptyset \\ L \cdot^I \emptyset & = & \emptyset \\ L \cdot^I (L' \cup L'') & = & L \cdot^I L' \cup L \cdot^I L'' \\ (L' \cup L'') \cdot^I L & = & L' \cdot^I L \cup L'' \cdot^I L\end{array}$$

We also have the following weak interchange law familiar from the concurrent Kleene algebra theory introduced in [33].

$$(L_1 \sqcup L_2) \cdot^I (L_1' \sqcup L_2') \quad \subseteq \quad (L_1 \cdot^I L_1') \sqcup (L_2 \cdot^I L_2')$$

We now have the necessary ingredients to introduce the non-standard semantics of regular expressions.

Definition 3.11. The *trace-closing semantics* $[\![\_]\!]^I : \mathrm{RE} \to \mathscr{P}(\Sigma^*)$ of regular expressions is defined recursively by

$$\begin{array}{rcl}[\![a]\!]^I & =_{\mathrm{df}} & \{a\} \\ [\![0]\!]^I & =_{\mathrm{df}} & \emptyset \\ [\![E+F]\!]^I & =_{\mathrm{df}} & [\![E]\!]^I \cup [\![F]\!]^I \\ [\![1]\!]^I & =_{\mathrm{df}} & \mathbf{1} \\ [\![EF]\!]^I & =_{\mathrm{df}} & [\![E]\!]^I \cdot^I [\![F]\!]^I \\ [\![E^*]\!]^I & =_{\mathrm{df}} & \mu X.\, \mathbf{1} \cup [\![E]\!]^I \cdot^I X\end{array}$$

Compared to the standard semantics of regular expressions, the difference is in the handling of the EF case (and consequently also the E^* case) due to the cross-commutation that happens in concatenation of traces and must be accounted for by $\cdot^I$.

With $I = \emptyset$, we fall back to the standard interpretation of regular expressions: $[\![E]\!]^\emptyset = [\![E]\!]$. For I a general independence relation, we obtain the desired property that the semantics delivers the trace closure of the language of the expression.

Proposition 3.12. *For any* E, $[\![E]\!]^I = [[\![E]\!]]^I$.

Proof. By induction on E.

- Case a: $[\![a]\!]^I = \{a\} = [\{a\}]^I = [[\![a]\!]]^I$.
- Case 0: $[\![0]\!]^I = \emptyset = [\emptyset]^I = [[\![0]\!]]^I$.
- Case $E+F$: By i.h. we have $[\![E]\!]^I = [[\![E]\!]]^I$ and $[\![F]\!]^I = [[\![F]\!]]^I$. Thus,

$$[\![E+F]\!]^I = [\![E]\!]^I \cup [\![F]\!]^I = [[\![E]\!]]^I \cup [[\![F]\!]]^I = [[\![E]\!] \cup [\![F]\!]]^I = [[\![E+F]\!]]^I.$$

- Case 1: $[\![1]\!]^I = \mathbf{1} = [\mathbf{1}]^I = [[\![1]\!]]^I$.
- Case EF: By i.h. we have $[\![E]\!]^I = [[\![E]\!]]^I$ and $[\![F]\!]^I = [[\![F]\!]]^I$. Thus, by Proposition 3.10, we have

$$[\![EF]\!]^I = [\![E]\!]^I \cdot^I [\![F]\!]^I = [[\![E]\!]]^I \cdot^I [[\![F]\!]]^I = [[\![E]\!] \cdot [\![F]\!]]^I = [[\![EF]\!]]^I.$$

- Case E^*: By i.h. we have $[\![E]\!]^I = [[\![E]\!]]^I$.

- $[\![E^*]\!]^I \subseteq [[\![E^*]\!]]^I$: Let $w \in [\![E^*]\!]^I$. By definition of $[\![E^*]\!]^I$, there exists $n \in \mathbb{N}$ such that w is a word of the n-th $\cdot^I$ power of $[\![E]\!]^I$, i.e., $w \in [\![E]\!]^I \cdot^I \ldots \cdot^I [\![E]\!]^I$. By i.h. we have $w \in [[\![E]\!]]^I \cdot^I \ldots \cdot^I [[\![E]\!]]^I$. By repeated application of Proposition 3.10 we get $w \in [[\![E]\!] \cdot \ldots \cdot [\![E]\!]]^I \subseteq [[\![E^*]\!]]^I$.
- $[[\![E^*]\!]]^I \subseteq [\![E^*]\!]^I$: Let $w \in [[\![E^*]\!]]^I$. By definition of $[\![E^*]\!]$, there exists $n \in \mathbb{N}$ such that w is in the closure of the n-th power of $[\![E]\!]$, i.e., $w \in [[\![E]\!] \cdot \ldots \cdot [\![E]\!]]^I$. By repeated application of Proposition 3.10 we have $w \in [[\![E]\!]]^I \cdot^I \ldots \cdot^I [[\![E]\!]]^I$. By i.h. we get $w \in [\![E]\!]^I \cdot^I \ldots \cdot^I [\![E]\!]^I \subseteq [\![E^*]\!]^I$. □

3.3 Reordering Derivatives

We are now ready to generalise the Brzozowski and Antimirov constructions to trace closures of regular languages. To this end, we switch to what we call reordering derivatives.

3.3.1 Reordering Derivative of a Language

Let (Σ, I) be a fixed independence alphabet. We generalise the concepts of (semantic) nullability and derivative of a language to reorderable part and reordering derivative of a language.

Definition 3.13. The *I-reorderable part* of a language L wrt. a word u is defined as

$$\mathsf{R}^I_u L =_{\mathrm{df}} \{v \in L \mid v \, I \, u\}$$

and the *I-reordering derivative* along u is defined as

$$\mathsf{D}^I_u L =_{\mathrm{df}} \{v \in \Sigma^* \mid \exists z \in L. \, u \sim_{\lhd} z \rhd v\}.$$

By Proposition 3.9, we can equivalently say that $\mathsf{D}^I_u L = \{v \mid \exists z \in L. z \in [u]^I \cdot^I v\}$. For a single-letter word a, we get $\mathsf{D}^I_a L = \{v_l v_r \mid v_l a v_r \in L \wedge v_l \, I \, a\} = \{v \mid \exists z \in L. z \in a \cdot^I v\}$. That is, we require some reordering of u (resp. a) to be a prefix, up to reordering, of some word z in L with v as the corresponding suffix. (In other words, we allow reordering of letters within u and across u and v, but not within v.)

Example 3.14. Let $\Sigma =_{\mathrm{df}} \{a, b, c\}$ and $a \, I \, b$.

Since $\varepsilon \, I \, b$ and $a \, I \, b$, but $ab \, D \, b$, we have $\mathsf{R}^I_b\{\varepsilon, a, ab\} = \{\varepsilon, a\}$. Another observation is that R^I_b and R^I_{bb} give the same result, i.e., $\mathsf{R}^I_{bb}\{\varepsilon, a, ab\} = \{\varepsilon, a\}$.

For the derivative we consider the language $L =_{\mathrm{df}} \{abacab\}$. What is $\mathsf{D}^I_b L$? By definition, it is the set of words v such that $b \sim_{\lhd} abacab \rhd v$, i.e., for each such v there must exist a u such that $b \sim^{I*} u$ and $u \lhd abacab \rhd v$. In this case $u = b$ and b can be scattered in $abacab$ as $a\underline{b}acab$. Since scatterings are unique, this is the only valid scattering and thus $\mathsf{D}^I_b L = \{aacab\}$ (while $\mathsf{D}_b L = \emptyset$). We can see that $\mathsf{D}^I_c L = \emptyset$ since there is no valid scattering of c in $abacab$ (since both $a \, D \, c$ and $b \, D \, c$). We do have $\mathsf{D}^I_{aabc} L = \{ab\}$ witnessed by the scattering $\underline{abac}ab$.

In the special case $I = \emptyset$, we have $\mathsf{R}^\emptyset_\varepsilon L = L$, $\mathsf{R}^\emptyset_u L = \{\varepsilon \mid L^{\not\varepsilon}_{\not u}\}$ for any $u \neq \varepsilon$, and $\mathsf{D}^\emptyset_u L = \mathsf{D}_u L$. In the general case, the reorderable part and reordering derivative enjoy the following properties.

Lemma 3.15. *For any languages L, L' and for any $u \in \Sigma^*$, if $L \subseteq L'$, then $\mathsf{R}^I_u L \subseteq \mathsf{R}^I_u L'$ and $\mathsf{D}^I_u L \subseteq \mathsf{D}^I_u L'$.*

Lemma 3.16. *For every L,*

1. $\mathsf{R}^I_\varepsilon L = L$; *for every* $u, v \in \Sigma^*$, $\mathsf{R}^I_v(\mathsf{R}^I_u L) = \mathsf{R}^I_{uv} L$;
2. *for every* $u, u' \in \Sigma^*$, *if* $\Sigma(u) = \Sigma(u')$, *then* $\mathsf{R}^I_u L = \mathsf{R}^I_{u'} L$.

Thus, reorderable part is a (right) monoid action and, furthermore, the reorderable part $\mathsf{R}^I_u L$ is determined by $\Sigma(u)$, the set of letters that occur in u. We will later also need to extend R^I to subsets of Σ: by $\mathsf{R}^I_X L$, we mean $\mathsf{R}^I_u L$ where u is any enumeration of X.

Lemma 3.17. *For every L,*

1. $\mathsf{D}^I_\varepsilon L = L$; *for any* $u, v \in \Sigma^*$, $\mathsf{D}^I_v(\mathsf{D}^I_u L) = \mathsf{D}^I_{uv} L$;
2. *for any* $u, u' \in \Sigma^*$ *such that* $u \sim^{I*} u'$, *we have* $\mathsf{D}^I_u L = \mathsf{D}^I_{u'} L$.

Thus, reordering derivative is also a (right) monoid action. Moreover, it is a trace monoid action.

Proposition 3.18. *For every L,*

1. *for any* $u \in \Sigma^*$, $\mathsf{D}_u([L]^I) = [\mathsf{D}^I_u L]^I$;
   if L is closed (i.e., $[L]^I = L$), then, for any $u \in \Sigma^*$, $\mathsf{D}^I_u L$ *is closed and* $\mathsf{D}_u L = \mathsf{D}^I_u L$;
2. *for any* $u, v \in \Sigma^*$, $uv \in [L]^I$ *iff* $v \in [\mathsf{D}^I_u L]^I$;
3. *for any* $u \in \Sigma^*$, $u \in [L]^I$ *iff* $(\mathsf{D}^I_u L)\lightning$;
4. $[L]^I = \{\varepsilon \mid L\lightning\} \cup \bigcup_{a \in \Sigma} \{a\} \cdot [\mathsf{D}^I_a L]^I$.

Proof. We show (1).

$$\begin{array}{rcl} \mathsf{D}_u[L]^I & = & \{v \in \Sigma^* \mid \exists z \in L.\, uv \sim^{I*} z\} \\ & = & \{v \in \Sigma^* \mid \exists z \in L.\, u \sim \lhd z \rhd \sim v\} \\ & = & [\{v \in \Sigma^* \mid \exists z \in L.\, u \sim \lhd z \rhd v\}]^I \\ & = & [\mathsf{D}^I_u L]^I \end{array}$$

□

Example 3.19. Let $\Sigma =_{\mathrm{df}} \{a, b\}$ and $a\, I\, b$. Take L to be the regular language $[\![(ab)^*]\!]$. We already noted (in Proposition 2.1) that the language

$$[L]^I = \{u \in \Sigma^* \mid |u|_a = |u|_b\}$$

is not regular. For any $n \in \mathbb{N}$,

$$\mathsf{D}^I_{b^n} L = \{a^n\} \cdot L = [\![a^n (ab)^*]\!]$$

whereas

$$\mathsf{D}_{b^n}([L]^I) = \{a^n\} \cdot^I [L]^I = \{u \in \Sigma^* \mid |u|_a = |u|_b + n\}.$$

We can see that $[L]^I$ has infinitely many derivatives, none of which are regular, and L has infinitely many reordering derivatives, all regular.

Example 3.20. In Example 2.13 we saw that for $\Sigma =_{\mathrm{df}} \{a, b\}$,

$$E =_{\mathrm{df}} 1 + a(1 + b(1 + a)) + b(1 + a(1 + b))$$

and $L =_{\mathrm{df}} [\![E]\!]$ we have $\mathsf{D}_a L = \{\varepsilon, b, ba\}$ and $\mathsf{D}_b L = \{\varepsilon, a, ab\}$. We now take $a\, I\, b$ and this results in $\mathsf{D}^I_a L = \{\varepsilon, b, ba, bb\}$ and $\mathsf{D}^I_b L = \{\varepsilon, a, ab, aa\}$.

We can see that $L = \{\varepsilon, a, ab, aba, b, ba, bab\}$. Now, $b \in \mathsf{D}^I_a L$ is witnessed by both $\underline{a}b \in L$ and $b\underline{a} \in L$; $bb \in \mathsf{D}^I_a L$ is witnessed only by $b\underline{a}b \in L$. Similarly, $a \in \mathsf{D}^I_b L$ is witnessed by both $a\underline{b} \in L$ and $\underline{b}a \in L$; $aa \in \mathsf{D}^I_b$ is witnessed by $a\underline{b}a \in L$.

3.3.2 Brzozowski Reordering Derivative

The reorderable parts and reordering derivatives of regular languages turn out to be regular. We now show that they can be computed syntactically, generalising the classical syntactic nullability and Brzozowski derivative operations [20].

Definition 3.21. The *I-reorderable part* and the *Brzozowski I-reordering derivative* of an expression are given by functions $\mathsf{R}^I, \mathsf{D}^I : \mathsf{RE} \times \Sigma \to \mathsf{RE}$ and $\mathsf{R}^I, \mathsf{D}^I : \mathsf{RE} \times \Sigma^* \to \mathsf{RE}$ defined recursively by

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{R}^I_a b &=_{\mathrm{df}}& \text{if } a\,I\,b \text{ then } b \text{ else } 0 & \mathsf{D}^I_a b &=_{\mathrm{df}}& \text{if } a = b \text{ then } 1 \text{ else } 0 \\
\mathsf{R}^I_a 0 &=_{\mathrm{df}}& 0 & \mathsf{D}^I_a 0 &=_{\mathrm{df}}& 0 \\
\mathsf{R}^I_a(E+F) &=_{\mathrm{df}}& \mathsf{R}^I_a E + \mathsf{R}^I_a F & \mathsf{D}^I_a(E+F) &=_{\mathrm{df}}& \mathsf{D}^I_a E + \mathsf{D}^I_a F \\
\mathsf{R}^I_a 1 &=_{\mathrm{df}}& 1 & \mathsf{D}^I_a 1 &=_{\mathrm{df}}& 0 \\
\mathsf{R}^I_a(EF) &=_{\mathrm{df}}& (\mathsf{R}^I_a E)(\mathsf{R}^I_a F) & \mathsf{D}^I_a(EF) &=_{\mathrm{df}}& (\mathsf{D}^I_a E)F + (\mathsf{R}^I_a E)(\mathsf{D}^I_a F) \\
\mathsf{R}^I_a(E^*) &=_{\mathrm{df}}& (\mathsf{R}^I_a E)^* & \mathsf{D}^I_a(E^*) &=_{\mathrm{df}}& (\mathsf{R}^I_a E)^*(\mathsf{D}^I_a E)E^* \\
\\
\mathsf{R}^I_\varepsilon E &=_{\mathrm{df}}& E & \mathsf{D}^I_\varepsilon E &=_{\mathrm{df}}& E \\
\mathsf{R}^I_{ua} E &=_{\mathrm{df}}& \mathsf{R}^I_a(\mathsf{R}^I_u E) & \mathsf{D}^I_{ua} E &=_{\mathrm{df}}& \mathsf{D}^I_a(\mathsf{D}^I_u E)
\end{array}
$$

The expression $\mathsf{R}^I_u E$ is just E with all occurrences of letters dependent with u replaced with 0. The definition of D^I is more interesting. Compared to the classical Brzozowski derivative, the nullability condition $E\!\downarrow$ in the EF case has been replaced with multiplication with the reorderable part $\mathsf{R}^I_a E$, and the E^* case has also been adjusted accordingly.

Example 3.22. Let $\Sigma =_{\mathrm{df}} \{a,b,c\}$, $a\,I\,c$, $b\,I\,c$ and $E =_{\mathrm{df}} a + ba + ca$. We have the following.

$$
\begin{array}{rcl}
\mathsf{R}^I_a E &=& R^I_a a + \mathsf{R}^I_a(ba) + \mathsf{R}^I_a(ca) \\
&=& 0 + (\mathsf{R}^I_a b)(\mathsf{R}^I_a a) + (\mathsf{R}^I_a c)(\mathsf{R}^I_a a) \\
&=& 0 + 00 + c0 \\
&\doteq& 0
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{R}^I_c E &=& R^I_c a + \mathsf{R}^I_c(ba) + \mathsf{R}^I_c(ca) \\
&=& a + (\mathsf{R}^I_c b)(\mathsf{R}^I_c a) + (\mathsf{R}^I_c c)(\mathsf{R}^I_c a) \\
&=& a + ba + 0a \\
&\doteq& a + ba
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{D}^I_a E &=& \mathsf{D}^I_a a + \mathsf{D}^I_a(ba) + \mathsf{D}^I_a(ca) \\
&=& 1 + ((\mathsf{D}^I_a b)a + (R^I_a b)(\mathsf{D}^I_a a)) + ((\mathsf{D}^I_a c)a + (\mathsf{R}^I_a c)(\mathsf{D}^I_a a)) \\
&=& 1 + (0a + 01) + (0a + c1) \\
&\doteq& 1 + c
\end{array}
$$

Note that without reordering (equivalent to taking $I = \emptyset$) we have $\mathsf{D}_a E \doteq 1$.

The functions R^I and D^I on expressions compute their semantic counterparts on the corresponding regular languages.

Proposition 3.23. *For any E,*

1. *for any $a \in \Sigma$, $\mathsf{R}^I_a[\![E]\!] = [\![\mathsf{R}^I_a E]\!]$ and $\mathsf{D}^I_a[\![E]\!] = [\![\mathsf{D}^I_a E]\!]$;*
2. *for any $u \in \Sigma^*$, $\mathsf{R}^I_u[\![E]\!] = [\![\mathsf{R}^I_u E]\!]$ and $\mathsf{D}^I_u[\![E]\!] = [\![\mathsf{D}^I_u E]\!]$.*

Proof.

1. Both claims by induction on E. We only show a few selected cases. First, we consider $\mathsf{R}^I_a[\![E]\!] = [\![\mathsf{R}^I_a E]\!]$.

   - Case b: If $a\,I\,b$, then we have

     $$\mathsf{R}^I_a[\![b]\!] = \{z \in \{b\} \mid z\,I\,a\} = \{b\} = [\![b]\!] = [\![\mathsf{R}^I_a b]\!].$$

     If $a\,D\,b$, then we have

     $$\mathsf{R}^I_a[\![b]\!] = \{z \in \{b\} \mid z\,I\,a\} = \emptyset = [\![0]\!] = [\![\mathsf{R}^I_a b]\!].$$

   - Case EF: By i.h. we have $\mathsf{R}^I_a[\![E]\!] = [\![\mathsf{R}^I_a E]\!]$ and $\mathsf{R}^I_a[\![F]\!] = [\![\mathsf{R}^I_a F]\!]$. We have

     $$\begin{array}{rcl}
     \mathsf{R}^I_a[\![EF]\!] & = & \{z \in \Sigma^* \mid z \in [\![EF]\!] \wedge z\,I\,a\} \\
     & = & \{z_e z_f \mid z_e \in [\![E]\!] \wedge z_f \in [\![F]\!] \wedge z_e\,I\,a \wedge z_f\,I\,a\} \\
     & = & \{z_e z_f \mid z_e \in \mathsf{R}^I_a[\![E]\!] \wedge z_f \in \mathsf{R}^I_a[\![F]\!]\} \\
     & = & (\mathsf{R}^I_a[\![E]\!]) \cdot (\mathsf{R}^I_a[\![F]\!]) \\
     & = & [\![\mathsf{R}^I_a E]\!] \cdot [\![\mathsf{R}^I_a F]\!] \\
     & = & [\![\mathsf{R}^I_a (EF)]\!].
     \end{array}$$

   Next, we consider $\mathsf{D}^I_a[\![E]\!] = [\![\mathsf{D}^I_a E]\!]$.

   - Case b: If $a = b$, then we have

     $$\mathsf{D}^I_a[\![b]\!] = \mathsf{D}^I_a[\![a]\!] = \{v \in \Sigma^* \mid a \sim_{\lhd a \rhd} v\} = \mathbf{1} = [\![1]\!] = [\![\mathsf{D}^I_a a]\!] = [\![\mathsf{D}^I_a b]\!].$$

     If $a \neq b$, then we have

     $$\mathsf{D}^I_a[\![b]\!] = \{v \in \Sigma^* \mid a \sim_{\lhd b \rhd} v\} = \emptyset = [\![0]\!] = [\![\mathsf{D}^I_a b]\!].$$

   - Case EF: By i.h. we have $\mathsf{D}^I_a[\![E]\!] = [\![\mathsf{D}^I_a E]\!]$ and $\mathsf{D}^I_a[\![F]\!] = [\![\mathsf{D}^I_a F]\!]$. We have

     $$\begin{array}{rcl}
     \mathsf{D}^I_a[\![EF]\!] & = & \{v_l v_r \mid v_l a v_r \in [\![EF]\!] \wedge v_l\,I\,a\} \\
     & = & \{e_l e_r z_f \mid e_l a e_r \in [\![E]\!] \wedge e_l\,I\,a \wedge z_f \in [\![F]\!]\} \cup \\
     & & \{z_e f_l f_r \mid z_e \in [\![E]\!] \wedge z_e f_l\,I\,a \wedge f_l a f_r \in [\![F]\!]\} \\
     & = & \{v_e z_f \mid v_e \in \mathsf{D}^I_a[\![E]\!] \wedge z_f \in [\![F]\!]\} \cup \\
     & & \{z_e v_f \mid z_e \in \mathsf{R}^I_a[\![E]\!] \wedge v_f \in \mathsf{D}^I_a[\![F]\!]\} \\
     & = & (\mathsf{D}^I_a[\![E]\!]) \cdot [\![F]\!] \cup (\mathsf{R}^I_a[\![E]\!]) \cdot (\mathsf{D}^I_a[\![F]\!]) \\
     & = & [\![\mathsf{D}^I_a E]\!] \cdot [\![F]\!] \cup [\![\mathsf{R}^I_a E]\!] \cdot [\![\mathsf{D}^I_a F]\!] \\
     & = & [\![\mathsf{D}^I_a (EF)]\!].
     \end{array}$$

2. Both claims by induction on u. The corresponding statement from (1) is used in the step case. □

The trace-closing interpretation of expressions corresponds to the Brzozowski reordering derivative D^I in the following sense. More precisely, it corresponds to the automaton induced by the Brzozowski reordering derivative that we describe next.

Proposition 3.24. *For any E,*

1. *for any $a \in \Sigma$, $v \in \Sigma^*$, $av \in [\![E]\!]^I \iff v \in [\![\mathsf{D}^I_a E]\!]^I$;*

2. *for any* $u, v \in \Sigma^*$, $uv \in [\![E]\!]^I \iff v \in [\![\mathsf{D}^I_u E]\!]^I$;

3. *for any* $u \in \Sigma^*$, $u \in [\![E]\!]^I \iff (\mathsf{D}^I_u E)\not\downarrow$.

Proof.

1. By propositions 3.12, 3.18.(1) and 3.23.(1), we have the following equivalences:

$$\begin{array}{rcl} av \in [\![E]\!]^I & \iff & av \in [[\![E]\!]]^I \\ & \iff & v \in \mathsf{D}_a[[\![E]\!]]^I \\ & \iff & v \in [\mathsf{D}^I_a[\![E]\!]]^I \\ & \iff & v \in [[\![\mathsf{D}^I_a E]\!]]^I \\ & \iff & v \in [\![\mathsf{D}^I_a E]\!]^I. \end{array}$$

2. By induction on u (and utilising (1) in the step case).

3. Follows from (2) for u and ε. □

As with the classical Brzozowski derivative, we can use the reordering Brzozowski derivative operation to construct deterministic automata. For an expression E, take $Q^E =_{\mathrm{df}} \{\mathsf{D}^I_u E \mid u \in \Sigma^*\}$, $q^E_0 =_{\mathrm{df}} E$, $F^E =_{\mathrm{df}} \{E' \in Q^E \mid E'\not\downarrow\}$, $\delta^E_a E' =_{\mathrm{df}} \mathsf{D}^I_a E'$ for $E' \in Q^E$. By Proposition 3.24, this automaton accepts the closure $[\![E]\!]^I$. But even quotiented by the full Kleene algebra theory, the quotient of Q^E is not necessarily finite, i.e., we may be able to construct infinitely many different languages by taking reordering derivatives.

For the expression from Example 3.19, we have $\mathsf{D}^I_{b^n}((ab)^*) \doteq a^n(ab)^*$, so it has infinitely many Brzozowski reordering derivatives even up to the Kleene algebra theory. This is only to be expected, as the closure $[\![(ab)^*]\!]^I$ is not regular and cannot possibly have an accepting finite automaton.

Example 3.25. In Example 2.17 we saw that for $\Sigma =_{\mathrm{df}} \{a, b\}$ and

$$E =_{\mathrm{df}} 1 + a(1 + b(1 + a)) + b(1 + a(1 + b))$$

we have $\mathsf{D}_a E \doteq 1 + b + ba$. We now take $a\,I\,b$ and this results in the following.

$$\begin{array}{rcl} \mathsf{D}^I_a E & = & \mathsf{D}^I_a 1 + \mathsf{D}^I_a(a(1 + b(1 + a))) + \mathsf{D}^I_a(b(1 + a(1 + b))) \\ & = & 0 + ((\mathsf{D}^I_a a)(1 + b(1 + a)) + (\mathsf{R}^I_a a)(\mathsf{D}^I_a(a + b(1 + a)))) \\ & & + ((\mathsf{D}^I_a b)(1 + a(1 + b)) + (\mathsf{R}^I_a b)(\mathsf{D}^I_a(a + a(1 + b)))) \\ & = & 0 + (1(1 + b(1 + a)) + 0(\mathsf{D}^I_a(a + b(1 + a)))) \\ & & + (0(1 + a(1 + b)) + b(\mathsf{D}^I_a(a + a(1 + b)))) \\ & \doteq & 1 + b(1 + a) + b(\mathsf{D}^I_a(a + a(1 + b))) \\ & = & 1 + b(1 + a) + b(\mathsf{D}^I_a a + ((\mathsf{D}^I_a a)(1 + b) + (\mathsf{R}^I_a a)(\mathsf{D}^I_a(1 + b)))) \\ & = & 1 + b(1 + a) + b(1 + (1(1 + b) + 0(\mathsf{D}^I_a(1 + b)))) \\ & \doteq & 1 + b(1 + a) + b(1 + b) \\ & \doteq & 1 + b + ba + b + bb \end{array}$$

This reflects what we saw in Example 3.20 where $\mathsf{D}^I_a[\![E]\!] = \{\varepsilon, b, ba, bb\}$. There we observed that $b \in \mathsf{D}^I_a[\![E]\!]$ was witnessed by both $\underline{a}b$ and $b\underline{a}$. Here we can see that b indeed occurs twice in the result.

It can be shown that $\mathsf{D}^I_{ab} E \doteq 1 + a + b$. Note that in Example 2.17 we had $\mathsf{D}_{ab} E \doteq 1 + a$. Similarly, it can be shown that $\mathsf{D}^I_{aa} E \doteq b$, but in Example 2.17 we had $\mathsf{D}_{aa} E \doteq 0$.

3.3.3 Antimirov Reordering Derivative

Like the classical Brzozowski derivative that was optimised by Antimirov [10], the Brzozowski reordering derivative construction can be optimised by switching from functions on expressions to multivalued functions or relations.

Definition 3.26. The *Antimirov I-reordering parts-of-derivative* of an expression along a letter and a word are given by relations $\to^I \subseteq \mathsf{RE} \times \Sigma \times \mathsf{RE}$ and $\to^{I*} \subseteq \mathsf{RE} \times \Sigma^* \times \mathsf{RE}$ defined inductively by

$$\frac{}{a \to^I (a,1)} \quad \frac{E \to^I (a,E')}{E+F \to^I (a,E')} \quad \frac{F \to^I (a,F')}{E+F \to^I (a,F')}$$

$$\frac{E \to^I (a,E')}{EF \to^I (a,E'F)} \quad \frac{F \to^I (a,F')}{EF \to^I (a,(\mathsf{R}^I_a E)F')} \quad \frac{E \to^I (a,E')}{E^* \to^I (a,(\mathsf{R}^I_a E)^* E' E^*)}$$

$$\frac{}{E \to^{I*} (\varepsilon,E)} \quad \frac{E \to^{I*} (u,E') \quad E' \to^I (a,E'')}{E \to^{I*} (ua,E'')}$$

Here R^I is defined as before. Similarly to the Brzozowski reordering derivative from the previous subsection, the condition $E\not\downarrow$ in the second EF rule has has been replaced by multiplication with $\mathsf{R}^I_a E$, and the E^* rule has also been adjusted accordingly.

Example 3.27. In Example 3.22 we saw that for $\Sigma =_{\mathrm{df}} \{a,b,c\}$, $a\,I\,c$, $b\,I\,c$ and $E =_{\mathrm{df}} a + ba + ca$ we have $\mathsf{D}^I_a E \doteq 1 + c$. Again, with the Antimirov derivatives we have a choice.

$$\frac{\dfrac{}{a \to^I (a,1)}}{a + ba + ca \to^I (a,1)} \qquad \frac{\dfrac{\dfrac{}{a \to^I (a,1)}}{ca \to^I (a,(\mathsf{R}^I_a c)1}}{ba + ca \to^I (a,(\mathsf{R}^I_a c)1)}}{a + ba + ca \to^I (a,(\mathsf{R}^I_a c)1)}$$

Since $\mathsf{R}^I_a c = c$ we have $a + ba + ca \to^I (a,c1)$. Note that we also have a derivation which follows the summand ba, namely, we can derive $a + ba + ca \to^I (a,01)$.

The following proposition shows that the Antimirov reordering parts-of-derivative of an expression E collectively compute the semantic reordering derivative of the language $[\![E]\!]$. In other words, each E' such that $E \to^I (a,E')$ gives a part of the (semantic) reordering derivative of $[\![E]\!]$.

Proposition 3.28. *For any E,*

1. *for any $a \in \Sigma$, $\mathsf{D}^I_a [\![E]\!] = \bigcup \{ [\![E']\!] \mid E \to^I (a,E') \}$;*
2. *for any $u \in \Sigma^*$, $\mathsf{D}^I_u [\![E]\!] = \bigcup \{ [\![E']\!] \mid E \to^{I*} (u,E') \}$.*

We also have that the trace-closing interpretation of expressions corresponds to the Antimirov reordering parts-of-derivative in the following sense. More precisely, it corresponds to the automaton induced by the Antimirov reordering parts-of-derivative that we describe next.

Proposition 3.29. *For any E,*

1. *for any $a \in \Sigma$, $v \in \Sigma^*$, $av \in [\![E]\!]^I \iff \exists E'.\, E \to^I (a,E') \wedge v \in [\![E']\!]^I$;*

2. *for any* $u, v \in \Sigma^*$, $uv \in [\![E]\!]^I \iff \exists E'.\, E \to^{I*} (u, E') \wedge v \in [\![E']\!]^I$;

3. *for any* $u \in \Sigma^*$, $u \in [\![E]\!]^I \iff \exists E'.\, E \to^{I*} (u, E') \wedge E' \not\downarrow$.

Proof.

1. By Propositions 3.12, 3.18.(1) and 3.28.(1), we have the following equivalences:

$$\begin{array}{rcl} av \in [\![E]\!]^I & \iff & av \in [[\![E]\!]]^I \\ & \iff & v \in \mathsf{D}_a[[\![E]\!]]^I \\ & \iff & v \in [\mathsf{D}_a^I[\![E]\!]]^I \\ & \iff & v \in [\bigcup\{[\![E']\!] \mid E \to^I (a, E')\}]^I \\ & \iff & v \in \bigcup\{[[\![E']\!]]^I \mid E \to^I (a, E')\} \\ & \iff & v \in \bigcup\{[\![E']\!]^I \mid E \to^I (a, E')\} \\ & \iff & \exists E'.\, E \to^I (a, E') \wedge v \in [\![E']\!]^I. \end{array}$$

2. By induction on u (and utilising (1) in the step case).

3. Follows from (2) for u and ε. □

Like the classical Antimirov construction, the reordering parts-of-derivative of an expression E give a nondeterministic automaton by $Q^E =_{\mathrm{df}} \{E' \mid \exists u \in \Sigma^*.\, E \to^{I*} (u, E')\}$, $q_0^E =_{\mathrm{df}} E$, $F^E =_{\mathrm{df}} \{E' \in Q^E \mid E' \not\downarrow\}$, $E' \to^E (a, E'') =_{\mathrm{df}} E' \to^I (a, E'')$ for $E', E'' \in Q^E$. This automaton accepts $[\![E]\!]^I$ by Proposition 3.29, but is generally infinite, even when quotiented by the full Kleene algebra theory.

Revisiting Example 3.19 again, $(ab)^*$ must have infinitely many Antimirov reordering parts-of-derivatives modulo the Kleene algebra theory since $[\![(ab)^*]\!]^I$ is not regular and cannot have a finite accepting nondeterministic automaton. More specifically, the expression $(a0)^*((a1)\ldots((a0)^*((a1)(ab)^*))\ldots) \doteq a^n(ab)^*$ is its single reordering part-of-derivative along b^n.

However, if quotienting the Antimirov automaton for E by some sound theory (a theory weaker than the Kleene algebra theory) makes it finite, then the Brzozowski automaton can also be quotiented to become finite.

Proposition 3.30. *For any* E,

1. *for any* $a \in \Sigma$, $\mathsf{D}_a^I E \doteq \sum\{E' \mid E \to^I (a, E')\}$;

2. *for any* $u \in \Sigma^*$, $\mathsf{D}_u^I E \doteq \sum\{E' \mid E \to^{I*} (u, E')\}$

(using the semilattice equations for $0, +$*, that* 0 *is zero for* $\cdot$*, and distributivity of* $\cdot$ *over* $+$*).*

Proof.

1. By induction on E. We only show a few cases.

   - Case b with $a = b$: There is exactly one E' such that $b \to^I (a, E')$, namely $E' = 1$. Thus we have:

$$\mathsf{D}_a^I b = \mathsf{D}_a^I a = 1 = \sum\{1\} = \sum\{E' \mid b \to^I (a, E')\}.$$

   - Case b with $a \neq b$: There is no E' such that $b \to^I (a, E')$. Thus we have:

$$\mathsf{D}_a^I b = 0 = \sum\emptyset = \sum\{E' \mid b \to^I (a, E')\}.$$

- Case EF: By i.h. for E and for F we have $\mathsf{D}^I_a E \doteq \sum\{E' \mid E \to^I (a,E')\}$ and $\mathsf{D}^I_a F \doteq \sum\{F' \mid F \to^I (a,F')\}$. Thus we have:

$$\begin{array}{lcl} \mathsf{D}^I_a(EF) & = & (\mathsf{D}^I_a E)F + (\mathsf{R}^I_a E)(\mathsf{D}^I_a F) \\ & \doteq & (\sum\{E' \mid E \to^I (a,E')\})F + (\mathsf{R}^I_a E)(\sum\{F' \mid F \to^I (a,F')\}) \\ & \doteq & \sum\{E'F \mid E \to^I (a,E')\} + \sum\{(\mathsf{R}^I_a E)F' \mid F \to^I (a,F')\} \\ & \doteq & \sum\{E' \mid EF \to^I (a,E')\}. \end{array}$$

2. By induction on u (utilising (1) in the step case). □

Corollary 3.31. *If some sound quotient of the Antimirov automaton for E (accepting $[\![E]\!]^I$) is finite, then also some sound quotient of the Brzozowski automaton is finite.*

Example 3.32. In Example 3.20, we saw that for $\Sigma =_{\mathrm{df}} \{a,b\}$, $a\,I\,b$ and

$$E =_{\mathrm{df}} 1 + a(1 + b(1 + a)) + b(1 + a(1 + b))$$

we have $b \in \mathsf{D}^I_a[\![E]\!]$ witnessed both by $\underline{a}b \in [\![E]\!]$ and $b\underline{a} \in [\![E]\!]$. We now show that we have two derivations for $E \to^I (a,E')$ such that $b \in [\![E']\!]$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a \to^I (a,1)}}{a(1+b(1+a)) \to^I (a,1(1+b(1+a)))}}{a(1+b(1+a)) + b(1+a(1+b)) \to^I (a,1(1+b(1+a)))}}{1 + a(1+b(1+a)) + b(1+a(1+b)) \to^I (a,1(1+b(1+a)))}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a \to^I (a,1)}}{a(1+b) \to^I (a,1(1+b))}}{1 + a(1+b) \to^I (a,1(1+b))}}{b(1+a(1+b)) \to^I (a,(\mathsf{R}^I_a b)(1(1+b)))}}{a(1+b(1+a)) + b(1+a(1+b)) \to^I (a,(\mathsf{R}^I_a b)(1(1+b)))}$$

$$\dfrac{}{1 + a(1+b(1+a)) + b(1+a(1+b)) \to^I (a,(\mathsf{R}^I_a b)(1(1+b)))}$$

As $\mathsf{R}^I_a b = b$, we have $1(1+b(1+a)) \doteq 1 + b + ba$ and $(\mathsf{R}^I_a b)(1(1+b)) = b + bb$.

3.3.4 Automaton Finiteness for Star-Connected Expressions

Looking at the rules in Definition 3.26, it is clear that to have $E \to^I (a,E')$ the letter a must occur in the expression E. Furthermore, if this occurrence of a is not under a star, then that particular occurrence is replaced with 1 in E' and the resulting expression E' is smaller than E. Thus, if E does not contain a star, then we can apply the rules only a certain number of times, after which there are no more letters left in the result and none of the rules apply. This means that the set of Antimirov reordering parts-of-derivative of such E is finite. The rule for E^*, however, allows the resulting expression E' to be larger than E^* and thus E^* is problematic for obtaining finiteness.

We now show that for star-connected expressions (Definition 2.9; these are expressions where the language of the expression under a star is connected) we obtain finiteness of the set of Antimirov reordering parts-of-derivative (modulo suitable equations). We do this by first showing that the parts-of-derivative of E^* are of a certain form (modulo some equations). We then show that for a star-connected expression we can bound the size of

this form. Finally, we show that all constituents of this form come from a finite set determined by the original expression E. Thus the set of parts-of-derivative of a star-connected expression (modulo certain equations) is finite.

Lemma 3.33. *If a language L is connected, then for any $u \in \Sigma^+$, $\mathsf{R}^I_u(\mathsf{D}^I_u L) \subseteq \mathbf{1}$.*

Proof. Let $v \in \mathsf{R}^I_u(\mathsf{D}^I_u L)$. By definition of R^I_u, we have $v\, I\, u$. Since L is connected and $v \in \mathsf{D}^I_u L$, if $v \neq \varepsilon$, then $a\, D\, b$ for some $a \in u$ and $b \in v$. But then it is not the case that $v\, I\, u$. Thus it must be that $v = \varepsilon$. □

Lemma 3.34. *For any E, if $[\![E]\!] \subseteq \mathbf{1}$, then either $E \doteq 0$ or $E \doteq 1$ (using the equations involving 0 and 1 only (e.g., $0 + 1 \doteq 1$ and $0^* \doteq 1$ etc.) and that 0 is zero).*

Lemma 3.35. *For any E, E' and $u \in \Sigma^+$, if $[\![E]\!]$ is connected and $E \to^{I*} (u, E')$, then $\mathsf{R}^I_u E' \doteq 0$ or $\mathsf{R}^I_u E' \doteq 1$ (using the equations involving 0 and 1 only and that 0 is zero).*

Proof. From $E \to^{I*} (u, E')$ by Proposition 3.28, $[\![E']\!] \subseteq \mathsf{D}^I_u [\![E]\!]$. Hence by Lemma 3.33, we get $[\![\mathsf{R}^I_u E']\!] = \mathsf{R}^I_u [\![E']\!] \subseteq \mathsf{R}^I_u(\mathsf{D}^I_u [\![E]\!]) \subseteq \mathbf{1}$. By Lemma 3.34, $\mathsf{R}^I_u E' \doteq 0$ or $\mathsf{R}^I_u E' \doteq 1$. □

Next is an observation about the interaction of reorderable part R^I and the Antimirov derivative: if we can obtain an expression by deriving a reorderable part of E, then we can obtain the same expression by first deriving E and then taking the reorderable part of the result.

Lemma 3.36. *For any E, E', u and a, if $\mathsf{R}^I_u E \to^I (a, E')$, then there exists E'' such that $E \to^I (a, E'')$ and $\mathsf{R}^I_u E'' = E'$.*

Proof. By induction on the derivation $\mathsf{R}^I_u E \to^I (a, E')$. □

This enables us to have the following development for E' when $E^* \to^{I*} (u, E')$.

Lemma 3.37. *For any E, E' and $u \in \Sigma^*$, if $E^* \to^{I*} (u, E')$, then there exist $n \in \mathbb{N}$, $E_1, \ldots, E_n$, $\emptyset \subset X_0, \ldots, X_n \subseteq \Sigma$ and $u_1, \ldots, u_n \in \Sigma^+$ such that $u \sim^{I*} u_1 \ldots u_n$ and*

$$E' \doteq (\mathsf{R}^I_{X_0} E)^* (\mathsf{R}^I_{X_1} E_1)(\mathsf{R}^I_{X_1} E)^* \ldots (\mathsf{R}^I_{X_n} E_n)(\mathsf{R}^I_{X_n} E)^*$$

where $\doteq$ uses only associativity of $\cdot$ and, furthermore, we have: $X_{i-1} = X_i \cup \Sigma(u_i)$, $X_n = \emptyset$ and $E \to^{I*} (u_i, E_i)$ for all i.

Lemma 3.38. *For any E, E' and $u \in \Sigma^*$, if $[\![E]\!]$ is connected, $E^* \to^{I*} (u, E')$ and, for the development of E' from the previous lemma, we have $X_{i-1} = X_i$ for some i, then $\mathsf{R}^I_{X_i} E_i \doteq 0$ or $\mathsf{R}^I_{X_i} E_i \doteq 1$ (using the equations involving 0 and 1 only, and that 0 is zero).*

Proof. We have $\Sigma(u_i) \subseteq X_{i-1} = X_i$. From $E \to^{I*} (u_i, E_i)$, by Lemma 3.35 either $\mathsf{R}^I_{u_i} E_i \doteq 0$ or $\mathsf{R}^I_{u_i} E_i \doteq 1$. Therefore also $\mathsf{R}^I_{X_i} E_i \doteq 0$ or $\mathsf{R}^I_{X_i} E_i \doteq 1$. □

Lemma 3.39. *For any E, E' and $u \in \Sigma^*$, if $[\![E]\!]$ is connected and $E^* \to^{I*} (u, E')$, then there exist $n \leq |\Sigma|$, $E_1, \ldots, E_n$ and $\emptyset \subset X_0, \ldots, X_n \subseteq \Sigma$ such that*

$$E' \doteq (\mathsf{R}^I_{X_0} E)^* (\mathsf{R}^I_{X_1} E_1)(\mathsf{R}^I_{X_1} E)^* \ldots (\mathsf{R}^I_{X_n} E_n)(\mathsf{R}^I_{X_n} E)^*$$

and $X_{i-1} \supset X_i$ for all i (using, in addition to the equations mentioned in the lemmata above, unitality of 1 and the equation $F^* \cdot F^* \doteq F^*$).

Proof. From Lemmata 3.37, 3.38 noting that at most $|\Sigma|$ of the inclusions $X_{i-1} \supseteq X_i$ can be proper. □

Definition 3.40. The functions $(\_)^{\to+}, (\_)^{\to*} : \mathrm{RE} \to \mathscr{P}(\mathrm{RE})$ are defined by

$$\begin{array}{rcl}
a^{\to+} & =_{\mathrm{df}} & \{1\} \\
0^{\to+} & =_{\mathrm{df}} & \emptyset \\
(E+F)^{\to+} & =_{\mathrm{df}} & E^{\to+} \cup F^{\to+} \\
1^{\to+} & =_{\mathrm{df}} & \emptyset \\
(EF)^{\to+} & =_{\mathrm{df}} & E^{\to+} \cdot \{F\} \cup \bigcup \{ \mathsf{R}^I_X(E^{\to*}) \cdot F^{\to+} \mid \emptyset \subset X \subseteq \Sigma \} \\
(E^*)^{\to+} & =_{\mathrm{df}} & \{ (\mathsf{R}^I_{X_0}E)^*(\mathsf{R}^I_{X_1}E_1)(\mathsf{R}^I_{X_1}E)^* \ldots (\mathsf{R}^I_{X_n}E_n)(\mathsf{R}^I_{X_n}E)^* \mid \\
 & & \quad n > 0, \emptyset \subset X_i \subseteq \Sigma, X_{i-1} \supseteq X_i, X_n = \emptyset, E_i \in E^{\to+} \} \\
 & & \\
E^{\to*} & =_{\mathrm{df}} & \{E\} \cup E^{\to+}
\end{array}$$

Proposition 3.41. *For any E, E' and $u \in \Sigma^*$, if $E \to^{I*} (u, E')$, then there exists E'' such that $E' \doteq E''$ and $E'' \in E^{\to*}$ (using only the equations mentioned in the above lemmata).*

Proposition 3.42. *If E is star-connected, then a suitable sound quotient of the state set $\{E' \mid \exists u \in \Sigma^*.\ E \to^{I*} (u, E')\}$ of the Antimirov automaton for E (accepting $[\![E]\!]^I$) is finite.*

Proof. By Lemma 3.39, for a star-connected expression E, we only need to consider $n \leq |\Sigma|$ in the definition of $(E^*)^{\to+}$ for Proposition 3.41 to hold. This restriction makes the set $E^{\to*}$ finite. □

3.4 Uniform Scattering Rank of a Language

We proceed to defining the notion of uniform scattering rank of a language and show that star-connected expressions define languages with uniform scattering rank.

3.4.1 Scattering Rank vs. Uniform Scattering Rank

The notion of scattering rank of a language (a.k.a. distribution rank, k-block testability) was introduced by Hashiguchi [32].

Definition 3.43. A language L has *(I-scattering) rank* at most N if

$$\forall u, v.\, uv \in [L]^I \Longrightarrow \exists z \in L.\, u \sim \lhd_N z \rhd \sim v.$$

We say that L has rank if it has rank at most N for some $N \in \mathbb{N}$. If it does, then, for the least such N, we say that L has rank N. The only languages with rank 0 are $\emptyset$ and $\mathbf{1}$. If a nontrivial language L is closed, it has rank 1: for any $uv \in [L]^I$, we also have $uv \in L$ and $u \lhd uv \rhd \varepsilon, v$. Still, having rank 1 does not imply that the language is closed: if $a\,I\,b$, then $L =_{\mathrm{df}} \{ab\}$ has rank 1 but its closure is $[L]^I = \{ab, ba\}$.

Having rank is a sufficient condition for regularity of the trace closure of a regular language. But it is not a necessary condition.

Proposition 3.44 (Hashiguchi [32])**.** *(cf. [60, Prop. 6.3.2]) If a regular language L has rank, then $[L]^I$ is regular.*

Proposition 3.45. *There exist regular languages L such that $[L]^I$ is regular but L is without a rank.*

Proof. Consider $\Sigma =_{\mathrm{df}} \{a, b\}$, $a\,I\,b$. The regular language $L =_{\mathrm{df}} [\![(ab)^*(a^* + b^*)]\!]$ is without a rank, since, for any n, we have $(ab)^n \in L$ and $a^n b^n \in [L]^I$ while the smallest N such that $a^n \sim \lhd_N (ab)^n \rhd \sim b^n$ is n. Nonetheless, $[L]^I = \Sigma^* = [\![(a+b)^*]\!]$ is regular. □

We wanted to show that a truncation of the refined Antimirov automaton (which we define in Section 3.5) is finite for expressions whose language has rank. But it turns out, as we shall see, that rank does not quite work for this. For this reason, we introduce a stronger notion that we call uniform scattering rank.

Definition 3.46. A language L has *uniform (I-scattering) rank* at most N if

$$\forall w \in [L]^I.\ \exists z \in L.\ \forall u, v.\ w = uv \Longrightarrow u \sim \lhd_N z \rhd \sim v.$$

The difference between the two definitions is that, in the uniform case, the choice of z depends only on w whereas, in the non-uniform case, it depends on the particular split of w as $w = uv$, i.e., for every such split of w we may choose a different z.

Lemma 3.47. *If L has uniform rank at most N, then L has rank at most N.*

The converse of the above lemma does not hold—there are languages with uniform rank greater than rank. Furthermore, there are languages that have rank but no uniform rank.

Proposition 3.48. *Let $\Sigma =_{\mathrm{df}} \{a, b, c\}$, $a\, I\, b$ and*

$$E =_{\mathrm{df}} a^* b^* c (ab)^* (a^* + b^*) + (ab)^* (a^* + b^*) c a^* b^*.$$

1. *The language $[\![E]\!]$ has rank 2.*
2. *The language $[\![E]\!]$ has no uniform rank.*

Proof. Note that c behaves like a separator in this independence alphabet—although a and b are independent, neither a nor b commutes with c. Thus the a's and b's before (after) a c in a word must stay before (after) that c in any equivalent word.

It can be seen that words in $[\![E]\!]^I$ are of the form $w_l c w_r$ where w_l and w_r consist of some number of a's and b's, i.e., $[\![E]\!]^I = [\![(a+b)^* c (a+b)^*]\!]$.

1. Let $uv \in [\![E]\!]^I$. We have to find u_1, u_2 and v_0, v_1, v_2 so that $u_1 u_2 \sim^{I*} u$, $v_0 v_1 v_2 \sim^{I*} v$, $v_0\, I\, u_1$, $v_0 v_1\, I\, u_2$ and $v_0 u_1 v_1 u_2 v_2 \in [\![E]\!]$. There are two cases to consider: either c is in the suffix v or it is in the prefix u.

   - Case $c \in v$: Let $x, y \in \Sigma^*$ be such that $v = xcy$. Take $u_1 =_{\mathrm{df}} \pi_a(u)$, $u_2 =_{\mathrm{df}} \pi_b(u)$, $v_0 =_{\mathrm{df}} \varepsilon$ and $v_1 =_{\mathrm{df}} \pi_a(x)$. Let $k =_{\mathrm{df}} |y|_a$, $l =_{\mathrm{df}} |y|_b$ and $m =_{\mathrm{df}} \min(k, l)$ and take $v_2 =_{\mathrm{df}} \pi_b(x) c (ab)^m a^{k-m} b^{l-m}$. Since u consists of only a's and b's and $a\, I\, b$, we have

     $$u_1 u_2 = \pi_a(u) \pi_b(u) \sim^{I*} u.$$

     Since x and y consist only of a's and b's, we have $\pi_a(x) \pi_b(x) \sim^{I*} x$ and similarly $(ab)^m a^{k-m} b^{l-m} \sim^{I*} y$. Thus, we also have

     $$v_0 v_1 v_2 = \varepsilon \pi_a(x) \pi_b(x) c (ab)^m a^{k-m} b^{l-m} \sim^{I*} xcy = v.$$

     Altogether, we have

     $$v_0 u_1 v_1 u_2 v_2 = \varepsilon \pi_a(u) \pi_a(x) \pi_b(u) \pi_b(x) c (ab)^m a^{k-m} b^{l-m}$$

     and thus

     $$v_0 u_1 v_1 u_2 v_2 \in [\![a^* b^* c (ab)^* (a^* + b^*)]\!].$$

Since $\varepsilon \mathrel{I} \pi_a(u)$ and $\varepsilon\pi_a(x) \mathrel{I} \pi_b(u)$, we have that $v_0 \mathrel{I} u_1$ and $v_0 v_1 \mathrel{I} u_2$. Hence

$$uv \sim^{I*} u_1 u_2 v_0 v_1 v_2 \sim^{I*} v_0 u_1 v_1 u_2 v_2.$$

We note that $v_0 u_1 v_1 u_2 v_2$ is a scattering when $u_1 \neq \varepsilon$, $u_2 \neq \varepsilon$ and $v_1 \neq \varepsilon$. If this is not the case, then we can construct a scattering with degree 1.

- Case $c \in u$: Similar to the previous case. Let x and y be such that $u = xcy$. Let $k =_{\text{df}} |x|_a$, $l =_{\text{df}} |x|_b$ and $m =_{\text{df}} \min(k,l)$. Take $u_1 =_{\text{df}} (ab)^m a^{k-m} b^{l-m} c\pi_a(y)$, $u_2 =_{\text{df}} \pi_b(y)$, $v_0 =_{\text{df}} \varepsilon$, $v_1 =_{\text{df}} \pi_a(v)$ and $v_2 =_{\text{df}} \pi_b(v)$. In this case we have

$$v_0 u_1 v_1 u_2 v_2 = \varepsilon(ab)^m a^{k-m} b^{l-m} c\pi_a(y)\pi_a(v)\pi_b(y)\pi_b(v)$$

and also

$$uv \sim^{I*} u_1 u_2 v_0 v_1 v_2 \sim^{I*} v_0 u_1 v_1 u_2 v_2$$

but we consider the other part of the expression E:

$$v_0 u_1 v_1 u_2 v_2 \in [\![(ab)^*(a^* + b^*)ca^*b^*]\!].$$

2. Assume $[\![E]\!]$ has uniform rank at most N. Take

$$w =_{\text{df}} a^{N+1} b^{N+1} c a^{N+1} b^{N+1} \in [\![E]\!]^I.$$

By assumption, there exists $z \in [\![E]\!]$ such that,

$$\forall u, v. w = uv \Longrightarrow u \sim \lhd_N z \rhd \sim v.$$

Take $u =_{\text{df}} a^{N+1}$ and $v =_{\text{df}} b^{N+1} c a^{N+1} b^{N+1}$. Thus, it must be that for some $n \leq N$, $z = v_0 u_1 v_1 \ldots u_n v_n$ and $u = a^{N+1} \sim^{I*} u_1 \ldots u_n$. Since we have $N+1$ letters a to divide into $n \leq N$ words, at least one u_i must consist of more than one a and thus z must contain two consecutive a's that are before c.

Take $u' =_{\text{df}} a^{N+1} b^{N+1} c a^{N+1}$ and $v' =_{\text{df}} b^{N+1}$. Again, it must be that for some $n \leq N$, $z = v'_0 u'_1 v'_1 \ldots u'_n v'_n$ and $v' = b^{N+1} \sim^{I*} v'_0 \ldots v'_n$. Note that c must be in one of the u'_i's and thus for all $j < i$ it must be that $v'_j = \varepsilon$. Hence $v'_0 = \varepsilon$ and we have $N+1$ letters b to divide into $n \leq N$ words and thus at least one v'_i consists of more than one b. Thus z must contain two consecutive b's that are after c.

We can observe that the only words in $[\![E]\!]$ equivalent to w are $a^{N+1} b^{N+1} c (ab)^{N+1}$ and $(ab)^{N+1} c a^{N+1} b^{N+1}$. Neither of these has both two consecutive a's before c as well as two consecutive b's after c, so neither qualifies as z. Thus there is no $z \in [\![E]\!]$ such that $u \sim \lhd_N z \rhd \sim v$ and $u' \sim \lhd_N z \rhd \sim v'$. □

In the following example we try to give a more visual explanation of the above proposition.

Example 3.49. Just as in Proposition 3.48, we take $\Sigma =_{\text{df}} \{a,b,c\}$, $a \mathrel{I} b$ and

$$E =_{\text{df}} a^*b^*c(ab)^*(a^* + b^*) + (ab)^*(a^* + b^*)ca^*b^*.$$

Observe that $a^3b^3ca^3b^3 \in [\![E]\!]^I$, witnessed by $a^3b^3c(ab)^3 \in [\![E]\!]$ and $(ab)^3ca^3b^3 \in [\![E]\!]$. We first consider $a^3b^3c(ab)^3 \in [\![E]\!]$. The word $u =_{\text{df}} aaa$ is scattered in $a^3b^3c(ab)^3$ as follows.

$$\underline{aaa}bbbcababab$$

We can see that we need only one underlined block (a word u_i in the scattering). In fact, for any prefix of a word in the equivalence class of $a^3b^3ca^3b^3$ which does not include c we need at most two underlined blocks for the scattering. For example, aab is scattered in $a^3b^3c(ab)^3$ as follows.

$$\underline{aa}\underline{a}bbbcababab$$

Things change when we consider prefixes that also include the letter c. For example, the word $u' =_{\mathrm{df}} aaabbbcaaa$ is scattered in $a^3b^3c(ab)^3$ as follows.

$$\underline{aaabbbca}\underline{b}\underline{ab}\underline{ab}$$

In this case we need three underlined blocks. Similarly, we can see that scattering the word $a^nb^nca^n$ in $a^nb^nc(ab)^n$ requires n underlined blocks.

We can also consider the witness $(ab)^3ca^3b^3 \in [\![E]\!]$. This time we require three underlined blocks to scatter the word u.

$$\underline{ab}\underline{ab}\underline{ab}caaabbb$$

Similarly, we can see that scattering the word a^n in $(ab)^nca^nb^n$ requires n underlined blocks. On the other hand, we only require one underlined block for the scattering of u'.

$$\underline{abababcaaa}bbb$$

In fact, for any prefix of a word in the equivalence class of $a^3b^3ca^3b^3$ which does include c, we need at most two underlined blocks for the scattering. For example, a^3b^3cabb is scattered in $(ab)^3ca^3b^3$ as follows.

$$\underline{abababca}aa\underline{bbb}$$

For a language L to have rank at most N, the definition requires that, for any word $uv \in [L]^I$, there exists $z \in L$ such that the scattering of u in z requires at most N underlined blocks. For the example above, we choose the witness z based on whether the prefix u contains the letter c or not.

For a language L to have uniform rank at most N, the definition requires that, for any word $w \in [L]^I$, there exists a $z \in L$ such that, for any u, v, if $uv = w$, then the scattering of u in z requires at most N underlined blocks. For the example above, we see that, roughly speaking, one witness is good for those u that include c and the other is good for those that do not, but neither of them is good for both.

3.4.2 Star-Connected Languages Have Uniform Rank

Star-connected expressions were described in Section 2.6. Informally, an expression is said to be star-connected (wrt. D) if star is only used over connected languages. Thus we can see that the expression

$$E =_{\mathrm{df}} a^*b^*c(ab)^*(a^*+b^*)+(ab)^*(a^*+b^*)ca^*b^*$$

from Example 3.49 is not star-connected as it contains $(ab)^*$ with $a\,I\,b$. However, the expression

$$E' =_{\mathrm{df}} (a+b)^*c(a+b)^*$$

is star-connected and, furthermore, $[\![E]\!]^I = [\![E']\!]^I$.

Klunder et al. [41] established that star-connectedness is a sufficient condition for a regular language to have rank, although not a necessary one.

Proposition 3.50 (Klunder et al. [41]). *Any star-connected language has rank.*

Proof (sketch). The language $\{a\}$ has rank 1. The languages $\emptyset$ and $\mathbf{1}$ have rank 0. If two languages L_1 and L_2 have ranks at most N_1 resp. N_2, then $L_1 \cup L_2$ has rank at most $\max(N_1, N_2)$ and $L_1 \cdot L_2$ has rank at most $N_1 + N_2$. If a general language L has rank at most N, then L^* need not have rank. For example, for $\Sigma =_{\mathrm{df}} \{a,b\}$, $a\,I\,b$, the language $\{ab\}$ has rank 1, but $\{ab\}^*$ is without rank. But if L is also connected, then L^* turns out to have rank at most $|\Sigma|(N+1)$. The claim follows by induction on the given star-connected expression. □

Proposition 3.51 (Klunder et al. [41]). *There exist regular languages with rank (and also with uniform rank) that are not star-connected.*

Proof. Consider $\Sigma =_{\mathrm{df}} \{a,b\}$, $a\,I\,b$. The language $L =_{\mathrm{df}} [\![(aa+ab+ba+bb)^*]\!]$ has rank 1, in fact even uniform rank 1, because it is closed. We can see that the regular expression $(aa+ab+ba+bb)^*$ is not star-connected, since the language $[\![aa+ab+ba+bb]\!]$ contains disconnected words ab and ba. But a more involved pumping argument also shows that L is not star-connected, i.e., there is no star-connected expression E such that $L = [\![E]\!]$. □

We will now show that star-connected languages also have uniform rank, by refining Klunder et al.'s proof of Proposition 3.50, especially the case of the Kleene star.

Let us analyse the case L^* where L is a connected language. When $w \in [L^*]^I$, then there exists $z \in L^*$ such that $w \sim^{I*} z$. This further means that there exist $n \in \mathbb{N}$ and $z_1, \ldots, z_n \in L$ such that $z = z_1 \ldots z_n$ where we can require that all z_i are nonempty. Since L is connected, each z_i is also connected. If $w = uv$, then there exist $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ such that $u \sim^{I*} u_1 \ldots u_n$, $v \sim^{I*} v_1 \ldots v_n$ and, for every i, $z_i \sim^{I*} u_i v_i$ and, for every $j < i$, $v_j\,I\,u_i$. In other words, u_i is the part of z_i that belongs to u and v_i is the part that belongs to v. In particular, if $u_i = \varepsilon$ (or $z_i \sim^{I*} v_i$), then all letters of z_i belong to the suffix v, and similarly if $v_i = \varepsilon$ (or $z_i \sim^{I*} u_i$), then all letters of z_i belong to the prefix u. We say that a z_i is:

- *unmarked* (by u) when $u_i = \varepsilon$;
- *partially marked* (by u) when $u_i \neq \varepsilon$ and $v_i \neq \varepsilon$;
- *completely marked* (by u) when $v_i = \varepsilon$.

We say that a z_i is *at least* partially marked when it is partially or completely marked. An important property for us is that, if we iterate a closed language L, then any prefix u of a word $w \in [L^*]^I$ can partially mark at most $|\Sigma|$ of the $z_i \in L$ in $z_1 \ldots z_n \in L^*$.

Lemma 3.52. *For any $u, v \in \Sigma^*$, $z_1, \ldots, z_n \in \Sigma^+$ and $u_1, \ldots, u_n, v_1, \ldots, v_n \in \Sigma^*$ such that $uv \sim^{I*} z_1 \ldots z_n$, $u \sim^{I*} u_1 \ldots u_n$, $v \sim^{I*} v_1 \ldots v_n$, for all i, z_i is connected and $z_i \sim^{I*} u_i v_i$, and, for all $j < i$, $v_j\,I\,u_i$, then at most $|\Sigma|$ of the words z_i can be partially marked (by u);*

Proof. If, for some i, we have that z_i is partially marked ($u_i \neq \varepsilon$ and $v_i \neq \varepsilon$), then, since $z_i \sim^{I*} u_i v_i$ is connected, there must exist letters a and b such that $a \in u_i$, $b \in v_i$ and $a\,D\,b$. Since $v_i\,I\,u_{i+1} \ldots u_n$, we have $a \notin u_{i+1} \ldots u_n$. This means that, if there are k partially marked z_i, then these z_i together must contain at least k distinct letters. □

Should it happen for some i that z_i and z_{i+1} are both completely marked ($v_i = v_{i+1} = \varepsilon$, i.e., $z_i \sim^{I*} u_i$ and $z_{i+1} \sim^{I*} u_{i+1}$), then z_i and z_{i+1} belong to the same block of u in the scattering $u \sim \lhd\, z \,\rhd \sim v$. This can potentially help keeping the uniform rank of L^* low as we need less blocks. The same holds for z_i and z_{i+1} that are unmarked: they belong to the same block of v. Having words z_i that are completely marked interspersed with other

types of words z_i (for example, having all odd-numbered z_i completely marked and all even-numbered unmarked), in contrast, is not helpful. It could thus be useful to be able to choose z, n and $z_1, \ldots, z_n$ in such a way that as many as possible of the z_i that are completely marked are adjacent in z for all splits of w as $w = uv$.

For example, take $\Sigma =_{\mathrm{df}} \{a, b\}$, $a \mathrel{I} b$ and $L =_{\mathrm{df}} \Sigma = [\![a + b]\!]$. We now consider two witnesses for $w =_{\mathrm{df}} a^m b^m \in [L^*]^I$. As the first witness we take $z =_{\mathrm{df}} w = a^m b^m \in L^*$ and as the second witness we take $z =_{\mathrm{df}} (ab)^m \in L^*$. Since the words in L are of length one, then so must be our z_i. Furthermore, we have $1 \leq i \leq 2m$. For the first witness we have $z_i =_{\mathrm{df}} a$ for $1 \leq i \leq m$ and $z_i =_{\mathrm{df}} b$ for $m+1 \leq i \leq 2m$. For the second witness we have $z_{2i-1} =_{\mathrm{df}} a$ and $z_{2i} =_{\mathrm{df}} b$ for $1 \leq i \leq m$. In the first case, the letters from u are adjacent in z for all prefixes u of w (as $w = z$). In the second case, they can be interleaved with the letters from v (in the most extreme case $u =_{\mathrm{df}} a^m$ and $v =_{\mathrm{df}} b^m$, the words u and v get scattered into m resp. $m+1$ blocks in z).

Definition 3.53. For any $u \in \Sigma^*$ and $z_1, \ldots, z_n \in \Sigma^+$, we say that u is *left-scattered* in $z_1, \ldots, z_n$ if there exist $u_1, \ldots, u_n, v_1, \ldots, v_n \in \Sigma^*$ such that $u \sim^{I*} u_1 \ldots u_n$, $z_i \sim^{I*} u_i v_i$, for all $j < i$, $v_j \mathrel{I} u_i$, and, for every completely marked z_i (i.e., $v_i = \varepsilon$), either $i = 1$ or z_{i-1} is at least partially marked (i.e., $u_{i-1} \neq \varepsilon$).

Thus, if u is left-scattered in $z_1, \ldots, z_n$, then the completely marked z_i cannot be interspersed with unmarked z_i: the completely marked z_i must occur as contiguous blocks $z_k, \ldots, z_l$ in $z_1, \ldots, z_n$ such that $k = 1$ or z_{k-1} is partially marked. The next lemma ensures that, if $w \in [L^*]^I$, then there exists a witness $z_1 \ldots z_n \in L^*$ (with $z_i \in L$) such that every prefix u of w is left-scattered in $z_1, \ldots, z_n$. This property is used in the following proposition to give a bound on the uniform rank in the case of E^*.

Lemma 3.54. *Let $u, v \in \Sigma^*$, $z_1, \ldots, z_n \in \Sigma^+$ be such that $uv \sim^{I*} z_1 \ldots z_n$. There exists a permutation $\sigma' = z'_1, \ldots, z'_n$ of $\sigma =_{\mathrm{df}} z_1, \ldots, z_n$ such that:*

1. *$z_1 \ldots z_n \sim^{I*} z'_1 \ldots z'_n$;*
2. *for any $\bar{u}, \bar{v} \in \Sigma^*$ such that $\bar{u}\bar{v} = u$, the word $\bar{u}$ is left-scattered in $z'_1, \ldots, z'_n$.*

Proof. By induction on u.

- Case ε: The identical permutation $\sigma' =_{\mathrm{df}} \sigma$ has property 1 trivially. It also enjoys property 2 since $\varepsilon = \bar{u}\bar{v}$ implies $\bar{u} = \bar{v} = \varepsilon$ and a suitable left-scattering of $\bar{u}$ in $z_1, \ldots, z_n$ is formed by taking $u_i = \varepsilon$ and $v_i = z_i$. As all u_i are empty, there are no completely marked z_i and thus it is a left-scattering.

- Case ua: We have to construct a permutation which satisfies property 1 and also property 2 (for any prefix of ua). By i.h. for u and av, we have a permutation $\sigma' = z'_1, \ldots, z'_n$ of σ satisfying properties 1 and 2 (for any prefix of u). From property 2 for $\bar{u} =_{\mathrm{df}} u$ and $\bar{v} =_{\mathrm{df}} \varepsilon$, we have that u is left-scattered in $z'_1, \ldots, z'_n$. Thus we have that $u \sim^{I*} u'_1 \ldots u'_n$, $z'_i \sim^{I*} u'_i v'_i$, $av \sim^{I*} v'_1 \ldots v'_n$ and $\forall j < i$, $v'_j \mathrel{I} u'_i$. This a is the leftmost a that has not been marked by u in $z'_1 \ldots z'_n$. In particular, it is in one of the v'_i, say v'_m. Since $av \sim^{I*} v'_1 \ldots v'_n$, there exists v''_m such that $v'_m \sim^{I*} av''_m$ and also $\forall j < m$, $v'_j \mathrel{I} a$.

  Next, we consider what happens when we mark this a in z'_m. For this we take $u''_m =_{\mathrm{df}} u'_m a$. We have $z'_m \sim^{I*} u''_m v''_m$. To emphasise, we only changed the marking of the word z'_m, everything else is the same. We now check whether the permutation σ' also satisfies property 2 with u''_m and v''_m, i.e., for $\bar{u} =_{\mathrm{df}} ua$ and $\bar{v} =_{\mathrm{df}} \varepsilon$.

  If $v''_m \neq \varepsilon$, then z'_m is not completely marked by the prefix ua. Hence σ' has property 2 also for the prefix ua.

If $v''_m = \varepsilon$, but $m = 1$ or $u_{m-1} \neq \varepsilon$, then σ' has property 2 also for the prefix ua.

In the critical case $v''_m = \varepsilon$, $m \neq 1$ and $u'_{m-1} = \varepsilon$, we construct a new permutation $\sigma'' =_{\mathrm{df}} z''_1, \ldots, z''_n$ from σ' by moving the block of words $z'_m, \ldots, z'_l$ (where we pick l the largest such that $v'_{m+1} \ldots v'_l = \varepsilon$) in front of $z'_k, \ldots, z'_{m-1}$ (where we pick k the smallest such that $u'_k \ldots u'_{m-1} = \varepsilon$). Moving this block of words rather than just z'_m alone ensures that the new permutation σ'' has property 2 also for all prefixes $\bar{u}$ of u and not just only for the prefix $\bar{u} =_{\mathrm{df}} ua$. The new permutation σ'' also has property 1. Indeed, σ' satisfies $z_1 \ldots z_n \sim^{I*} z'_1 \ldots z'_n$. Since $z'_k \ldots z'_{m-1} \sim^{I*} v'_k \ldots v'_{m-1}$, $v'_k \ldots v'_{m-1} \, I \, u''_m u'_{m+1} \ldots u'_l$ and $u''_m u'_{m+1} \ldots u'_l \sim^{I*} z'_m \ldots z'_l$, we have $z'_k \ldots z'_{m-1} \, I \, z'_m \ldots z'_l$. Hence $z'_1 \ldots z'_n \sim^{I*} z''_1 \ldots z''_n$. □

Proposition 3.55. *If E is star-connected, then the language $[\![E]\!]$ has uniform rank.*

Proof. By induction on E. We only look at the case E^*.

- Case E^*: By assumption, E is star-connected and $[\![E]\!]$ is connected. By i.h. $[\![E]\!]$ has uniform rank at most N for some $N \in \mathbb{N}$. We show that $[\![E^*]\!]$ has uniform rank at most $(|\Sigma| + 1)N$.

  Let $w \in [\![E^*]\!]^I$. Then exist n and $w_1, \ldots, w_n$ such that $w \in w_1 \cdot^I \ldots \cdot^I w_n$, $w_i \in [\![E]\!]^I$, and we can also require that $w_i \neq \varepsilon$. Since $[\![E]\!]$ has uniform rank at most N, then, for every i, there exists a nonempty word $z_i \in [\![E]\!]$ such that, for any split of w_i as $u_i v_i$, we have $u_i \sim \lhd_N z_i \rhd \sim v_i$. We can see that $w \sim^{I*} w_1 \ldots w_n \sim^{I*} z_1 \ldots z_n$. By connectedness of $[\![E]\!]$, all z_i are connected.

  Take $z^\circ =_{\mathrm{df}} z^\circ_1 \ldots z^\circ_n$ where $\sigma^\circ =_{\mathrm{df}} z^\circ_1, \ldots, z^\circ_n$ is the permutation of $\sigma = z_1, \ldots, z_n$ obtained by Lemma 3.54 for $u =_{\mathrm{df}} w$ and $v =_{\mathrm{df}} \varepsilon$. By Lemma 3.54(1), we have $w \sim^{I*} z_1 \ldots z_n \sim^{I*} z^\circ_1 \ldots z^\circ_n = z^\circ$. Thus $z^\circ \in [\![E^*]\!]$ is a witness for $w \in [\![E^*]\!]^I$.

  Next, we show that, for any split of w as $w = uv$, we have $u \sim \lhd_{(|\Sigma|+1)N} z^\circ \rhd \sim v$.

  Let $w = uv$ be any split of w. By Lemma 3.54(2), u is left-scattered in $z^\circ_1, \ldots, z^\circ_n$. Thus there exist $u^\circ_1, \ldots, u^\circ_n, v^\circ_1, \ldots, v^\circ_n$ such that $u \sim^{I*} u^\circ_1 \ldots u^\circ_n$, $v \sim^{I*} v^\circ_1 \ldots v^\circ_n$, for all i, $z^\circ_i \sim^{I*} u^\circ_i v^\circ_i$, and, for all $j < i$, $v^\circ_j \, I \, u^\circ_i$. Since $uv = w \sim^{I*} z^\circ$, we have $u \sim \lhd z^\circ \rhd \sim v$.

  As z°_i are nonempty and connected, by Lemma 3.52, at most $|\Sigma|$ of the z°_i can be partially marked (by u). Each of the partially marked z°_i contributes at most N to the degree of $u \sim \lhd z^\circ \rhd \sim v$, so altogether they contribute at most $|\Sigma|N$.

  Since u is left-scattered in $z^\circ_1, \ldots, z^\circ_n$, between any partially marked z°_i and also before the first and after the last one of them, there are some z°_i that are completely marked followed by some z°_i that are unmarked. Each such sequence contributes at most 1 to the degree of $u \sim \lhd z^\circ \rhd \sim v$. If there are less than $|\Sigma|$ partially marked z°_i, then these sequences thus contribute altogether at most $|\Sigma|$ to the degree. If there are exactly $|\Sigma|$ partially marked z°_i, then the z°_i after the last of them are all unmarked, so their sequence belongs to the last v-block generated by the last partially marked z°_i and thus contributes 0. Again, these sequences contribute at most $|\Sigma|$ to the degree.

  Altogether, the degree of $u \sim \lhd z^\circ \rhd \sim v$ is at most $(|\Sigma| + 1)N$. □

3.5 Antimirov Reordering Derivative and Uniform Rank

We have seen that the reordering language derivative $\mathsf{D}^I_u L$ allows u to be scattered in a word $z \in L$ as $u_1, \ldots, u_n \lhd z \rhd v_0, \ldots, v_n$ where $u \sim^{I*} u_1 \ldots u_n$. We now consider a version of the Antimirov reordering derivative operation that delivers lists of expressions for the possible $v_0, \ldots, v_n$ rather than just single expressions for their concatenations $v_0 \ldots v_n$.

3.5.1 Refined Antimirov Reordering Derivative

The refined reordering parts-of-derivative of an expression E along a letter a are pairs of expressions E_l, E_r. For any word $w = av \in [\![E]\!]^I$, there must be an equivalent word $z = v_l a v_r \in [\![E]\!]$. Instead of describing the words $v_l v_r$ obtainable by removing a minimal occurrence of a in a word $z \in [\![E]\!]$, the refined parts-of-derivative describe the subwords v_l, v_r that were to the left and right of this a in z: it must be the case that $v_l \in [\![E_l]\!]$ and $v_r \in [\![E_r]\!]$ for one of the pairs E_l, E_r. For a longer word u, the refined reordering derivative operation gives lists of expressions $E_0, \ldots, E_n$ fixing what the lists of subwords $v_0, \ldots, v_n$ can be in words $z = v_0 u_1 v_1 \ldots u_n v_n \in [\![E]\!]$ equivalent to a given word $w = uv \in [\![E]\!]^I$.

Definition 3.56. The *(unbounded and bounded) refined Antimirov I-reordering* parts-of-derivative of an expression along a letter and a word are given by relations $\to^I \subseteq \mathsf{RE} \times \Sigma \times \mathsf{RE} \times \mathsf{RE}$, $\Rightarrow^I \subseteq \mathsf{RE}^+ \times \Sigma \times \mathsf{RE}^+$, $\to^{I*} \subseteq \mathsf{RE} \times \Sigma^* \times \mathsf{RE}^+$, $\Rightarrow^I_N \subseteq \mathsf{RE}^{+\leq N+1} \times \Sigma \times \mathsf{RE}^{+\leq N+1}$, and $\to^{I*}_N \subseteq \mathsf{RE} \times \Sigma^* \times \mathsf{RE}^{+\leq N+1}$ defined inductively by

$$\frac{}{a \to^I (a; 1, 1)} \quad \frac{E \to^I (a; E_l, E_r)}{E + F \to^I (a; E_l, E_r)} \quad \frac{F \to^I (a; F_l, F_r)}{E + F \to^I (a; F_l, F_r)}$$

$$\frac{E \to^I (a; E_l, E_r)}{EF \to^I (a; E_l, E_r F)} \quad \frac{F \to^I (a; F_l, F_r)}{EF \to^I (a; (\mathsf{R}^I_a E) F_l, F_r)} \quad \frac{E \to^I (a; E_l, E_r)}{E^* \to^I (a; (\mathsf{R}^I_a E)^* E_l, E_r E^*)}$$

$$\frac{E \to^I (a; E_l, E_r) \quad |\Gamma, \Delta| < N}{\Gamma, E, \Delta \Rightarrow^I_N (a; \mathsf{R}^I_a \Gamma, E_l, E_r, \Delta)} \quad \frac{E \to^I (a; E_l, E_r) \quad E_l \not\downarrow \quad |\Gamma| > 0}{\Gamma, E, \Delta \Rightarrow^I_N (a; \mathsf{R}^I_a \Gamma, E_r, \Delta)}$$

$$\frac{E \to^I (a; E_l, E_r) \quad E_r \not\downarrow \quad |\Delta| > 0}{\Gamma, E, \Delta \Rightarrow^I_N (a; \mathsf{R}^I_a \Gamma, E_l, \Delta)} \quad \frac{E \to^I (a; E_l, E_r) \quad E_l \not\downarrow \quad E_r \not\downarrow \quad |\Gamma| > 0 \quad |\Delta| > 0}{\Gamma, E, \Delta \Rightarrow^I_N (a; \mathsf{R}^I_a \Gamma, \Delta)}$$

$$\frac{}{E \to^{I*}_N (\varepsilon; E)} \quad \frac{E \to^{I*}_N (u; \Gamma) \quad \Gamma \Rightarrow^I_N (a; \Gamma')}{E \to^{I*}_N (ua; \Gamma')}$$

By $\mathsf{RE}^{+\leq N+1}$ we mean nonempty lists of expressions of length at most $N+1$. The relations $\Rightarrow^I$ and $\to^{I*}$ are defined exactly as $\Rightarrow^I_N$ and $\to^{I*}_N$ but with the condition $|\Gamma, \Delta| < N$ of the first rule of $\Rightarrow^I_N$ dropped. The operation R^I_a is extended to lists of expressions by applying R^I_a to every expression in the list.

We have several rules for deriving a list of expressions along a. This is so that we can truncate the list precisely—only lists of length at most $N+1$ can be derived (in the bounded case).

If E is split into E_l, E_r (by $\to^I$) and neither of them is nullable, then, in the N-bounded case, we require that the given list (Γ, E, Δ) is shorter than $N+1$ since the new list (this is $\mathsf{R}^I_a \Gamma, E_l, E_r, \Delta$ when deriving along a) will be longer by one. If one of E_l, E_r is nullable, not the first resp. last in the list, and we choose to drop it, then the new list will be of the same length. If both are nullable, not the first resp. last, and we opt to drop both, then the new list will be shorter by one. They must be droppable under these conditions to handle the situation when a word z has been split as $v_0 u_1 v_1 \ldots u_k v_k u_{k+1} \ldots u_n v_n$ and v_k is further being split as $v_l a v_r$ while v_l or v_r is empty. If $k \neq 0$ and v_l is empty, we must join u_k and a into $u_k a$. If $k \neq n$ and v_r is empty, we must join a and u_{k+1} into $a u_{k+1}$. If k is neither 0 nor n and both v_l and v_r are empty, we must join all three of u_k, a and u_{k+1} into $u_k a u_{k+1}$. The length of the new list of expressions is always at least 2.

Proposition 3.57. *For any E,*

1. *for any $a \in \Sigma, v_l, v_r \in \Sigma^*$,*

$$v_l \, I \, a \wedge v_l a v_r \in [\![E]\!] \iff \exists E_l, E_r . E \to^I (a; E_l, E_r) \wedge v_l \in [\![E_l]\!] \wedge v_r \in [\![E_r]\!];$$

2. *for any $u \in \Sigma^*, n \in \mathbb{N}, v_0 \in \Sigma^*, v_1, \ldots, v_{n-1} \in \Sigma^+, v_n \in \Sigma^*$,*

$$\begin{array}{c} \exists z \in [\![E]\!], u_1, \ldots, u_n \in \Sigma^+ . u \sim^{I*} u_1 \ldots u_n \wedge u_1, \ldots, u_n \lhd z \rhd v_0, \ldots, v_n \\ \iff \\ \exists E_0, \ldots, E_n . E \to^{I*} (u; E_0, \ldots, E_n) \wedge \forall j .\, v_j \in [\![E_j]\!]. \end{array}$$

Proof.

1. $\Longrightarrow$: By induction on E.

   - Case a' where $a' \neq a$: $v_l a v_r \in [\![a']\!] = \{a'\}$ is impossible.
   - Case a: Suppose $v_l a v_r \in [\![a]\!] = \{a\}$. Then $v_l = v_r = \varepsilon$. We have $a \to^I (a; 1, 1)$ and $\varepsilon \in [\![1]\!]$ as required.
   - Case 0: $v_l a v_r \in [\![0]\!] = \emptyset$ is impossible.
   - Case $E_1 + E_2$: Suppose $v_l a v_r \in [\![E_1 + E_2]\!] = [\![E_1]\!] \cup [\![E_2]\!]$ and $v_l \, I \, a$. It must be that $v_l a v_r \in [\![E_i]\!]$ for one of two possible i. By i.h. for E_i, a, v_l, v_r, there are E_l, E_r such that $E_i \to^I (a; E_l, E_r)$, $v_l \in [\![E_l]\!]$, $v_r \in [\![E_r]\!]$ and from which we obtain $E_1 + E_2 \to^I (a; E_l, E_r)$.
   - Case 1: $v_l a v_r \in [\![1]\!] = \mathbf{1}$ is impossible.
   - Case EF: Suppose $v_l a v_r \in [\![EF]\!] = [\![E]\!] \cdot [\![F]\!]$ and $v_l \, I \, a$. Then $v_l a v_r = xy$ for some $x \in [\![E]\!]$ and $y \in [\![F]\!]$. Either (i) there exists v' such that $x = v_l a v'$ and $v_r = v' y$ or (ii) there exists v' such that $y = v' a v_r$ and $v_l = x v'$.
     If (i), then, by i.h. for E, a, v_l, v', there are E_l, E_r such that $E \to^I (a; E_l, E_r)$ and $v_l \in [\![E_l]\!]$, $v' \in [\![E_r]\!]$. Thus we have $EF \to^I (a; E_l, E_r F)$ and $v_r = v' y \in [\![E_r F]\!]$.
     If (ii), then $x \, I \, a$ and $v' \, I \, a$, so $x \in [\![\mathsf{R}^I_a E]\!]$ and, by i.h. for F, a, v', v_r, there are F_l, F_r such that $F \to^I (a; F_l, F_r)$, and $v' \in [\![F_l]\!]$, $v_r \in [\![F_r]\!]$. From these we obtain $EF \to^I (a; (\mathsf{R}^I_a E) F_l, F_r)$ and $v_l = x v' \in [\![(\mathsf{R}^I_a E) F_l]\!]$.
   - Case E^*: Suppose $v_l a v_r \in [\![E^*]\!]$ and $v_l \, I \, a$. Then $v_l = x v'_l$ and $v_r = v'_r y$ for some $x, y \in [\![E^*]\!]$ and v'_l, v'_r such that $v'_l a v'_r \in [\![E]\!]$. As $x \, I \, a$ and $v'_l \, I \, a$ we have that $x \in [\![\mathsf{R}^I_a E^*]\!]$ and, by i.h. for E, a, v'_l, v'_r, we get that there are E_l, E_r such that $E \to^I (a; E_l, E_r)$ and $v'_l \in [\![E_l]\!]$, $v'_r \in [\![E_r]\!]$. We also obtain $v_l = x v'_l \in [\![(\mathsf{R}^I_a E^*) E_l]\!]$ and $v_r = v'_r y \in [\![E_r E^*]\!]$.

   $\Longleftarrow$: By induction on the derivation of $E \to^I (a; E_l, E_r)$.

   - Case $a \to^I (a; 1, 1)$ as an axiom: Suppose $v_l, v_r \in [\![1]\!] = \mathbf{1}$. Then $v_l = v_r = \varepsilon$ and we have $\varepsilon a \varepsilon = a \in [\![a]\!]$ and $\varepsilon \, I \, a$ as required.
   - Case $E_1 + E_2 \to^I (a; E_l, E_r)$ inferred from $E_i \to^I (a; E_l, E_r)$ where i is 1 or 2: Suppose $v_l \in [\![E_l]\!]$, $v_r \in [\![E_r]\!]$. We apply i.h. to the subderivation, v_l, v_r and obtain $v_l \, I \, a$ and $v_l a v_r \in [\![E_i]\!]$. Thus $v_l a v_r \in [\![E_1]\!] \cup [\![E_2]\!] = [\![E_1 + E_2]\!]$.
   - Case $EF \to^I (a; E_l, E_r F)$ inferred from $E \to^I (a; E_l, E_r)$: Suppose $v_l \in [\![E_l]\!]$, $v_r \in [\![E_r F]\!] = [\![E_r]\!] \cdot [\![F]\!]$. Then $v_r = xy$ for some $x \in [\![E_r]\!]$ and $y \in [\![F]\!]$. We apply i.h. to the subderivation, v_l, x and obtain that $v_l \, I \, a$ and $v_l a x \in [\![E]\!]$. As $v_r = xy$, we have $v_l a v_r = (v_l a x) y \in [\![E]\!] \cdot [\![F]\!] = [\![EF]\!]$.

- Case $EF \to^I (a, (\mathsf{R}^I_a E)F_l, F_r)$ inferred from $F \to^I (a; F_l, F_r)$: Suppose that $v_l \in [\![(\mathsf{R}^I_a E)F_l]\!] = \mathsf{R}^I_a[\![E]\!] \cdot [\![F_l]\!]$, $v_r \in [\![F_r]\!]$. Then $v_l = xy$ for some $x \in \mathsf{R}^I_a[\![E]\!]$ and $y \in [\![F_l]\!]$, which means $x\, I\, a$ and $x \in [\![E]\!]$. We apply i.h. to the subderivation, y, v_r and obtain that $y\, I\, a$ and $yav_r \in [\![F]\!]$. As $v_l = xy$, we have $v_l\, I\, a$ and thus $v_l a v_r = x(yav_r) \in [\![E]\!] \cdot [\![F]\!] = [\![EF]\!]$.
- Case $E^* \to^I (a; (\mathsf{R}^I_a E^*)E_l, E_r E^*)$ inferred from $E \to^I (a; E_l, E_r)$: Suppose that $v_l \in [\![(\mathsf{R}^I_a E^*)E_l]\!] = \mathsf{R}^I_a[\![E^*]\!] \cdot [\![E_l]\!]$, $v_r \in [\![E_r E^*]\!] = [\![E_r]\!] \cdot [\![E^*]\!]$. Then $v_l = xy$ for some $x \in \mathsf{R}^I_a[\![E^*]\!]$ and $y \in [\![E_l]\!]$, which means $x\, I\, a$ and $x \in [\![E^*]\!]$. Also, $v_r = zw$ for some $z \in [\![E_r]\!]$ and $w \in [\![E^*]\!]$. We apply i.h. to the subderivation, y, z and obtain that $y\, I\, a$ and $yaz \in [\![E]\!]$. As $v_l = xy$ and $v_r = zw$, we get that $v_l\, I\, a$ and $v_l a v_r = x(yaz)w \in [\![E^*]\!] \cdot [\![E]\!] \cdot [\![E^*]\!] \subseteq [\![E^*]\!]$.

2. $\Longrightarrow$: By induction on u.

   - Case ε: Suppose $\varepsilon \sim^{I*} u_1 \dots u_n$ and $z = v_0 u_1 v_1 \dots u_n v_n \in [\![E]\!]$. As the u_i are nonempty, then necessarily $n = 0$, which means we have $v_0 \in [\![E]\!]$. Also, we have $E \to^{I*} (\varepsilon; E)$ as required.
   - Case ua: Suppose $ua \sim^{I*} u_1 \dots u_n$, $\forall i. \forall j < i.\, v_j\, I\, u_i$ and $z = v_0 u_1 v_1 \dots u_n v_n \in [\![E]\!]$. It must be that $n > 0$ and there must exist k and $u_l, u_r \in \Sigma^*$ such that $u_k = u_l a u_r$, $v_j\, I\, a$ for all $j < k$, $a\, I\, u_r$, $a\, I\, u_i$ for all $i > k$ and $u \sim^{I*} u_1 \dots u_{k-1} u_l u_r u_{k+1} \dots u_n$.
     - If $u_l = u_r = \varepsilon$, then, as $v_{k-1} a v_k\, I\, u_i$ for all $i > k$, we can apply i.h. to u, $n-1$, $v_0, v_1, \dots, v_{k-2}, v_{k-1}av_k, v_{k+1}, \dots, v_n$, z, $u_1, \dots, u_{k-1}, u_{k+1}, \dots, u_n$. We get $E_0, \dots, E_{k-2}$, E', $E_{k+1}, \dots, E_n$ such that

       $$E \to^{I*} (u; E_0, \dots, E_{k-2}, E', E_{k+1}, \dots, E_n)$$

       and $v_j \in [\![E_j]\!]$ for all $j < k-1$, $v_{k-1} a v_k \in [\![E']\!]$, $v_j \in [\![E_j]\!]$ for all $j > k$. As $v_{k-1}\, I\, a$, we can apply 1. to E', a, v_{k-1}, v_k and get E_{k-1}, E_k such that $E' \to^I (a; E_{k-1}, E_k)$, $v_{k-1} \in [\![E_{k-1}]\!]$ and $v_k \in [\![E_k]\!]$. This allows us to infer

       $$E \to^{I*} (ua; \mathsf{R}^I_a E_0, \dots, \mathsf{R}^I_a E_{k-2}, E_{k-1}, E_k, E_{k+1}, \dots, E_n).$$

       As $v_j\, I\, a$ for all $j < k-1$, we also have $v_j \in [\![\mathsf{R}^I_a E_j]\!]$ for all $j < k-1$.
     - If $u_l \neq \varepsilon$, $u_r = \varepsilon$, we note that $a v_k\, I\, u_i$ for all $i > k$ and apply i.h. to u, n, $v_0, v_1, \dots, v_{k-1}, av_k, v_{k+1}, \dots, v_n$, z, $u_1, \dots, u_{k-1}, u_l, u_{k+1}, \dots, u_n$. We get $E_0, \dots, E_{k-1}$, E', $E_{k+1}, \dots, E_n$ such that

       $$E \to^{I*} (u; E_0, \dots, E_{k-1}, E', E_{k+1}, \dots, E_n)$$

       and $v_j \in [\![E_j]\!]$ for all $j < k$, $av_k \in [\![E']\!]$, $v_j \in [\![E_j]\!]$ for all $j > k$. As $\varepsilon\, I\, a$, we can apply 1. to E', a, ε, v_k and get E'', E_k such that $E' \to^I (a; E'', E_k)$, $\varepsilon \in [\![E'']\!]$, $v_k \in [\![E_k]\!]$. As $\varepsilon \in [\![E'']\!]$ tells us that $E'' \not\downarrow$, we can infer

       $$E \to^{I*} (ua; \mathsf{R}^I_a E_0, \dots, \mathsf{R}^I_a E_{k-1}, E_k, E_{k+1}, \dots, E_n).$$

       As $v_j\, I\, a$ also for all $j < k$, we in fact also have $v_j \in [\![\mathsf{R}^I_a E_j]\!]$ for all $j < k$.
     - The cases $u_l = \varepsilon$, $u_r \neq \varepsilon$ and $u_l \neq \varepsilon$, $u_r \neq \varepsilon$ are handled similarly to the previous case.

$\Longleftarrow$: By induction on the derivation of $E \to^{I*} (u; E_0, \ldots, E_n)$.

- Case $E \to^{I*} (\varepsilon; E)$ as an axiom:
  Here $n = 0$ and $E_0 = E$. Suppose that $v_0 \in [\![E]\!]$. We have $\varepsilon \sim^{I*} \varepsilon$ as well as $z = v_0 \in [\![E]\!]$ directly.
- Case $E \to^{I*} (ua; \mathsf{R}^I_a E_0, \ldots, \mathsf{R}^I_a E_{k-1}, E_l, E_r, E_{k+1}, \ldots, E_n)$ inferred from derivations $E \to^{I*} (u; E_0, \ldots, E_n)$ and $E_k \to^I (a; E_l, E_r)$:
  Suppose that $v_0 \in [\![\mathsf{R}^I_a E_0]\!], \ldots, v_{k-1} \in [\![\mathsf{R}^I_a E_{k-1}]\!]$, $v_l \in [\![E_l]\!]$, $v_r \in [\![E_r]\!]$, and that $v_{k+1} \in [\![E_{k+1}]\!], \ldots, v_n \in [\![E_n]\!]$. From this we also have $v_0 \mathrel{I} a, \ldots, v_{k-1} \mathrel{I} a$ and $v_0 \in [\![E_0]\!], \ldots, v_{k-1} \in [\![E_{k-1}]\!]$.
  By applying (1.$\Longleftarrow$) to E_k, a, v_l, v_r, we learn that $v_l \mathrel{I} a$ and $v_l a v_r \in [\![E_k]\!]$.
  By applying i.h. to u, n, $v_0, \ldots, v_{k-1}$, $v_l a v_r$, $v_{k+1}, \ldots, v_n$ and the subderivation, we obtain the words $z, u_1, \ldots, u_n \in \Sigma^+$ such that $u \sim^{I*} u_1 \ldots u_n$, $\forall i. \forall j < i. v_j \mathrel{I} u_i$, $\forall i > k. v_l a v_r \mathrel{I} u_i$ and $z = v_0 u_1 v_1 \ldots v_{k-1} u_k (v_l a v_r) u_{k+1} v_{k+1} \ldots u_n v_n \in [\![E]\!]$.
  Now clearly $ua \sim^{I*} u_1 \ldots u_k a u_{k+1} \ldots u_n$ and
  $$z = v_0 u_1 v_1 \ldots v_{k-1} u_k v_l a v_r u_{k+1} v_{k+1} \ldots u_n v_n \in [\![E]\!].$$
- Case $E \to^{I*} (ua; \mathsf{R}^I_a E_0, \ldots, \mathsf{R}^I_a E_{k-1}, E_r, E_{k+1}, \ldots, E_n)$ inferred from derivations $E \to^{I*} (u; E_0, \ldots, E_n)$ and $E_k \to^I (a; E_l, E_r)$ with $E_l \not\downarrow$ whereby $k \neq 0$:
  Let us now suppose now that $v_0 \in [\![\mathsf{R}^I_a E_0]\!], \ldots, v_{k-1} \in [\![\mathsf{R}^I_a E_{k-1}]\!]$, $v_r \in [\![E_r]\!]$ and $v_{k+1} \in [\![E_{k+1}]\!], \ldots, v_n \in [\![E_n]\!]$. From this we also have $v_0 \mathrel{I} a, \ldots, v_{k-1} \mathrel{I} a$ and $v_0 \in [\![E_0]\!], \ldots, v_{k-1} \in [\![E_{k-1}]\!]$.
  Applying (1.$\Longleftarrow$) to E_k, a, ε, v_r, we learn that $a v_r \in [\![E_k]\!]$.
  Applying i.h. to u, n, $v_0, \ldots, v_{k-1}, a v_r, v_{k+1}, \ldots, v_n$ and the subderivation, we obtain $z, u_1, \ldots, u_n \in \Sigma^+$ such that $u \sim^{I*} u_1 \ldots u_n$, $\forall i. \forall j < i. v_j \mathrel{I} u_i$, $\forall i > k. a v_r \mathrel{I} u_i$ and $z = v_0 u_1 v_1 \ldots v_{k-1} u_k (a v_r) u_{k+1} v_{k+1} \ldots u_n v_n \in [\![E]\!]$.
  Now clearly $ua \sim^{I*} u_1 \ldots (u_k a) u_{k+1} \ldots u_n$ and
  $$z = v_0 u_1 v_1 \ldots v_{k-1} (u_k a) v_r u_{k+1} v_{k+1} \ldots u_n v_n \in [\![E]\!].$$
- The two remaining cases are treated similarly to the previous case. □

Proposition 3.58. *For any E,*

1. *for any $a \in \Sigma, v \in \Sigma^*$, the following are equivalent:*
   - *(a)* $av \in [\![E]\!]^I$;
   - *(b)* $\exists v_l, v_r \in \Sigma^*. v \sim^{I*} v_l v_r \wedge v_l \mathrel{I} a \wedge v_l a v_r \in [\![E]\!]$;
   - *(c)* $\exists v_l, v_r \in \Sigma^*.$ $v \sim^{I*} v_l v_r \wedge \exists E_l, E_r. E \to^I (a; E_l, E_r) \wedge v_l \in [\![E_l]\!] \wedge v_r \in [\![E_r]\!]$;
   - *(d)* $\exists v_l, v_r \in \Sigma^*.$ $v \in v_l \mathbin{.^I} v_r \wedge \exists E_l, E_r. E \to^I (a; E_l, E_r) \wedge v_l \in [\![E_l]\!]^I \wedge v_r \in [\![E_r]\!]^I$.
2. *for any $u, v \in \Sigma^*$, the following are equivalent:*
   - *(a)* $uv \in [\![E]\!]^I$;
   - *(b)* $\exists z \in [\![E]\!]. u \sim \lhd z \rhd \sim v$;

(c) $\exists n \in \mathbb{N}, v_0 \in \Sigma^*, v_1, \ldots, v_{n-1} \in \Sigma^+, v_n \in \Sigma^*. v \sim^{I*} v_0 v_1 \ldots v_n \wedge$
$\exists E_0, \ldots, E_n. E \to^{I*} (u; E_0, \ldots, E_n) \wedge \forall j. v_j \in [\![E_j]\!]$;

(d) $\exists n \in \mathbb{N}, v_0 \in \Sigma^*, v_1, \ldots, v_{n-1} \in \Sigma^+, v_n \in \Sigma^*. v \in v_0 \cdot^I v_1 \cdot^I \ldots \cdot^I v_n \wedge$
$\exists E_0, \ldots, E_n. E \to^{I*} (u; E_0, \ldots, E_n) \wedge \forall j. v_j \in [\![E_j]\!]^I$.

3. *for any* $u \in \Sigma^*$,

$$u \in [\![E]\!]^I \iff (u = \varepsilon \wedge E\!\downarrow) \vee (u \neq \varepsilon \wedge \exists E_0, E_1. E \to^{I*} (u; E_0, E_1) \wedge E_0\!\downarrow \wedge E_1\!\downarrow).$$

Proof.

1. (a) $\iff$ (b) follows from Proposition 3.6.
   (b) $\iff$ (c) follows from Proposition 3.57(1).
   (c) $\iff$ (d) follows from Proposition 3.10.
2. (a) $\iff$ (b) follows from Proposition 3.6.
   (b) $\iff$ (c) follows from Proposition 3.57(2).
   (c) $\iff$ (d) follows from Proposition 3.10.
3. From (2) for E, u, ε. □

We have thus established a correspondence between the trace-closing interpretation of regular expressions and refined Antimirov reordering derivative. We continue towards a similar property for expressions defining languages with uniform rank.

Proposition 3.59. *For any* E, $N \in \mathbb{N}$, $u \in \Sigma^*$, $z \in [\![E]\!]$,
if

$$\forall u', u''. u = u'u'' \Longrightarrow \exists v. u' \sim \lhd_N z \rhd v$$

then

$$\exists E_0, \ldots, E_n. E \to_N^{I*} (u; E_0, \ldots, E_n) \wedge \forall j.\ v_j \in [\![E_j]\!]$$

for the unique n, $u_1, \ldots, u_n$, $v_0, \ldots, v_n$ *such that* $u \sim^{I*} u_1 \ldots u_n$ *and* $u_1, \ldots, u_n \lhd z \rhd v_0, \ldots, v_n$.

Proof. This is basically a replay of the proof of Proposition 3.57(2.$\Longrightarrow$) with the extra assumption about uniform rank up to u. The proof is by induction on u.

- Case ε: Since $u = \varepsilon$, it must be that $n = 0$ in $u_1, \ldots, u_n \lhd z \rhd v_0, \ldots, v_n$. Hence $v_0 = z$. We take $E_0 = E$ and thus have $E \to_N^{I*} (\varepsilon; E)$ and $v_0 \in [\![E]\!]$.
- Case ua: By assumption (for $u' = ua$, $u'' = \varepsilon$), we have v such that $ua \sim \lhd_N z \rhd v$. Thus we have n, $u_1, \ldots, u_n, v_0, \ldots, v_n$ such that $n \leq N$, $ua \sim^{I*} u_1 \ldots u_n$, $v_0 \ldots v_n = v$ and ua is scattered in z as follows.

  $$u_1, \ldots, u_n \lhd z \rhd v_0, \ldots, v_n$$

  Since $ua \sim^{I*} u_1 \ldots u_n$, the letter a must be in one of the u_i, i.e., there exist k, u_l and u_r such that $u_k = u_l a u_r$, $\forall i < k. v_i \, I \, a$, $a \, I \, u_r$ and $\forall i > k. a \, I \, u_i$.

  Next, we construct from the above scattering of ua in z a scattering of u in z by splitting u_k into u_l, a, u_r and considering a to be a v-letter. To construct a valid scattering, we have four cases to consider: $u_l = \varepsilon$ and $u_r = \varepsilon$; $u_l = \varepsilon$ and $u_r \neq \varepsilon$; $u_l \neq \varepsilon$ and $u_r = \varepsilon$; $u_l \neq \varepsilon$ and $u_r \neq \varepsilon$. We only consider the last case here.

Since $u_l \neq \varepsilon$ and $u_r \neq \varepsilon$, we have that the following is a valid scattering of u in z as $a\,I\,u_r$ and $\forall i > k.\, a\,I\,u_i$.

$$u_1, \ldots, u_{k-1}, u_l, u_r, u_{k+1}, \ldots, u_n \lhd z \rhd v_0, \ldots, v_{k-1}, a, v_k, \ldots, v_n$$

This scattering, however, has degree $n+1$. By Lemma 3.4, we know that scatterings are unique, i.e., if a word u is scattered in a word z, then this is the only way to scatter u in z. This implies, by the assumption of uniform rank (for $u' = u$ and $u'' = a$), that $n+1 \leq N$.

Since the assumption of uniform rank also holds for all prefixes of u, we can apply i.h. to E, N, u, z. Since the above scattering is the unique scattering of u in z, we obtain $E_0, \ldots, E_{k-1}, E', E_k, \ldots, E_n$ such that

$$E \to_N^{I*} (u; E_0, \ldots, E_{k-1}, E', E_k, \ldots, E_n)$$

and $v_0 \in [\![E_0]\!], \ldots, v_{k-1} \in [\![E_{k-1}]\!], a \in [\![E']\!], v_k \in [\![E_k]\!], \ldots, v_n \in [\![E_n]\!]$.

As $\varepsilon\,I\,a$ and $a \in [\![E']\!]$, we can apply Proposition 3.57(1.$\Longrightarrow$) to E', a, ε, ε to obtain E'_l, E'_r such that $E' \to^I (a; E'_l, E'_r)$, $\varepsilon \in [\![E'_l]\!]$ and $\varepsilon \in [\![E'_r]\!]$. Thus $E'_l \not\downarrow$ and $E'_r \not\downarrow$. As $u_l \neq \varepsilon$ and $u_r \neq \varepsilon$ imply $|E_0, \ldots, E_{k-1}| > 0$ and $|E_k, \ldots, E_n| > 0$, we can use the fourth rule of $\Rightarrow_N^I$ to drop the nullable expressions E'_l and E'_r when deriving along a.

$$E_0, \ldots, E_{k-1}, E', E_k, \ldots, E_n \Rightarrow_N^I (a; \mathsf{R}_a^I E_0, \ldots, \mathsf{R}_a^I E_{k-1}, E_k, \ldots, E_n)$$

This allows us to conclude the following.

$$E \to_N^{I*} (ua; \mathsf{R}_a^I E_0, \ldots, \mathsf{R}_a^I E_{k-1}, E_k, \ldots, E_n)$$

As we have $v_j \in [\![E_j]\!]$ and also $v_j\,I\,a$ for all $j < k$, we thus have $v_j \in [\![\mathsf{R}_a^I E_j]\!]$. □

Corollary 3.60. *For any E such that $[\![E]\!]$ has uniform rank at most N,*

1. *for any $u, v \in \Sigma^*$, the following are equivalent:*

   (a) $uv \in [\![E]\!]^I$;

   (b) $\exists z \in [\![E]\!].\, \forall u', u''.\, u = u'u'' \Longrightarrow u' \sim \lhd_N z \rhd \sim u''v$;

   (c) $\exists n \leq N, v_0 \in \Sigma^*, v_1, \ldots, v_{n-1} \in \Sigma^+, v_n \in \Sigma^*.\, v \sim^{I*} v_0 v_1 \ldots v_n \wedge$
   $\exists E_0, \ldots, E_n.\, E \to_N^{I*} (u; E_0, \ldots, E_n) \wedge \forall j.\, v_j \in [\![E_j]\!]$;

   (d) $\exists n \leq N, v_0 \in \Sigma^*, v_1, \ldots, v_{n-1} \in \Sigma^+, v_n \in \Sigma^*.\, v \in v_0 \cdot^I v_1 \cdot^I \ldots \cdot^I v_n \wedge$
   $\exists E_0, \ldots, E_n.\, E \to_N^{I*} (u; E_0, \ldots, E_n) \wedge \forall j.\, v_j \in [\![E_j]\!]^I$.

2. *for any $u \in \Sigma^*$,*

$$u \in [\![E]\!]^I \iff (u = \varepsilon \wedge E \not\downarrow) \vee (u \neq \varepsilon \wedge \exists E_0, E_1.\, E \to_N^{I*} (u; E_0, E_1) \wedge E_0 \not\downarrow \wedge E_1 \not\downarrow).$$

Proof of 1.
(a) $\Longrightarrow$ (b) is from E having uniform rank at most N.
(b) $\Longrightarrow$ (c) follows from Proposition 3.59.
(c) $\Longrightarrow$ (d) and (d) $\Longrightarrow$ (a) are those from Proposition 3.58(2). □

Example 3.61. Let $\Sigma = \{a,b\}$, aIb, $E =_{\mathrm{df}} aa+ab+b$ and $E_b =_{\mathrm{df}} \mathsf{R}_b^I E = aa+a0+0$. There are two possibilities to derive the expression E along b: either from the summand ab or from the summand b. Here are the two derivations.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{b \to^I (b;1,1)}}{ab \to^I (b;a1,1)}}{ab+b \to^I (b;a1,1)}}{aa+ab+b \to^I (b;a1,1)}}{E^* \to^I (b;E_b^*(a1),1E^*)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{b \to^I (b;1,1)}}{ab+b \to^I (b;1,1)}}{aa+ab+b \to^I (b;1,1)}}{E^* \to^I (b;E_b^*1,1E^*)}$$

By denoting the left derivation by D_{ab} and the right one by D_b, we can write one of the refined reordering parts-of-derivative of E^* along bb as follows.

$$\dfrac{\dfrac{}{E^* \to_2^{I*} (\varepsilon;E^*)} \quad \dfrac{\overset{D_{ab}}{\cdots} \quad 0<2}{E^* \Rightarrow_2^I (b;E_b^*(a1),1E^*)}}{E^* \to_2^{I*} (b;E_b^*(a1),1E^*)} \quad \dfrac{\dfrac{\overset{D_{ab}}{\cdots}}{1E^* \to^I (b;1(E_b^*(a1)),1E^*)} \quad 1<2}{E_b^*(a1),1E^* \Rightarrow_2^I (b;E_b^*(a1),1(E_b^*(a1)),1E^*)}}{E^* \to_2^{I*} (bb;E_b^*(a1),1(E_b^*(a1)),1E^*)}$$

In this example we chose $N =_{\mathrm{df}} 2$ and we chose the ab summand both times. The expression $1(E_b^*(a1)) \doteq (aa)^*a$ is not nullable, so we could not have dropped it in the $\Rightarrow_2^I$ rule. From here, we cannot continue by deriving along a third b by again taking it from the summand ab of E in $1E^*$, as this would produce another nondroppable $1(E_b^*(a1))$ and make the list too long (longer than 3).

If we would choose, for example, the summand b for the second step, then we could arrive at a list of length one less.

$$\dfrac{\dfrac{}{E^* \to_2^{I*} (\varepsilon;E^*)} \quad \dfrac{\overset{D_{ab}}{\cdots} \quad 0<2}{E^* \Rightarrow_2^I (b;E_b^*(a1),1E^*)}}{E^* \to_2^{I*} (b;E_b^*(a1),1E^*)} \quad \dfrac{\dfrac{\overset{D_b}{\cdots}}{1E^* \to^I (b;1(E_b^*1),1E^*)} \quad 1(E_b^*1)\not\downarrow \quad 1>0}{E_b^*(a1),1E^* \Rightarrow_2^I (b;E_b^*(a1),1E^*)}}{E^* \to_2^{I*} (bb;E_b^*(a1),1E^*)}$$

For example, we are not allowed to establish (for $N=2$) that $w =_{\mathrm{df}} bbbaaa \in [\![E^*]\!]^I$ by deriving E^* along w and checking that we can arrive at E_0, E_1 with both E_0, E_1 nullable as mandated by $z =_{\mathrm{df}} ababab \in [\![E^*]\!]$. We are allowed to do so because $z' =_{\mathrm{df}} bbabaa \in [\![E^*]\!]$.

The word z is not useful since, among the splits of w as $w=uv$, there is $u =_{\mathrm{df}} bbb$, $v =_{\mathrm{df}} aaa$, in which u scatters into z in three blocks as $z = a\underline{b}a\underline{b}a\underline{b}$ (we underline the letters from u). The full sequence of scatterings corresponding to every split of w is: $ababab$, $a\underline{b}abab$, $a\underline{b}a\underline{b}ab$, $a\underline{b}a\underline{b}a\underline{b}$, $ab\underline{a}\underline{b}ab$, $ab\underline{a}b\underline{a}b$, $\underline{a}\underline{b}a\underline{b}a\underline{b}$.

The word z', on the contrary, is fine because, for every split of w as $w=uv$, the word u scatters into z using at most two blocks. The full sequence is: $bbabaa$, $\underline{b}babaa$, $\underline{bb}abaa$, $bba\underline{b}aa$, $\underline{bba}baa$, $\underline{bbab}aa$, $\underline{bbabaa}$.

The choice $N=2$ suffices for accepting all of $[\![E^*]\!]^I$, since $[\![E^*]\!]$ happens to have uniform rank 2.

Just as in the non-refined case, the refined Antimirov reordering parts-of-derivative of an expression E give a nondeterministic automaton by taking the state set to be $Q^E =_{\mathrm{df}} \{\Gamma \mid \exists u \in \Sigma^*.\, E \to^{I*} (u;\Gamma)\}$, the initial state to be $q_0^E =_{\mathrm{df}} E$, the final states to be $F^E =_{\mathrm{df}} \{E \mid E\not\downarrow\} \cup \{E_0, E_1 \in Q^E \mid E_0\not\downarrow \wedge E_1\not\downarrow\}$, and the transitions to be given by $\Gamma \to^E (a;\Gamma') =_{\mathrm{df}} \Gamma \Rightarrow^I (a;\Gamma')$ for $\Gamma, \Gamma' \in Q^E$. By Proposition 3.58, this automaton accepts $[\![E]\!]^I$. It is generally not finite as Q^E can contain states Γ of any length.

Given $N \in \mathbb{N}$, another automaton is obtained by restricting Q^E, F^E and $\to^E$ to $Q^E_N =_{\mathrm{df}} \{\Gamma \mid \exists u \in \Sigma^*.\, E \to^{I*}_N (u;\Gamma)\}$, $F^E_N =_{\mathrm{df}} \{E \mid E \not\downarrow\} \cup \{E_0, E_1 \in Q^E_N \mid E_0 \not\downarrow \wedge E_1 \not\downarrow\}$, $\Gamma \to^E_N (a;\Gamma') =_{\mathrm{df}} \Gamma \Rightarrow^I_N (a;\Gamma')$ for $\Gamma, \Gamma' \in Q^E_N$. By Corollary 3.60, if $[\![E]\!]$ has uniform rank at most N, then this smaller automaton accepts $[\![E]\!]^I$ despite the truncation. If $[\![E]\!]$ does not have uniform rank or we choose N smaller than the uniform rank, then the N-truncated automaton recognises a proper subset of $[\![E]\!]^I$. In other words, by increasing N we get a better (regular) approximation of $[\![E]\!]^I$, but if $[\![E]\!]$ does not have uniform rank, then we will not reach $[\![E]\!]^I$. Proposition 3.48 gives an example of this: however we choose N, the N-truncated automaton fails to accept the word $a^n b^n c a^n b^n$ for $n > N$. This happens because $[\![E]\!]$ does not have uniform rank (and that it has rank 2 does not help).

3.5.2 Automaton Finiteness for Regular Expressions with Uniform Rank

Is the N-truncated Antimirov automaton finite? The states Γ of Q^E_N are all of length at most $N+1$, so there is hope. The automaton will be finite if we can find a finite set containing all the individual expressions E' appearing in the states Γ. We now define such a set $E^{\to *}$.

Definition 3.62. We define functions $(\_)^{\rightsquigarrow +}, \mathbf{R}, (\_)^{\to +}, (\_)^{\to *} : \mathrm{RE} \to \mathscr{P}(\mathrm{RE})$ by

$$\begin{array}{rcl}
a^{\rightsquigarrow +} & =_{\mathrm{df}} & \{1\} \\
0^{\rightsquigarrow +} & =_{\mathrm{df}} & \emptyset \\
(E+F)^{\rightsquigarrow +} & =_{\mathrm{df}} & E^{\rightsquigarrow +} \cup F^{\rightsquigarrow +} \\
1^{\rightsquigarrow +} & =_{\mathrm{df}} & \emptyset \\
(EF)^{\rightsquigarrow +} & =_{\mathrm{df}} & E^{\rightsquigarrow +} \cup F^{\rightsquigarrow +} \cup E^{\rightsquigarrow +} \cdot \{F\} \cup \{E\} \cdot F^{\rightsquigarrow +} \cup E^{\rightsquigarrow +} \cdot F^{\rightsquigarrow +} \\
(E^*)^{\rightsquigarrow +} & =_{\mathrm{df}} & E^{\rightsquigarrow +} \cup \{E^*\} \cdot E^{\rightsquigarrow +} \cup E^{\rightsquigarrow +} \cdot \{E^*\} \cup \\
 & & E^{\rightsquigarrow +} \cdot (\{E^*\} \cdot E^{\rightsquigarrow +}) \cup (E^{\rightsquigarrow +} \cdot \{E^*\}) \cdot E^{\rightsquigarrow +}
\end{array}$$

$$\begin{array}{rcl}
\mathbf{R}E & =_{\mathrm{df}} & \{\mathsf{R}^I_X E \mid X \subseteq \Sigma\} \\
E^{\to +} & =_{\mathrm{df}} & \mathbf{R}(E^{\rightsquigarrow +}) \\
E^{\to *} & =_{\mathrm{df}} & \{E\} \cup E^{\to +}
\end{array}$$

Proposition 3.63.

1. *For any E, the set $E^{\to *}$ is finite.*
2. *For any E and X, we have $(\mathsf{R}^I_X E)^{\to *} \subseteq \mathsf{R}^I_X(E^{\to *})$.*
3. *For any E, a and E_l, E_r, if $E \to^I (a; E_l, E_r)$, then $E_l \in \mathsf{R}^I_a(E^{\rightsquigarrow +})$ and $E_r \in E^{\rightsquigarrow +}$.*
4. *For any E, E', X, a, E'_l, E'_r, if $E' \in \mathsf{R}^I_X(E^{\rightsquigarrow +})$ and $E' \to^I (a; E'_l, E'_r)$, then $E'_l \in \mathsf{R}^I_{Xa}(E^{\rightsquigarrow +})$ and $E'_r \in \mathsf{R}^I_X(E^{\rightsquigarrow +})$.*
5. *For any E, u and $E_0, \ldots, E_n$, if $E \to^{I*} (u; E_0, \ldots, E_n)$, then $\forall j.\, E_j \in E^{\to *}$.*

Proposition 3.64. *For every E and N, the state set $\{\Gamma \mid \exists u \in \Sigma^*.\, E \to^{I*}_N (u;\Gamma)\}$ of the N-truncated refined Antimirov automaton for E is finite. If $[\![E]\!]$ has uniform rank at most N, then the N-truncated automaton accepts $[\![E]\!]^I$.*

3.6 Related Work

Syntactic derivative constructions for regular expressions extended with constructors for (versions of) the shuffle operation have been considered, for example, by Sulzmann and Thiemann [76] for the Brzozowski derivative and by Broda et al. [19] for the Antimirov

derivative. This is relevant to our derivatives since $L \cdot^I L'$ is by definition a language between $L \cdot L'$ and $L \sqcup\!\sqcup L'$. Thus our Brzozowski and Antimirov reordering derivatives of EF must be between the classical Brzozowski and Antimirov derivatives of EF and $E \sqcup\!\sqcup F$.

Finite asynchronous automata were introduced by Zielonka [80] as a way to characterise recognisable trace languages. It is a theorem that a trace language T is recognisable if and only if there is a finite asynchronous automaton such that T is the language accepted by that automaton. Since all recognisable trace languages have a star-connected expression defining them, we can also construct a finite automaton for every recognisable trace language given its star-connected expression. Asynchronous automata allow concurrent execution of independent actions but our construction yields a traditional automaton.

With the reordering derivatives we can construct a (possibly infinite) automaton for a regular expression E that accepts the closure $[\![E]\!]^I$. We accomplish this by essentially having more states and transitions in the automaton when compared to the automaton obtained by the usual derivative construction. Another possible approach is to change the way the automaton processes the input word. For example, Nagy and Otto [55] describe automata with *translucent letters*. The basic idea is that with each state q is associated a set of letters $\tau(q) \subseteq \Sigma$ that are considered translucent in q. Translucent letters are invisible to the machine. Thus in state q we can take a transition labelled a to q' when the current input word is waw' with $\Sigma(w) \subseteq \tau(q)$ and $a \notin \tau(q)$, i.e., a is the first letter that is not translucent. In our terminology we could say that a is the minimal letter of waw' in state q. It would be more natural for us to associate translucent letters with the alphabet (labels of the transitions) so that we can directly encode the independence I as translucent letters. Then the automaton can take a transition labelled a when a is a minimal letter of the input word according to I. Thus we can take an automaton for $[\![E]\!]$ and by modifying only the transitions to include translucent letters we would get an automaton for $[\![E]\!]^I$. As the translucent letters allow us to consider different factorisations of the input word, we can see the above construction as producing a $\Sigma^*/\!\sim^{I*}$-automaton with the additional restriction that the transitions are labelled only by the generators of the monoid.

A somewhat similar notion is that of *jumping finite automata* [54] where the machine can jump to an arbitrary position in the input word. The languages accepted by such automata are necessarily closed under permutation.

3.7 Conclusion and Future Work

We have shown that the Brzozowski and Antimirov derivative operations generalise to trace closures of regular languages in the form of reordering derivative operations. The sets of Brzozowski resp. Antimirov reordering (parts-of-)derivatives of an expression are generally infinite, so the deterministic and nondeterministic automata that they give, accepting the trace closure, are generally infinite. Still, if the expression is star-connected, then their appropriate quotients are finite. Also, the set of N-bounded refined Antimirov reordering parts-of-derivative is finite without quotienting, and we showed that, if the language of the expression has uniform rank at most N, the N-truncated refined Antimirov automaton accepts the trace closure. We also proved that star-connected expressions define languages with finite uniform rank.

In summary, with this work we have established the picture shown in Figure 1.

Our motivation for developing these reordering derivatives is to use the Antimirov reordering derivative as a guiding idea for describing operational semantics of relaxed memory models. Usually, when we consider sequential composition EF of programs E and F, then, to start executing the program F, we must have already successfully executed the program E. In the jargon of derivatives, this is to say that for an action from F to become

executable, what is left of E has to have become nullable (i.e., one can consider the execution of E completed). With reordering derivatives, we can execute an action from F successfully even when what is left of E is not yet nullable. It suffices that some sequence of actions to complete the residual of E is reorderable with the selected action of F. This is precisely the topic of exploration in Chapter 5.

In the definitions of the derivative operations, we only use I in one direction, i.e., we do not make use of its symmetry. It would be interesting to see which of the results from this chapter can be generalised to the setting of semicommutations [24]. In fact, in Chapter 5 where we develop an operational semantics based on the Antimirov reordering derivatives, we deliberately exclude the requirement that the independence relation must be symmetric. We do not, however, investigate the same questions as we did in this chapter for the symmetric case.

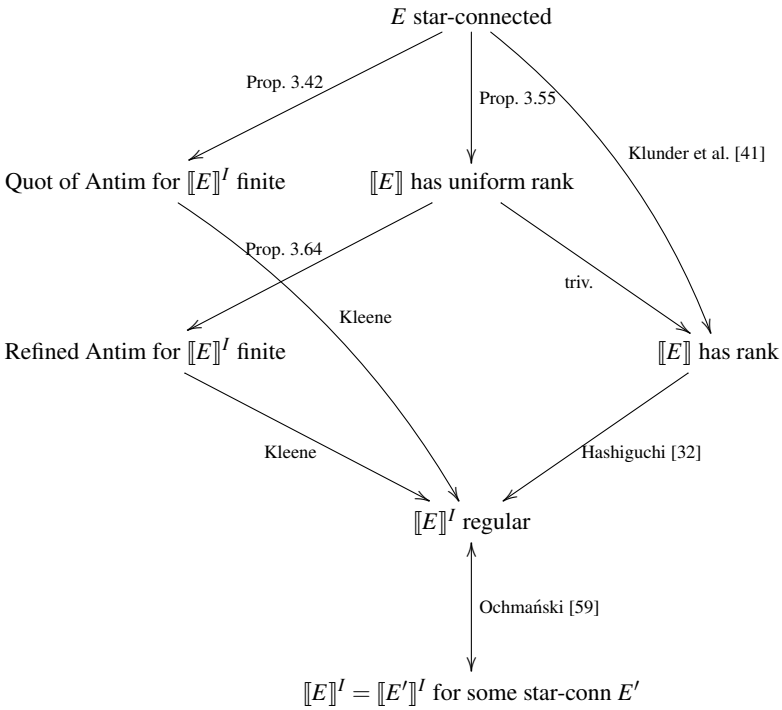


Figure 1 – Reordering derivatives in relation to earlier results

4 Normal Forms of Generalised Traces

In this chapter, we consider normal forms for a generalisation of traces where the commutability of a pair of adjacent letters in a word may depend on their left context (this is the prefix of the pair in the word). We develop both Foata and lexicographic normalisation for this generalisation. As an example application of this work, we describe an independence alphabet for a very simple TSO-like system where the context-dependence of the independence relation is used to express whether a particular read instruction reads its value from the shared memory or from the local write buffer.

4.1 Motivation

We have already introduced Mazurkiewicz traces and also the Foata and lexicographic normal forms in Section 2.3. Before continuing with the generalisation of traces in this chapter, we briefly motivate this development by describing our use of normal forms and also why this kind of generalisation of traces might be useful for us.

In formal languages, the alphabet Σ is usually treated simply as a collection of symbols. In applications, on the other hand, the letters of the alphabet typically have some meaning attached to them. In the example in Section 4.5 and in Chapter 5, we will take the alphabet Σ to be the set of instructions or actions that a machine can execute. Depending on the particular set of instructions, there may be pairs of instructions a and b that are independent, i.e., we do not see a semantic difference between executing a first and then b or the other way around. Words over this alphabet are then seen as program executions and the independence relation induces an equivalence relation on executions. Some programs may have several equivalent executions and a Mazurkiewicz trace corresponds to an equivalence class of executions according to this equivalence relation.

To check that the set of executions of a program satisfy some property, we can of course check each execution separately. When we know that this particular property is stable under an equivalence relation, then it is sufficient to check a single representative of each equivalence class. We are interested in the result of an execution and thus prefer equivalence relations where equivalent executions lead to the same result. If the equivalence relation induced by an independence relation is such, then we can use normal forms as the representative executions.

A useful property that normal forms may have and that the normal forms we consider do have is prefix-closedness, i.e., if uv is a normal form, then so is u. Contrapositively, this says that if u is not a normal form, then there is no v such that uv is a normal form. We will use this property to cut off search for a representative execution as soon as we discover that our current execution so far (the prefix) is not in normal form (this can also be seen in Definition 5.42 where it is only possible to construct executions that are in normal form).

Our development in this chapter is motivated by the fact that, in some cases, an independence relation might not be expressive enough. More precisely, in Mazurkiewicz traces, the independence relation is a binary relation on the alphabet and thus it is static. By this we mean that we cannot have a pair of letters that are independent at some instant, but not in general. The generalisation of traces introduced by Sassone et al. [74] that we consider in this chapter makes the independence relation depend on a word parameter representing the context in which the independence of the letters is considered. This context can be seen as a form of state. This allows us to say that a pair of letters (or instructions) commute in some context (machine state) but not in others. In the example in this chapter we use this feature to construct an independence alphabet where the meaning (commutability) of a read instruction depends on whether there is a pending

write instruction currently in the write buffer.

We develop generalisations of Foata and lexicographic normal forms for these generalised traces and also the corresponding normalisation algorithms and correctness proofs. A more detailed description of this development which documents the corresponding Agda [58] formalisation is in Appendix A. The Agda formalisation itself is available here: `http://cs.ioc.ee/~hendrik/code/phd/isse.zip`

4.2 Generalised Mazurkiewicz Traces

There are several generalisations of trace theory. We are considering the generalisation given by Sassone et al. [74]. In this setting, the essential difference is that independence is no longer a binary relation but instead it is an assignment of an irreflexive and symmetric (independence) relation to every word u. More precisely, we assume that we have an alphabet Σ and a context-dependent independence relation $I : \Sigma^* \to \mathscr{P}(\Sigma \times \Sigma)$. The word parameter to I is the context.

In this chapter, we seek to follow the following lexical convention where reasonable:

- a, b, c and d are letters (elements of Σ);
- s, t, u and v are words (elements of Σ^*);
- x, y and z are lists of words (elements of Σ^{**}).

The smallest difference between two equivalent words is that they differ by the ordering of a pair of adjacent independent letters.

Definition 4.1. For any $u \in \Sigma^*$, the words $s, s' \in \Sigma^*$ are *one-step Mazurkiewicz equivalent* in context u, denoted by $s \sim_u^I s'$, when there exist $a, b \in \Sigma$ and $t, t' \in \Sigma^*$ such that $a\, I_{us}\, b$, $s = tabt'$ and $s' = tbat'$.

Thus the words $tabt'$ and $tbat'$ are equivalent in the context u when the letters a and b are independent in the context $us \in \Sigma^*$. Mazurkiewicz equivalence is the reflexive-transitive closure of the above relation.

Definition 4.2. For any $u \in \Sigma^*$, two words $s, s' \in \Sigma^*$ are *Mazurkiewicz equivalent* in context u, denoted by $s \sim_u^{I*} s'$, when $s = s'$ or there exists $s'' \in \Sigma^*$ such that $s \sim_u^I s''$ and $s'' \sim_u^{I*} s'$.

A witness of $s \sim_u^{I*} s'$ can be thought of as a sequence of instructions for transforming s into s' by swapping pairs of adjacent independent letters. No letters from u can be involved in these swaps.

The original motivation for context-dependent independence relation in [74] was to make the independence relation more expressive or finer. However, without any restrictions, the context-dependent independence relation could become too fine. Consider an alphabet with $a\, I_\varepsilon\, b$, $c\, I_{ab}\, d$ and $c\, D_{ba}\, d$. Although we say that a and b are independent in the empty context, we could argue that they are not. If c and d are independent in the context ab, but dependent in the context ba, then we can see a difference between ab and ba and thus the ordering of a and b (in the empty context) matters.

Definition 4.3. A context-dependent independence relation $I : \Sigma^* \to \mathscr{P}(\Sigma \times \Sigma)$ is said to be *consistent* when it is stable under equivalence, i.e., for any $u, v \in \Sigma^*$ and $a, b \in \Sigma$, if $a\, I_u\, b$ and $u \sim_\varepsilon^{I*} v$, then $a\, I_v\, b$.

Definition 4.4. A context-dependent independence relation $I : \Sigma^* \to \mathscr{P}(\Sigma \times \Sigma)$ is said to be *coherent* when, for any $u \in \Sigma^*$ and $a, b, c \in \Sigma$, it satisfies the following conditions:

1. if $a\, I_u\, b$, $b\, I_{ua}\, c$ and $a\, I_{ub}\, c$, then $a\, I_u\, c$;
2. if $a\, I_u\, b$, $b\, I_u\, c$ and $a\, I_u\, c$, then $a\, I_{ub}\, c$;
3. if $a\, I_u\, b$, $b\, I_u\, c$ and $a\, I_{ub}\, c$, then $a\, I_u\, c$.

In generalised traces, the context-dependent independence relation is required to be both consistent and coherent. The consistency and coherence conditions ensure that a context-dependent independence relation is sufficiently coarse. This turns out to be necessary for our generalisations of the Foata and lexicographic normal forms and the corresponding normalisation algorithms.

Consistency is a very basic hygiene condition. It just states that equivalent words must give the same independence relation. This rules out the undesirable behaviour described above. The coherence conditions are more interesting since they involve different contexts. Looking at their shape, we could say that the essence of the coherence conditions is that, if we have a letter b so that both a and b as well as b and c are independent (in some context), then b is "independent enough" (wrt. a and c), so that the independence or dependence of a and c is not affected by adding b to or removing it from the end of the context. The important details in the rules are the contexts. Note that there is no need for a version of the first coherence condition for extending the context as that is derivable from the second and third condition.

One way to see the coherence conditions is to say that they are the smallest set of conditions guaranteeing that any choice of three conditions, one from each of the following three pairs, implies the other three: ($a\, I_u\, b$, $a\, I_{uc}\, b$), ($b\, I_u\, c$, $b\, I_{ua}\, c$), ($a\, I_u\, c$, $a\, I_{ub}\, c$). This is with the exception of the choice of the second condition from each pair; from these three conditions one cannot conclude anything. For example, $a\, I_u\, b$, $b\, I_{ua}\, c$ and $a\, I_{ub}\, c$ imply not only $a\, I_u\, c$ and $b\, I_u\, c$ (both by the first condition), but also $a\, I_{uc}\, b$ (follows from those by the second condition).

We illustrate generalised traces with a modification of the example independence alphabet we considered in Section 2.3. We now take I to be the least consistent and coherent family of symmetric relations such that $a\, I_\varepsilon\, b$, $a\, I_\varepsilon\, d$, $b\, I_a\, d$, $b\, I_{ac}\, d$ and $c\, I_{ab}\, d$. Explicitly, this means that we also have $b\, I_\varepsilon\, d$ (by third coherence condition), $a\, I_d\, b$, $a\, I_b\, d$ (by second coherence condition) and $c\, I_{ba}\, d$ (by consistency). Now $abcd$ has the same equivalence class as before, but $acbd$ is only equivalent to $acdb$, leaving $adcb$ and $dacb$ in a different equivalence class.

We now give some explanation why requiring the coherence conditions of Definition 4.4 is desirable. Our argument is that without coherence we can have conflicting definitions of minimal (and maximal) letters in a word. We consider a to be minimal in xay when $xay \sim_\varepsilon^{I^*} av$ for some v. Before, we also said that a letter a is minimal in a word when it is possible to commute it past its prefix (x in this case) to the beginning of the word. We describe normal forms as predicates on words and thus we would like to decide whether a word is a normal form by just considering this word and not by first having to enumerate its entire equivalence class.

We can violate condition (1) by taking $a\, I_\varepsilon\, b$, $a\, D_c\, b$, $b\, D_\varepsilon\, c$, $b\, I_a\, c$, $a\, D_\varepsilon\, c$ and $a\, I_b\, c$. (Note that condition (1) would require both $b\, I_\varepsilon\, c$ and $a\, I_\varepsilon\, c$. Conditions (2) and (3) are satisfied.) We have $acb \sim_\varepsilon^I abc \sim_\varepsilon^I bac \sim_\varepsilon^I bca$. We can see that a is a maximal letter as the equivalence class contains bca. By considering the equivalent word acb, we do not see this as $a\, D_\varepsilon\, c$ and thus a cannot be commuted past its suffix to the end of the word.

We can violate condition (2) by taking $a\, I_\varepsilon\, b$, $a\, I_c\, b$, $b\, I_\varepsilon\, c$, $b\, I_a\, c$, $a\, I_\varepsilon\, c$ and $a\, D_b\, c$. (Note that condition (2) would require $a\, I_b\, c$. Conditions (1) and (3) are satisfied.) This means we have $bac \sim_\varepsilon^I abc \sim_\varepsilon^I acb \sim_\varepsilon^I cab \sim_\varepsilon^I cba \sim_\varepsilon^I bca$. We consider c to be minimal because of

the word cab, but we cannot see this from the word bac itself as $a\,D_b\,c$. Another anomaly here is that although $a\,D_b\,c$, we still have $bac \sim_\varepsilon^{I^*} bca$.

We can violate condition (3) by taking $a\,I_\varepsilon\,b$, $a\,D_c\,b$, $b\,I_\varepsilon\,c$, $b\,D_a\,c$, $a\,D_\varepsilon\,c$ and $a\,I_b\,c$. (Note that condition (3) would require $a\,I_\varepsilon\,c$. Conditions (1) and (2) are satisfied.) This means we have $abc \sim_\varepsilon^I bac \sim_\varepsilon^I bca \sim_\varepsilon^I cba$. We consider a to be minimal because of the word abc, but we cannot see this from the word cba itself as $a\,D_c\,b$.

4.3 Generalised Foata Normalisation

In this section, we describe Foata normal forms for generalised traces and the corresponding normalisation algorithm. We conclude with the correctness proof of the algorithm.

4.3.1 Normal Forms

The Foata normal form of a generalised trace is still a well-formed sequence of well-formed steps as in the standard case (Section 2.3.1), but we change the well-formedness condition of steps and sequences to take into account the context-dependence of the independence relation.

First, we extend the independence relation to words (considered as steps) and letters by $\varepsilon \blacksquare I_u\, a =_{\mathrm{df}} \mathrm{tt}$ and $sb \blacksquare I_u\, a =_{\mathrm{df}} s \blacksquare I_u\, a \wedge b\, I_u\, a$. Thus $s \blacksquare I_u\, a$ just says that all letters in s are independent of a in the context u. Note that the context u stays fixed in the definition, i.e., the letters in s have to be independent of a in the same context u. We also use $s \blacklozenge D_u\, a$ as the negation of this to say that there exists a letter in s such that $s\,D_u\,a$. The relation $\prec$ is the strict total order on the alphabet. The function $\lfloor \_ \rfloor : \Sigma^{**} \to \Sigma^*$ flattens the given list of words by concatenation.

Definition 4.5. For any $u \in \Sigma^*$, the set $\mathit{Step}(\Sigma, I, u) \subseteq \Sigma^*$ of well-formed steps in context u is given by:

1. for any $a \in \Sigma$, $a \in \mathit{Step}(\Sigma, I, u)$;
2. for any $s \in \Sigma^*$ and $a, b \in \Sigma$,
   if $sa \in \mathit{Step}(\Sigma, I, u)$, $a \prec b$ and $sa \blacksquare I_u\, b$, then $sab \in \mathit{Step}(\Sigma, I, u)$.

Thus a well-formed step (in context u) is either a singleton letter or it consists of a well-formed step to which a new letter is added on the right, which has to be greater than the previous rightmost letter. The added letter and the step must be independent. This means that if we have $sas'bs'' \in \mathit{Step}(\Sigma, I, u)$, then it must be that $a\,I_u\,b$ and $a \prec b$. We require the letters of a step to be independent in the fixed context since we think of a step as a set of independent letters and thus should be allowed to move the letters freely within a step. We require the letters to be ordered to have a concrete representation (an enumeration) of this set as a word.

We now define when a sequence of steps support a letter. Intuitively, if a sequence of steps $x \in \Sigma^{**}$ supports the letter a, then either the last step in x contains a dependent letter or x is empty.

Definition 4.6. For any $a \in \Sigma$, the set $\mathit{Sup}(\Sigma, I, a) \subseteq \Sigma^{**}$ of sequences of steps that *support* the letter a is given by:

1. $\varepsilon \in \mathit{Sup}(\Sigma, I, a)$;
2. for any $x \in \Sigma^{**}$ and $s \in \Sigma^*$, if $s \blacklozenge D_{\lfloor x \rfloor}\, a$, then $xs \in \mathit{Sup}(\Sigma, I, a)$.

We now define normal forms as sequences of steps where every letter in a step is supported by the preceding steps.

Definition 4.7. The set $\mathit{Foata}(\Sigma, I) \subseteq \Sigma^{**}$ of Foata normal forms is given by:

1. $\varepsilon \in \mathit{Foata}(\Sigma, I)$;
2. for any $x \in \mathit{Foata}(\Sigma, I)$ and $s \in \mathit{Step}(\Sigma, I, \lfloor x \rfloor)$,
   if, for every $a \in s$, we have $x \in \mathit{Sup}(\Sigma, I, a)$, then $xs \in \mathit{Foata}(\Sigma, I)$.

Thus a Foata normal form is a sequence of steps where every step is well-formed in the context of its prefix and every letter in a step has a dependent letter in the previous step (unless it is the first step). When a letter in a step has a dependent letter in the previous step, then the letter does not "fit" into the previous step and thus it is at its earliest possible position.

The function emb that embeds a normal form (an element of $\mathit{Foata}(\Sigma, I)$) back to words is defined as $\mathit{emb}\, x = \lfloor x \rfloor$, i.e., it just concatenates the steps of the normal form.

Continuing with the example independence alphabet from the last subsection, we have that $(abd)(c)$ is a Foata normal form since we have $a\, I_\varepsilon\, b$, $a\, I_\varepsilon\, d$ and $b\, I_\varepsilon\, d$ making (abd) a valid step and $a\, D_\varepsilon\, c$ ensuring that the sole letter in the step (c) is supported. We also have that $(a)(c)(bd)$ is a normal form since $b\, I_{ac}\, d$ ensures that the step (bd) is well-formed and $a\, D_\varepsilon\, c$, $c\, D_a\, b$ and $c\, D_a\, d$ provide the requisite support for the letters in (c) and (bd).

4.3.2 Normalisation

The main ingredient in the normalisation algorithm is a function that takes a normal form and a letter and inserts the letter into its right place in the normal form. Intuitively, the correct step for this letter is the leftmost step in the normal form that can be reached with this letter, starting from the rightmost step. This is because our normalisation algorithm traverses the given input word (from left to right) and inserts each letter into an accumulating normal form starting from the empty normal form.

We use Agda-like notation for describing the algorithms. Words are represented as cons- or snoc-lists (`List` or `List>`) over an alphabet Σ. We use `<:` and `<+` for the cons operation and concatenation of cons-lists (the corresponding snoc-list operations are `:>` and `+>`). A more thorough description of the Agda formalisation is in Appendix A.

First, we define a function `find>` parameterised by a decider `P?` of a predicate `P` on a context (a snoc-list) and an element. It splits a given snoc-list `xs` into two parts, `ls` and `rs`, so that all of the elements in `rs` satisfy the predicate and the rightmost element in `ls` violates the predicate.

```
find> : (∀ xs x → Dec (P xs x)) → List> X → List> X × List> X
find> P? [] = [] , []
find> P? (xs :> x) with P? xs x
find> P? (xs :> x) | yes _ =
  let ls , rs = find> P? xs in ls , rs :> x
find> P? (xs :> x) | no  _ = xs :> x , []
```

Given a step and a letter, we can use `find>` to find the right position of the letter in the step.

```
insertStep : Step → Σ → Step
insertStep s a =
  let ls , rs = find> (\ _ b → a ≺? b) s in
  ls :> a +> rs
```

Here we use `find>` with a predicate that ignores the context. The step `s` is split into `ls` and `rs` so that everything in `rs` is greater than `a` and the rightmost letter in `ls` is not. We

assume that the ordering relation $\prec$ is decidable, with `≺?` as the decider. Hence `a ≺? b` is either `yes` (together with a proof of `a ≺ b`) or `no` (together with a proof of `¬(a ≺ b)`).

Given a normal form and a letter, we use `find>` to find the correct step for the letter.

```
insert : Foata → Σ → Foata
insert x a with find> (\ y s → ■I? y s a) x
insert x a | ls , [] = ls :> ([] :> a)
insert x a | ls , rs :> r =
  let s , rs' = first rs r in
  ls :> insertStep s a +> rs'
```

Here `find>` splits the normal form into two parts, `ls` and `rs`, so that all the steps in `rs` are independent of `a` (note the context used) and the rightmost step in `ls` is dependent (or `ls` is empty). If `rs` is empty, then we add a new step to the normal form as `ls` supports the new letter. Otherwise, we insert `a` into the leftmost step in `rs` (the function `first` extracts the leftmost element in a non-empty snoc-list). We assume that the independence relation I is decidable. Here `■I? y s a` decides whether it is the case that $s \blacksquare I_{\lfloor y \rfloor}\, a$, i.e., whether the letters in s are independent of a in the context $\lfloor y \rfloor$.

The normalisation function just traverses the input word from the left to the right and inserts each letter into the correct position in the accumulated normal form.

```
norm' : Foata → List Σ → Foata
norm' x []       = x
norm' x (a <: t) = norm' (insert x a) t

norm : List Σ → Foata
norm t = norm' [] t
```

We continue with our example and look at the evolution of the accumulator as the word $bacd$ is normalised. First, the letter b is inserted into the empty normal form, resulting in the normal form (b). Next, the letter a is inserted into this normal form, which results in (ab) because of $a\, I_\varepsilon\, b$. Next, the letter c is inserted into the result. We have $a\, D_\varepsilon\, c$, which means that a new step must be added and the result is $(ab)(c)$. We now need to insert d into the normal form. We have $c\, I_{ab}\, d$ and in addition we also have $a\, I_\varepsilon\, d$ and $b\, I_\varepsilon\, d$. This makes the first step the earliest possible step for d and the result is $(abd)(c)$.

4.3.3 Correctness

We have defined the Foata normalisation function, but we have no assurance yet that it produces well-formed Foata normal forms (i.e., elements of $Foata(\Sigma, I)$). We will now proceed to show that the function *norm* constructs a well-formed normal form from the input word.

We start by showing that inserting a letter (provided that it is independent of the step) into a well-formed step produces a well-formed step.

Lemma 4.8. *For any $u, s \in \Sigma^*$ and $a \in \Sigma$,*
if $s \in Step(\Sigma, I, u)$ and $s \blacksquare I_u\, a$, then $insertStep\ s\ a \in Step(\Sigma, I, u)$.

To outline what we need to do next, let us look at a small example. Suppose we have a normal form $stuv$ consisting of steps s, t, u, and v, and we wish to insert the letter a into this normal form. It so happens that a will go into the step t. This means that, instead of the old context st, the letters in u must now be independent in the new context $s(insertStep\ t\ a)$. Likewise, the letters in v must now be independent in the context

$s(\mathit{insertStep}\ t\ a)u$. Furthermore, every letter in v must now be supported by a letter in u in the context $s(\mathit{insertStep}\ t\ a)$.

To show that the independence of letters in a step is preserved during an insert that inserts a letter into the context, we have the following lemma.

Lemma 4.9. *For any $u, s \in \Sigma^*$ and $a \in \Sigma$,*
if $s \in \mathit{Step}(\Sigma, I, u)$ and $s \blacksquare I_u\, a$, then $s \in \mathit{Step}(\Sigma, I, ua)$.

Next, we are considering the situation where we are inserting the letter a into the normal form xst and we have determined that a must go into some step in x. We wish to show that the letters in t are still supported (by something in s) after the insert. We use $PW(I_u, s)$ to express that the predicate I_u holds between any pair of letters in s, i.e., the letters in s are pairwise independent in the context u. The normal form x is considered in the lemma as the context u and b is a letter from the step t that requires the support.

Lemma 4.10. *For any $u, s \in \Sigma^*$ and $a, b, \in \Sigma$,*
if $b\, I_{us}\, a$, $PW(I_u, s)$, $s \blacksquare I_u\, a$ and $s \blacklozenge D_u\, b$, then $s \blacklozenge D_{ua}\, b$.

This shows that, under suitable conditions, we can add a letter (in this case a) to the end of the context and still have a dependent letter in the previous step (for step s and letter b in this case). By consistency of I, we know that this support is then preserved for any equivalent context. This allows us to show that the function *insert* preserves the equivalence class in the following sense.

Lemma 4.11. *For any $x \in \Sigma^{**}$ and $a \in \Sigma$,*
if $x \in \mathit{Foata}(\Sigma, I)$, then $\mathit{emb}\ (\mathit{insert}\ x\ a) \sim_\varepsilon^{I^*} (\mathit{emb}\ x)a$.

The next lemma is similar to Lemma 4.10 which said that we can add a letter to the context and preserve the dependent pairs of letters between steps. This says a similar thing for *insert*.

Lemma 4.12. *For any $x \in \Sigma^{**}$, $s \in \Sigma^*$ and $a, b \in \Sigma$,*
if $xs \in \mathit{Foata}(\Sigma, I)$, $a\, I_{\lfloor xs \rfloor}\, b$, $xs \notin \mathit{Sup}(\Sigma, I, b)$, $x \notin \mathit{Sup}(\Sigma, I, b)$ and $xs \in \mathit{Sup}(\Sigma, I, a)$, then $\mathit{insert}\ xs\ b \in \mathit{Sup}(\Sigma, I, a)$.

This allows us to show that inserting a letter into a normal form again produces a normal form.

Lemma 4.13. *For any $x \in \Sigma^{**}$ and $a \in \Sigma$, if $x \in \mathit{Foata}(\Sigma, I)$, then $\mathit{insert}\ x\ a \in \mathit{Foata}(\Sigma, I)$.*

Since the normalisation function just traverses the input word and inserts each letter to the accumulating normal form, we have that the normalisation function produces elements of $\mathit{Foata}(\Sigma, I)$.

Proposition 4.14. *For any $s \in \Sigma^*$, $\mathit{norm}\ s \in \mathit{Foata}(\Sigma, I)$.*

By now we have shown that $\mathit{norm}\ s$ produces an element of $\mathit{Foata}(\Sigma, I)$. Next, we show that these indeed are the normal forms. In other words, there is exactly one normal form for an equivalence class.

The correctness proof of the normalisation algorithm consists of the proofs of the soundness and completeness properties. By soundness we mean that equivalent words must get assigned the same normal form. By completeness we mean that any two words that get assigned the same normal form must be equivalent. With these properties we have a bijection between the set of equivalence classes and the image of the normalisation function *norm*. We will also show that *norm* is surjective.

The key lemma for completeness is that the result of normalising a word (and then embedding it) is equivalent to that word, i.e., for every word there exists a normal form.

Proposition 4.15. *For any* $s \in \Sigma^*$, $emb\ (norm\ s) \sim^{I*}_{\varepsilon} s$.

This leads to completeness as $norm\ s = norm\ s'$ of course implies that the embeddings of the normal forms are equivalent, i.e., $emb\ (norm\ s) \sim^{I*}_{\varepsilon} emb\ (norm\ s')$.

Corollary 4.16. *For any* $s, s' \in \Sigma^*$, *if* $norm\ s = norm\ s'$, *then* $s \sim^{I*}_{\varepsilon} s'$.

To prove soundness of the normalisation algorithm, we first show the commutativity of the normalisation algorithm for independent letters. We start by showing that the order in which we insert two independent letters into the same step does not matter.

Lemma 4.17. *For any* $u, s \in \Sigma^*$ *and* $a, b \in \Sigma$, *if* $s \in Step(\Sigma, I, u)$, $a\ I_{us}\ b$, $s \blacksquare I_u\ a$ *and* $s \blacksquare I_u\ b$, *then* $insertStep\ (insertStep\ s\ a)\ b = insertStep\ (insertStep\ s\ b)\ a$.

Next, we show that the order in which we insert two independent letters to a normal form also does not matter.

Lemma 4.18. *For any* $x \in \Sigma^{**}$ *and* $a, b \in \Sigma$,
if $x \in Foata(\Sigma, I)$ *and* $a\ I_{\lfloor x \rfloor}\ b$, *then* $insert\ (insert\ x\ a)\ b = insert\ (insert\ x\ b)\ a$.

This implies that it does not matter whether we normalise the word ab or ba when the letters are independent.

Lemma 4.19. *For any* $x \in \Sigma^{**}$ *and* $a, b \in \Sigma$,
if $x \in Foata(\Sigma, I)$ *and* $a\ I_{\lfloor x \rfloor}\ b$, *then* $norm'\ x\ ab = norm'\ x\ ba$.

Now we can show that normalising two words that differ only by the ordering of a single pair of adjacent letters gives the same result.

Lemma 4.20. *For any* $x \in \Sigma^{**}$ *and* $s, s' \in \Sigma^*$,
if $x \in Foata(\Sigma, I)$ *and* $s \sim^{I}_{\lfloor x \rfloor} s'$, *then* $norm'\ x\ s = norm'\ x\ s'$.

This can then be extended to equivalent words.

Lemma 4.21. *For any* $x \in \Sigma^{**}$ *and* $s, s' \in \Sigma^*$,
if $x \in Foata(\Sigma, I)$ *and* $s \sim^{I*}_{\lfloor x \rfloor} s'$, *then* $norm'\ x\ s = norm'\ x\ s'$.

This finally implies the desired soundness property.

Proposition 4.22. *For any* $s, s' \in \Sigma^*$, *if* $s \sim^{I*}_{\varepsilon} s'$, *then* $norm\ s = norm\ s'$.

The soundness and completeness proofs give us a certified decision procedure for checking whether two words are equivalent: first normalise the two words and then check whether the normal forms are the same.

```
equivalent? : (s s' : List Σ) → Dec (s ∼[]^I* s')
equivalent? s s' with foata-eq? (norm s) (norm s')
equivalent? s s' | yes feq = yes (completeness feq)
equivalent? s s' | no ¬feq = no (\ eqv → ¬feq (soundness eqv))
```

This procedure will either return `yes`, together with instructions how to turn `s` into `s'` (which letters need to be exchanged), or `no`, together with a proof that it is not possible to turn `s` into `s'`. Here `foata-eq?` uses the decidable equality on the alphabet to decide whether the two normal forms are the same.

Finally, we have that the normalisation function is stable in the sense that normalising a normal form produces the same normal form. Hence the normalisation function is surjective, i.e., every normal form is the normal form of something.

Proposition 4.23. *For any $x \in \Sigma^{**}$, if $x \in Foata(\Sigma, I)$, then $norm\ (emb\ x) = x$.*

Soundness says that words in an equivalence class get assigned the same normal form. With the above proposition, we also have that there is at most one normal form for an equivalence class (wrt. to emb), i.e., normal forms are unique.

Corollary 4.24. *For any $x, y \in \Sigma^{**}$, if $x, y \in Foata(\Sigma, I)$ and $emb\ x \sim_{\varepsilon}^{I*} emb\ y$, then $x = y$.*

4.4 Generalised Lexicographic Normalisation

In this section, we describe the lexicographic normal form for generalised traces and the corresponding normalisation algorithm. We conclude with the correctness proof of the normalisation algorithm.

4.4.1 Normal Forms

In contrast to Foata normal forms that were sequences of steps, lexicographic normal forms are just certain elements of Σ^*. Hence the embedding function emb turning normal forms to words is here the identity function and we omit it.

We consider a list of letters to be a well-formed lexicographic normal form when each letter in it is in a correct position. Whether a letter is in a correct position is determined by its prefix, i.e., in a word uav the prefix u determines whether it is correct to follow it with an a. If it is, then we say that the word u supports a. A word is in normal form if every letter in it is supported.

Definition 4.25. For any $a \in \Sigma$, the set $Sup(\Sigma, I, a) \subseteq \Sigma^*$ of words that support the letter a is given by:

1. $\varepsilon \in Sup(\Sigma, I, a)$;
2. for any $u \in \Sigma^*$ and $b \in \Sigma$, if $b\, D_u\, a$, then $ub \in Sup(\Sigma, I, a)$;
3. for any $u \in \Sigma^*$ and $b \in \Sigma$, if $u \in Sup(\Sigma, I, a)$, $b\, I_u\, a$ and $b \prec a$, then $ub \in Sup(\Sigma, I, a)$.

Hence a letter is supported by a word if either it is the empty word ε or the word ends with a dependent letter or the word ends with an independent letter that is before it in the ordering and the prefix supports the letter. A sequence of letters is a lexicographic normal form when every letter in the sequence is supported by its prefix.

Definition 4.26. The set $Lex(\Sigma, I) \subseteq \Sigma^*$ of lexicographic normal forms is given by

1. $\varepsilon \in Lex(\Sigma, I)$;
2. for any $u \in \Sigma^*$ and $a \in \Sigma$, if $u \in Lex(\Sigma, I)$ and $u \in Sup(\Sigma, I, a)$, then $ua \in Lex(\Sigma, I)$.

We continue with our example independence alphabet and show that $abcd$ is a lexicographic normal form. Since $\varepsilon \in Lex(\Sigma, I)$ and $\varepsilon \in Sup(\Sigma, I, a)$, we get $a \in Lex(\Sigma, I)$. Next, we have $a \in Sup(\Sigma, I, b)$ since $\varepsilon \in Sup(\Sigma, I, b)$, $a\, I_\varepsilon\, b$ and $a \prec b$. Thus we also have $ab \in Lex(\Sigma, I)$. We have $ab \in Sup(\Sigma, I, c)$ since $b\, D_a\, c$ resulting in $abc \in Lex(\Sigma, I)$. Finally, we get $abc \in Sup(\Sigma, I, d)$ from $\varepsilon \in Sup(\Sigma, I, d)$ by applying condition 4.25.(3) three times. Thus we have $abcd \in Lex(\Sigma, I)$.

Before we wrote $s \blacksquare I_u\, a$ to say that every letter in the word s (that represents a step) is independent of a in the context u. Here, we will need a variation where, for a letter b in s, we also consider its prefix in s as part of the context. We extend the independence relation to words and letters by $\varepsilon\, I_u\, a =_{\mathrm{df}} \mathrm{tt}$ and $sb\, I_u\, a =_{\mathrm{df}} s\, I_u\, a \wedge b\, I_{us}\, a$. Note the modified

context in $b\, I_{us}\, a$. When $s\, I_u\, a$, then we say that s is a "chain" of independent letters wrt. a. In this case we can "slide" the letter a through s without changing the equivalence class, i.e., $sa \sim_u^{I*} as$.

Anisimov and Knuth's characterisation of lexicographic normal forms forbids the "bua" pattern. Our definition forbids this pattern for the generalised case that we consider here.

Proposition 4.27. *For any $t,u,v \in \Sigma^*$ and $a,b \in \Sigma$, if $tbuav \in Lex(\Sigma,I)$, $a\, I_t\, b$ and $a \prec b$, then $u\, D_{tb}\, a$.*

The strict total order $\prec$ on Σ induces the lexicographic order relation $\preceq_{Lex}$ on Σ^*. By definition, the lexicographic normal form is the least element in its equivalence class wrt. the lexicographic order $\preceq_{Lex}$. The normal forms we have defined for the generalised case are also least elements in their equivalence classes.

Proposition 4.28. *For any $s,t \in \Sigma^*$, if $s \in Lex(\Sigma,I)$ and $s \sim_\varepsilon^{I*} t$, then $s \preceq_{Lex} t$.*

4.4.2 Normalisation

The main ingredient in the normalisation algorithm is a function that inserts a letter into its correct position in a list (which is assumed to be a well-formed normal form). Given a word s and a letter a, the idea is to split s into three parts: s_D, $s_\prec$, and s_I so that s_D ends with a letter dependent on a (or it is empty), all letters in $s_\prec$ are independent of and less than a, and letters in s_I are independent of a and the first letter of s_I is greater than a (or it is empty). The idea is that the word $s_D s_\prec$ is the longest prefix of s that supports the letter a.

```
findPos : List> Σ → Σ → List> Σ × List> Σ × List> Σ
findPos []       a = [] , [] , []
findPos (s :> b) a with I? s b a
findPos (s :> b) a | no  _ = s :> b , [] , []
findPos (s :> b) a | yes _ with findPos s a
findPos (s :> b) a | yes _ | sd , sp , si :> i =
  sd , sp , si :> i :> b
findPos (s :> b) a | yes _ | sd , sp , [] with b ≺? a
findPos (s :> b) a | yes _ | sd , sp , [] | no  _ =
  sd , sp , [] :> b
findPos (s :> b) a | yes _ | sd , sp , [] | yes _ =
  sd , sp :> b , []
```

The function *findPos* implements the described functionality. Like before, we assume that the independence relation I and the order relation $\prec$ are decidable, with deciders $I?$ and $\prec?$. The *insert* function now just plugs the letter between $s_\prec$ and s_I in the result of *findPos*.

```
insert : List> Σ → Σ → List> Σ
insert s a =
  let sd , sp , si = findPos s a in
  sd +> sp :> a +> si
```

The normalisation algorithm just traverses the input word letter by letter and inserts the letters into the accumulating normal form, just as in Foata normalisation.

```
norm' : List> Σ → List Σ → List> Σ
norm' s []       = s
norm' s (a <: t) = norm' (insert s a) t
```

```
norm : String → List> Σ
norm t = norm' [] t
```

We continue with our example and look at what are the intermediate steps when normalising $bacd$. First, when inserting b into the empty normal form, *insert* splits it into $\varepsilon, \varepsilon, \varepsilon$ and the result is b. Next, when inserting a, the normal form b is split into $\varepsilon, \varepsilon, b$ since $a \prec b$ and $b\, I_\varepsilon\, a$. The result is ab. When inserting c into ab, the split is $ab, \varepsilon, \varepsilon$ and the result is abc. Finally, when inserting d into abc, the split is $\varepsilon, abc, \varepsilon$ and the result is $abcd$.

4.4.3 Correctness

We have now defined the lexicographic normalisation algorithm. This produces "raw" normal forms, i.e., just words. We will now show that these words are well-formed normal forms. We begin with a couple of lemmas exhibiting that *findPos* behaves as expected. The first lemma says that *findPos* just splits the input word.

Lemma 4.29. *For any $s \in \Sigma^*$ and $a \in \Sigma$, if $findPos\ s\ a = s_D, s_\prec, s_I$, then $s_D s_\prec s_I = s$.*

The next lemma ensures that the s_I component in the result of *findPos* consists of a "chain" of independent letters wrt. a.

Lemma 4.30. *For any $s \in \Sigma^*$ and $a \in \Sigma$, if $findPos\ s\ a = s_D, s_\prec, s_I$, then $s_I\, I_{s_D s_\prec}\, a$.*

The next lemma ensures that the leftmost letter of s_I in the result of *findPos* is greater than the letter a. The proposition $a \prec_{first} s_I$ holds when a is less than the first letter of s_I.

Lemma 4.31. *For any $s \in \Sigma^*$, $a \in \Sigma$, if $findPos\ s\ a = \_, \_, s_I$, then $a \prec_{first} s_I$.*

The next property is that *insert* preserves the equivalence class.

Lemma 4.32. *For any $s \in \Sigma^*$ and $a \in \Sigma$, $insert\ s\ a \sim_\varepsilon^{I*} sa$.*

The next lemma ensures that, under certain conditions, the support of a letter is preserved when another letter is inserted into the supporting word.

Lemma 4.33. *For any $s, t \in \Sigma^*$ and $a, b, c \in \Sigma$, if $stc \in Sup(\Sigma, I, a)$, $tc\, I_s\, b$, $b\, I_{stc}\, a$, $b \prec_{first} tc$, then $sbtc \in Sup(\Sigma, I, a)$.*

This allows us to say that inserting a letter into a normal form results in a normal form.

Lemma 4.34. *For any $s \in \Sigma^*$ and $a \in \Sigma$, if $s \in Lex(\Sigma, I)$, then $insert\ s\ a \in Lex(\Sigma, I)$.*

Thus we get that the normalisation function produces elements of $Lex(\Sigma, I)$.

Proposition 4.35. *For any $t \in \Sigma^*$, $norm\ t \in Lex(\Sigma, I)$.*

By now, we have shown that *norm* produces an element of $Lex(\Sigma, I)$. Next, we show that these indeed are the normal forms. In other words, there is exactly one normal form for an equivalence class.

As was the case for Foata normalisation, the correctness proof of the normalisation algorithm consists of the proofs of the soundness and completeness properties. By soundness we mean that equivalent words must get assigned the same normal form. By completeness we mean that any two words that get assigned the same normal form must be equivalent. We do things in a slightly different order than what we did for Foata normal forms: we show soundness as a corollary of uniqueness of normal forms.

The key property for the completeness proof is that the result of normalising a word is equivalent to that word, i.e., every word has a normal form.

Proposition 4.36. *For any $t \in \Sigma^*$, $norm\ t \sim_\varepsilon^{I*} t$.*

This leads to completeness as $norm\ t = norm\ t'$ of course implies $norm\ t \sim_\varepsilon^{I*} norm\ t'$.

Corollary 4.37. *For any $t, t' \in \Sigma^*$, if $norm\ t = norm\ t'$, then $t \sim_\varepsilon^{I*} t'$.*

Continuing towards soundness, we first prove the uniqueness of normal forms, i.e., if two normal forms are equivalent, then they must be the same.

Proposition 4.38. *For any $s, s' \in \Sigma^*$, if $s, s' \in Lex(\Sigma, I)$ and $s \sim_\varepsilon^{I*} s'$, then $s = s'$.*

Proof. From the assumptions, by Proposition 4.28, we have both $s \preceq_{Lex} s'$ and $s' \preceq_{Lex} s$, from which, by antisymmetry of $\preceq_{Lex}$, we get $s = s'$. □

Then we have soundness as a corollary.

Corollary 4.39. *For any $t, t' \in \Sigma^*$, if $t \sim_\varepsilon^{I*} t'$, then $norm\ t = norm\ t'$.*

Proof. Applying Proposition 4.36 to both t and t', we get $norm\ t \sim_\varepsilon^{I*} norm\ t'$. The result follows from this by Proposition 4.35 and Proposition 4.38. □

As normalising a word produces a normal form that is equivalent to the original, then by uniqueness of normal forms we also get the stability of the normalisation algorithm.

Corollary 4.40. *For any $s \in \Sigma^*$, if $s \in Lex(\Sigma, I)$, then $norm\ s = s$.*

An alternative approach to soundness would have been to prove the following lemma.

Lemma 4.41. *For any $s \in \Sigma^*$ and $a, b \in \Sigma$,*
if $s \in Lex(\Sigma, I)$ and $a\, I_s\, b$, then $insert\ (insert\ s\ a)\ b = insert\ (insert\ s\ b)\ a$.

This leads to soundness, stability and uniqueness as in the previous section. Finally, we can now prove the converses of Proposition 4.27 and Proposition 4.28 showing that a word with no forbidden patterns is a lexicographic normal form and that the least word in an equivalence class is the lexicographic normal form.

Proposition 4.42. *For any $s \in \Sigma^*$, if, for every decomposition of s as $s = tbuav$ we have $\neg(a\, I_t\, b)$ or $\neg(a \prec b)$ or $\neg(u\, I_{tb}\, a)$, then $s \in Lex(\Sigma, I)$.*

Proposition 4.43. *For any $s \in \Sigma^*$, if, for any $t \in \Sigma^*$, we have $s \sim_\varepsilon^{I*} t$ implies $s \preceq_{Lex} t$, then $s \in Lex(\Sigma, I)$.*

4.5 Example: TSO-like Independence Alphabet

Here we will give a small example where generalised traces are needed to describe the behaviour of a concurrent system reasonably precisely. The example is from shared-memory concurrency with write buffers. It is meant to resemble the Total Store Order (TSO) relaxed memory model of the SPARC family [75].

The machine that we are going to model consists of processors and shared memory where each processor has a single write buffer. A program consists of lists of read and write instructions (each list representing a single processor). The execution of a write instruction proceeds in two stages: the write is first enqueued in the processor's write buffer and some time later it is dequeued and written (committed) to memory. The execution of a read instruction reads the memory "through" the local write buffer: if there is a pending write to the location of the read, then the read operation reads its result from the latest pending write to that location, otherwise it reads the value from memory.

We think of program executions on this machine as words over an alphabet Σ of events. The letters in this alphabet are tuples $Proc \times Id \times Action$ where $Proc$ (the processor identifier) and Id (the event identifier) are both natural numbers and $Action$ is a pair $Op \times Loc$ where Op is either R (read), W (write) or C (commit) and Loc is the memory location, which is also represented as a natural number. In this simple example, we ignore the values read and written by the events because the independence relation we define does not consider them.

Two events from different processors are dependent when they access the same memory location and at least one of them is a write. We do not consider a W event to access the memory as it only affects the local write buffer. Similarly, an R event only accesses the memory when it reads its value from memory (and not from the write buffer). Two events from the same processor are dependent if they are W or R events (we respect the program order) or both are C events (the buffer is *first-in-first-out*) or they are a corresponding pair of a W and C event (this C is the commit of this W event).

We have defined this context-dependent independence relation also in our formalisation and shown that it satisfies the consistency (Definition 4.3) and coherence (Definition 4.4) conditions. This allows us to use the normalisation algorithms we have developed for deciding whether two executions (words over the alphabet) on this machine are equivalent.

Let us consider the following program where two processors write to variable `x` and one of them also reads from `x`.

P_1		P_2	
`(a,a')`	`[x] := 1`	`(c,c')`	`[x] := 2`
`(b)`	`r1 := [x]`		

We can represent the instructions as events in the following way. As mentioned before, the execution of write instructions proceeds in two stages. Thus, the events representing `[x] := 1` are $a = 1,1,W,1$ and $a' = 1,1,C,1$; they share the event identifier as they correspond to the write and commit stages of the same instruction. The event for the read `r1 := [x]` is $b = 1,2,R,1$. The events representing `[x] := 2` are $c = 2,1,W,1$ and $c' = 2,1,C,1$.

We consider the possible executions of the first processor to be $aa'b$ and aba'. This is a consequence of the independence relation: after performing a, the event a' is pending; since a' and b are independent, the processor has a choice which one to perform next. The second processor can only execute as cc'. The possible executions of the whole program are all the possible interleavings of an execution from the first processor with an execution from the second processor. This program has 20 possible executions in total and these are partitioned into three equivalence classes. This is summarised in the following table.

A consequence of this independence relation is that the executions $acc'ba'$ and $acbc'a'$ are equivalent. Interestingly, the two executions only differ by the ordering of b and c'. The equivalence of the two, justified by b and c' being independent in a context containing a but not a', may seem counterintuitive as b is supposed to read x and c' is supposed to write to x. It may look as if the value read by b could be affected by c'. But this is not so. The two executions are semantically equivalent because b appears before a' (i.e., in the presence of a pending write to x) and thus b reads the value of x from the write buffer and not from memory.

The effect described in the previous paragraph is also visible from the Foata normal forms in the table. The second one, $(ac)(a')(c')(b)$, implies that c' and b are dependent in the context aca'. The third one, $(ac)(bc')(a')$, implies that b and c' are independent in the context ac.

Execution	Foata	Lexicographic
$aa'bcc'$ $aba'cc'$ $aa'cbc'$ $abca'c'$ $aca'bc'$ $acba'c'$ $caa'bc'$ $caba'c'$	$(ac)(a'b)(c')$	$aa'bcc'$
$aa'cc'b$ $aca'c'b$ $caa'c'b$	$(ac)(a')(c')(b)$	$aa'cc'b$
$abcc'a'$ $acbc'a'$ $cabc'a'$ $acc'a'b$ $acc'ba'$ $cac'a'b$ $cac'ba'$ $cc'aa'b$ $cc'aba'$	$(ac)(bc')(a')$	$abcc'a'$

If we were to model this using ordinary traces (without context-dependence), then we would have to choose whether to set b and c' to be dependent or independent. The only reasonable choice is to let them be dependent as the equivalence relation induced by this is a safe approximation of the one above: equivalent executions according to the new relation are equivalent according to the previous relation. The downside of this is that there will be more equivalence classes. If we would set b and c' to be independent, then we would have that $aca'c'b$ and $aca'bc'$ are equivalent. This does not agree with the equivalence relation defined above, but more importantly, it does not make sense semantically as b reads the value of x from memory (instead of the write buffer) and is thus affected by c'.

4.6 Related Work

Traces were introduced into concurrency theory by Mazurkiewicz [52], but they originate from the enumerative combinatorics work by Cartier and Foata [21]. In particular, Foata normalization is from that work. The lexicographic normalization was first investigated by Anisimov and Knuth [9]. These two normal forms and normalization algorithms are described in many of the standard expositions of trace theory, e.g., [2, 26].

Generalising traces for context-dependent independence has been considered by several authors, but with different well-behavedness conditions on independence. Sassone et al. [74] introduced context-dependent independence as we have considered it. Katz and Peled [38] introduced conditional independence, considering a coherence condition that in our setting would amount to $a\, I_u\, b$ and $b\, I_{ua}\, c$ implying ($a\, I_u\, c$ iff $a\, I_{ub}\, c$). This condition is equivalent to the conjunction of conditions (1) and (2) of Definition 4.4. Droste [28], in a work on concurrent automata, again with state-dependent independence, required what would in this setting amount to $a\, I_u\, b$, $b\, I_u\, c$, $a\, I_{ub}\, c$ implying $a\, I_{uc}\, b$, $b\, I_{ua}\, c$, $a\, I_u\, c$, i.e.,

condition (3) of Definition 4.4 and a little more.

Hoogers et al. [34] developed local traces where independence relates lists of steps to steps. This is a different setup where coherence conditions like those of Sassone et al. do not arise, because one only works with contexts of steps, not contexts of individual letters.

Partial-order reduction (POR) and use of representatives in model-checking, originally proposed by Godefroid [31] and Peled [64], are in wide use. We mention that dynamic POR for stateless model-checking of relaxed memory concurrent programs in particular has been considered by Abdulla et al. [4] and Zhang et al. [79]. In our own previous work [47], we used Foata normal forms of generalised traces as the representative executions when we considered memory models of the SPARC hierarchy.

Chou and Peled [23] have formalised standard Mazurkiewicz traces in the context of formally verifying a partial-order reduction technique in HOL. Yang et al. [78], Aspinall and Sevčik [11] and Owens et al. [61] pioneered the formalisation of semantics of relaxed memory models with proof assistants, using HOL, Isabelle/HOL, HOL4.

There are some parallels of Lipton's theory of reduction (movers) [45] to trace theory, or more precisely, semicommutations [24], where independence is non-symmetric. Movers have been used in reasoning about relaxed memory concurrency by Bouajjani et al. [15].

4.7 Conclusion and Future Work

We believe it to be important to exercise care when choosing the semantic domain for behaviours for a class of concurrent systems. Descriptions of behaviours in terms of an apparently more involved abstraction can sometimes be more precise, yet still as analysable. In this chapter, we certified two normalisation algorithms for generalised Mazurkiewicz traces. The example from Section 4.5 demonstrates that standard Mazurkiewicz traces are not flexible enough in some circumstances and generalised traces can lead to fewer equivalence classes. This is good in any situation where one needs to exhaustively check an equivalence-invariant property on all equivalence classes. In Section 5.7 we will see how (an extension of) the independence alphabet from Section 4.5 can be used to give a TSO-like semantics to a simple programming language.

5 Operational Semantics with Semicommutations

In this chapter, we describe an operational semantics that allows to execute the instructions of a program in an order that is different from the order given by the program, i.e., an instruction might be executed before its preceding instructions have been executed. The idea is that such an operational semantics can capture some optimisations that a compiler or a runtime environment might apply to the program. To accomplish this, we further generalise the Antimirov reordering derivatives from Chapter 3. First, we include parallel composition in the syntax. Second, we let go of the requirement that an independence relation has to be symmetric. Third, we interpret letters of the alphabet as state transformers and we also include machine states in the rules. Altogether, this enables us to describe the semantics of a simple While-like language that allows some reordering of instructions. We then describe how representative executions can be used to alleviate the combinatorial explosion when computing the set of final states of litmus-tests. Finally, we consider a few extensions to the operational semantics that can account for more fine-grained reordering relations. With these extensions we give an operational semantics for a TSO-like system where a read instruction can read its value from a write instruction earlier in the program that has not yet been executed.

5.1 Motivation

In this chapter, we come back to the example we showed in the Introduction where we looked at the following program representing the message-passing pattern.

$$x := 41; y := 1 \parallel r_1 := y; r_2 := x$$

To recollect, here the first thread stores to variable x the result of some computation (which just happens to be 41) and then sets the variable y to 1 to indicate that it has finished this computation and the result is now stored in x. The question that interests us is if it is possible that some other thread at some point sees the new value of y but the old value of x. In the usual interleaving semantics, which is sequentially consistent, this is not possible. If, for example, we consider the instructions $r_1 := y$ and $r_2 := x$ to be independent enough so that some optimisation could reorder them, then we could end up with the following program

$$x := 41; y := 1 \parallel r_2 := x; r_1 := y$$

which under the interleaving semantics can lead to a final state where $r_1 = 1$ and $r_2 = 0$.

The motivation for this work is that concurrent programs typically are not executed in the intuitive interleaving fashion. The reason is that both the compilation process and the hardware itself (during the execution) may change the program slightly so as to make the program execution more efficient. A reasonable requirement is that these optimisations are invisible for a single-threaded program. But as we saw in the example above, the effects of such optimisations may become visible in a concurrent setting.

Our goal in this chapter is to describe the execution of programs in a manner that is weaker than sequential consistency. In other words, we consider execution of programs under some weak memory model. In the following, we describe an operational semantics which is able to capture some optimisations like the one above where two instructions were reordered. The main idea is that, given a program $p; q$, we allow, under certain conditions, to execute an instruction from q even when p is not yet fully executed.

5.2 Preliminaries

Before we move on to operational semantics, we briefly describe the differences between semitraces (that we use here) and Mazurkiewicz traces. We also describe how we represent programs in this framework.

5.2.1 Semicommutations

Semicommutations [24] are a generalisation of Mazurkiewicz traces where the independence relation is not necessarily symmetric. A semicommutation alphabet (Σ, θ) is a pair consisting of an alphabet Σ and an irreflexive relation $\theta \subseteq \Sigma \times \Sigma$, called the *semicommutation* relation.

We extend the semicommutation θ to a relation on words and letters by $\theta(\varepsilon, a) =_{\mathrm{df}} \mathrm{tt}$ and $\theta(ub, a) =_{\mathrm{df}} \theta(u,a) \wedge \theta(b,a)$.

We write $\Rightarrow^{\theta} \subseteq \Sigma^* \times \Sigma^*$ for the least relation such that $\theta(a,b)$ implies $uabv \Rightarrow^{\theta} ubav$ for all $u, v \in \Sigma^*$ (this corresponds to $uabv \sim^{I} ubav$ in Mazurkiewicz traces). Similarly, we write $\Rightarrow^{\theta *}$ for the reflexive-transitive closure of $\Rightarrow^{\theta}$ (this corresponds to $\sim^{I*}$ in Mazurkiewicz traces). The relation $\Rightarrow^{\theta *}$ is the rewriting relation induced by θ.

When $\theta(u,a)$, then ua can be rewritten to au, i.e., $ua \Rightarrow^{\theta *} au$ and later we will also say that in this case a can be reordered before u. This (multi-step) rewriting relation is monotone in θ in the sense that, if $\theta \subseteq \theta'$, then $u \Rightarrow^{\theta *} u'$ implies $u \Rightarrow^{\theta' *} u'$. The closure of a word u under θ, denoted by $\theta(u)$, is the set of words it can be rewritten to, i.e., $\theta(u) =_{\mathrm{df}} \{v \in \Sigma^* \mid u \Rightarrow^{\theta *} v\}$. Thus we have that $\theta \subseteq \theta'$ implies $\theta(u) \subseteq \theta'(u)$. The closure of a language is defined as $\theta(L) =_{\mathrm{df}} \bigcup\{\theta(u) \mid u \in L\}$ and, for any L, it satisfies $L \subseteq \theta(L)$ and $\theta(\theta(L)) \subseteq \theta(L)$.

In the case where θ is also symmetric, we obtain the usual Mazurkiewicz traces. Then, for a word u, the closure $\theta(u)$ is the equivalence class of u (the trace $[u]^{\theta}$).

5.2.2 Programs

We use regular expressions over some alphabet of instructions as the abstract syntax with which we describe programs. More precisely, here we use regular expressions RE extended with the shuffle operation. The set RES of *regular expressions with shuffle* is given by the grammar:

$$E ::= a \mid 0 \mid E + E \mid 1 \mid EE \mid E^* \mid E \parallel E$$

where a ranges over Σ. We consider expressions $E \in \mathrm{RES}$ to be programs over an alphabet Σ of instructions. Multiplication EF stands for sequential composition, addition $E + F$ is nondeterministic choice, Kleene star E^* is iteration and $E \parallel F$ is parallel composition. We consider 0 to represent failure and 1 to be the no-op program representing successful termination. In this chapter, we write $E \doteq F$ to mean that E and F are equal modulo associativity of multiplication.

The set of machine states is denoted by $\mathbb{S}$. Each instruction $a \in \Sigma$ is interpreted as a (partial) state transformer: $[\![a]\!] : \mathbb{S} \rightharpoonup \mathbb{S}$. We write $[\![a]\!]\sigma\downarrow$ to say that $[\![a]\!]\sigma$ is defined. We often use the following postfix notation for the state transformers: $(\sigma)a =_{\mathrm{df}} [\![a]\!]\sigma$. This extends to words as: $(\sigma)\varepsilon =_{\mathrm{df}} \sigma$ and $(\sigma)au =_{\mathrm{df}} ((\sigma)a)u$ if $(\sigma)a\downarrow$ and $\perp$ otherwise. For instructions a and b, if $\theta(a,b)$, then we allow to execute b before a in their sequential composition ab.

5.3 Reordering Semantics

We consider an execution of a program (from some initial state) to be a sequence of instructions that in a sense is allowed by the program. The result of this execution is the state obtained by applying this sequence of instructions to the initial state. The operational semantics should specify, given a program and a machine state, what are the allowed executions. This is accomplished by specifying the set of instructions (or instruction occurrences) in the program from which we are allowed to choose the next instruction to execute and, for each such instruction, also the program that we are left with after executing the instruction. This in turn describes how to construct an (allowed) execution (a sequence of instructions) of a program. Our goal is to define the rules of the semantics so that we can also include certain reorderings in the semantics. The general idea is that we allow reorderings by making more instructions of the program available for execution.

5.3.1 Word Language Interpretation of Programs

Before we turn to operational semantics, we describe a word language interpretation of programs that is closed under the rewriting relation $\Rightarrow^{\theta *}$ (this interpretation corresponds to $[\![\_]\!]^I$ which is closed under $\sim^{I*}$). This can be seen as an overapproximation of the allowed executions: here we consider all executions that a program might generate and ignore the question whether these executions are well-defined, i.e., if an execution applied to the initial state is defined at all. First, we define the "loosened concatenation" of two words.

Definition 5.1. The *θ-reordering concatenation* of words $\cdot^\theta : \Sigma^* \times \Sigma^* \to \mathscr{P}(\Sigma^*)$ is defined by

$$\begin{array}{rcl} \varepsilon \cdot^\theta v & =_{\mathrm{df}} & \{v\} \\ u \cdot^\theta \varepsilon & =_{\mathrm{df}} & \{u\} \\ au \cdot^\theta bv & =_{\mathrm{df}} & \{a\} \cdot (u \cdot^\theta bv) \cup \{b \mid \theta(au, b)\} \cdot (au \cdot^\theta v) \end{array}$$

and the lifting of θ-reordering concatenation to *languages* is defined by

$$L \cdot^\theta L' \quad =_{\mathrm{df}} \quad \bigcup \{u \cdot^\theta v \mid u \in L \wedge v \in L'\}.$$

We can see that $u \cdot^\theta v$ is sublanguage of $u \sqcup\!\sqcup v$. Similarly to $\cdot^I$, this concatenation operation is also biased towards the left argument: $\theta(au, b)$ is a test for whether to include the language $\{b\} \cdot (au \cdot^\theta v)$ in the result.

The θ-reordering concatenation satisfies the following two useful properties. First, it is monotone in θ.

Lemma 5.2. *For any θ, θ' and $L, L' \subseteq \Sigma^*$, if $\theta \subseteq \theta'$, then $L \cdot^\theta L' \subseteq L \cdot^{\theta'} L'$.*

Second, the closure operation distributes over concatenation in the following way. (Or we can say that θ is a monoid morphism from languages to θ-closed languages.)

Lemma 5.3. *For any θ and $L, L' \subseteq \Sigma^*$, we have $\theta(L \cdot L') = \theta(L) \cdot^\theta \theta(L')$.*

The shuffle operation satisfies a weaker property.

Lemma 5.4. *For any θ and $L, L' \subseteq \Sigma^*$, we have $\theta(L \sqcup\!\sqcup L') \subseteq \theta(L) \sqcup\!\sqcup \theta(L')$.*

Next, we define the word language semantics of programs. We see this as a description of all the possible ways a program *could* execute—the words in this interpretation of a program are those sequences of instructions that could possibly occur as executions of the program.

Definition 5.5. The *word language semantics* $[\![\_]\!]^\theta : \mathrm{RES} \to \mathscr{P}(\Sigma^*)$ of programs is defined recursively by

$$\begin{array}{rcl} [\![a]\!]^\theta & =_{\mathrm{df}} & \{a\} \\ [\![0]\!]^\theta & =_{\mathrm{df}} & \emptyset \\ [\![E+F]\!]^\theta & =_{\mathrm{df}} & [\![E]\!]^\theta \cup [\![F]\!]^\theta \\ [\![1]\!]^\theta & =_{\mathrm{df}} & \mathbf{1} \\ [\![EF]\!]^\theta & =_{\mathrm{df}} & [\![E]\!]^\theta \cdot^\theta [\![F]\!]^\theta \\ [\![E^*]\!]^\theta & =_{\mathrm{df}} & \mu X.\, \mathbf{1} \cup [\![E]\!]^\theta \cdot^\theta X \\ [\![E \parallel F]\!]^\theta & =_{\mathrm{df}} & [\![E]\!]^\theta \sqcup [\![F]\!]^\theta \end{array}$$

The interpretation $[\![E]\!]^\theta$ is monotone in the semicommutation θ and it is closed under the rewriting relation $\Rightarrow^{\theta *}$.

Proposition 5.6. *For any θ, θ', and $E \in \mathrm{RES}$, if $\theta \subseteq \theta'$, then $[\![E]\!]^\theta \subseteq [\![E]\!]^{\theta'}$.*

Proof. By induction on E. We only show two cases here.

- Case EF: By i.h. we have $[\![E]\!]^\theta \subseteq [\![E]\!]^{\theta'}$ and $[\![F]\!]^\theta \subseteq [\![F]\!]^{\theta'}$. Thus, by Lemma 5.2 we have: $[\![EF]\!]^\theta = [\![E]\!]^\theta \cdot^\theta [\![F]\!]^\theta \subseteq [\![E]\!]^\theta \cdot^{\theta'} [\![F]\!]^\theta \subseteq [\![E]\!]^{\theta'} \cdot^{\theta'} [\![F]\!]^{\theta'} = [\![EF]\!]^{\theta'}$.
- Case $E \parallel F$: By i.h. we have $[\![E]\!]^\theta \subseteq [\![E]\!]^{\theta'}$ and $[\![F]\!]^\theta \subseteq [\![F]\!]^{\theta'}$. Thus we have the following: $[\![E \parallel F]\!]^\theta = [\![E]\!]^\theta \sqcup [\![F]\!]^\theta \subseteq [\![E]\!]^{\theta'} \sqcup [\![F]\!]^{\theta'} = [\![E \parallel F]\!]^{\theta'}$. □

Proposition 5.7. *For any θ and $E \in \mathrm{RES}$, we have $\theta([\![E]\!]^\theta) = [\![E]\!]^\theta$.*

Proof. By induction on E. We only show two cases here.

- Case EF: By i.h. and Lemma 5.3 we have the following equations.

$$\begin{array}{rcl} \theta([\![EF]\!]^\theta) & = & \theta([\![E]\!]^\theta \cdot^\theta [\![F]\!]^\theta) \\ & = & \theta(\theta([\![E]\!]^\theta) \cdot^\theta \theta([\![F]\!]^\theta)) \\ & = & \theta(\theta([\![E]\!]^\theta \cdot [\![F]\!]^\theta)) \\ & = & \theta([\![E]\!]^\theta \cdot [\![F]\!]^\theta) \\ & = & \theta([\![E]\!]^\theta) \cdot^\theta \theta([\![F]\!]^\theta) \\ & = & [\![E]\!]^\theta \cdot^\theta [\![F]\!]^\theta \\ & = & [\![EF]\!]^\theta \end{array}$$

- Case $E \parallel F$: By i.h. $[\![E]\!]^\theta$ and $[\![F]\!]^\theta$ are closed and thus by Lemma 5.4 we have that $\theta([\![E \parallel F]\!]^\theta) = \theta([\![E]\!]^\theta \sqcup [\![F]\!]^\theta) \subseteq \theta([\![E]\!]^\theta) \sqcup \theta([\![F]\!]^\theta) = [\![E]\!]^\theta \sqcup [\![F]\!]^\theta = [\![E \parallel F]\!]^\theta$ and by the closure property of θ we have that $\theta([\![E \parallel F]\!]^\theta) \supseteq [\![E \parallel F]\!]^\theta$. □

Note that although the interpretation $[\![E]\!]^\theta$ is closed, unlike what we had in Chapter 3, it is not the case, in general, that it is the closure $\theta([\![E]\!])$. (Here we write $[\![E]\!]$ for $[\![E]\!]^\emptyset$.) In general, we only have $\theta([\![E]\!]) \subseteq [\![E]\!]^\theta$.

Proposition 5.8. *For any semicommutations θ and θ', and $E \in \mathrm{RES}$, if $\theta \subseteq \theta'$, then $\theta'([\![E]\!]^\theta) \subseteq [\![E]\!]^{\theta'}$.*

Proof. Let $u' \in \theta'([\![E]\!]^\theta)$. Then there must exist $u \in [\![E]\!]^\theta$ such that $u \Rightarrow^{\theta' *} u'$. By Proposition 5.6 we have that $u \in [\![E]\!]^{\theta'}$ and by Proposition 5.7 we thus have $u' \in [\![E]\!]^{\theta'}$. □

The following example demonstrates that we do not have $\theta'([\![E]\!]^\theta) \supseteq [\![E]\!]^{\theta'}$ in general.

Example 5.9. Let $\Sigma =_{\mathrm{df}} \{a,b,c\}$, $\theta = \emptyset$ and $\theta'(a,b)$. Then we have the following.

$$\begin{array}{rcl}
\theta'([\![ab \parallel c]\!]) & = & \theta'([\![ab]\!] \sqcup [\![c]\!]) \\
& = & \theta'(\{ab\} \sqcup \{c\}) \\
& = & \theta'(\{abc, acb, cab\}) \\
& = & \{abc, bac, acb, cab, cba\} \\
& \not\supseteq & \{abc, acb, cab, bac, bca, cba\} \\
& = & \{ab, ba\} \sqcup \{c\} \\
& = & [\![ab]\!]^{\theta'} \sqcup [\![c]\!]^{\theta'} \\
& = & [\![ab \parallel c]\!]^{\theta'}
\end{array}$$

This mismatch is due to parallel composition. Intuitively, in $E \parallel F$ there are no ordering constraints between the instructions in E and instructions in F, but with $[\![E]\!] \sqcup [\![F]\!]$ we pick an ordering for the pairs of instructions that do not commute. If we only consider regular expressions $E \in \mathrm{RE}$, then we indeed have $\theta([\![E]\!]) = [\![E]\!]^{\theta}$.

Proposition 5.10. *For any θ and θ', and $E \in \mathrm{RE}$, if $\theta \subseteq \theta'$, then $\theta'([\![E]\!]^{\theta}) = [\![E]\!]^{\theta'}$.*

Proof. By induction on E. We only show the case for multiplication.

- Case EF: By Proposition 5.7, Lemma 5.3, the assumption that $\theta \subseteq \theta'$, and, by i.h., we have the following equations.

$$\begin{array}{rcl}
\theta'([\![EF]\!]^{\theta}) & = & \theta'([\![E]\!]^{\theta} \cdot^{\theta} [\![F]\!]^{\theta}) \\
& = & \theta'(\theta([\![E]\!]^{\theta}) \cdot^{\theta} \theta([\![F]\!]^{\theta})) \\
& = & \theta'(\theta([\![E]\!]^{\theta} \cdot [\![F]\!]^{\theta})) \\
& = & \theta'([\![E]\!]^{\theta} \cdot [\![F]\!]^{\theta}) \\
& = & \theta'([\![E]\!]^{\theta}) \cdot^{\theta'} \theta'([\![F]\!]^{\theta}) \\
& = & [\![E]\!]^{\theta'} \cdot^{\theta'} [\![F]\!]^{\theta'} \\
& = & [\![EF]\!]^{\theta'}
\end{array}$$

□

The above proposition highlights why it might be interesting to use this approach for the (relaxed) semantics of programs. We think of $E \in \mathrm{RE}$ as a single-threaded program and we interpret $\theta([\![E]\!]) = [\![E]\!]^{\theta}$ as saying that the "optimisations" described by θ are valid (in a sequential context) in the sense that no new behaviour is introduced—for every execution u' in $[\![E]\!]^{\theta}$ there exists u in $[\![E]\!]$ such that $u \Rightarrow^{\theta *} u'$, i.e., u and u' are "equivalent". At the same time, optimisations valid in a sequential context might not be valid in a concurrent context in the sense that applying such optimisations to individual threads of a concurrent program might introduce new behaviours as shown in Example 5.9.

5.3.2 Reorderability

The key idea in the closing interpretation is the use of the θ-reordering concatenation. The important part of the reordering concatenation was the use of $\theta(au, b)$ as a test for whether to include $\{b\} \cdot (au \cdot^{\theta} v)$ in the concatenation $au \cdot^{\theta} bv$. Our goal now is to define the analogue of $\theta(au, b)$ for programs. This notion of reorderability is the essential difference compared to ordinary operational semantics. Intuitively, we would like to say that executing an instruction a before executing the preceding sequence of instructions u is valid when we know that both ua and au would lead to the same state.

Definition 5.11. A semicommutation θ is *conservative* when, for every $a, b \in \Sigma$,

$$\theta(a,b) \Longrightarrow \forall \sigma.(\sigma)ab = (\sigma)ba.$$

The equality in the above definition is the Kleene equality: if either side is defined, then so is the other and they are equal. The motivation for the conservativity condition on the semicommutation relation is apparent in the following lemma.

Lemma 5.12. *For any conservative θ and $u, u' \in \Sigma^*$, if $u \Rightarrow^{\theta *} u'$, then $(\sigma)u = (\sigma)u'$.*

Of course, as a special case of the above lemma, if $\theta(u,a)$, then $ua \Rightarrow^{\theta *} au$ and thus we have $(\sigma)ua = (\sigma)au$. In the context of operational semantics, we consider conservative semicommutation relations and thus we take $\theta(u,a)$ to be the justification which allows us to reorder a before u in the sequence of instructions ua.

Intuitively, given a program Ea, we would like to find a program E' such that, for every execution u of E', u is also an execution of E and a can be reordered before u. Put another way, when we reorder a in front of E in the program Ea, we would like to restrict E to the subprogram E' and the result of the reordering is aE'. This program E' corresponds to the I-reorderable part.

Definition 5.13. The *θ-reorderable part* of a program is given by the function $\mathsf{R}^\theta : \mathrm{RES} \times \Sigma \to \mathrm{RES}$ defined recursively by

$$\begin{array}{rcl}
\mathsf{R}^\theta_a b & =_{\mathrm{df}} & \text{if } \theta(b,a) \text{ then } b \text{ else } 0 \\
\mathsf{R}^\theta_a 0 & =_{\mathrm{df}} & 0 \\
\mathsf{R}^\theta_a (E+F) & =_{\mathrm{df}} & \mathsf{R}^\theta_a E + \mathsf{R}^\theta_a F \\
\mathsf{R}^\theta_a 1 & =_{\mathrm{df}} & 1 \\
\mathsf{R}^\theta_a (EF) & =_{\mathrm{df}} & (\mathsf{R}^\theta_a E)(\mathsf{R}^\theta_a F) \\
\mathsf{R}^\theta_a (E^*) & =_{\mathrm{df}} & (\mathsf{R}^\theta_a E)^* \\
\mathsf{R}^\theta_a (E \parallel F) & =_{\mathrm{df}} & \mathsf{R}^\theta_a E \parallel \mathsf{R}^\theta_a F
\end{array}$$

Similarly to $\mathsf{R}^I_a E$, we have that $\mathsf{R}^\theta_a E$ just replaces those instructions b in E with 0 for which $\theta(b,a)$ does not hold.

Example 5.14. Let $\Sigma =_{\mathrm{df}} \{a,b,c\}$ and $\theta(a,b)$ and $\theta(c,a)$. Then

- $\mathsf{R}^\theta_a((a \parallel 1)a(a+b+c)aa^*) = (0 \parallel 1)0(0+0+c)00^*$;
- $\mathsf{R}^\theta_b((a \parallel 1)a(a+b+c)aa^*) = (a \parallel 1)a(a+0+0)aa^*$;
- $\mathsf{R}^\theta_c((a \parallel 1)a(a+b+c)aa^*) = (0 \parallel 1)0(0+0+0)00^*$.

Lemma 5.15. *For any semicommutation θ, $a, b \in \Sigma$ and $E \in \mathrm{RES}$, the θ-reorderable part R^θ enjoys the following properties:*

- $\mathsf{R}^\theta_b(\mathsf{R}^\theta_a E) = \mathsf{R}^\theta_a(\mathsf{R}^\theta_b E)$;
- $\mathsf{R}^\theta_a(\mathsf{R}^\theta_a E) = \mathsf{R}^\theta_a E$.

If we were to extend θ-reorderable part to words, i.e., by having $\mathsf{R}^\theta_\varepsilon E =_{\mathrm{df}} E$ and $\mathsf{R}^\theta_{au} E =_{\mathrm{df}} \mathsf{R}^\theta_u(\mathsf{R}^\theta_a E)$, then, by the previous lemma, it would be the case that $\mathsf{R}^\theta_u E = \mathsf{R}^\theta_{\Sigma(u)} E$.

Definition 5.16. The relation $\preceq \subseteq \mathrm{RES} \times \mathrm{RES}$ is the precongruence on RES generated by $0 \preceq a$ where $a \in \Sigma$.

Thus $E \preceq F$ holds when E is otherwise exactly the same as F except that some of the instructions in it may have been replaced with 0. We have $a + 0 \preceq a + b$, but note that $a \not\preceq a + b$. If $E \preceq F$, then $[\![E]\!]^\theta \subseteq [\![F]\!]^\theta$ but the converse does not hold. We have the following properties for the reorderable part.

Lemma 5.17. *For any semicommutations θ and θ', $E, E' \in \mathsf{RES}$ and $a \in \Sigma$,*

1. $\mathsf{R}_a^\theta E \preceq E$;
2. *if $\theta \subseteq \theta'$, then $\mathsf{R}_a^\theta E \preceq \mathsf{R}_a^{\theta'} E$;*
3. *if $E \preceq E'$, then $\mathsf{R}_a^\theta E \preceq \mathsf{R}_a^\theta E'$.*

Importantly, the reorderable part of a program corresponds to the reorderable words in the word language semantics.

Proposition 5.18. *For any semicommutation θ, $a \in \Sigma$, $u \in \Sigma^*$ and $E \in \mathsf{RES}$, we have that $u \in [\![\mathsf{R}_a^\theta E]\!]^\theta \iff u \in [\![E]\!]^\theta \wedge \theta(u,a)$.*

5.3.3 Operational Semantics

With the θ-reorderable part defined we can now define the θ-reordering operational semantics. This is essentially the Antimirov reordering derivative we defined in Section 3.3.3 with some modifications.

Definition 5.19. The *θ-reordering single-step reduction* of a program is given by the relation $\to^\theta \subseteq \mathbb{S} \times \mathsf{RES} \times \Sigma \times \mathbb{S} \times \mathsf{RES}$:

$$\frac{(\sigma)a\downarrow}{\langle \sigma, a\rangle \to^\theta (a, \langle (\sigma)a, 1\rangle)}$$

$$\frac{\langle \sigma, E\rangle \to^\theta (a, \langle \sigma', E'\rangle)}{\langle \sigma, E+F\rangle \to^\theta (a, \langle \sigma', E'\rangle)} \quad \frac{\langle \sigma, F\rangle \to^\theta (a, \langle \sigma', F'\rangle)}{\langle \sigma, E+F\rangle \to^\theta (a, \langle \sigma', F'\rangle)}$$

$$\frac{\langle \sigma, E\rangle \to^\theta (a, \langle \sigma', E'\rangle)}{\langle \sigma, EF\rangle \to^\theta (a, \langle \sigma', E'F\rangle)} \quad \frac{\langle \sigma, F\rangle \to^\theta (a, \langle \sigma', F'\rangle)}{\langle \sigma, EF\rangle \to^\theta (a, \langle \sigma', (\mathsf{R}_a^\theta E)F'\rangle)}$$

$$\frac{\langle \sigma, E\rangle \to^\theta (a, \langle \sigma', E'\rangle)}{\langle \sigma, E^*\rangle \to^\theta (a, \langle \sigma', (\mathsf{R}_a^\theta E)^* E' E^*\rangle)}$$

$$\frac{\langle \sigma, E\rangle \to^\theta (a, \langle \sigma', E'\rangle)}{\langle \sigma, E \parallel F\rangle \to^\theta (a, \langle \sigma', E' \parallel F\rangle)} \quad \frac{\langle \sigma, F\rangle \to^\theta (a, \langle \sigma', F'\rangle)}{\langle \sigma, E \parallel F\rangle \to^\theta (a, \langle \sigma', E \parallel F'\rangle)}$$

The key difference with ordinary operational semantics is in the second rule of multiplication and the rule for Kleene star. These are the only places where R^θ is used since these are the only rules where we consider the instruction we are executing (reordering) to cross a sequential composition (the "boundary" between two sequentially composed programs). Another way to say this is that we apply R_a^θ (in the original program E) to the left context of the instruction a that we are executing. By left context of an instruction a in program E we mean the part of the program E that is before a wrt. sequential composition.

Example 5.20. Consider the program $ab \parallel (c+d)ef+g$. If, for example, we have $\theta(c,e)$, $\neg\theta(d,e)$ and $(\sigma)e\downarrow$, then we have the following derivation for executing e as the next

instruction. The left context of e in this example is $c+d$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\sigma)e\downarrow}{\langle\sigma,e\rangle \to^{\theta} (e,\langle(\sigma)e,1\rangle)}}{\langle\sigma,ef\rangle \to^{\theta} (e,\langle(\sigma)e,1f\rangle)}}{\langle\sigma,(c+d)ef\rangle \to^{\theta} (e,\langle(\sigma)e,(c+0)1f\rangle)}}{\langle\sigma,(c+d)ef+g\rangle \to^{\theta} (e,\langle(\sigma)e,(c+0)1f\rangle)}}{\langle\sigma,ab \parallel (c+d)ef+g\rangle \to^{\theta} (e,\langle(\sigma)e,ab \parallel (c+0)1f\rangle)}$$

The instruction g disappeared from the program since the rules for $+$ resolve the nondeterminism. The instruction d became 0 since it was before e and reordering e with d was not justified. More precisely, we have $\mathrm{R}^{\theta}_{e}(c+d) = c+0$ since $\theta(c,e)$ and $\neg\theta(d,e)$.

The single-step reduction rules respect $\preceq$ and $\doteq$.

Lemma 5.21. *For any semicommutations θ and θ', $\sigma,\sigma' \in \mathbb{S}$, $E,E',F \in \mathrm{RES}$ and $a \in \Sigma$,*

1. *if $\langle\sigma,E\rangle \to^{\theta} (a,\langle\sigma',E'\rangle)$, $\theta \subseteq \theta'$ and $E \preceq F$, then exists $F' \in \mathrm{RES}$ such that $\langle\sigma,F\rangle \to^{\theta'} (a,\langle\sigma',F'\rangle)$ and $E' \preceq F'$;*
2. *if $\langle\sigma,E\rangle \to^{\theta} (a,\langle\sigma',E'\rangle)$ and $E \doteq F$, then exists $F' \in \mathrm{RES}$ such that $\langle\sigma,E\rangle \to^{\theta} (a,\langle\sigma',E'\rangle)$ and $E' \doteq F'$.*

We also have that the single-step reduction rules commute when the labels commute. To show this we need a couple of properties. First, the label of the rule is the instruction that is applied (successfully) to the state.

Lemma 5.22. *For any semicommutation θ, $a \in \Sigma$, $\sigma,\sigma' \in \mathbb{S}$ and $E,E' \in \mathrm{RES}$, if $\langle\sigma,E\rangle \to^{\theta} (a,\langle\sigma',E'\rangle)$, then $(\sigma)a\downarrow$ and $(\sigma)a = \sigma'$.*

The same derivation can be made from any state where the instruction can be successfully applied to the state.

Lemma 5.23. *For any semicommutation θ, $a \in \Sigma$, $\sigma,\sigma' \in \mathbb{S}$ and $E,E' \in \mathrm{RES}$, if $\langle\sigma,E\rangle \to^{\theta} (a,\langle(\sigma)a,E'\rangle)$ and $(\sigma')a\downarrow$, then $\langle\sigma',E\rangle \to^{\theta} (a,\langle(\sigma')a,E'\rangle)$.*

If we can make a step from a reorderable part of a program, then we can also make the same step from the original program.

Lemma 5.24. *For any semicommutation θ, $a,b \in \Sigma$, $\sigma \in \mathbb{S}$ and $E,E' \in \mathrm{RES}$, if $\langle\sigma,\mathrm{R}^{\theta}_{b}E\rangle \to^{\theta} (a,\langle(\sigma)a,E'\rangle)$, then there exists $E'' \in \mathrm{RES}$ such that $\mathrm{R}^{\theta}_{b}E'' = E'$ and $\langle\sigma,E\rangle \to^{\theta} (a,\langle(\sigma)a,E''\rangle)$.*

If we can make a step with an instruction a and we have $\theta(a,b)$, then we can also make this step from the θ-reorderable part (wrt. b) of the program.

Lemma 5.25. *For any semicommutation θ, $a,b \in \Sigma$, $\sigma \in \mathbb{S}$ and $E,E' \in \mathrm{RES}$, if $\langle\sigma,E\rangle \to^{\theta} (a,\langle(\sigma)a,E'\rangle)$ and $\theta(a,b)$, then also $\langle\sigma,\mathrm{R}^{\theta}_{b}E\rangle \to^{\theta} (a,\langle(\sigma)a,\mathrm{R}^{\theta}_{b}E'\rangle)$.*

If $\theta(a,b)$, then we can commute the steps labelled by a and b in the following sense.

Lemma 5.26. *For any semicommutation θ, $a,b \in \Sigma$, $\sigma \in \mathbb{S}$ and E,E',E'', if $\theta(a,b)$,*

$$\langle\sigma,E\rangle \to^{\theta} (a,\langle(\sigma)a,E'\rangle) \quad \textit{and} \quad \langle(\sigma)a,E'\rangle \to^{\theta} (b,\langle(\sigma)ab,E''\rangle),$$

then exist $F',F'' \in \mathrm{RES}$ such that $E'' \doteq F''$,

$$\langle\sigma,E\rangle \to^{\theta} (b,\langle(\sigma)b,F'\rangle) \quad \textit{and} \quad \langle(\sigma)b,F'\rangle \to^{\theta} (a,\langle(\sigma)ba,F''\rangle).$$

The rules we defined above describe how to perform individual steps. We now continue with executions, i.e., sequences of steps.

Definition 5.27. The *θ-reordering multiple-step reduction* is given by the relation $\to^{\theta*} \subseteq \mathbb{S} \times \mathrm{RES} \times \Sigma^* \times \mathbb{S} \times \mathrm{RES}$:

$$\frac{}{\langle \sigma, E\rangle \to^{\theta*} (\varepsilon, \langle \sigma, E\rangle)} \qquad \frac{\langle \sigma, E\rangle \to^{\theta} (a, \langle \sigma'', E''\rangle) \quad \langle \sigma'', E''\rangle \to^{\theta*} (u, \langle \sigma', E'\rangle)}{\langle \sigma, E\rangle \to^{\theta*} (au, \langle \sigma', E'\rangle)}$$

We say that a word u is an execution of program E from initial state σ if there is a derivation for $\langle \sigma, E\rangle \to^{\theta*} (u, \langle \sigma', E'\rangle)$. The multiple-step reduction relation is monotone in the following sense.

Proposition 5.28. *For any semicommutation θ, θ', $u \in \Sigma^*$, $\sigma \in \mathbb{S}$ and E, E', F, if $\theta \subseteq \theta'$, $E \preceq F$ and $\langle \sigma, E\rangle \to^{\theta*} (u, \langle (\sigma)u, E'\rangle$, then exists $F' \in \mathrm{RES}$ such that $E' \preceq F'$ and $\langle \sigma, F\rangle \to^{\theta'*} (u, \langle (\sigma)u, F'\rangle$.*

The multiple-step reduction is closed under the rewriting relation (where $\doteq$ refers only to associativity of multiplication).

Proposition 5.29. *For any semicommutation θ, $u, u' \in \Sigma^*$, $\sigma \in \mathbb{S}$ and E, E', if $u \Rightarrow^{\theta*} u'$ and $\langle \sigma, E\rangle \to^{\theta*} (u, \langle (\sigma)u, E'\rangle$, then exists $E'' \in \mathrm{RES}$ such that $E' \doteq E''$ and $\langle \sigma, E\rangle \to^{\theta*} (u', \langle (\sigma)u', E''\rangle$.*

Terminal configurations are those configurations $\langle \sigma, E\rangle$ where the execution of the program is allowed to terminate with σ as the final state.

Definition 5.30. The *nullability* (or *empty word property*) of a program is given by the function $\_\!\downarrow : \mathrm{RES} \to \mathbb{B}$ defined recursively by

$$\begin{aligned}
b\!\downarrow &=_{\mathrm{df}} \mathsf{ff} \\
0\!\downarrow &=_{\mathrm{df}} \mathsf{ff} \\
(E+F)\!\downarrow &=_{\mathrm{df}} E\!\downarrow \vee F\!\downarrow \\
1\!\downarrow &=_{\mathrm{df}} \mathsf{tt} \\
(EF)\!\downarrow &=_{\mathrm{df}} E\!\downarrow \wedge F\!\downarrow \\
(E^*)\!\downarrow &=_{\mathrm{df}} \mathsf{tt} \\
(E \parallel F)\!\downarrow &=_{\mathrm{df}} E\!\downarrow \wedge F\!\downarrow
\end{aligned}$$

Nullability is essentially a special case of θ-reorderability where we require $\theta(b, a)$ for every $a \in \Sigma$. This means that all letters in the expression would be replaced with 0. Nullability is stable under $\preceq$ and $\doteq$.

Lemma 5.31. *For any $E, F \in \mathrm{RES}$, if $E\!\downarrow$, then $E \preceq F$ implies $F\!\downarrow$, $F \preceq E$ implies $F\!\downarrow$ and $E \doteq F$ implies $F\!\downarrow$.*

As a remark, if we would drop the $\mathrm{R}_a^\theta E$ terms in the rules in Definition 5.19 and add $E\!\downarrow$ as a side condition to the second rule of multiplication, then we would obtain the usual interleaving semantics which does not allow any reorderings. This is essentially the same as taking $\theta = \emptyset$.

Definition 5.32. A configuration $\langle \sigma, E\rangle$ is *terminal* when $E\!\downarrow$.

We say that an execution u of program E from initial state σ is terminal when there is a derivation along u that ends with a terminal configuration. From Propositions 5.28, 5.29 by Lemma 5.31, we have as corollaries that terminal executions are also monotone in θ and closed under rewriting. Terminal executions of a program correspond to the word language interpretation of the program in the following sense.

Proposition 5.33. *For any semicommutation θ, $E \in \mathrm{RES}$ and $\sigma \in \mathbb{S}$,*

1. *for any $a \in \Sigma$, $v \in \Sigma^*$,*

$$av \in [\![E]\!]^\theta \wedge (\sigma)a{\downarrow} \iff \exists E'.\, \langle \sigma, E\rangle \to^\theta (a, \langle (\sigma)a, E'\rangle) \wedge v \in [\![E']\!]^\theta;$$

2. *for any $u, v \in \Sigma^*$,*

$$uv \in [\![E]\!]^\theta \wedge (\sigma)u{\downarrow} \iff \exists E'.\, \langle \sigma, E\rangle \to^{\theta *} (u, \langle (\sigma)u, E'\rangle) \wedge v \in [\![E']\!]^\theta;$$

3. *for any $u \in \Sigma^*$,*

$$u \in [\![E]\!]^\theta \wedge (\sigma)u{\downarrow} \iff \exists E'.\, \langle \sigma, E\rangle \to^{\theta *} (u, \langle (\sigma)u, E'\rangle) \wedge E' \not\downarrow .$$

Proof.

1. $\Longrightarrow$: By induction on E. We show two cases.

   - Case EF: Here $av \in [\![E]\!]^\theta \cdot^\theta [\![F]\!]^\theta$. Hence there exist $x \in [\![E]\!]^\theta$ and $y \in [\![F]\!]^\theta$ such that $av \in x \cdot^\theta y$.
     If $x = ax'$, then by i.h. we have $\langle \sigma, E\rangle \to^\theta (a, \langle (\sigma)a, E'\rangle)$ and $x' \in [\![E']\!]^\theta$. Thus $\langle \sigma, EF\rangle \to^\theta (a, \langle (\sigma)a, E'F\rangle)$, and, since $v \in x' \cdot^\theta y$, we have $v \in [\![E']\!]^\theta \cdot^\theta [\![F]\!]^\theta = [\![E'F]\!]^\theta$.
     Otherwise $y = ay'$ and $\theta(x, a)$. By i.h. we have $\langle \sigma, F\rangle \to^\theta (a, \langle (\sigma)a, F'\rangle)$ and $y' \in [\![F']\!]^\theta$. Thus $\langle \sigma, EF\rangle \to^\theta (a, \langle (\sigma)a, (\mathrm{R}_a^\theta E)F'\rangle)$. We have $v \in x \cdot^\theta y'$. Since $x \in [\![E]\!]^\theta$ and $\theta(x, a)$, we also have $x \in [\![\mathrm{R}_a^\theta E]\!]^\theta$. Thus $v \in [\![\mathrm{R}_a^\theta E]\!]^\theta \cdot^\theta [\![F']\!]^\theta = [\![(\mathrm{R}_a^\theta E)F']\!]^\theta$.
   - Case $E \parallel F$: Here $av \in [\![E]\!]^\theta \sqcup\!\sqcup [\![F]\!]^\theta$. Hence there exist $x \in [\![E]\!]^\theta$ and $y \in [\![F]\!]^\theta$ such that $av \in x \sqcup\!\sqcup y$.
     If $x = ax'$, then by i.h. we have $\langle \sigma, E\rangle \to^\theta (a, \langle (\sigma)a, E'\rangle)$ and $x' \in [\![E']\!]^\theta$. Thus $\langle \sigma, E \parallel F\rangle \to^\theta (a, \langle (\sigma)a, E' \parallel F\rangle)$. As $v \in x' \sqcup\!\sqcup y$, we have $v \in [\![E']\!]^\theta \sqcup\!\sqcup [\![F]\!]^\theta = [\![E' \parallel F]\!]^\theta$.
     The other case is symmetric.

1. $\Longleftarrow$: By induction on the derivation $\langle \sigma, E\rangle \to^\theta (a, \langle (\sigma)a, E'\rangle)$. We show two cases.

   - Case $\langle \sigma, EF\rangle \to^\theta (a, \langle (\sigma)a, (\mathrm{R}_a^\theta E)F'\rangle$ inferred from $\langle \sigma, F\rangle \to^\theta (a, \langle (\sigma)a, F'\rangle$: We have $v \in [\![(\mathrm{R}_a^\theta E)F']\!]^\theta$ and thus there exist $x \in [\![\mathrm{R}_a^\theta E]\!]^\theta$ and $y \in [\![F']\!]^\theta$ such that $v \in x \cdot^\theta y$. By i.h. we have $ay \in [\![F]\!]^\theta$ and $(\sigma)a{\downarrow}$. Since $x \in [\![\mathrm{R}_a^\theta E]\!]^\theta$, we have that $\theta(x, a)$. Thus $av \in x \cdot^\theta ay \subseteq [\![\mathrm{R}_a^\theta E]\!]^\theta \cdot^\theta [\![F]\!]^\theta \subseteq [\![E]\!]^\theta \cdot^\theta [\![F]\!]^\theta = [\![EF]\!]^\theta$.
   - Case $\langle \sigma, E \parallel F\rangle \to^\theta (a, \langle (\sigma)a, E' \parallel F\rangle$ inferred from $\langle \sigma, E\rangle \to^\theta (a, \langle (\sigma)a, E'\rangle$: We have $v \in [\![E' \parallel F]\!]^\theta$ and thus there exist $x \in [\![E']\!]^\theta$ and $y \in [\![F]\!]^\theta$ such that $v \in x \sqcup\!\sqcup y$. By i.h. we have $ax \in [\![E]\!]^\theta$ and $(\sigma)a{\downarrow}$. Thus we have $av \in ax \sqcup\!\sqcup y \subseteq [\![E]\!]^\theta \sqcup\!\sqcup [\![F]\!]^\theta = [\![E \parallel F]\!]^\theta$.

2. By induction on u (utilising (1) in the step case).

3. Follows from (2) for u and ε. □

Finally, we define the semantic function which, given a program and an initial state, gives all final states for this program reachable from the given initial state. These are the final states of all terminal executions.

Definition 5.34. The *semantic function* $\mathscr{S}_\theta[\![\_]\!] : \mathsf{RES} \to \mathbb{S} \to \mathscr{P}(\mathbb{S})$ is given by

$$\mathscr{S}_\theta[\![E]\!]\sigma =_{\mathrm{df}} \{\sigma' \mid \langle \sigma, E\rangle \to^{\theta *} (u, \langle \sigma', E'\rangle) \wedge E' \not\downarrow\}.$$

The semantic function is also monotone in the semicommutation relation. In the special case where we exclude parallel composition we get a stronger result.

Proposition 5.35. *For any semicommutations θ, θ' and $\sigma \in \mathbb{S}$, if $\theta \subseteq \theta'$, then we have:*

1. *for any $E \in \mathsf{RES}$, $\mathscr{S}_\theta[\![E]\!]\sigma \subseteq \mathscr{S}_{\theta'}[\![E]\!]\sigma$;*
2. *for any $E \in \mathsf{RE}$, if θ' is conservative, then $\mathscr{S}_\theta[\![E]\!]\sigma = \mathscr{S}_{\theta'}[\![E]\!]\sigma$.*

Proof.

1. Follows from Proposition 5.28 and Lemma 5.31.
2. We show the inclusion $\mathscr{S}_\theta[\![E]\!]\sigma \supseteq \mathscr{S}_{\theta'}[\![E]\!]\sigma$. By Proposition 5.33.(3) we know that, for each $\sigma' \in \mathscr{S}_{\theta'}[\![E]\!]\sigma$, its corresponding execution u' is in $[\![E]\!]^{\theta'}$ and $\sigma' = (\sigma)u'\downarrow$. By Proposition 5.10 we have $\theta'([\![E]\!]^\theta) = [\![E]\!]^{\theta'}$. Thus exists $u \in [\![E]\!]^\theta$ such that $u \Rightarrow^{\theta' *} u'$. Since θ' is conservative, $(\sigma)u = (\sigma)u' = \sigma'$ and thus $(\sigma)u\downarrow$. By Proposition 5.33.(3) there exists E'' such that $\langle \sigma, E\rangle \to^{\theta *} (u, \langle (\sigma)u, E''\rangle)$ with $E'' \not\downarrow$. Hence $\sigma' = (\sigma)u \in \mathscr{S}_\theta[\![E]\!]\sigma$. □

In our interpretation θ represents a stronger memory model and θ' represents a weaker memory model. The first item in the above proposition tells us that all final states that we can observe on the stronger model we can also observe on the weaker model. The second item tells us that if the weaker model is conservative, then the stronger and the weaker model are indistinguishable for sequential programs. This is similar to saying that the weaker model is individually sequentially consistent.

5.3.4 Parallel-Independent Programs

In Proposition 5.35 we saw that for regular expressions (without shuffle) the semantic function is the same for θ and θ' when $\theta \subseteq \theta'$ and θ' is conservative. In other words, when we exclude parallel composition, then all executions obtained with a weaker semicommutation relation are explainable as executions obtained with a stronger semicommutation relation. Importantly, this property does not hold when we include parallel composition. We now illustrate this by considering a special case of parallel programs for which this property does hold.

Intuitively, two instructions a and b (or instruction instances) are parallel in a program when they come from the different threads of some parallel composition $E \parallel F$. For this discussion, we assume that there is some mechanism which, for a given expression E, describes the parallel pairs of instructions. For example, this could be a (symmetric) relation on the alphabet which contains all instruction pairs which occur in parallel in this expression E.

We say here that two instructions a and b are *independent* when they commute in both directions, i.e., both $\theta(a,b)$ and $\theta(b,a)$ hold. We say that a program E is *parallel-independent* for θ when the parallel pairs of instructions in E are independent. The word language interpretation of parallel-independent programs satisfies the following property.

Lemma 5.36. *For any semicommutation θ and $E, F \in \mathsf{RES}$, if $E \parallel F$ is parallel-independent, then $[\![E]\!]^\theta \sqcup [\![F]\!]^\theta = [\![E]\!]^\theta \cdot^\theta [\![F]\!]^\theta$.*

This then guarantees that Proposition 5.10 holds for any expression that is parallel-independent.

Proposition 5.37. *For any semicommutations θ and θ', and $E \in \mathsf{RES}$, if $\theta \subseteq \theta'$ and E is parallel-independent for θ', then $\theta'(\llbracket E \rrbracket^{\theta}) = \llbracket E \rrbracket^{\theta'}$.*

This in turn guarantees that Proposition 5.35 holds for any expression that is parallel-independent. In other words, we have the following proposition.

Proposition 5.38. *For any semicommutations θ and θ', $\sigma \in \mathbb{S}$ and $E \in \mathsf{RES}$, if $\theta \subseteq \theta'$, θ' is conservative and E is parallel-independent for θ', then $\mathscr{S}_{\theta} \llbracket E \rrbracket \sigma = \mathscr{S}_{\theta'} \llbracket E \rrbracket \sigma$.*

This can be interpreted as saying that, for a parallel-independent program, the executions of the weaker memory model are explainable as executions of the stronger model.

On the other hand, parallel-independent programs might not be very interesting in the sense that the parallel threads in such a program do not communicate in a meaningful way with each other. If they would, then this should mean that some parallel instructions are not independent (for a conservative θ).

Even when a program E is not parallel-independent for θ, we can consider a slight variation that is. First, we assume we can label the instructions in E in such manner that we obtain a program $E^{\|}$ such that the parallel instructions relation for $E^{\|}$ is irreflexive. Then we construct from θ a semicommutation relation $\theta^{\|}$ such that $E^{\|}$ is parallel-independent for $\theta^{\|}$.

Definition 5.39. The parallel extension of θ, denoted by $\theta^{\|}$, is defined as

$$\theta^{\|}(a,b) =_{\mathrm{df}} \begin{cases} \mathrm{tt} & \text{if } a \text{ and } b \text{ are parallel,} \\ \theta(a,b) & \text{otherwise.} \end{cases}$$

Setting parallel instructions to be independent makes $\theta^{\|}$ more permissive than θ and, in general, $\theta^{\|}$ is not conservative even when θ is. As an alternative to labelling the instructions in E to obtain $E^{\|}$, we could also consider including a similar labelling mechanism into the rules of the semantics.

5.4 Example: While Language

We now take a closer look at the example from the Introduction. We do so by instantiating the framework we have defined by describing an alphabet of instructions and a semicommutation θ so that we arrive at an operational semantics for a While-like language which also allows the final state in question.

We assume to have variables **Var** (ranged over by x), states $\mathbb{S} = \mathbf{Var} \to \mathbb{Z}$ (ranged over by σ), arithmetic expressions **AExp** (ranged over by a) and Boolean expressions **BExp** (ranged over by b). For arithmetic and Boolean expressions we also assume the corresponding evaluation functions $\llbracket \_ \rrbracket_{\mathbf{AExp}} : \mathbf{AExp} \to \mathbb{S} \to \mathbb{Z}$ and $\llbracket \_ \rrbracket_{\mathbf{BExp}} : \mathbf{BExp} \to \mathbb{S} \to \mathbb{B}$. By $\sigma[x \mapsto v]$ we mean the state σ with the value of variable x updated to v. We also assume a function *vars* which computes the set of variables that occur in an expression.

The instructions we consider are assignments, assertions and fences. (Fences are instructions meant to forbid the reordering of certain pairs of instructions where one of the instructions is before the fence and the other is after the fence in the program.)

$$\mathbf{Instr} ::= x := a \mid b? \mid \texttt{fence}$$

The semantics of instructions is given by the function $[\![\_]\!] : \mathbf{Instr} \to \mathbb{S} \rightharpoonup \mathbb{S}$.

$$\begin{array}{rcl} [\![x := a]\!]\,\sigma & = & \sigma[x \mapsto [\![a]\!]_{\mathbf{AExp}}\,\sigma] \\ [\![b?]\!]\,\sigma & = & \begin{cases} \sigma & \text{if } [\![b]\!]_{\mathbf{BExp}}\,\sigma \\ \bot & \text{otherwise} \end{cases} \\ [\![\texttt{fence}]\!]\,\sigma & = & \sigma \end{array}$$

The semicommutation relation θ is defined as follows.

$$\begin{array}{rcl} \theta(x := a, x' := a') & = & \{x\} \cap (\{x'\} \cup vars(a')) = \emptyset \wedge \\ & & \{x'\} \cap (\{x\} \cup vars(a)) = \emptyset \\ \theta(x := a, b?) & = & \{x\} \cap vars(b) = \emptyset \\ \theta(\_,\_) & = & \mathsf{ff} \end{array}$$

Hence two assignments can be reordered if they satisfy the *concurrent-read-exclusive-write* property (neither of the instructions writes to a variable that is read or written by the other instruction). This condition is symmetric. A test can be reordered before an assignment when the assignment does not write to a variable that is read by the test. This case is not symmetric: an assignment can never be reordered before a test. All other pairs of instructions cannot be reordered. This means that nothing can be reordered with the fence operation and that two tests cannot be reordered either.

With the alphabet of instructions and semicommutation just defined and the operational rules defined before, we can consider again the example from the Introduction. (In the example, we write ; for $\cdot$, `skip` for 1 and `fail` for 0.)

$$x := 41; y := 1 \parallel r_1 := y; r_2 := x$$

The question in the introduction was: is it possible for this program, starting from the initial state σ where each variable is initialised to 0, to end in a state where $r_1 = 1$ and $r_2 = 0$ (assuming that all of the variables are distinct)?

We first observe that both $\theta(x := 41, y := 1)$ and $\theta(r_1 := y, r_2 := x)$ hold. Either of these is sufficient for allowing the final state in question. The derivation for the case utilising $\theta(r_1 := y, r_2 := x)$ is the following (we have omitted the labels here).

$$\begin{array}{lll} \langle \sigma, & x := 41; y := 1 \parallel r_1 := y; r_2 := x \rangle & \to^{\theta} \\ \langle \sigma[r_2 \mapsto 0], & x := 41; y := 1 \parallel r_1 := y; \texttt{skip} \rangle & \to^{\theta} \\ \langle \sigma[r_2 \mapsto 0][x \mapsto 41], & \texttt{skip}; y := 1 \parallel r_1 := y; \texttt{skip} \rangle & \to^{\theta} \\ \langle \sigma[r_2 \mapsto 0][x \mapsto 41][y \mapsto 1], & \texttt{skip}; \texttt{skip} \parallel r_1 := y; \texttt{skip} \rangle & \to^{\theta} \\ \langle \sigma[r_2 \mapsto 0][x \mapsto 41][y \mapsto 1][r_1 \mapsto 1], & \texttt{skip}; \texttt{skip} \parallel \texttt{skip}; \texttt{skip} \rangle & \end{array}$$

In the first step of the derivation we make use of the fact that $\theta(r_1 := y, r_2 := x)$ and thus have $\mathsf{R}^{\theta}_{r_2:=x}(r_1 := y) = r_1 := y$. The rest of the derivation did not use any (non-trivial) reorderings. Since the program $\texttt{skip}; \texttt{skip} \parallel \texttt{skip}; \texttt{skip}$ is nullable, we have reached a terminal configuration.

The same program with two fences inserted does not allow the final state in question. If we would try the same derivation as before with the fenced program, we would get the following derivation. Note that we have $\neg\theta(r_1 := y, \texttt{fence})$ and $\neg\theta(\texttt{fence}, r_2 := x)$.

$$\begin{array}{lll} \langle \sigma, & x := 41; \texttt{fence}; y := 1 \parallel r_1 := y; \texttt{fence}; r_2 := x \rangle & \to^{\theta} \\ \langle \sigma[r_2 \mapsto 0], & x := 41; \texttt{fence}; y := 1 \parallel r_1 := y; \texttt{fail}; \texttt{skip} \rangle & \end{array}$$

The fence instruction in the second thread became `fail` since the reordering is not allowed, i.e., $\mathsf{R}^{\theta}_{r_2:=x}(\texttt{fence}) = \texttt{fail}$. The resulting configuration does not lead to a terminal configuration. Similarly, if we would try to execute the fence instruction early, then we would have the following derivation.

$$\begin{array}{lll} \langle \sigma, & x := 41; \texttt{fence}; y := 1 \parallel r_1 := y; \texttt{fence}; r_2 := x \rangle & \rightarrow^{\theta} \\ \langle \sigma, & x := 41; \texttt{fence}; y := 1 \parallel \texttt{fail}; \texttt{skip}; r_2 := x \rangle & \end{array}$$

Since $\mathsf{R}^{\theta}_{\texttt{fence}}(r_1 := y) = \texttt{fail}$, executing the fence early leads again to a configuration from which no terminal configuration is reachable.

We would get a similar result if we would take the original program (without fences) and take θ to be the empty set. This means that we are essentially considering the usual interleaving semantics.

$$\begin{array}{lll} \langle \sigma, & x := 41; y := 1 \parallel r_1 := y; r_2 := x \rangle & \rightarrow^{\emptyset} \\ \langle \sigma[r_2 \mapsto 0], & x := 41; y := 1 \parallel \texttt{fail}; \texttt{skip} \rangle & \end{array}$$

Since no reorderings are allowed in this example, the instruction $r_1 := y$ became `fail`, i.e., $\mathsf{R}^{\emptyset}_{r_2:=x}(r_1 := y) = \texttt{fail}$ and, again, the resulting configuration does not lead to a terminal configuration.

For a slightly different example, we briefly take a look at a program that also includes a test. This demonstrates that, although an assignment can never be reordered before a test, it is possible that an assignment can (eventually) be reordered with a preceding assignment even when there is a test between them. The program is

$$r_1 := x; y := r_1 \parallel r_2 := y; (x = 0)?; x := 41$$

and the question is whether the final state where $r_2 = 41$ is reachable from the initial state where every variable is initialised to zero. For this to be the case, it must be that the instruction $r_2 := y$ reads the value that is written (to a different variable) by the assignment $x := 41$ that appears later in the program. A suitable derivation is the following.

$$\begin{array}{lll} \langle \sigma, & r_1 := x; y := r_1 \parallel r_2 := y; (x = 0)?; x := 41 \rangle & \rightarrow^{\theta} \\ \langle \sigma, & r_1 := x; y := r_1 \parallel r_2 := y; \texttt{skip}; x := 41 \rangle & \rightarrow^{\theta} \\ \langle \sigma[x \mapsto 41], & r_1 := x; y := r_1 \parallel r_2 := y; \texttt{skip}; \texttt{skip} \rangle & \rightarrow^{\theta} \\ \langle \sigma[x \mapsto 41][r_1 \mapsto 41], & \texttt{skip}; y := r_1 \parallel r_2 := y; \texttt{skip}; \texttt{skip} \rangle & \rightarrow^{\theta} \\ \langle \sigma[x \mapsto 41][r_1 \mapsto 41][y \mapsto 41], & \texttt{skip}; \texttt{skip} \parallel r_2 := y; \texttt{skip}; \texttt{skip} \rangle & \rightarrow^{\theta} \\ \langle \sigma[x \mapsto 41][r_1 \mapsto 41][y \mapsto 41][r_2 \mapsto 41], & \texttt{skip}; \texttt{skip} \parallel \texttt{skip}; \texttt{skip}; \texttt{skip} \rangle & \end{array}$$

If, for example, we would replace the test $(x = 0)?$ in the second thread with $(r_2 = 0)?$ then the final state would not be valid as it is not allowed to reorder $(r_2 = 0)?$ before $r_2 := y$.

5.5 Partial-Order Reduction

In the Introduction, we were interested in finding out whether a certain final state is allowed for a given program. The question basically is: how to compute the semantic function $\mathscr{S}_{\theta}[\![E]\!]\sigma$ for given θ, E and σ? We follow a very simple idea: we just enumerate all possible derivations according to the semantics and collect the final states that we find. The problem is that, even for very small programs, we can get a very large number of unique terminal executions. In this section, we describe a method to slightly alleviate this problem by considering only representative executions when we are calculating the set of final states.

The operational semantics (implicitly) defines a labelled transition system (lts) where the states are configurations $\langle \sigma, E \rangle$ and the transitions (together with the labels) are given by the relation $\rightarrow^{\theta}$. When we calculate the possible final states for a given program E and initial state σ, then we essentially just explore the lts from the configuration $\langle \sigma, E \rangle$ and collect all the states σ' from all the terminal configurations $\langle \sigma', E' \rangle$ that we reach.

It may very well be that in the lts we have paths u and u' from the initial configuration $\langle \sigma, E \rangle$ such that u and u' are "equivalent" in the sense that $u \Rightarrow^{\theta *} u'$. Thus we have $\langle \sigma, E \rangle \rightarrow^{\theta *} (u, \langle (\sigma)u, E' \rangle)$ and $\langle \sigma, E \rangle \rightarrow^{\theta *} (u', \langle (\sigma)u', E'' \rangle)$ for some E' and E''. What we describe next allows us to say that under certain conditions it is not necessary to further explore both of the configurations $\langle (\sigma)u, E' \rangle$ and $\langle (\sigma)u', E'' \rangle$.

5.5.1 Representative Executions

By Proposition 5.29 we know that the set of executions of a program is closed under the rewriting relation $\Rightarrow^{\theta *}$. In other words, if we have a derivation for $\langle \sigma, E \rangle \rightarrow^{\theta *} (u, \langle \sigma', E' \rangle)$, θ is conservative and $u \Rightarrow^{\theta *} u'$, then we also have $\langle \sigma, E \rangle \rightarrow^{\theta *} (u', \langle \sigma', E'' \rangle)$ for some E'' such that $E' \doteq E''$. Hence we have two different executions u and u' that take us to configurations with the same state component σ' and almost the same program component ($\doteq$ refers only to associativity of multiplication). As a corollary of Lemma 5.21.(2) we have that any possible execution of E' is also an execution of E''. Thus any final state reached from $\langle \sigma', E' \rangle$ can also be reached from $\langle \sigma', E'' \rangle$.

In the case where we have explored the lts from $\langle \sigma, E \rangle$ along the path (execution) u, we have the following question: is there another path u' such that $u \Rightarrow^{\theta *} u'$ thus possibly making further exploration of $\langle \sigma', E' \rangle$ redundant? To resolve this problem we use the notion of representative executions. The idea is that we will explore a path as long as it is representative, i.e., as soon as we discover that it is not a representative anymore, then further exploration is redundant.

Definition 5.40. The set of representatives (wrt. to semicommutation θ) is given by a predicate $\mathsf{N}^{\theta} : \Sigma^* \rightarrow \mathbb{B}$ which is required to satisfy the following properties:

1. $\mathsf{N}^{\theta}(uv) \Longrightarrow \mathsf{N}^{\theta}(u)$;
2. $\neg \mathsf{N}^{\theta}(u) \Longrightarrow \exists u' . u \Rightarrow^{\theta *} u' \wedge \mathsf{N}^{\theta}(u')$.

Thus we require that the set of representatives is *prefix-closed*, i.e., an execution which is not a representative cannot be extended to a representative one. We also require that an execution that is not a representative can always be rewritten to a representative one, i.e., there exists an "equivalent" execution that is a representative. Thus we have the following proposition.

Proposition 5.41. *For any semicommutation θ, representatives N^{θ}, $E, E' \in \mathsf{RES}$, $\sigma, \sigma' \in \mathbb{S}$ and $u \in \Sigma^*$, if θ is conservative, $\neg \mathsf{N}^{\theta}(u)$ and $\langle \sigma, E \rangle \rightarrow^{\theta *} (u, \langle \sigma', E' \rangle)$, then exist u' and E'' such that $u \Rightarrow^{\theta *} u'$, $\mathsf{N}^{\theta}(u')$, $\langle \sigma, E \rangle \rightarrow^{\theta *} (u', \langle \sigma', E'' \rangle)$ and $E' \doteq E''$.*

As a consequence, if we have an execution u leading to a state σ', then there is also a representative execution u' that leads to the same state. This means that, when we are exploring the lts (for a conservative θ) by walking down its tree unwinding and we discover that our current prefix is no longer representative, then we do not have to explore the subtree ahead: all final states σ'' that we would reach are also reachable by further exploring u'.

In fact, we can straightforwardly include such a mechanism into the operational semantics by redefining the multiple-step reduction relation $\rightarrow^{\theta *}$ with an additional side condition in the step case.

Definition 5.42. The *θ-reordering N^θ-representative multiple-step reduction* is given by the relation $\rightarrow_{\mathsf{N}}^{\theta *} \subseteq \mathbb{S} \times \mathsf{RES} \times \Sigma^* \times \mathbb{S} \times \mathsf{RES}$:

$$\frac{}{\langle \sigma, E \rangle \rightarrow_{\mathsf{N}}^{\theta *} (\varepsilon, \langle \sigma, E \rangle)}$$

$$\frac{\langle \sigma, E \rangle \rightarrow_{\mathsf{N}}^{\theta *} (u, \langle \sigma'', E'' \rangle) \quad \langle \sigma'', E'' \rangle \rightarrow^{\theta} (a, \langle \sigma', E' \rangle) \quad \mathsf{N}^\theta(ua)}{\langle \sigma, E \rangle \rightarrow_{\mathsf{N}}^{\theta *} (ua, \langle \sigma', E' \rangle)}$$

If we denote the semantic function defined in terms of the N^θ-representative reduction as $\mathscr{S}_{\mathsf{N},\theta}[\![E]\!]\sigma$, then we have the following property.

Proposition 5.43. *For any conservative θ, $E \in \mathsf{RES}$ and $\sigma \in \mathbb{S}$, $\mathscr{S}_\theta[\![E]\!]\sigma = \mathscr{S}_{\mathsf{N},\theta}[\![E]\!]\sigma$.*

5.5.2 Normal Forms

We now describe a possible way to implement the predicate N^θ. The theory of Mazurkiewicz traces is a special case of semicommutations and semitraces that we are considering here. We will use normal forms known from Mazurkiewicz traces to provide an implementation of N^θ. Two normal forms common in Mazurkiewicz traces are the lexicographic normal form [9] and the Foata normal form [21].

The difference between semicommutations and Mazurkiewicz traces is precisely the fact that the independence relation in Mazurkiewicz traces is required to be symmetric. Although we do not require θ to be symmetric, we can consider the relation θ^s which is the (largest) symmetric subrelation of θ. The relation θ^s is an independence relation and thus induces a (Mazurkiewicz trace) equivalence relation on executions. If θ is conservative, then so is θ^s.

The normal forms essentially describe which execution of an equivalence class to pick as a (unique) representative. Hence we take $\mathsf{N}^\theta(u)$ to mean that the execution u is the normal form in its equivalence class according to the equivalence relation induced by θ^s. Both the lexicographic and the Foata normal form are prefix-closed. Since θ^s is symmetric, every execution in an equivalence class can be rewritten by $\Rightarrow^{\theta^s *}$ to every other execution in the equivalence class and thus also to the (unique) representative. By monotonicity of the rewriting relation, the same rewriting can be done also with $\Rightarrow^{\theta *}$. Hence the requirements for representatives are satisfied by both Foata and lexicographic normal forms.

By considering the symmetric subrelation θ^s we are of course discarding some information about pairs of instructions that commute (in one direction). Thus it might be desirable to consider some other implementation of N^θ that can take advantage of this extra information and could possibly lead to better reduction.

5.6 Extending the Framework

The framework we have defined so far allows us to describe only those reorderings which are in a sense static, i.e., the reorderings are described by a binary relation on the alphabet of instructions and that is it. We now look at a couple of extensions of the framework to consider more fine-grained reorderings.

The first extension we consider accounts for the possibility that a pair of instructions commute in some machine states but not necessarily in all of them. An example of this are memory reads and writes whose addresses are not statically determined. If we take $[r_1] := 1$ to denote a write to a memory location whose address is stored in register (local variable) r_1, then $[r_1] := 1$ and $[r_2] := 2$ could be considered independent in states where r_1 and r_2 contain different addresses.

Another extension is for the possibility that reordering two instructions has an effect on those instructions. Intuitively, in the program $y := 2; x := y$ we should not reorder the instructions $y := 2$ and $x := y$ as the second instruction reads the variable that is written to by the first instruction and this is a form of data-dependency. In a sequential setting it is valid to say that the programs $y := 2; x := y$ and $x := 2; y := 2$ give the same result. What we will say is that when we reorder $x := y$ before $y := 2$, then the instruction $y := 2$ acts on the instruction $x := y$ (from the left) so that it becomes $x := 2$ and thus the result of the reordering is $x := 2; y := 2$. The (right) action of $x := y$ on $y := 2$ is trivial here.

This kind of reordering allows two threads to have different views of the memory as a read instruction can read its value from its left context and not from memory. Furthermore, this allows us to forbid "longer" chains of reordering. For example, we can have $\theta(a, c)$, $\theta(b, c)$ and $\neg\theta(a, {}_bc)$ where ${}_bc$ is the result of b acting on c (from the left). Thus we cannot reorder c before ab in the sequence abc although $\theta(a, c)$ and $\theta(b, c)$.

We also allow to execute instructions in multiple steps. This is described by assigning to each instruction a a residual program (a continuation) $\kappa(a)$. The idea is that, when we execute a, we replace it in the program with $\kappa(a)$. In other words, after executing a we still have to execute $\kappa(a)$.

5.6.1 Operational Semantics in Context

Before we continue with a more precise treatment of the extensions described above, we first change our setup a bit to allow the extensions mentioned above.

First, here we consider a subset of RES where parallel composition appears only at the top-level. In other words, we consider expressions

$$\mathsf{RE}^{\|} ::= E \mid \mathsf{RE}^{\|} \parallel \mathsf{RE}^{\|}$$

where E ranges over RE.

Second, we consider the interpretation of instructions to be $[\![a]\!] : \mathbb{S} \to \mathbb{S}$, i.e., the state transformers are not partial anymore. We do introduce an additional predicate $\sigma\downarrow a$ to say that applying the state transformer $[\![a]\!]$ to σ is allowed. We use this predicate to encode that some instructions actually are partial, i.e., we are not allowed to execute them in some states. This enables us, for example, to define a test to always denote the identity state transformer while considering it allowed only in those states where the condition holds. We extend this to words by $\sigma\downarrow\varepsilon =_{\mathrm{df}} \mathrm{tt}$ and $\sigma\downarrow au =_{\mathrm{df}} \sigma\downarrow a \wedge (\sigma)a\downarrow u$.

Before we used the θ-reorderable part of a program to restrict the left context of the instruction that we were executing. The idea was that all executions of the restricted left context should allow the reordering needed to execute this instruction early. The θ-reorderable part essentially discards all executions of the left context that would not allow the reordering. In this section, we take this even further and consider every execution of the left context separately. This way we can describe more precisely the reorderings that we allow as the left context is a single execution and not a program (which would correspond to a set of executions).

To pick an execution of $E \in \mathsf{RE}$ (considered as a left context), we just pick a word e from the word language interpretation of E. In the rules we now have to keep track of the current context. This is essentially an execution that takes us to the position in the original program where the subprogram we are currently considering is located.

Where we before (in Definition 5.19) had $\mathsf{R}^{\theta}_{a}E$ in the rules for sequential composition and Kleene star, we now pick an execution e of E and extend the context with e for the inductive step.

Definition 5.44. The *θ-reordering single-step reduction* of sequential programs is given by the relation $\rightsquigarrow^{\theta} \subseteq \mathbb{S} \times \Sigma^* \times \mathsf{RE} \times \Sigma \times \mathbb{S} \times \Sigma^* \times \mathsf{RE}$:

$$\frac{\theta(t,a) \quad \sigma \downarrow a}{\langle \sigma, t, a \rangle \rightsquigarrow^{\theta} (a, \langle (\sigma)a, t, 1 \rangle)}$$

$$\frac{\langle \sigma, t, E \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', t', E' \rangle)}{\langle \sigma, t, E + F \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', t', E' \rangle)} \quad \frac{\langle \sigma, t, F \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', t', F' \rangle)}{\langle \sigma, t, E + F \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', t', F' \rangle)}$$

$$\frac{\langle \sigma, t, E \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', t', E' \rangle)}{\langle \sigma, t, EF \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', t', E'F \rangle)} \quad \frac{e \in [\![E]\!] \quad \langle \sigma, te, F \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', t', F' \rangle)}{\langle \sigma, t, EF \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', t', F' \rangle)}$$

$$\frac{e \in [\![E^*]\!] \quad \langle \sigma, te, E \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', t', E' \rangle)}{\langle \sigma, t, E^* \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', t', E'E^* \rangle)}$$

With $\langle \sigma, \varepsilon, E \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', t', E' \rangle)$ we have that t' is an execution of the left context of a in the program E. Furthermore, E' is the right context of a.

As another remark, here we can see the reason why we have restricted ourselves to programs as elements of $\mathsf{RE}^{\|}$. In a sense, we would like to keep all reordering of instructions to occur in the base case. In the second rule of multiplication, we pick an execution $e \in [\![E]\!]$ and this e could later be executed also in a reordered fashion. If E would contain a parallel composition, then we could miss some possible reorderings because, as we have seen in Example 5.9, there is a difference between first shuffling and then reordering and the other way around.

Definition 5.45. The *θ-reordering single-step reduction* of parallel programs is given by the relation $\rightsquigarrow^{\theta} \subseteq \mathbb{S} \times \mathsf{RE}^{\|} \times \Sigma \times \mathbb{S} \times \mathsf{RE}^{\|}$:

$$\frac{\langle \sigma, \varepsilon, E \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', t', E' \rangle)}{\langle \sigma, E \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', t'E' \rangle)}$$

$$\frac{\langle \sigma, E \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', E' \rangle)}{\langle \sigma, E \parallel F \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', E' \parallel F \rangle)} \quad \frac{\langle \sigma, F \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', F' \rangle)}{\langle \sigma, E \parallel F \rangle \rightsquigarrow^{\theta} (a, \langle \sigma', E \parallel F' \rangle)}$$

The rules for $\rightsquigarrow^{\theta}$ together allow us to delay the reorderability check until we reach the base case (a single instruction) as opposed to the rules in Definition 5.19, where this was treated "on the fly" with $\mathsf{R}^{\theta}_{a}E$ in the rules for sequential composition and the Kleene star. Here in the base case we have already determined both the instruction a we wish to execute next and the particular execution t leading to it (its left context). The extensions that we consider next just modify what happens in the base case.

Note that the base case only requires $\sigma \downarrow a$ and nothing is said about $\sigma \downarrow ta$, i.e., we only care about whether we are allowed to execute the instruction a and whether it can be reordered with its left context t. The justification for this is that, although we might have $\neg(\sigma \downarrow t)$, executing t could be allowed in some future state σ' (for example, when instructions executed by other threads have changed the state so that t becomes allowed).

Definition 5.46. The *θ-reordering multiple-step reduction* of parallel programs is given by the relation $\rightsquigarrow^{\theta *} \subseteq \mathbb{S} \times \mathsf{RE}^{\|} \times \Sigma^* \times \mathbb{S} \times \mathsf{RE}^{\|}$:

$$\frac{}{\langle \sigma, E \rangle \rightsquigarrow^{\theta *} (\varepsilon, \langle \sigma, E \rangle)} \quad \frac{\langle \sigma, E \rangle \rightsquigarrow^{\theta} (a, \langle \sigma'', E'' \rangle) \quad \langle \sigma'', E'' \rangle \rightsquigarrow^{\theta *} (u, \langle \sigma', E' \rangle)}{\langle \sigma, E \rangle \rightsquigarrow^{\theta *} (au, \langle \sigma', E' \rangle)}$$

It can be shown that there is a correspondence between the old and the new semantics. When we continue with the extensions, then this will not be the case in general.

Proposition 5.47. *The semantics given in Definition 5.27 agrees with the semantics given in Definition 5.46 in the sense that for every semicommutation θ and $\sigma, \sigma' \in \mathbb{S}$, $E \in \mathsf{RE}^{\|}$ and $u \in \Sigma^*$,*

$$\exists E'.\langle \sigma, E\rangle \to^{\theta *} (u, \langle \sigma', E'\rangle) \wedge E' \not\downarrow \iff \exists E''.\langle \sigma, E\rangle \rightsquigarrow^{\theta *} (u, \langle \sigma', E''\rangle) \wedge E'' \not\downarrow .$$

5.6.2 Context-Dependent Semicommutation Relation

With this extension we do not consider the semicommutation relation to be a static relation anymore. For each state σ, there may now be a separate semicommutation relation.

Definition 5.48. A *context-dependent semicommutation* θ is a family of irreflexive relations, i.e., a mapping $\mathbb{S} \to \mathscr{P}(\Sigma \times \Sigma)$.

With this modification we also need to modify the conservativity condition to match the context-dependent semicommutation.

Definition 5.49. A context-dependent semicommutation relation θ is *conservative* when, for every $\sigma \in \mathbb{S}$ and $a, b \in \Sigma$, we have

$$\theta_\sigma(a,b) \Longrightarrow (\sigma \downarrow ab \iff \sigma \downarrow ba) \wedge (\sigma)ab = (\sigma)ba.$$

We extend the context-dependent semicommutation on letters to words and letters.

Definition 5.50. A context-dependent semicommutation relation $\theta : \mathbb{S} \to \mathscr{P}(\Sigma \times \Sigma)$ is extended to $\theta : \mathbb{S} \to \mathscr{P}(\Sigma^* \times \Sigma)$ in the following way.

$$\begin{array}{lll} \theta_\sigma(\varepsilon, a) & =_{\mathrm{df}} & \mathsf{tt} \\ \theta_\sigma(tb, a) & =_{\mathrm{df}} & \theta_\sigma(t,a) \wedge \theta_{(\sigma)t}(b,a) \end{array}$$

Note the use of context $(\sigma)t$ in the case for $\theta_\sigma(tb,a)$. Since state transformers are total, $(\sigma)t$ is defined and we can check reorderability in the context $(\sigma)t$, even if $\neg(\sigma \downarrow t)$. For example, we might want to consider executing instructions early from either branch of a conditional $b; p + \bar{b}; q$ although exactly one of those is the correct branch in any state σ, i.e., either $\sigma \downarrow b$ or $\sigma \downarrow \bar{b}$. Until we commit to either b or $\bar{b}$, we wish to keep both possibilities.

Similarly to what we had before, we take $\theta_\sigma(t,a)$ to be the justification for reordering a before an execution t (its left context).

Lemma 5.51. *For any conservative context-dependent θ, $\sigma \in \mathbb{S}$, $t \in \Sigma^*$ and $a \in \Sigma$, we have*

$$\theta_\sigma(t,a) \Longrightarrow (\sigma \downarrow ta \iff \sigma \downarrow at) \wedge (\sigma)ta = (\sigma)at.$$

To extend the semantics with a context-dependent θ, we modify the rule for the single instruction (the base case) of Definition 5.44 to be the following.

$$\frac{\theta_\sigma(t,a) \quad \sigma \downarrow a}{\langle \sigma, t, a\rangle \to^{\theta} (a, \langle (\sigma)a, t, 1\rangle)}$$

The only modification here is that the reorderability check becomes context-dependent.

5.6.3 Reordering Actions

We now add the possibility that reordering two instructions might modify them. We formulate this as instructions acting on instructions, i.e., by reordering the instructions a and b (in the program ab) the instruction b acts on a from the right and the instruction a acts on b from the left. We include the possibility that these actions might also be context-dependent.

Definition 5.52. The left action of letter a on letter b in state σ is given by $a \overset{\sigma}{\searrow} b$ and the right action of letter b on letter a in state σ is given by $a \overset{\sigma}{\swarrow} b$.

Since two letters now act on each other when reordered, we also refine the conservativity condition on θ.

Definition 5.53. A context-dependent semicommutation θ is conservative when, for every $\sigma \in \mathbb{S}$ and $a, b \in \Sigma$, we have

$$\theta_\sigma(a,b) \Longrightarrow (\sigma \downarrow ab \iff \sigma \downarrow b'a') \wedge (\sigma)ab = (\sigma)b'a'$$

where $a' = a \overset{\sigma}{\swarrow} b$ and $b' = a \overset{\sigma}{\searrow} b$.

We extend the left action to an action of a word on a letter. Similarly, we also extend the right action to an action of a letter on a word.

Definition 5.54. The left action $\searrow : \Sigma \times \mathbb{S} \times \Sigma \to \Sigma$ is extended to $\searrow : \Sigma^* \times \mathbb{S} \times \Sigma \to \Sigma$ and the right action $\swarrow : \Sigma \times \mathbb{S} \times \Sigma \to \Sigma$ is extended to $\swarrow : \Sigma^* \times \mathbb{S} \times \Sigma \to \Sigma^*$ in the following way.

$$\begin{array}{lcl} \varepsilon \overset{\sigma}{\searrow} a & =_{\mathrm{df}} & a \\ tb \overset{\sigma}{\searrow} a & =_{\mathrm{df}} & t \overset{\sigma}{\searrow} (b \overset{(\sigma)t}{\searrow} a) \end{array}$$

$$\begin{array}{lcl} \varepsilon \overset{\sigma}{\swarrow} a & =_{\mathrm{df}} & \varepsilon \\ tb \overset{\sigma}{\swarrow} a & =_{\mathrm{df}} & (t \overset{\sigma}{\swarrow} (b \overset{(\sigma)t}{\searrow} a))(b \overset{(\sigma)t}{\swarrow} a) \end{array}$$

Ignoring the presence of state-dependence, what we have is a pair of mutual actions between the free monoids Σ^* and Σ^*. Such pairs of mutual actions between two (generally different) monoids are considered in algebra in the context of Zappa-Szép products of monoids and groups [18].

We also have to consider the reordering actions when we look at the reorderability of an execution and a letter. (We define the general case where θ is also context-dependent.)

Definition 5.55. A context-dependent semicommutation relation $\theta : \mathbb{S} \to \mathscr{P}(\Sigma \times \Sigma)$ is extended to $\theta : \mathbb{S} \to \mathscr{P}(\Sigma^* \times \Sigma)$ in the following way.

$$\begin{array}{lcl} \theta_\sigma(\varepsilon, a) & =_{\mathrm{df}} & \mathsf{tt} \\ \theta_\sigma(tb, a) & =_{\mathrm{df}} & \theta_\sigma(t, b \overset{(\sigma)t}{\searrow} a) \wedge \theta_{(\sigma)t}(b, a) \end{array}$$

Importantly, if we can reorder a before t with the context-dependent and conservative θ and the reordering actions, then the reordering is justified.

Lemma 5.56. *For any conservative θ, $\sigma \in \mathbb{S}$, $t \in \Sigma^*$ and $a \in \Sigma$, we have*

$$\theta_\sigma(t,a) \Longrightarrow (\sigma \downarrow ta \iff \sigma \downarrow a't') \wedge (\sigma)ta = (\sigma)a't'$$

where $t' = t \overset{\sigma}{\swarrow} a$ and $a' = t \overset{\sigma}{\searrow} a$.

To extend the semantics with reordering actions we modify the rule for the single instruction (the base case) of Definition 5.44 to be

$$\frac{\theta_\sigma(t,a) \quad \sigma \downarrow a'}{\langle \sigma, t, a \rangle \to^\theta (a', \langle (\sigma)a', t', 1 \rangle)}$$

where $a' = t \overset{\sigma}{\searrow} a$ and $t' = t \overset{\sigma}{\swarrow} a$.

5.6.4 Non-Atomic Instructions

Finally, we consider the possibility that some instructions might not be atomic. For example, writing an 8-byte value to memory might be implemented as two 4-byte writes. Another possibility is that the execution of an instruction proceeds in multiple steps. For example, if the instruction is $x := y + z$, then in the first step we might determine the value of the expression $y + z$, say v, and in the second step we write the value v to variable x in memory.

The mechanism we use to model this multiple-step execution is to have a function $\kappa : \Sigma \times \mathbb{S} \to \mathrm{RE}$ which, given a state σ and an instruction a, determines the "continuation" of executing the first step of a in state σ, i.e., an expression which represents the part of the instruction that we have not yet executed.

Take the example $x := y + z$ from above. We could set things up so that executing $x := y + z$ has no effect on the state, but we define its continuation to be the instruction $x := v$ where v is the value of $y + z$ in the state σ. Executing $x := v$ would then update the state accordingly.

We extend the semantics with "continuations" (and context-dependent θ and reordering actions) by modifying the rule for the single instruction of Definition 5.44 to be

$$\frac{\theta_\sigma(t,a) \quad \sigma \downarrow a'}{\langle \sigma, t, a \rangle \to^\theta (a', \langle (\sigma)a', t', \kappa(a', \sigma) \rangle)}$$

where $a' = t \overset{\sigma}{\searrow} a$ and $t' = t \overset{\sigma}{\swarrow} a$. Essentially we just plug the "continuation" of an instruction into the program where the instruction used to be. Before we implicitly used a constant function for κ that is always 1.

5.6.5 Extensions and Partial-Order Reduction

In Section 5.5, we described a form of partial-order reduction so that computing the set of final states $\mathscr{S}_\theta[\![E]\!]\sigma$ can be less expensive. The situation is more complicated when θ is a context-dependent semicommutation relation and we also have to account for the left- and right-actions and the continuation function κ.

We just mention that we could take a similar approach as we took in Section 5.5. First, we would need to modify the definition of the rewriting relation $\Rightarrow^{\theta *}$ to account for the context-dependence of θ, the reordering actions $\searrow$ and $\swarrow$, and the continuation function κ. This way we would relate executions u and u' in state σ, denoted by $u \Rightarrow_\sigma^{\theta *} u'$, if u can be rewritten to u' in state σ according to the context-dependent θ, reordering actions $\searrow$ and $\swarrow$, and the continuation function κ. We could then give the axioms for the set of representatives similarly to what we did in Definition 5.40. Namely, we would require that the set of representatives in state σ is prefix-closed (corresponding to 5.40.1) and any execution u that is not a representative can be rewritten (in state σ) to a representative u' (corresponding to 5.40.2), i.e., $u \Rightarrow_\sigma^{\theta *} u'$. If the conservativity conditions are satisfied, then u and u' lead to the same final state.

5.6.6 Context-Dependence of θ and Actions

It might be that the context-dependent θ and the reordering actions fit together well enough so that we do not need to modify the state (from σ to $(\sigma)t$) when we are checking $\theta_\sigma(tb,a)$ to determine reorderability of b and a. For this we require three properties.

Definition 5.57. A context-dependent semicommutation θ and the reordering actions $\searrow$ and $\swarrow$ are *stable* when for any σ, t, a, b:

1. $\theta_\sigma(t, b \overset{(\sigma)t}{\searrow} a) \Longrightarrow \theta_\sigma(b,a) = \theta_{(\sigma)t}(b,a)$;
2. $\theta_\sigma(t, b \overset{(\sigma)t}{\searrow} a) \vee \theta_\sigma(t, b \overset{\sigma}{\searrow} a) \Longrightarrow b \overset{\sigma}{\searrow} a = b \overset{(\sigma)t}{\searrow} a$;
3. $\theta_\sigma(t, b \overset{(\sigma)t}{\searrow} a) \Longrightarrow b \overset{\sigma}{\swarrow} a = b \overset{(\sigma)t}{\swarrow} a$.

What this essentially requires is that, in the sequence tba, if after reordering a before b, the result can be reordered before t, then t does not change the part of the state σ which determines the commutativity of a and b and how they act on each other.

We now define a different way to extend a context-dependent semicommutation to words and letters. The difference is that we do not change the context σ to $(\sigma)t$ in the step case.

Definition 5.58. We extend a context-dependent semicommutation θ to words and letters in the following way.

$$\begin{aligned} \hat{\theta}_\sigma(\varepsilon, a) &=_{\mathrm{df}} \mathsf{tt} \\ \hat{\theta}_\sigma(tb, a) &=_{\mathrm{df}} \hat{\theta}_\sigma(t, b \overset{\sigma}{\searrow} a) \wedge \theta_\sigma(b,a) \end{aligned}$$

We also define a different way to extend the reordering actions to words and letters. Again, the difference is that we do not change the context.

Definition 5.59. We extend the left and right actions to words and letters in the following way.

$$\begin{aligned} \varepsilon \overset{\sigma}{\curvearrowright} a &=_{\mathrm{df}} a \\ tb \overset{\sigma}{\curvearrowright} a &=_{\mathrm{df}} t \overset{\sigma}{\curvearrowright} (b \overset{\sigma}{\searrow} a) \\ \varepsilon \overset{\sigma}{\curvearrowleft} a &=_{\mathrm{df}} \varepsilon \\ tb \overset{\sigma}{\curvearrowleft} a &=_{\mathrm{df}} (t \overset{\sigma}{\curvearrowleft} (b \overset{\sigma}{\searrow} a))(b \overset{\sigma}{\swarrow} a) \end{aligned}$$

Next we show that these definitions are suitable for justification of reorderings when θ and the reordering actions are stable and θ is conservative.

Lemma 5.60. *For any semicommutation θ and left action $\searrow$, if conditions 5.57.1 and 5.57.2 are satisfied, then, for any $\sigma \in \mathbb{S}$, $t \in \Sigma^*$ and $a \in \Sigma$, we have that $\theta_\sigma(t,a)$ implies $\hat{\theta}_\sigma(t,a)$.*

Lemma 5.61. *For any semicommutation θ and left action $\searrow$, if conditions 5.57.1 and 5.57.2 are satisfied, then, for any $\sigma \in \mathbb{S}$, $t \in \Sigma^*$ and $a \in \Sigma$, we have that $\hat{\theta}_\sigma(t,a)$ implies $\theta_\sigma(t,a)$.*

Lemma 5.62. *For any semicommutation θ and left action $\searrow$, if condition 5.57.2 is satisfied, then, for any $\sigma \in \mathbb{S}$, $t \in \Sigma^*$ and $a \in \Sigma$, we have $\theta_\sigma(t,a)$ implies $t \overset{\sigma}{\searrow} a = t \overset{\sigma}{\curvearrowright} a$.*

Lemma 5.63. *For any semicommutation θ and actions $\searrow$ and $\swarrow$, if conditions 5.57.2 and 5.57.3 are satisfied, then, for any $\sigma \in \mathbb{S}$, $t \in \Sigma^*$ and $a \in \Sigma$, we have $\theta_\sigma(t,a)$ implies $t \overset{\sigma}{\swarrow} a = t \overset{\sigma}{\frown} a$.*

Proposition 5.64. *For any semicommutation θ and actions $\searrow$ and $\swarrow$, if conditions 5.57.1, 5.57.2 and 5.57.3 are satisfied and θ is conservative, then, for any $\sigma \in \mathbb{S}$, $t \in \Sigma^*$ and $a \in \Sigma$, we have $\hat{\theta}_\sigma(t,a)$ implies $(\sigma)ta = (\sigma)(t \overset{\sigma}{\frown} a)(t \overset{\sigma}{\frown} a)$.*

We have seen that, if the semicommutation and the reordering actions satisfy the stability conditions, then $\theta_\sigma(t,a)$ and $\hat{\theta}_\sigma(t,a)$ are equivalent. Furthermore, we have a conservativity result for the reordering actions that do not modify the context. This ensures that, when we are checking reorderability, we can keep the state σ fixed.

5.7 Example: TSO-like Memory Model

We now expand on the context-dependent independence relation described in Section 4.5 by giving an operational semantics for a TSO-like machine in the sense that we model write buffers in the system. The machine consists of shared memory and a fixed number of processors, each with its own local memory. As in Section 5.4, we have variables **Var** (ranged over by x; shared memory), arithmetic expressions **AExp** (ranged over by a, only mentioning the local memory of a processor) and Boolean expressions **BExp** (ranged over by b, only mentioning the local memory of a processor). The local memory of a processor is represented as registers **Reg** (ranged over by r).

In this example, we do not consider the state set $\mathbb{S}$ to be a mapping from variables to values. Instead we take the set of machine states $\mathbb{S}$ to be Σ^*, i.e., we consider the execution itself as the state. Thus we also define the functions $mval : \textbf{Var} \to \mathbb{S} \to \mathbb{Z}$ and $rval : \textbf{Proc} \to \textbf{Reg} \to \mathbb{S} \to \mathbb{Z}$ to *view* the current state. We assume to have evaluation functions $[\![\_]\!]_{\textbf{AExp}} : \textbf{AExp} \to \textbf{Proc} \to \mathbb{S} \to \mathbb{Z}$ and $[\![\_]\!]_{\textbf{BExp}} : \textbf{BExp} \to \textbf{Proc} \to \mathbb{S} \to \mathbb{B}$ that are defined in terms of these view functions.

The alphabet consists of reads, writes, tests and fences. We also encode processor identifiers and program order into the alphabet. More precisely, we take $\Sigma =_{\mathrm{df}} \textbf{Proc} \times \mathbb{N} \times \Sigma'$ where in Σ' we have:

- $R(r,x,v)$: a read instruction which reads the value v from variable x and stores the result in local register r. Before execution, the value v is undefined and in program text we just write $r := [x]$.
- $W(x,a)$: a write initiate (buffer enqueue) operation which represents a write of the value of expression a to variable x. In program text, we write $[x] := a$.
- $C(x,v)$: a write commit (buffer dequeue) operation which writes the value v to variable x. In program text, we write $x \mapsto v$.
- $T(b)$: a test of Boolean expression b. In program text, we write $b?$.
- F: a fence instruction. In program text, we write `fence`.

We take the ordering relation $\prec$ on the alphabet Σ to be the lexicographic order on Σ given by $\prec_{\textbf{Proc}}$, $\prec_{\mathbb{N}}$ and $\prec_{\Sigma'}$ where $R \prec_{\Sigma'} W \prec_{\Sigma'} C \prec_{\Sigma'} T \prec_{\Sigma'} F$.

Next, we define the view functions on $\sigma \in \mathbb{S} = \Sigma^*$. The function $mval$: **Var** $\to \mathbb{S} \to \mathbb{Z}$ specifies the current values in shared memory.

$$\begin{array}{lll} mval(x, \varepsilon) & =_{\mathrm{df}} & 0 \\ mval(x, \sigma \cdot (p, i, C(x', v'))) & =_{\mathrm{df}} & \begin{cases} v' & \text{if } x = x' \\ mval(x, \sigma) & \text{otherwise} \end{cases} \\ mval(x, \sigma \cdot \alpha) & =_{\mathrm{df}} & mval(x, \sigma) \end{array}$$

The function $rval$ specifies the current values in the registers of a processor.

$$\begin{array}{lll} rval(p, r, \varepsilon) & =_{\mathrm{df}} & 0 \\ rval(p, r, \sigma \cdot (p', i', R(r', x', v'))) & =_{\mathrm{df}} & \begin{cases} v' & \text{if } p = p' \wedge r = r' \\ rval(p, r, \sigma) & \text{otherwise} \end{cases} \\ rval(p, r, \sigma \cdot \alpha) & =_{\mathrm{df}} & rval(p, r, \sigma) \end{array}$$

We say that a processor p has a pending write to variable x when the last write instruction to x is $W(x, a)$ and not $C(x, v)$.

$$\begin{array}{lll} pending(p, x, \varepsilon) & =_{\mathrm{df}} & \mathsf{ff} \\ pending(p, x, \sigma \cdot (p', i', W(x', a'))) & =_{\mathrm{df}} & (p = p' \wedge x = x') \vee pending(p, x, \sigma) \\ pending(p, x, \sigma \cdot (p', i', C(x', v'))) & =_{\mathrm{df}} & (p \neq p' \vee x \neq x') \wedge pending(p, x, \sigma) \\ pending(p, x, \sigma \cdot \alpha) & =_{\mathrm{df}} & pending(p, x, \sigma) \end{array}$$

We now continue with the context-dependent independence relation, reordering actions and the continuation function necessary for the operational semantics. We start with the continuation function.

$$\begin{array}{llll} \kappa((p, i, W(x, a)), & \sigma) & =_{\mathrm{df}} & (p, i, C(x, [\![a]\!]_{\mathbf{AExp}}\ p\ \sigma)) \\ \kappa(\alpha, & \sigma) & =_{\mathrm{df}} & 1 \end{array}$$

Thus we can see that, when executing an instruction $[x] := a$, we plug in its place in the program the corresponding commit instruction $x \mapsto v$. Note that C receives the same program order identifier as W. We also assume that initially a program does not contain any commit instructions.

The reordering actions are used only on instructions from the same processor. The $\swarrow$ action is identity in this example. We define the reordering action $\searrow$ as follows.

$$\begin{array}{lll} C(x, v) \overset{\sigma}{\searrow} R(r', x', \bot) & =_{\mathrm{df}} & \begin{cases} R(r', x', v) & \text{if } x = x' \\ R(r', x', \bot) & \text{otherwise} \end{cases} \\ \alpha \overset{\sigma}{\searrow} \beta & =_{\mathrm{df}} & \beta \end{array}$$

This just says that, if the value of the read instruction is not yet determined, then its value can be determined from its left context (when there is a suitable C in the context). Next, we define a semicommutation relation θ on Σ'.

$$\begin{array}{llll} \theta(C(x, v), & R(r', x', v')) & =_{\mathrm{df}} & \mathsf{tt} \\ \theta(C(x, v), & W(x', a')) & =_{\mathrm{df}} & \mathsf{tt} \\ \theta(T(b), & R(r', x', v')) & =_{\mathrm{df}} & r' \notin regs(b) \\ \theta(\alpha, & \beta) & =_{\mathrm{df}} & \mathsf{ff} \end{array}$$

The interpretation of instructions as state transformers basically just extends the state σ with the new instruction. For read instructions, we may still need to determine the

value that is read.

$$\begin{array}{rcl} [\![(p,i,R(r,x,\bot))]\!]\sigma & =_{\mathrm{df}} & \sigma \cdot (p,i,R(r,x,mval(x,\sigma))) \\ [\![\alpha]\!]\sigma & =_{\mathrm{df}} & \sigma \cdot \alpha \end{array}$$

Tests are the only state transformers that are partial, i.e., we are always allowed to execute an instruction that is not a test.

$$\begin{array}{lcl} \sigma\downarrow(p,i,T(b)) & =_{\mathrm{df}} & [\![b]\!]_{\mathbf{BExp}}\ p\ \sigma \\ \sigma\downarrow\alpha & =_{\mathrm{df}} & \mathtt{tt} \end{array}$$

With the ingredients defined above, we can now construct executions using the rules in Definition 5.46 together with the extensions.

Following the definitions given in Section 4.5, we can also give a context-dependent independence relation I on Σ that can be used for partial-order reduction. The subrelation I^s is for instructions from the same processor. To emphasise, this is not the symmetric subrelation θ^s that we described earlier. Here we construct a symmetric relation by using the program order identifiers to keep track of in which way did we check reorderability.

$$(p,i,\alpha)\quad I^s\quad (p',i',\alpha')\quad =_{\mathrm{df}}\quad p=p' \wedge (i \prec_{\mathbb{N}} i' \wedge \theta(\alpha,\alpha') \vee i' \prec_{\mathbb{N}} i \wedge \theta(\alpha',\alpha))$$

Note that this implies that instructions from the same processor with the same program order identifier are dependent. This can only be the case for corresponding pairs of W and C. The subrelation I^d is for instructions from different processors.

$$\begin{array}{llll} (p,i,C(x,v)) & I^d_\sigma & (p',i',C(x',v')) & =_{\mathrm{df}}\ p\neq p' \wedge x\neq x' \\ (p,i,C(x,v)) & I^d_\sigma & (p',i',R(r',x',v')) & =_{\mathrm{df}}\ p\neq p' \wedge (x=x' \Longrightarrow pending(p',x',\sigma)) \\ (p,i,R(r,x,v)) & I^d_\sigma & (p',i',C(x',v')) & =_{\mathrm{df}}\ p\neq p' \wedge (x=x' \Longrightarrow pending(p,x,\sigma)) \\ (p,i,\alpha) & I^d_\sigma & (p',i',\alpha') & =_{\mathrm{df}}\ p\neq p' \end{array}$$

We see that a read and a commit instruction by different processors to the same variable are independent when the read instruction gets its value from the pending write (i.e., from its left context). Finally, we take the context-dependent independence relation on Σ to be the following.

$$\alpha\, I_\sigma\, \beta =_{\mathrm{df}} \alpha\, I^s\, \beta \vee \alpha\, I^d_\sigma\, \beta$$

As an example, we look at a program similar to what we saw in Section 4.5.

$$[x] := 1; r_1 := [x] \parallel [x] := 2$$

Here the first thread first writes to location x and then reads from location x. The second thread just writes to location x. Before, in Section 4.5, we essentially argued that the executions (here we omit the program order identifier and write the processor identifier as an index)

$$W_1(x,1)\cdot W_2(x,2)\cdot C_2(x,2)\cdot R_1(r_1,x,1)\cdot C_1(x,1)$$

and

$$W_1(x,1)\cdot W_2(x,2)\cdot R_1(r_1,x,1)\cdot C_2(x,2)\cdot C_1(x,1)$$

are equivalent according to the equivalence relation induced by the context-dependent I. Indeed, the instructions $C_2(x,2)$ and $R_1(r_1,x,1)$ are independent in the context $W_1(x,1)\cdot W_2(x,2)$ since, although they are to the same location, the processor executing the read instruction also has a pending write to the same location. We can also check that the second execution is in Foata normal form as

$$(W_1(x,1)\cdot W_2(x,2))\ (R_1(r_1,x,1)\cdot C_2(x,2))\ (C_1(x,1)).$$

The first execution is obtained as follows (where we write '$\ldots$' for the state in the previous configuration).

$$\begin{array}{llcll}
\langle \varepsilon, & [x] := 1; r_1 := [x] & \| & [x] := 2\rangle & \to^I \\
\langle \ldots \cdot W_1(x,1), & x \mapsto 1; r_1 := [x] & \| & [x] := 2\rangle & \to^I \\
\langle \ldots \cdot W_2(x,2), & x \mapsto 1; r_1 := [x] & \| & x \mapsto 2\rangle & \to^I \\
\langle \ldots \cdot C_2(x,2), & x \mapsto 1; r_1 := [x] & \| & \texttt{skip}\rangle & \to^I \\
\langle \ldots \cdot R_1(r_1,x,1), & x \mapsto 1; \texttt{skip} & \| & \texttt{skip}\rangle & \to^I \\
\langle \ldots \cdot C_1(x,1), & \texttt{skip};\texttt{skip} & \| & \texttt{skip}\rangle &
\end{array}$$

Note that the value read by $r_1 := [x]$ was determined to be 1 during reordering as

$$mval(x, W_1(x,1) \cdot W_2(x,2) \cdot C_2(x,2)) = 2.$$

Finally, we show that this setup allows the relaxed behaviour in the store buffering example, i.e., that in the program

$$[x] := 1; r_1 := [y] \;\|\; [y] := 1; r_2 := [x]$$

it is possible to reach a final state where $r_1 = 0$ and $r_2 = 0$. One possible derivation is the following (where we write '$\ldots$' for the state in the previous step).

$$\begin{array}{llcll}
\langle \varepsilon, & [x] := 1; r_1 := [y] & \| & [y] := 1; r_2 := [x]\rangle & \to^I \\
\langle \ldots \cdot W_1(x,1), & x \mapsto 1; r_1 := [y] & \| & [y] := 1; r_2 := [x]\rangle & \to^I \\
\langle \ldots \cdot R_1(r_1,y,0), & x \mapsto 1; \texttt{skip} & \| & [y] := 1; r_2 := [x]\rangle & \to^I \\
\langle \ldots \cdot W_2(y,1), & x \mapsto 1; \texttt{skip} & \| & y \mapsto 1; r_2 := [x]\rangle & \to^I \\
\langle \ldots \cdot C_2(y,1), & x \mapsto 1; \texttt{skip} & \| & \texttt{skip}; r_2 := [x]\rangle & \to^I \\
\langle \ldots \cdot R_2(r_2,x,0), & x \mapsto 1; \texttt{skip} & \| & \texttt{skip};\texttt{skip}\rangle & \to^I \\
\langle \ldots \cdot C_1(x,1), & \texttt{skip};\texttt{skip} & \| & \texttt{skip};\texttt{skip}\rangle &
\end{array}$$

The only non-trivial reordering we used was in the second step when we were executing $r_1 := [y]$ with $x \mapsto 1$ as the left context. From the definition of I^s we know that these instructions commute and, since the read is from a different location (y), the reordering action is trivial.

5.8 Related Work

In this chapter, we have described a small-step operational semantics to execute programs in a relaxed manner, i.e., to allow more behaviours than sequential consistency [44]. We chose a presentation of syntax more in the style of Kleene algebra with tests [43]. This is just to focus on the part that is central to this chapter (the reordering of instructions) and to abstract from the surface constructions of a concrete programming language. Roughly speaking, the operational semantics is then given by the Antimirov derivatives [10] of a regular expression. More precisely, here we build on the Antimirov reordering derivatives we developed in Chapter 3 where we extended the Antimirov derivatives to include the reordering of letters. Here we also included machine states in the rules. Hence the rules define transitions from configurations to configurations similarly to how structural operational semantics was defined by Plotkin [66].

The use of an independence relation and reordering derivatives relates this work to Mazurkiewicz traces [52] and trace languages (or in this case to semitraces [24]). A further generalisation of this would be to consider pomset-languages [30]. The notion of weak sequential composition (that is induced by an independence relation as we have

considered here) has also been investigated by Rensink and Wehrheim [70] in the context of process algebra.

The intended application of this work is to provide a framework for describing relaxed memory models in an operational manner. There are many earlier works that take an operational approach to relaxed memory. For example, the description of x86 was given (also) operationally by Owens et al. [61] and established that x86 follows the Total Store Order (TSO) memory model by SPARC [75]. An operational description of TSO was given by Jagadeesan et al. [35]. An operational approach to relaxed memory models as a minimalistic core calculus was given by Boudol and Petri [16]. Operational models have also been developed for POWER [73] and ARM [67].

Many of the operational descriptions mentioned above make use of (write) buffers. A simple buffer is just a first-in-first-out queue to delay the execution of some of the instructions, e.g., write instructions must pass through the buffer while other instructions do not. More generally, a buffer can be seen as a "reordering box" [62] and in this case all instructions are enqueued in the buffer and the order in which enqueued operations can be dequeued basically defines the memory model. Our approach does not explicitly use buffers since the left context of an instruction is essentially its buffer (these are the instructions that the given instruction will be reordered before). While the work by Boudol et al. [17] also includes a buffer (a temporary store), it is similar to ours in that they also have a commutability predicate (which corresponds to $\theta(u,a)$ in our setting) to describe the allowed reorderings in the buffer. In this respect, our approach is even more similar to the operational semantics by Colvin and Smith [25] where they consider reordering and forwarding of instructions in the semantics.

5.9 Conclusion and Future Work

We have described an operational semantics that is parameterised by the set of instructions (the alphabet Σ) and the allowed reorderings of pairs of instructions (the semicommutation θ). We have shown that the set of executions is closed under the rewriting relation induced by θ, i.e., if u is an execution of a program and θ allows to rewrite u to u', then u' is also an execution of this program. Furthermore, if θ happens to be conservative, then the executions u and u' lead to the same final state. We make use of this fact when we are calculating the set of final states for a given program as it is sufficient to consider only those executions that are representative. (This can be used for any property of executions that is stable under the rewriting relation induced by θ.) We then considered some extensions to this framework that would allow to describe more fine-grained reorderings. The central idea for this was to consider the left context of an instruction not as a program but as an execution as this allows to reason more precisely about which reorderings are allowed.

An obvious question for future work is: what are the memory models that can be precisely described in this framework? And what exactly are the details? For example, how is θ defined for a particular memory model? It is likely that some memory models require extensions or modifications in addition to those that we already considered in Section 5.6.

For example, we might want to include some form of collapsing of conditionals in our semantics to account for the fact that the conditional in the program

$$r := x; \texttt{if } r = 1 \texttt{ then } y := 1 \texttt{ else } y := 1$$

might be optimised away so that the program becomes

$$r := x; y := 1.$$

One possible solution for this might be to also consider "parallel" execution of conditionals in addition to what we have considered so far. By this we mean adding a rule that would look something like this:

$$\frac{\langle \sigma, p\rangle \to^{\theta} (a, \langle (\sigma)a, p'\rangle) \quad \langle \sigma, q\rangle \to^{\theta} (a, \langle (\sigma)a, q'\rangle) \quad \theta'(b,a)}{\langle \sigma, \texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ q\rangle \to^{\theta} (a, \langle (\sigma)a, \texttt{if}\ b\ \texttt{then}\ p'\ \texttt{else}\ q'\rangle)}$$

where θ' is meant to be weaker than θ but should still forbid reorderings that change the branch determined by b. With such a rule we can execute early those instructions that are available for execution in both branches without determining the branch that is taken eventually. To be more closer to optimising away the conditional we can also include variations of the above rule where, for example, if p' and q' are nullable, then the residual program is 1. We can also include a similar treatment of conditionals in the definition of the reorderable part: to reorder a before $\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ q$ along both branches we require that a can be reordered with both p and q (according to θ) and with b (according to θ'). Note that with this approach we would not have to require the left context to always be a single execution as we did in Definition 5.44.

6 Example: Multicopy-Atomic ARMv8

In this chapter, we briefly describe an experiment where we put the framework from Chapter 5 to the test by instantiating it to a fragment of the multicopy-atomic ARMv8 memory model [67]. Hence, our goal is to define an alphabet Σ of instructions, a semicommutation relation θ, the reordering actions and the "continuation" function for instructions so that (for a large number of sample programs) the executions generated are precisely those allowed by the memory model. It is not our goal here to propose an operational specification for the ARMv8 memory model. Instead, our focus is on describing how the framework from the previous chapter can be used for describing memory models and finding out if it is flexible enough.

We take the axiomatic model of ARMv8 (given in [67]) as our reference point and describe what we should say in the operational framework to match the axiomatic description. Ideally, there should be a proof that our translation of this axiomatic model precisely matches it. We do not provide such a proof here nor do we claim that this translation is in any way optimal. We do, however, validate our operational description against the axiomatic model as simulated by the `herd` tool [8]. For this purpose, we implement a prototype of our framework and compare its results with the results of `herd` on a large number of litmus-tests.

6.1 Abstract Machine

The machine model we consider consists of shared memory (a mapping from addresses to values) together with a fixed number of processors. We also instrument each memory location with a counter to keep track of how many writes to that memory location have occurred. This information is useful, for example, when determining whether two read instructions read the value written by the same write instruction. Each processor also has its own local memory (a mapping from registers to values). This altogether is the state set $\mathbb{S}$ in our semantics, i.e., $\sigma \in \mathbb{S}$ just gives the current shared memory state and the local memory of each processor. We consider the local memory of a processor to be invisible to other processors, i.e., processors can only communicate via shared memory.

Multicopy-atomicity requires that, if the effect of an instruction executed by some processor becomes visible to some other processor, then it is visible to all processors. Our abstract machine model seems well-suited for this: if the effect of an instruction becomes visible to another processor, then the effect must have modified the shared memory and is thus visible to all processors.

The set of instructions that we consider includes `MOV`, `EOR`, `AND`, `ADD`, `LDR`, `LDAR`, `STR`, `STLR`, `DMB.{LD,ST,SY}`, and `ISB`. We also include instructions like `CBNZ` and `CMP`, but we represent them using a generic instruction `TEST b` which is just a test of the Boolean condition `b`.

More concretely, the alphabet consists of instructions augmented with some additional information. For example, an `LDR` instruction consists of the target register (where the result of the load will be stored locally) and an address register (which holds the memory address from which to load the value). Similarly to the example in Section 5.7, we also include a field that, if set, contains the result of the load. The idea is that this field might be set during the reordering phase, i.e., when this `LDR` is reordered with its left context. After reordering, if this field is set, then this `LDR` does not read the shared memory—the result is already determined (for example, by reading a write early) and the value of this field is written to the target register. Whether this field is set might also affect the commutability relation.

Another addition is that we allow an instruction to keep track of the address or data registers of the instructions that it has already been reordered with. For example, we will see later that whether we can reorder an `STR` instruction with an earlier instruction depends on the address registers of the instructions that this `STR` has already been reordered with.

We also encounter situations where we may have several possible justifications for reordering two instructions. An example of this is described in the `(addr | data); rfi` rule that we will see later. We solve this by (speculatively) considering all possibilities and then later discarding those that turn out to be unjustified. For this reason, we consider the reordering actions in this section to be nondeterministic and thus describe them by a relation. For example, we can have $a, b \searrow\!\!\!\swarrow b', a'$ meaning that reordering a and b results in b' and a'. Furthermore, we may have several such pairs b', a' for a, b. In this example, we use the trivial "continuation" function κ that is always 1.

In the previous paragraph, we said that we may reorder two instructions without knowing whether this reordering is justified. To later check whether the reordering was justified we allow to add constraints to instructions during reordering. We set up the $\sigma \downarrow a$ relation so that executing an instruction in a state where its constraints are not satisfied is not allowed.

The programs we consider here are just parallel compositions of a fixed number of sequential threads, i.e., parallel composition only appears at the top-level of the expression. This matches the flat layout of the abstract machine.

6.2 From Axiomatic to Operational

We gave a brief introduction to axiomatic models in Section 2.9. The key ingredient of axiomatic models is the predicate which determines whether a candidate execution is allowed on the memory model. Importantly, this predicate is used on complete executions. An operational model, on the other hand, should say how to construct an execution, i.e., how a program can be executed on the memory model. The goal of the translation described here is to have an operational description that, for a given program, generates a set of executions that collectively produce precisely the set of final states that is allowed by the axiomatic model.

Another difference is that in the axiomatic model we can define arbitrary relations on the events, including events from different processors, and then define the allowed executions in terms of these relations, i.e., we have a global view of the execution. What we try to do with the operational model is much more local: we define the memory model in terms of the allowed reorderings. More precisely, we define when it is allowed to reorder an instruction with its left context (thus only referring to instructions from the same processor and possibly also to the current state) and what effect this has on the instruction and the context.

We already hinted at the motivation for left and right actions when we described the alphabet above: the actions allow us to keep track of the reorderings that have happened to an instruction. As a result, we can describe how reordering two instructions can modify their meaning and also their commutability with other instructions.

When we write $\theta_\sigma(a, b)$, then we are considering the case where a is before b in program order and we wish to reorder b before a. When we omit the σ then we mean that $\theta_\sigma(a, b)$ holds for every σ.

Recall that the axiomatic model is given in terms of relations on events representing instructions. In the `cat` language of `herd`, if `A` and `B` are relations, then their union is denoted by `A | B`, intersection by `A & B`, relational composition by `A; B` and transitive

closure by A+. The notation [W] denotes identity relations on certain kinds of events (write events in this case). Hence, the relation addr; po; [W] relates events a and c if there is an event b such that a and b are in the address dependency relation (addr), b and c are in the program order relation (po), and c is a write instruction (W).

We now look at the relations used in the axiomatic model and describe briefly how we would translate these into the operational model. First, we look at the *dependency-ordered-before* relation (denoted by dob) which is the union of the following relations. Informally, we interpret these relations as rules saying that it is not allowed to execute the instruction b when its left context u is of the form $u'au''$ and the instructions a and b are in the relation. We describe how to construct the semicommutation θ and the reordering actions by saying, for each of the following relations, what we require from the semicommutation θ and reordering actions to capture the relation in the operational model.

- addr: This is a subrelation of po. Let a and b be instructions so that a writes to a register that is used by b to determine the memory address it operates on. This is referred to as address dependency and we require $\neg\theta(a,b)$.

- data: This is a subrelation of po. Let a and b be instructions so that a writes to a register that is used by b to determine the value it writes to local or shared memory. This is referred to as data dependency and we require $\neg\theta(a,b)$.

- ctrl; [W]: The relation ctrl is a subrelation of po. Let a, b and c be instructions so that a writes to a register that is used by a test b and c is a write. We require $\neg\theta(a,b)$ and $\neg\theta(b,c)$ thus constructing a chain of dependencies from a to c.

  Alternatively, we could instead require that the left action of the test b on c annotates c with the registers used in the Boolean condition of the test. We then would also require that, if a write instruction (c) is annotated with a register r, then it cannot be reordered with an instruction that writes to r.

  (In the axiomatic model two events a' and b' are in the ctrl relation when a' is a read event and b' is an event in a conditional branch whose condition depends on the result of a'.)

- ctrl; [ISB]; po; [R]: Let a, b, c and d be instructions so that a writes to a register used by a test b, c is an ISB instruction and d is a read (that is after c in program order). We require $\neg\theta(a,b)$, $\neg\theta(b,c)$ and $\neg\theta(c,d)$.

- addr; po; [ISB]; po; [R]: Let a, b, c and d be instructions so that there is address dependency from a to b, c is an ISB and d is a read. As in the previous case, we require $\neg\theta(c,d)$.

  In addition, we require that the left action of b on c annotates c with the registers used to compute b's memory address. We then require that an ISB instruction (c) that has been annotated with a register r cannot be reordered with an instruction that writes to r. Since a and b are in the address dependency relation, a must write to r and thus we have $\neg\theta(a,{}_bc)$ where ${}_bc$ is c after reordering with b (i.e., c with the annotations from b).

- addr; po; [W]: Let a, b and c be instructions so that a and b are in the address dependency relation, b and c are in the program order relation and c is a write. Similarly to the previous case we require that the left action of b on c annotates c with the registers used to compute b's memory address. We then also require that

a write instruction that has been annotated with a register r cannot be reordered with an instruction that writes to r. Thus in this case we have $\neg\theta(a, {}_b c)$.

- `ctrl; coi`: Let a, b, c and d be instructions so that a writes to a register used by a test b and both c and d are write instructions to the same address. By the `ctrl; [W]` rule we already have $\neg\theta(a,b)$ and $\neg\theta(b,c)$. We also require $\neg\theta_\sigma(c,d)$ when in state σ the (write) instructions c and d are to the same address, i.e., the values of their address registers are the same.

  Note that for d to be reorderable with its left context, the `addr` rule requires the address register of d to be stable, i.e., nothing in the left context of d can write to its address register. Similarly, the `addr; po; [W]` rule (for d) requires that nothing in the left context of c and d can write to the address register of c.

- `data; coi`: Let a, b and c be instructions so that there is data dependency from a to b and both b and c are write instructions to the same memory address. By the `data` rule we have $\neg\theta(a,b)$. Just as in the previous case, we also require $\neg\theta_\sigma(b,c)$ when in state σ the (write) instructions b and c are to the same address.

- `(addr | data); rfi`: Let a, b and c be instructions so that there is address or data dependency from a to b, b is a write (with address register $addr(b)$ and data register $data(b)$) and c is a read (with address register $addr(c)$). There is an `rfi` edge from b to c when the registers $addr(b)$ and $addr(c)$ hold the same value when the corresponding instructions are executed and c reads the value written by b.

  We allow such b and c to be reordered in the operational model. The subtlety is that we consider two different justifications for this reordering.

  The first justification is that, in the end, there will not be an `rfi` edge, i.e., $addr(b)$ and $addr(c)$ will hold different values. In this case we require that the right action of c on b adds a constraint to b that, when executed, the address register $addr(b)$ must hold a value not equal to the current value of $addr(c)$.

  The other justification is that there will be an `rfi` edge. In this case we require that the right action of c on b adds a constraint to b that, when executed, the address register $addr(b)$ must hold the current value of $addr(c)$. Furthermore, the left action of b on c sets the read value of c to be the current value of $data(b)$ and it also must annotate c with its address and data registers. We then require that ${}_b c$ cannot be reordered with an instruction that writes to the registers it is annotated with. Thus we have $\neg\theta_\sigma(a, {}_b c)$. Since the read value of ${}_b c$ is now determined (by $data(b)$), the instruction does not access the memory at all.

  The two different justifications are possible precisely because we take the reordering actions to be given by a relation (a multivalued function).

We now consider the *barrier-ordered-before* relation (denoted by `bob`) which is the union of the following relations.

- `po; [dmb.full]; po`: Let a and c be any memory instructions (like reads or writes) and let b be a `DMB.SY`. We require both $\neg\theta(a,b)$ and $\neg\theta(b,c)$.

  Alternatively, we could also require that the left action of b on c annotates c with the information that it has been reordered with a `DMB.SY`. Then we would require that such ${}_b c$ cannot be reordered with a.

- `[L]; po; [A]`: Let a and b be instructions so that a is a release write and b is an acquire read. We require $\neg\theta(a,b)$.

- `[R]; po; [dmb.ld]; po`: Let a, b and c be instructions so that a is a read, b is a `DMB.LD`, and c is any memory instruction. We require $\neg\theta(a,b)$ and $\neg\theta(b,c)$.

- `[A]; po`: Let a and b be instructions so that a is an acquire read and b is any memory instruction. We require $\neg\theta(a,b)$.

- `[W]; po; [dmb.st]; po; [W]`: Let a, b and c be instructions so that a and c are write instructions and b is a `DMB.ST`. We require $\neg\theta(a,b)$ and $\neg\theta(b,c)$.

- `po; [L]`: Let a and b be instructions so that a is any memory instruction and b is a release write. We require $\neg\theta(a,b)$.

- `po; [L]; coi`: Let a, b and c be instructions so that a is any memory instruction, b is a release write, and c is a write to the same address. By the previous rule we have $\neg\theta(a,b)$. By the `addr; po; [W]` rule we know that to execute c early, the address register of b must be stable. We require that in a state σ where b and c write to the same memory address ($addr(b)$ and $addr(c)$ hold the same value) we have $\neg\theta_\sigma(b,c)$.

We have informally described what the context-dependent θ and the reordering actions should be to represent the *dependency-ordered-before* and *barrier-ordered-before* relations from the axiomatic model. We have currently excluded *read-modify-write* instructions from consideration and thus ignore the *atomic-ordered-before* relation (denoted by `aob`). The *observed-by* relation (denoted by `obs`) is defined in the axiomatic model as `rfe | fre | coe` and thus it is the union of the *read-from*, *from-read* and *coherence* relations between different processors. We will not translate these as we only consider reordering of instructions and this only happens on an individual processor, not between processors.

The *ordered-before* relation `ob` is defined as `(obs | dob | aob | bob)+` and it is required to be irreflexive. Our operational semantics constructs executions in a step-by-step manner, adding a new letter to a previously constructed execution. Since we forbid reordering of instructions that would have `dob` or `bob` edges between them, our executions should be such that `dob` and `bob` edges only go from an earlier instruction in the execution to a later one. The `rfe`, `fre` and `coe` relations represent communication between different processors and, on our abstract machine, this happens only through shared memory. This means that these edges should also only go from an earlier instruction in the execution to a later one. Thus we should not be able to construct an execution with a reflexive `ob` relation (i.e., one with a cycle along the relations `dob`, `bob`, `rfe`, `fre` and `coe`).

The memory model also requires the relation `po-loc | fr | co | rf` to be acyclic, i.e., its transitive closure to be irreflexive. The relation `po-loc` represents *program order per location* and it relates events that are in program order and to the same memory location. We interpret the acyclicity condition as saying that we can reorder instructions to the same location as long as this reordering is not visible in terms of the `co`, `rf` and `fr` relations. Let a and b be instructions in the `po-loc` relation (to memory location x) and say we reorder a and b (execute b early). What we wish to do next is rule out executions which would result in the instruction a (which is earlier in the program order) accessing the memory location x when it holds a later value than what it held when b (which is later in the program order) accessed it.

The only memory operations we consider are reads and writes. If we allowed to reorder write-write pairs to the same location, then we would violate the above condition and thus we forbid it. We also do not allow to reorder read-write pairs to the same location. We do allow to reorder write-read pairs to the same location (this was one case in the rule `(addr | data); rfi`). However, this reordering (the reordering action) determines the value of the read: although the instructions are reordered, the read instruction reads its value from the write instruction and thus does not access the memory. This preserves program order per location. We also allow to reorder read-read pairs to the same location. To preserve the desired property we must ensure that the two read instructions read values that are in agreement with their program order. In other words, we wish to forbid executions where a write to that memory location occurs between the reordered reads.

Let a and b be instructions so that both a and b are read instructions with $addr(a)$ and $addr(b)$ as their address registers. Similarly to the `(addr | data); rfi` rule we have two justifications for reordering a and b.

The first case is that the two instructions end up accessing different memory addresses. For this case we require the right action of b on a to add a constraint to a saying that, when executed, the value of $addr(a)$ is different from the current value of $addr(b)$.

The other case is that the two instructions end up accessing the same memory location. We require that the right action of b on a adds a constraint to a saying that, when executed, the value of $addr(a)$ is the same as the current value of $addr(b)$. Furthermore, we add a constraint to a that it must read its value from the same write as b did. (We accomplish this by checking that no writes have occurred to this location in between. This is why we instrumented memory locations with write counters.)

6.3 Prototype

We have implemented the part of the ARM memory model described above as a prototype in Haskell (available here: `http://cs.ioc.ee/~hendrik/code/phd/prototype.zip`).

The prototype contains implementations of the following functions where `Arm` is the alphabet (instructions annotated with some additional information) and `State` is the machine state.

```
theta :: State → Arm → Arm → Bool
act   :: State → Arm → Arm → [(Arm, Arm)]
sem   :: Arm → State → State
allow :: State → Arm → Bool
```

Thus `theta` is a context-dependent binary relation on the alphabet, `act` is the relational context-dependent reordering action (given as a multivalued function), `sem` is the interpretation of letters as (total) state transformers and `allow` is the relation which describes when an instruction is allowed to execute in a state.

This prototype allows us to take a litmus-test (a small concurrent program in pseudo-assembly), convert it to an element of RES, and, following the operational rules, collect all final states by enumerating all possible ways to execute this particular litmus-test from a given initial state. This allows us to compare the results (the set of final states obtained) to what is obtained by the memory model simulation tool `herd` (which is based on the axiomatic model) on the same litmus-test. We have also implemented the pruning mechanism described in Section 5.5 in the prototype. Also, the semicommutation relation and the reordering actions we have defined for ARM satisfy the conditions given in Defini-

tion 5.57. So far we have compared our prototype with `herd` on more than 8000 litmus-tests. The two tools agree on all of these litmus-tests.

6.4 Related Work

A significant part of the research on relaxed memory has been about hardware memory models. The description of x86 was given both operationally and axiomatically by Owens et al. [61] and they also showed that it follows the Total Store Order (TSO) memory model by SPARC [75]. A denotational (and operational) description of TSO together with a full abstraction result was given by Jagadeesan et al. [35]. A denotational semantics for TSO based on partially-ordered multisets (pomsets) was given by Kavanagh and Brookes [39]. Park and Dill defined an executable specification of Relaxed Memory Order (RMO) [62] by SPARC. Both axiomatic and operational models have also been considered for POWER [73] and ARM [67]. An extensive description of a (generic) axiomatic framework and several instantiations to different memory models was given by Alglave et al. [8]. Relaxed memory has also seen much interest in the context of model checking [7, 4, 5].

In addition to hardware memory models, the memory models of (concurrent) programming languages have also received attention. Ideally, a language specification should be precise enough to determine which optimisations are allowed. This in turn determines how relaxed this particular language is. Perhaps the most prominent of this line of work is formally describing the C/C++ memory model. The goal is to define a memory model that is sufficiently relaxed so that it allows common compiler optimisations but still excludes the unreasonable ones. The C/C++ memory model is given in the axiomatic style. Operational descriptions of (fragments of) the C/C++ memory model have been considered, for example, by Nienhuis et al. [57] and Doherty et al. [27]. These operational models stay very close to the axiomatic specification in the sense that they incrementally construct a valid axiomatic execution.

A problematic aspect of the C/C++ memory model is that it allows certain undesirable "out-of-thin-air" executions. The complexity of this issue is witnessed by the fact that, as it was observed by Batty et al. [12], the "out-of-thin-air" problem cannot be solved in a simple "per-candidate-execution" way. As the axiomatic models are "per-candidate-execution", this has led to the consideration of other methods in addition to axiomatic models.

An operational approach to the "out-of-thin-air" problem is the promising semantics by Kang et al. [37] which has also been adapted for ARM [68]. The operational framework we defined in Chapter 5 allows to execute an instruction early. The promising semantics allows a thread to promise to do something in the future while allowing other threads to see the effects of this in advance. Various forms of event structures [77] have also been considered for the "out-of-thin-air" problem [65, 22, 36, 63].

A slightly different concurrency model is considered by Fava et al. [29] who describe an operational semantics for the combination of weak memory and channel-based programming in the Go programming language.

6.5 Conclusion and Future Work

In this chapter, we described how to instantiate our framework from Chapter 5 to match (a fragment of) the multicopy-atomic ARM memory model. We validated a prototype implementation of our framework instantiation against the `herd` tool on a large number of litmus-tests.

The reorderings we include to represent the ARM memory model allow even relatively

small litmus-tests to have millions of unique executions. We are only interested in the result, i.e., the final state an execution produces. Representative executions allow us to consider a subset of all possible executions of a program. This of course relies on the fact that the executions we consider equivalent indeed produce the same final state.

We did not prove that our translation of the axiomatic ARM memory model is correct. It would be very nice to have this proof so that this operational translation could be used with confidence, even better if this were certified in a proof-assistant.

A similar problem would be to investigate if axiomatic descriptions of relaxed memory models could be translated into this operational framework in a systematic way, i.e., if it is the case that, for some well-delineated class of axiomatic models, the corresponding operational model can be constructed mechanically from the axiomatic model.

7 Conclusions and Future Work

7.1 Conclusions

In this dissertation, we have developed a framework for describing operational semantics where sequential composition of programs is interpreted in a weak manner. This weakness allows to describe certain kinds of relaxed memory models in the framework. This means that we can instantiate the framework so that it can also produce program executions that are not sequentially consistent, i.e., it is possible that the result of some execution cannot be obtained just by interleaving the program-order instruction sequences of individual threads. To achieve this, our framework allows the execution mechanism (processors) to modify the order of instructions that is specified by the program. More precisely, our use of an independence relation makes sequential composition weaker for certain pairs of instructions and so these pairs lose the ordering constraints otherwise introduced by sequential composition. For example, this means that in certain cases we can execute a as the first instruction in the program $p; a; q$.

The framework we have developed has several parameters: the alphabet of instructions, the semicommutation relation that may be context-dependent, the reordering action and the continuation function. As a result, there are many possibilities to tune the framework by slightly modifying the parameters. Thus we hope that this framework admits descriptions of several memory models. The framework does not have any reorderings built in and the reorderings that are allowed are controlled by the parameters: when we take an alphabet of instructions and trivially instantiate the other parameters, then we obtain sequential consistency.

Since we are interleaving instruction sequences of multiple threads, our approach is susceptible to the combinatorial explosion problem. The reordering of instructions makes this even worse. In this work, we were interested in the final states of the executions. Thus we should check each and every execution to see to what final state it takes us. We used normal forms as a sound mechanism to eagerly discard some executions from consideration. While this is quite effective for many of the litmus-tests we consider (even when the independence relation used for normal forms is smaller than the independence relation used for constructing the executions), it will run into problems at some point as the input programs grow.

Overall, we find the operational semantics we have defined based on the Antimirov reordering derivatives to be quite intuitive, at least when trying to explain or justify why some program on some memory model can behave in a certain way. Roughly speaking, at every step of the execution, we just select an instruction from the current residual program and try to "drag" it to the front for execution. The derivative operation then tells us what is the new residual program, i.e., what is still left to execute after we have executed the instruction we chose.

7.2 Future Work

In Chapter 3, we considered reordering derivative operations for a symmetric independence relation. It would be good to see which of these results continue to hold or which modifications are needed when we let go of the assumption that an independence relation has to be symmetric.

Another question to investigate might be to see what is the relationship between the reordering derivatives we have defined and Zielonka's asynchronous automata. It is a theorem that a trace-closed language is recognisable iff it is recognised by a finite asynchronous automaton. At the same time, a trace-closed language is recognisable iff there

exists a star-connected expression with the same closure. For a star-connected expression we can construct a finite automaton accepting the closure of the language.

In Chapter 4, we developed Foata and lexicographic normal forms for a generalisation of traces. In Chapter 6, we did not use these generalised normal forms as there we considered a semicommutation relation and it was context-dependent in a slightly different way. It would be good to see how to improve this situation.

The type of axiomatic semantics that are often used to describe relaxed memory models is quite different from the operational approach we have developed here. A useful direction for further work would be to investigate how to systematically construct an operational description corresponding to a given axiomatic description. It would also be interesting to consider the opposite direction, i.e., how to proceed from an operational description to an axiomatic one. This of course raises the question: which memory models can be represented in the current framework? Investigating the translation between several axiomatic and operational descriptions can reveal issues and shortcomings and thus lead to a more refined operational framework with more precise control over the relaxedness of the system.

References

[1] I. J. Aalbersberg and H. J. Hoogeboom. Characterizations of the decidability of some problems for regular trace languages. *Math. Syst. Theory*, 22(1):1–19, 1989.

[2] I. J. Aalbersberg and G. Rozenberg. Theory of traces. *Theor. Comput. Sci.*, 60(1):1–82, 1988.

[3] I. J. Aalbersberg and E. Welzl. Trace languages defined by regular string languages. *Theor. Inf. Appl.*, 20(2):103–119, 1986.

[4] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. Stateless model checking for TSO and PSO. *Acta Inf.*, 54(8):789–818, 2017.

[5] P. A. Abdulla, M. F. Atig, B. Jonsson, and C. Leonardsson. Stateless model checking for POWER. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 134–156. Springer, 2016.

[6] J. Alglave. *A shared memory poetics*. PhD thesis, Université Paris Diderot, 2010.

[7] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013.

[8] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[9] A. V. Anisimov and D. E. Knuth. Inhomogeneous sorting. *Int. J. Parallel Program.*, 8(4):255–260, 1979.

[10] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[11] D. Aspinall and J. Sevcík. Formalising Java's data race free guarantee. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2007.

[12] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 283–307. Springer, 2015.

[13] A. Bertoni, G. Mauri, and N. Sabadini. Equivalence and membership problems for regular trace languages. In M. Nielsen and E. M. Schmidt, editors, *Automata, Languages and Programming: 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings*, volume 140 of *Lecture Notes in Computer Science*, pages 61–71. Springer, 1982.

[14] A. Bertoni, G. Mauri, and N. Sabadini. Unambiguous regular trace languages. In J. Demetrovics, G. Katona, and A. Salomaa, editors, *Algebra, Combinatorics, and Logic in Computer Science*, volume 42 of *Collquia Mathematica Societas János Bolyai*, pages 113–123. North-Holland, 1986.

[15] A. Bouajjani, C. Enea, S. O. Mutluergil, and S. Tasiran. Reasoning about TSO programs using reduction and abstraction. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 336–353. Springer, 2018.

[16] G. Boudol and G. Petri. Relaxed memory models: an operational approach. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 392–403. ACM, 2009.

[17] G. Boudol, G. Petri, and B. P. Serpette. Relaxed operational semantics of concurrent programming languages. In B. Luttik and M. A. Reniers, editors, *Proceedings Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structural Operational Semantics, EXPRESS/SOS 2012, Newcastle upon Tyne, UK, September 3, 2012*, volume 89 of *EPTCS*, pages 19–33. Open Publishing Association, 2012.

[18] M. G. Brin. On the Zappa-Szép product. *Communications in Algebra*, 33(2):393–424, 2005.

[19] S. Broda, A. Machiavelo, N. Moreira, and R. Reis. Partial derivative automaton for regular expressions with shuffle. In J. Shallit and A. Okhotin, editors, *Descriptional Complexity of Formal Systems: 17th International Workshop, DCFS 2015, Waterloo, ON, Canada, June 25-27, 2015, Proceedings*, volume 9118 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 2015.

[20] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[21] P. Cartier and D. Foata. *Problemes combinatoires de commutation et rearrangements*, volume 85 of *Lecture Notes in Mathematics*. Springer, 1969.

[22] S. Chakraborty and V. Vafeiadis. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.*, 3(POPL):70:1–70:28, 2019.

[23] C. Chou and D. A. Peled. Formal verification of a partial-order reduction technique for model checking. *J. Autom. Reasoning*, 23(3-4):265–298, 1999.

[24] M. Clerbout and M. Latteux. Semi-commutations. *Inf. Comput.*, 73(1):59–74, 1987.

[25] R. J. Colvin and G. Smith. A wide-spectrum language for verification of programs on weak memory models. In K. Havelund, J. Peleska, B. Roscoe, and E. P. de Vink, editors, *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, volume 10951 of *Lecture Notes in Computer Science*, pages 240–257. Springer, 2018.

[26] V. Diekert and Y. Métivier. Partial commutation and traces. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Volume 3: Beyond Words*, pages 457–533. Springer, 1997.

[27] S. Doherty, B. Dongol, H. Wehrheim, and J. Derrick. Verifying C11 programs operationally. In J. K. Hollingsworth and I. Keidar, editors, *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pages 355–365. ACM, 2019.

[28] M. Droste. Concurrency, automata and domains. In M. Paterson, editor, *Automata, Languages and Programming, 17th International Colloquium, ICALP '90, Warwick University, England, UK, July 16-20, 1990, Proceedings*, volume 443 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 1990.

[29] D. S. Fava, M. Steffen, and V. Stolz. Operational semantics of a weak memory model with channel synchronization. *J. Log. Algebr. Meth. Program.*, 103:1–30, 2019.

[30] J. L. Gischer. The equational theory of pomsets. *Theor. Comput. Sci.*, 61(2-3):199–224, 1988.

[31] P. Godefroid. Using partial orders to improve automatic verification methods. In E. M. Clarke and R. P. Kurshan, editors, *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 1990.

[32] K. Hashiguchi. Recognizable closures and submonoids of free partially commutative monoids. *Theor. Comput. Sci.*, 86(2):233–241, 1991.

[33] T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra and its foundations. *J. Log. Algebr. Program.*, 80(6):266–296, 2011.

[34] P. W. Hoogers, H. C. M. Kleijn, and P. S. Thiagarajan. A trace semantics for Petri nets. *Inf. Comput.*, 117(1):98–114, 1995.

[35] R. Jagadeesan, G. Petri, and J. Riely. Brookes is relaxed, almost! In L. Birkedal, editor, *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7213 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2012.

[36] A. Jeffrey and J. Riely. On thin air reads: Towards an event structures model of relaxed memory. *Logical Methods in Computer Science*, 15(1), 2019.

[37] J. Kang, C. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 175–189. ACM, 2017.

[38] S. Katz and D. A. Peled. Defining conditional independence using collapses. *Theor. Comput. Sci.*, 101(2):337–359, 1992.

[39] R. Kavanagh and S. Brookes. A denotational semantics for SPARC TSO. *Logical Methods in Computer Science*, 15(2), 2019.

[40] S. C. Kleene. Representation of events in nerve sets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 3–42. Princeton University Press, 1956.

[41] B. Klunder, E. Ochmański, and K. Stawikowska. On star-connected flat languages. *Fund. Inf.*, 67(1–3):93–105, 2005.

[42] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Inf. Comput.*, 110(2):366–390, 1994.

[43] D. Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997.

[44] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[45] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[46] H. Maarand and T. Uustalu. Operational semantics with semicommutations. *Accepted for publication in J. Log. Algebr. Methods Program.*

[47] H. Maarand and T. Uustalu. Generating representative executions [extended abstract]. In V. T. Vasconcelos and P. Haller, editors, *Proceedings of the Tenth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2017, Uppsala, Sweden, 29th April 2017*, volume 246 of *EPTCS*, pages 39–48. Open Publishing Association, 2017.

[48] H. Maarand and T. Uustalu. Certified Foata normalization for generalized traces. In A. Dutle, C. A. Muñoz, and A. Narkawicz, editors, *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, volume 10811 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2018.

[49] H. Maarand and T. Uustalu. Certified normalization of generalized traces. *Innovations in Systems and Software Engineering*, 15(3-4):253–265, 2019.

[50] H. Maarand and T. Uustalu. Operational semantics with semicommutations. In T. Uustalu and J. Vain, editors, *31st Nordic Workshop on Programming Theory, NWPT 2019, November 13-15, 2019, Tallinn, Estonia, Abstracts*, pages 40–43. TTU, 2019.

[51] H. Maarand and T. Uustalu. Reordering derivatives of trace closures of regular languages. In W. J. Fokkink and R. van Glabbeek, editors, *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*, volume 140 of *LIPIcs*, pages 40:1–40:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[52] A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB-78, University of Aarhus, 1978.

[53] J. D. McKnight. Kleene quotient theorems. *Pac. J. Math.*, 14(4):1343–1352, 1964.

[54] A. Meduna and P. Zemek. Jumping finite automata. *Int. J. Found. Comput. Sci.*, 23(7):1555–1578, 2012.

[55] B. Nagy and F. Otto. Finite-state acceptors with translucent letters. In G. Bel-Enguix, V. Dahl, and A. O. de la Puente, editors, *Proceedings of the 1st International Workshop on AI Methods for Interdisciplinary Research in Language and Biology (BILC-2011)*, pages 3–13. SciTePress, 2011.

[56] H. R. Nielson and F. Nielson. *Semantics with applications - a formal introduction*. Wiley professional computing. Wiley, 1992.

[57] K. Nienhuis, K. Memarian, and P. Sewell. An operational semantics for C/C++11 concurrency. In E. Visser and Y. Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 111–128. ACM, 2016.

[58] U. Norell. Dependently typed programming in Agda. In P. W. M. Koopman, R. Plasmeijer, and S. D. Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2009.

[59] E. Ochmański. Regular behaviour of concurrent systems. *Bull. EATCS*, 27:56–67, 1985.

[60] E. Ochmański. Recognizable trace languages. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*, pages 167–204. World Scientific, 1995.

[61] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009.

[62] S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In C. E. Leiserson, editor, *7th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '95, Santa Barbara, California, USA, July 17-19, 1995*, pages 34–41. ACM, 1995.

[63] M. Paviotti, S. Cooksey, A. Paradis, D. Wright, S. Owens, and M. Batty. Modular relaxed dependencies in weak memory concurrency. In P. Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 599–625. Springer, 2020.

[64] D. A. Peled. All from one, one for all: on model checking using representatives. In C. Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993.

[65] J. Pichon-Pharabod and P. Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 622–633. ACM, 2016.

[66] G. D. Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, 1981.

[67] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.*, 2(POPL):19:1–19:29, 2018.

[68] C. Pulte, J. Pichon-Pharabod, J. Kang, S. H. Lee, and C. Hur. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1–15. ACM, 2019.

[69] M. O. Rabin and D. S. Scott. Finite automata and their decision problems. *IBM J. Res. Devel.*, 3(2):114–125, 1959.

[70] A. Rensink and H. Wehrheim. Weak sequential composition in process algebras. In B. Jonsson and J. Parrow, editors, *CONCUR '94, Concurrency Theory, 5th International Conference, Uppsala, Sweden, August 22-25, 1994, Proceedings*, volume 836 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 1994.

[71] J. Sakarovitch. On regular trace languages. *Theor. Comput. Sci.*, 52(1-2):59–75, 1987.

[72] J. Sakarovitch. The "last" decision problem for rational trace languages. In I. Simon, editor, *LATIN '92, 1st Latin American Symposium on Theoretical Informatics, São Paulo, Brazil, April 6-10, 1992, Proceedings*, volume 583 of *Lecture Notes in Computer Science*, pages 460–473. Springer, 1992.

[73] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 175–186. ACM, 2011.

[74] V. Sassone, M. Nielsen, and G. Winskel. Deterministic behavioural models for concurrency. In A. M. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings*, volume 711 of *Lecture Notes in Computer Science*, pages 682–692. Springer, 1993.

[75] SPARC International Inc and D. L. Weaver. *The SPARC Architecture Manual - version 8*. Prentice Hall, 1994.

[76] M. Sulzmann and P. Thiemann. Derivatives for regular shuffle expressions. In A. Dediu, E. Formenti, C. Martín-Vide, and B. Truthe, editors, *Language and Automata Theory and Applications: 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*, volume 8977 of *Lecture Notes in Computer Science*, pages 275–286. Springer, 2015.

[77] G. Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer, 1986.

[78] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings*. IEEE Computer Society, 2004.

[79] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In D. Grove and S. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 250–259. ACM, 2015.

[80] W. Zielonka. Notes on finite asynchronous automata. *Theor. Inf. Appl.*, 21(2):99–135, 1987.

[81] W. Zielonka. Asynchronous automata. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*, pages 205–247. World Scientific, 1995.

Appendix A Certified Normalisation of Generalised Traces

Here we give a more detailed account of the generalised Foata and lexicographic normalisation we developed in Chapter 4. The main difference is that here we stay closer to the Agda formalisation. The following are the relevant sections from Publication III.

In the electronic version, the definitions and lemmas given here also contain a hyperlink to the corresponding location in the HTML listing of the Agda code. The Agda formalisation itself is available here: `http://cs.ioc.ee/~hendrik/code/phd/isse.zip`

Generalised Mazurkiewicz Traces

We consider the generalisation of traces introduced by Sassone et al. [74]. In this setting the essential difference compared to ordinary traces is that the independence relation is no longer a binary relation but an assignment of an irreflexive and symmetric independence relation to every word `u`. More precisely, we assume that we have an alphabet `A` and a context-dependent independence relation

$$\mathtt{I} : \mathtt{A} \to \mathtt{List>\ A} \to \mathtt{A} \to \mathtt{Set}$$

The second parameter to `I` is for the context. We use cons-lists over `A` (elements of `List A`) to represent strings (`String = List A`) and snoc-lists (`List> A`) to represent contexts of strings and also (steps of) normal forms. Our notation for list operations follows the convention that the angle bracket points to the direction where the head element is. In the formal development, `A` and `I` together with their properties are module parameters.

We use both cons- and snoc-lists in the development, as this allows function definition by structural recursion and proof by structural induction from the correct end of the list which can be on the left or on the right depending on what is being done in a given situation. Typically, we want to work somewhere in the middle of a cons-list. We then split it into two parts, the left half (prefix) being a snoc-list and the right half (suffix) being a cons-list. Such pairs of snoc- and cons-lists are zippers for the cons-list type.

We seek to follow the lexical convention described below where reasonable:

- `a`, `b`, `c` and `d` are letters;
- `s`, `t`, `u` and `v` are snoc- or cons-lists;
- `ss` and `tt` are snoc-lists of snoc-lists.

Next, we describe the equivalence relation induced by the context-dependent independence relation. First we define when two strings differ only by the ordering of two adjacent independent letters.

Definition A.1. $\sim : \mathtt{List\ A} \to \mathtt{List>\ A} \to \mathtt{List\ A} \to \mathtt{Set}$

$$\frac{\mathtt{a}\ \mathtt{I}_{\mathtt{u +>\ s}}\ \mathtt{b}}{\mathtt{s >+<\ a <:\ b <:\ t} \ \sim_{\mathtt{u}}\ \mathtt{s >+<\ b <:\ a <:\ t}}\ \mathtt{swap}$$

This says that the strings `sabt` and `sbat` are equivalent in the context `u` when the letters `a` and `b` are independent in the context `u +> s`. The context is represented as a snoc-list as we usually need to access it from the right while strings are represented as cons-lists as we usually need to access them from the left. We use `<:` or for cons, `<+` for cons-append, `:>` for snoc and `+>` for snoc-append. We also use a mixed append operation `>+<` that takes a snoc- and a cons-list and produces a cons-list. When we need to translate

between the two representations, we use `s2c` for snoc-to-cons and `c2s` for cons-to-snoc translation.

Mazurkiewicz equivalence is the reflexive-transitive closure of the above relation.

Definition A.2. $\sim^* : \texttt{List A} \to \texttt{List> A} \to \texttt{List A} \to \texttt{Set}$

$$\frac{\texttt{s} = \texttt{t}}{\texttt{s} \sim^*_{\texttt{u}} \texttt{t}}\ \texttt{refl*} \qquad \frac{\texttt{s} \sim_{\texttt{u}} \texttt{v} \quad \texttt{v} \sim^*_{\texttt{u}} \texttt{t}}{\texttt{s} \sim^*_{\texttt{u}} \texttt{t}}\ \texttt{swap-trans*}$$

An element of $\texttt{s} \sim^*_{\texttt{u}} \texttt{t}$ can be seen as a sequence of instructions for transforming `s` into `t` by swapping adjacent independent letters. No letters from `u` can be involved in these swaps.

In generalised traces the family of independence relations is required to be *consistent*, i.e., stable under equivalence:

$$\texttt{I-cons} : \texttt{u} \sim^*_{[]} \texttt{v} \to \texttt{a}\ \texttt{I}_{\texttt{u}}\ \texttt{b} \to \texttt{a}\ \texttt{I}_{\texttt{v}}\ \texttt{b}$$

It is also required to be *coherent*:

$$\begin{aligned} \texttt{I-co1} &: \texttt{a}\ \texttt{I}_{\texttt{u}}\ \texttt{b} \to \texttt{b}\ \texttt{I}_{\texttt{ua}}\ \texttt{c} \to \texttt{a}\ \texttt{I}_{\texttt{ub}}\ \texttt{c} \to \texttt{a}\ \texttt{I}_{\texttt{u}}\ \texttt{c} \\ \texttt{I-co2-e} &: \texttt{a}\ \texttt{I}_{\texttt{u}}\ \texttt{b} \to \texttt{b}\ \texttt{I}_{\texttt{u}}\ \texttt{c} \to \texttt{a}\ \texttt{I}_{\texttt{u}}\ \texttt{c} \to \texttt{a}\ \texttt{I}_{\texttt{ub}}\ \texttt{c} \\ \texttt{I-co2-r} &: \texttt{a}\ \texttt{I}_{\texttt{u}}\ \texttt{b} \to \texttt{b}\ \texttt{I}_{\texttt{u}}\ \texttt{c} \to \texttt{a}\ \texttt{I}_{\texttt{ub}}\ \texttt{c} \to \texttt{a}\ \texttt{I}_{\texttt{u}}\ \texttt{c} \end{aligned}$$

The `e` and `r` suffixes in the name `I-co2` refer to extending and reducing the context. We have suppressed the formal notation for snoc-lists in the contexts here as the usual "silent" notation for strings is more readable.

We take the same independence alphabet to be our running example as we did in Chapter 4. Namely, the alphabet `A` consists of the letters `a`, `b`, `c`, `d` and the independence `I` is the least consistent and coherent family of symmetric relations such that $\texttt{a}\ \texttt{I}_{[]}\ \texttt{b}$, $\texttt{a}\ \texttt{I}_{[]}\ \texttt{d}$, $\texttt{b}\ \texttt{I}_{\texttt{a}}\ \texttt{d}$, $\texttt{b}\ \texttt{I}_{\texttt{ac}}\ \texttt{d}$, $\texttt{c}\ \texttt{I}_{\texttt{ab}}\ \texttt{d}$. Explicitly, this means that we also have $\texttt{b}\ \texttt{I}_{[]}\ \texttt{d}$ (by `I-co2-r`), $\texttt{a}\ \texttt{I}_{\texttt{d}}\ \texttt{b}$, $\texttt{a}\ \texttt{I}_{\texttt{b}}\ \texttt{d}$ (by `I-co2-e`) and $\texttt{c}\ \texttt{I}_{\texttt{ba}}\ \texttt{d}$ (by `I-cons`).

Generalised Foata Normalisation

In this section we describe Foata normal forms for generalised traces and the corresponding normalisation algorithm. We conclude with the correctness proof of the algorithm.

Normal Forms

We represent a Foata normal form as a snoc-list of steps which in turn are snoc-lists of letters. We define `Step` as a synonym for `List> A` and `Foata` as a synonym for `List Step`. These are the types of "raw" steps and normal forms.

In order to define well-formed normal forms, we introduce some auxiliary notation. We define $\texttt{s}\ {\blacksquare}\texttt{I}_{\texttt{u}}\ \texttt{a}$ to be `All (\ b → b I_u a) s`, expressing that, for every letter `b` in `s`, we have that `b` and `a` are independent in the context `u`. Similarly, we define $\texttt{s}\ {\blacklozenge}\texttt{D}_{\texttt{u}}\ \texttt{a}$ to be `Any (\ b → b D_u a) s`, expressing that there is a letter `b` in `s` such that `b` and `a` are dependent in the context `u`. Generally, the proposition `All P xs` holds when the predicate `P` holds on every element of the list `xs`. A proof of `Any P xs` points to some element in the list `xs` that satisfies the predicate `P`.

A step (in a context `u`) is considered well-formed if it satisfies the following predicate.

Definition A.3. $\texttt{StepOk} : \texttt{List A} \to \texttt{Step} \to \texttt{Set}$

$$\frac{}{\texttt{StepOk u [a]}}\ \texttt{sngl}$$

$$\frac{\texttt{StepOk u (s :> b)} \quad \texttt{b} \prec \texttt{a} \quad \texttt{(s :> b)} \blacksquare \texttt{I}_{\texttt{u}}\, \texttt{a}}{\texttt{StepOk u (s :> b :> a)}}\ \texttt{snoc}$$

A well-formed step (in a context `u`) is either a singleton or it consists of a well-formed step to which a new letter is added on the right, which has to be greater than the previous rightmost letter. The added letter and the step must be independent.

We now turn to well-formed normal forms. A letter in a step of a Foata normal form must have a dependent letter supporting it in the previous step. We formalise this by saying that the preceding normal form `ss` has to support the letter `a`, written as `Sup ss a`. This support is defined as `P-ne (\ tt t → t ♦D`$_{\texttt{concat>tt}}$ `a) ss`. Here we use a small helper `P-ne P xs` which holds trivially when `xs` is empty and when `xs` is non-empty it requires that `P ys y` holds where `ys` and `y` are the tail and head of the snoc-list `xs`. A "raw" Foata normal form (a list of steps) is well-formed if it satisfies the following predicate.

Definition A.4. `FoataOk : Foata → Set`

$$\frac{}{\texttt{FoataOk []}}\ \texttt{empty}$$

$$\frac{\texttt{FoataOk ss} \quad \texttt{StepOk (concat> ss) s} \quad \texttt{All (Sup ss) s}}{\texttt{FoataOk (ss :> s)}}\ \texttt{step}$$

Thus a well-formed Foata normal form can either be the empty list of steps or consist of a well-formed normal form with an added step. This step must be well-formed in the context of the normal form and every letter in the added step must be supported by the normal form.

As strings and normal forms are represented by different data types, we need to associate to a normal form its string representation. The function `emb` for embedding a normal form back into strings is defined as `emb ss = s2c (concat> ss)`. In other words, it just concatenates the steps in the normal form.

With our example independence alphabet, we have that `(abd)(c)` is a Foata normal form since we have `a` $\texttt{I}_{[]}$ `b`, `a` $\texttt{I}_{[]}$ `d` and `b` $\texttt{I}_{[]}$ `d` making `(abd)` a valid step and `a` $\texttt{D}_{[]}$ `c` ensuring that the sole letter in the step `(c)` is supported. We also have that `(a)(c)(bd)` is a normal form since `b` $\texttt{I}_{\texttt{ac}}$ `d` ensures that the step `(bd)` is well-formed and `a` $\texttt{D}_{[]}$ `c`, `c` $\texttt{D}_{\texttt{a}}$ `b` and `c` $\texttt{D}_{\texttt{a}}$ `d` provide the requisite support for the letters in the steps `(c)` and `(bd)`.

Normalisation

The normalisation algorithm traverses the input string (from the left) and inserts each letter into an accumulating normal form (from the right). The main ingredient thus is a function that takes a normal form and a letter and inserts the letter into its right place in the normal form. Given a normal form `nf` and a letter `a`, inserting the letter `a` into `nf` should produce a normal form `nf'` such that `emb nf'` is equivalent to `emb nf <+ [a]`.

We define a function `find>` parameterised by a decider `P?` of a predicate `P` on a context (a snoc-list) and an element. It splits a given snoc-list `xs` into two parts, `ls` and `rs`, so that all of the elements in `rs` satisfy the predicate and the rightmost element in `ls` violates the predicate.

Algorithm A.5 (`find>`)**.**

```
find> : (∀ xs x → Dec (P xs x)) → List> X → List> X × List> X
find> P? [] = [] , []
find> P? (xs :> x) with P? xs x
find> P? (xs :> x) | yes _ =
```

```
  let ls , rs = find> P? xs in ls , rs :> x
find> P? (xs :> x) | no  _ = xs :> x , []
```

Given a step and a letter, we use `find>` to find the right position for the letter in the step.

Algorithm A.6 (`insert-s`)**.**

```
insert-s : Step → A → Step
insert-s s a =
  let ls , rs = find> (\ _ b → a ≺? b) s in
  ls :> a +> rs
```

Notice that we use `find>` with a predicate that ignores the context. The step `s` is split into `ls` and `rs` so that everything in `rs` is greater than `a` and the rightmost letter in `ls` is not. We assume that the ordering relation $\prec$ is decidable, with $\prec$`?` as the decider. Hence `a ≺? b` is either `yes` (together with a proof of `a ≺ b`) or `no` (together with a proof of $\neg$(`a` $\prec$ `b`)).

Given a normal form and a letter, we use `find>` to find the correct step for the letter.

Algorithm A.7 (`insert`)**.**

```
insert : Foata → A → Foata
insert ss a with find> (\ tt t → ■I? tt t a) ss
insert ss a | ls , []      = ls :> ([] :> a)
insert ss a | ls , rs :> r =
  let s , rs' = first rs r in
  ls :> insert-s s a +> rs'
```

Here `find>` splits the normal form into two parts, `ls` and `rs`, so that all the steps in `rs` are independent of `a` and the rightmost step in `ls` is dependent (or `ls` is empty). If `rs` is empty, then we add a new step to the normal form. Otherwise, we insert `a` into the leftmost step in `rs` (the function `first` extracts leftmost element in a non-empty snoc-list). We assume that the independence relation `I` is decidable, with a decider `I?`. Here we use a derived decider ■`I?` for deciding whether a step and a letter are independent in the given context.

The normalisation function just traverses the input string from the left to the right and inserts each letter into the correct position in the accumulated normal form.

Algorithm A.8 (`norm`)**.**

```
norm' : Foata → String → Foata
norm' ss []       = ss
norm' ss (a <: t) = norm' (insert ss a) t

norm : String → Foata
norm t = norm' [] t
```

We continue with our example and look at the evolution of the accumulator as the string `bacd` is normalised. First, the letter `b` is inserted into the empty normal form, resulting in the normal form `(b)`. Next, the letter `a` is inserted into this normal form, which results in `(ab)` because of `a` $\mathtt{I}_{[]}$ `b`. Next, the letter `c` is inserted into the result. We have

a $D_{[]}$ c, which means that a new step must be added and the result is (ab)(c). We now need to insert d into the normal form. We have c I_{ab} d and in addition we also have a $I_{[]}$ d and b $I_{[]}$ d. This makes the first step the earliest possible step for d and the result is (abd)(c).

Correctness

We have now defined the Foata normalisation function, but we have no assurance yet that it produces well-formed Foata normal forms (i.e., elements of Foata satisfying the FoataOk predicate). Furthermore, we have no assurance that the elements satisfying FoataOk indeed are normal forms. We will now proceed to show that the function norm constructs a well-formed normal form from the input string and that the elements satisfying FoataOk are in bijection with equivalence classes of strings.

We start by showing that inserting a letter into a well-formed step gives a well-formed step.

Lemma A.9 (insert-sOk). $\forall$ u s a $\rightarrow$
StepOk u s $\rightarrow$ s $\blacksquare I_u$ a $\rightarrow$ StepOk u (insert-s s a)

Proof. Since s is a well-formed step, the letters in it are sorted wrt. $\prec$ and unique (by irreflexivity of the independence relation). By definition, insert-s splits s into ls and rs so that a is less than every letter in rs and a is not less than the rightmost letter in ls. We also have that s and a are independent, which implies independence of ls and rs of a. This allows us to construct StepOk u (ls :> a +> rs). □

To outline what we need to do next, let us look at a small example. Suppose we have a normal form stuv consisting of steps s, t, u, and v, and we wish to insert the letter a into this normal form. It so happens that a will go into the step t. This means that, instead of the old context st, the letters in u must now be independent in the new context s(insert-s t a). Likewise, the letters in v must now be independent in the context s(insert-s t a)u. Furthermore, every letter in v must now be supported by a letter in u in the context s(insert-s t a).

To show that the independence of letters in a step is preserved during an insert that inserts a letter into the context, we have the following lemma.

Lemma A.10 (step-ext). $\forall$ u s a $\rightarrow$
StepOk u s $\rightarrow$ s $\blacksquare I_u$ a $\rightarrow$ StepOk (u :> a) s

Proof. Since s is a well-formed step, we know that for any two distinct letters b and c from s we have b I_u c. We also have b I_u a and c I_u a. Using I-co2-e, we can derive b $I_{u:>a}$ c. This means that pairwise independence of letters in s is preserved in the extended context and the step is still well-formed. □

Next, we are considering the situation where we are inserting the letter a into the normal form ss :> s :> t and we have determined that a must go into a step in ss. We wish to show that the letters in t are still supported after the insert. We use PW I_u s to express that the predicate I_u holds between any two letters in s, i.e., the letters in s are pairwise independent in the context u. The normal form ss is considered here as the context u and b is a letter from the step t.

Lemma A.11 ($\blacklozenge$D-ext-lem). $\forall$ u s a b $\rightarrow$
b $I_{u+>s}$ a $\rightarrow$ PW I_u s $\rightarrow$ s $\blacksquare I_u$ a $\rightarrow$ s $\blacklozenge D_u$ b $\rightarrow$ s $\blacksquare I_{u:>a}$ b $\rightarrow \bot$

Proof. From the assumptions, we have that there is a letter `d` in `s` such that `d` $\mathtt{D_u}$ `b` and this `d` must also satisfy `d` $\mathtt{I_u}$ `a` and `d` $\mathtt{I_{u:>a}}$ `b`. We have derived a version of `I-co1` (named $\blacksquare$`I-co1`) that decreases the context not by a letter but by a step. We use this on `s` $\blacksquare\mathtt{I_u}$ `a`, `a` $\mathtt{I_{u+>s}}$ `b`, `s` $\blacksquare\mathtt{I_{u:>a}}$ `b` and `PW` $\mathtt{I_u}$ `s` to derive `a` $\mathtt{I_u}$ `b`. We use `I-co2-r` on `d` $\mathtt{I_u}$ `a`, `a` $\mathtt{I_u}$ `b`, and `d` $\mathtt{I_{u:>a}}$ `b` to get `d` $\mathtt{I_u}$ `b`. This contradicts `d` $\mathtt{D_u}$ `b`. □

This shows that, under suitable conditions, we can add a letter to the end of the context and still have a supporting letter in the previous step. From `I-cons`, we know that this support is then preserved for any equivalent context. To show that `insert` preserves the equivalence class, we first show that we can "slide" an independent letter past a step without changing the equivalence class.

Lemma A.12 (`slide-step`). `∀ u s a →`
`s` $\blacksquare\mathtt{I_u}$ `a →` `PW` $\mathtt{I_u}$ `s →` `s2c (u +> s :> a)` $\sim^*_{[]}$ `s2c (u :> a +> s)`

Proof. The proof is by induction on `s`. In the case where `s = s' :> b`, we have `s'` $\blacksquare\mathtt{I_u}$ `a`, `b` $\mathtt{I_u}$ `a`, `PW` $\mathtt{I_u}$ `s'`, `s'` $\blacksquare\mathtt{I_u}$ `b` and we get `b` $\mathtt{I_{u+>s'}}$ `a` by a derived version of `I-co2-e` that allows us to extend the context by a step. This allows us to swap `b` and `a` in the string `u +> s' :> b :> a` to obtain `u +> s' :> a :> b` and then apply induction hypothesis. □

Lemma A.13 (`insert-lem`). `∀ ss a →`
`FoataOk ss →` `emb (insert ss a)` $\sim^*_{[]}$ `emb ss <+ [a]`

Proof. The proof follows the analysis of `ss` done by `insert`. When a new singleton step is added (`ss` is empty or ends with a step that supports `a`), then the two sides are equal and we are done. When `a` is inserted into the last step `s`, then `s` and `a` must be independent and `s` is split into `ls` and `rs`. According to `slide-step`, we can slide `a` past `rs` to the end of the normal form without changing the equivalence class. When `a` is inserted into an earlier step, then we use induction hypothesis to slide the letter `a` from its inserted position to the beginning of the last step and, since `a` and the last step must have been independent to begin with, we can slide it past that step to the end of the normal form. □

Lemma A.14 (`sup-insert-lem`). `∀ ss s a b →`
`FoataOk (ss :> s) →` `a` $\mathtt{I_{concat>(ss:>s)}}$ `b →`
`¬ Sup (ss :> s) b →` `¬ Sup ss b →` `Sup (ss :> s) a →`
`Sup (insert (ss :> s) b) a`

Proof. Since neither `ss :> s` nor `ss` support `b`, we know that `insert (ss :> s) b` is the same as `insert ss b :> s`. Thus we need to show `Sup (insert ss b :> s) a`. Since `Sup (ss :> s) a` just denotes the existence of a dependent letter in `s`, we use ♦`D-ext-lem` to show that it cannot be the case that `s` $\blacksquare\mathtt{I_{concat>ss:>b}}$ `a`. From `insert-lem` we know that this context is equivalent to `concat> (insert ss b)` and so there must still be a supporting letter in `s` after the insert, thus `Sup (insert ss b :> s) a`. □

Lemma A.15 (`insertOk`). `∀ ss a →` `FoataOk ss →` `FoataOk (insert ss a)`

Proof. The proof follows the analysis of `ss` done by `insert`. When a letter `a` is inserted into a particular step `s`, then `insert-sOk` ensures that the resulting step is valid. When `insert` goes past a step `s` with the letter `a`, then `step-ext` ensures that the step `s` is still valid in the context extended with `a` and `insert-lem` ensures that `s` is valid after the `insert`. When `insert` goes past two steps, `s` and `t`, with the letter `a`, then `sup-insert-lem` ensures that all the letters in `t` are still supported by `s` in the context resulting from the `insert`. □

Lemma A.16 (norm'Ok). `∀ ss t → FoataOk ss → FoataOk (norm' ss t)`

Proof. The proof is by induction on `t` and just applies `insertOk` in the step case. □

Proposition A.17 (normOk). `∀ t → FoataOk (norm t)`

The correctness proof of the normalisation algorithm consists of the proofs of the soundness and completeness properties. By soundness we mean that equivalent strings must get assigned the same normal form. By completeness we mean that any two strings that get assigned the same normal form must be equivalent. With these properties we have a bijection between equivalence classes of strings and elements in the image of the normalisation function. We also show that the set of elements satisfying the `FoataOk` predicate is contained in the image of the normalisation function.

The key lemma for completeness is that the result of normalising a string (and then embedding it) is equivalent to that string. In other words, every string has a normal form.

Lemma A.18 (nf-exists'). `∀ ss t →`
`FoataOk ss → emb (norm' ss t)` $\sim^*_{[]}$ `emb ss <+ t`

Proof. The proof is by induction on `t`. In the step case, we use `insertOk` to show that inserting the first letter of `t` into `ss` is a normal form and then apply induction hypothesis. The equivalence follows from `insert-lem`. □

Proposition A.19 (nf-exists). `∀ t → emb (norm t)` $\sim^*_{[]}$ `t`

Corollary A.20 (completeness). `∀ t t' → norm t = norm t' → t` $\sim^*_{[]}$ `t'`

Proof. Apply `nf-exists` to both `t` and `t'`. □

To prove soundness of the normalisation algorithm, we first show the commutativity of the normalisation algorithm for independent letters. We start by showing that the order in which we insert two independent letters into a step does not matter.

Lemma A.21 (insert-s-commutes). `∀ u s a b →`
`StepOk u s → a` $\mathtt{I}_{\mathtt{u+>s}}$ `b → s ■`$\mathtt{I}_{\mathtt{u}}$ `a → s ■`$\mathtt{I}_{\mathtt{u}}$ `b →`
`insert-s (insert-s s a) b = insert-s (insert-s s b) a`

Proof. By `insert-sOk`, we know that `insert-s` produces well-formed steps. The letters in a well-formed step are sorted wrt. $\prec$ and unique. This means that the two ways to insert the two letters must result in the same step. Hence the order of the inserts does not matter. □

Lemma A.22 (insert-commutes). `∀ ss a b →`
`FoataOk ss → a` $\mathtt{I}_{\mathtt{concat>ss}}$ `b →`
`insert (insert ss a) b = insert (insert ss b) a`

Proof. There are three cases to consider: both letters are supported by `ss`, only one of them is, or neither of them is.

In the first case, a new step is added no matter whether we insert `a` first or `b` first. Since `a` and `b` are independent and supported by `ss`, both of them end up in the new step. The order in which the letters are inserted into the new step does not matter as the result must agree with the ordering on the alphabet.

In the second case, say that `a` is the letter supported by `ss`. When we first insert `a` and then `b`, then a singleton step for `a` is added. Since `a` and `b` are independent and

b is not supported by ss, it must be that b is inserted to some step in ss, i.e., we have insert (ss :> [a]>) b = insert ss b :> [a]>. This step is the same where b would be inserted if it were inserted first. Since a is supported by ss, then insert ss b must also support a (this follows from sup-insert-lem) and thus a new singleton step must be added also in this case.

In the third case, both letters are inserted into ss. Since ss cannot be empty, we take ss = ss' :> s. Here we perform another case analysis on into which steps the letters go. If both go into s, then we apply insert-s-commutes. If one of the letters, say b, goes into ss' and the other into s, then we argue similarly to the second case: if a is supported by ss', then it is also supported by insert ss' b, and, if s and b are independent, then so are insert-s s a and b. Thus the order of inserts does not matter. If both a and b go into ss', then we apply induction hypothesis. □

Lemma A.23 (norm'-commutes)**.** $\forall$ ss a b $\rightarrow$
FoataOk ss $\rightarrow$ a $\mathrm{I}_{\text{concat>ss}}$ b $\rightarrow$
norm' ss (a <: b <: []) = norm' ss (b <: a <: [])

Proof. This follows from insert-commutes. □

Lemma A.24 (norm'-append)**.** $\forall$ ss s t $\rightarrow$
norm' ss (s <+ t) = norm' (norm' ss s) t

Proof. By induction on s. □

Lemma A.25 (sound$\sim$)**.** $\forall$ ss t t' $\rightarrow$
FoataOk ss $\rightarrow$ t $\sim_{\text{concat>ss}}$ t' $\rightarrow$ norm' ss t = norm' ss t'

Proof. We have that t and t' differ only by the ordering of two adjacent independent letters, i.e., t = uabv and t' = ubav for some u, v, a and b. The result follows from norm'-commutes (norm' ss u) a b. We use norm'-append twice on both sides to get the result. □

Lemma A.26 (soundness')**.** $\forall$ ss t t' $\rightarrow$
FoataOk ss $\rightarrow$ t $\sim^*_{\text{concat>ss}}$ t' $\rightarrow$ norm' ss t = norm' ss t'

Proof. By induction on t $\sim^*_{\text{concat>ss}}$ t'. □

Proposition A.27 (soundness)**.** $\forall$ t t' $\rightarrow$ t $\sim^*_{[]}$ t' $\rightarrow$ norm t = norm t'

The soundness and completeness proofs give us a certified decision procedure for checking whether two strings are equivalent: normalise the two strings and check whether the normal forms are the same.

Algorithm A.28 (equivalent?)**.**

```
equivalent? : (t t' : String) → Dec (t ∼*[] t')
equivalent? t t' with foata-eq? (norm t) (norm t')
equivalent? t t' | yes feq = yes (completeness feq)
equivalent? t t' | no ¬feq = no (\ eqv → ¬feq (soundness eqv))
```

This procedure will either return yes, together with instructions how to turn u into v (which letters need to be exchanged), or no, together with a proof that it is not possible to turn u into v. Here foata-eq? uses the decidable equality on the alphabet to decide whether the two normal forms are the same.

We also have that the normalisation function is stable in the sense that normalising a normal form produces the same normal form. In other words, every normal form is the normal form of something.

Proposition A.29 (stability). $\forall$ ss $\to$ FoataOk ss $\to$ norm (emb ss) = ss

Proof. This is by induction on the normal form. In the non-empty case, we have to show that renormalising a step s in the context of the preceding normal form ss' results in the same step. This is the case since we started with a normal form and thus the step is well-formed and every letter in the step is supported by the preceding normal form. By induction hypothesis, we know that renormalising ss' results in ss'. Since letters from s are supported by ss', this means that no letter from s can be inserted into ss'. Similarly, all the letters from s fit into the same step. Hence the result is s. □

Finally, we have that two normal forms (more precisely, their embeddings) can be equivalent only if the normal forms are the same.

Corollary A.30 (nf-unique). $\forall$ ss ss' $\to$
FoataOk ss $\to$ FoataOk ss' $\to$ emb ss $\sim^*_{[]}$ emb ss' $\to$ ss = ss'

Proof. This follows from stability and soundness. □

Generalised Lexicographic Normalisation

In this section, we give a characterisation of lexicographic normal forms for generalised traces and the corresponding normalisation algorithm. We conclude with the correctness proof of the normalisation algorithm.

Normal Forms

We represent a "raw" lexicographic normal form as a snoc-list of letters (List> A). The embedding function emb of normal forms into strings is s2c.

We consider a list of letters to be a well-formed lexicographic normal form when each letter in it is in a valid position. Similarly to the previous section, a letter is in a valid position in a normal form if it is supported by the preceding normal form.

Definition A.31. Sup : List> A $\to$ A $\to$ Set

$$\frac{\texttt{P-ne (\ s' b} \to \texttt{b}\, \mathrm{D}_{\texttt{s}}\texttt{, a) s}}{\texttt{Sup s a}}\ \texttt{D-sup}$$

$$\frac{\texttt{Sup s a}\quad \texttt{b}\, \mathrm{I}_{\texttt{s}}\, \texttt{a}\quad \texttt{b} \prec \texttt{a}}{\texttt{Sup (s :> b) a}}\ \texttt{I-sup}$$

Hence a letter is supported by a (snoc-)list if either the list is empty or ends with a dependent letter or the tail of the list supports the letter and the head is independent of and smaller than the letter. A list of letters is a well-formed lexicographic normal form when every letter in the list is supported.

Definition A.32. LexOk : List> A $\to$ Set

$$\frac{}{\texttt{LexOk []}}\ \texttt{nil} \qquad \frac{\texttt{LexOk s}\quad \texttt{Sup s a}}{\texttt{LexOk (s :> a)}}\ \texttt{snoc}$$

We continue with our example and show that `abcd` is a lexicographic normal form (according to our definition). From `LexOk []` and `Sup [] a`, we get `LexOk a`. Next, we get `Sup a b` from `Sup [] b` using `I-sup` and thus get `LexOk ab`. We have `Sup ab c` by `D-sup` and $\mathtt{b}\,\mathtt{D}_{\mathtt{a}}\,\mathtt{c}$, resulting in `LexOk abc`. Finally, we get `Sup abc d` from `Sup [] d` by applying `I-sup` three times. Thus the list `abcd` is a well-formed normal form.

We now define what is a "chain" of independent letters wrt. a letter.

Definition A.33. $\mathtt{CI} : \mathtt{List>\ A} \to \mathtt{List>\ A} \to \mathtt{A} \to \mathtt{Set}$

$$\frac{}{\mathtt{[]}\ \mathtt{CI_u}\ \mathtt{a}} \qquad \frac{\mathtt{b}\ \mathtt{I_{u+>s}}\ \mathtt{a} \quad \mathtt{s}\ \mathtt{CI_u}\ \mathtt{a}}{\mathtt{(s\ :>\ b)}\ \mathtt{CI_u}\ \mathtt{a}}$$

When `s` is a chain of independent letters wrt. `a`, then we can "slide" `a` past `s`, i.e., we have the following equivalence: $\mathtt{emb\ (s\ :>\ a)} \sim^*_{\mathtt{u}} \mathtt{a\ <:\ emb\ s}$.

The characterisation of lexicographic normal forms by Anisimov and Knuth [9] forbids the "bua" pattern. Our definition also forbids this pattern in the generalised case.

Proposition A.34 (`LexOk-bua`)**.** $\forall\ \mathtt{t\ u\ v\ a\ b} \to$
$\mathtt{LexOk\ (t\ :>\ b\ +>\ u\ :>\ a\ +>\ v)} \to \mathtt{a}\ \mathtt{I_t}\ \mathtt{b} \to \mathtt{a} \prec \mathtt{b} \to \mathtt{u}\ \mathtt{CI_{t:>b}}\ \mathtt{a} \to \bot$

Proof. The proof is by induction on `v`. In the empty case, we have that `a` is supported by something since it is the last letter in a normal form. The actual support must come from `t` since `a` is independent of both `b` and `u` (in the relevant contexts). This however means that $\mathtt{b} \prec \mathtt{a}$ since the support for `a` must have been constructed by `I-sup`. This contradicts our assumption. In the case where `v` is non-empty, we apply induction hypothesis. □

We use the strict total order $\prec$ on `A` to define the corresponding lexicographic order relation on strings, both in the non-strict and strict versions, and prove that it is a total order.

Definition A.35. $\preceq_{\mathtt{Lex}} : \mathtt{String} \to \mathtt{String} \to \mathtt{Set}$

$$\frac{}{\mathtt{[]} \preceq_{\mathtt{Lex}} \mathtt{t}}\ \mathtt{nil}$$

$$\frac{\mathtt{a} \prec \mathtt{b}}{\mathtt{a\ <:\ s} \preceq_{\mathtt{Lex}} \mathtt{b\ <:\ t}}\ \mathtt{lt} \qquad \frac{\mathtt{a} = \mathtt{b} \quad \mathtt{s} \preceq_{\mathtt{Lex}} \mathtt{t}}{\mathtt{a\ <:\ s} \preceq_{\mathtt{Lex}} \mathtt{b\ <:\ t}}\ \mathtt{eq}$$

Lemma A.36 (`antisym-`$\preceq_{\mathtt{Lex}}$)**.** The relation $\preceq_{\mathtt{Lex}}$ is antisymmetric.

By definition, the lexicographic normal form is the least element in its equivalence class wrt. the lexicographic order $\preceq_{\mathtt{Lex}}$. Here we show that the normal forms we have defined are indeed lexicographically smaller than any other string in their equivalence class.

Lemma A.37 (`LexOk-lex'`)**.** $\forall\ \mathtt{u\ s\ t} \to$
$\mathtt{LexOk\ (u\ +>\ s)} \to \mathtt{emb\ s} \sim^*_{\mathtt{u}} \mathtt{t} \to \mathtt{emb\ s} \preceq_{\mathtt{Lex}} \mathtt{t}$

Proof. The proof is by induction on the strings `s` and `t`. If both are empty, then we are done since $\mathtt{[]} \preceq_{\mathtt{Lex}} \mathtt{[]}$. In the cases where one is empty and the other is not, we have a contradiction since equivalent strings must have the same length. In the case where both are non-empty ($\mathtt{a\ <:\ emb\ s}$ and $\mathtt{b\ <:\ t}$), we perform case analysis on the head elements. If $\mathtt{a} \prec \mathtt{b}$, then we are done. If $\mathtt{b} \prec \mathtt{a}$, then we have a contradiction since `b` must be somewhere in `emb s` (by the equivalence) and the letters before `b` (including `a`) must be independent with it. This creates a forbidden pattern in `u +> s`. In the case of `a = b`, we get by induction hypothesis, after moving `a` to the context `u`, that $\mathtt{emb\ s} \preceq_{\mathtt{Lex}} \mathtt{t}$. □

Proposition A.38 (`LexOk-lex`)**.** $\forall\ \mathtt{s\ t} \to$
$\mathtt{LexOk\ s} \to \mathtt{emb\ s} \sim^*_{\mathtt{[]}} \mathtt{t} \to \mathtt{emb\ s} \preceq_{\mathtt{Lex}} \mathtt{t}$

Normalisation

The main ingredient in the normalisation algorithm is a function that inserts a letter into its correct position in a list (which is assumed to be a well-formed normal form). Given a string `s` and a letter `a`, the idea is to split the string `s` into three parts: `sd`, `sp`, and `si` so that `sd` ends with a letter dependent on `a`, all letters in `sp` are independent of and less than `a`, and letters in `si` are independent of `a` and the first letter of `si` is greater than `a`.

Algorithm A.39 (`findPos`)**.**

```
findPos : List> A → A → List> A × List> A × List> A
findPos []       a = [] , [] , []
findPos (s :> b) a with I? s b a
findPos (s :> b) a | no _ = s :> b , [] , []
findPos (s :> b) a | yes _ with findPos s a
findPos (s :> b) a | yes _ | sd , sp , si :> i =
  sd , sp , si :> i :> b
findPos (s :> b) a | yes _ | sd , sp , [] with b ≺? a
findPos (s :> b) a | yes _ | sd , sp , [] | no _ =
  sd , sp , [] :> b
findPos (s :> b) a | yes _ | sd , sp , [] | yes _ =
  sd , sp :> b , []
```

The function `findPos` implements the described functionality. Like before, we assume that the independence relation `I` and the order relation $\prec$ are decidable, with deciders `I?` and $\prec$`?`. The `insert` function now just plugs the letter between `sp` and `si` in the result of `findPos`.

Algorithm A.40 (`insert`)**.**

```
insert : List> A → A → List> A
insert s a =
  let sd , sp , si = findPos s a in
  sd +> sp :> a +> si
```

The normalisation algorithm just traverses the input string letter by letter and inserts the letters into the accumulating normal form, just as in Foata normalisation.

Algorithm A.41 (`norm`)**.**

```
norm' : List> A → String → List> A
norm' s []       = s
norm' s (a <: t) = norm' (insert s a) t

norm : String → List> A
norm t = norm' [] t
```

We continue with our example and look at what are the intermediate steps when normalising `bacd`. First, when inserting `b` into the empty normal form, `insert` splits it into `[]`, `[]`, `[]` and the result is `b`. Next, when inserting `a`, the normal form `b` is split into `[]`, `[]`, `b` since `a` $\prec$ `b` and `b` $I_{[]}$ `a`. The result is `ab`. When inserting `c` into `ab`, the split is `ab`, `[]`, `[]` and the result is `abc`. Finally, when inserting `d` into `abc`, the split is the triple `[]`, `abc`, `[]` and thus the result is `abcd`.

Correctness

We have now defined the lexicographic normalisation algorithm. This produces "raw" normal forms, i.e., just snoc-lists. Next we show that the snoc-lists constructed by the normalisation function are well-formed normal forms in the sense that they satisfy the predicate `LexOk`. We then show that the strings satisfying `LexOk` are in bijection with the equivalence classes of strings.

We begin with a couple of lemmas exhibiting that `findPos` behaves as expected. The first lemma says that `findPos` just splits the input string.

Lemma A.42 (`findPos-split`). `∀ s a →`
```
let sd , sp , si = findPos s a in
sd +> sp +> si = s
```

Proof. From the definition of `findPos` it is clear that it does not rearrange the letters in `s` (`b` always stays to the right of the result of the recursive call to `findPos`). The proof just follows the analysis of `s` done by `findPos`. □

The next lemma ensures that the `si` component in the result of `findPos` consists of a "chain" of independent letters.

Lemma A.43 (`findPos-I`). `∀ s a →`
```
let sd , sp , si = findPos s a in
si CI_{sd +> sp} a
```

Proof. Here the proof also follows the analysis of `s` done by `findPos`. When `b` is added to the `si` component in the result, then we know that `b` and `a` must be independent. Induction hypothesis is used when the `si` component of the result of the recursive call is non-empty. □

The next lemma ensures that the leftmost letter of `si` in the result of `findPos` is greater than the letter `a`. The proposition `a ≺first si` holds when `a` is less than the first letter of `si`.

Lemma A.44 (`findPos-≺first`). `∀ s a →`
```
let _ , _ , si = findPos s a in
a ≺first si
```

Proof. From the definition of `findPos` we see that when the first letter is added to the `si` component, then it must be greater than `a` since it is not smaller than `a` and cannot be equal by irreflexivity of independence. □

We now show that `insert` preserves the equivalence class in the following sense: the normalisation algorithm uses `insert` in the situation where a prefix `s` has been normalised to `nf` and the suffix `a <: t` is yet to be normalised. Then `insert` will find the right place for `a` in `nf` such that the result of is equivalent to `nf :> a`.

Lemma A.45 (`insert-lem`). `∀ s a →`
```
emb (insert s a) ∼*_[] emb s <+ [ a ]
```

Proof. By definition `insert` plugs the letter `a` between `sp` and `si`. From `findPos-I` we get that `si` is a chain of independent letters and thus we can move `a` past it without changing the equivalence class. □

The next lemma ensures that under certain conditions the support of a letter is preserved when another letter is inserted into the supporting string.

Lemma A.46 (slideSup). ∀ s ii i b a →
Sup (s +> ii :> i) a → (ii :> i) CI_s b →
b $I_{s+>ii:>i}$ a → b ≺first (ii :> i) →
Sup (s :> b +> ii :> i) a

Proof. The proof is by induction on Sup (s +> ii :> i) a. The base case is D-sup, which means that we have i $D_{s+>ii}$ a and we need to show that i $D_{s:>b+>ii}$ a. This follows from I-co1 and I-cons. In the I-sup case, we have i $I_{s+>ii}$ a, but we do not know which is the dependent letter that supports a. If ii is empty, then b is inserted immediately before i and we construct I-sup using I-co2-r and I-co2-e. In the non-empty case, we construct the support from induction hypothesis and use I-co2-r and I-co2-e to show that i and a are still independent when the head of ii is added to the support. □

Lemma A.47 (insertOk). ∀ s a → LexOk s → LexOk (insert s a)

Proof. The proof follows the analysis of s done by findPos. The result follows from slideSup and the preceding lemmas about findPos. □

Lemma A.48 (norm'Ok). ∀ s t → LexOk s → LexOk (norm' s t)

Proof. The proof is by induction on t and applies insertOk in the step case. □

Proposition A.49 (normOk). ∀ t → LexOk (norm t)

Thus we have that the set of strings which satisfy the predicate LexOk (i.e., the normal forms) contains the image of the normalisation function. We continue with the soundness and completeness properties of the normalisation algorithm. By soundness we mean that equivalent strings get assigned the same normal form. By completeness we mean that any two strings that get assigned the same normal form must be equivalent. With these properties we have a bijection between equivalence classes of strings and the image of the normalisation function. We also show that the set of strings which satisfy the predicate LexOk is contained in the image of the normalisation function.

The key lemma for the completeness proof is that the result of normalising a string is equivalent to that string. In other words, every string has a normal form.

Lemma A.50 (nf-exists'). ∀ s t → emb (norm' s t) $\sim^*_{[]}$ emb s <+ t

Proof. The proof is by induction on t. In the step case, we use induction hypothesis together with insert-lem to show that inserting the first letter of t into s does not change the equivalence class. □

Proposition A.51 (nf-exists). ∀ t → emb (norm t) $\sim^*_{[]}$ t

Corollary A.52 (completeness). ∀ t t' → norm t = norm t' → t $\sim^*_{[]}$ t'

Proof. Apply nf-exists to both sides of the equation. □

Continuing towards soundness, we first prove the uniqueness of normal forms.

Proposition A.53 (nf-unique). ∀ s s' →
LexOk s → LexOk s' → emb s $\sim^*_{[]}$ emb s' → s = s'

Proof. By LexOk-lex we obtain from the assumptions both emb s $\preceq_{Lex}$ emb s' and emb s' $\preceq_{Lex}$ emb s, from which, by antisymmetry of $\preceq_{Lex}$, we have emb s = emb s'. Since emb = s2c is injective (inverted by c2s), s = s' follows. □

Corollary A.54 (soundness). $\forall$ t t' $\to$ t $\sim^*_{[]}$ t' $\to$ norm t = norm t'

Proof. By nf-exists both t and t' have a normal form and by assumption these are equivalent, i.e., we have emb (norm t) $\sim^*_{[]}$ emb (norm t'). The result follows from this by nf-unique and normOk. □

From uniqueness of normal forms we also get the stability of the normalisation algorithm, i.e., every normal form is the normal form of something. Thus the set of normal forms is contained in the image of the normalisation function.

Corollary A.55 (stability). $\forall$ s $\to$ LexOk s $\to$ norm (emb s) = s

Proof. This follows from nf-unique, normOk and nf-exists. □

An alternative approach to soundness would have been to prove the following lemma.

Lemma A.56 (insert-commutes). $\forall$ s a b $\to$
LexOk s $\to$ a I_s b $\to$ insert (insert s a) b = insert (insert s b) a

This leads to soundness, stability and uniqueness similarly to what we did for Foata normal forms.

Finally, we can now prove the converses of LexOk-lex and LexOk-bua showing that the least string in its equivalence class is the lexicographic normal form and that a string with no forbidden patterns is a lexicographic normal form.

Proposition A.57 (lex-LexOk). $\forall$ s $\to$
($\forall$ t $\to$ emb s $\sim^*_{[]}$ t $\to$ emb s $\preceq_{\mathrm{Lex}}$ t) $\to$ LexOk s

Proof. By nf-exists we have that emb s $\sim^*_{[]}$ emb (norm (emb s)). Thus, by assumption, emb s $\preceq_{\mathrm{Lex}}$ emb (norm (emb s)). At the same time, by normOk we have LexOk (norm (emb s)), from which emb (norm (emb s)) $\preceq_{\mathrm{Lex}}$ emb s follows by LexOk-lex. By antisymmetry of $\preceq_{\mathrm{Lex}}$, we have emb s = emb (norm (emb s)). As emb is injective, this entails s = norm (emb s). Since we have LexOk (norm (emb s)), then we also have LexOk s. □

Proposition A.58 (bua-LexOk). $\forall$ s $\to$
($\forall$ t u v a b $\to$
t :> b +> u :> a +> v = s $\to$ a I_t b $\to$ a $\prec$ b $\to$ u $\mathrm{CI}_{t:>b}$ a $\to \bot$) $\to$
LexOk s

Proof. If s = norm (emb s), then the result follows by normOk. If not, then we can factor the two as s = tbuav and norm (emb s) = tau'bv'. By nf-exists, we have emb tbuav $\sim^*_{[]}$ emb tau'bv'. The letter after t is the first position where the two differ (a $\neq$ b). We have a I_t b, u $\mathrm{CI}_{t:>b}$ a and u' $\mathrm{CI}_{t:>a}$ b since norm has moved them past each other. If a $\prec$ b, then we contradict the assumption that there are no forbidden patterns in s. If b $\prec$ a, then there is a forbidden pattern in norm (emb s), which is a normal form and thus, by LexOk-bua, does not contain a forbidden pattern. □

Acknowledgements

I am very grateful to my supervisor for his care and encouragement during my studies. Also, I am thankful to my colleagues at the institute for the nice atmosphere they have provided. I am also grateful to my friends for helping me relax and unwind. Most of all, I am indebted to my family for my upbringing.

My doctoral studies and the research reported here was supported by the ERDF funded Estonian national CoE project EXCITE (2014-2020.4.01.15-0018), the Estonian Ministry of Education and Research institutional grant no. IUT33-13, the Estonian IT Academy programme, the ERDF funded Dora Pluss programme, the EU COST action CA15123 (EUTYPES) and the Estonian Research Council/Campus France Estonian-French research cooperation programme Parrot.

Abstract

Operational Semantics of Weak Sequential Composition

In this dissertation, we propose an operational semantics where the effect of sequential composition can be relaxed for certain pairs of instructions. This allows, in the program $p;q$, to start with the execution of q even when p is not yet fully executed. The motivation for this work is that programs are often executed in a similarly relaxed manner: modern hardware often allows *out-of-order* execution and compiler optimisations can also move code around. When the effects of such optimisations become visible in a system, then the system is said to have a weak consistency model. This is often referred to as weak or relaxed memory models.

Our approach is to consider the set of possible instructions as an alphabet and use an independence relation on the alphabet (as in Mazurkiewicz traces) to describe those pairs of instructions that can be reordered during execution. We represent programs as regular expressions over the alphabet of instructions and consider program executions to be words over this alphabet.

As a first step, we develop reordering derivatives. More precisely, we provide syntactic derivative-like operations, defined by recursion on regular expressions, in the styles of both Brzozowski and Antimirov, for trace closures of regular languages. Just as the ordinary Brzozowski and Antimirov derivative operations correspond to the standard interpretation of regular expressions as regular languages, the derivative operations we develop here correspond to a non-standard interpretation of regular expressions as trace-closed languages. Similarly, these derivative operations can also be used to construct deterministic and non-deterministic automata, respectively. The trace-closing interpretation of regular expressions, however, does not yield a regular language in general, hence these automata cannot be finite in general.

We show that for star-connected expressions the Antimirov and Brzozowski automata, suitably quotiented, are finite. Furthermore, we also define a refined version of the Antimirov reordering derivative operation where parts-of-derivative (states of the automaton) are nonempty lists of regular expressions rather than single regular expressions. We define the uniform scattering rank of a language and show that, for an expression whose language has finite uniform scattering rank, the truncation of the (generally infinite) refined Antimirov automaton, obtained by removing long states, is finite without any quotienting, but still accepts the trace closure. We also show that star-connected languages have finite uniform scattering rank.

The operational semantics is then based on the Antimirov reordering derivative. To accomplish this we add parallel composition to the syntax, we let go of the requirement that the independence relation has to be symmetric, we thread machine states through the rules and we interpret letters of the alphabet as state transformers.

Representing executions as words allows the use of normal forms of Mazurkiewicz traces to alleviate the combinatorial explosion caused by reordering and interleaving the instruction sequences of different threads. We do this by eagerly discarding non-normal-form executions.

The specification of some memory models might require us to say that two instructions are independent in some state but not in others. With this in mind, we also develop Foata and lexicographic normal forms and corresponding normalisation algorithms for a generalisation of traces introduced by Sassone et al. and formalise it in Agda. This generalisation of traces makes the independence relation depend on a word parameter representing the left context which can be seen as a form of state. This context-dependent

approach can lead to larger equivalence classes. As a consequence, there are also more non-normal-form executions which can be discarded.

We then further extend the operational semantics to allow more intricate behaviours. For example, we will say that the semicommutation relation can be context-dependent (two instructions might commute in some machine state but not in others). We also consider what we call reordering actions. These allow us to describe how the reordering of two instructions might modify these instructions. We also add the possibility to execute an instruction in multiple steps. As an experiment, we describe a fragment of the multicopy-atomic ARMv8 memory model in this framework and validate a prototype implementation of the instantiation of the framework against the memory model simulation tool `herd` on a number of litmus-tests.

Kokkuvõte

Nõrga jadakompositsiooni operatsioonsemantika

Käesolevas doktoritöös arendame välja operatsioonsemantika, kus jadakompositsiooni toime teatud käsupaaridel võib olla lõtv ehk mitte täiesti järjestikune. See tähendab, et programmis $p;q$ on võimalik q täitmist alustada juba siis, kui p ei ole veel lõpuni täidetud. Selle töö ajendiks on asjaolu, et programme sageli täidetaksegi taolisel lõdval viisil: kaasaegne riistvara lubab käskude täitmist "väljaspool järjekorda" ning samuti võivad kompilaatorid programmikoodi muuta. Kui taolised optimisatsioonid muutuvad süsteemis nähtavaks, siis see süsteem järgib nõrka kooskõlamudelit. Sageli öeldakse siis, et tegu on nõrga või lõdva mälumudeliga.

Selles töös me käsitleme võimalike käskude hulka kui tähestikku ning kirjeldame ümberjärjestatavad käsupaarid sõltumatuse seosega sellel tähestikul nagu Mazurkiewiczi jälgede teoorias. Programme esitame me regulaaravaldistena sellel tähestikul ning programmijooksud on sõnad samuti sellel tähestikul.

Esimese sammuna taolise operatsioonsemantika suunas me töötame välja ümberjärjestavad tuletised. Teisisõnu, me defineerime regulaarkeelte jälgsulundite jaoks nii Brzozowski kui ka Antimirovi stiilis süntaktilised, regulaaravaldistel opereerivad, tuletiselaadsed tehted. Nii nagu tavalised Brzozowski ja Antimirovi tuletised vastavad regulaarvaldiste harilikule interpretatsioonile regulaarkeeltena, vastavad meie poolt defineeritud ümberjärjestavad tuletised regulaaravaldiste ebaharilikule interpretatsioonile regulaarkeelte jälgsulunditena. Samuti on võimalik ka ümberjärjestavaid Brzozowski ja Antimirovi tuletisi kasutada regulaaravaldisest deterministliku ja mittedeterministliku automaadi moodustamiseks. Need automaadid aktsepteerivad avaldise keele jälgsulundi. Kuna aga regulaarkeele jälgsulund ei ole üldjuhul regulaarne, ei saa ka need automaadid üldjuhul olla lõplikud.

Lisaks me näitame, et piisaval faktoriseerimisel on tärn-sidusate avaldiste Antimirovi ja Brzozowski automaadid lõplikud. Me defineerime Antimirovi ümberjärjestavast tuletisest ka rafineeritud variandi, kus üksikute avaldiste asemel on tuletise osadeks (automaadi olekuteks) mittetühjad avaldiste loendid. Me defineerime ka ühtlase laotusastaku ning näitame, et avaldiste korral, mille keelel on lõplik ühtlane laotusastak, on avaldise kärbitud rafineeritud Antimirovi automaat lõplik, on seda täiesti faktoriseerimata ning aktsepteerib avaldise keele sulundi. Kärpimine eemaldab automaadist pikad olekud. Lisaks näitame ka, et tärn-sidusatel avaldistel on lõplik ühtlane laotusastak.

Meie kirjeldatav operatsioonsemantika põhineb eelnimetatud Antimirovi ümberjärjestaval tuletisel. Selleks lisame avaldiste hulka paralleelkompositsiooni, loobume nõudest, et sõltumatuse seos peab olema sümmeetriline ning põimime masina olekud tuletise reeglitesse ja interpreteerime tähti olekuteisendajatena.

Programmijooksude esitamine sõnadena võimaldab käsujadade ümberjärjestamisel ning seejärgsel vaheldamisel tekkiva kombinatoorse plahvatuse pehmendamiseks kasutada Mazurkiewiczi jälgede normaalkujusid. Selleks me katkestame programmijooksu genereerimise niipea, kui märkame, et see pole normaalkujuline.

Mõningate mälumudelite spetsifikatsioon võib nõuda, et teatud käsupaarid on ühes olekus sõltumatud, aga teises mitte. Seda silmas pidades laiendame Foata ja leksikograafilised normaalkujud ning vastavad normaliseerimisalgoritmid ka Sassone jt. poolt defineeritud jälgede üldistusele ning formaliseerime need Agdas. See jälgede üldistus parametriseerib sõltumatuse seose sõnaparameetriga, mis esitab tähepaari vasakut konteksti. Seda konteksti saab vaadelda ka kui olekut. Taoline kontekstitundlikkus võib viia suuremate ekvivalentsiklassideni. Tulemusena saame rohkem mittenormaalkujulisi programmijookse,

mille võime varakult katkestada.

Viimaks laiendame me kirjeldatud operatsioonsemantikat veelgi, et võimaldada ka keerukamate programmikäitumiste esitamist. Näiteks lubame me kontekstitundlikke poolkommuteeruvuse seoseid ehk et käsupaar võib ühes masina olekus olla ümberjärjestatav, aga teises mitte. Me lubame ka ümberjärjestamise toimed, mis võimaldavad meil kirjeldada, kuidas käsupaari ümberjärjestamine võib neid käske muuta. Lisaks lubame me käske täita ka mitme sammu kaupa. Selle raamistiku võimekuse proovile panekuks me kirjeldame selles osa ARMv8 mälumudelist ning valideerime vastava prototüübi hulgal lakmustestidel mälumudelite simuleerimistööriista `herd` suhtes.

Curriculum Vitae

Personal data

Name	Hendrik Maarand
Date and place of birth	27 October 1988, Tallinn, Estonia
Nationality	Estonian

Contact information

Address	Department of Software Science, Tallinn University of Technology Akadeemia tee 21B, 12618, Tallinn, Estonia
E-mail	hendrik@cs.ioc.ee

Education

2015–2020	Tallinn University of Technology Information and Communication Technology, PhD studies
2012–2014	Tallinn University of Technology Informatics, MSc studies
2008–2011	Tallinn University of Technology Informatics, BSc studies

Language competence

Estonian	native
English	fluent
Russian	basic

Professional employment

2017–...	Department of Software Science, Tallinn University of Technology
2015–2016	Institute of Cybernetics, Tallinn University of Technology
2010–2015	Proekspert AS

Papers

1. H. Maarand and T. Uustalu. Certified normalization of generalized traces. *Innovations in Systems and Software Engineering*, 15(3-4):253–265, 2019

2. H. Maarand and T. Uustalu. Reordering derivatives of trace closures of regular languages. In W. J. Fokkink and R. van Glabbeek, editors, *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*, volume 140 of *LIPIcs*, pages 40:1–40:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019

3. H. Maarand and T. Uustalu. Operational semantics with semicommutations. In T. Uustalu and J. Vain, editors, *31st Nordic Workshop on Programming Theory, NWPT 2019, November 13-15, 2019, Tallinn, Estonia, Abstracts*, pages 40–43. TTU, 2019

4. H. Maarand and T. Uustalu. Certified Foata normalization for generalized traces. In A. Dutle, C. A. Muñoz, and A. Narkawicz, editors, *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, volume 10811 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2018

5. H. Maarand and T. Uustalu. Generating representative executions [extended abstract]. In V. T. Vasconcelos and P. Haller, editors, *Proceedings of the Tenth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2017, Uppsala, Sweden, 29th April 2017*, volume 246 of *EPTCS*, pages 39–48. Open Publishing Association, 2017

Elulookirjeldus

Isikuandmed

Nimi	Hendrik Maarand
Sünniaeg ja -koht	27.10.1988, Tallinn, Eesti
Kodakondsus	Eesti

Kontaktandmed

Aadress	Tallinna Tehnikaülikool, Tarkvarateaduse instituut, Akadeemia tee 21B, 12618, Tallinn, Eesti
E-post	hendrik@cs.ioc.ee

Haridus

2015–2020	Tallinna Tehnikaülikool Info- ja kommunikatsioonitehnoloogia, doktoriõpe
2012–2014	Tallinna Tehnikaülikool Informaatika, magistriõpe
2008–2011	Tallinna Tehnikaülikool Informaatika, bakalaureuseõpe

Keelteoskus

eesti keel	emakeel
inglise keel	kõrgtase
vene keel	algtase

Teenistuskäik

2017– …	Tarkvarateaduse instituut, Tallinna Tehnikaülikool
2015–2016	Küberneetika instituut, Tallinna Tehnikaülikool
2010–2015	Proekspert AS

Teadustegevus
Teadusartiklite loetelu on toodud ingliskeelse elulookirjelduse juures.