

Behavioural Analysis of Systems with Weights and Conditions

Coalgebraic and Algorithmic Perspectives on Behavioural
Analysis

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Informatik und Angewandte Kognitionswissenschaften
der
Universität Duisburg-Essen
zur Erlangung des akademischen Grades eines
Dr. rer. nat.

genehmigte Dissertation

von
Sebastian Küpper
aus
Düsseldorf

Gutachter: Prof. Dr. Barbara König

Gutachter: Prof. Dr. Filippo Bonchi

Tag der mündlichen Prüfung: 06.11.2017

Acknowledgements

First, I would like to thank my supervisor *Barbara König* for the opportunity to pursue a doctorate under her guidance. My work with her was a great and instructive experience. Her patience with beginner's problems and passion for her work allowed me to learn and enjoy scientific research as well as teaching.

One of the most important aspects of scientific research is learning about new perspectives and different approaches as only possible by the way of international collaboration. Thus, I would like to thank in particular my co-authors *Filippo Bonchi*, *Alexandra Silva* and *Thorsten Wißmann* who I enjoyed working with a lot, both from a professional and a personal perspective.

When it comes to local cooperation, I worked closely with *Harsh Beohar* and *Christina Mika*. It was a pleasure to discuss research and develop ideas together for several papers that are now the basis for this thesis. Both also provided thorough comments on drafts of the thesis, for which I am grateful.

I would also like to thank my colleagues *Christoph Blume*, *Sander Bruggink*, *Benjamin Cabrera*, *Mathias Hülsbusch*, *Dennis Nolte* and *Jan Stückrath* for a great working environment. A special thanks goes to *Henning Kerstan*, with whom I shared my office for four years. Besides being a great DJ at my wedding, he helped me understand coalgebraic notions in the beginning of my work and to overcome the hardships of having to use console applications on a Unix basis.

Though not directly involved in the research process, I would like to express my gratitude to my family and friends, who put up with the emotional lows and highs of scientific work and provided a warm and comfortable atmosphere in my life outside the university. Naturally, my wife *Tosha*, father *Rolf*, sister *Sarah* and grandmother *Erika* played a vital role. Finally, I would like to emphasise my late mother *Regina*, who unexpectedly died during my research on CTS, and proved to be caring and thoughtful right until our last conversation.

Inhaltsverzeichnis

1	Introduction	1
1.1	State-Based Systems with Conditions or Weights	1
1.2	Structure of the Thesis	3
1.3	Publications	5
2	Mathematical Foundations	7
2.1	Semirings and Semimodules	8
2.1.1	Semirings	8
2.1.2	Semimodules	10
2.2	Lattices and Order Theory	11
2.2.1	Partially Ordered Sets	12
2.2.2	Lattices	14
2.2.3	Birkhoff's Representation Theorem	17
2.2.4	Embedding into Boolean Algebras and Approximation into Lattices	21
2.3	State-Based Systems	24
2.3.1	From (Non-)Deterministic Automata to Weighted Auto- mata	25
2.3.2	From Labelled Transition Systems to Conditional Tran- sition Systems	30
2.4	Category Theory and Coalgebra	34
2.4.1	Categories and Morphisms	34
2.4.2	Functors	38
2.4.3	Natural Transformations and Monads	42
2.4.4	Coalgebra	46
2.4.5	Coalgebraic Behavioural Equivalence	48

3	Generic Partition Refinement Algorithms for Coalgebras	55
3.1	Introduction	55
3.2	Preliminaries	56
3.3	Generic Algorithms	58
3.4	Applications to Various Automata Models	70
3.4.1	Deterministic Automata and Labelled Transition Systems: The Classical Cases	71
3.4.2	Branching Bisimulation for LTS	76
3.4.3	Weighted Automata	81
3.4.4	HD-Automata	94
3.5	Conclusion	98
4	Language Equivalence for Weighted Automata: An Instantiation of the Final Chain Algorithm	101
4.1	Introduction	101
4.2	The Prototype Algorithm	103
4.2.1	The Operator F	104
4.2.2	Equivalences and Preorders on Matrices	105
4.2.3	Comparison to Conjugacy	108
4.2.4	Algorithm A for Checking Language Equivalence	111
4.2.5	Algorithm B for Checking Language Equivalence	113
4.3	Algorithmic Issues and Case Studies	120
4.3.1	A Concrete Instantiation of Algorithm B	120
4.3.2	Case Study: l -Monoids and Fuzzy Automata	123
4.4	Conclusion	128
5	Up-To Techniques for Weighted Systems	133
5.1	Introduction	133
5.2	Congruence Closure	136
5.2.1	Problem Statement	136
5.2.2	Congruence Closure for Rings	137
5.2.3	Embedding Semirings into Fields	139
5.2.4	Congruence Closure for l -Monoids	141
5.3	Up-To Techniques for Weighted Automata	157
5.3.1	Coinduction and Up-to Techniques	157

5.3.2	Language Equivalence for Weighted Automata	159
5.3.3	Language Inclusion	165
5.3.4	Threshold Problem for Automata over the Tropical Semiring	169
5.3.5	Exploiting Similarity	175
5.3.6	An Exponential Pruning	179
5.4	Runtime Results for the Threshold Problem	181
5.5	The Shortest Path Problem in Directed Weighted Graphs	187
5.6	Conclusion and Future Work	190
6	Algorithmic Issues and Applications for Conditional Transiti-	
	on Systems with Upgrades	191
6.1	Introduction	191
6.2	Conditional Transition Systems	193
6.3	Lattice Transition Systems	196
6.3.1	Correspondence to Fitting's Bisimulation	201
6.4	Computation of Lattice Bisimulation	202
6.4.1	A Fixpoint Approach	202
6.4.2	Lattice Bisimilarity is Finer than Boolean Bisimilarity .	208
6.4.3	Matrix Multiplication	209
6.4.4	Bisimulation Game	212
6.4.5	Deactivating Transitions	215
6.5	Application to Software Product Lines	223
6.5.1	Featured Transition Systems	223
6.5.2	BDDs as Models for Boolean Formulae	224
6.5.3	BDDs for Lattices	228
6.5.4	Implementation and Run-Time Results	233
6.6	Conclusion, Related Work, Future Work	234
7	Conditional Transition Systems Coalgebraically	239
7.1	Introduction	239
7.2	Preliminaries	241
7.3	Equivalence of Lattice Monad and Reader Monad	244
7.4	Modelling CTS without Upgrades using \mathcal{P}	253
7.5	Modelling CTS with Upgrades in $\mathbf{Kl}(_\Phi)$	257
7.6	Computing Behavioural Equivalence	264

7.7	Conclusion, Related Work and Future Work	274
8	Implementation, Future Work and Conclusion	277
8.1	PAWS: A Tool for the Analysis of Weighted Systems	277
8.2	Future Work	281
8.3	Conclusion	282
A	Additional Proofs (Chapter 3)	285
B	Additional Proofs (Chapter 5)	289
B.1	Proofs on the Embedding of Semirings	289
B.2	Termination of HKP without Abstraction	293
C	The Lattice Monad is a Monad (Chapter 7)	295

Chapter 1

Introduction

1.1 State-Based Systems with Conditions or Weights

State-based system models have a long history in computer science. The analysis of formal languages and state-based machines has brought forth a rich theory of system models with varying expressiveness results and algorithmic properties. To balance versatility of modelling and computability of key constructions and properties, numerous types of models have been investigated. Common among many of these models is that they are primarily used for modelling qualitative properties, such as the acceptance of a word and describe exactly one system per model.

Determining equivalent states in state-based systems is advantageous in many applications. As a system developer it might be interesting to know if a model matches the behaviour of the requirements. If (aspects of) both can be regarded as state-based machines, behavioural analysis can help answering questions that concern the comparison of a system model with its requirements model. Moreover, redundancies can be identified in a model. If two states are found to be behaviourally equivalent, it may be prudent to merge both and obtain a smaller model that behaves in the same way.

More recently, models that come equipped with a quantitative aspect that allow to express properties such as the likelihood or cost of a transition have been studied. Weighted automata over arbitrary semirings are well-suited for

modelling systems with quantitative aspects. The choice of semiring provides a vast amount of modelling opportunities. However, this versatility comes at a cost when it comes to analysing system models given in the form of weighted automata. In particular, it has been shown that language equivalence of weighted automata is undecidable in general, whereas for many specific semirings, e.g. the Boolean algebras or fields, rather efficient solutions to the problem exist. In this thesis, we will discuss two algorithms aimed at deciding language equivalence where it is decidable. The first, more general algorithm is applicable to weighted automata over all semirings in principle, provided containment of vectors in a semimodule is decidable. A second algorithm that uses up-to techniques to reduce the number of steps required to decide language equivalence is presented afterwards. This algorithm is limited to a certain range of semirings, but when it is applicable, it may significantly reduce the number of steps the algorithm takes and, consequently, lead to a gain in run-time.

In modern software development, it is common to develop not a single piece of software on its own, but to design a family of systems that is based on a common code base. Traditional state-based systems are insufficient to model such families of systems, because one would need to give a distinct model for each system. Conditional transition systems offer a modelling technique that allows for modelling software product lines with multiple products uniformly. Originally, the products in a CTS were considered as separate concerns, but in this thesis we will discuss a generalised model that allows for a hierarchy of products that allow for upgrading to improved products and an efficient procedure to determine behavioural equivalence taking all products into account.

It can be observed that behavioural analysis for various different kinds of state-based systems is based on a common base structure. To capture the essential concepts of procedures and structures is a central motivation for the theory of coalgebras. An algorithm generalising partition refinement from classical labelled transition systems to arbitrary coalgebraically specified state-based systems is the root point for the more fundamental aspects of this thesis. Extending the applicability of this standard procedure to further kinds of state-based systems, in particular, weighted automata, and identifying commonalities in particular in terms of termination conditions is the main driving force behind the coalgebraic aspects in this thesis. For both, weighted automata and CTS,

the same general procedure can be applied, even though the notion of behaviour differs, while taking a generic way of optimisation into account.

When considering state-based systems, different notions of behaviour have been developed, that highlight different perspectives. For weighted automata, which generalise non-deterministic automata, one is typically interested in language equivalence. Considering our previous examples, two states may be considered equivalent, if they perform the same tasks using the same amount of resources or succeeding with the same probability. For CTS, however, which generalise labelled transition systems in turn, a finer notion of behaviour is considered. Instead of looking at the complete runs and comparing possible traces from any given state, bisimulation also considers the decisions that are possible in each step. For a user of a system, it can make a difference, whether he has to decide at an earlier or at a later point during the run of a system, if he wants to perform an a action or a b action at some point. Therefore, bisimulation takes into account the time a choice is made, rather than purely the available sequences of decisions.

1.2 Structure of the Thesis

The thesis is covering three main lines of work, which focus on behavioural equivalence of various kinds of automata. From a coalgebraic point of view, a general procedure and some possible optimisations have been developed, which is defining a template algorithm that captures the common structure of various algorithms that can be found in the literature. This work builds on previous work in [ABH⁺12] where iteration on the final chain in the presence of a factorisation structure was investigated. The presented algorithm generalises the optimisations considered in the previous work and provides a more flexible termination condition.

The main motivator for this line of work was language equivalence for weighted automata, which could not be covered by the algorithm in its original form. The instantiation to weighted automata moreover generalises techniques that were presented in the literature for specific semirings and is therefore of interest in itself. However, to obtain further optimisations over the optimisations already present in the coalgebraic template algorithm, we needed to restrict the

class of semirings under consideration. This lead to an on-the-fly algorithm to decide language equivalence, that is additionally modified to solve the related problem of language inclusion and the threshold problem.

Finally, we also considered behavioural equivalence between any given pair of conditional transition systems (CTS). We have first considered these systems algorithmically, but later we have additionally developed a coalgebraic model for CTS which can be analysed using the aforementioned family of coalgebraic algorithms.

This is reflected in the structure of the thesis, comprising of five main chapters (Chapters 3 to 7), where the results from all three lines of work are presented independently of each other. Additionally, the work on coalgebraic behavioural equivalence and on conditional transition systems is split into two chapters each, one dealing with the coalgebraic reasoning and one chapter where an algorithmic perspective on the systems under consideration is taken.

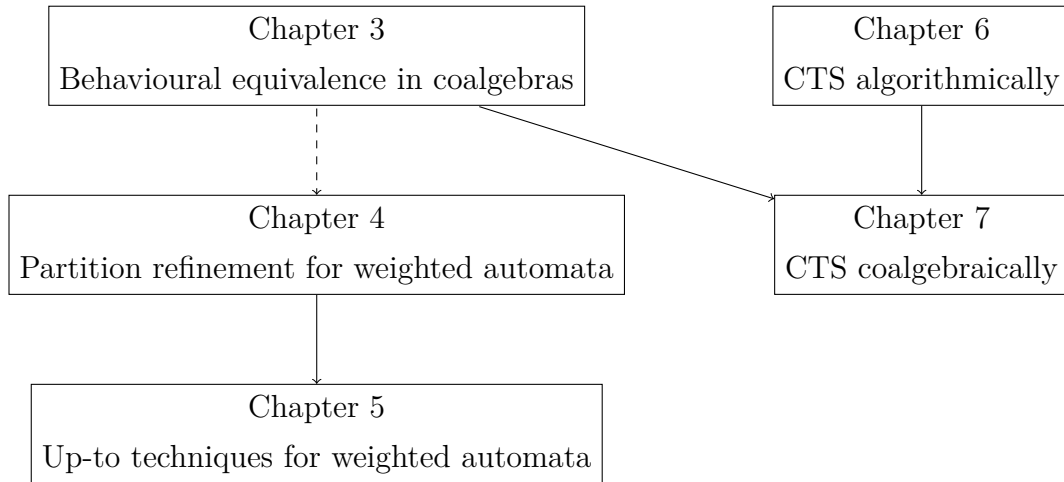
The common prerequisites to understand the content of the main chapters are presented in Chapter 2. Here, we focus on the fundamentals of category theory and coalgebra, as well as lattice theory.

There are two dimensions to reading this thesis. If one is primarily interested in weighted systems or systems with conditions, and if one is mainly interested in a coalgebraic treatment or a classical treatment. This thesis is designed to be approachable for any combination of reader interest. The following table shows, which chapters are covering which points of view. For instance, if one is only interested in the classical point of view, one may skip Chapters 3 and 7. In turn, a reader that is solely interested in the coalgebraic treatment of behavioural equivalence, may focus solely on these two chapters. In a similar way, a reader who prefers to only read about weighted systems, may be referred to Chapters 3 to 5, whereas for the conditional transition systems, the reader should focus on Chapters 6 and 7.

	Weighted systems	Systems with conditions
Algorithmic	Chapters 4 and 5	Chapter 6
Coalgebraic	Chapter 3	Chapter 7

Though all main chapters in the above table can be read and understood independently of each other, i.e. for each chapter only Chapters 1 and 2 are

prerequisites, there are some chapters that work on related concepts. Therefore, the following graph details recommended reading, where an arrow going from a node A to a node B signifies that it is recommended to read the chapter represented by node A before reading the chapter represented by node B . The dashed line indicates that the dependence is only relevant if the reader is interested in the coalgebraic aspects of this work.



Over the course of my work at the department, I have developed, together with my colleague Christina Mika, a tool called PAWS, short for Program for the Analysis of Weighted Systems, that aims to illustrate the algorithms developed in all three lines of work. Furthermore, PAWS was also used to obtain runtime results for the algorithms presented in the main chapters. It focusses particularly on the weighted automata line of work, where a special emphasis was put on flexibility. Section 8.1 serves as an overview over the design and usage of PAWS.

1.3 Publications

This thesis is based on the following publications:

- Barbara König and Sebastian Küpper. Generic partition refinement algorithms for coalgebras and an instantiation to weighted automata. In *Proc. of TCS '14*, IFIP AICT, pages 311–325. Springer, 2014. LNCS 8705 – Chapter 3

- Barbara König and Sebastian Küpper. A generalized partition refinement algorithm, instantiated to language equivalence checking for weighted automata. *Soft Computing*, pages 1–18, 2016 – Chapter 4
- Filippo Bonchi, Barbara König, and Sebastian Küpper. Up-to techniques for weighted systems. In *Proc. of TACAS '17, Part I*, pages 535–552. Springer, 2017. LNCS 10205 – Chapter 5
- Harsh Beohar, Barbara König, Sebastian Küpper, and Alexandra Silva. Conditional transition systems with upgrades. In *Proc. of TASE '17 (Theoretical Aspects of Software Engineering)*, 2017. to appear – Chapter 6
- Harsh Beohar, Barbara König, Sebastian Küpper, Alexandra Silva, and Thorsten Wißmann. Conditional transition systems coalgebraically. submitted. arXiv:1612.05002 – Chapter 7

Additionally, some contents from the following two publications were used in Chapter 4 and Section 8.1, respectively:

- Harsh Beohar and Sebastian Küpper. On path-based coalgebras and weak notions of bisimulation. In *Proc. of CALCO '17*, 2017. LIPIcs Vol. 72, to appear
- Barbara König, Sebastian Küpper, and Christina Mika. Paws: A tool for the analysis of weighted systems. In *Proc. of QAPL '17 (International Workshop on Quantitative Aspects of Programming Languages and Systems)*, 2017. to appear

My previous publications are thematically detached from this thesis and therefore are not represented in this work:

- H.J. Sander Bruggink, Barbara König, and Sebastian Küpper. Concatenation and other closure properties of recognizable languages in adhesive categories. In *Proc. of GT-VMT '13 (Workshop on Graph Transformation and Visual Modeling Techniques)*, volume 58 of *Electronic Communications of the EASST*, 2013
- H. J. Sander Bruggink, Barbara König, and Sebastian Küpper. Robustness and closure properties of recognizable languages in adhesive categories. *Sci. Comput. Program.*, 104:71–98, 2015

Chapter 2

Mathematical Foundations

This thesis builds on three fields of mathematical research: category theory, in particular coalgebraic modelling techniques, semirings and lattice theory. In this chapter, we will discuss all basic notions of these fields that we will require throughout the thesis, motivate the notions and give detailed examples. Additionally, we will discuss the previously established system models that we are working on, including weighted automata. This introductory chapter is broken up into four parts, focussing on

- Semirings and Semimodules
- Lattices and Order Theory
- Automata and Transition Systems
- Category Theory and Coalgebra

Section 2.4, which serves as an overview over the fundamentals of category theory, is of course only relevant if the reader is interested in Chapters 3 and 7.

We presuppose an intuitive understanding of sets. Throughout this chapter, at a few points we need to distinguish sets and classes, where the intuitive difference is that a class may contain any set, but a set may not. The finer details are not discussed here, because they do not play a role in the main chapters of this thesis. We fix some notions for sets A, B . $A \times B = \{(a, b) \mid a \in A, b \in B\}$, $A \setminus B = \{a \in A \mid a \notin B\}$, $A^B = \{f: B \rightarrow A\}$, $A \cap B = \{a \in A \mid a \in B\}$, $A \cup B = \{a \in A, b \in B\}$. Furthermore we write $A + B$ to denote the disjoint union of A and B . Typically we assume $A \cap B = \emptyset$, but if this is not the case,

this set can be characterised as $A + B = \{(a, \bullet), (\bullet, b) \mid a \in A, b \in B\}$, where $\bullet \notin A \cup B$. Moreover, we sometimes refer to a single element set as $1 = \{\bullet\}$.

2.1 Semirings and Semimodules

2.1.1 Semirings

In mathematics, sets with corresponding operations like addition and multiplication are ubiquitous, that is foundational for most areas, be it calculus, algebra or stochastics. In this subsection we want to discuss the building blocks for a structure that admits an addition and a multiplication, a semiring. The most basic algebraic structure we need to consider are monoids.

Definition 2.1.1 (Monoid) *Let M be a set and $\bullet: M \times M \rightarrow M$ be an operation on M , i.e. a function.*

- *If \bullet is associative, i.e. for all $m_1, m_2, m_3 \in M$ it holds that $(m_1 \bullet m_2) \bullet m_3 = m_1 \bullet (m_2 \bullet m_3)$, and there exists a neutral element $e \in M$ such that $e \bullet m = m \bullet e = m$ for all $m \in M$, we call (M, \bullet, e) a monoid.*
- *If \bullet is additionally commutative, i.e. for all $m_1, m_2 \in M$ it holds that $m_1 \bullet m_2 = m_2 \bullet m_1$, we call (M, \bullet, e) a commutative monoid.*

We will now consider some examples of monoids:

Example 2.1.2 • $(\mathbb{N}_0, +, 0)$ is a monoid: Addition is associative and $n + 0 = 0 + n = n$ for all $n \in \mathbb{N}_0$. Moreover, it is commutative, since addition is commutative.

- Similarly, $(\mathbb{N}_0, \cdot, 1)$ is a monoid.
- An example of a non-commutative monoid is $(\{0, 1, e\}, \bullet, e)$ where

$$\begin{array}{lll} 0 \bullet 1 = 1 & 1 \bullet 0 = 0 & 1 \bullet 1 = 1 \\ 0 \bullet 0 = 0 & 0 \bullet e = 0 & 1 \bullet e = 1 \end{array}$$

The operation \bullet is obviously not commutative, because $0 \bullet 1 = 1 \neq 0 = 1 \bullet 0$. It is associative though, because a product of three elements either contains an e and thus collapses to a product of two elements, or it equals 1 if and only if the rightmost element is 1.

Monoids only have a single operation, but when combining two monoids over the same carrier set, one may obtain a semiring.

Definition 2.1.3 (Semiring) *Let S be a set. A semiring is a tuple $\mathbb{S} = (S, +, \cdot, 0, 1)$, where $0 \in S$, $1 \in S$ and $1 \neq 0$, $(S, +, 0)$ is a commutative monoid, $(S, \cdot, 1)$ is a monoid, $0 \cdot a = a \cdot 0 = 0$ for all $a \in S$ and the distributive laws*

$$(a + b) \cdot c = a \cdot c + b \cdot c \text{ and } c \cdot (a + b) = c \cdot a + c \cdot b$$

hold for all $a, b, c \in S$. In the sequel we will identify \mathbb{S} with the set S .

Note, that semirings need not be commutative wrt. multiplication. Consequently, most of the work on semirings presented in later chapters works in the non-commutative case, unless otherwise stated. As previously hinted at, we can combine the monoids $(\mathbb{N}_0, +, 0)$ and $(\mathbb{N}_0, \cdot, 1)$ to obtain the prototypical example of a semiring – the semiring $(\mathbb{N}_0, +, \cdot, 0, 1)$ of natural numbers. This is not the only way to equip the natural numbers with the structure of a semiring though, another option that is of particular importance for the theory of weighted automata, is the tropical semiring:

Example 2.1.4 *The semiring $\mathbb{T} = (\mathbb{N}_0 \cup \{\infty\}, \min, +, \infty, 0)$, where \mathbb{T} -addition is the minimum and \mathbb{T} -multiplication is addition, is called the tropical semiring.*

In many cases, semiring operations have inverses, which gives rise to the more well-known structures ring and field.

Definition 2.1.5 (Ring, Field) *Let $(\mathbb{S}, +, \cdot, 0, 1)$ be a semiring.*

- *If there exists for each element $s \in \mathbb{S}$ an element $-s \in \mathbb{S}$ such that $s + (-s) = 0$, we call \mathbb{S} a ring. In that case, we also define the operation $-: \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S}$ where $s_1 - s_2 = s_1 + (-s_2)$.*
- *If additionally, for each element $s \in \mathbb{S} \setminus \{0\}$ there exists an element s^{-1} such that $s \cdot s^{-1} = 1$, and multiplication is commutative, then we call \mathbb{S} a field. In that case, we also define the operation $/: \mathbb{S} \times (\mathbb{S} \setminus \{0\}) \rightarrow \mathbb{S}$ according to $s_1/s_2 = s_1 \cdot (s_2^{-1})$.*

Example 2.1.6 *We start with two examples of semirings that are not rings.*

- $(\mathbb{N}_0, +, \cdot, 0, 1)$ is not a ring, because there exists no element $n \in \mathbb{N}_0$ such that $1 + n = 0$.
- The tropical semiring $\mathbb{T} = (\mathbb{N}_0 \cup \{\infty\}, \min, +, \infty, 0)$ is not a ring, because there exists no element $s \in \mathbb{N}_0 \cup \{\infty\}$ such that $\min(1, s) = \infty$.
- The prototypical ring is $(\mathbb{Z}, +, \cdot, 0, 1)$. Different from the carrier set \mathbb{N}_0 , the additive inverses for all elements exist in \mathbb{Z} . However, $(\mathbb{Z}, +, \cdot, 0, 1)$ is not a field, because e.g. 2 has no multiplicative inverse.
- Extending the carrier set further, we obtain the field $(\mathbb{Q}, +, \cdot, 0, 1)$. \mathbb{Q} contains exactly those elements that can be written as $\frac{z}{n}$, $z \in \mathbb{Z}, n \in \mathbb{N}$ and thus any element $\frac{z}{n} \in \mathbb{Q} \setminus \{0\}$ has a multiplicative inverse, namely $\frac{\text{sign}(z) \cdot n}{|z|}$.

2.1.2 Semimodules

Matrices and vectors over fields are a well-studied concept that plays an important role, for instance, when solving linear equations or in computer graphics. This concept can be generalised to semirings in a straight-forward way.

Definition 2.1.7 (Semimodule, Matrix) *Let X be an index set, let \mathbb{S} be a semiring and let \mathbb{S}^X be the set of all functions $s: X \rightarrow \mathbb{S}$ of finite support, i.e. functions where $s(x) \neq 0$ for only finitely many elements $x \in X$. Then \mathbb{S}^X is a semimodule, i.e., \mathbb{S}^X is a set closed under pointwise addition and multiplication with a constant (from the right). Every subset of \mathbb{S}^X closed under these operations is called a subsemimodule of \mathbb{S}^X .*

An $X \times Y$ -matrix a with weights over \mathbb{S} is given by a function $a: X \times Y \rightarrow \mathbb{S}$, i.e., it is an element of $\mathbb{S}^{X \times Y}$, such that in each column there are only finitely many entries different from 0. X indexes the rows and Y indexes the columns of the matrix. We can multiply an $X \times Y$ -matrix a and a $Y \times Z$ -matrix b in the usual way, obtaining an $X \times Z$ -matrix $a \cdot b$ where $(a \cdot b)(x, z) = \sum \{a(x, y) \cdot b(y, z) \mid y \in Y\}$ for all $x \in X, z \in Z$.

By $a_y: X \rightarrow \mathbb{S}$, where $y \in Y$, we denote the y -th column of a , i.e., $a_y(x) = a(x, y)$.

For a generating set $G \subseteq \mathbb{S}^Y$ of vectors we denote by $\langle G \rangle$ the subsemimodule spanned by G , i.e., the set that contains all linear combinations of vectors of G . Similarly, given an $X \times Y$ -matrix, we denote by $\langle a \rangle \subseteq \mathbb{S}^X$ the subsemimodule of \mathbb{S}^X that is spanned by the columns of a , i.e., $\langle a \rangle = \langle \{a_y \mid y \in Y\} \rangle$.

Note that we write matrices with small letters instead of capital letters. The reason is that when dealing with matrices over semirings in a coalgebraic setting, the matrices will be the arrows of the category under consideration and arrows are usually written in small letters. For consistency, we have decided to use small letters for matrices throughout this thesis.

Semimodules are a generalisation of vector spaces in the following sense:

Remark 2.1.8 (Comparison to vector spaces) *If the semiring \mathbb{S} happens to be a field, the notions from Definition 2.1.7 turn into well-known concepts from linear algebra. A (sub-)semimodule is a (sub-)vector space, a generating set for a semimodule is a generator for a vector space.*

Similarly, semimodules over rings are modules.

The usual notions of linear algebra (vector spaces, matrices, linear maps) can be extended from fields to semirings. Note that we will allow index sets X different from $\{1, \dots, n\}$.

Whenever we are talking about a semiring (ring, field) throughout this thesis, we assume that the operations $+$ and \cdot ($-$, $/$) of all discussed semirings \mathbb{S} can be computed effectively. This is relevant with regards to the algorithms we will later develop because they naturally rely on these operations.

2.2 Lattices and Order Theory

This overview over lattices and order theory contains all basic concepts we will require in the main chapters of the thesis. Many aspects of lattices and order theory will not be discussed here, because they are not required to understand the main parts of the thesis. For a more thorough introduction to lattices and order theory, including also the proofs missing in this section, the reader is referred to e.g. [DP02].

2.2.1 Partially Ordered Sets

Ordering objects according to a notion of size and identifying objects that are the same are two basic operations in mathematics that play a very important role in many concepts. For instance, the field of all real numbers may be characterised by its property of being the greatest Archimedean ordered field (with an order compatible with $+$ and \cdot), where an order plays a role twice, first by saying that \mathbb{R} is ordered, and second by saying it is the greatest field with that property – using a different notion of order, namely the subfield ordering. Orders will play a central role in all main chapters of this thesis. So we first recall the four basic properties required to specify what an order is:

Definition 2.2.1 *Let M be any set then we call any $R \subseteq M \times M$ a relation on M . The relation R is*

- reflexive if for all $x \in M$ it holds that $(x, x) \in R$.
- symmetric if for all $(x, y) \in R$ it holds that $(y, x) \in R$.
- transitive if whenever $(x, y) \in R$ and $(y, z) \in R$, then also $(x, z) \in R$.
- antisymmetric if for all $(x, y) \in R$ it holds that, if $(y, x) \in R$, then it must hold that $x = y$.

Using this terminology, we can define the notions of order and equivalence.

Definition 2.2.2 (Equivalence Relations and Partial Orders) *Let M be any set and $R \subseteq M \times M$ be a relation on M .*

- If R is reflexive, symmetric and transitive, we call R an equivalence relation and call any pair $(x, y) \in R$ equivalent.
- If R is reflexive, antisymmetric and transitive, we call R a (partial) order and (M, R) a partially ordered set (poset). If, in addition, for all pairs (x, y) it holds that $(x, y) \in R$ or $(y, x) \in R$, then we call R a total order.
- A function $f: (M, R) \rightarrow (M', R')$, where R and R' are partial orders is called order-preserving or monotone, if whenever $x \in M$, $y \in M$ and $(x, y) \in R$ it holds that $(f(x), f(y)) \in R'$.

Note that an equivalence relation is often written as \equiv and then \equiv is also written infix, i.e. instead of $(x, y) \in \equiv$, we will often write $x \equiv y$. Similarly, orders are often written \leq or \sqsubseteq . In either case, we will also write the order infix, i.e. instead of $(x, y) \in \sqsubseteq$ we write $x \sqsubseteq y$.

We will have a look at some simple examples of well-known orderings and equivalence relations.

Example 2.2.3 *We consider the set of all natural numbers including 0, \mathbb{N}_0 and identify some orders and equivalence relations:*

- *The traditional order \leq on \mathbb{N}_0 is a partial order: For all natural numbers n it holds that $n \leq n$, whenever $n \leq m$ and $m \leq n$ for any natural numbers $n, m \in \mathbb{N}_0$, it must hold that $m = n$ and for any natural numbers $n_1, n_2, n_3 \in \mathbb{N}_0$ it is true that $n_1 \leq n_2$ and $n_2 \leq n_3$ implies $n_1 \leq n_3$. \leq is also a total order on the natural numbers.*
- *Similarly, \geq is a (total) partial order as well. More generally, whenever R is a (total) partial order, the set $R' = \{(x, y) \mid (y, x) \in R\}$ is a (total) partial order as well.*
- *Another partial order on \mathbb{N}_0 , which is not total, is*

$$R = \{(n_1, n_2) \in \mathbb{N}_0 \times \mathbb{N}_0 \mid n_1 \leq n_2 \wedge n_1 \bmod 5 = n_2 \bmod 5\}.$$

Note that for instance 2 and 3 are incomparable, since $2 \bmod 5 = 2 \neq 3 = 3 \bmod 5$

- *An example that will play an important role throughout this thesis is the powerset $\mathcal{P}(M)$ of any given set M ordered by inclusion, i.e.*

$$\subseteq_M = \{(M', M'') \in \mathcal{P}(M) \times \mathcal{P}(M) \mid M' \subseteq M''\}.$$

Note, that the subset ordering on the class of all sets also meets all criteria to be regarded as a partial order, except for not being defined on a set, because the class of all sets is not a set itself.

- *$R_=$ on \mathbb{N}_0 , defined as*

$$R_= = \{(n_1, n_2) \in \mathbb{N}_0 \times \mathbb{N}_0 \mid n_1 = n_2\}$$

is both, a partial order and an equivalence relation. In fact, equality gives rise to a partial order and an equivalence relation on all base sets.

- Any natural number $n \in \mathbb{N}$ except 0 gives rise to an equivalence relation

$$R_n = \{(n_1, n_2) \in \mathbb{N}_0 \times \mathbb{N}_0 \mid n_1 \bmod n = n_2 \bmod n\}$$

None of these relations is a partial order, because $(n, 2 \cdot n) \in R_n$, but $n \neq 2 \cdot n$.

2.2.2 Lattices

When working with partially ordered sets, a helpful property is the existence of suprema and infima. This allows to reason about sets of elements from a partially ordered set in various ways.

Definition 2.2.4 (Supremum and Infimum) Let (M, \sqsubseteq) be any partially ordered set and $M' \subseteq M$ be any subset of M .

- If there exists an element $x \in M$ such that $x \sqsubseteq x'$ for all $x' \in M'$, then we call x a lower bound of M' .
- A lower bound x of M' is called the infimum of M' if it is maximal among all lower bounds, i.e. for all lower bounds $x'' \in M$, it holds that $x'' \sqsubseteq x$. We then write $x = \bigwedge M'$. If, in addition, $x \in M'$, we call x the minimum of M' and we write $x = \min(M')$. If M' has only two elements, we also write the infimum infix as $x = m_1 \sqcap m_2$, when $M' = \{m_1, m_2\}$. The infimum is also known as greatest lower bound.
- If there exists an element $x \in M$ such that $x \supseteq x'$ for all $x' \in M'$, then we call x an upper bound of M' .
- An upper bound x of M' is called the supremum of M' if it is minimal among all upper bounds, i.e. for all upper bounds $x'' \in M$, it holds that $x'' \supseteq x$. We then write $x = \bigvee M'$. If, in addition, $x \in M'$, we call x the maximum of M' and we write $x = \max(M')$. If M' has only two elements, we also write the supremum infix as $x = m_1 \sqcup m_2$, when $M' = \{m_1, m_2\}$. The supremum is also known as least upper bound.

Suprema and infima need not always exist, for instance, if we consider the set $\{1, 2\}$ and the partial order $R = \{(n_1, n_2) \in \mathbb{N}_0 \times \mathbb{N}_0 \mid n_1 \leq n_2 \wedge n_1 \bmod 5 =$

$n_2 \bmod 5\}$ from the previous example, then there exists no element that is the infimum or supremum of $\{1, 2\}$, since for an element n to be related to 1, it must hold that $n \bmod 5 = 1$, but for n to be related to 2 as well, it is required that $n \bmod 5 = 2$, which cannot be true at the same time.

Suprema and infima can exist of course, for instance when considering the partially ordered set (\mathbb{N}_0, \leq) and any two numbers n_1, n_2 , then

$$n_1 \sqcap n_2 = \begin{cases} n_1 & \text{if } n_1 \leq n_2 \\ n_2 & \text{otherwise} \end{cases}.$$

Similarly,

$$n_1 \sqcup n_2 = \begin{cases} n_1 & \text{if } n_2 \leq n_1 \\ n_2 & \text{otherwise} \end{cases}.$$

Whenever suprema and infima exist, they are unique.

Note that for (\mathbb{N}_0, \geq) , suprema and infima are swapped when compared with (\mathbb{N}_0, \leq) , i.e.

$$n_1 \sqcap n_2 = \begin{cases} n_1 & \text{if } n_2 \leq n_1 \\ n_2 & \text{otherwise} \end{cases} \quad \text{and} \quad n_1 \sqcup n_2 = \begin{cases} n_1 & \text{if } n_1 \leq n_2 \\ n_2 & \text{otherwise} \end{cases}.$$

The reason is that the order \sqsubseteq is always the basis for the notions of \sqcap and \sqcup . This kind of inversal, also called duality, happens at several points throughout this thesis, particularly in Chapter 7 and Chapter 5.

Suprema and infima are central operations on lattices, which we will now introduce.

Definition 2.2.5 (Lattice) *Let (L, \sqsubseteq) be a partially ordered set.*

- *If for all $\ell_1, \ell_2 \in L$ there exists the supremum $\ell_1 \sqcup \ell_2$ and the infimum $\ell_1 \sqcap \ell_2$, then \mathbb{L} is called a lattice. Sometimes, we refer to the lattice by its base set together with the supremum and infimum operator as (L, \sqcup, \sqcap) , from which the order can uniquely be derived.*
- *If, in addition, a top element $\top = \sqcup L$ and a bottom element $\perp = \sqcap L$ exist, \mathbb{L} is called bounded.*
- *If for all $L' \subseteq L$ the infimum $\sqcap L'$ and the supremum $\sqcup L'$ exist, we call $\mathbb{L} = (L, \sqsubseteq)$ a complete lattice.*

- If in a (complete, bounded) lattice the distributive law

$$(\ell_1 \sqcup \ell_2) \sqcap \ell_3 = (\ell_1 \sqcap \ell_3) \sqcup (\ell_2 \sqcap \ell_3)$$

holds, then we call it a distributive (complete, bounded) lattice.

- If (L, \sqcup, \sqcap) , (L', \sqcup', \sqcap') are lattices and $f: (L, \sqcup, \sqcap) \rightarrow (L', \sqcup', \sqcap')$ is a function, we call f a lattice homomorphism, if f preserves suprema and infima, i.e. $f(\ell_1 \sqcup \ell_2) = f(\ell_1) \sqcup' f(\ell_2)$ and $f(\ell_1 \sqcap \ell_2) = f(\ell_1) \sqcap' f(\ell_2)$. A lattice homomorphism is necessarily order preserving.

Note that any *finite* lattice is automatically bounded and complete and that complete lattices are always bounded. We will often identify the base set L of a lattice \mathbb{L} with \mathbb{L} itself.

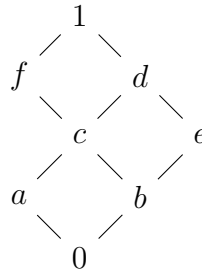
We will now consider several examples of lattices. Note that we have already discussed examples of partially ordered sets that are not lattices, particularly $R = \{(n_1, n_2) \in \mathbb{N}_0 \times \mathbb{N}_0 \mid n_1 \leq n_2 \wedge n_1 \bmod 5 = n_2 \bmod 5\}$ on \mathbb{N}_0 , since, as we have seen, not all pairs of elements have a supremum or an infimum.

Example 2.2.6

- $(\mathbb{N}_0, \max, \min)$ is a lattice. As seen before, for any given pair of natural numbers, the maximum and minimum exists. However, $(\mathbb{N}_0, \max, \min)$ is not bounded. While $\min \mathbb{N}_0 = 0$ exists, no element $\max \mathbb{N}_0$ exists. Thus, the lattice cannot be complete either. Still, $(\mathbb{N}_0, \max, \min)$ is distributive.
- $(\mathbb{N}_0 \cup \{\infty\}, \max, \min)$ is a distributive lattice as well, but it is additionally bounded, where $\max \mathbb{N}_0 \cup \{\infty\} = \infty$. It is also complete, because for all $M \subseteq \mathbb{N}_0 \cup \{\infty\}$, the supremum of M is defined according to

$$\bigsqcup M = \begin{cases} \infty & \text{if } \max M \text{ does not exist} \\ \max M & \text{otherwise} \end{cases}.$$

- We are particularly interested in finite distributive lattices. Consider the lattice $\mathbb{L} = \{0, a, b, c, d, e, f, 1\}$ with the order depicted below.



As one can easily verify, this lattice is distributive. Completeness follows from its finiteness. Using this graphical way of writing the order, known as Hasse diagram in the literature, it is easy to see the infimum and the supremum, by just following downwards or upwards the paths in the graph from any two given lattice elements. For instance, the supremum of e and a can be seen to be d .

The following example will be used later in Chapter 6

Example 2.2.7 Given a set N , consider $\mathbb{B}(N)$, the set of all Boolean expressions over N , i.e., the set of all formulae of propositional logic, where the elements of N are the atomic propositions. We equate every subset $C \subseteq N$ with the evaluation that assigns true to all $f \in C$ and false to all $f \in N \setminus C$. For $b \in \mathbb{B}(N)$, we write $C \models b$ whenever C satisfies b . Furthermore we define $\llbracket b \rrbracket = \{C \subseteq N \mid C \models b\} \in \mathcal{P}(\mathcal{P}(N))$. Two Boolean expressions b_1, b_2 are called equivalent whenever $\llbracket b_1 \rrbracket = \llbracket b_2 \rrbracket$. Furthermore b_1 implies b_2 ($b_1 \models b_2$), whenever $\llbracket b_1 \rrbracket \subseteq \llbracket b_2 \rrbracket$.

2.2.3 Birkhoff's Representation Theorem

We have already seen that lattices are just special partially ordered sets. However, there is a particularly deep correspondence between (finite) distributive lattices and (finite) partially ordered sets. To understand this connection, we first have to specify what a (join-)irreducible element is.

Definition 2.2.8 ((Join-)Irreducible elements, downward-closed sets)

Let \mathbb{L} be a lattice. An element $n \in \mathbb{L}$ is said to be (join-) irreducible if it is not the bottom element of \mathbb{L} and whenever $n = \ell \sqcup m$ for elements $\ell, m \in \mathbb{L}$, it always holds that $n = \ell$ or $n = m$. We write $\mathcal{J}(\mathbb{L})$ for the set of all irreducible elements of \mathbb{L} .

Let (Φ, \leq_Φ) be a partially ordered set. A subset $\Phi' \subseteq \Phi$ is downward-closed, whenever $\varphi' \in \Phi'$ and $\varphi \leq_\Phi \varphi'$ implies $\varphi \in \Phi'$. We write $\mathcal{O}(\Phi)$ for the set of all downward-closed subsets of Φ . Given $\varphi \in \Phi$, we write $\downarrow \varphi = \{\varphi' \mid \varphi' \leq_\Phi \varphi\}$ for the downward-closure of φ .

Example 2.2.9

- For our Example 2.2.7, $\mathbb{B}(N)$ quotiented by equivalence, the irreducibles are the complete conjunctions of literals, or, alternatively, all sets $C \subseteq N$.
- In Example 2.2.6, for all three infinite lattices, all elements other than 0 are irreducibles, since pairwise suprema are defined as maxima. In the example of a finite distributive lattice, the irreducible elements are a, b, e, f , i.e. exactly those elements that have a unique direct predecessor.

Distributive lattices give rise to an interesting duality result, which was first stated for finite lattices by Birkhoff and extended to the infinite case by Priestley [DP02]. The infinite case is significantly more complicated and requires notions from general topology. Since we do not need topology throughout this these, nor the infinite duality result, we will focus solely on the finite case here.

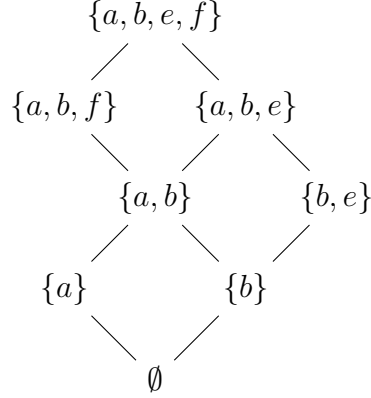
Theorem 2.2.10 (Birkhoff's representation theorem, [DP02]) *Let \mathbb{L} be a finite distributive lattice, then $(\mathbb{L}, \sqcup, \sqcap) \cong (\mathcal{O}(\mathcal{J}(\mathbb{L})), \cup, \cap)$ via the isomorphism $\eta : \mathbb{L} \rightarrow \mathcal{O}(\mathcal{J}(\mathbb{L}))$, defined as $\eta(\ell) = \{\ell' \in \mathcal{J}(\mathbb{L}) \mid \ell' \sqsubseteq \ell\}$. Furthermore, given a finite partially ordered set (Φ, \leq_Φ) , the downward-closed subsets of Φ , $(\mathcal{O}(\Phi), \cup, \cap)$ form a distributive lattice, with inclusion (\subseteq) as the partial order. The irreducibles of this lattice are all downward-closed sets of the form $\downarrow \varphi$ for $\varphi \in \Phi$.*

Going from \mathbb{L} to the isomorphic $\mathcal{O}(\mathcal{J}(\mathbb{L}))$, each lattice element $\ell \in \mathbb{L}$ is mapped to the set of all irreducible elements that are smaller than ℓ , i.e. $\{\ell' \in \mathcal{J}(\mathbb{L}) \mid \ell' \sqsubseteq \ell\}$.

Similarly, there is a correspondence between monotone functions from Φ to Φ' and lattice homomorphisms from $\mathcal{O}(\Phi')$ to $\mathcal{O}(\Phi)$.

We illustrate this result on our previous example of a finite distributive lattice:

Example 2.2.11 *We consider the finite distributive lattice from Example 2.2.6 and illustrate its dual representation of the lattice in terms of downward-closed sets of irreducibles, ordered by inclusion:*



Indeed, each element ℓ from the lattice can be represented by the set of all irreducible elements that are smaller than ℓ and the order is given by inclusion.

Bounded distributive lattices (with at least two elements) can also be interpreted as semirings in the following way: Let $(\mathbb{L}, \sqcup, \sqcap)$ be bounded distributive lattice, then there exist $\perp = \bigsqcap \mathbb{L}$ and $\top = \bigsqcup \mathbb{L}$. Additionally, we can compute for all $\ell \in \mathbb{L}$:

- $\perp \sqcup \ell = \ell$, since $\perp \sqsubseteq \ell$, i.e. \perp is the neutral element for \sqcup .
- $\top \sqcap \ell = \ell$, since $\top \supseteq \ell$, i.e. \top is the neutral element for \sqcap .
- $\perp \sqcap \ell = \perp$, i.e. \perp is cancellative for \sqcap .

Therefore, $(\mathbb{L}, \sqcup, \sqcap, \perp, \top)$ is a semiring.

As we have discussed before, it can be nice to have an inverse for the operations of a semiring, making the semiring a ring or even a field. However, sadly, an inverse in that sense cannot exist for the operations of a lattice, for the following reason: Assume in a bounded lattice with at least two elements, \top had an inverse (for \sqcup), which we call \top^{-1} . Then $\top \sqcup \top^{-1} = \perp$ since \top^{-1} is the inverse of \top . On the other hand $\top \sqcup \top^{-1} = \top \sqcup \top \sqcup \top^{-1} = \top \sqcup \perp = \top$, because a lattice is idempotent. Therefore, $\perp = \top$, which is a contradiction, since in a semiring $0 \neq 1$ must hold for the neutral elements of the operations.

Definition 2.2.12 (Heyting Algebra, Boolean Algebra) *Let $(\mathbb{L}, \sqcup, \sqcap)$ be a bounded distributive lattice.*

- If for any $\ell_1, \ell_2 \in \mathbb{L}$, there is a greatest element ℓ such that $\ell_1 \sqcap \ell \sqsubseteq \ell_2$, we call \mathbb{L} a Heyting algebra. Furthermore, we call ℓ the residuum $\ell = \ell_1 \rightarrow \ell_2 = \bigsqcup \{\ell' \mid \ell_1 \sqcap \ell' \sqsubseteq \ell_2\}$.
- If \mathbb{L} is equipped with a complement, i.e., for each $\ell \in \mathbb{L}$ there exists an element $\neg \ell$ such that $\neg \ell \sqcap \ell = \perp$ and $\neg \ell \sqcup \ell = \top$, we call \mathbb{L} a Boolean algebra. If the complement exists, it can be defined using the residuum as $\neg \ell = \ell \rightarrow \perp$.

Sometimes we say $\neg \ell = \ell \rightarrow \perp$ even in a Heyting algebra that is not a Boolean algebra. This negation is then well-defined, but it does not admit the de Morgan laws, which always hold in a Boolean algebra, i.e. $\neg(\ell_1 \sqcap \ell_2) = \neg \ell_1 \sqcup \neg \ell_2$ and $\neg(\ell_1 \sqcup \ell_2) = \neg \ell_1 \sqcap \neg \ell_2$ for all $\ell_1, \ell_2 \in \mathbb{L}$. Moreover, $\neg \ell \sqcup \ell = \top$ need not hold.

Also note that a complete distributive lattice \mathbb{L} need not be a Heyting algebra: $\ell_1 \rightarrow \ell_2 = \bigsqcup \{\ell \mid \ell_1 \sqcap \ell \sqsubseteq \ell_2\}$ must exist for all $\ell_1, \ell_2 \in \mathbb{L}$, but $(\ell_1 \rightarrow \ell_2) \sqcap \ell_1 \sqsubseteq \ell_2$ need not hold.

Example 2.2.13 Let $M \subseteq \mathbb{R}$, then we call M open, if for all $m \in M$ it holds that there exists an $\varepsilon > 0$ such that $m - \varepsilon \in M$ and $m + \varepsilon \in M$. We consider the partially ordered set $\mathbb{L} = (\{M \mid M \subseteq \mathbb{R} \mid M \text{ is open}\}, \supseteq)$. This forms a lattice, where $\sqcup = \cap$ and $\sqcap = \cup$. It is also complete: The union of open sets necessarily is open again, so infinite infima exist. The intersection of open sets need not exist in the infinite case, but its interior serves as the infinite supremum. The interior of an arbitrary set $M \subseteq \mathbb{R}$ can be constructed as the intersection of all open sets $M' \subseteq M$. It is also distributive, since intersection and union of sets distribute.

Now, to see that it is not a Heyting algebra, consider $\mathbb{R} \setminus \{0\} \rightarrow \mathbb{R}$. Note, that for all open subsets M that contain 0, it holds that $\mathbb{R} \setminus \{0\} \cup M \supseteq \mathbb{R}$, i.e. $\mathbb{R} \setminus \{0\} \sqcap M \sqsubseteq \mathbb{R}$. The supremum of all these intervals is the interior of their intersection. Their intersection is just the set $\{0\}$, which is not open. Its interior is \emptyset – trivially, it is the only open subset of $\{0\}$. Therefore, we can conclude that $\mathbb{R} \setminus \{0\} \rightarrow \mathbb{R} = \bigsqcup \{M \in \mathbb{L} \mid \mathbb{R} \setminus \{0\} \sqcap M \sqsubseteq \mathbb{R}\} = \emptyset$. Yet, $\mathbb{R} \setminus \{0\} \sqcap \emptyset \not\sqsubseteq \mathbb{R}$, so this infimum does not actually yield a proper residuum.

If in a complete distributive lattice, in addition the infinite distributive law $\ell \sqcap \bigsqcup_{i \in I} \ell_i = \bigsqcup_{i \in I} (\ell \sqcap \ell_i)$ holds for all $\ell, \ell_i \in \mathbb{L}$, $i \in I$, \mathbb{L} is always a Heyting algebra.

Example 2.2.14 *The set $\mathbb{B}(N)$ from Example 2.2.7, quotiented by equivalence, is a Boolean algebra, isomorphic to $\mathcal{P}(\mathcal{P}(N))$, where $\llbracket b_1 \rrbracket \sqcup \llbracket b_2 \rrbracket = \llbracket b_1 \rrbracket \cup \llbracket b_2 \rrbracket = \llbracket b_1 \vee b_2 \rrbracket$, analogously for \sqcap, \cap, \wedge , $\neg \llbracket b \rrbracket = \mathcal{P}(N) \setminus \llbracket b \rrbracket = \llbracket \neg b \rrbracket$, and $\llbracket b_1 \rrbracket \rightarrow \llbracket b_2 \rrbracket = \mathcal{P}(N) \setminus \llbracket b_1 \rrbracket \cup \llbracket b_2 \rrbracket = \llbracket \neg b_1 \vee b_2 \rrbracket$.*

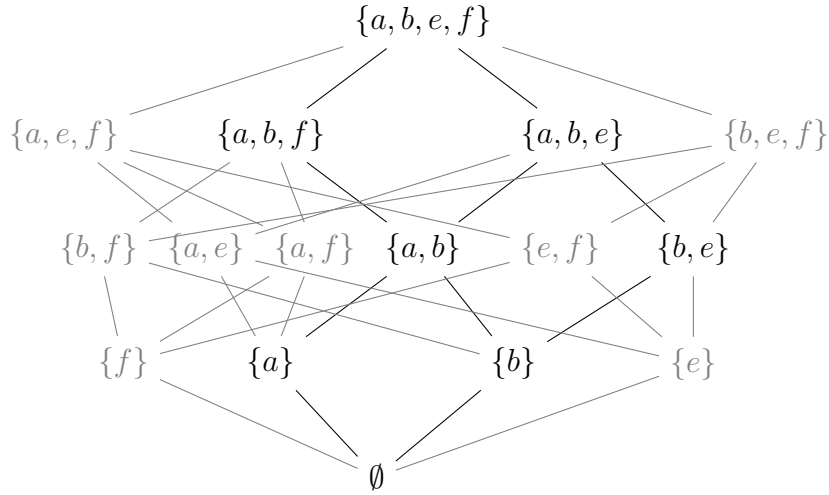
2.2.4 Embedding into Boolean Algebras and Approximation into Lattices

Using the Birkhoff duality, we can now see that it is always possible to embed a finite distributive lattice into a Boolean algebra, which can make computations, in particular for the residuum, significantly easier: Note that the Birkhoff duality naturally also holds for Boolean algebras and that the inverse of an element $\ell \in \mathbb{B}$ in a Boolean algebra \mathbb{B} can be easily identified as the element corresponding to the complement set $\mathcal{J}(\mathbb{B}) \setminus \eta(\ell)$. In this sense, all computations in Boolean algebras can be expressed in terms of set manipulations.

Proposition 2.2.15 (Embedding) *Let \mathbb{L} be a finite distributive lattice, then \mathbb{L} embeds into the Boolean algebra $\mathbb{B} = \mathcal{P}(\mathcal{J}(\mathbb{L}))$ via the monomorphism $\eta : \mathbb{L} \rightarrow \mathbb{B}$, defined as $\eta(\ell) = \{\ell' \in \mathcal{J}(\mathbb{L}) \mid \ell' \subseteq \ell\}$.*

This embedding will prove to be fruitful when computing operations on lattices, since they can be computed using well-known operations in Boolean algebras. The following example will illustrate the embedding.

Example 2.2.16 *We continue our Example 2.2.6, investigating further the finite example. We have already identified the join irreducible elements, so it is easy to construct the Boolean algebra \mathbb{B} the lattice \mathbb{L} embeds into, for it can be represented as the powerset of $\{a, b, e, f\}$ and \mathbb{L} can be embedded into \mathbb{B} by mapping a to $\{a\}$, b to $\{b\}$, e to $\{b, e\}$, f to $\{a, b, f\}$, \perp to \emptyset , \top to $\{a, b, e, f\}$, c to $\{a, b\}$ and d to $\{a, b, e\}$. Below we have visualised the Boolean algebra and the embedding of the lattice into it. The lattice is printed black, whereas the remaining elements in the Boolean algebra are grey.*



We will simply assume that $\mathbb{L} \subseteq \mathbb{B}$. Since an embedding is a lattice homomorphism, supremum and infimum coincide in \mathbb{L} and \mathbb{B} and we simply write \sqcup, \sqcap for both versions. Negation and residuum may however differ and we distinguish them via a subscript, writing $\neg_{\mathbb{L}}, \neg_{\mathbb{B}}$ and $\rightarrow_{\mathbb{L}}, \rightarrow_{\mathbb{B}}$. Given such an embedding, we can approximate elements of a Boolean algebra in the embedded lattice.

Definition 2.2.17 (Approximation) *Let \mathbb{L} be a complete distributive lattice that embeds into a Boolean algebra \mathbb{B} . Then we define the approximation of $\ell \in \mathbb{B}$ as $\lfloor \ell \rfloor_{\mathbb{L}} = \sqcup \{ \ell' \in \mathbb{L} \mid \ell' \sqsubseteq \ell \}$.*

In the sequel, we will drop the subscript \mathbb{L} and simply write $\lfloor \ell \rfloor$, if the lattice is clear from the context. For instance, in the previous example, the set of irreducibles $\{a, e, f\}$, which is not downward-closed, is approximated by $\lfloor \{a, e, f\} \rfloor = \{a\}$.

Next, we present some computational rules for approximations that are helpful in the sequel.

Lemma 2.2.18 *Let \mathbb{L} be a complete distributive lattice that embeds into a Boolean algebra \mathbb{B} . For $\ell_1, \ell_2 \in \mathbb{B}$, we have $\lfloor \ell_1 \sqcap \ell_2 \rfloor = \lfloor \ell_1 \rfloor \sqcap \lfloor \ell_2 \rfloor$ and furthermore that $\ell_1 \sqsubseteq \ell_2$ implies $\lfloor \ell_1 \rfloor \sqsubseteq \lfloor \ell_2 \rfloor$. If $\ell_1, \ell_2 \in \mathbb{L}$, then $\lfloor \ell_1 \sqcup \neg \ell_2 \rfloor = \ell_2 \rightarrow_{\mathbb{L}} \ell_1$.*

Proof: Let $\ell_1, \ell_2 \in \mathbb{B}$. Monotonicity of the approximation is immediate from the definition.

We next show $\lfloor \ell_1 \sqcap \ell_2 \rfloor \sqsupseteq \lfloor \ell_1 \rfloor \sqcap \lfloor \ell_2 \rfloor$: by definition we have $\lfloor \ell_1 \rfloor \sqsubseteq \ell_1$, $\lfloor \ell_2 \rfloor \sqsubseteq \ell_2$ and hence $\lfloor \ell_1 \rfloor \sqcap \lfloor \ell_2 \rfloor \sqsubseteq \ell_1 \sqcap \ell_2$. Since $\lfloor \ell_1 \sqcap \ell_2 \rfloor$ is the best approximation of $\ell_1 \sqcap \ell_2$ and $\lfloor \ell_1 \rfloor \sqcap \lfloor \ell_2 \rfloor$ is one approximation, the inequality follows.

In order to prove $\lfloor \ell_1 \sqcap \ell_2 \rfloor \sqsubseteq \lfloor \ell_1 \rfloor \sqcap \lfloor \ell_2 \rfloor$ observe that $\lfloor \ell_1 \rfloor \sqsupseteq \lfloor \ell_1 \sqcap \ell_2 \rfloor$ and $\lfloor \ell_2 \rfloor \sqsupseteq \lfloor \ell_1 \sqcap \ell_2 \rfloor$ by monotonicity of the approximation. Hence $\lfloor \ell_1 \sqcap \ell_2 \rfloor$ is a lower bound of $\lfloor \ell_1 \rfloor, \lfloor \ell_2 \rfloor$, which implies $\lfloor \ell_1 \rfloor \sqcap \lfloor \ell_2 \rfloor \sqsupseteq \lfloor \ell_1 \sqcap \ell_2 \rfloor$.

Now let $\ell_1, \ell_2 \in \mathbb{L}$. Recall the definitions $\lfloor \ell_1 \sqcup \neg \ell_2 \rfloor = \bigsqcup \{ \ell \in \mathbb{L} \mid \ell \sqsubseteq \ell_1 \sqcup \neg \ell_2 \}$ and $\ell_2 \rightarrow_{\mathbb{L}} \ell_1 = \bigsqcup \{ \ell \in \mathbb{L} \mid \ell_2 \sqcap \ell \sqsubseteq \ell_1 \}$. We will prove that both sets are equal.

Assume $\ell \in \mathbb{L}$ with $\ell \sqsubseteq \ell_1 \sqcup \neg \ell_2$, then $\ell_2 \sqcap \ell \sqsubseteq \ell_2 \sqcap (\ell_1 \sqcup \neg \ell_2) = (\ell_2 \sqcap \ell_1) \sqcup (\ell_2 \sqcap \neg \ell_2) = (\ell_2 \sqcap \ell_1) \sqcup \perp = \ell_2 \sqcap \ell_1 \sqsubseteq \ell_1$. For the other direction assume $\ell_2 \sqcap \ell \sqsubseteq \ell_1$, then $\ell_1 \sqcup \neg \ell_2 \sqsupseteq (\ell_2 \sqcap \ell) \sqcup \neg \ell_2 = (\ell_2 \sqcup \neg \ell_2) \sqcap (\ell \sqcup \neg \ell_2) = \top \sqcap (\ell \sqcup \neg \ell_2) = \ell \sqcup \neg \ell_2 \sqsupseteq \ell$.

□

Note that in general it does not hold that $\lfloor \ell_1 \sqcup \ell_2 \rfloor = \lfloor \ell_1 \rfloor \sqcup \lfloor \ell_2 \rfloor$ and $\lfloor \ell_1 \sqcup \neg \ell_2 \rfloor = \lfloor \ell_2 \rfloor \rightarrow_{\mathbb{L}} \lfloor \ell_1 \rfloor$ for arbitrary $\ell_1, \ell_2 \in \mathbb{B}$. To witness why these equations fail to hold, take $\ell_1 = \{a, e\}$ and $\ell_2 = \{b, f\}$ in the previous example as counterexample.

We have already discussed that a bounded distributive lattice can be viewed as a semiring. However, there is another relevant way of getting a semiring structure from a lattice, by adding an additional multiplication operation to the lattice.

Definition 2.2.19 (*l*-Monoid) Let (L, \sqcup, \sqcap) be a lattice and (L, \cdot, e) be a monoid. If \cdot distributes over \sqcup , i.e. $x \cdot (y \sqcup z) = (x \cdot y) \sqcup (x \cdot z)$ and $(x \sqcup y) \cdot z = (x \cdot z) \sqcup (y \cdot z)$ for all $x, y, z \in L$, we call (L, \sqcup, \cdot) an *l-monoid*. If (L, \sqcup, \sqcap) is complete, we call the *l-monoid complete*. It is called *completely distributive* if (L, \sqcup, \sqcap) is complete and multiplication distributes over arbitrary suprema. A complete *l-monoid* is called *integral* provided that $\top = e$.

Moreover, if L has a \perp -element 0 and $x \cdot 0 = 0 = 0 \cdot x$ for all $x \in L$, we call (L, \sqcup, \cdot) *bounded*. Every bounded *l-monoid* is a semiring $(L, \sqcup, \cdot, 0, e)$.

Here, we slightly deviate from the usual definition of *l-monoids* by not requiring the existence of a \top -element, since this is not needed for our algorithms.

Note, that every bounded distributive lattice is a bounded *l-monoid*, where $\cdot = \sqcap$. On the other hand, a bounded *l-monoid* structure can also be found for non-distributive lattices.

Completely distributive *l-monoids* are often referred to as *unital quantales*.

Example 2.2.20 An example of an *l-monoid* that we will use as a running example throughout the thesis is $\mathbb{M} = ([0, 1], \max, \cdot, 0, 1)$, which is based on the

lattice $([0, 1], \max, \min, 0, 1)$ and the order \leq . Multiplication distributes over arbitrary maxima and therefore it is a distributive l -monoid. Furthermore, it is a bounded complete l -monoid, since $\min[0, 1] = 0$ and $x \cdot 0 = 0 = 0 \cdot x$ for all $x \in [0, 1]$.

Another example of an l -monoid is the tropical semiring \mathbb{T} , based on the order \geq .

The l -monoid \mathbb{M} is isomorphic to \mathbb{T} via the isomorphism $\varphi: \mathbb{T} \rightarrow \mathbb{M}, x \mapsto 2^{-x}$.

In very much the same way as for lattices, we can also define a residuation operation for completely distributive lattices as follows:

Definition 2.2.21 *The residuation operation for a completely distributive l -monoid \mathbb{L} is defined for all $\ell_1, \ell_2 \in \mathbb{L}$ as $\ell_1 \rightarrow \ell_2 = \bigsqcup \{\ell \in \mathbb{L} \mid \ell_1 \cdot \ell \sqsubseteq \ell_2\}$, also called residuum of ℓ_1, ℓ_2 . We extend this to \mathbb{L} -vectors, replacing ℓ_1, ℓ_2 by $v_1, v_2 \in \mathbb{L}^X$.*

Example 2.2.22 *Recall the semirings \mathbb{T}, \mathbb{M} in Example 2.2.20. For $\ell_1, \ell_2 \in \mathbb{T}$ we have $\ell_1 \rightarrow \ell_2 = \min\{\ell \in \mathbb{R}_0^+ \cup \{\infty\} \mid \ell_1 + \ell \geq \ell_2\} = \ell_2 \dot{-} \ell_1$ (modified subtraction). For $\ell_1, \ell_2 \in \mathbb{M}$, we have $\ell_1 \rightarrow \ell_2 = \max\{\ell \in [0, 1] \mid \ell_1 \cdot \ell \leq \ell_2\} = \min\{1, \frac{\ell_2}{\ell_1}\}$.*

Another example where the residuation operation can be easily characterised is any Boolean algebra $(\mathbb{B}, \vee, \wedge, 0, 1)$. For $\ell_1, \ell_2 \in \mathbb{B}$ we have $\ell_1 \rightarrow \ell_2 = \neg \ell_1 \vee \ell_2$.

Similar to the semirings before, we assume throughout this thesis that all relevant operations, i.e. supremum, infimum, negation, multiplication and residuum on lattices and l -monoids are effectively computable.

2.3 State-Based Systems

The focus of this thesis lies on the analysis of state-based systems, so we want to give a brief overview over classical state-based systems, weighted automata and conditional transition systems (CTS) without upgrades. Weighted automata are the systems under investigation for Chapters 3 to 5. CTS without upgrades, in turn, are generalised by CTS (with upgrades) in Chapter 6 which are the systems under investigation in Chapters 6 and 7. For both, weighted automata and CTS, we need to consider words over an alphabet A . Formally, we will work in the

free monoid A^* where we will consider sequences of symbols $a \in A$. We call the empty sequence ε , and write a sequence of symbols $a_1, a_2, \dots, a_n \in A$ as $a_1 a_2 \dots a_n$. More detailed introductions to traditional state-based systems can be found in several textbooks, e.g. [HMRU00] for deterministic and nondeterministic automata and [Mil89a] for an introduction to bisimilarity. More on weighted automata can be found in [DKV09] and conditional transition systems without upgrades have previously been introduced in [ABH⁺12].

Definition 2.3.1 (Free Monoid) *Let A be a set, then we can define the sets A^n , $n \in \mathbb{N}_0$ according to $A^0 = \{\varepsilon\}$, $A^1 = A$, $A^{n+1} = \{aw \mid a \in A, w \in A^n\}$. Then $A^* = \bigcup_{n \in \mathbb{N}_0} A^n$. Defining the concatenation operation according to $\bullet: A^* \times A^* \rightarrow A^*$ by $w_1 \bullet w_2 = w_1 w_2$ for all $w_1, w_2 \in A^*$, we obtain a monoid structure $(A^*, \bullet, \varepsilon)$ which we call the free monoid over A .*

2.3.1 From (Non-)Deterministic Automata to Weighted Automata

(Non-)Deterministic Automata

Often, computer systems can be understood as abstract systems that are always in one of finitely many possible states and change their state upon receiving an input by a user. One or more states may signify a finished computation. This basic concept lies at the core of deterministic automata.

Definition 2.3.2 (Deterministic Automaton) *A deterministic automaton is a four tuple $M = (X, A, \delta, F)$, where*

- X is a set, which we call the set of states
- A is a finite set, which we call the set of actions
- $\delta: X \times A \rightarrow X$ is the transition function
- $F \subseteq X$ is a subset of X which we call the final states of M .

We write $x \xrightarrow{a} x'$ if $\delta(x, a) = x'$.

We are considering automata without initial states and consider the language of any state instead. This may be uncommon, but since our work is rooted in

coalgebra, where initial states are unusual, we adopt a model that does not include initial states. The language of a state $x_0 \in X$ is defined as

$$L(x_0) = \{w = a_1a_2\dots a_n \in A^* \mid \exists x_1, x_2, \dots x_n \in X : \\ \forall 0 < i \leq n : x_i = \delta(x_{i-1}, a_i) \wedge x_n \in F\}$$

While we allow infinite state sets in the definition of deterministic automata, in most applications we restrict to finite state systems.

Determinism is restricting the modelling power of automata, in particular, even for those transitions that may be considered illegal, we still need to define a successor state. Non-deterministic automata enable us to not define a successor state for a given pair of state and action, or to define more than one successor, which may be selected in a non-deterministic way. This allows for exponentially smaller automata accepting the same language as a deterministic automaton, but in turn it can make analysis of non-deterministic automata more difficult when compared to deterministic ones.

Definition 2.3.3 (Non-deterministic Automaton) *A non-deterministic automaton is a four tuple $M = (X, A, \delta, F)$, where*

- X is a set, which we call the set of states
- A is a finite set, which we call the set of actions
- $\delta: X \times A \rightarrow \mathcal{P}(X)$ is the transition function
- $F \subseteq X$ is a subset of X which we call the final states of M .

We write $x \xrightarrow{a} x'$ if $x' \in \delta(x, a)$.

The language of a state $x_0 \in X$ is defined as

$$L(x_0) = \{w = a_1a_2\dots a_n \in A^* \mid \exists x_1, x_2, \dots x_n \in X : \\ \forall 0 < i \leq n : x_i \in \delta(x_{i-1}, a_i) \wedge x_n \in F\}$$

Note that for a given word w there may exist different paths¹ from any given state, where some may end in a final state and others do not. However, as long as one path exists that is accepting, the word is in the language of the state.

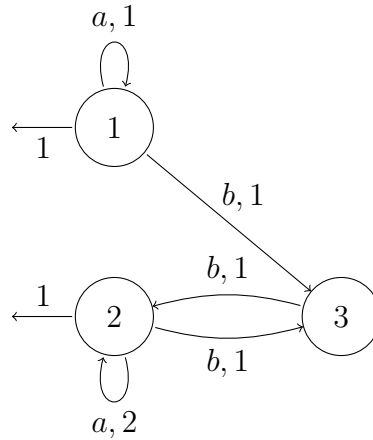
¹In this context, a path for a word w is a sequence of subsequent transitions labelled with the symbols from w .

Thus, it is not sufficient to consider a single path to determine whether a word is not in the language of a state, but instead one has to consider all paths.

Both, deterministic automata and non-deterministic automata, are well studied classical system models that are limited in expressiveness, but offer – in the case of deterministic automata even polynomial time – algorithms for most interesting constructions. However, when dealing with system models, in many cases one is not purely interested in the sequences of actions a system may accept, but additionally in quantitative information. For instance, one may wonder how long a computation takes, how many resources are used by a transition or how likely it is that a transition is taken. This can be interpreted as a weight associated with the transition.

Weighted Automata

Weighted automata can be seen as a generalisation of non-deterministic automata and differ from non-deterministic automata by a weight from a given semiring S that is assigned to each transition. Consider for instance the weighted automaton depicted in the sequel:



This automaton may be considered over the semiring $(\mathbb{N}_0, +, \cdot, 0, 1)$, which is just the natural numbers with usual addition and multiplication. All transitions carry a label over the alphabet $\{a, b\}$ and additionally a weight from \mathbb{N}_0 . If we were to ignore the weights and consequently interpret it as a non-deterministic automaton, the language of the automaton when considering all states initial would just be the set of all sequences over $\{a, b\}$ that correspond to a path in the automaton ending in final state, marked by an outgoing arrow not pointed

towards another state. For instance, any sequence of as is in the language, because from state 1 or state 2, one can input an arbitrary number of as and still end up in a final state. Moreover, any sequence of as may be interrupted by any even number of bs . In addition, a word may also start with an odd number of bs . Overall, the language of the non-deterministic automaton can be identified as ²: $L((b|\varepsilon)(a^*(bb)^*)^*)$.

Now, if we consider the weights associated with each transition, the words that do not get assigned the value 0 are exactly those that are accepted by the underlying non-deterministic automaton. However, words do not simply get accepted or rejected, but they get assigned a value from \mathbb{S} in accordance with the weights associated to the transitions. To determine the value of a word, we first must consider the weight of a path. Assume we want to compute the value of the word $w = abba$. One path that corresponds to w starts in state 1, takes an a transition to 1, then transitions via b to 3 and via an additional b to 2. Then, via another a , a transition to state 2 can be taken and, since 2 is a final state, this path is actually accepting $abba$. Along the way, the following weights were associated with the transitions we chose: 1, 1, 1, 2 and finally a 1 when terminating. These values are multiplied to obtain the value of the path, in this case 2. However, there exists another path, that also accepts $abba$, starting in 2, taking the loop to 2 with the first a , followed by a b transition to 3 and back to 1 via another b transition. The a loop in 2 allows to read the final a of the word and finally, this path is actually accepting, because 2 has a final weight of 1. Overall, we get a value of 4 for this path. To obtain the weight of the word assigned by the automaton, we must add up all the weights for paths accepting the word. Since there are only those two paths accepting $abba$, the weight assigned to $abba$ by the automaton is therefore $4 + 2 = 6$.

The language of a weighted automaton consequently is not just the set of words that get assigned a non-zero value by the automaton, but a function that maps each word to its corresponding value. In this particular case, the weighted language of the automaton assigns 0 to all words that are not in $L((b|\varepsilon)(a^*(bb)^*)^*)$, $2^{\#_a(w)}$ to all words in $L((b|\varepsilon)(a^*(bb)^*)^*)$ that start with an

²Here we use standard notation of regular expressions, i.e. a $*$ means the preceding expression may be repeated arbitrarily often (finitely, including zero) and a $|$ means alternative choice.

odd number of bs and to all remaining words of type $a^n bw \in L((b|\varepsilon)(a^*(bb)^*)^*)$ it assigns the value $(1 + 2^n) \cdot 2^{\#_a(w)}$. Analogously, the language of a state x is a function that assigns to each word the sum of the weight of all paths that start in x .

To formalise this, a weighted automaton is defined as follows:

Definition 2.3.4 (Weighted Automaton) *Let A be a finite set of alphabet symbols and X be a set of states. Then a weighted automaton is an $X \times (A \times X + 1)$ -matrix with entries from \mathbb{S} . We write $x \xrightarrow{a,s} x'$ if $\alpha(x, (a, x')) = s$.*

For a weighted automaton α , $\alpha(x, \bullet)$ denotes the final weight of state $x \in X$ and $\alpha(x, (a, y))$ denotes the weight of the a -transition from x to y . Weighted automata over l -monoids are also referred to as fuzzy automata.

We are mainly interested in weighted automata where the state set X is finite. The reason for this becomes clear when considering the definition of the language of a weighted automaton.

Definition 2.3.5 (Language of a Weighted Automaton [DKV09])

Let (X, α) be a weighted automaton over alphabet A , a semiring \mathbb{S} and a finite state set X . The language $L_\alpha : A^ \rightarrow \mathbb{S}^X$ of α is recursively defined as*

- $L_\alpha(\varepsilon)(x) = \alpha(x, \bullet)$
- $L_\alpha(aw)(x) = \sum_{x' \in X} \alpha(x, (a, x')) \cdot L_\alpha(w)(x')$ for $a \in A$, $w \in A^*$

We will call $L_\alpha(w)(x)$ the weight that state x assigns to the word w . Two states $x, y \in X$ are language equivalent if $L_\alpha(w)(x) = L_\alpha(w)(y)$ for all $w \in A^$.*

Note that the language of a (state in a) weighted automaton might not be defined if X is not finite, because weighted automata need not be finitely branching and therefore computing the language of a state may depend on computing an infinite sum, which may not be well-defined in the underlying semiring. Therefore, we will restrict our analysis to weighted automata over a finite state set throughout this thesis.

To see that weighted automata are actually a generalisation of non-deterministic automata, consider the semiring $\mathbb{B} = (\{0, 1\}, \vee, \wedge, 0, 1)$. Then the transition matrix α has entries from 0 and 1, where for indexes $x, y \in X$ and $a \in A$, $\alpha(x, (a, y)) = 1$ means that there is a transition going from x to y for

the input a . Analogously $\alpha(x, \bullet) = 1$ indicates that x is a final state. Weighted automata over the Boolean semiring can thus be translated to non-deterministic automata in a straight forward way and vice-versa.

When discussing up-to techniques for weighted automata in Chapter 5, we will employ a different way of writing weighted automata, for the sake of convenience, which will be introduced in that chapter.

2.3.2 From Labelled Transition Systems to Conditional Transition Systems

Conceptually, labelled transition systems (LTS) are very close to non-deterministic automata, however LTS are not considered as acceptors of languages, but instead are often perceived as a model for reactive systems. Thus, LTS do not have final states. Additionally, for LTS one is typically interested in a different notion of behavioural equivalence, called bisimulation, rather than language acceptance. Bisimulation is a behavioural notion that takes into consideration the choices a user may have at each point of an execution, thus making even states that are language equivalent distinguishable in some cases.

Definition 2.3.6 (Labelled Transition Systems) *A labelled transition system is a triple (X, A, \rightarrow) , where*

- X is a set which we call the set of states,
- A is a finite set which we call the set of actions and
- $\rightarrow: X \times A \rightarrow \mathcal{P}(X)$ is called the transition function.

If the set of actions is clear from the context, we omit the set of actions from the definition of an LTS and only write (X, \rightarrow) . Moreover, we write $x \xrightarrow{a} x'$ if $x' \in \rightarrow(x, a)$.

The language of a state in an LTS is defined analogous to the language of a state in a finite automaton where all states are considered final. Since LTS have no final states, language equivalence is often referred to as trace equivalence instead. However, as noted before, we are usually not interested in language equivalence, when discussing LTS, but bisimulation instead.

Definition 2.3.7 (Bisimulation) Let (X, A, \rightarrow_1) , (Y, A, \rightarrow_2) be LTS, then a relation $R \subseteq X \times Y$ is called a bisimulation if for all pairs $(x, y) \in R$ it holds that

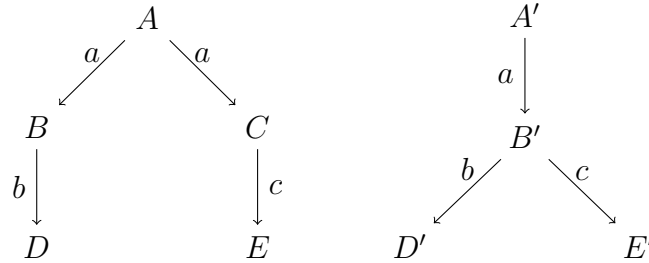
- For all $a \in A$ $x' \in X$ such that $x \xrightarrow{a}_1 x'$ there exists a $y' \in Y$ such that $y \xrightarrow{a}_2 y'$ and $(x', y') \in R$.
- For all $a \in A$ $y' \in Y$ such that $y \xrightarrow{a}_2 y'$ there exists an $x' \in X$ such that $x \xrightarrow{a}_1 x'$ and $(x', y') \in R$.

Bisimulations are closed under union and thus, there exists a greatest bisimulation, called bisimilarity, between any pair of LTS. We call two states bisimilar if there exists a bisimulation that relates the two states. To motivate bisimulation for LTS, consider the following example:

Example 2.3.8 Let

$$(\{A, B, C, D, E\}, \{a, b, c\}, \rightarrow_1), (\{A', B', D', E'\}, \{a, b, c\}, \rightarrow_2)$$

be LTS where $\rightarrow_1, \rightarrow_2$ are given graphically below:



States A and A' are language equivalent, since both states allow transitions for the words ab and ac , however they are not bisimilar. A bisimulation containing the pair (A, A') would also need to contain the pairs (B, B') and (C, B') , because from state A an a transition is possible to both, B and C and the only available a transition from A' goes to B' . However, B and B' cannot be bisimilar, because B' can do a b transition, whereas B cannot. Indeed, from a user perspective, A and A' behave differently. A well-known figurative explanation for this example is that of a coffee and tea machine. Action a may be interpreted as inputting a coin, whereas b may be interpreted as choosing coffee and c may be interpreted as choosing tea. In state A the user can input a coin, but then the system

non-deterministically decides, whether to offer tea or coffee and the user can only accept the machine's choice. On the other hand, in A' after inputting a coin, the user may choose on his own whether to get coffee or tea.

The greatest bisimulation can be computed using a partition refinement algorithm, which lies at the core of both, the coalgebraic algorithm to compute behavioural equivalence (cf. Chapter 3) and the matrix multiplication to compute the greatest bisimulation for a CTS (cf. Chapter 6) :

Algorithm 2.3.9 (Partition Refinement Algorithm for Bisimilarity)

Let (X, A, \rightarrow_1) and (Y, A, \rightarrow_2) be two LTS over the same label alphabet, then the following algorithm computes the greatest bisimulation between the two systems.

- *Start by setting $R_0 = X \times Y$*
- *As long as $R_i \neq R_{i+1}$ compute $R_{i+1} \subseteq X \times Y$ based on R_i as follows:*
 - $(x, y) \in R_{i+1}$ if and only if*
 - *$(x, y) \in R_i$*
 - *For all $x' \in X$ such that $x \xrightarrow{a}_1 x'$ there exists a $y' \in Y$ such that $y \xrightarrow{a}_2 y'$ and $(x', y') \in R_i$*
 - *For all $y' \in Y$ such that $y \xrightarrow{a}_2 y'$ there exists an $x' \in x$ such that $x \xrightarrow{a}_1 x'$ and $(x', y') \in R_i$*

Alternatively, bisimulation can also be characterised by a bisimulation game. We will state the traditional bisimulation game for LTS here, because it is similar in nature to the bisimulation game we will later discuss for CTS.

Definition 2.3.10 (Bisimulation Game) *Bisimulation can be characterised using the following game. Given two LTSs (X, A, \rightarrow_1) and (Y, A, \rightarrow_2) , a state $x \in X$ and a state $y \in Y$ the bisimulation game is a round-based two-player game that uses both the LTSs as game boards. Let (x, y) be a game instance indicating that there is a marked state in each of X and Y at any given time. The game progresses to the next game instance as follows:*

- *Player 1 is the first one to move. Player 1 can choose either the marked state $x \in X$ or $y \in Y$ and must perform a transition $x \xrightarrow{a}_1 x'$ or $y \xrightarrow{a}_2 y'$ for some $a \in A$.*

- Player 2 then has to simulate the last step, i.e., if Player 1 made a step $x \xrightarrow{a}_1 x'$, Player 2 is required to make step $y \xrightarrow{a}_2 y'$ and vice-versa.
- In turn, the new game instance is (x', y') .

Player 1 wins if Player 2 cannot simulate the last step performed by Player 1. Player 2 wins if the game never terminates or Player 1 cannot make another step.

The correspondence between bisimulation and bisimulation games is that whenever Player 1 has a winning strategy, the initial pair of states is not bisimilar and whenever Player 2 has a winning strategy, the initial pair of states is bisimilar. Note, however, that winning a specific game against a non-optimal opponent does not prove any claim about bisimilarity either way.

System models can often be designed with numerous products in mind, but with a common code base. Moreover, some systems may behave differently, when put into different contexts, e.g. the same system model may be used to model a coffee machine that only offers coffee and one that also offers the choice of tea. In order to be able to model and analyse transition systems that offer different configurations, conditional transition systems (CTS) were proposed in [ABH⁺12]. Each transition in a CTS, when compared to LTS, is additionally equipped with a condition, which is a subset of a finite set of products over which the CTS is defined. When a CTS is executed, first an atom of the Boolean algebra must be selected, which instantiates the CTS to an LTS that contains all those transitions of the CTS that carry a guard greater than the selected atom. Formally a CTS is defined as follows:

Definition 2.3.11 (CTS (without upgrades)) *Let Φ be a finite set. Then, a conditional transition system (CTS) is a triple (X, A, f) consisting*

- a set of states X ,
- a finite set A called the label alphabet and
- a function $f: X \times A \rightarrow (\Phi \rightarrow \mathcal{P}(X))$ mapping every ordered pair in $X \times A$ to a function of type $\Phi \rightarrow \mathcal{P}(X)$.

We call the elements of Φ the conditions of the CTS. As usual, we write $x \xrightarrow{a, \varphi} y$ whenever $y \in f(x, a)(\varphi)$.

Two states from two CTS are bisimilar if they are bisimilar under all conditions, i.e. bisimilar in the LTSs one obtains when instantiating to each condition. In Chapters 6 and 7 we will explore a generalisation of CTS which offers the option to perform a change in condition during run-time. CTS are strongly related to featured transition systems (FTS), where instead of conditions each transition is labelled by a set of features, that a system may or may not have. Similar to CTS, upon start a set of features is set and transitions are activated or deactivated accordingly. As we will discuss in more detail in Chapter 6, FTS correspond exactly to CTS without upgrades (with a set of conditions whose cardinality is a power of two), where the powerset of all features acts as the set of conditions.

2.4 Category Theory and Coalgebra

We are now turning our attention towards category theory and coalgebra. In this section we will give an overview over all notions concerning category theory and coalgebra that will be required throughout this thesis. Even though we aim at giving a thorough introduction to those concepts that we are going to use, this is of course only a small look onto the basics of category theory. For a more complete introduction, the reader may be referred to e.g. [AHS90, ML71].

2.4.1 Categories and Morphisms

Category theory is a mathematical theory that aims at finding common structures in different fields that are linked by the property that they are all concerned with morphisms that are a mapping between certain objects. The idea is to identify core concepts that can be found in many different fields that would otherwise be viewed as unconnected. Thus, category theory aims to provide an abstract view on morphism that can help identifying the fundamentals of different constructions.

Definition 2.4.1 (Category) *A category $\mathbf{C} = (\text{Obj}, \text{Arr}, \circ, \text{id})$ consists of*

- *A class Obj whose members we call objects.*

- A class \mathbf{Arr} whose members we call arrows or morphisms. Each $f \in \mathbf{Arr}$ has a domain $\text{dom}(f) \in \mathbf{Obj}$ and a codomain $\text{cod}(f) \in \mathbf{Obj}$ and we write $f: \text{dom}(f) \rightarrow \text{cod}(f)$.
- A (partial) composition operation $\circ: \mathbf{Arr} \times \mathbf{Arr} \rightarrow \mathbf{Arr}$ that allows to concatenate arrows $f: A \rightarrow B$ and $g: B \rightarrow C$ to obtain an arrow $g \circ f: A \rightarrow C$. The operation \circ is required to be associative in the following sense: Given three arrows $f: A \rightarrow B$ and $g: B \rightarrow C$, $h: C \rightarrow D$, it must hold that $(h \circ g) \circ f = h \circ (g \circ f)$.
- A class of identity arrows id such that for each object $A \in \mathbf{Obj}$ there exists an arrow id_A and for all objects B and all arrows $f: A \rightarrow B$, $g: B \rightarrow A$ it holds that $\text{id}_B \circ f = f$ and $g \circ \text{id}_A = g$.

There exist several different ways of defining categories, all of which are equivalent: One may remove the objects from the definition and identify them with their identity arrows instead or one may define the arrows as classes $\mathbf{Arr}(A, B)$ for each pair of objects A, B . Throughout this thesis we will adhere to the notation given in the above definition though.

Sometimes we will consider more than one category at a time. The components, i.e. objects, arrows, composition and units, may then be distinguished by an index indicating the category they belong to. If there is no danger of confusion, we will still forego making explicit, for instance which category a concatenation operation belongs to.

A motivating example for the definition of a category is the category **Set**, which will play an important role in all our category theoretic endeavours.

Example 2.4.2 (Set) *The category **Set** is defined as follows:*

Objects *The objects are all sets.*

Arrows *The arrows are all functions.*

Concatenation *Concatenation is function composition, i.e. $f: A \rightarrow B$, $g: B \rightarrow C$ can be composed as $g \circ f: A \rightarrow C$ according to $g \circ f(x) = g(f(x))$ for all $x \in A$.*

Identity *The identity arrows $\text{id}_A: A \rightarrow A$ are just the identity functions where $\text{id}_A(x) = x$ for all $x \in A$.*

A closely related category is **Poset**, the category of partially ordered sets and order preserving functions:

Example 2.4.3 (Poset) *The category **Poset** is defined as follows:*

Objects *The objects are all pairs (M, \leq_M) of sets M partially ordered by \leq_M .*

Arrows *The arrows are all order-preserving functions (cf. Definition 2.2.2).*

Concatenation *Concatenation is function composition, i.e. $f: (A, \leq_A) \rightarrow (B, \leq_B)$, $g: (B, \leq_B) \rightarrow (C, \leq_C)$ can be composed as $g \circ f: (A, \leq_A) \rightarrow (C, \leq_C)$ according to $g \circ f(x) = g(f(x))$ for all $x \in (A, \leq_A)$.*

Identity *The identity arrows $\text{id}_{(A, \leq_A)}: (A, \leq_A) \rightarrow (A, \leq_A)$ are just the identity functions where $\text{id}_{(A, \leq_A)}(x) = x$ for all $x \in A$.*

Even though our first two examples of categories both use sets and functions as objects and arrows, this need not be true in general, the following example serves to give an impression of how versatile the notion of a category is. Different from the previous two examples, these two categories will not be important in the main chapters of this thesis.

Example 2.4.4

- *Let (M, \leq_M) be any partially ordered set, then M can be considered as a category in the following way: The objects are all elements $m \in M$, between two elements $m, m' \in M$ there exists an arrow $f: m \rightarrow m'$ if and only if $m \leq_M m'$. Concatenation is given by transitivity and, since partial orders are reflexive, this also yields an identity arrow id_m for all $m \in M$.*
- **Rel** *has as objects all sets and morphisms $f: A \rightarrow B$ are relations $f \subseteq A \times B$. The identity arrow on a set A is the identity relation given by $\text{id}_A = \{(x, x) \mid x \in A\}$ and concatenation of any two arrows $f: A \rightarrow B$, $g: B \rightarrow C$ is relation composition, i.e. $g \circ f: A \rightarrow C$ is the relation given by $g \circ f = \{(a, c) \mid \exists b \in B : (a, b) \in f, (b, c) \in g\}$.*

An important general construction of a new category from any given category is the $^{\text{op}}$ category.

Example 2.4.5 Let \mathbf{C} be any category. Then \mathbf{C}^{op} defined according to:

Objects The objects of \mathbf{C}^{op} are just $\text{Obj}_{\mathbf{C}}$.

Arrows \mathbf{C}^{op} contains one arrow $f^{\text{op}}: Y \rightarrow X$ for each arrow $f: X \rightarrow Y \in \text{Arr}(\mathbf{C})$.

Composition Given two arrows $f^{\text{op}}: Y \rightarrow X$, $g^{\text{op}}: Z \rightarrow Y$, the arrow $f^{\text{op}} \circ g^{\text{op}}: Z \rightarrow X$ is defined as $(g \circ f)^{\text{op}}$.

Identities The identity arrow of any object $X \in \text{Obj}_{\mathbf{C}^{\text{op}}}$ is id_X^{op} .

is a category as well.

When discussing arrows, three distinct types of arrows are of particular importance: Monomorphisms, epimorphisms and isomorphisms.

Definition 2.4.6 (Mono-, Epi- and Isomorphisms) Let \mathbf{C} be any category and $f \in \text{Arr}$ be any arrow of \mathbf{C} . Then we call $f: A \rightarrow B$

- a monomorphism (or *mono*) if for all pairs of arrows $g, h: C \rightarrow A$ it holds that $f \circ g = f \circ h$ implies $g = h$.
- an epimorphism (or *epi*) if for all pairs of arrows $g, h: B \rightarrow C$ it holds that $g \circ f = h \circ f$ implies $g = h$.
- an isomorphism (or *iso*) if there exists an arrow $f^{-1}: B \rightarrow A$ such that $f^{-1} \circ f = \text{id}_A$ and $f \circ f^{-1} = \text{id}_B$. We then call f^{-1} the inverse of f and call A and B isomorphic. Additionally, if for any arrows $f: X \rightarrow Y$, $g: X \rightarrow Z$ there exists an isomorphism $i: Y \rightarrow Z$ such that $i \circ f = g$, we call f and g isomorphic.

Monomorphisms generalise injective functions, epimorphisms generalise surjective functions and isomorphisms generalise bijective functions. Indeed, the epis in **Set** are the surjective functions, the monos are the injective functions and the isos are the bijective functions. Note, that in general an arrow that is monomorphic and epimorphic need not be isomorphic. However, isomorphisms are always monomorphisms and epimorphisms. This can be observed by our Example 2.4.4. Given a non-trivially partially ordered set (M, \leq_M) , we can interpret this partially ordered set as a category. In this category, for each

pair of objects, i.e. elements $m, m' \in M$, there either exists no arrow from m to m' , in case $m \not\leq_M m'$, or there exists exactly one arrow from m to m' , in case $m \leq_M m'$. Thus, all arrows are both epimorphisms and monomorphisms. However, due to antisymmetry of partial orders, only the identity arrows are isomorphisms.

Finally, we also want to identify one special kind of object, the final object.

Definition 2.4.7 (Final Object) *Let \mathbf{C} be any category. An object Y of \mathbf{C} is final if for every object X in \mathbf{C} there exists a unique arrow $f: X \rightarrow Y$.*

Final objects need not always exist, for instance the category with two objects and only identity arrows has no final object, since there is no arrow connecting the two distinct objects. On the other hand, final objects need not be unique. For instance, in **Set** and **Poset**, the final objects are all one-element sets (the order is uniquely given in **Poset** for a single-element set). However, final objects naturally are unique up to isomorphism. Unique up to isomorphism – a uniqueness property which we will use often when arguing category theoretically – means, that whenever two entities (e.g. objects or arrows) exist that have a property, they are connected by an isomorphism. In the case of final objects, this can easily be checked as follows: Assume there are two final objects X, Y in a category \mathbf{C} . Then there must exist a unique arrow $f: X \rightarrow Y$ because Y is final, also there must be a unique arrow $g: Y \rightarrow X$ because X is final. Then $f \circ g: Y \rightarrow Y$ must be equal to id_Y because Y is final. Analogously $g \circ f = \text{id}_X$. Thus, f and g are each other's inverses and therefore the final object is unique up to isomorphism. If an entity satisfying a property P is unique up to isomorphism, we often call it 'the entity' instead of 'an entity' satisfying P , e.g. we say 'the final object' even though several final objects may exist.

2.4.2 Functors

Since morphisms are the primary object of interest in category theory, it is only natural to consider morphisms between categories. This role of 'mapping categories to categories' is taken by the functor.

Definition 2.4.8 (Functor) *Let \mathbf{C} and \mathbf{D} be categories. A functor $F : \mathbf{C} \rightarrow \mathbf{D}$ maps objects from \mathbf{C} to objects from \mathbf{D} and arrows from \mathbf{C} to arrows of \mathbf{D} in the following way:*

- *Let $f : A \rightarrow B \in \text{Arr}_{\mathbf{C}}$. Then $Ff : FA \rightarrow FB \in \text{Arr}_{\mathbf{D}}$.*
- *F preserves identities, i.e. for all objects $A \in \text{Obj}_{\mathbf{C}}$, $F\text{id}_A = \text{id}_{FA}$.*
- *F respects composition, i.e. for all arrows $f : A \rightarrow B \in \text{Arr}_{\mathbf{C}}$, $g : B \rightarrow C \in \text{Arr}_{\mathbf{C}}$, $F(g \circ_{\mathbf{C}} f) = Fg \circ_{\mathbf{D}} Ff$.*

If $\mathbf{C} = \mathbf{D}$, we call F an endofunctor.

The most basic functor is the identity functor:

Definition 2.4.9 (Identity Functor) *Let \mathbf{C} be any category, then we can define the identity functor $\text{id}_{\mathbf{C}}$ according to $\text{id}_{\mathbf{C}}(X) = X$ for all objects $X \in \text{Obj}_{\mathbf{C}}$ and $\text{id}_{\mathbf{C}}(f) = f$ for all arrows $f \in \text{Arr}_{\mathbf{C}}$.*

We will now consider an example of a **Set**-endofunctor that is a prominent example in particular in coalgebraic modelling.

Definition 2.4.10 (Powerset Functor) *The powerset functor $\mathcal{P} : \mathbf{Set} \rightarrow \mathbf{Set}$ is defined according to*

Objects *Let X be any object of \mathbf{Set} , i.e. a set, then $\mathcal{P}X = \{X' \subseteq X\}$ is the powerset of X .*

Arrows *Let $f : X \rightarrow Y$ be any arrow in \mathbf{Set} , i.e. a function, then $\mathcal{P}f : \mathcal{P}X \rightarrow \mathcal{P}Y$ is defined for all $X' \subseteq X$ according to*

$$\mathcal{P}f(X') = \{f(x) \mid x \in X'\}$$

Several variations of the powerset functor exist. Probably the most prominent one being the finite powerset functor \mathcal{P}_{fin} , which differs only in the definition for objects, where the definition changes to $\mathcal{P}_{\text{fin}}X = \{X' \subseteq X \mid |X'| < \infty\}$. Furthermore, similar functors can be defined on **Poset** by choosing a suitable order on $\mathcal{P}(X)$, e.g. subset ordering.

As previously stated, the category **Set** will play an important role throughout this thesis, even when working with categories different from **Set**. The

connection to **Set** naturally comes from a functor, namely the concretisation functor, mapping to **Set**.

Definition 2.4.11 (Concrete Category)

- A functor $F: \mathbf{C} \rightarrow \mathbf{D}$ is called *faithful* if, for all pairs of arrows $f: X \rightarrow Y$, $g: X \rightarrow Y$ with common domain and codomain, $Ff = Fg$ implies $f = g$. I.e. F is injective on the class of arrows between a given pair of objects.
- Let \mathbf{C} be a category. If there exists a faithful functor $U: \mathbf{C} \rightarrow \mathbf{Set}$, we call (\mathbf{C}, U) a *concrete category*.

Note, that faithful functors may map two different arrows to the same arrow, provided their domain or codomain differs. It is important to specify the concretisation functor for a concrete category, because concretisation functors need not be unique. We will however omit U at times when it is clear from the context which concretisation functor we are referring to. Concretisation functors need not exist for all categories, however, throughout this thesis, all categories we will consider are concrete categories.

Example 2.4.12

- The category **Set**, together with the identity functor $\text{Id}: \mathbf{Set} \rightarrow \mathbf{Set}$ is concrete.
- Moreover, the category **Poset** can be equipped with a concretisation functor U that is often called the *forgetful functor* and acts by just dropping the order, so $U(X, \leq_X) = X$ for any partially ordered set (X, \leq_X) and $Uf = f$ for any monotone function $f \in \mathbf{Arr}_{\mathbf{Poset}}$. Observe that this example of a concretisation functor is not injective on arrows in general. E.g. the identity arrow on the set X equipped with two different orders is mapped to the same arrow in **Set**, the identity function on X .
- Interpreting a partially ordered set (M, \leq_M) as a category, we can find different ways of identifying the category as concrete. For one, we could map each element $m \in M$ to the set $Um = \{m\} \in \mathbf{Obj}_{\mathbf{Set}}$ and each arrow $f: m \rightarrow m' \in \mathbf{Arr}_{(\mathbf{M}, \leq_M)}$ to the (unique) function $Uf: \{m\} \rightarrow \{m'\}$.

Alternatively, we could choose any $\bullet \notin M$ and map each element $m \in M$ to the set $U'm = \{m, \bullet\} \in \mathbf{Obj}_{\mathbf{Set}}$ and each arrow $f: m \rightarrow m' \in \mathbf{Arr}_{(\mathbf{M}, \leq_{\mathbf{M}})}$ to the constant function $U'f: \{m, \bullet\} \rightarrow \{m', \bullet\}$ mapping every element to \bullet .

- Finally, the category **Rel** can be identified as concrete as well, via the functor $U: \mathbf{Rel} \rightarrow \mathbf{Set}$ defined according to $U(X) = \mathcal{P}(X)$ for all objects, i.e. sets, $X \in \mathbf{Obj}_{\mathbf{Rel}}$ and for all arrows, i.e. relations $f: X \rightarrow Y \in \mathbf{Arr}_{\mathbf{Rel}}$, and all $X' \leq X$, we define

$$Uf(X') = \{y \in Y \mid \exists x \in X' : (x, y) \in f\}$$

Similar to morphisms, functors can also be composed, yielding another functor.

Definition 2.4.13 (Composition Functor) *Let $\mathbf{C}, \mathbf{D}, \mathbf{E}$ be categories and $F: \mathbf{C} \rightarrow \mathbf{D}$ and $G: \mathbf{D} \rightarrow \mathbf{E}$ be functors. Then the composition functor $G \circ F$ is defined according to:*

Objects *Objects $X \in \mathbf{Obj}_{\mathbf{C}}$ are mapped to $GFX \in \mathbf{Obj}_{\mathbf{E}}$*

Arrows *Arrows $f \in \mathbf{Arr}_{\mathbf{C}}$ are mapped to $G F f \in \mathbf{Arr}_{\mathbf{E}}$*

Since the collection of all categories, together with their functors, the identity functors and functor composition mimic exactly the definition of a category, it is only natural to extend the notion of isomorphic objects also to categories. Note that calling this structure itself a category poses problems similar to those from set theory when allowing a set that contains all sets. The reason being that the objects need not form a class. However, if we restrict to the categories where the classes of objects and arrows are just sets, this structure can be identified as a category. Note that this category itself does *not* have only a set of objects and arrows and is thus not an object of itself.

Definition 2.4.14 (Isomorphic Categories) *Two categories \mathbf{C} and \mathbf{D} are said to be isomorphic if there exists a functor $F: \mathbf{C} \rightarrow \mathbf{D}$ and a functor $G: \mathbf{D} \rightarrow \mathbf{C}$ such that $F \circ G = \text{Id}_{\mathbf{D}}$ and $G \circ F = \text{Id}_{\mathbf{C}}$.*

We will soon see a prominent example of isomorphic categories.

2.4.3 Natural Transformations and Monads

Many category theoretic theorems can be illustrated by means of a diagram, which is a directed graph, where the nodes are labelled by objects and the edges are labelled by arrows. To say a diagram commutes means to say that for all directed paths in the graph from the same starting node to the same end node, the concatenation of the labels on the path are equal. We will demonstrate this in the following definition of a natural transformation.

As stated previously, the main focus in category theory lies on the study of morphisms. We have already seen that this extends to considering morphisms between morphisms, which we called functors. Adding another layer on top, we are also interested in a notion of morphisms between functors, which is given by natural transformations.

Definition 2.4.15 (Natural Transformation) *Let $F: \mathbf{C} \rightarrow \mathbf{D}$ and $G: \mathbf{C} \rightarrow \mathbf{D}$ be two functors between the same pair of categories \mathbf{C} and \mathbf{D} . A natural transformation $\alpha: F \Rightarrow G$ from F to G is a family of arrows α_A , one for each object A in \mathbf{C} , such that for all arrows $f: X \rightarrow Y$ the following diagram commutes, i.e. it holds that $Gf \circ \alpha_X = \alpha_Y \circ Ff$. The arrows α_A of α are called its components.*

$$\begin{array}{ccc} FX & \xrightarrow{\alpha_X} & GX \\ Ff \downarrow & & \downarrow Gf \\ FY & \xrightarrow{\alpha_Y} & GY \end{array}$$

If all components of a natural transformation α are isomorphisms, we call α a natural isomorphism.

Very similar to functors before, natural transformations give rise to a categorical structure³, where the objects are all functors between a given pair of categories \mathbf{C} , \mathbf{D} and the arrows are natural transformations. The identity natural transformation id_F for any given functor $F: \mathbf{C} \rightarrow \mathbf{D}$ has, for each $X \in \text{Obj}_{\mathbf{C}}$ as its components the identity arrows $\alpha_X = \text{id}_{FX}$ of \mathbf{D} . The composition of natural transformations is defined in the following way:

³This structure again need not be a category itself, in general. However, if the classes of objects and arrows of \mathbf{C} and \mathbf{D} are just sets, it is a category

Definition 2.4.16 (Composition of Natural Transformations) *Let \mathbf{C}, \mathbf{D} be categories and $F, G: \mathbf{C} \rightarrow \mathbf{D}$ be functors. Furthermore, let $\alpha: F \Rightarrow G$ and $\beta: G \Rightarrow H$ be natural transformations. Then the composition of α and β , $\beta \circ \alpha: F \Rightarrow H$ is a natural transformation, where the component $(\beta \circ \alpha)_X: FX \rightarrow HX$ is defined for all objects $X \in \mathbf{Arr}_{\mathbf{C}}$ according to $(\beta \circ \alpha)_X = \beta_X \circ \alpha_X$.*

Natural transformations are the building blocks of monads, which can be seen as a category theoretical analogue to a monoid.

Definition 2.4.17 (Monad) *Let \mathbf{C} be a category. A monad on \mathbf{C} is a triple (T, η, μ) , where $T: \mathbf{C} \rightarrow \mathbf{C}$ is an endofunctor and $\eta: \text{Id} \Rightarrow T$, $\mu: T^2 \Rightarrow T$, where $T^2 = T \circ T$, are natural transformations such that the following diagrams commute, i.e. for all objects $X \in \mathbf{Obj}_{\mathbf{C}}$ it must hold that $\mu_X \circ \mu_{TX} = \mu_X \circ T\mu_X$ and $\mu_X \circ \eta_{TX} = \text{id}_{TX} = \mu_X \circ T\eta_X$. We call the first law the associative law and the second law the unit law.*

$$\begin{array}{ccc}
 T^3 & \xrightarrow{T\mu} & T^2 \\
 \mu T \downarrow & & \downarrow \mu \\
 T^2 & \xrightarrow{\mu} & T
 \end{array}
 \qquad
 \begin{array}{ccc}
 T & \xrightarrow{\eta T} & T^2 \\
 T\eta \downarrow & \searrow \text{id}_T & \downarrow \mu \\
 T^2 & \xrightarrow{\mu} & T
 \end{array}$$

We call η the unit of (T, η, μ) and μ the multiplication of (T, η, μ) . In the sequel, we may identify a monad with its functor if unit and multiplication are clear from the context.

The analogy to a monoid can already be derived from the naming of the natural transformations, namely the unit plays the role of the identity element in a monad and the multiplication can be seen as the analogue to the monoid operation. We will now consider two examples of monads on **Set**.

Definition 2.4.18 (Reader Monad) *Let Φ be a set. Then, for an object $X \in \mathbf{Obj}_{\mathbf{Set}}$ and an arrow $f \in \mathbf{Arr}_{\mathbf{Set}}$, we define the reader monad $(_{}^\Phi, \eta, \mu)$ where the functor $_{}^\Phi$ is defined as*

Objects *For any object $X \in \mathbf{Obj}_{\mathbf{Set}}$ we define $X^\Phi = \{C: \Phi \rightarrow X\}$*

Arrows *For any arrow $f: X \rightarrow Y$, we define $f^\Phi: X^\Phi \rightarrow Y^\Phi$ where for all $C \in X^\Phi$, we define $f^\Phi(C) = f \circ C$.*

Furthermore, the unit $\nu_X : X \rightarrow X^\Phi$ is given as $\nu_X(x)(\varphi) = x$ for all $x \in X$. Lastly, the multiplication $\zeta_X : (X^\Phi)^\Phi \rightarrow X^\Phi$ is defined as $\zeta_X(\psi)(\varphi) = \psi(\varphi)(\varphi)$, for any $\psi \in (X^\Phi)^\Phi$, $\varphi \in \Phi$.

It is easy to see that the reader monad is indeed a monad.

- $_^\Phi$ is a functor: Preservation of identities follows from the definition of identities: $\text{id}_X^\Phi(C) = \text{id}_X \circ C = C$, the fact that $_^\Phi$ respects concatenation is similarly simple: $(g \circ f)^\Phi(C) = (g \circ f)(C) = g(f(C)) = g(f^\Phi(C)) = g^\Phi \circ f^\Phi(C)$.
- ν is a natural transformation: For any function $f : X \rightarrow Y$, $x \in X$, $\varphi \in \Phi$ it holds that $\nu_Y \circ f(x)(\varphi) = f(x) = f(\nu_X(x)(\varphi)) = f^\Phi \circ \nu_X(x)(\varphi)$.
- ζ is a natural transformation: For any function $f : X \rightarrow Y$, any function $\psi : \Phi \rightarrow X^\Phi$ and any $\varphi \in \Phi$ it holds that $f^\Phi \circ \zeta_X(\psi)(\varphi) = f(\zeta_X(\psi)(\varphi)) = f(\psi(\varphi)(\varphi)) = \zeta_Y \circ ((f^\Phi)^\Phi(\psi))(\varphi)$
- The associative law holds: Let $C \in T^3X$, $\varphi \in \Phi$ be given, then $\zeta_X \circ \zeta_X^\Phi(C)(\varphi) = (\zeta_X^\Phi(C)(\varphi))(\varphi) = C(\varphi)(\varphi)(\varphi) = \zeta_{X^\Phi}(C)(\varphi)(\varphi) = \zeta_X \circ \zeta_{X^\Phi}(C)(\varphi)$.
- The unit law holds: Let $C \in X^\Phi$ and $\varphi \in \Phi$, then $\zeta_X \circ \nu_{X^\Phi}(C)(\varphi) = \nu_{X^\Phi}(C)(\varphi)(\varphi) = C(\varphi) = \nu_X(C(\varphi))(\varphi) = \nu_X^\Phi(C)(\varphi)(\varphi) = \zeta_X \circ \nu_X^\Phi(C)(\varphi)$.

Definition 2.4.19 (Powerset Monad) *The powerset functor \mathcal{P} gives rise to a monad as well, where the component $\eta_X : X \rightarrow \mathbb{P}(X)$ for any object X of the unit η is defined by $\eta_X(x) = \{x\}$ for any $x \in X$ and the component $\mu_X : \mathcal{P}\mathcal{P}X \rightarrow \mathcal{P}X$ is given by $\mu_X(X') = \bigcup X'$ for any $X' \subseteq \mathcal{P}X$.*

Monads are of particular interest, because they give rise to new categories, so-called Kleisli categories, in the following way.

Definition 2.4.20 (Kleisli Category) *Let (T, η, μ) be a monad on a category \mathbf{C} where η is the unit and μ the multiplication. Then its Kleisli category $\mathbf{Kl}(T)$ is defined as follows:*

Objects *The objects are just $\text{Obj}_{\mathbf{C}}$*

Arrows For each arrow $f: X \rightarrow TY \in \mathbf{Arr}_{\mathbf{C}}$ there is an arrow $f_T: X \rightarrow Y \in \mathbf{Arr}_{\mathbf{Kl}(T)}$

Identity The identity arrows id_X in $\mathbf{Kl}(T)$ for any object X are given by η_X

Composition the composition of two arrows $f_T: X \rightarrow Y$, $g_T: Y \rightarrow Z$ in $\mathbf{Kl}(T)$ is given by $\mu_Z \circ Tg \circ f$ in \mathbf{C} .

We will often identify $f_T: X \rightarrow Y$ in $\mathbf{Kl}(T)$ and $f: X \rightarrow TY$ in \mathbf{C} and just state, when necessary, in which category the arrow is interpreted. We have already seen a category that is (isomorphic to) a Kleisli category, namely the Kleisli category for the powerset functor on \mathbf{Set} , which is isomorphic to \mathbf{Rel} . An arrow $f: X \rightarrow Y$ in $\mathbf{Kl}(\mathcal{P})$ corresponds to a function $f: X \rightarrow \mathcal{P}(Y)$. A function of type $X \rightarrow \mathcal{P}(Y)$ can be interpreted as a relation $R_f = \{(x, y) \mid x \in X, y \in f(x)\}$ and, vice versa, any relation $R \subseteq X \times Y$, i.e. arrow of type $X \rightarrow Y$ in \mathbf{Rel} , can be interpreted as a function $f_R: X \rightarrow \mathcal{P}(Y)$ where $f_R(x) = \{y \in Y \mid (x, y) \in R\}$. This way we can identify an isomorphism between $\mathbf{Kl}(\mathcal{P})$ and \mathbf{Rel} by mapping each set to itself and arrows as indicated above.

Functors in Kleisli categories are often obtained as extensions of functors in their base categories. For this purpose a special natural transformation, called a distributive law is needed.

Definition 2.4.21 (Distributive Law, Extension) Let \mathbf{C} be a category, (T, η, μ) a monad on \mathbf{C} and F an endofunctor on \mathbf{C} . Then a natural transformation $\lambda: TF \Rightarrow FT$ is called a distributive law in case the following diagrams commute:

$$\begin{array}{ccc}
 F & & \\
 \eta F \downarrow & \searrow F\eta & \\
 TF & \xrightarrow{\lambda} & FT
 \end{array}
 \qquad
 \begin{array}{ccccc}
 T^2F & \xrightarrow{T\lambda} & TFT & \xrightarrow{\lambda T} & FT^2 \\
 \mu F \downarrow & & & & \downarrow F\mu \\
 TF & \xrightarrow{\lambda} & & & FT
 \end{array}$$

Then, the functor \overline{F} that acts as $\overline{F}X = FX$ on objects X of $\mathbf{Kl}(T)$ and as $\overline{F}f_T = \lambda_X \circ Ff$ on arrows $f_T: X \rightarrow Y$ of $\mathbf{Kl}(T)$ is called the extension of F to $\mathbf{Kl}(T)$.

2.4.4 Coalgebra

We now want to put the categorical framework to use by modelling state-based systems. For that purpose, we need coalgebras. A coalgebra is an arrow $\alpha: X \rightarrow FX$ where the endofunctor F describes the branching type of the system under consideration, X plays the role of the set of states and α specifies transitions.

Definition 2.4.22 (Coalgebra) *Let $F: \mathbf{C} \rightarrow \mathbf{C}$ be an endofunctor on a category \mathbf{C} . An F -coalgebra is a pair $(X, \alpha: X \rightarrow FX)$, where X is an object of \mathbf{C} and α is an arrow in \mathbf{C} .*

Given two F -coalgebras (X, α) , (Y, β) , a coalgebra homomorphism from (X, α) to (Y, β) is a \mathbf{C} -arrow $f: X \rightarrow Y$, so that the diagram below commutes, i.e. $Ff \circ \alpha = \beta \circ f$.

$$\begin{array}{ccc} X & \xrightarrow{\alpha} & FX \\ f \downarrow & & \downarrow Ff \\ Y & \xrightarrow{\beta} & FY \end{array}$$

Coalgebra homomorphisms can be considered as structure-preserving maps between coalgebras, they correspond to functional bisimulations. F denotes the branching type of the system. For instance, the powerset functor yields non-deterministic branching. The distribution functor, that maps sets to probability distributions, can be used for probabilistic branching and given any semiring \mathbb{S} , the functor mapping each set X to a function $X \rightarrow \mathbb{S}$ expresses weighted branching. Other prominent examples of **Set**-functors used in the coalgebraic context are polynomial functors, i.e. those that act on sets as any combinations of cross product and disjoint union, the former being used to model input and the latter being used to model a system with output. So, for instance, given a set X , it may be mapped to $A \times X$ for a fixed set A by the functor $A \times _$, so each transition corresponds to picking one input symbol $a \in A$ and a corresponding successor state. Analogously, given a functor that maps any set X to $X + B$, for a fixed set B , in each step, the system may choose to reach another state (from X) or to output one output symbol $b \in B$.

Again, coalgebras give rise to a category. For any given endofunctor F on a category \mathbf{C} , the category **Coalg**(F) has as objects all F -coalgebras, as arrows

all F -coalgebra homomorphisms, as identity arrows the identity arrows of \mathbf{C} – note that identity arrows are coalgebra homomorphisms because functoriality of F guarantees that $F\text{id}_X = \text{id}_{FX}$ and therefore commutativity of the defining diagram – and composition is just the composition of \mathbf{C} limited to coalgebra homomorphisms. In the sequel, whenever the functor is clear from the context, we will just write coalgebra instead of F -coalgebra and coalgebra homomorphism instead of F -coalgebra homomorphism.

To get a better understanding of how coalgebras can be used to model state-based systems, we take a look at the prototypical state-based system, non-deterministic automata.

Example 2.4.23 (Non-Deterministic Automata as Coalgebras) *In order to model a non-deterministic automaton over the input alphabet A , we need to map each state from the state set X to a possibly empty set of a successor states for each $a \in A$. Moreover, we need to model if the state in question is a final state or not. This can be captured by the functor $F: \mathbf{Set} \rightarrow \mathbf{Set}$ defined according to:*

Objects *Each set X is mapped to the set $FX = \mathcal{P}(X)^A \times \{0, 1\}$*

Arrows *Each function $f: X \rightarrow Y$ is mapped to the function $Ff: FX \rightarrow FY$ where each pair $(g, t) \in \mathcal{P}(X)^A \times \{0, 1\}$ is mapped as follows: $Ff(g, t) = (\mathcal{P}f \circ g, t)$. Here, $\mathcal{P}f$ is defined as for the powerset functor, i.e. $\mathcal{P}f(X') = \{f(x) \mid x \in X'\}$.*

Using this functor, we can model a non-deterministic automaton M coalgebraically in the following way: Let X be the state set of $M = (X, A, \delta, F)$, then we define the coalgebra $(X, \alpha: X \rightarrow FX)$ by defining $\alpha(x) = (f, t)$, where $t = 1$ if $x \in F$, $t = 0$ otherwise and $f(x)(a) = \{x' \in X \mid x' \in \delta(x, a)\}$ for all $a \in A$.

Note that this is not the only way to model non-deterministic automata coalgebraically. For instance, one could alternatively work in the category \mathbf{Rel} and choose the functor F that maps any set X to $FX = A \times X \cup \{\bullet\}$ and each relation $f \subseteq X \times Y$ to a relation $Ff: FX \rightarrow FY$ where $Ff(\bullet) = \bullet$ and $Ff(a, x) = \{(a, x') \mid (x, x') \in f\}$. Here, a coalgebraic model for a non-deterministic automaton collects in $\alpha(x)$ all those pairs (a, x') where x' can

be reached from x via a and additionally, $\bullet \in \alpha(x)$ if x is final. Straight-forward computations can be used to show that these functors are actually isomorphic, i.e. there exists a natural isomorphism between $\mathcal{P}(X \times A \cup \{\bullet\})$ and $\mathcal{P}(X)^A \times \{0, 1\}$.

In similar ways, various automaton models can be modelled as coalgebras in **Set** or **Rel**. For instance, in Chapter 3 we will model deterministic automata and labelled transition systems as **Set**-coalgebras.

2.4.5 Coalgebraic Behavioural Equivalence

Now that we have seen how to model state-based systems as coalgebras, we turn our attention towards analysing their behaviour.

In the theory of coalgebras, traditional bisimilarity [Par81] is captured in more than one way, namely: coalgebraic bisimulation, via an arrow into any coalgebra (so-called cocongruences), via the arrow into the final object of the coalgebra category (if that exists), called final coalgebra, and a related approach that minimises the coalgebra via factorisation (see Section 7.6).

We will first fix the notion of behavioural equivalence used in this thesis.

Definition 2.4.24 (Behavioural equivalence) *Let (\mathbf{C}, U) be a concrete category and $(X, \alpha : X \rightarrow FX)$ be an F -coalgebra. We call the elements $x \in UX$ the states of (X, α) . Two states $x, y \in UX$ are behaviourally equivalent ($x \sim y$) if and only if there exists an F -coalgebra (Y, β) and a coalgebra-homomorphism $f : (X, \alpha) \rightarrow (Y, \beta)$ such that $Uf(x) = Uf(y)$.*

This notion is closely related to the notion of behavioural equivalence induced by the arrow into the final coalgebra. If in the above definition we do not allow any coalgebra (Y, β) , but only the final one, if it exists, then we obtain a notion of behavioural equivalence that coincides with our notion whenever a final coalgebra exists. Final coalgebras need not exist in general, so our notion of behavioural equivalence is more general than the alternative characterisation using final coalgebras. On the flipside, characterising behavioural equivalence via final coalgebras leads to a unique representative of all systems with common behaviour, i.e. their image in the final coalgebra.

For our previously discussed example of non-deterministic automata, (coal-

gebraic) behavioural equivalence coincides with bisimilarity ⁴, as we can see by characterising coalgebra homomorphisms. Let $(X, \alpha: X \rightarrow FX)$ and $(Y, \beta: Y \rightarrow FY)$ be coalgebras for F from Example 2.4.23 and assume there exists a coalgebra homomorphism $f: X \rightarrow Y$ such that $Ff \circ \alpha = \beta \circ f$. Furthermore, assume $f(x) = f(x')$ for some pair of states $x, x' \in X$. We claim that then x and x' are bisimilar, as witnessed by the bisimulation relation $R = \{(x, x') \mid f(x) = f(x')\}$. If x is final, then $\pi_1^2(\alpha(x)) = 1$ and thus $\pi_1^2(Ff \circ \alpha(x)) = 1$. Since f is a coalgebra homomorphism, also $\pi_1^2(\beta \circ f(x)) = 1$, and, since $f(x) = f(x')$, $\pi_1^2(\beta \circ f(x')) = 1$, which in turn means $\pi_1^2(Ff \circ \alpha(x')) = 1$ and thus that x' is final. Since the situation is symmetric, this shows that the acceptance behaviour of x and x' is the same. For the more interesting part, assume there exists a transition $x \xrightarrow{a} \tilde{x}$, then $\tilde{x} \in \pi_2^2(\alpha(x)(a))$ and thus $f(\tilde{x}) \in \pi_2^2(Ff \circ \alpha(x)(a))$. The same chain of arguments as before yields that $f(\tilde{x}) \in \pi_2^2(Ff \circ \alpha(x')(a))$. Therefore, there must exist a state \tilde{x}' such that $x' \xrightarrow{a} \tilde{x}'$ and $f(\tilde{x}) = f(\tilde{x}')$. The other direction, that bisimulation implies behavioural equivalence, can be checked by factoring the state set by the bisimilarity and to map each state into its equivalence class. It can then easily be seen that this defines a coalgebra homomorphism. Thus, we can conclude that behavioural equivalence for coalgebras over this functor indeed induces bisimulation.

However, behavioural equivalence need not coincide with bisimulation in general. In fact, depending on the way one models systems, various behavioural equivalences such as weak bisimulation or language equivalence can be captured. In Chapter 3, we will discuss several examples of various notions of behaviour and system types. For non-deterministic automata, one is often interested not in bisimulation but language equivalence, instead. So the question arises how to model a non-deterministic automaton in such a way that language equivalence rather than bisimulation is characterised by behavioural equivalence. We have seen that there are alternative ways of modelling non-deterministic automata in **Set**, but we cannot expect a different notion of behavioural equivalence for

⁴Bisimilarity for non-deterministic automata is defined in the same way as for labelled transition systems, but additionally, two states need to agree on termination as well to be bisimilar (which then propagates to other states, so that bisimilar states are language equivalent)

isomorphic functors. Instead, we will employ a change of category. Rather than modelling in **Set**, we work in a Kleisli category.

The idea behind the use of Kleisli categories is, to distinguish observable choice that is intended to differentiate behaviour and side effects that may occur internally and not be observed. Assume a Kleisli category for a monad (T, η, μ) on **C** and an endofunctor F in $\mathbf{Kl}(T)$ is given. Then an F -coalgebra has the form $X \rightarrow TFX$ in **C**. While the effect that comes through the functor F is explicit in the arrow, the effect of T is thought of to be hidden in the category as a side effect. This way, Kleisli categories can be used to obtain a different notion of behaviour by splitting the branching behaviour into two parts, one monad and one functor in the Kleisli category.

In order to demonstrate this, we will turn our attention back towards non-deterministic automata.

Example 2.4.25 *We now want to model non-deterministic automata in such a way that behavioural equivalence coincides with language equivalence, rather than bisimilarity as before. While we want the choice of a label of a transition to influence behavioural equivalence, we would like to hide the concrete choice of the next state. Therefore, we choose as the monad (T, η, μ) the powerset monad to obtain the Kleisli category $\mathbf{Kl}(\mathcal{P})$. As functor F we use $FX = A \times X + 1$ on objects. On arrows, we need to take into consideration, that arrows $f: X \rightarrow Y$ in the category $\mathbf{Kl}(\mathcal{P})$ are actually functions $f: X \rightarrow \mathcal{P}(Y)$. So we define $Ff: FX \rightarrow FY$ according to $Ff(a, x) = \{(a, y) \mid y \in f(x)\}$, $Ff(\bullet) = \{\bullet\}$.*

Modelling non-deterministic automata in the Kleisli category now works the same way as modelling them in **Set** using the functor $\mathcal{P}(A \times X + 1)$. The difference in behavioural equivalence arises when considering coalgebra homomorphisms, since now a coalgebra homomorphism $f: X \rightarrow Y$ actually maps a state $x \in X$ to a set of states $Y' \subseteq Y$ rather than a single state. So, if $f(x) = f(x')$ and $x \xrightarrow{a} \tilde{x}$, then $f(\tilde{x})$ must still be in the image of $Ff \circ \alpha(x')$, but it need not be the image of any single state \tilde{x}' s.t. $x' \xrightarrow{a} \tilde{x}'$, but it may be collected from several a successors of x' , thus giving more freedom in the choice of coalgebra homomorphisms.

Now that we have characterised behavioural equivalence in coalgebras and demonstrated different modelling techniques fit to obtain different notions of

behaviour, we will turn our focus towards an algorithm to (semi-)decide behavioural equivalence. For this purpose, the final chain construction generalises the classical partition refinement algorithm for labelled transition systems.

In parallel, we will revisit the partition refinement algorithm to compute the bisimilarity of a single labelled transition system⁵ and translate the coalgebraic concepts to their counterpart in the case of labelled transition systems.

We consider a coalgebra $(X, \alpha: X \rightarrow FX)$ for a functor F	We consider a labelled transition system (X, A, \rightarrow) .
We start by setting $d_0: X \rightarrow 1$, the unique arrow from X into a final object.	We start by defining R_0 as the relation that puts all pairs of states into relation. R_0 can be obtained from d_0 by checking for each pair of states whether they get mapped to the same element (which is trivially true for d_0).
Given an arrow $d_i: X \rightarrow F^i 1$, we compute $d_{i+1} = Fd_i \circ \alpha$	Given a relation R_i , relating all pairs of states that have the same image under d_i , we check each pair of states $(x, y) \in R_i$ whether the set of successors of x and the set of successors of y are in relation R_i . Two sets of states are said to be in relation R_i whenever for each element x' in the first set there exists an element y' in the second set such that $(x', y') \in R_i$ and vice-versa. The sets of successor states can be obtained from α , whereas multiplication with Fd_i yields the correspondence to the previous relation R_i .
When an index n is found such that d_n is isomorphic to d_{n+1} , the algorithm terminates.	The algorithm terminates when $R_n = R_{n+1}$. Since two set functions d_n and d_{n+1} are isomorphic if and only if there exists a bijection between d_n and d_{n+1} , this corresponds exactly to the condition that $R_n = R_{n+1}$.

⁵We have introduced the partition refinement algorithm for two systems, originally, for this analogy we just assume that $(X, A, \rightarrow_1) = (Y, A, \rightarrow_2)$ in Algorithm 2.3.9.

Note, that the strong correspondence between the final chain algorithm and the partition refinement algorithm for bisimilarity of labelled transition systems is strongly dependent on properties of **Set**. In particular, the termination condition that allows to cease computation as soon as the final partitions of states is reached, in the case of labelled transition systems is less well-behaved in different categories. We will later see examples where the algorithm may never terminate, even though it finds the right partition of the state set at one point.

This construction is a well-known construction that was initially used to characterise final coalgebras [Wor05], though in [ABH⁺12] the final chain was also used to obtain minimisations. For this purpose, an optimisation of the algorithm by the use of factorisation structures was proposed.

Definition 2.4.26 (Factorisation Structures) *Let \mathbf{C} be a category and let \mathcal{E}, \mathcal{M} be classes of morphisms in \mathbf{C} . The pair $(\mathcal{E}, \mathcal{M})$ is called a factorisation structure for \mathbf{C} whenever*

- \mathcal{E} and \mathcal{M} are closed under composition with isos.
- \mathbf{C} has $(\mathcal{E}, \mathcal{M})$ -factorisations of morphisms, i.e., each morphism f of \mathbf{C} has a factorisation $f = m \circ e$ with $e \in \mathcal{E}$ and $m \in \mathcal{M}$.
- \mathbf{C} has the unique $(\mathcal{E}, \mathcal{M})$ -diagonalisation property: for each commutative square $g \circ e = m \circ f$ with $e \in \mathcal{E}$ and $m \in \mathcal{M}$ there exists a unique diagonal, i.e., a morphism d such that $d \circ e = f$ and $m \circ d = g$, cf. the following figure.

$$\begin{array}{ccc} A & \xrightarrow{e} & B \\ f \downarrow & \nearrow d & \downarrow g \\ C & \xrightarrow{m} & D \end{array}$$

Adámek et al. have shown in [ABH⁺12], that behavioural equivalence can be characterised by the final chain algorithm, if in each step, the arrow d_i is factored to $d_i = m_i \circ e_i$ using a factorisation structure and the computation of the next step is changed to $d_{i+1} = F(e_i) \circ \alpha$, provided \mathcal{M} arrows are closed

under application of F and mapped to monos by the concretisation functor U . Considering our running example of non-deterministic automata, this idea means the following: In some steps of the partition refinement algorithm, not all possible subsets of $F^i 1$ correspond to a new equivalence class. Factorisation prunes the codomain of d_i to only include those elements that are actually in the image of d_i . This leads to a more compact representative, if a function is always given with explicit domain and codomain. This gained compactness of representatives propagates to later steps.

To conclude the review of the state-of-the-art on behavioural equivalence, we note that in [ABH⁺12] the authors already proposed a relaxation of the condition of using a factorisation structure. Instead, it is possible to use a pseudo-factorisation. A pseudo-factorisation structure arises if a reflective subcategory of the category \mathbf{C} under consideration possesses a factorisation structure. First we need to define reflective subcategories:

Definition 2.4.27 *Let \mathbf{S} be a subcategory of \mathbf{C} and let X be an object of \mathbf{C} . Then, an \mathbf{S} -reflection for X is a \mathbf{C} -arrow $\rho_X : X \rightarrow X'$ into some object X' of \mathbf{S} such that the following universal property is satisfied: for any \mathbf{C} -arrow $f : X \rightarrow Y$ into some object Y of \mathbf{S} , there exists a unique \mathbf{S} -arrow $f' : X' \rightarrow Y$ (called ρ -reflection of f) such that $f' \circ \rho_X = f$. Moreover, \mathbf{S} is a reflective subcategory of \mathbf{C} if each object of \mathbf{C} has an \mathbf{S} -reflection.*

Now, if a category \mathbf{C} has a reflective subcategory that has a factorisation structure, the pseudo-factorisation structure in \mathbf{C} arises in the following way.

Definition 2.4.28 (Pseudo-Factorisation) *Given a reflective subcategory \mathbf{S} of \mathbf{C} with its reflection ρ that has an $(\mathcal{E}, \mathcal{M})$ -factorisation structure. Then, the category \mathbf{C} has a so-called $(\mathcal{E}, \mathcal{M})$ -pseudo-factorisation structure since every \mathbf{C} -arrow $f : X \rightarrow Y$ into some object Y of \mathbf{S} can be factored as $f = m \circ (e \circ \rho_X)$ where $m \circ e$ (for $m \in \mathcal{M}, e \in \mathcal{E}$) is the decomposition of the ρ -reflection of f . For such pseudo-factorisations, we do not necessarily have a diagonal arrow, but one can show that such an arrow exists whenever g is in \mathbf{S} .*

In the final chain algorithm, it is permissible to use a pseudo-factorisation structure instead of a factorisation structure to the same effect.

Chapter 3

Generic Partition Refinement Algorithms for Coalgebras

3.1 Introduction

Coalgebra [Rut00] (see Section 2.4) offers a unifying theory in which we can model and reason about various types of transition systems and automata and comes equipped with a canonical notion of behavioural equivalence (see e.g. [Sta09] for an overview on behavioural equivalences for coalgebras). An important contribution of coalgebra is the provision of generic algorithms for checking behavioural equivalence, independent of the type of transition system. Such generic algorithms would be useful for two reasons: first, for classifying and comparing existing algorithms; second for obtaining prototype algorithms (that might be further optimised) when studying a new class of systems. One example of such generic methods that have recently been studied are up-to techniques [RBB⁺15, BPPR14a].

Here we are interested in generic algorithms for checking behavioural equivalence, akin to minimisation or partition refinement techniques. A rather general account for such minimisation techniques has been presented in [ABH⁺12], by using factorisation structures and factoring the arrows into the final chain. For coalgebras over **Set** this boils down to classical partition refinement, encompassing minimisation of deterministic automata [HU79] or the computation of bisimulation equivalence classes for probabilistic transition systems [LS89, Bai96]. In [ABH⁺12] it was also shown how to handle some coalgebras

in categories different from **Set**, especially in Kleisli categories, which allow to specify side-effects and hence trace equivalence (where non-determinism is abstracted away as a side-effect).

However, some examples do not fall into this framework, most notably weighted automata over semirings, for which the underlying category does not possess any suitable factorisation structures. We found that a different notion of factorisation can be used to capture behavioural equivalence. Furthermore it is unnecessary to look for a unique minimisation or canonical representative, rather it is sufficient to compute *some* representative coalgebra and use it to precisely answer questions about behavioural equivalence. For weighted automata over semirings this yields an algorithm that we did not find in the literature as such. However, for many specific (classes of) semirings, instances of this algorithm have been presented in the past: For fields a method is discussed in [Bor09] and a result for rings, based on results by Schützenberger, is given in [DK13]. The notion of coalgebra homomorphism is strongly related to conjugacy [BLS06], this correspondence will be worked out in Subsection 4.2.3. For probabilistic automata there is a related procedure for checking language equivalence [KMO⁺11] which uses the same base concept, but improves on the expected run time by making use of randomisation.

We will present a generic algorithm, based on the notion of equivalence classes of arrows. We will compare this algorithm to the algorithm of [ABH⁺12] that uses factorisation structures [AHS90]. An important special case and the motivating example that we will discuss in detail is the setting of weighted automata.

3.2 Preliminaries

In order to be able to describe our algorithms, we will introduce some relations on objects and arrows. We first need the notion of a coslice category.

Definition 3.2.1 (Coslice category) *Let \mathbf{C} be a category and let X be an object of \mathbf{C} . The coslice category $X \downarrow \mathbf{C}$ under X has all arrows $a: X \rightarrow A$ as objects. An arrow between two objects $a: X \rightarrow A$, $b: X \rightarrow B$ is a \mathbf{C} -arrow $c: A \rightarrow B$ such that $c \circ a = b$.*

Definition 3.2.2 (Relations on objects and arrows) *Let X, Y be two objects of a category \mathbf{C} . We write $X \leq Y$ whenever there is an arrow $f: X \rightarrow Y$. Furthermore we write $X \equiv Y$ whenever $X \leq Y, Y \leq X$.*

Let $a: X \rightarrow A, b: X \rightarrow B$ be two arrows in \mathbf{C} with the same domain. We write $a \leq^X b$ whenever $a \leq b$ holds in the coslice category $X \downarrow \mathbf{C}$, i.e., there exists an arrow $d: A \rightarrow B$ with $d \circ a = b$. Similarly we write $a \equiv^X b$.

If the objects of a category formed a set, \leq would be a preorder (or quasi-order) and \equiv would be an equivalence relation (i.e. reflexivity, transitivity and symmetry hold for \equiv). Note that if a category has a final object 1 , then $X \leq 1$ holds for any other object X . Furthermore if $f: X \rightarrow 1$ is an arrow into the final object, we have that $g \leq^X f$ for any other arrow $g: X \rightarrow Y$.

Example 3.2.3 *Let $f: X \rightarrow A, g: X \rightarrow B$ with $X \neq \emptyset$ be two functions in **Set**. It holds that $f \equiv^X g$ if and only if both functions induce the same partition on X , i.e., for all $x, y \in X$ it holds that $f(x) = f(y) \iff g(x) = g(y)$. Similarly $f \leq^X g$ means for all $x, y \in X$ that $f(x) = f(y) \Rightarrow g(x) = g(y)$.*

Hence, if $a \equiv^X b$ holds for two arrows $a: X \rightarrow A, b: X \rightarrow B$ in a concrete category (\mathbf{C}, U) , we can conclude that Ua, Ub induce the same partition on UX .

Proposition 3.2.4 *Let X, Y be objects of a category \mathbf{C} and let $a: X \rightarrow A, b: X \rightarrow B$ be arrows in \mathbf{C} . Furthermore let $F: \mathbf{C} \rightarrow \mathbf{C}$ be an endofunctor.*

1. $X \leq Y$ implies $FX \leq FY$.
2. $a \leq^X b$ implies $Fa \leq^{FX} Fb$.
3. $a \leq^X b$ implies $a \circ c \leq^Y b \circ c$ for any arrow $c: Y \rightarrow X$.

Proof:

1. This is true due to functoriality of F . If $X \leq Y$, then there is an arrow $f: X \rightarrow Y$ and thus also an arrow $Ff: FX \rightarrow FY$, so $FX \leq FY$.
2. Similar to case 1 this also follows from functoriality of F . $a \leq^X b$ implies there is an arrow $f: A \rightarrow B$ such that $f \circ a = b$ and thus $Ff \circ Fa = Fb$, i.e. $Fa \leq^{FX} Fb$.
3. If $a \leq^X b$ then there is an arrow $f: A \rightarrow B$ such that $f \circ a = b$ and thus $f \circ a \circ c = b \circ c$.

□

The notion of equivalent arrows is connected to the notion of split-mono. An arrow $m: X \rightarrow Y$ is called split-mono if it has a left inverse, i.e., if there exists an arrow $m^\leftarrow: Y \rightarrow X$ such that $m^\leftarrow \circ m = id_X$. Now, assume two arrows $a: X \rightarrow Y$, $b: X \rightarrow Z$ are equivalent ($a \equiv^X b$). Then there is an arrow $m: Y \rightarrow Z$ and an arrow $m^\leftarrow: Z \rightarrow Y$ such that $m \circ a = b$ and $m^\leftarrow \circ b = a$. Then, $m^\leftarrow \circ m \circ a = a$, hence m behaves like a split-mono, relative to a . More concretely, split-monos $m: X \rightarrow Y$ are exactly the arrows that are equivalent to id_X .

In Section 3.4.3, we will show that equivalence on arrows boils down to a very natural notion in the setting of weighted automata: the fact that two sets of vectors (with entries from a semiring) generate the same semimodule.

3.3 Generic Algorithms

Before introducing the algorithms, based on the construction of the final chain [AK95], we will first discuss how behavioural equivalence can be expressed as a post-fixpoint using the terminology of the previous section.

Proposition 3.3.1 *Let F be an endofunctor on a concrete category (\mathbf{C}, U) and let $\alpha: X \rightarrow FX$ be a coalgebra on \mathbf{C} . Furthermore let $f: X \rightarrow Y$ be a \mathbf{C} -arrow. It holds that $f \leq^X Ff \circ \alpha$ (f is a post-fixpoint) if and only if there exists a coalgebra $\beta: Y \rightarrow FY$ such that f is a coalgebra homomorphism from (X, α) to (Y, β) .*

For every such post-fixpoint f we have that $x, y \in UX$ and $Uf(x) = Uf(y)$ implies $x \sim y$. If, in addition, it holds for every other post-fixpoint $g: X \rightarrow Z$ that $g \leq^X f$ (f is the largest post-fixpoint), we can conclude that f induces behavioural equivalence, i.e., $Uf(x) = Uf(y) \iff x \sim y$.

Proof: The first statement is almost trivial, since $f \leq^X Ff \circ \alpha$ means the existence of an arrow $\beta: Y \rightarrow FY$ with $\beta \circ f = Ff \circ \alpha$, which is exactly the condition that β is a coalgebra homomorphism. Hence, by definition of behavioural equivalence, $Uf(x) = Uf(y)$ implies $x \sim y$. It is left to show that $x \sim y$ implies $Uf(x) = Uf(y)$ if f is the largest fixpoint. Since $x \sim y$, there must be some coalgebra $\gamma: Z \rightarrow FZ$ and a coalgebra homomorphism

$g: X \rightarrow Z$ such that $\gamma \circ g = Fg \circ \alpha$ and $Ug(x) = Ug(y)$. This implies that $g \leq^X Fg \circ \alpha$ and hence $g \leq^X f$. Finally, we obtain $Uf(x) = Uf(y)$. \square

In **Set** one can imagine the largest fixpoint $f: X \rightarrow Y$ as a function that maps every state into its equivalence class.

We now consider the the Final Chain Algorithm A.

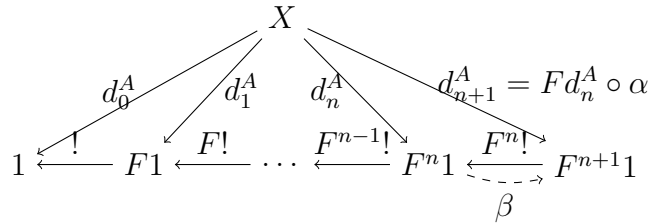
Algorithm 3.3.2 Final Chain Algorithm A

Let F be an endofunctor on a concrete category (\mathbf{C}, U) and let $\alpha: X \rightarrow FX$ be a coalgebra in \mathbf{C} . Moreover, let 1 be the final object of \mathbf{C} . We define the following algorithm.

Step 0: Take the (unique) arrow $d_0^A: X \rightarrow 1$.

Step $i + 1$: Compute $d_{i+1}^A = Fd_i^A \circ \alpha: X \rightarrow F^{i+1}1$.

Termination: If there exists an arrow $\beta: F^n 1 \rightarrow F^{n+1} 1$ such that $\beta \circ d_n^A = d_{n+1}^A$, i.e., if $d_n^A \leq^X d_{n+1}^A$, the algorithm terminates and returns d_n^A and $(F^n 1, \beta)$ as its result.



The algorithm is easy to analyse using the terminology introduced earlier. Specifically, it yields a sequence of arrows $d_0^A \geq^X d_1^A \geq^X d_2^A \geq^X \dots$ that approximates behavioural equivalence from above. It terminates whenever this sequence becomes stationary, i.e., $d_n^A \equiv^X d_{n+1}^A$.

Lemma 3.3.3 *Let $g: X \rightarrow Z$ be any post-fixpoint, i.e., $g \leq^X Fg \circ \alpha$. Then for all d_i^A obtained in Algorithm A we have $d_i^A \geq^X g$.*

Proof: Clearly $d_0^A \geq^X g$, because d_0^A is the arrow into the final object of the category. Therefore, we can find an arrow g' from g 's codomain to 1 and the composition $g' \circ g$ is an arrow from X to 1 , which is unique, due to finality of 1 , i.e. it is the same as d_0^A . By induction, using Proposition 3.2.4, we can show

that $d_i^A \geq^X g$ implies $d_{i+1}^A = Fd_i^A \circ \alpha \geq^X Fg \circ \alpha \geq^X g$. The first step is the induction hypothesis, the second uses the fact that g is a post-fixpoint. \square

Proposition 3.3.4 *If Algorithm A terminates in step n , its result d_n^A induces behavioural equivalence, i.e. $x, y \in UX$ are behaviourally equivalent ($x \sim y$) if and only if $Ud_n^A(x) = Ud_n^A(y)$.*

Proof: First, $d_0^A \geq^X d_1^A$ since d_0^A is an arrow into the final object. Since \leq^X is preserved by functors and by right composition with arrows, we obtain by induction, using Proposition 3.2.4, that $d_i^A \geq^X d_{i+1}^A$. The termination condition says that $d_n^A \leq^X d_{n+1}^A = Fd_n^A \circ \alpha$. In order to conclude with Proposition 3.3.1 that the resulting arrow d_n^A induces behavioural equivalence, we have to show that it is the largest post-fixpoint. Assume another post-fixpoint $g: X \rightarrow Z$ with $g \leq^X Fg \circ \alpha$. Then, Lemma 3.3.3 shows that $d_n^A \geq^X g$ and thus d_n^A is the largest post-fixpoint. \square

Furthermore whenever $d_n^A \equiv^X d_{n+1}^A$ we have that $d_n^A \equiv^X d_m^A$ for every $m \geq n$. Hence every arrow d_m^A obtained at a later step induces behavioural equivalence as well. We can show that d_n^A is equivalent to the arrow into the final coalgebra (if it exists).

Lemma 3.3.5 *Let (Z, κ) be the final coalgebra, i.e., a coalgebra into which there exists a unique coalgebra homomorphism from every other coalgebra. Take a coalgebra (X, α) and the unique homomorphism $f: (X, \alpha) \rightarrow (Z, \kappa)$ and assume that Algorithm A terminates in step n . Then $f \equiv^X d_n^A$.*

Proof: Since (Z, κ) is the final object in the coalgebra category, there is a (unique) coalgebra homomorphism $h: (Y, \beta) \rightarrow (Z, \kappa)$. Thus, $d_n^A \leq^X f$ via h . The other way around, the arrow f into the final coalgebra is a fixpoint and thus, via Lemma 3.3.3 we also have $d_n^A \geq^X f$. We can conclude $d_n^A \equiv^X f$. \square

Note that after each step of the algorithm, it is permissible to replace d_i^A with any representative e_i^A of the equivalence class of d_i^A , i.e., any e_i^A with $d_i^A \equiv^X e_i^A$. This holds because $d_i^A \equiv^X e_i^A$ implies $d_{i+1}^A = Fd_i^A \circ \alpha \equiv^X Fe_i^A \circ \alpha$ and checking the termination condition $d_n^A \leq^X d_{n+1}^A$ can instead be done for any representatives of d_n^A, d_{n+1}^A . This gives rise to the following algorithm:

Algorithm 3.3.6 Final Chain Algorithm B

Let F be an endofunctor on a concrete category (\mathbf{C}, U) and let $\alpha : X \rightarrow FX$ be a coalgebra in \mathbf{C} . Moreover, let 1 be the final element of \mathbf{C} and \mathcal{R} , the class of representatives, be a class of arrows of \mathbf{C} such that for any arrow d in \mathbf{C} we have an arrow $e \in \mathcal{R}$ that is equivalent to d , i.e. $d \equiv^X e$. We define the following algorithm:

Step 0: Take the (unique) arrow $d_0^B : X \rightarrow 1$.

Step $i + 1$: Find a representative $e_i^B \in \mathcal{R}$, for d_i^B , i.e. factor $d_i^B = m_i^B \circ e_i^B$ such that $(e_i^B : X \rightarrow Y_i) \in \mathcal{R}$, $m_i^B : Y_i \rightarrow FY_{i-1}$. Determine $d_{i+1}^B = Fe_i^B \circ \alpha : X \rightarrow FY_i$.

Termination: If there exists an arrow $\gamma : Y_n \rightarrow FY_n$ such that $\gamma \circ e_n^B = d_{n+1}^B$, i.e., $d_{n+1}^B \geq^X e_n^B$, the algorithm terminates and returns e_n^B and (Y_n, γ) as its result.

The choice good and compact representatives can substantially mitigate state space explosion. Naturally, in concrete applications, one has to choose a suitable class \mathcal{R} and a strategy for factoring. For the case of weighted automata we will discuss how such a strategy looks like (see Section 3.4.3). Algorithm B is an optimization of Algorithm A and it terminates in exactly as many steps as Algorithm A (if it terminates).

Proposition 3.3.7 *Algorithm B terminates in n steps if and only if Algorithm A terminates in n steps. Furthermore two states $x, y \in UX$ are behaviourally equivalent ($x \sim y$) if and only if $Ue_n^B(x) = Ue_n^B(y)$.*

Proof: First, we observe that we always have $d_i^B \equiv^X d_i^A$. This can be shown inductively. $d_0^A = d_0^B$ by definition. Now, assume $d_i^A \equiv^X d_i^B$, then $Fd_i^A \equiv^{FX} Fd_i^B$ and, since by definition $e_i^B \equiv^X d_i^B$, $Fe_i^B \equiv^{FX} Fd_i^A$. In both cases, we have used Proposition 3.2.4, Item (2). Item (3) then yields $d_{i+1}^B = Fe_i^B \circ \alpha \equiv^X Fd_i^A \circ \alpha = d_{i+1}^A$.

In addition, we factor in such a way that $e_i^B \equiv^X d_i^B$.

If Algorithm B terminates in n steps, then $d_{n+1}^B \geq^X e_n^B$. This implies $d_{n+1}^A \equiv^X d_{n+1}^B \geq^X e_n^B \equiv^X d_n^B \equiv_X d_n^A$ and Algorithm A terminates as well.

If, on the other hand, Algorithm A terminates in n steps, then $d_{n+1}^A \geq^X d_n^A$. We obtain $d_{n+1}^B \geq^X d_n^A \equiv^X d_n^B \equiv^X e_n^B$. Hence Algorithm B also terminates in n steps.

Since $d_n^A \equiv^X e_n^B$, we have seen in Example 3.2.3 that they induce the same partition on X . Hence it follows that one can check behavioural equivalence also with Algorithm B. \square

Note that termination for Algorithm B is independent of the choice of the representatives e_i^B .

In [ABH⁺12] an algorithm similar to ours is being discussed. The algorithm also works on the final chain and uses a factorisation system to trim the intermediate results with the aim of finding a so-called minimisation, i.e., a unique and minimal representative of the given coalgebra α . We will discuss a variation of said algorithm that has a more liberal termination condition akin to the termination conditions for Algorithm A and Algorithm B.

Algorithm 3.3.8 Final Chain Algorithm C

Let F be an endofunctor on a concrete category (\mathbf{C}, U) and let $\alpha : X \rightarrow FX$ be a coalgebra in \mathbf{C} . Moreover, let 1 be the final element of \mathbf{C} and $(\mathcal{E}, \mathcal{M})$ be a factorisation structure such that U maps \mathcal{M} -morphisms to injections and F preserves \mathcal{M} . We define the following algorithm:

Step 0: Take the (unique) arrow $d_0^C : X \rightarrow 1$.

Step $i + 1$: Factor $d_i^C = m_i^C \circ e_i^C$ via the factorisation structure, i.e. $(e_i^C : X \rightarrow Y_i) \in \mathcal{E}$, $(m_i^C : Y_i \rightarrow FY_{i-1}) \in \mathcal{M}$. Determine $d_{i+1}^C = Fe_i^C \circ \alpha : X \rightarrow FY_i$.

Termination: If there exists an arrow $\delta : Y_n \rightarrow FY_n$ such that $\delta \circ e_n^C = d_{n+1}^C$, the algorithm terminates and returns e_n^C and (Y_n, δ) as its result.

If a category has a suitable factorisation structure that ensures compact intermediate results, Algorithm C might terminate faster than Algorithm B. However, not every category has a suitable factorisation structure (see our examples in Section 3.4.3), which drove us to investigate alternatives such as Algorithms A and B introduced earlier. For the trivial factorisation structure where \mathcal{E} is the class of all arrows and \mathcal{M} are just the isomorphism, we obtain just the unoptimised Algorithm A. Algorithm C is a variation of the algorithm in

[ABH⁺12] in the sense that it terminates as soon as a coalgebra homomorphism is found as opposed to running until an m_i^C is found that is an isomorphism. This termination condition is sufficient to decide behavioural equivalence.

Under certain conditions, which basically amount to a factorisation structure possessing \mathcal{E} morphisms for all equivalence classes, Algorithm C terminates in the same number of steps as Algorithm A and Algorithm B, where in the latter case we can choose $\mathcal{R} = \mathcal{E}$ to obtain the same intermediate results, as well.

Proposition 3.3.9 *Let F be an endofunctor on a concrete category (\mathbf{C}, U) and let $\alpha : X \rightarrow FX$ be a coalgebra in \mathbf{C} . Moreover, let $(\mathcal{E}, \mathcal{M})$ be a factorisation structure for \mathbf{C} .*

1. *Assume that F preserves \mathcal{M} -arrows. If Algorithm A terminates in n steps, then Algorithm C terminates in n steps, as well.*
2. *If Algorithm C terminates in n steps and for each arrow d_i^A , $i = 0, \dots, n$ there exists an arrow $e_i \in \mathcal{E}$ with $e_i \equiv^X d_i^A$ (i.e., there is an arrow in \mathcal{E} in all relevant equivalence classes), then Algorithm B terminates in n steps, as well.*

Furthermore two states $x, y \in UX$ are behaviourally equivalent ($x \sim y$) if and only if $Ue_n^C(x) = Ue_n^C(y)$.

Proof:

1. Since F preserves \mathcal{M} -morphisms and $(\mathcal{E}, \mathcal{M})$ -factorisations are unique up to isomorphism we obtain, up to iso, the same \mathcal{E} -arrow e_i^C when factoring d_i^A as we do when factoring d_i^C . This is clear by induction: $d_0^A = d_0^C$ and if the statement is true for i , we have $d_i^A = m'_i \circ e_i^C$ for $m'_i \in \mathcal{M}$, $e_i \in \mathcal{E}$ and $d_i^C = m_i^C \circ e_i^C$ for $m_i^C \in \mathcal{M}$. Then, $d_{i+1}^A = Fd_i^A \circ \alpha = F(m'_i \circ e_i^C) \circ \alpha = Fm'_i \circ Fe_i^C \circ \alpha = Fm'_i \circ d_{i+1}^C$. Now assume that $d_{i+1}^C = m_{i+1}^C \circ e_{i+1}^C$, for $e_{i+1}^C \in \mathcal{E}$, $m_{i+1}^C \in \mathcal{M}$, we can conclude that $d_{i+1}^A = Fm'_i \circ m_{i+1}^C \circ e_{i+1}^C$. We have that $Fm'_i \circ m_{i+1}^C$ is an \mathcal{M} -arrow, since \mathcal{M} -arrows are preserved by F and by composition. Hence we again obtain e_{i+1}^C when factoring out an \mathcal{E} -morphism from d_{i+1}^A .

Assume Algorithm A terminates in n steps. Then we have an arrow $\beta : F^n 1 \rightarrow F^{n+1} 1$ such that $\beta \circ d_n^A = d_{n+1}^A$. Now consider the factorisations

of $d_{n+1}^A = m'_{n+1} \circ e_{n+1}^C$ and $d_n^A = m'_n \circ e_n^C$ and observe that this gives rise to the following commuting diagram and thus, by diagonalisation (see Definition 2.4.26) the diagonal arrow d , depicted as a dashed line, exists.

$$\begin{array}{ccc} X & \xrightarrow{e_n^C} & B \\ e_{n+1}^C \downarrow & \swarrow d & \downarrow \beta \circ m'_n \\ C & \xrightarrow{m'_{n+1}} & F^{n+1}1 \end{array}$$

The existence of this diagonal arrow d means that $(m_{n+1}^C \circ d) \circ e_n^C = m_{n+1}^C \circ e_{n+1}^C = d_{n+1}^C$, which is precisely the termination condition for Algorithm C.

2. First we will prove that the following holds: if for an arrow $f: X \rightarrow Y$ there exists an arrow $(e': X \rightarrow Z) \in \mathcal{E}$ such that $e' \equiv^X f$ then in the factorisation of f into $m \circ e$ using $(\mathcal{E}, \mathcal{M})$, we have $e \equiv^X f$. We assume that $e: X \rightarrow V$, $m: V \rightarrow Y$.

Since $e' \equiv^X f$, there is an arrow $m': Z \rightarrow Y$ such that $m' \circ e' = f = m \circ e$. Then, there has to be a diagonal arrow $d: Z \rightarrow V$ such that $d \circ e' = e$ and $m \circ d = m'$. Hence $e' \leq^X e$ via d .

Moreover, $e \leq^X e'$ holds as well. Since $e \equiv^X f$, we also have an arrow $m'': Y \rightarrow Z$ such that $m'' \circ f = e'$. Thus $(m'' \circ m) \circ e = m'' \circ (m \circ e) = m'' \circ f = e'$, which implies $e \leq^X e'$.

Now, assume Algorithm C terminates in n steps and the arrows d_i^A , $i = 0, \dots, n$ are such that there always exists an $e_i^C \equiv^X d_i^A$ such that $e_i^C \in \mathcal{E}$, then we have just shown that $d_i^A \equiv^X e_i^C$ for all i . Therefore, in Algorithm B we can choose $e_i^B = e_i^C$ and since the termination conditions for Algorithm B and Algorithm C coincide, Algorithm B also terminates in step n .

Above we have shown that $d_i^A = m'_i \circ e_i^C$ for $m'_i \in \mathcal{M}$. Since m'_i is mapped to an injection by U , d_i^A and e_i^C induce the same partition on X and we can also use Algorithm C to check behavioural equivalence. \square

Note that in our comparison of Algorithm B and Algorithm C, we did not discuss the class of representatives \mathcal{R} . In many cases, it is possible, given a

factorisation structure $(\mathcal{E}, \mathcal{M})$, to choose $\mathcal{R} = \mathcal{E}$. In those cases, the algorithms do not only terminate after the same amount of steps, but each intermediate result also coincides. However, there also exist cases where this does not hold and a class \mathcal{E} is not a suitable class of representatives, because the back-arrows are missing. An example for this will be discussed in Chapter 7 in form of the category **Poset**, which we will use to model conditional transition systems (CTS). The class of injective, order preserving functions can be identified as a suitable class \mathcal{M} giving rise to a factorisation structure in this category, however, since all arrows in **Poset** are required to be order preserving, a back arrow for an arrow $m \in \mathcal{M}$ need not exist. Even though termination must still occur after the same number of iterations, when using Algorithm B with a trivial class of representatives as in Algorithm C, it can then be advantageous to use Algorithm C over B, because potentially, a more compact representation of the intermediate results can be obtained. Depending on the representation of the intermediate results, in the case of CTS, this advantage may be negligible, because the only difference is the size of the codomain of arrows, but in other cases, e.g. for weighted automata, where matrices are the intermediate results and the choice of representatives reduces the matrix size, it is a significant difference.

One example where all both algorithms actually coincide completely, is **Set**. In **Set**, every equivalence class of arrows, except the equivalence classes of \equiv^\emptyset , contains a surjection. Therefore, a suitable class \mathcal{R} of representatives for Algorithm B would be the surjections. Hence, if we choose surjections and injections as a factorisation structure in **Set**, we obtain exactly the same intermediate steps for Algorithm C as for Algorithm B, provided that the state set is not empty. This yields classical partition refinement algorithms.

It is not satisfactory that Algorithm B and Algorithm C do not strictly generalise one another, so it is natural to ask for a more general algorithm that subsumes both optimisations. For this purpose we propose a fourth algorithm, Algorithm D, which moves the equivalence of arrows towards the category **Set**. This move to the image of the concretisation functor allows to capture factorisation structures in terms of representative arrows as well. Algorithm D certainly is less intuitive with respect to the category the coalgebras live in, but we will see that it subsumes both Algorithm B and Algorithm C.

Algorithm 3.3.10 Final Chain Algorithm D

Let F be an endofunctor on a concrete category (\mathbf{C}, U) and let $\alpha : X \rightarrow FX$ be a coalgebra in \mathbf{C} . Moreover, let 1 be the final element of \mathbf{C} and \mathcal{R} , the class of representatives, be a class of arrows of \mathbf{C} such that for any arrow $d : X \rightarrow Y$ in \mathbf{C} we have an arrow $e \in \mathcal{R}$ such that $UF^n d$ is equivalent to $UF^n e$ for all $n \in \mathbb{N}_0$, i.e. $UF^n d \equiv^{U(F^n X)} UF^n e$. We define the following algorithm:

Step 0: Take the (unique) arrow $d_0^D : X \rightarrow 1$.

Step $i + 1$: Find a representative $e_i^D \in \mathcal{R}$, for d_i^D , i.e. factor $Ud_i^D = m_i^D \circ Ue_i^D$ such that $(e_i^D : X \rightarrow Y_i) \in \mathcal{R}$ and $UF^j e_i^D \equiv^{U(F^j X)} UF^j d_i^D$ for all $j \in \mathbb{N}_0$. Determine $d_{i+1}^D = Fe_i^D \circ \alpha : X \rightarrow FY_i$.

Termination: If there exists an arrow $\gamma : Y_n \rightarrow FY_n$ such that $\gamma \circ e_n^D = d_{n+1}^D$, i.e., $d_{n+1}^D \geq^X e_n^D$, the algorithm terminates and returns e_n^D and (Y_n, γ) as its result.

Note that factoring in this algorithm depends strongly on the category of **Set**.

Lemma 3.3.11 *If Algorithm D terminates and returns e_n^D and (Y_n, γ) as its result, then e_n^D induces behavioural equivalence, i.e. $e_n^D(x) = e_n^D(y)$ if and only if $x \sim y$.*

Proof: We use the correctness of Algorithm A and show that it always holds that $Ud_i^D(x) = Ud_i^D(y) \Leftrightarrow Ud_i^A(x) = Ud_i^A(y)$.

First we show inductively, that $d_i^A = F^j d_{i-j}^A \circ F^{j-1} \alpha \circ F^{j-2} \alpha \circ \dots \circ F^0 \alpha$ holds for all $0 \leq j \leq i$

Induction Start For $j = 0$, the claim holds trivially, because then the term

$$F^{j-1} \alpha \circ F^{j-2} \alpha \circ \dots \circ F^0 \alpha \text{ has length } 0 \text{ and we just have } d_i^A = F^0 d_{i-0}^A.$$

Induction Hypothesis $d_i^A = F^j d_{i-j}^A \circ F^{j-1} \alpha \circ F^{j-2} \alpha \circ \dots \circ F^0 \alpha$ is true for all

$$0 \leq j < k < i$$

Induction Step We can now compute, starting with the induction hypothesis:

$$\begin{aligned}
d_i^A &= F^j d_{i-j}^A \circ F^{j-1} \alpha \circ F^{j-2} \alpha \circ \dots \circ F^0 \alpha \\
&= F^j (F d_{i-j-1}^A \circ \alpha) \circ F^{j-1} \alpha \circ F^{j-2} \alpha \circ \dots \circ F^0 \alpha \\
&= F^{j+1} d_{i-j-1}^A \circ F^j(\alpha) \circ F^{j-1} \alpha \circ F^{j-2} \alpha \circ \dots \circ F^0 \alpha
\end{aligned}$$

So we can conclude in particular $d_i^A = F^i d_0^A \circ F^{i-1} \alpha \circ F^{i-2} \alpha \circ \dots \circ F^0 \alpha$. Now we show that for all $j < i$ it holds that

$$\begin{aligned}
&U(F^{i-j} d_j^D \circ F^{i-j-1} \alpha \circ \dots \circ F^0 \alpha)(x) = U(F^{i-j} d_j^D \circ F^{i-j-1} \alpha \circ \dots \circ F^0 \alpha)(y) \\
&\Leftrightarrow U(F^{i-j-1} d_{j+1}^D \circ F^{i-j-2} \alpha \circ \dots \circ F^0 \alpha)(x) \\
&= U(F^{i-j-1} d_{j+1}^D \circ F^{i-j-2} \alpha \circ \dots \circ F^0 \alpha)(y)
\end{aligned}$$

For that purpose we observe that $U(F^{i-j} d_j^D) \equiv^{U(F^{i-j} X)} U(F^{i-j} e_j^D)$, so we can use the fact that in **Set** it holds that $f \equiv g \Rightarrow (f(x) = f(y) \Leftrightarrow g(x) = g(y))$ and we can compute:

$$\begin{aligned}
&U(F^{i-j} d_j^D \circ F^{i-j-1} \alpha \circ \dots \circ F^0 \alpha)(x) = U(F^{i-j} d_j^D \circ F^{i-j-1} \alpha \circ \dots \circ F^0 \alpha)(y) \\
&\Leftrightarrow U(F^{i-j} d_j^D) \circ U(F^{i-j-1} \alpha \circ \dots \circ F^0 \alpha)(x) \\
&= U(F^{i-j} d_j^D) \circ U(F^{i-j-1} \alpha \circ \dots \circ F^0 \alpha)(y) \\
&\Leftrightarrow U(F^{i-j} e_j^D) \circ U(F^{i-j-1} \alpha \circ \dots \circ F^0 \alpha)(x) \\
&= U(F^{i-j} e_j^D) \circ U(F^{i-j-1} \alpha \circ \dots \circ F^0 \alpha)(y) \\
&\Leftrightarrow U(F^{i-j} e_j^D \circ F^{i-j-1} \alpha \circ \dots \circ F^0 \alpha)(x) = U(F^{i-j} e_j^D \circ F^{i-j-1} \alpha \circ \dots \circ F^0 \alpha)(y) \\
&\Leftrightarrow U(F^{i-j-1} (F e_j^D \circ \alpha) \circ F^{i-j-2} \alpha \circ \dots \circ F^0 \alpha)(x) \\
&= U(F^{i-j-1} (F e_j^D \circ \alpha) \circ F^{i-j-2} \alpha \circ \dots \circ F^0 \alpha)(y) \\
&\Leftrightarrow U(F^{i-j-1} d_{j+1}^D \circ F^{i-j-2} \alpha \circ \dots \circ F^0 \alpha)(x) \\
&= U(F^{i-j-1} d_{j+1}^D \circ F^{i-j-2} \alpha \circ \dots \circ F^0 \alpha)(y)
\end{aligned}$$

Observe that the formula degenerates to $U d_i^D(x) = U d_i^D(y)$ for $j = i - 1$

So chaining these two arguments together we get

$$\begin{aligned}
&U d_i^A(x) = U d_i^A(y) \\
&\Leftrightarrow U(F^i d_0^A \circ F^{i-1} \alpha \circ \dots \circ F^0 \alpha)(x) = U(F^i d_0^A \circ F^{i-1} \alpha \circ \dots \circ F^0 \alpha)(y) \\
&\Leftrightarrow U(F^i d_0^D \circ F^{i-1} \alpha \circ \dots \circ F^0 \alpha)(x) = U(F^i d_0^D \circ F^{i-1} \alpha \circ \dots \circ F^0 \alpha)(y) \\
&\Leftrightarrow^* U(d_i^D)(x) = U(d_i^D)(y)
\end{aligned}$$

□

It is easy to see that Algorithm B is an instance of Algorithm D since we have already shown that the functor property preserves \leq^X . Moreover, in Algorithm C, the $F^n m_i^D$ are mapped to monomorphisms by U .

Lemma 3.3.12 *Let (\mathbf{C}, U) be a concrete category and F an endofunctor on \mathbf{C} . Furthermore, let $d = m \circ e$ be an arrow \mathbf{C} , such that $UF^n m$ is a mono for all $n \in \mathbb{N}_0$ and Ud has non-empty codomain. Then $UF^n e \equiv UF^n d$ for all $n \in \mathbb{N}_0$.*

Proof: Due to functoriality it holds that $F^n d = F^n m \circ F^n e$, so it suffices to show that a function f exists such that $f \circ UF^n d = UF^n e$. Since $UF^n m$ is a mono, i.e. an injective function, it holds that the pre-image of any element $x \in \text{Im}(UF^n m)$ is unique, written as $UF^n m^{-1}(x)$. Let $y \in \text{cod}(UF^n d)$ be given arbitrarily, then we can define

$$f(x) = \begin{cases} UF^n m^{-1}(x) & \text{if } x \in \text{Im}(UF^n m) \\ y & \text{otherwise} \end{cases}$$

We can then compute $f \circ UF^n d(x) = UF^n m^{-1} \circ UF^n m \circ UF^n e(x) = UF^n e(x)$.

□

We can also see that, assuming U and F preserve monos, a suitable choice for factorisation in Algorithm D is to factor out any monomorphisms. A factorisation structure therefore is not required for the purpose of checking behavioural equivalence using Algorithm C. Thus, in practice, one could forego the proof of the diagonalisation property in a factorisation structure and still compute behavioural equivalence as in Algorithm C. This condition was already weakened in [ABH⁺12], where instead of a factorisation structure a pseudo-factorisation structure, i.e. a factorisation structure in a reflective subcategory, was considered. However, the diagonalisation property is not completely redundant, because it guarantees uniqueness up to isomorphism of the representative coalgebra that is returned when the algorithms terminate. In general, this does not hold when factoring out arbitrary monomorphisms.

Another case, where Algorithm D can easily be applied, is when the coalgebra functor F has a lifting \overline{F} to **Set**. In this case, the complicated condition of $UF^j d_i^D \equiv UF^j e_i$ for all $j \in \mathbb{N}_0$ collapses to just showing that $Ud_i^D \equiv Ue_i^D$. Note, however, that in many cases functors in Kleisli categories arise as extensions of

Set-functors, in which case it is uncommon, that also a lifting to **Set** exists. So to conclude, the following table shows a collection of conditions that individually are valid strategies to choose representatives in Algorithm D and therefore give rise to an instance of Algorithm D.

Choice of Factor e	Conditions on F	Conditions on U
Any arrow $e \equiv^X d$	-	-
Factorise $d = m \circ e$ using $(\mathcal{E}, \mathcal{M})$	Preserves \mathcal{M}	Maps \mathcal{M} to monos
Factor out any mono m , $d = m \circ e$	Preserves monos	Preserves monos
Choose any e s.t. $Ue \equiv^{UX} Ud$	Lifts to functor on Set	-

In this table, the first line corresponds to Algorithm B and the second line corresponds to Algorithm C. The remaining lines outline additional strategies for Algorithm D that do not warrant a distinct name here, because we will not use those strategies in our examples unless they coincide with Algorithm B or Algorithm C.

Termination: The algorithms naturally terminate whenever \equiv^X has only finitely many equivalence classes. In **Set** this is the case for finite sets X . However, there are other categories where this condition holds for certain objects X : in **Vect**, \equiv^X is of finite index, whenever X is a finite-dimensional vector space.

Note that it is not sufficient to iterate until the partition induced by Ud_i is not refined from one step to the next. We will discuss weighted automata in the next section and the tropical semiring provides an example where behavioural equivalence is undecidable, but all steps of Algorithm B are computable and the partition induced by Ud_i can only be refined finitely often, rendering this criterion invalid.

Remark 3.3.13 *In the algorithm for behavioural equivalence, if we interpret the arrows d_i^D as partitioning the set of states UX into the equivalence classes $[\bar{x}]_i := \{\bar{y} \in UX \mid Ud_i^D(\bar{x}) = Ud_i^D(\bar{y})\}$, then*

- d_0^D maps every state $\bar{x} \in UX$ into a single equivalence class.
- In each step, the equivalence classes remain unchanged or some equivalence classes are split into several equivalence classes, i.e., the equivalence is refined.

However, we differ from classical partition refinement in two regards: first, we have to remember not only the partition, but also the arrows d_i^D in order to perform the next step. Second, the termination condition is different from usual partition refinement algorithms, because it need not hold that whenever in one step the equivalence classes are not refined, they will stay unchanged forever. This follows from the undecidability of the language equivalence problem for weighted automata over the tropical semiring [Kro94], where the algorithm does not necessarily terminate even if the state set is finite.

However, if we work with a category in **Set**, we obtain a classical partition refinement algorithm. In this case the arrows d_i^D are functions and two functions are equivalent whenever they induce the same partition (see Example 3.2.3). Here, we should choose surjections as representatives. We look into deterministic automata over an alphabet A to get some intuition: if we consider the functor $FX = \{0, 1\} \times X^A$, then a coalgebra $\alpha: X \rightarrow FX$ describes a deterministic automaton where to each state we assign $\alpha(x) = (o, \delta_x)$, where $o \in \{0, 1\}$ indicates whether the state is final and $\delta_x: A \rightarrow X$ returns the successor state $\delta_x(a)$ for each alphabet symbol a . It is well-known that in this case coalgebraic behavioural equivalence coincides with language equivalence (and bisimilarity). Applying our algorithms yields the well-known minimisation algorithm for deterministic automata [Hop71], based on partition refinement. Furthermore, in this case m_{n+1}^D is unique (up to isomorphism) and coincides with the minimised automaton.

3.4 Applications to Various Automata Models

In this section we will demonstrate the applicability of the final chain based algorithms to several different automata models. We will start by reviewing the classical examples of language equivalence for deterministic automata and bisimulation of labelled transition systems. We will also demonstrate that a modification of the modelling technique allows us to decide a variety of bisimilarity notions in the presence of silent transitions, this is a technique developed in [BK17]. We will then describe the motivating example of weighted automata. However, we will not go into detail with regards to algorithmic issues when instantiating to weighted automata, more details about this can be found

in Chapter 4. We will conclude this case study by showing that in addition, a technique developed in [FMT05] to determine behavioural equivalence in history dependent automata can be simulated by Algorithm A. Another case study for conditional transition systems can be found in Chapter 7, we will just note here that Algorithm A (without the factorisation) and Algorithm D (using the factorisation) can be used in the way described in Chapter 7, whereas for Algorithm B the factors e_i could in general not be chosen as small as with Algorithm D, so Algorithm B is not a good choice for CTS.

Even though Algorithm B is always applicable when Algorithm A is – by choosing \mathcal{R} to be the set of all arrows – we will only reference Algorithm B as applicable explicitly if a significant improvement over Algorithm A is possible by a suitable choice of \mathcal{R} . Naturally, Algorithm D is always applicable whenever either Algorithm B or Algorithm C are applicable, so we forego making this fact explicit.

3.4.1 Deterministic Automata and Labelled Transition Systems: The Classical Cases

Both deterministic automata (with final states, but without initial states) and labelled transition systems can be modelled in the base category **Set** – which is trivially concrete using the identity functor as concretisation.

Definition 3.4.1 (Deterministic Automata and LTS Coalgebraically)

*Deterministic automata over a finite set of actions A can be expressed as coalgebras for the following **Set**-functor:*

Objects $DX = X^A \times \{0, 1\}$, i.e. each set X is mapped to a set of pairs (f, t) where $t \in \{0, 1\}$ and f is a function $f: A \rightarrow X$.

Arrows Let $f: X \rightarrow Y$ be any **Set**-arrow, i.e. function, then $Df: X^A \times \{0, 1\} \rightarrow Y^A \times \{0, 1\}$ is the function $Df(g, t) = (f \circ g, t)$ that preserves the second component of a pair and precomposes f with the first component.

*In a similar way the **Set**-functor L that can be used to model LTS can be defined over a finite set of actions A :*

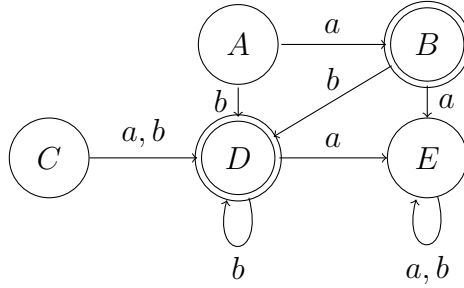
Objects $LX = \mathcal{P}(X)^A$, i.e. each set X is mapped to the set of functions $f: A \rightarrow \mathcal{P}(X)$.

Arrows *Let $f: X \rightarrow Y$ be any **Set**-arrow, i.e. function, then $Lf: \mathcal{P}(X)^A \rightarrow \mathcal{P}(Y)^A$ is the function $Lf(g) = f \circ g$ where a function is applied to a subset of its domain component-wise.*

A deterministic automaton can be seen as a D -coalgebra in the following way: A D -coalgebra is a function $\alpha: X \rightarrow X^A \times \{0, 1\}$. The value $\pi_2^2(\alpha(x))$ indicates whether the state x is final – it is if $\pi_2^2(\alpha(x)) = 1$ – and $\pi_1^2(\alpha(x))(a)$ is the a successor of x . In turn, an L -coalgebra can be identified as an LTS in the following way: An L -coalgebra is a function $\alpha: X \rightarrow \mathcal{P}(X)^A$, where each state x is mapped to a function $f: A \rightarrow \mathcal{P}(X)$ that maps each action $a \in A$ to the set of successor states of x under a . We illustrate both classical cases – which are well-known case studies for coalgebraic modelling – together, because they share a suitable factorisation structure, namely the set of all surjective functions as \mathcal{E} and the set of all injective functions as \mathcal{M} . Additionally, \mathcal{E} can be chosen as a class of representatives, so for the classical cases, both Algorithm B and C are applicable and coincide. Note, that our previously discussed (in Chapter 2) example of non-deterministic automata is defined in a Kleisli category over **Set** rather than **Set**, if one is interested in language equivalence, and is subsumed by the weighted case we will discuss hereafter, thus we will not discuss an example for this third classical type of systems here.

In both cases, the notion of behavioural equivalence is bisimilarity. In the case of deterministic automata, bisimulation and language equivalence coincide, so here, in contrast to non-deterministic automata and weighted automata, we do not need a Kleisli-category to hide side effects in order to obtain language equivalence. We will illustrate Algorithm B (and hence, Algorithm C) on one example for deterministic automata and one example for labelled transition systems:

Example 3.4.2 *Consider the following deterministic finite automaton over the input alphabet $\{a, b\}$ and the state set $\{A, B, C, D, E\}$:*



For this example, we will write elements $(f, t) \in Y^{\{a,b\}} \times \{0, 1\}$ as $(f(a), f(b), t)$ to get a compact representation. The automaton can be modelled by the coalgebra $\alpha: X \rightarrow X^{\{a,b\}} \times \{0, 1\}$ defined as follows:

$$\alpha(A) = (B, D, 0) \quad \alpha(B) = (E, D, 1) \quad \alpha(C) = (D, D, 0)$$

$$\alpha(D) = (E, D, 1) \quad \alpha(E) = (E, E, 0)$$

Now we can use Algorithm B (or C):

- Let $d_0: X \rightarrow \{\bullet\}$ be the function mapping into the single element set $\{\bullet\}$.
- We compute $d_1 = Fd_0 \circ \alpha: X \rightarrow \{\bullet\}^{\{a,b\}} \times \{0, 1\}$ where

$$d_1(A) = (\bullet, \bullet, 0) \quad d_1(B) = (\bullet, \bullet, 1) \quad d_1(C) = (\bullet, \bullet, 0)$$

$$d_1(D) = (\bullet, \bullet, 1) \quad d_1(E) = (\bullet, \bullet, 0)$$

We could choose e_1 to be just d_1 , because d_1 is surjective, but since the

- symbols do not carry any information, we will choose the function $e_1: X \rightarrow \{0, 1\}$ where $e_1(A) = e_1(C) = e_1(E) = 0$ and $e_1(B) = e_1(D) = 1$ instead. Then $m_1: \{0, 1\} \rightarrow \{\bullet\}^{\{a,b\}} \times \{0, 1\}$ is defined as $m_1(0) = (\bullet, \bullet, 0)$ and $m_1(1) = (\bullet, \bullet, 1)$

- We compute $d_2 = Fe_1 \circ \alpha: X \rightarrow \{0, 1\}^{\{a,b\}} \times \{0, 1\}$ where

$$d_2(A) = (1, 1, 0) \quad d_2(B) = (0, 1, 1) \quad d_2(C) = (1, 1, 0)$$

$$d_2(D) = (0, 1, 1) \quad d_2(E) = (0, 0, 0)$$

d_2 is not surjective, so we need to factorise d_2 by simply taking

$$e_2: X \rightarrow \{(1, 1, 0), (0, 1, 1), (0, 0, 0)\}$$

where $e_2(i) = d_2(i)$ for all $i \in \{A, B, C, D, E\}$ and

$$m_2: \{(1, 1, 0), (0, 1, 1), (0, 0, 0)\} \rightarrow \{0, 1\}^{\{a,b\}} \times \{0, 1\}$$

as the obvious injection.

- We compute $d_3 = Fe_2 \circ \alpha: X \rightarrow \{(1, 1, 0), (0, 1, 1), (0, 0, 0)\}^{\{a,b\}} \times \{0, 1\}$, where

$$d_3(A) = ((0, 1, 1), (0, 1, 1), 0) \quad d_3(B) = ((0, 0, 0), (0, 1, 1), 1)$$

$$d_3(C) = ((0, 1, 1), (0, 1, 1), 0) \quad d_3(D) = ((0, 0, 0), (0, 1, 1), 1)$$

$$d_3(E) = ((0, 0, 0), (0, 0, 0), 0)$$

Since d_3 induces the same partition on X as d_2 , there exists the function

$$f: \{(1, 1, 0), (0, 1, 1), (0, 0, 0)\} \rightarrow \{(1, 1, 0), (0, 1, 1), (0, 0, 0)\}^{\{a,b\}} \times \{0, 1\}$$

defined according to

$$f(1, 1, 0) = ((0, 1, 1), (0, 1, 1), 0) \quad f(0, 1, 1) = ((0, 0, 0), (0, 1, 1), 1)$$

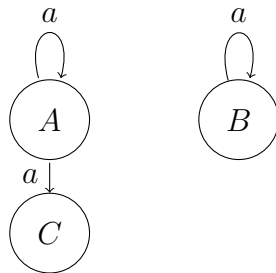
$$f(0, 0, 0) = ((0, 0, 0), (0, 0, 0), 0)$$

such that $f \circ e_2 = d_3$, so we can stop the computation.

The final result e_2 now induces language equivalence in the following way: All states that have the same image under e_2 are language equivalent and all states that have different images under e_2 are not language equivalent. In fact, we can read the acceptance behaviour for all words of length up to 1 from e_2 and the acceptance behaviour for all words of length up to 2 from d_3 . Take for instance $d_3(D) = ((0, 0, 0), (0, 1, 1), 1)$, that can be read as follows: Due to the last 1, we can see that D accepts the empty word. The first tuple $(0, 0, 0)$ now yields information about the acceptance behaviour of D for all words that start with an a : The words aa , ab and a are all rejected. Similarly, one can read from the second tuple $(0, 1, 1)$ the acceptance behaviour for words starting with a b . ba is rejected, bb and b are accepted by D .

Now we can turn our attention to an example for labelled transition systems.

Example 3.4.3 We consider the labelled transition system over the set of actions $\{a\}$ depicted below:



As a coalgebra, this LTS can be modelled as follows:

$\alpha: \{A, B, C\} \rightarrow \mathcal{P}(\{A, B, C\})^{\{a\}}$	
(A, a)	$\{A, C\}$
(B, a)	$\{B\}$
(C, a)	\emptyset

Now we can compute:

$d_0: \{A, B, C\} \rightarrow \{\bullet\}$	
A	\bullet
B	\bullet
C	\bullet

This function is surjective, so $e_0 = d_0$.

$d_1: \{A, B, C\} \rightarrow \mathcal{P}(\{\bullet\})^{\{a\}}$	
(A, a)	$\{\bullet\}$
(B, a)	$\{\bullet\}$
(C, a)	\emptyset

This function is surjective, so $e_1 = d_1$.

$d_2: \{A, B, C\} \rightarrow \mathcal{P}(\mathcal{P}(\{\bullet\})^{\{a\}})^{\{a\}}$	
(A, a)	$\{[a \mapsto \{\bullet\}], [a \mapsto \emptyset]\}$
(B, a)	$\{[a \mapsto \{\bullet\}]\}$
(C, a)	\emptyset

d_2 is not surjective, so we have to factorise and obtain

$$e_2: \{A, B, C\} \rightarrow \{\{[a \mapsto \{\bullet\}], [a \mapsto \emptyset]\}, \{[a \mapsto \{\bullet\}]\}, \emptyset\}^{\{a\}}$$

where $e_2(X, a) = d_2(X, a)$ for all $X \in \{A, B, C\}$.

$$m_2: \{\{[a \mapsto \{\bullet\}], [a \mapsto \emptyset]\}, \{[a \mapsto \{\bullet\}]\}, \emptyset\}^{\{a\}} \rightarrow \mathcal{P}(\mathcal{P}(\{\bullet\})^{\{a\}})^{\{a\}}$$

is just the obvious injection.

$d_3: \{A, B, C\} \rightarrow \mathcal{P}(\{\{[a \mapsto \{\bullet\}], [a \mapsto \emptyset]\}, \{[a \mapsto \{\bullet\}]\}, \emptyset\}^{\{a\}})^{\{a\}}$	
(A, a)	$\{[a \mapsto \{\{[a \mapsto \{\bullet\}], [a \mapsto \emptyset]\}, [a \mapsto \emptyset]\}$
(B, a)	$\{[a \mapsto \{\{[a \mapsto \{\bullet\}]\}]\}$
(C, a)	\emptyset

Now there exists a function

$f: \{\{[a \mapsto \{\bullet\}], [a \mapsto \emptyset]\}, \{[a \mapsto \{\bullet\}]\}, \emptyset\}^{\{a\}} \\ \rightarrow \mathcal{P}(\{\{[a \mapsto \{\bullet\}], [a \mapsto \emptyset]\}, \{[a \mapsto \{\bullet\}]\}, \emptyset\}^{\{a\}})^{\{a\}}$	
$[a \rightarrow \{[a \mapsto \{\bullet\}]\}]$	$[a \rightarrow \{[a \mapsto \{[a \mapsto \{\bullet\}]\}]\}]$
$[a \rightarrow \{[a \mapsto \{\bullet\}], [a \mapsto \emptyset]\}]$	$[a \rightarrow \{[a \mapsto \{[a \mapsto \{\bullet\}], [a \mapsto \emptyset]\}], [a \mapsto \emptyset]\}]$
$[a \mapsto \emptyset]$	$[a \mapsto \emptyset]$

such that $f \circ e_2 = d_3$, so the algorithm terminates.

From e_2 we can now see that all three states of the LTS are not bisimilar. We can also observe the possible behaviour from e_2 . C can do no steps, A always has the choice of doing one last a -step (via the function $[a \rightarrow \emptyset]$) or to do an a -step and again have the option to do further a -steps. B can always do further A steps. Thus C and A, B can be separated in just one step, but A and B only get separated after two steps, because after one step, an execution starting from A may have no further steps available whereas an execution starting from B can always do more a -steps.

3.4.2 Branching Bisimulation for LTS

LTS can also be defined in the presence of silent transitions. The idea behind silent transitions is that these are internal transitions the system can take, which are not visible to an observer. Of course, adding silent transitions to the set of actions A would allow us to model LTS with silent transitions just fine, but the notion of behavioural equivalence would not regard silent transitions any different from any other kind of transition. In the presence of silent transitions, several different notions of bisimilarity have been proposed that handle silent transitions differently. In a joint work with Harsh Beohar, [BK17] we have investigated means of expressing LTS using a different functor that takes paths rather than single transitions into consideration. Using this functor and appropriate modelling of the LTS, we can obtain four commonly investigated notions of bisimulation in the presence of silent transitions, namely branching, delay, η and weak bisimulation. For these notions, we can again make use of the factorisation structure $(\mathcal{E}, \mathcal{M})$ of surjective and injective functions and thus Algorithm B and C coincide again. However, here, we will solely focus on branching bisimulation and will not consider weak, η and delay bisimulation.

We will give a quick overview on how to model LTS using the *Path*-functor to obtain branching bisimulation. Weak, η and delay bisimulation can be modelled in a similar way. Since this only serves as an illustration of the final chain technique, we will not provide the proof that this way of modelling LTS actually yields the right notion of bisimulation as behavioural equivalence. The interested reader can find the corresponding proofs in [BK17].

We will write A_τ to denote the set of all non-silent actions A together with a distinct silent action τ .

We first need to define the notions of path and execution which are essential to our modelling technique:

Definition 3.4.4 (Path)

- A path p on a set X is a sequence of type $X(A_\tau X)^*$.
- We say a path p can be reduced to a path $p' = x_1 a_1 x_2 \dots a_{n-1} x_n$ if there exist indices i_1, \dots, i_n such that $p = x_1 (\tau x_1)^{i_1} a_1 x_2 (\tau x_2)^{i_2} \dots a_{n-1} x_n (\tau x_n)^{i_n}$, i.e. p' can be obtained by removing τ -loops from p .
- For any two given paths p and q , we say they are equivalent $p \sim q$ if and only if they can be reduced to the same path r .
- The endofunctor Path_\sim on **Set** is defined as follows:

Objects Let X be any set, then $\text{Path}_\sim(X)$ is the set of all equivalence classes of paths on X .

Arrows Let $f: X \rightarrow Y$ be any function and $[p] \in \text{Path}_\sim(X)$ any equivalence class of paths, then $\text{Path}_\sim(f)([p]) = [f \circ p]$.

- An execution is a path p where for all subsequences (x, a, x') of p , there is an a transition from x to x' .

We will identify minimal paths, i.e. paths that cannot be further reduced, with their equivalence class of paths. Intuitively, a representative of an equivalence class of paths can easily be identified, for it is the (unique) path in the equivalence class that does not contain any τ -loops. Before specifying how to define LTS using this functor, we will first define branching bisimulation.

Definition 3.4.5 (Branching Bisimulation) *Let an LTS (X, \rightarrow) over A_τ be given, where τ is considered a silent transition. Then, a relation $R \subseteq X \times X$ is called a branching bisimulation, if for all pairs of states $(x_1, x_2) \in R$ and all action $a \in A$ it holds that*

- *If $x_1 \xrightarrow{a} x'_1$ then there exist states x'_2, x''_2 such that $x_2 \xrightarrow{\tau^*} x'_2$, $x'_2 \xrightarrow{a} x''_2$, where $(x_1, x'_2) \in R$ and $(x'_1, x''_2) \in R$.*
- *Vice versa, if $x_2 \xrightarrow{a} x'_2$ then there exist states x'_1, x''_1 such that $x_1 \xrightarrow{\tau^*} x'_1$, $x'_1 \xrightarrow{a} x''_1$, where $(x'_1, x_2) \in R$ and $(x''_1, x'_2) \in R$.*
- *If $x_1 \xrightarrow{\tau} x'_1$, then $x_2 \xrightarrow{\tau^*} x'_2 \xrightarrow{\tau} x''_2$ where $(x_1, x'_2) \in R$ and $(x'_1, x''_2) \in R$.*
- *Vice versa, if $x_2 \xrightarrow{\tau} x'_2$, then $x_1 \xrightarrow{\tau^*} x'_1 \xrightarrow{\tau} x''_1$ where $(x_2, x'_1) \in R$ and $(x'_2, x''_1) \in R$.*

Conceptually, branching bisimulation treats states as equivalent if their observable behaviour is the same and no internal action can take away choices on observable behaviour.

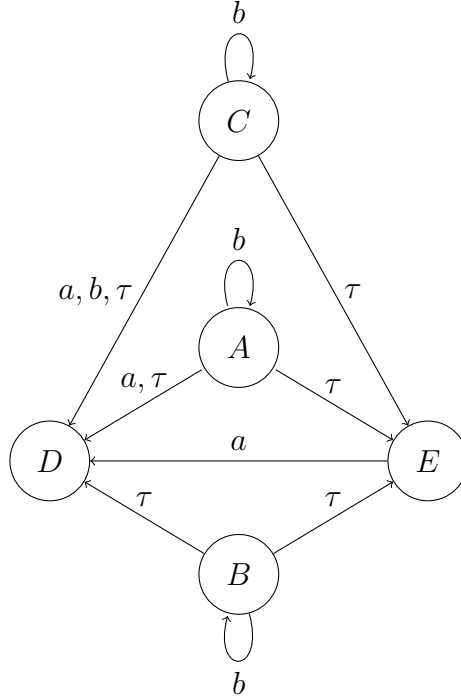
We can now explain how to define a labelled transition system as a $\mathcal{P} \circ \text{Path}_\sim$ coalgebra in order to obtain branching bisimilarity as behavioural equivalence. For that purpose, let an LTS (X, A_τ, \rightarrow) be given. We model (X, A_τ, \rightarrow) coalgebraically as $\alpha: X \rightarrow \mathcal{P}(\text{Path}_\sim(X))$ as follows:

$$\begin{aligned} \alpha(x) = \{ & [p] \mid \text{dom}(p) = \tau^n a, \\ & \forall i < n : p(\tau^i) \xrightarrow{\tau} p(\tau^{i+1}), \\ & p(\tau^n) \xrightarrow{a} p(\tau^n a), a \in A_\tau \} \\ & \cup \{ [x] \mid x \in X \} \end{aligned}$$

Intuitively, for branching bisimulation, all τ -sequences are made explicit.

Since we are working in the category of **Set**, and both \mathcal{P} and Path_\sim preserve monomorphisms, we can use the factorisation structure $(\mathcal{E}, \mathcal{M})$ where \mathcal{E} is the set of all surjective functions and \mathcal{M} is the set of all injective functions to apply Algorithm B or Algorithm C and both algorithms coincide. We will illustrate the application of the algorithm on a small example. If the reader is familiar with all four notions of bisimilarity in the presence of silent transitions, he may observe that this example distinguishes all four notions.

Example 3.4.6 Consider the set of actions $A_\tau = \{a, b, \tau\}$ that contains two non-silent transition a and b . We define a transition system over A_τ and the state set $X = \{A, B, C, D, E\}$ as depicted below:



We will now model this system using the $\mathcal{P} \circ \text{Path}_\sim$ functor in order to obtain branching bisimilarity as the behavioural equivalence and compute Algorithm B (or, equivalently, Algorithm C). We will identify each equivalence class of paths with its paths and write down the computed sets of paths explicitly. The paths that are redundant are printed in grey. Redundant in this context means that there exists another representative of the same equivalence class in the set.

To obtain branching bisimulation as behavioural equivalence, we model the transition system by the following table:

	$\alpha: X \rightarrow \mathcal{P}(\text{Path}_\sim(X))$
A	$\{(A), (A, b, A), (A, a, D), (A, \tau, D), (A, \tau, E), (A, \tau, E, a, D)\}$
B	$\{(B), (B, b, B), (B, \tau, E), (B, \tau, E, a, D), (B, \tau, D)\}$
C	$\{(C), (C, b, C), (C, a, D), (C, b, D), (C, \tau, D), (C, \tau, E), (C, \tau, E, a, D)\}$
D	$\{(D)\}$
E	$\{(E), (E, a, D)\}$

This model consists of the paths corresponding to each single-step transition and all τ^* -transitions, including the ε -loops.

We will illustrate the intermediate results of Algorithm B in triples of tables: In each step an arrow (i.e. a function) d_i is computed, then a representative e_i is chosen and there exists another arrow m_i such that $m_i \circ e_i = d_i$. Each of these functions are again depicted in tables. Note that we always factorise by just naming the different images with natural numbers instead of restricting the function to its image. The reason for this is that the visual representation of the intermediate results would otherwise end up unwieldy large already from the second step.

$d_0 = e_0: X \rightarrow \{\bullet\}$	
A	•
B	•
C	•
D	•
E	•

$m_0: \{\bullet\} \rightarrow \{\bullet\}$	
•	•

d_0 is just the unique (up to iso) function mapping all states to •. Therefore, d_0 is surjective and $e_0 = d_0$.

$d_1: X \rightarrow \mathcal{P}(\text{Path}_{\sim}(\{\bullet\}))$	
A	$\{(\bullet), (\bullet, b, \bullet), (\bullet, a, \bullet), (\bullet, \tau, \bullet), (\bullet, \tau, \bullet), (\bullet, \tau, \bullet, a, \bullet)\}$
B	$\{(\bullet), (\bullet, b, \bullet), (\bullet, \tau, \bullet), (\bullet, \tau, \bullet), (\bullet, \tau, \bullet, a, \bullet)\}$
C	$\{(\bullet), (\bullet, b, \bullet), (\bullet, a, \bullet), (\bullet, b, \bullet)(\bullet, \tau, \bullet)\}$
D	$\{(\bullet)\}$
E	$\{(\bullet), (\bullet, a, \bullet)\}$

$e_1: X \rightarrow \{1, 2, 3\}$	
A	1
B	1
C	1
D	2
E	3

$m_1: \{1, 2, 3\} \rightarrow \mathcal{P}(\text{Path}_{\sim}(\{\bullet\}))$	
1	$\{(\bullet), (\bullet, a, \bullet), (\bullet, b, \bullet)\}$
2	$\{(\bullet)\}$
3	$\{(\bullet), (\bullet, a, \bullet)\}$

Observe that in the table for d_1 there are several entries that are printed in grey. This is to illustrate, which paths we need not consider further, because another representative of the same path has already been found. E.g., the path $(\bullet, \tau, \bullet, a, \bullet)$ is not reduced, it contains a τ -loop. In fact, it is equivalent to (\bullet, a, \bullet) , so we only keep the representative (\bullet, a, \bullet) . In a similar way, all other grey entries can be found to be just non-minimal representatives of equivalence

classes already present in the entry. Note that for the image of B , we left the non-minimal element $(\bullet, \tau, \bullet, a, \bullet)$ in the table to make it easier to see which paths are generated by composition with α . It turns out that the states A, B, C are all mapped to the same set of equivalence classes of paths, so after factorisation, only three different images remain in e_1 .

$d_2: X \rightarrow \mathcal{P}(\text{Path}_\sim(\{1, 2, 3\}))$	
A	$\{(1), (1, b, 1), (1, a, 2), (1, \tau, 2), (1, \tau, 3), (1, \tau, 3, a, 2)\}$
B	$\{(1), (1, b, 1), (1, \tau, 3), (1, \tau, 3, a, 2), (1, \tau, 2)\}$
C	$\{(1), (1, b, 1), (1, a, 2), (1, b, 2), (1, \tau, 2), (1, \tau, 3), (1, \tau, 3, a, 2)\}$
D	$\{(2)\}$
E	$\{(3), (3, a, 2)\}$

$e_2: X \rightarrow \{1, 2, 3, 4, 5\}$	
A	1
B	2
C	3
D	4
E	5

$m_2: \{1, 2, 3, 4, 5\} \rightarrow \mathcal{P}(\text{Path}_\sim(\{1, 2, 3\}))$	
1	$\{(1), (1, b, 1), (1, a, 2), (1, \tau, 2), (1, \tau, 3), (1, \tau, 3, a, 2)\}$
2	$\{(1), (1, b, 1), (1, \tau, 3), (1, \tau, 3, a, 2), (1, \tau, 2)\}$
3	$\{(1), (1, b, 1), (1, a, 2), (1, b, 2), (1, \tau, 2), (1, \tau, 3), (1, \tau, 3, a, 2)\}$
4	$\{(2)\}$
5	$\{(3), (3, a, 2)\}$

In d_2 , all states now have been separated and found not to be branching bisimilar. We can stop here, because d_3 is just a renaming of α , which again has five different equivalence classes for the five states. We can conclude that no pair of states in this automaton is branching bisimilar.

3.4.3 Weighted Automata

We will show that we can apply our approach to weighted automata with weights taken from a semiring. Weighted automata [DKV09] are a versatile formalism to specify and analyse systems equipped with quantitative information. They are a generalisation of non-deterministic automata (NFA), but instead of just

accepting or rejecting words over a given alphabet, weighted automata assign values taken from a semiring to words.

Coalgebraic treatment for weighted automata was already discussed in [BMS13, BBB⁺12], but making use of different categories. Here, we propose a category closely related to the ideas of [HJS07], which shows how to obtain trace or language equivalence by working in a Kleisli category.

We will mainly consider Algorithm B. The algorithm will not terminate for every possible choice of the semiring (for example it is known that language equivalence is undecidable for tropical semirings). However, we will characterise the cases for which it terminates.

Definition 3.4.7 (Category of Matrices, Linear Maps) *We consider a category $\mathbf{M}(\mathbb{S})$ of \mathbb{S} -matrices where objects are sets and an arrow $a: X \rightarrow Y$ is an $X \times Y$ -matrix as defined in Definition 2.1.7. Arrow composition is performed by matrix multiplication, i.e., for $a: X \rightarrow Y$, $b: Y \rightarrow Z$ we have $b \circ a = a \cdot b$. The identity arrow $\text{id}_X: X \rightarrow X$ is the $X \times X$ unit matrix.*

For $x \in X$, the unit vector $\hat{x} \in \mathbb{S}^X$ is defined as $\hat{x}(x) = 1$ and $\hat{x}(y) = 0$ if $y \neq x$.

There exists a concretisation functor U where $UX = \mathbb{S}^X$ and for $a: X \rightarrow Y$ $Ua: UX \rightarrow UY$ is the linear map from \mathbb{S}^X to \mathbb{S}^Y represented by the matrix a , i.e., $Ua(\bar{x}) = \bar{x}^t \cdot a$ for all $\bar{x} \in \mathbb{S}^X$. (Note that \bar{x}^t , the transpose of \bar{x} , is a row vector which is identified with the corresponding $1 \times X$ -matrix.)

Therefore, $(\mathbf{M}(\mathbb{S}), U)$ is a concrete category. In the sequel we will only reference $\mathbf{M}(\mathbb{S})$, but whenever a concrete category is required, it is understood to be $(\mathbf{M}(\mathbb{S}), U)$.

We can now define weighted automata in a coalgebraic notation. Note that, different from the weighted automata in [DKV09], our automata do not have initial states, as it is customary in coalgebra, and hence no initial weights, but only final weights. However, for language equivalence we can easily simulate initial weights by adding a new state to the automaton with edges going to each state of the automaton, carrying the initial weights of these states and labelled with some (new) symbol.

Definition 3.4.8 (Weighted Automaton Coalgebraically) *Let $\mathbf{M}(\mathbb{S})$ be the category defined above. Let A be a finite set of alphabet symbols. We define*

an endofunctor $F: \mathbf{M}(\mathbb{S}) \rightarrow \mathbf{M}(\mathbb{S})$ as follows: on objects¹ $FX = A \times X + 1$ for a set X . For an arrow $f: X \rightarrow Y$ we have $Ff: A \times X + 1 \rightarrow A \times Y + 1$ where

- $Ff((a, x), (a, y)) = f(x, y)$ for $a \in A, x \in X, y \in Y$,
- $Ff(\bullet, \bullet) = 1$
- $Ff(c, d) = 0$ for all remaining $c \in A \times X + 1, d \in A \times Y + 1$.

A weighted automaton is an F -coalgebra, i.e., an arrow $\alpha: X \rightarrow FX$ in the category $\mathbf{M}(\mathbb{S})$ or, alternatively, an $(X \times FX) = X \times (A \times X + 1)$ -matrix with entries from \mathbb{S} .

For a weighted automaton α , $\alpha(x, \bullet)$ denotes the final weight of state $x \in X$ and $\alpha(x, (a, y))$ denotes the weight of the a -transition from x to y .

We now want to show that language equivalence for weighted automata coincides with behavioural equivalence. For this purpose, we restate the definition of the language of a weighted automaton. Note, that we restrict to finite-state weighted automata, here, because otherwise this definition may not be well-defined, considering it is then potentially based on an infinite sum.

Definition 3.4.9 (Language of a Weighted Automaton [DKV09])

Let (X, α) be a weighted automaton over alphabet A , a semiring \mathbb{S} and a finite state set X . The language $L_\alpha: A^* \rightarrow \mathbb{S}^X$ of α is recursively defined as

- $L_\alpha(\varepsilon)(x) = \alpha(x, \bullet)$
- $L_\alpha(aw)(x) = \sum_{x' \in X} \alpha(x, (a, x')) \cdot L_\alpha(w)(x')$ for $a \in A, w \in A^*$

We will call $L_\alpha(w)(x)$ the weight that state x assigns to the word w . Two states $x, y \in X$ are language equivalent if $L_\alpha(w)(x) = L_\alpha(w)(y)$ for all $w \in A^*$.

It can be shown that two states $x, y \in X$ of a finite weighted automaton (X, α) are behaviourally equivalent in the coalgebraic sense ($x \sim y$) if and only if they assign the same weight to all words, i.e. $L_\alpha(w)(x) = L_\alpha(w)(y)$ for all $w \in A^*$.

¹Here $X + Y$ denotes the disjoint union of two sets X, Y and 1 stands for the singleton set $\{\bullet\}$.

Proposition 3.4.10 *Let $(X, \alpha: X \rightarrow FX)$, where X is a finite set, be a weighted automaton over the finite alphabet A and the semiring S . Then, two states $x, y \in X$ are language equivalent if and only if \hat{x}, \hat{y} are behaviourally equivalent ($\hat{x} \sim \hat{y}$).*

Proof: The first part of the proof is based on two lemmas from Section 3.3.

- Assume \hat{x}, \hat{y} are behaviourally equivalent, but x, y are not language equivalent. Then there is a coalgebra homomorphism $f: (X, \alpha) \rightarrow (Y, \beta)$ such that $Uf(\hat{x}) = Uf(\hat{y})$ (i.e., $\hat{x}^t \cdot f = \hat{y}^t \cdot f$, which means that the rows in f corresponding to x, y coincide), but there is also a word w such that $L_\alpha(w)(x) \neq L_\alpha(w)(y)$.

Note that the sequence of matrices $d_0^A, d_1^A, d_2^A, \dots$ that we obtain when applying Algorithm A to a weighted automaton have the following property: For any index i , d_i^A is a matrix that has one column for each word w' of length $j < i$, which contains for each row index $x \in X$ the weight that is assigned to w' by state x . In Subsection 4.2.5 a bit more intuition about this property is given.

Thus, $U(d_{|w|+1}^A)(\hat{x}) \neq U(d_{|w|+1}^A)(\hat{y})$, i.e. the rows corresponding to x, y in $d_{|w|+1}^A$ differ (see Lemma 4.2.17). According to Lemma 3.3.3 we have $d_{|w|+1}^A \geq^X f$ (i.e., there exists a matrix a such that $d_{|w|+1}^A = f \cdot a$), which is a contradiction since $U(d_{|w|+1}^A)(\hat{x}) \neq U(d_{|w|+1}^A)(\hat{y})$ and $Uf(\hat{x}) = Uf(\hat{y})$.

- Assume x, y are language equivalent. We need to show that there exists a coalgebra homomorphism $f: (X, \alpha) \rightarrow (Y, \beta)$ such that $Uf(\hat{x}) = Uf(\hat{y})$. We choose $Y = A^*$ and β is an $A^* \times (A \times A^* + 1)$ -matrix with $\beta(aw, (a, w)) = 1$, $\beta(\varepsilon, \bullet) = 1$, all other entries of β are 0. Then, β is obviously an arrow of $\mathbf{M}(S)$, since in every column (a, w) only the entry in row aw is different from 0 and in the column \bullet only the row ε is different from 0.

Now, we choose $f: X \rightarrow A^*$ as the matrix with the entries $f(z, w) = L_\alpha(w)(z)$, where $w \in A^*$, $z \in X$. Note, that f is well-defined, because we assume that X is finite. Also, since x and y are language equivalent, per definition of f we have that the rows corresponding to x and y in f coincide.

It is left to be shown that that f is a coalgebra homomorphism, i.e. we need to prove that $Ff \circ \alpha = \beta \circ f$. The matrix $Ff \circ \alpha$ has dimension $X \times (A \times A^* + 1)$, so we compute for an arbitrary $z \in X$, $a \in A$, $w \in A^*$:

$$\begin{aligned} & (Ff \circ \alpha)(z, (a, w)) \\ &= \sum_{(a', z') \in A \times X} \alpha(z, (a', z')) \cdot Ff((a', z'), (a, w)) + \alpha(z, \bullet) \cdot Ff(\bullet, (a, w)) \\ &= \sum_{z' \in X} \alpha(z, (a, z')) \cdot f(z', w) = f(z, aw) \end{aligned}$$

as well as:

$$\begin{aligned} & \beta \circ f(z, (a, w)) = \sum_{w' \in A^*} f(z, w') \cdot \beta(w', (a, w)) \\ &= \sum_{w' = aw} f(z, w') \cdot \beta(w', (a, w)) = f(z, aw) \cdot \beta(aw, (a, w)) = f(z, aw) \end{aligned}$$

For an arbitrary $z \in X$ we can compute:

$$\begin{aligned} & (Ff \circ \alpha)(z, \bullet) \\ &= \sum_{(a', z') \in A \times X} \alpha(z, (a', z')) \cdot Ff((a', z'), \bullet) + \alpha(z, \bullet) \cdot Ff(\bullet, \bullet) \\ &= \alpha(z, \bullet) \cdot Ff(\bullet, \bullet) = \alpha(z, \bullet) = f(z, \varepsilon) \end{aligned}$$

as well as:

$$\begin{aligned} & (\beta \circ f)(z, \bullet) = \sum_{w' \in A^*} f(z, w') \cdot \beta(w', \bullet) \\ &= \sum_{w' = \varepsilon} f(z, w') \cdot \beta(w', \bullet) = f(z, \varepsilon) \cdot \beta(\varepsilon, \bullet) = f(z, \varepsilon) \end{aligned}$$

So we indeed have $Ff \circ \alpha = \beta \circ f$.

□

Due to the change of category (from **Set** to $\mathbf{M}(\mathbb{S})$) we obtain language equivalence instead of a notion of bisimilarity. We will see that $\mathbf{M}(\mathbb{S})$ is closely related to a Kleisli category over the semiring monad (see also [HJS07] which explains the effects that the implicit branching or side-effects of the monad have on behavioural equivalence).

Relation to Kleisli Categories

Definition 3.4.11 (Kleisli Category of Matrices) *We consider the category $\mathbf{Kl}(S)$ for the monad (S, ν, μ) on **Set**, defined according to:*

Objects: $SX = \mathbb{S}^X$, i.e. functions of finite support from X to \mathbb{S} .

Arrows: Let $f: X \rightarrow Y$ be any arrow of **Set**, then we define $Sf: \mathbb{S}^X \rightarrow \mathbb{S}^Y$ according to $Sf(a)(y) = \sum \{a(x) \mid x \in X, f(x) = y\}$ for any $a \in \mathbb{S}^X$, $y \in Y$.

Unit: $\eta_X: X \rightarrow SX$ is defined according to

$$\eta_X(x)(y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

for all $x, y \in X$.

Multiplication: $\mu_X: SSX \rightarrow SX$ is defined according to

$$\mu_X(m)(x) = \sum \{m(f) \cdot f(x) \mid f \in \mathbb{S}^X\}$$

for all $m \in \mathbb{S}^{\mathbb{S}^X}$ and $x \in X$.

It is a standard result that (S, η, μ) is a monad. If the reader is interested in how to prove this, a proof can be found in Appendix A, Lemma A.1.

We will now discuss how it can be seen that $\mathbf{Kl}(S)$ coincides with $\mathbf{M}(\mathbb{S})$ for all arrows that correspond to functions of finite support. Note, that the categories $\mathbf{Kl}(S)$ and $\mathbf{M}(\mathbb{S})$ are not isomorphic, because $\mathbf{Kl}(S)$ has finite row support, whereas $\mathbf{M}(\mathbb{S})$ has finite column support. Beyond this subsection, we will continue working with $\mathbf{M}(\mathbb{S})$, but we will limit our discussion to matrices of finite dimension, so for the cases that are of interest for us, both categories act the same.

To show that $\mathbf{Kl}(S)$ and $\mathbf{M}(\mathbb{S})$ coincide for finite carrier sets X and Y , we first have a look at arrows in $\mathbf{Kl}(S)$. An arrow $f: X \rightarrow Y$ is a function $f: X \rightarrow \mathbb{S}^Y$ where \mathbb{S}^Y is a function of finite support. Now, assume additionally, that X and Y are finite. Then we can curry and obtain a function of type $F: X \times Y \rightarrow \mathbb{S}$, which can be interpreted as an $X \times Y$ -matrix with (trivially) finite row and column support. Consequently, $\eta_X: X \times X \rightarrow \mathbb{S}$ can be seen as the $X \times X$ unit-matrix. Next, we will show that arrow composition coincides with matrix multiplication.

Let $f: X \times Y$ and $g: Y \times Z$ be two arrows in $\mathbf{Kl}(S)$, i.e. f is a $Y \times X$ matrix and g is a $Z \times Y$ matrix. Then the multiplication $g \cdot f$ is defined and for any

$x \in X, z \in Z$ we get

$$f \cdot g(x, z) = \sum_{y \in Y} f(x, y) \cdot g(y, z).$$

On the other hand,

$$\begin{aligned} g \circ f(x)(z) &= \mu_Z Sg \circ f(x)(z) = \sum_{h: Z \rightarrow \mathbb{S}} \{Sg(f(x))(h) \cdot h(z)\} \\ &= \sum_{h: Z \rightarrow \mathbb{S}} \{\sum \{f(x)(y) \mid y \in Y, g(y) = h\} \cdot h(z)\} = \sum \{f(x)(y) \cdot g(y)(z) \mid y \in Y\}. \end{aligned}$$

So indeed, matrix multiplication corresponds to the concatenation of arrows in the following way: $g \circ f = f \cdot g$. This shows that $\mathbf{Kl}(\mathbb{S})$ coincides with $\mathbf{M}(\mathbb{S})$ for all arrows between finite sets.

Adapting Proposition 3.4.10 for $\mathbf{Kl}(S)$ Note, that in the second part of the proof of Proposition 3.4.10, we used the fact that arrows in $\mathbf{M}(\mathbb{S})$ can have infinite row support – in order to be able to define the coalgebra homomorphism.

Interestingly, whenever Algorithm A terminates in either category, it also terminates in the other category and the result is the same. Indeed, the proof of Proposition 3.4.10 can then be adapted for $\mathbf{Kl}(S)$ in the following way:

Assume the semimodule generated by A^* in Proposition 3.4.10 is finitely generated, then the proof of Proposition 3.4.10 can be adapted to $\mathbf{Kl}(S)$, by restricting the carrier set of the coalgebra in the image to a finite set of words $A^{\leq n}$ for some $n \in \mathbb{N}_0$. This can be seen as follows. Let n be chosen such that the semimodule generated by $A^{\leq n}$ is the same as the semimodule generated by A^* , which exists by the assumption that the latter is finitely generated. Now for any $w \in A^{n+1}$, the vector $(L_{x_1}(w), L_{x_2}(w), \dots, L_{x_m}(w))^2$ can be obtained via a (not necessarily unique) linear combination of all vectors $(L_{x_1}(w'), L_{x_2}(w'), \dots, L_{x_m}(w'))$, where $w' \in A^{\leq n}$. For each word $w \in A^{n+1}$, choose one such linear combination $C(w): A^{\leq n} \rightarrow \mathbb{S}$. Now, $\beta: A^{\leq n} \rightarrow FA^{\leq n}$ may be defined just as β from the proof of Proposition 3.4.10 except for the cases $\beta(w', (a, w))$, where $|w| = n$. In those cases, let $\beta(w', (a, w)) = C(aw)(w')$. Then $f: X \rightarrow A^{\leq n}$, defined just as in the proof of Proposition 3.4.10, can be shown to be a coalgebra homomorphism in the same way as in the proof of

²Here, we assume $X = \{x_1, \dots, x_m\}$

Proposition 3.4.10. The sole exception is considering $(\beta \circ f)(z, (a, w))$, where $|w| = n$. Here, the computation is slightly more involved, we obtain

$$\begin{aligned} \sum_{w' \in A^{\leq n}} f(z, w') \cdot \beta(w', (a, w)) &= \sum_{w' \in A^{\leq n}} L_\alpha(z)(w') \cdot \beta(w', (a, w)) \\ &= \sum_{w' \in A^{\leq n}} L_\alpha(z)(w') \cdot C(aw)(w') = L_\alpha(z)(aw). \end{aligned}$$

The corresponding computation for $Ff \circ \alpha$ remains unchanged.

Preorder and Representatives From now onwards we consider only finite index sets. We will study the category $\mathbf{M}(\mathbb{S})$ and show what the preorder on arrows (see Definition 3.2.2) means in this setting.

If the semiring is a field, there exists a factorisation structure which factors every matrix into a matrix of full row rank and a matrix of full column rank. This factorisation is unique up-to isomorphism. However, for generic semirings, semimodule theory does not provide a similarly elegant notion of base as in vector spaces, and such unique factorisations are not possible in general. Hence Algorithm C is usually not applicable.

Proposition 3.4.12 *Let $a: X \rightarrow Y$, $b: X \rightarrow Z$ be two arrows in $\mathbf{M}(\mathbb{S})$, i.e., a is an $X \times Y$ -matrix and b is an $X \times Z$ -matrix. It holds that $a \leq^X b \iff \langle a \rangle \subseteq \langle b \rangle$. That is, two matrices a, b are ordered if and only if the subsemimodule spanned by a is included in the subsemimodule spanned by b . Hence also $a \equiv^X b \iff \langle a \rangle = \langle b \rangle$.*

Proof: Let $a \leq^X b$, hence there exists a $Y \times Z$ -matrix c with $a \cdot c = b$. This means that the columns of b (indexed by X) arise as linear combinations of the columns of a . Thus, every vector in $\langle b \rangle$ can be generated from the vectors in $\langle a \rangle$.

For the other direction, assume that $\langle a \rangle \subseteq \langle b \rangle$. This means that all column vectors of a , which are included in $\langle a \rangle$, must also be included in $\langle b \rangle$. Hence they can be obtained as linear combinations of the column vectors of b , which means that there exists a matrix c with $a \cdot c = b$ and hence $a \leq^X b$. \square

For *weighted automata*, a class of representatives \mathcal{R} must have the property that for every subsemimodule of \mathbb{S}^X there exists a matrix $a \in \mathcal{R}$ generating this subsemimodule, i.e., the subsemimodule corresponds to $\langle a \rangle$.

Depending on the semiring, the class of representatives can be rather simple, for instance for fields, one could choose matrices corresponding to the bases of vector spaces of dimension X . In general, there is no notion of basis for semimodules over a semiring. However, a suitable choice for the class of representatives is the class of all generating matrices without redundant column vectors.

Definition 3.4.13 (Class of Representatives for Weighted Automata)

We define \mathcal{R} as the class of all matrices a that do not contain a column that is a linear combination of the other columns of a .

We will now give a sufficient condition and some examples that guarantee termination of Algorithm B. The corresponding proofs can be found in Subsection 4.2.5.

Proposition 3.4.14 *If $\langle G^* \rangle = \{L_\alpha(w) \mid w \in A^*\}$ for a weighted automaton (X, α) is finitely generated, Algorithm B terminates.*

If \mathbb{S} is a *field*, $\langle G^* \rangle$ is always finitely generated, since semimodules over fields are vector spaces and the algorithm coincides with earlier work by Schützenberger [Sch61]. Similar algorithms were presented in [Bor09, ABH⁺12]. A similar argument also holds for skew-fields, structures that are almost fields, but where multiplication need not be commutative, as shown by Flouret and Laugerotte, [DK13, FL97], extending Schützenberger’s result. A further extension can be found in [BLS06, BLS05], extending the decidability result to principal ideal domains such as $(\mathbb{Z}, +, \cdot)$. In Chapter 4, we will show that the algorithm is also strongly related to conjugacy, investigated for instance in [Sak09].

We can easily specify some classes where the algorithm necessarily terminates. If \mathbb{S} is a *finite semiring*, the algorithm terminates, since there are only finitely many different column vectors of a fixed dimension $|X|$. If \mathbb{S} is a *distributive complete lattice*, the algorithm will terminate as well.

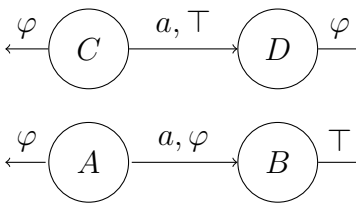
Corollary 3.4.15 *Algorithm B always terminates on weighted automata with weights from a distributive complete lattice.*

Proving termination does not mean that Algorithm B is effectively computable in itself, because for this it is also necessary to be able to decide whether

two matrices are equivalent, i.e., whether a vector is generated by a given set of generators. This need not be decidable in general, but it is decidable in all the cases above.

We now consider several examples.

Example 3.4.16 We use as a semiring the complete distributive lattice $\mathbb{L} = \{\top, \perp, \varphi, \psi\}$ where $\perp \leq \varphi \leq \top$, $\perp \leq \psi \leq \top$ and consider the labelled transition system (X, α) , $X = \{A, B, C, D\}$ with weights over \mathbb{L} and labels from $\{a\}$ represented by the transition matrix and automaton (transitions with weight \perp are omitted):

$$\alpha = \begin{matrix} & \begin{matrix} a, A & a, B & a, C & a, D & \bullet \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \end{matrix} & \begin{pmatrix} \perp & \varphi & \perp & \perp & \varphi \\ \perp & \perp & \perp & \perp & \top \\ \perp & \perp & \perp & \top & \varphi \\ \perp & \perp & \perp & \perp & \varphi \end{pmatrix} \end{matrix}$$


We apply Algorithm B to this automaton. Below e_i denotes the chosen representative in \mathcal{R} .

- We start with $d_0 = e_0$, a 4×0 -matrix

$$\bullet Fe_0 = \begin{pmatrix} \perp \\ \perp \\ \perp \\ \perp \\ \top \end{pmatrix}, d_1 = Fe_0 \cdot \alpha = \begin{pmatrix} \varphi \\ \top \\ \varphi \\ \varphi \end{pmatrix}. \text{ This is in } \mathcal{R}, \text{ so } e_1 = d_1.$$

$$\bullet Fe_1 = \begin{pmatrix} \varphi & \perp \\ \top & \perp \\ \varphi & \perp \\ \varphi & \perp \\ \perp & \top \end{pmatrix}, d_2 = Fe_1 \cdot \alpha = \begin{pmatrix} \varphi & \varphi \\ \perp & \top \\ \varphi & \varphi \\ \perp & \varphi \end{pmatrix}. \text{ This is in } \mathcal{R}, \text{ so } e_2 = d_2.$$

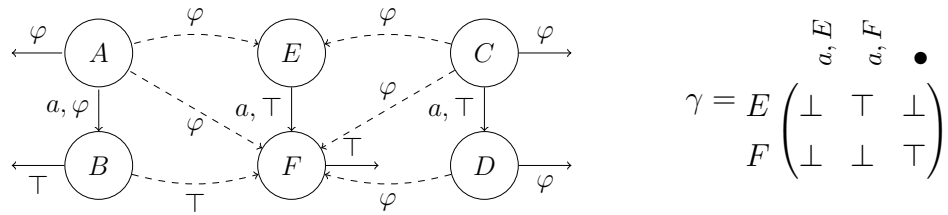
$$\bullet Fe_2 = \begin{pmatrix} \varphi & \varphi & \perp \\ \perp & \top & \perp \\ \varphi & \varphi & \perp \\ \perp & \varphi & \perp \\ \perp & \perp & \top \end{pmatrix}, d_3 = Fe_2 \cdot \alpha = \begin{pmatrix} \perp & \varphi & \varphi \\ \perp & \perp & \top \\ \perp & \varphi & \varphi \\ \perp & \perp & \varphi \end{pmatrix}.$$

This is not in \mathcal{R} , we can for example see that the first row equals \perp times the second plus \perp times the third row. So we factorise:

$$d_3 = e_3 \cdot \gamma = \begin{pmatrix} \varphi & \varphi \\ \perp & \top \\ \varphi & \varphi \\ \perp & \varphi \end{pmatrix} \cdot \begin{pmatrix} \perp & \top & \perp \\ \perp & \perp & \top \end{pmatrix}$$

Now $e_3 = e_2$ and we can stop our computation.

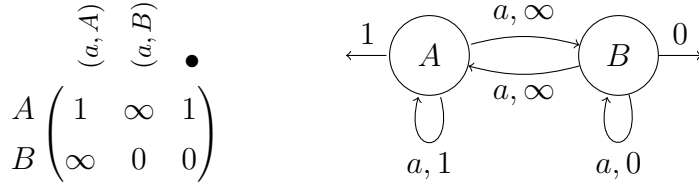
The resulting automaton is a two-state automaton, where the states will be called E, F . Looking at e_3 we can see that the states A and C of X are equivalent, since their columns coincide (in fact, both accept ε and a with weight φ). States B and D on the other hand are not equivalent to any other state. We see that $e_2: X \rightarrow Y$, where $Y = \{E, F\}$, is a coalgebra homomorphism from (X, α) to (Y, γ) . The coalgebra (Y, γ) can be considered as a minimal representative of (X, α) . The following diagram depicts automata (X, α) and (Y, γ) where the coalgebra homomorphism e_2 is drawn with dashed lines.



We can also see that our method of factoring is not unique, because in γ we could have chosen $\gamma(E, (x, E)) = \psi$ (and all other entries as before). In this case, there would be an a, ψ -loop on state E in the diagram above. Since all dashed arrows going into E carry weight φ and $\varphi \sqcap \psi = \perp$, this loop would not have any effect and the equivalence one obtains is the same.

In the next example, we will investigate the tropical semiring (cf. Example 2.1.4). We will write \oplus and \otimes for the \mathbb{S} -addition respectively the \mathbb{S} -multiplication to avoid confusion. Language equivalence is in general undecidable, hence the algorithm can not terminate in general.

Example 3.4.17 Consider the (rather simple) transition system over the one-letter alphabet $\{a\}$, given by the matrix α :



Applying Algorithm B to α , we obtain the following (intermediate) results:

- $d_0 = e_0$ is the 2×0 -matrix
- $F e_0 = \begin{pmatrix} \infty \\ \infty \\ 0 \end{pmatrix}$, $d_1 = F e_0 \cdot \alpha = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = e_1$
- $d_2 = \begin{pmatrix} 2 & 1 \\ 0 & 0 \end{pmatrix} \in \mathcal{R}$, so we choose $e_2 = d_2$.
- $d_3 = \begin{pmatrix} 3 & 2 & 1 \\ 0 & 0 & 0 \end{pmatrix} \notin \mathcal{R}$, because we can obtain the second column as a linear combination of the first and the third column:

$$1 \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \oplus 0 \otimes \begin{pmatrix} 3 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 3 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

However, we cannot obtain the first row via linear combination of the other two rows, so the algorithm cannot stop in the third iteration.

So we can choose: $m_3 = \begin{pmatrix} 0 & 0 & \infty \\ \infty & 1 & 0 \end{pmatrix}$ and $e_3 = \begin{pmatrix} 3 & 1 \\ 0 & 0 \end{pmatrix}$

- From now on, each step is analogous to the third step, we obtain $e_i = \begin{pmatrix} i & 1 \\ 0 & 0 \end{pmatrix}$ in each iteration i , but we will never reach a d_{i+1} such that $d_{i+1} \equiv^X e_i$.

Algorithm B therefore does not terminate for α . However, since the two states are already separated from the first step onwards, we can at least conclude that the states are not behaviourally equivalent.

We have seen, for an arbitrary semiring, it is possible that the algorithm does not terminate. If it terminates, it always yields behavioural equivalence, if it does not terminate, we can still look at the intermediate results and use this as a means to semi-decide if two states are not equivalent. If two states ever get

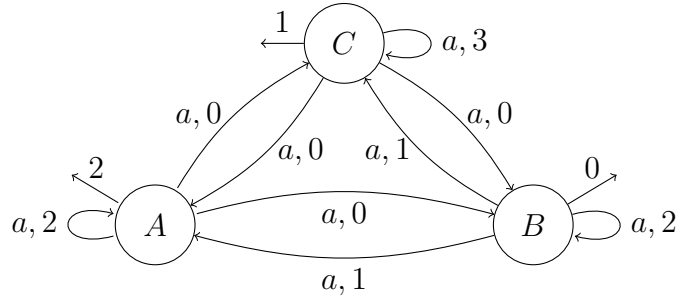
separated, they are not behaviourally equivalent, if they never get separated, they are behaviourally equivalent.

This example also demonstrates that the termination condition for Algorithm B can be not met – and, since both algorithms terminate at the same time, neither is Algorithm A’s –, although we already have all information regarding behavioural equivalence. Even from the first step onwards in the non-terminating example, both states are proven not to be language equivalent, but Algorithm B’s termination condition is never met. However, for an arbitrary semiring we cannot find a termination condition that guarantees that the algorithm always terminates whenever all states, that are not behaviourally equivalent, are separated. Since for two states to not be language equivalent, there needs to be some word they assign a different weight to, which can be found in a finite span of time, such a termination condition would yield a decision algorithm for the undecidable case of the tropical semiring, as well.

We have already seen that Algorithm B does not terminate for some automata over the tropical semiring, we will investigate another example where the algorithm does terminate. So even if there is no termination guarantee, it is reasonable to apply the algorithm even to semirings where termination is not guaranteed in general.

Example 3.4.18 Consider the automaton over the single-letter alphabet $\{a\}$ given by the matrix α below.

$$\begin{array}{c} \begin{array}{ccc} (a, A) & (a, B) & (a, C) \\ A & \begin{pmatrix} 2 & 0 & 0 & 2 \end{pmatrix} \\ B & \begin{pmatrix} 1 & 2 & 1 & 0 \end{pmatrix} \\ C & \begin{pmatrix} 0 & 0 & 3 & 1 \end{pmatrix} \end{array} \bullet \end{array}$$



- $d_0 = e_0$ is the 0×3 -matrix

- $d_1 = \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} = e_1$

$$\begin{aligned}
\bullet \ d_2 &= \begin{pmatrix} 0 & 2 \\ 2 & 0 \\ 0 & 1 \end{pmatrix} = e_2 \\
\bullet \ d_3 &= \begin{pmatrix} 0 & 0 & 2 \\ 1 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 2 \\ 2 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & \infty \\ 1 & \infty & 0 \end{pmatrix} = e_2 \cdot \gamma
\end{aligned}$$

So the algorithm terminates and we can observe that none of the three states are equivalent, since all columns in e_3 are different.

3.4.4 HD-Automata

History-dependent automata (HD-automata) are an automaton model for name passing calculi such as the π -calculus. States and transitions as in traditional non-deterministic automata are enhanced with a notion of names, that are present in each transition. Since names can be generated over the course of the execution of a name passing calculus (e.g. using the ν operator in the π -calculus) or changed (e.g. using the renaming operator in π -calculus), names are dealt with in an abstract way in HD-automata, as natural numbers and permutations on natural numbers, allowing for renaming.

In [FMT05], a coalgebraic way of modelling HD-automata, as well as a final chain based algorithm to compute behavioural equivalence has been presented. HD-automata were defined by specifying a category they live in, the category of named sets and named functions. For one specific kind of HD-automata, those modelling π -calculus, Ferrari et al. have furthermore specified a proper choice of functor to model them coalgebraically. The algorithm to decide behavioural equivalence, which is early bisimulation in the specific case considered in [FMT05], is specified independently of the concrete choice of functor.

We will quickly recall the definitions from [FMT05] using our notations and will then show that Algorithm A can simulate the proposed algorithm. Note, that names are abstracted as natural numbers and a change in names is realised by permutation in this formalism.

Definition 3.4.19 (Category \mathbf{H}) *The category \mathbf{H} is defined as follows:*

Objects *Objects are named sets $A = (Q_A, \lesssim_A, |_|_A, \mathcal{G}_A)$ where*

- Q_A is a set
- $\lesssim_A \subseteq Q_A \times Q_A$ is a total order
- $|_A| \in Q_A \rightarrow \overline{\mathbb{N}}$, here, $\overline{\mathbb{N}}$ denotes the set of natural numbers excluding 0 but including ∞ .
- \mathcal{G}_A maps $q \in Q$ to a finite permutation group on $\{x \mid x \leq |q|_A\}$ – note that $|q|_A$ may be infinity.

Arrows Arrows $f: D \rightarrow C$ are named functions $f = (D, C, h, \Sigma)$ where

- $D = (Q_D, \lesssim_D, |_D, \mathcal{G}_D)$
- $C = (Q_C, \lesssim_C, |_C, \mathcal{G}_C)$
- $h: Q_D \rightarrow Q_C$ is a function
- Σ maps $q \in Q_D$ to a finite set of functions $\sigma: \{x \mid x \leq |h(q)|_C\} \rightarrow \{x \mid x \leq |q|_D\}$ such that
 1. for all $\sigma \in \Sigma(q)$ it holds that $\sigma \circ \mathcal{G}_C(h(q)) = \Sigma(q)$
 2. for all $\sigma \in \Sigma(q)$ it holds that $\mathcal{G}_D(h(q)) \circ \sigma \subseteq \Sigma(q)$
 3. all $\sigma \in \Sigma(q)$ are injective

Composition Let $f = (D_f, C_f, h_f, \Sigma_f)$ and $g = (D_g, C_g, h_g, \Sigma_g)$ be given such that $C_f = D_g$, i.e. g and f can be composed to form $g \circ f$, then $g \circ f$ is defined according to

$$g \circ f = (D_f, C_g, h_g \circ h_f, \Sigma_{g \circ f})$$

$$\text{where } \Sigma_{g \circ f}(q) = \Sigma_f(q) \circ \Sigma_g(h_f(q))$$

Identity arrows The identity arrows have the form $\text{id}_D = (D, D, \text{id}_D, \Sigma_{\text{id}_D})$

$$\text{where } \Sigma_{\text{id}_D}(q) = \text{id}_{\{x \mid x \leq |q|\}}.$$

Intuitively, a named set A can be understood as a set of states Q_A , together with a number of names specified by $|_A$ and a permutation on the set of names for each state q that is the symmetry of q , i.e. the set of permutations on names that do not change the process modelled by q . Consider for instance the π -calculus process $P(x, y, z) = (\nu c)(\bar{c}\langle x \rangle.0 + \bar{c}\langle y \rangle.0 + \bar{c}\langle z \rangle.\bar{c}\langle z \rangle.0)$. The state representing P , q_P , has three names x, y, z , so $|q_P|_A = 3$. Moreover, the process P is independent of the order of x and y , but exchanging x for z will possibly

change the behaviour of P . Thus, $\mathcal{G}_A(q_P)$ contains the identity permutation as well as the permutation exchanging x and y . The order \lesssim is used to identify a (canonical) representative state among a given set of states. This is important, because a transition in an HD-automaton yields a set of possible successor states, of which one needs to be selected.

A named function $f = (D, C, h, \Sigma)$ in turn maps states q from its domain D to states q' from its codomain C via h such that Σ translates the names belonging to q to names belonging to q' in a way that respects the symmetries of q and q' . So all symmetries of q must be translated to symmetries of q' , symmetries of q are not generating any additional transitions (which is basically mirroring our intuitive understanding of the symmetries) and no two names get conflated by a transition.

The final chain based algorithm proposed in [FMT05] uses an order to determine termination. To define this order, we need the notion of the kernel of a function $f: X \rightarrow Y$, where we define

$$\ker f = \{\{y \in X \mid f(x) = f(y)\} \mid x \in X\}.$$

This notion can also be lifted to named functions $f = (D, C, h, \Sigma)$ in the following way: $\ker(f) = (\ker(h), \lesssim, |_|, \mathcal{G})$ where $|A| = |h(a)|_C$ and $\mathcal{G}(A) = \mathcal{G}_C(h(a))$ for any $a \in A$. The order \lesssim is defined by $A \lesssim B$ if and only there exist $a \in A, b \in B$ such that $a \lesssim_C b$, so we extend \lesssim_C to sets in the typical way. Now we can define a preorder on named functions according to:

Definition 3.4.20 (Preorder on Named Functions)

Let $f_1 = (D, C_1, h_1, \Sigma_1)$, $f_2 = (D, C_2, h_2, \Sigma_2)$ be two named functions with a common domain. We write $f_1 \preceq f_2$ if and only if the following conditions hold on their kernels $\ker(f_1) = (Q, \lesssim, |_|, \mathcal{G})$ and $\ker(f_2) = (Q', \lesssim', |_|', \mathcal{G}')$

1. The partition Q is coarser than the partition Q' in the following sense:

$$\forall A \in Q \exists B \in Q' : A \supseteq B$$
2. $\forall A \in Q, B \in Q' : A \cap B \neq \emptyset \Rightarrow |A| \leq |B|'$
3. $\forall A \in Q, B \in Q', q \in A \cap B : \Sigma(q) \subseteq \Sigma'(q)$

In their paper, Ferrari et al. [FMT05] propose to iterate on the final chain until $f_n \preceq f_{n+1}$ and $f_{n+1} \preceq f_n$ in order to compute the greatest bisimulation

between two HD-automata defined coalgebraically in the category **H**. We will now show that then also $f_n \equiv f_{n+1}$, i.e. Algorithm A can simulate this procedure.

Lemma 3.4.21 *Let $f_1 = (D, C_1, h_1, \Sigma_1)$, $f_2 = (D, C_2, h_2, \Sigma_2)$ be two named functions such that $f_1 \preceq f_2$ and $f_2 \preceq f_1$. Then $f_1 \equiv_D f_2$.*

Proof: Throughout the proof, let $D = (Q_D, \lesssim_D, |_|_D, \mathcal{G}_D)$, $C_1 = (Q_1, \lesssim_1, |_|_1, \mathcal{G}_1)$ and $C_2 = (Q_2, \lesssim_2, |_|_2, \mathcal{G}_2)$.

We will only show that there exists a named function $f = (C_1, C_2, h, \Sigma)$ such that $f \circ f_1 = f_2$. Since the conditions are symmetric, it automatically follows that then $f_1 \equiv_D f_2$.

Let $e \in Q_2$ be chosen arbitrarily, then we can define h according to

$$h(b) = \begin{cases} h_2(a) & \text{if } \exists a \in Q_1 : h_1(a) = b \\ e & \text{otherwise} \end{cases}$$

This is well-defined, because h_1 and h_2 induce the same partition (using Condition 1 for \preceq in both directions), so $h_2(a)$ is independent of the choice of representative a such that $h_1(a) = b$.

Additionally, we define

$$\Sigma(b) = \begin{cases} \mathcal{G}_1(b) & \text{if } b \in \text{Im}(h_1) \\ \emptyset & \text{otherwise} \end{cases}.$$

Now we must show two things, that $f \circ f_1 = f_2$ and that f is a named function. We first show that $f \circ f_1 = f_2$, since this is quite immediate. The types match, because $f \circ f_1 = (D, C_2, h_f \circ h_n, \Sigma \circ \Sigma_1)$ and $f_2 = (D, C_2, h_2, \Sigma_2)$. Moreover, $(h \circ h_1)(a) = h(h_1(a)) = h_2(a)$. Moreover, we can compute

$$\Sigma_1(q) \circ \Sigma(h(q)) = \Sigma_1(q) \circ \mathcal{G}_1(h_1(q)) = \Sigma_1(q) = \Sigma_2(q)$$

The last step is true because Ferrari et al. have already shown that from $f_n \preceq f_{n+1}$ and $f_{n+1} \preceq f_n$ it follows that $\Sigma_1(q) = \Sigma_2(q)$ (for this step, Condition 2 and Condition 3 are needed), the second-to-last step is true because of the first condition on the last component of a named function (Condition 1). So we have seen that $f \circ f_1 = f_2$, it remains to be shown that f is actually a named function.

It is clear that D and C are named sets and that f is a function, so we need to show the three conditions on Σ . For this purpose let any $q \in Q_1$ be given.

1. Let any $\sigma \in \Sigma(q)$ be given, then, since $\Sigma(q)$ is not empty, it follows that $q \in \text{Im}(h_1)$. Therefore there exists an $a \in Q_D$ such that $h_1(a) = q$. Then we can compute

$$\begin{aligned} \sigma \circ \mathcal{G}_2(h(q)) &= \sigma \circ \mathcal{G}_2(h(h_1(a))) = \sigma \circ \mathcal{G}_2(h_2(a)) \\ &= \sigma \circ \mathcal{G}_1(h_1(a)) = \sigma \circ \mathcal{G}_1(q) = \mathcal{G}_1(q) \end{aligned}$$

2. Let any $\sigma \in \Sigma(q)$ be given, then

$$\mathcal{G}_1(q) \circ \sigma = \mathcal{G}_1(q) = \Sigma(q)$$

because $\mathcal{G}_1(q)$ is a permutation group and thus closed under composition.

3. Since $\Sigma(q)$ only contains permutations, i.e. bijections, all functions in $\Sigma(q)$ are necessarily injective.

□

Remark 3.4.22 *Note that \preceq does not correspond to \leq_D but rather \geq_D .*

In order to give a concrete example for the computation of Algorithm A for the case of HD-automata, we would need to specify the functor T from [FMT05], that allows to model π calculus processes coalgebraically. The definition of this functor is rather involved, but it is very specific to the π -calculus, so we finish our discussion of the example of HD-automata without considering a concrete example.

3.5 Conclusion

We have seen how the final chain construction gives rise to an algorithm to decide behavioural equivalence for coalgebras on concrete categories. Optimisations in the form of the choice of representatives or factoring out monomorphisms can lead to more compact intermediate results and thus to faster run times. This work is inspired by the minimisation techniques presented in [ABH⁺12], but the more flexible termination condition of equivalence of arrows can lead to a

faster termination. Moreover, the choice of representatives or the factoring out of monos instead of using factorisation structures allows the approach to be applied to a greater class of systems.

The motivating example for this are weighted automata, modelled in a Kleisli category to obtain language equivalence as the behavioural equivalence. We have seen how to model weighted automata as coalgebras and how the generic Algorithm B can be instantiated to this system model. The choice of representatives naturally coincides with the choice of a minimal generating set for a semi-module, which is a generalisation of a vector space to arbitrary semirings rather than fields. Naturally, Algorithm B does not necessarily terminate for all semirings, since language equivalence for weighted automata is undecidable. Provided linear combinations can be computed, it still yields a semi-decision procedure for language equivalence, even if the algorithm is not guaranteed to terminate.

Related Work Our work is closely related to [ABH⁺12] which uses factorisation structures in order to obtain generic algorithms for partition refinement. However, the algorithm in [ABH⁺12] could not handle general weighted automata over semirings, due to the absence of suitable factorisation structures.

[Sta09] also discusses several coalgebraic methods to check behavioural equivalence, however the paper focusses more on the relation-based view with fixpoint iteration to obtain bisimulations. Staton compares with the final chain and can prove that whenever the arrow $F^i!$ in the final chain is a mono, then the relation refinement sequence converges at the latest. In our examples, the algorithm usually terminates earlier, since we only need a relative inverse β of $F^i!$ wrt. d_i^A .

Bonsangue, Milius and Silva [BMS13, BBB⁺12] have also investigated language equivalence of weighted automata in a coalgebraic setting, where they use Eilenberg-Moore instead of Kleisli categories. They present an axiomatisation of weighted automata and give a set of sound and complete equational laws axiomatising language equivalence.

Recently, in [UH14], Urabe and Hasuo studied simulation and language inclusion for weighted automata, where the coalgebraic theory provides the basis for an efficient implementation.

We have already discussed the related work by Ferrari et al. [FMT05] in detail. The authors have shown how to model HD-automata coalgebraically and have given a final chain based algorithm to compute the greatest bisimulation of an HD-automaton. As we have seen, their algorithm can be seen as an instance of our Algorithm A.

If instantiated to the case of weighted automata, Algorithm B coincides with common algorithms that were developed directly for weighted automata. We have already seen this in particular for the case of fields, where it is the same procedure as the one described in [ABH⁺12] and also coincides with the work of Schützenberger [Sch61]. In the next chapter, we will focus more on the weighted automata point of view and will more thoroughly compare our algorithm to classic work on weighted automata, including in particular conjugacy based techniques, while also discussing specific (classes of) semirings, where the choice of representatives can be done effectively.

Chapter 4

Language Equivalence for Weighted Automata: An Instantiation of the Final Chain Algorithm

4.1 Introduction

Building on the general coalgebraic work of Chapter 3, we now want to take closer look into the application to the analysis of the language of weighted automata (Subsection 2.3.1). Due to the flexible nature of weighted automata – remember that the weights can come from any semiring and varying choice of semirings allow to express different things in a model – they are a powerful modelling tool that can express various distinct kinds of models. For instance, weights from the interval $[0, 1]$ are often used to express the probability that a transition takes place and weights from the tropical semiring (natural numbers with minimum as addition and addition as multiplication) can be used to express cost for a transition. On the flipside, weighted automata are computationally complex and as a result, language equivalence, which we are mainly interested in, is in general undecidable.

We aim to compute which pairs of states are language equivalent, i.e. which pairs of states in a given weighted automaton assign the same value to all words. This can be of interest for instance for optimisation purposes or for

checking if a system modelled as a weighted automaton behaves according to specification. As mentioned previously, in general this problem is undecidable, however, we have investigated it from a coalgebraic perspective and have found an algorithm as a special instance for a generic coalgebraic algorithm that allows to compute the language equivalence for various semirings where it is decidable. The coalgebraic view on this algorithm is presented in Chapter 3. Consequently, this chapter will only discuss its instantiation to weighted automata.

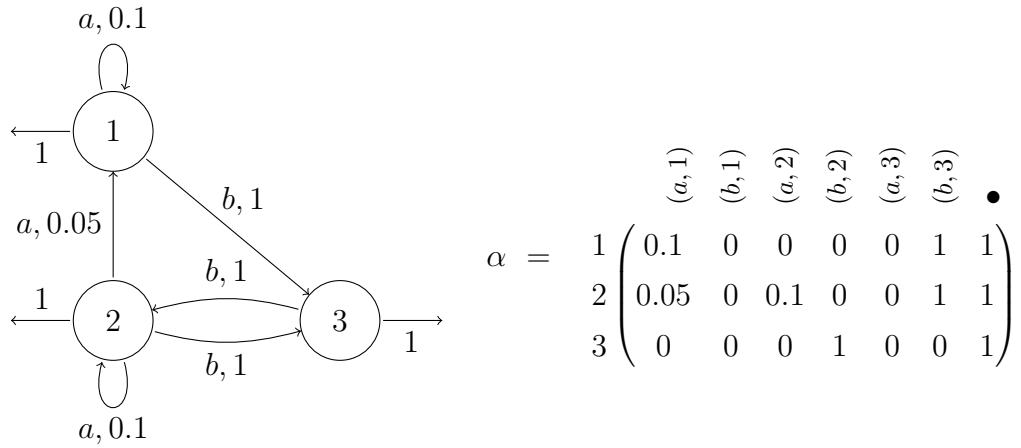
The instantiation yields an algorithm that does appear in the literature for specific semirings. For probabilistic automata, there is a related procedure for checking language equivalence [KMO⁺11], for fields, a method is discussed in [Bor09] and a result for rings, based on results by Schützenberger, is given in [DK13]. Division rings (or skew-fields) have been treated in [FL97] and principal ideal domains in [BLS05]. Moreover, the algorithm is working in a very similar way to techniques based on conjugacy, which Béal, Lombardy and Sakarovitch [BLS06] worked out for semirings such as fields, natural numbers and integers. We also show that the notion of coalgebra homomorphism (i.e. matrices translating between any two given weighted automata in this context) is strongly related to conjugacy as described in [BLS06, BÉ93, ÉK01]. Here, we will make explicit how these two concepts are related.

In order to apply the algorithm to weighted automata, we have to solve systems of linear equations for a given semiring. We will specifically treat fuzzy automata with weights from an l -monoid as a case study.

One of the aims of this chapter is to present an algorithm to decide or, depending on the semiring, semi-decide language equivalence for weighted automata. The algorithm is defined in a general coalgebraic setting and can be used for various types of automata. For a full discussion of the coalgebraic aspects of the algorithm, as well as the corresponding proofs, refer to Chapter 3.

We will introduce an example weighted automaton in order to give some intuition. For our examples we primarily use the l -monoid $([0, 1], \max, \cdot, 0, 1)$, hence we are considering fuzzy automata, where the weights are taken from a distributive lattice, or more generally, an l -monoid [Rah09].

Example 4.1.1 *We consider the automaton α over the l -monoid $([0, 1], \max, \cdot, 0, 1)$ from Example 2.2.20, represented visually on the left and as a matrix α on the right.*



In this automaton, states 1 and 2 only accept words w from the language $L(a^*(bba^*)^*(b|\varepsilon))$ and assign the value¹ $0.1^{\#_a(w)}$ to them, whereas state 3 only accepts words of the form $w \in L((ba^*b)^*(ba^*|\varepsilon))$ and again assigns the value $0.1^{\#_a(w)}$ to them. Hence, states 1 and 2 are language equivalent and the intuitive reason for this is the fact that the a -labelled transition with weight 0.05 from state 2 to state 1 is somehow redundant, subsumed by the loops with weight 0.1. Note that state 3 is not language equivalent to states 1 and 2.

Throughout this chapter we consider only finite index sets for weighted automata.

This chapter is strongly based on [KK16].

4.2 The Prototype Algorithm

In this section we will describe the prototype algorithm and give all necessary definitions for the procedure. It works in a general coalgebraic setting, but since we are primarily concerned with weighted automata we will explain each concept in the setting of weighted automata right away.

The theorems in this section will only be stated and proven for weighted automata, but proofs for the more general coalgebraic variants of the theorems can be found in Chapter 3. Naturally, if the reader has read Chapter 3, this section is redundant.

Throughout this section, we consider a fixed, finite set A of actions.

¹ $\#_a(w)$ stands for the number of occurrences of the letter a in the word w .

4.2.1 The Operator F

In the algorithm to decide language equivalence for weighted automata, we enumerate the weights a word gets assigned by each state in an automaton in order of length. So first we compute the weight each state assigns to the word of length 0, then the weight each state assigns to the words of length 1 and so on. One way of doing this inductively is the following:

- The function $d_1: X \times \{\varepsilon\} \rightarrow \mathbb{S}$ that assigns to each state its weight for the empty word in the automaton $\alpha: X \times (A \times X + 1) \rightarrow \mathbb{S}$ is just the last column of α .
- Given a function $d_i: X \times A^i \rightarrow \mathbb{S}$ that assigns to each state x and each word $w \in A^i$ of length smaller than or equal to $i - 1$ the weight for w of x in α , we can compute the corresponding function $d_{i+1}: X \times A^{i+1} \rightarrow \mathbb{S}$ via the following observation: The weight a word aw of length at most $i + 1$ has in state x can be computed as the sum over all states y of the product of the weight $\alpha(x, (a, y))$ and the weight $d_i(y, w)$

Consequently, we want to capture this via matrix multiplication. However, a matrix d_i as above cannot be multiplied with a weighted automaton α . In order to build a matrix that has the correct type based on the information present in α , we define the operator F as follows:

Definition 4.2.1 (Operator F) We define the operator $F_{X,Y}: (X \times Y \rightarrow \mathbb{S}) \rightarrow ((A \times X \cup \{\bullet\}) \times (A \times Y \cup \{\bullet\}) \rightarrow \mathbb{S})$ as follows: Let $f: X \times Y \rightarrow \mathbb{S}$ be any matrix, $x \in X$, $a \in A$, $y \in Y$, $b \in A$, then

$$F_{X,Y}f(x, a)(y, b) = \begin{cases} f(x, y) & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}$$

Moreover,

$$F_{X,Y}f(x, a)(\bullet) = F_{X,Y}f(\bullet)(y, b) = 0$$

and

$$F_{X,Y}f(\bullet)(\bullet) = 1$$

In the sequel, we will write just Ff instead of $F_{X,Y}f$ to denote the (unique) operator that matches the type of f .

It is immediate that F preserves identity matrices:

Corollary 4.2.2 *Let $\text{id}_X: X \times X \rightarrow \mathbb{S}$ be the $X \times X$ unit matrix, then $F\text{id}_X$ is the $X \times A \cup \{\bullet\}$ unit matrix.*

Moreover, F is also compatible with multiplication:

Lemma 4.2.3 *Let $f: X \times Y \rightarrow \mathbb{S}$ and $g: Y \times Z \rightarrow \mathbb{S}$ be two matrices, then $F(f \cdot g) = Ff \cdot Fg$.*

Proof: Let $x \in X$, $a \in A$, $z \in Z$, $c \in C$, then

$$F(f \cdot g)(x, a)(z, c) = \begin{cases} (f \cdot g)(x, z) & \text{if } a = c \\ 0 & \text{otherwise} \end{cases}$$

and

$$\begin{aligned} & (Ff \cdot Fg)(x, a)(z, c) \\ &= \sum_{(y,b) \in Y \times A} \{Ff(x, a)(y, b) \cdot Fg(y, b)(z, c)\} + Ff(x, a)(\bullet) \cdot Fg(\bullet)(z, c) \\ &= \sum_{(y,b) \in Y \times A} \{Ff(x, a)(y, b) \cdot Fg(y, b)(z, c)\} \\ &= \begin{cases} \sum_{y \in Y} \{Ff(x, a)(y, a) \cdot Fg(y, a)(z, a)\} & \text{if } c = a \\ 0 & \text{otherwise} \end{cases} \\ &= \begin{cases} \sum_{y \in Y} \{f(x)(y) \cdot g(y)(z)\} & \text{if } c = a \\ 0 & \text{otherwise} \end{cases} = \begin{cases} (f \cdot g)(x, z) & \text{if } c = a \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

If one or both indices are \bullet , the computation is similar but simpler because more entries are guaranteed to be 0. \square

This operator F now allows us to express the desired computation via matrix multiplication where $d_{i+1} = \alpha \cdot Fd_i$. This is actually the main step in the algorithms to decide language equivalence.

Remark 4.2.4 *For ease of notation, we will write FX for the set $X \times A \cup \{\bullet\}$.*

4.2.2 Equivalences and Preorders on Matrices

We first start by defining preorders and equivalences on matrices:

Definition 4.2.5 (Relations on Matrices) Let $a: X \times A \rightarrow \mathbb{S}$, $b: X \times B \rightarrow \mathbb{S}$ be two matrices with the same row indexes. We write $a \leq^X b$ whenever there exists a matrix $d: A \times B \rightarrow \mathbb{S}$ with $a \cdot d = b$. Similarly, we write $a \equiv^X b$, provided $a \leq^X b$ and $b \leq^X a$.

These relations can be understood using the notion of generators for semimodules:

Proposition 4.2.6 Let a be an $X \times Y$ -matrix and b be an $X \times Z$ -matrix. It holds that $a \leq^X b \iff \langle a \rangle \subseteq \langle b \rangle$. That is, two matrices a, b are ordered if and only if the subsemimodule spanned by a is included in the subsemimodule spanned by b . Hence also $a \equiv^X b \iff \langle a \rangle = \langle b \rangle$.

This result is immediate by considering linear combinations of the columns.

In addition, the following simple properties hold for \leq (and similarly for \equiv).

Lemma 4.2.7 Let $a: X \times Y \rightarrow \mathbb{S}$ and $b: X \times B \rightarrow \mathbb{Z}$ be matrices. Then it holds that

1. $a \leq^X b$ implies $Fa \leq^{FX} Fb$
2. $a \leq^X b$ implies $c \cdot a \leq^W c \cdot b$ for any matrix $c: W \rightarrow X$.

Proof:

1. This holds because F is compatible with multiplication. $a \leq^X b$ implies there is a matrix $f: Y \times Z \rightarrow \mathbb{S}$ such that $a \cdot f = b$ and thus $Fa \cdot Ff = Fb$, i.e. $Fa \leq^{FX} Fb$.
2. If $a \leq^X b$ then there is a matrix $f: A \times B \rightarrow \mathbb{S}$ such that $a \cdot f = b$ and thus $c \cdot a \cdot f = c \cdot b$.

□

Intuitively, the second statement means that multiplying two matrices a, b , that span semimodules included in one another, with a matrix c will yield matrices whose corresponding semimodules are again ordered by inclusion.

Before introducing the algorithms, based on the construction of the final chain [AK95], we will first discuss how language equivalence can be expressed as a post-fixpoint. We will use the following alternative characterisation of language equivalence:

Lemma 4.2.8 *Let $\alpha: X \times (A \times X \cup \{\bullet\}) \rightarrow \mathbb{S}$ be a weighted automaton, then two states $x, x' \in X$ are language equivalent if and only if there exists a weighted automaton $\beta: Y \times (A \times Y \cup \{\bullet\}) \rightarrow \mathbb{S}$ and a matrix $f: X \times Y \rightarrow \mathbb{S}$ such that $\alpha \cdot Ff = f \cdot \beta$ and $f(x, y) = f(x', y)$ for all $y \in Y$. We call a matrix f such that $\alpha \cdot Ff = f \cdot \beta$ coalgebra homomorphism from α to β .*

A full proof for this correspondence can be found in Chapter 3, however, the proof idea can be understood as follows:

- If an f and a β exist such that $\alpha \cdot Ff = f \cdot \beta$ and $f(x)(y) = f(x')(y)$ for all $y \in Y$, but x and x' are not language equivalent, then there exists a word w that has different weights in x and x' . Now consider the sequence d_1, d_2, \dots from the motivation for the operator F . We will see that $d_i \geq^X f$ for all i , so in particular for $i = |w| + 1$, but $d_{|w|+1}(x, w) \neq d_{|w|+1}(x', w)$, yet $f(x, y) = f(x', y)$ for all y , so there cannot exist a matrix g such that $d_{|w|+1} = f \cdot g$ due to the way matrix multiplication is defined, which is a contradiction to $d_i \geq^X f$.
- Let x and x' be language equivalent states, then we can define a weighted automaton that has A^* as its state space and transition weight 1 from each word aw to the word w , $w \in A^*$, $a \in A$, 0 otherwise – note that β defined this way has finite support in its columns and actually is a weighted automaton. By mapping $f(\tilde{x}, w)$ to the weight the word w has in \tilde{x} for all states \tilde{x} and all words (i.e. β -states) w , it is immediate that $f(x)(w) = f(x')(w)$ for all $w \in A^*$. A simple computation shows that $\alpha \cdot Ff = f \cdot \beta$ holds.

The following observation about this alternative characterisation of language equivalence is the basis for the corresponding algorithm.

Proposition 4.2.9 *Let $\alpha: X \times (A \times X \cup \{\bullet\}) \rightarrow \mathbb{S}$ be a weighted automaton. Furthermore let $f: X \times Y \rightarrow \mathbb{S}$ be a matrix. It holds that $f \leq^X \alpha \cdot Ff$ (we say: f is a post-fixpoint) if and only if there exists a weighted automaton $\beta: Y \times (A \times Y \cup \{\bullet\}) \rightarrow \mathbb{S}$ such $\alpha \cdot Ff = f \cdot \beta$.*

For every such post-fixpoint f we have that $f(x)(y) = f(x')(y)$ for all $y \in Y$ implies that x and x' are language equivalent $x \sim x'$. If, in addition, it holds for every other post-fixpoint $g: X \times Z \rightarrow \mathbb{S}$ that $g \leq^X f$ (f is the

largest post-fixpoint), we can conclude that f induces language equivalence, i.e., $\forall y \in Y : f(x)(y) = f(x')(y) \iff x \sim x'$.

Proof: The first statement follows directly from the alternative characterisation of language equivalence, since $f \leq^X \alpha \cdot Ff$ means the existence of a matrix $\beta : Y \times (A \times Y \cup \{\bullet\}) \rightarrow \mathbb{S}$ with $f \cdot \beta = \alpha \cdot Ff$, which by the alternative characterisation of language equivalence means, $f(x, y) = f(x', y)$ for all $y \in Y$ implies $x \sim x'$. It is left to show that $x \sim y$ implies $f(x, y) = f(x', y)$ for all $y \in Y$ if f is the largest fixpoint. Since $x \sim y$, there must be some weighted automaton $\gamma : Z \times (A \times Z \cup \{\bullet\}) \rightarrow \mathbb{S}$ and a matrix $g : X \times Z \rightarrow \mathbb{S}$ such that $g \cdot \gamma = \alpha \cdot Fg$ and $g(x, z) = g(x', z)$ for all $z \in Z$. This implies that $g \leq^X \alpha \cdot Fg$ and hence $g \leq^X f$. Finally, we obtain $f(x, y) = f(x', y)$ for all $y \in Y$. \square

One can imagine the largest fixpoint $f : X \times Y \rightarrow \mathbb{S}$ as a matrix that maps every state into a vector that represents its (language) equivalence class.

4.2.3 Comparison to Conjugacy

We will now compare the alternative characterisation of language equivalence for weighted automata (i.e. the notion of coalgebra homomorphism) to a strongly related concept that has been used for deciding language equivalence for weighted automata, the notion of conjugacy [BLS05, BLS06, Sak09, BÉ93]. A general notion of conjugacy (called simulation) appears in [ÉK01].

The alternative characterisation of language equivalence for weighted automata via coalgebra homomorphisms is strongly related to the notion of conjugacy (see [BLS05, Sak09]). In fact, if there exists a matrix $f : X \times Y \rightarrow \mathbb{S}$ such that $\alpha \cdot Ff = f \cdot \beta$ and $f(x, y) = f(x', y)$ for all $y \in Y$ then the weighted automaton (X, α) is conjugate to (Y, β) and vice versa (disregarding initial weights).

We will first define what a conjugacy is, following [BLS05]: a weighted automaton (I, E, T) over a state set X consists of a square matrix E (the transition matrix) of dimension X , whose entries are linear combinations of letters in A (or, alternatively, vectors of the form \mathbb{S}^A). Furthermore I, T are vectors of dimension X , specifying initial and terminal weights.

A weighted automaton specified this way can be transformed into the corresponding weighted automaton in our notation $\alpha \in \mathbb{S}^{X \times (A \times X + 1)}$ by omitting

the initial weights and setting $\alpha(x, (a, y)) = E(x, y)(a)$, $\alpha(x, \bullet) = T(x)$ for $x, y \in X$, $a \in A$.

A weighted automaton (I_1, E_1, T_1) over state set X is *conjugate* to a weighted automaton (I_2, E_2, T_2) over state set Y , if there exists a matrix $Z \in \mathbb{S}^{X \times Y}$ such that $I_1 \cdot Z = I_2$, $E_1 \cdot Z = Z \cdot E_2$, $T_1 = Z \cdot T_2$.

We will argue that the last two equations are equivalent to the existence of a matrix $f: X \rightarrow Y$ between weighted automata $\alpha: X \rightarrow FX$, $\beta: Y \rightarrow FY$, such that $\alpha \cdot Ff = f \cdot \beta$, where $\alpha \in \mathbb{S}^{X \times (A \times X + 1)}$ is derived from (I_1, E_1, T_1) and $\beta \in \mathbb{S}^{Y \times (A \times Y + 1)}$ from (I_2, E_2, T_2) .

We set $Z = f$ and observe that in the equation $T_1 = Z \cdot T_2$ the vector T_1 corresponds to the last column (indexed by \bullet) of $\alpha \cdot Ff$, since T_1 is the last column of α , left unchanged by multiplication with Ff . On the other hand, $Z \cdot T_2$ equals the last column of $f \cdot \beta$, with T_2 being the last column of β and $Z = f$.

Furthermore observe that α minus its last column is a matrix of dimension $X \times (A \times X)$, which is isomorphic to a matrix of $(\mathbb{S}^{X \times X})^A$, which corresponds to the type of E_1 . Similarly, there is a correspondence between β and E_2 . Finally, if we omit the last row and last column (again indexed \bullet) from Ff we obtain a matrix of dimension $(A \times X) \times (A \times Y)$, which can be transformed into a matrix of $(\mathbb{S}^{A \times A})^{X \times Y}$, whose entries of type $\mathbb{S}^{A \times A}$ are diagonal matrices (all entries on the diagonal of a matrix in position (x, y) are equal and correspond to $f(x, y) = Z(x, y)$; all other entries are 0). Hence, multiplying α with Ff (minus last row and column) corresponds to a multiplication with f where vectors are multiplied with scalars instead of diagonal matrices. In this way we obtain from $\alpha \cdot Ff = f \cdot \beta$ the equation $E_1 \cdot Z = Z \cdot E_2$.

Similarly, a conjugacy matrix Z can be transformed into the corresponding matrix $f = Z$ such that $\alpha \cdot Ff = f \cdot \beta$.

We do not consider initial weights, hence there is no correspondence for the equation $I_1 \cdot Z = I_2$.

To illustrate this transformation, consider the fuzzy automaton α from Example 4.1.1.

$$\alpha = \begin{pmatrix} 0.1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0.05 & 0 & 0.1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

There exists a weighted automaton β and a coalgebra homomorphism f as follows:

$$\beta = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0.1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \quad f = \begin{pmatrix} 0 & 0.1 & 1 \\ 0 & 0.1 & 1 \\ 0.1 & 0 & 1 \end{pmatrix}$$

We quickly check that f actually is a coalgebra homomorphism:

$$\begin{aligned} \alpha \cdot Ff &= \begin{pmatrix} 0.1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0.05 & 0 & 0.1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0.1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0.1 & 0 & 1 & 0 \\ 0 & 0 & 0.1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0.1 & 0 & 1 & 0 \\ 0.1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0.1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 0.1 & 0.01 & 0 & 0.1 & 1 & 1 \\ 0 & 0.1 & 0.01 & 0 & 0.1 & 1 & 1 \\ 0 & 0 & 0 & 0.1 & 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0.1 & 1 \\ 0 & 0.1 & 1 \\ 0.1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0.1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \\ &= f \cdot \beta \end{aligned}$$

Now we can translate all matrices according to the above construction. From α , we obtain (E_α, T_α) (no I_α since there are no initial states):

$$E_\alpha = \begin{pmatrix} 0.1a & 0 & b \\ 0.05a & 0.1a & b \\ 0 & b & 0 \end{pmatrix} \quad T_\alpha = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

The translation of β yields:

$$E_\beta = \begin{pmatrix} 0 & b & b \\ b & 0.1a & a+b \\ 0 & 0 & b \end{pmatrix} \quad T_\beta = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$Z = f$, so we do not list it again. Now we can check that (E_α, T_α) and (E_β, T_β) are conjugate:

$$E_\alpha \cdot Z = \begin{pmatrix} 0.1a & 0 & b \\ 0.05a & 0.1a & b \\ 0 & b & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0.1 & 1 \\ 0 & 0.1 & 1 \\ 0.1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0.1b & 0.01a & 0.1a+b \\ 0.1b & 0.01a & 0.1a+b \\ 0 & 0.1b & b \end{pmatrix}$$

$$\begin{aligned}
Z \cdot E_\beta &= \begin{pmatrix} 0 & 0.1 & 1 \\ 0 & 0.1 & 1 \\ 0.1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & b & b \\ b & 0.1a & a+b \\ 0 & 0 & b \end{pmatrix} = \begin{pmatrix} 0.1b & 0.01a & 0.1a+b \\ 0.1b & 0.01a & 0.1a+b \\ 0 & 0.1b & b \end{pmatrix} \\
Z \cdot T_\beta &= \begin{pmatrix} 0 & 0.1 & 1 \\ 0 & 0.1 & 1 \\ 0.1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = T_\alpha
\end{aligned}$$

4.2.4 Algorithm A for Checking Language Equivalence

We will now, in two steps, work out an algorithm that can be used to find the largest post-fixpoint for a weighted automaton. This first version of the algorithm, named Algorithm A, is a non-optimised version of the algorithm and is based on the final chain construction [AK95], which we have already begun to illustrate in the motivation for the operator F . We will use this version of the algorithm to prove correctness of the algorithm, later we will introduce an optimisation named Algorithm B.

Both algorithms take as input a weighted automaton $\alpha: X \times (A \times X + \{\bullet\}) \rightarrow \mathbb{S}$ and return a weighted automaton $\beta: Y \times (A \times Y \cup \{\bullet\}) \rightarrow \mathbb{S}$ and a matrix $f: X \times Y \rightarrow \mathbb{S}$ such that $\alpha \cdot Ff = f \cdot \beta$. The matrix f , the largest fixpoint, is of special interest, since it induces exactly language equivalence on X . In order to specify the index sets of all matrices involved, we will write $FX = A \times X \cup \{\bullet\}$ and $F^i X = F(F^{i-1}(X))$ from here on out.

Algorithm 4.2.10 Final Chain Algorithm A

Let $\alpha: X \rightarrow (A \times X \cup \{\bullet\})$ be a weighted automaton. We define the following algorithm.

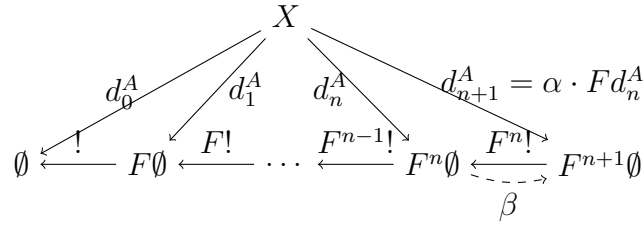
Step 0: Take the (unique) matrix $d_0^A: X \times \emptyset \rightarrow \mathbb{S}$.

Step $i + 1$: Compute $d_{i+1}^A = \alpha \cdot Fd_i^A: X \times F^{i+1}\emptyset \rightarrow \mathbb{S}$.

Termination: If there exists a matrix $\beta: F^n\emptyset \rightarrow F^{n+1}\emptyset$ such that $d_n^A \cdot \beta = d_{n+1}^A$, i.e., if $d_n^A \leq^X d_{n+1}^A$, the algorithm terminates and returns d_n^A and β as its result.

Algorithm A uses the coalgebraic concept of iterating along the final chain,

except for the termination condition. It can easily be seen to be a direct instantiation of Algorithm A from the previous chapter to weighted automata.



The algorithm yields a sequence of matrices $d_0^A \geq^X d_1^A \geq^X d_2^A \geq^X \dots$ that approximates language equivalence from above. If and as soon as this sequence becomes stationary, i.e., $d_n^A \equiv^X d_{n+1}^A$, the algorithm terminates. The above image visualises the algorithm as a commuting diagram. The edges of the graph signify matrices that occur in the algorithm, whereas the nodes are labelled with their respective dimensions. Two adjacent edges signify matrices that may be multiplied. The notable property of this diagram is commutativity, i.e. whenever there are two sequences of arrows starting in a common vertex and ending in a common vertex, their corresponding matrix products are the same. For instance, the edges labelled d_n^A , β and d_{n+1}^A form a triangle, where d_n^A starts in the same vertex as d_{n+1}^A and β ends in the same vertex as d_{n+1}^A . Thus, $d_n^A \cdot \beta = d_{n+1}^A$, due to commutativity. So this diagram shows how d_i^A can be retained from any d_j^A , where $j \geq i$ via matrix multiplication. The point at which this correspondence goes both ways (if it exists), is where the algorithm terminates and yields β and d_n^A as its output.

Remark 4.2.11 *When applied to a weighted automaton α , Algorithm A computes, for $i = 0, 1, \dots$, the values $L_\alpha(w)(x)$ for a state x and words w of length $|w| \leq i$. It then checks whether the semimodule spanned by $L_\alpha(w)$ for words up to length $i + 1$ equals the semimodule spanned for words up to length i . If the semimodules are the same, the algorithm terminates.*

We will now show that the algorithm indeed yields the greatest post-fixpoint whenever it terminates, i.e. it can serve as a semidecision procedure for language equivalence.

Lemma 4.2.12 *Let $g: X \times Z \rightarrow \mathbb{S}$ be any post-fixpoint, i.e. $g \leq^X \alpha \cdot Fg$, then for all d_i^A obtained in Algorithm A we have $d_i^A \geq^X g$.*

Proof: Clearly $d_0^A \geq^X g$, because d_0^A is a 0-dimensional matrix, and it is immediate, that any matrix can be transformed into a matrix with zero rows by multiplying it to a matrix that has zero rows. By induction, using Lemma 4.2.7, we can show that $d_i^A \geq^X g$ implies $d_{i+1}^A = \alpha \cdot Fd_i^A \geq^X \alpha \cdot Fg \geq^X g$. The first step is the induction hypothesis, the second uses the fact that g is a post-fixpoint. \square

Proposition 4.2.13 *If Algorithm A terminates in step n , its result d_n^A induces language equivalence, i.e. $x, x' \in X$ are language equivalent ($x \sim x'$) if and only if $d_n^A(x, y) = d_n^A(x', y)$ for all $y \in F^n\emptyset$.*

Proof: First, $d_0^A \geq^X d_1^A$ since d_0^A is a matrix with zero rows. Since \leq^X is preserved by functors and by left multiplication with matrices, we obtain by induction, using Lemma 4.2.7, that $d_i^A \geq^X d_{i+1}^A$. The termination condition says that $d_n^A \leq^X d_{n+1}^A = \alpha \cdot Fd_n^A$. This shows that the matrix d_n^A is actually a fixpoint. In order to conclude with Proposition 4.2.9 that the resulting matrix d_n^A induces behavioural equivalence, we have to show that it is the largest post-fixpoint. Assume there is another post-fixpoint $g: X \times Z \rightarrow \mathbb{S}$ with $g \leq^X \alpha \cdot Fg$. Then, Lemma 4.2.12 shows that $d_n^A \geq^X g$ and thus d_n^A is the largest post-fixpoint. \square

Furthermore, whenever $d_n^A \equiv^X d_{n+1}^A$ we have that $d_n^A \equiv^X d_m^A$ for every $m \geq n$. Therefore, every arrow d_m^A obtained at a later step induces behavioural equivalence as well. In that sense, the algorithm stabilises upon reaching a fixed point wrt. partitioning the states.

4.2.5 Algorithm B for Checking Language Equivalence

Algorithm A as we have defined it so far can be an effective way of deciding language equivalence, however we are exploring all words up to a given length and collect their values for each state in a matrix. The resulting matrix can quickly grow to unfeasible sizes, when it comes to performing further iteration steps. We are therefore interested in finding an optimisation of the algorithm that can shrink the intermediate results as much as possible. For this purpose, observe that after each step of the algorithm, it is permissible to replace d_i^A with any representative e_i^A of the equivalence class of d_i^A , i.e., any matrix that spans the same semimodule. This holds because $d_i^A \equiv^X e_i^A$ implies $d_{i+1}^A = \alpha \cdot Fd_i^A \equiv^X \alpha \cdot Fe_i^A$ and checking the termination condition $d_n^A \leq^X d_{n+1}^A$ can

instead be done for any representatives of d_n^A, d_{n+1}^A . Consequently, it is clear that this additional step does not interfere with termination speed. We will see that in general, this optimisation does not come at an added cost, because to check the termination condition of Algorithm A, all necessary computations to find a suitable representative must be computed anyway. This gives rise to the following algorithm:

Algorithm 4.2.14 Final Chain Algorithm B

Let $\alpha: X \times (A \times X \cup \{\bullet\}) \rightarrow \mathbb{S}$ be a weighted automaton. Moreover, let \mathcal{R} , the class of representatives, be a class of matrices such that for any matrix d we have a matrix $e \in \mathcal{R}$ that is equivalent to d , i.e. $d \equiv^X e$. We define the following algorithm:

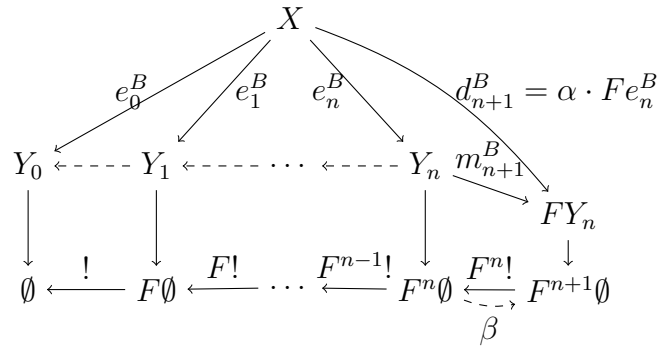
Step 0: Take the (unique) matrix $d_0^B: X \times \emptyset \rightarrow \mathbb{S}$.

Step $i + 1$: Find a representative $e_i^B \in \mathcal{R}$, for d_i^B , i.e. factor $d_i^B = e_i^B \cdot m_i^B$ such that $(e_i^B: X \times Y_i \rightarrow \mathbb{S}) \in \mathcal{R}$, $m_i^B: Y_i \times FY_{i-1} \rightarrow \mathbb{S}$. Determine $d_{i+1}^B = \alpha \cdot Fe_i^B: X \times FY_i \rightarrow \mathbb{S}$.

Termination: If there exists a matrix $\gamma: Y_n \times FY_n \rightarrow \mathbb{S}$ such that $e_n^B \cdot \gamma = d_{n+1}^B$, i.e., $d_{n+1}^B \geq^X e_n^B$, the algorithm terminates and returns e_n^B and γ as its result.

Choosing good and compact representatives can substantially mitigate state space explosion by reducing the number of columns of the matrices that form the intermediate results. Hence, Algorithm B is an optimisation of Algorithm A that (potentially) reduces the number of computations needed in every step. This optimisation has a cascading effect, since the number of columns of the matrix in step $i + 1$ depends on the number of columns of the matrix in the previous step i .

The algorithm can be graphically represented via the following diagram, where the matrices from the Y_i to the $F^i\emptyset$ can be constructed from m_0^B, \dots, m_i^B .



Algorithm B terminates in exactly as many steps as Algorithm A (if it terminates).

Proposition 4.2.15 *Algorithm B terminates in n steps if and only if Algorithm A terminates in n steps. Furthermore, two states $x, x' \in X$ are behaviourally equivalent ($x \sim x'$) if and only if $e_n^B(x, y) = Ue_n^B(x', y)$ for all $y \in Y_n$.*

Proof: First, we observe that we always have $d_i^B \equiv^X d_i^A$. This can be shown inductively. $d_0^A = d_0^B$ by definition. Now, assume $d_i^A \equiv^X d_i^B$, then $Fd_i^A \equiv^{FX} Fd_i^B$ and, since by definition $e_i^B \equiv^X d_i^B$, $Fe_i^B \equiv^{FX} Fd_i^A$. In both cases, we have used Lemma 4.2.7, Item (1). Item (2) then yields $d_{i+1}^B = \alpha \cdot Fe_i^B \equiv^X \alpha \cdot Fd_i^A = d_{i+1}^A$.

In addition, we factor in such a way that $e_i^B \equiv^X d_i^B$.

If Algorithm B terminates in n steps, then $d_{n+1}^B \geq^X e_n^B$. This implies $d_{n+1}^A \equiv^X d_{n+1}^B \geq^X e_n^B \equiv^X d_n^B \equiv^X d_n^A$ and Algorithm A terminates as well.

If, on the other hand, Algorithm A terminates in n steps, then $d_{n+1}^A \geq^X d_n^A$. We obtain $d_{n+1}^B \geq^X d_n^A \equiv^X d_n^B \equiv^X e_n^B$. Hence Algorithm B also terminates in n steps.

Since $d_n^A \equiv^X e_n^B$, they induce the same partition on X – the argument for this is analogous the first part of the proof idea for Lemma 4.2.8. It follows that one can check behavioural equivalence also with Algorithm B. \square

Termination for Algorithm B is independent of the choice of the representatives e_i^B .

However, Algorithm B need not terminate in general. It does terminate whenever \equiv^X has only finitely many equivalence classes. For some choices of \mathbb{S} , this holds for finite state sets X : if $\mathbb{S} = \mathbb{F}$ is a field, \equiv^X is of finite index, whenever X is finite, because finite dimensional vector spaces have a convenient

concept of basis that ensures that a vector space of dimension X has a basis of at most $|X|$ vectors. This means that the algorithm terminates for weighted automata over fields, whenever the state set X is finite.

However, it need not terminate for all weighted automata, since language equivalence of weighted automata is undecidable in general. It was shown in [Kro94, ABK11] that language equivalence is not decidable for weighted automata with weights over the *tropical semiring*. Termination hence depends on the semiring and possibly on the automaton we investigate. In the case of weighted automata \equiv^X is of finite index if and only if \mathbb{S}^X has only finitely many subsemimodules, but a weaker condition discussed in the next section is already sufficient for termination.

A class of representatives \mathcal{R} must have the property that for every subsemimodule of \mathbb{S}^X there exists a matrix $a \in \mathcal{R}$ generating this subsemimodule, i.e., the subsemimodule corresponds to $\langle a \rangle$.

Depending on the semiring, the class of representatives can be rather simple, for instance for fields, one could choose matrices corresponding to the bases of vector spaces of dimension X . In general, there is no notion of basis for semimodules over a semiring. However, a suitable choice for the class of representatives is the class of all generating matrices without redundant column vectors.

Definition 4.2.16 (Class of Representatives for Weighted Automata)

We define \mathcal{R} as the class of all matrices a that do not contain a column that is a linear combination of the other columns of a .

To state a termination condition for the case of *weighted automata* we define the following generating sets for a given weighted automaton (X, α) : $G^n = \{L_\alpha(w) \mid w \in A^*, |w| \leq n\}$ and $G^* = \{L_\alpha(w) \mid w \in A^*\} = \bigcup_{n=0}^\infty G^n$. It can be shown that $G^* \subseteq \mathbb{S}^X$ is finitely generated if and only if there exists an index n such that $\langle G^n \rangle = \langle G^* \rangle$. (Note that $\langle G^* \rangle$ consists of all linear combinations of vectors of G^* . Thus, if it has a finite generator set G , the vectors of G can again be composed of finitely many vectors of G^* . And these must be contained in some G^n .)

We will show that Algorithm B terminates whenever $\langle G^* \rangle$ is finitely generated (see also [DK13]). We start with the following lemma. (Note that the notation $(d_i)_w$ is based on Definition 2.1.7.)

Lemma 4.2.17 *Let $d_i: X \times F^i\emptyset$, $i \in \mathbb{N}_0$ be the sequence of matrices generated by Algorithm A. Hence $F^i\emptyset$ contains tuples of the form $(a_1, a_2, \dots, a_{i-1}, \bullet)$, where $a_j \in A$. We will identify such a tuple with the word $w = a_1a_2 \dots a_{i-1}$. Then, for an index $i \in \mathbb{N}_0$ and a word w with $|w| < i$, we have $(d_i)_w = L_\alpha(w)$. This means that $\langle d_{i+1} \rangle = \langle G^i \rangle$.*

Proof: We will prove this via induction on i .

Base case: For $i = 0$ there is nothing to show.

Inductive step: We assume that $(d_i)_w = L_\alpha(w)$ for all words w with $|w| < i$. Then we have:

$$\begin{aligned} (d_{i+1})_w(x) &= (\alpha \cdot Fd_i)_w(x) = (\alpha \cdot Fd_i)(x, w) \\ &= \sum_{(a, x') \in FX} \alpha(x, (a, x')) \cdot Fd_i((a, x'), w) + \alpha(x, \bullet) \cdot Fd_i(\bullet, w) \end{aligned}$$

We now distinguish the following cases: if $w = \varepsilon$, then by the naming convention $Fd_i((a, x'), \varepsilon) = Fd_i((a, x'), \bullet) = 0$, $Fd_i(\bullet, \varepsilon) = Fd_i(\bullet, \bullet) = 1$. And hence the term above reduces to $\alpha(x, \bullet) = L_\alpha(\varepsilon)(x)$.

If $w = bu$, $b \in A$, $u \in A^*$, we have $Fd_i((a, x'), bu) = Fd_i((a, x'), (b, u)) = d_i(x', u)$ if $a = b$ and 0 otherwise, and $Fd_i(\bullet, bu) = Fd_i(\bullet, (b, u)) = 0$. Hence we obtain:

$$\begin{aligned} &\sum_{(b, x') \in FX} \alpha(x, (b, x')) \cdot d_i(x', u) = \sum_{x' \in X} \alpha(x, (b, x')) \cdot d_i(x', u) \\ &= \sum_{x' \in X} \alpha(x, (b, x')) \cdot L_\alpha(u)(x') = L_\alpha(bu)(x) = L_\alpha(w)(x) \end{aligned}$$

□

Proposition 4.2.18 *If $\langle G^* \rangle$ for a weighted automaton (X, α) is finitely generated, Algorithm B terminates.*

Proof: From Lemma 4.2.17 it follows that $\langle d_{i+1}^A \rangle = \langle G^i \rangle$. Since $\langle G^* \rangle$ is finitely generated, there must be an index n with $\langle G^n \rangle = \langle G^{n+1} \rangle$, hence $\langle d_{n+1}^A \rangle = \langle d_{n+2}^A \rangle$. Proposition 4.2.6 then implies that $d_{n+1}^A \equiv^X d_{n+2}^A$ and hence $d_{n+1}^A \leq^X d_{n+2}^A$, which is exactly the termination condition. Note that this is not impaired when we consider representatives e_i in Algorithm B, since $e_{n+1}^B \leq^X e_{n+2}^B$ holds as well. In fact, a representative e_i^B generates the same subsemimodule as d_i^A . □

If \mathbb{S} is a *field*, $\langle G^* \rangle$ is always finitely generated (since each subsemimodule is). Note that for fields, minimal generating sets of semimodules, i.e. bases for

vector spaces, always have the same size, so each intermediate result will be of the smallest size possible. In fact, we can prove that the algorithm terminates after at most $|X|$ steps.

Lemma 4.2.19 *If \mathbb{S} is a field, then Algorithm B terminates within n iterations provided the automaton under investigation has n states.*

Proof: If the automaton under investigation has n states, then we are constructing a sub vector space of \mathbb{S}^n . In a field, a linearly independent set of vectors of dimension n can have cardinality of at most n . In each iteration of the algorithm, the number of vectors in e_i^B either increases over e_{i-1}^B , or it stays the same and the algorithm terminates. Therefore, in each non-terminating iteration the number of columns of e_i^B increases by at least 1 over its predecessor. Since the maximum number of columns an e_i^B can have is n , the algorithm terminates after at most n steps. \square

Corollary 4.1 *Given two states s and t in a weighted automaton (X, α) over a field where $|X| = n$, if there exists a word w such that the weight of w under s is different from the weight of w under t , then there exists a word w' where $|w'| \leq n$ such that the weight of w' differs between s and t , as well.*

Proof: This directly follows from Lemma 4.2.19. In iteration i , Algorithm B computes a generator for the vector space $\langle G^i \rangle$, so after n steps – where the algorithm terminates at the latest – it has computed a generator for the vector space $\langle G^n \rangle$. Since the algorithm is correct, it terminates after n steps at the latest and finds two states s, t not to be language equivalent if and only if there exists a word w that separates s and t within $\langle G^m \rangle \subseteq \langle G^n \rangle$, where m is the iteration where the algorithm terminates. \square

For fields the algorithm coincides with earlier work by Schützenberger [Sch61] who investigated a similar procedure, taking into account initial weights and exploring the words from left to right instead of from right to left. A detailed presentation of the algorithm for fields can be found in Chapter 3 of [BR88]. Similar algorithms were presented in [Bor09, ABH⁺12]. Schützenberger’s work was extended by Flouret and Laugerotte, [DK13, FL97], to skew-fields, structures that are almost fields, but where multiplication need not be commutative. A further extension can be found in [BLS06, BLS05], extending the decidability

result to principal ideal domains such as $(\mathbb{Z}, +, \cdot)$. As we have shown earlier, the algorithm is also strongly related to conjugacy, investigated for instance in [Sak09].

We can easily specify some classes where the algorithm necessarily terminates. If \mathbb{S} is a *finite semiring*, the algorithm terminates since there are only finitely many different column vectors of a fixed dimension $|X|$. If \mathbb{S} is a *distributive complete lattice*, the algorithm will terminate as well.

Corollary 4.2.20 *Algorithm B always terminates on weighted automata with weights from a distributive complete lattice.*

Proof: [Sketch] This is a corollary of Proposition 4.2.18. First, observe that there are only finitely many weights contained in a finite automaton. Since we are working in a distributive lattice \mathbb{S} , we can write the weights $L_\alpha(w)$ for every word w in disjunctive normal form, using distributivity. Due to idempotence, only finitely many elements of \mathbb{S} can be obtained via meet and join and thus there are only finitely many different column vectors of a fixed dimension. □

Note that this argument cannot be used for arbitrary l -monoids, since the multiplication of l -monoids need not be idempotent, a property which is used in the proof above. In fact, the algorithm need not terminate for all l -monoids, an example for this is presented later in Example 4.3.7.

For distributive lattices, it also does not follow that each spanned semimodule is finite, only that there are only finitely many different semimodules that can occur. For instance, in an infinite distributive lattice the vector (\top, \dots, \top) generates a vector $(\top, \top, \dots, \top) \sqcap l = (l, l, \dots, l)$ for each lattice element l and the generated subsemimodule is hence infinite in size.

Note that termination does not necessarily imply decidability of language equivalence. For this it is also necessary to be able to decide whether two matrices are equivalent, i.e., whether a vector is generated by a given set of generators. This need not be decidable in general, but it is decidable in all the cases above.

4.3 Algorithmic Issues and Case Studies

4.3.1 A Concrete Instantiation of Algorithm B

For easier understanding, we will now state Algorithm B for weighted automata in a more programming language-like notation. Note that we will assume that matrices have attributes `rows` and `columns` that contain the number of rows, respectively columns, of a matrix. Moreover, `concatenate` takes two matrices with the same number of rows and adds the columns of the second matrix as new columns at the end of the first matrix. For a matrix m , $m[i \dots j]$ denotes the submatrix of m consisting of the columns $i, i + 1, \dots, j$.

Algorithm 4.3.1 Language Equivalence Check (Algorithm B)

- **Input:** A weighted automaton α with state set X and labels A , with weights from a semiring $\mathbb{S} = (S, +, \cdot, 0, 1)$. That is, α is a matrix of dimension $X \times (A \times X + 1)$ with entries from \mathbb{S} .
- **Output:** (β, e) , where β is a weighted automaton with state set Y and $e: X \rightarrow Y$ is a coalgebra homomorphism.

Algorithm B (α)

```

1  $e_0 := \text{the unique } (X \times \emptyset)\text{-matrix}$ 
2  $i := 0$ 
3 do
4    $i := i + 1$ 
5    $d_i := \alpha \cdot F(e_{i-1})$ 
6    $(m_i, e_i) := \text{factorise}(d_i)$ 
7 while  $e_i \neq e_{i-1}$ 
8 return  $(m_i, e_i)$ 

```

From the output e we can infer language equivalence by checking whether $\hat{x}^t \cdot e = \hat{y}^t \cdot e$, where \hat{x}, \hat{y} are the unit vectors of dimension X that correspond to x, y .

The algorithm above uses a (non-deterministic) function `factorise`, which still needs to be specified. The function `factorise` is expected to find for a

given matrix d two matrices e and m such that $e \cdot m = d$ and e is a representative matrix, i.e. no row of e can be obtained via linear combination from the other rows of e . We will now discuss one possible implementation of **factorise**, where the general idea is to check for the columns one after another whether they can be eliminated since they can be reconstructed from the remaining columns.

For an arbitrary semiring, this function can be implemented as follows.

Algorithm 4.3.2 Factorisation

- **Input:** An \mathbb{S} -valued $n \times k$ -matrix d .
- **Output:** A pair (m, e) of \mathbb{S} -valued matrices, where $e \cdot m = d$ and $e \in \mathcal{R}$.

factorise (d)

```

1  $e := d$ 
2  $m := (n \times n)$  unit matrix
3  $i := 1$ 
4 while  $i \leq m.\text{columns}$  do
5    $e' := \text{concatenate}(e[i+1..n], e[1..i-1])$ 
6    $x := \text{findLinearCombination}(e', e[i])$ 
7   if  $x \neq \text{undefined}$  then
8      $e := e'$ 
9      $m' := (m.\text{columns}-1) \times (m.\text{columns}-1)$  unit matrix
10    insert  $x$  into  $m'$  after the  $(i-1)$ -th column
11     $m := m' \cdot m$ 
12   else
13      $i := i + 1$ 
14 return  $(m, e)$ 

```

In the factorisation algorithm above we check in line 6 and line 7 whether a column $e[i]$ can be obtained as a linear combination of other columns in e . If this is the case we construct a new matrix e' by removing this redundant column from e (line 8) and define the matrix m' such that $e' \cdot m'$ equals the original matrix e (line 9 and line 10). This matrix m' is then used to update m (line 11).

Intuitively, the entire algorithm computes, for each state and words of increasing length, the corresponding weight. A column (corresponding to a word) is removed from the intermediate result if it can be represented as a linear combination of other columns. As a consequence, in the next steps, no extensions of this word are explored. It is important to note, that the columns are checked one after another, so if e.g. two columns exist that are multiples of each other but cannot be obtained via linear combinations of the other columns, one of the columns will remain. The algorithm continues until the subsemimodule spanned by the columns remains stable.

Note that, due to this specific way of implementing **factorise**, two matrices e_i, e_{i+1} in Algorithm B span the same subsemimodule if and only if they are equal (cf. line 7 in the language equivalence check). The reason for this is that new columns get added over e_i when computing d_{i+1} to the left, whereas e_i is retained in the right-most columns.

The algorithm calls a function **findLinearCombination(A,b)** that returns a vector x such that $A \cdot x = b$, provided such a vector exists. Otherwise, i.e. if there is no linear combination of the columns of A that yields the vector b , the return value of the function call **findLinearCombination(A,b)** is undefined.

Moreover, Algorithm B applies the operator F to a matrix e . In order to make this explicit, we define the following method, where matrices have indices numbered from 1 to a fixed bound. More specifically, if e is an $(n \times m)$ -matrix and $k = |A|$ is the size of the alphabet $A = \{a_1, \dots, a_k\}$, then Fe is a $(k \cdot n + 1) \times (k \cdot m + 1)$ -matrix. The column index of $(a_{i,j})$ is $k \cdot (i - 1) + j$ and the index of \bullet is $k \cdot m + 1$, analogously, the indices can be computed for the rows.

Algorithm 4.3.3 Operator application $F(e)$

- **Input:** An \mathbb{S} -valued $(n \times m)$ -matrix e .
- **Output:** An \mathbb{S} -valued $(k \cdot n + 1) \times (k \cdot m + 1)$ -matrix e' with $e' = Fe$.

$\underline{F(e)}$

```

1  $e' := (k \cdot n + 1) \times (k \cdot m + 1)$ -zero matrix
2 for  $a_1 := 1$  to  $a$  do
3   for  $x := 1$  to  $n$  do
4     for  $a_2 := 1$  to  $a$  do
5       for  $y := 1$  to  $m$  do
6         if  $a_1 = a_2$  then
7            $e'[k \cdot (x - 1) + a_1, k \cdot (y - 1) + a_2] := e[x, y]$ 
8  $e'[a \cdot n + 1, a \cdot m + 1] := 1$ 
9 return  $e'$ 

```

The algorithm still is not fully specified, since we did not state how the procedure `findLinearCombination` works. The reason for this is, that this problem cannot be solved in general for all semirings. Instead, individual solutions for specific semirings or classes of semirings must be sought. Of course, for fields and division rings, i.e. structures that are almost fields but where multiplication need not be commutative, this problem can be solved via Gaussian elimination.

Hence, we will now inspect another class of weighted automata, so-called fuzzy automata, where the weights are taken from an l -monoid.

4.3.2 Case Study: l -Monoids and Fuzzy Automata

For l -monoids, we can use a procedure for finding linear combinations that uses an algorithm based on residuation [BJ72]. It has been used to solve linear equations of the form $A \cdot x = b$ as early as 1979 [CG79].

We require that arbitrary joins in the l -monoid $(L, \sqcup, \cdot, 0, 1)$ exist and that the multiplication \cdot distributes over arbitrary joins.

In this case we can define a residuation operation for $a, b \in L$:

$$(a \rightarrow b) = \bigsqcup \{x \in L \mid a \cdot x \leq b\}$$

Note that since multiplication distributes over joins we have $a \cdot (a \rightarrow b) \leq b$.

Example 4.3.4 *Residuation is defined for the l -monoid $([0, 1], \max, \cdot, 0, 1)$ from Example 2.2.20 as follows:*

$$(a \rightarrow b) = \max\{x \in [0, 1] \mid a \cdot x \leq b\} = \begin{cases} \frac{b}{a} & \text{if } a > b \\ 1 & \text{otherwise} \end{cases}$$

Now, given $A \in L^{m \times n}$, $b \in L^m$. We define a candidate solution vector $\tilde{x} \in L^m$ as follows, where $1 \leq j \leq m$:

$$\tilde{x}_j = \bigcap_{i=1}^m (a_{ij} \rightarrow b_i).$$

Note that, since arbitrary joins exist, we can also take meets (since the meet of a set S is the join of all elements smaller than all the elements in S).

Now it follows from results in [CG79] that $A \cdot x = b$ has a solution if and only if $A \cdot \tilde{x} = b$. Furthermore, in this case \tilde{x} is the greatest solution.

Using the l -monoid $([0, 1], \max, \cdot, 0, 1)$ from Example 2.2.20, we illustrate the algorithm for finding linear combinations. We will give an example where no solution exists.

Example 4.3.5 *Given $A = \begin{pmatrix} 0.1^{n-1} & 1 \\ 1 & 1 \end{pmatrix}$, $b = \begin{pmatrix} 0.1^n \\ 1 \end{pmatrix}$, we can show that the equation $A \cdot x = b$ has no solution, for all $n \in \mathbb{N}$. Again, we compute \tilde{x}_1 and \tilde{x}_2 :*

$$\tilde{x}_1 = \min\{0.1^{n-1} \rightarrow 0.1^n, 1 \rightarrow 1\} = \min\left\{\frac{0.1^n}{0.1^{n-1}}, \frac{1}{1}\right\} = 0.1$$

$$\tilde{x}_2 = \min\{1 \rightarrow 0.1^n, 1 \rightarrow 1\} = \min\left\{\frac{0.1^n}{1}, \frac{1}{1}\right\} = 0.1^n$$

Now, we have found a candidate $\tilde{x} := \begin{pmatrix} 0.1 \\ 0.1^n \end{pmatrix}$ and need to check whether $A \cdot \tilde{x} = b$. However, this is not the case:

$$\begin{pmatrix} 0.1^{n-1} & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0.1 \\ 0.1^n \end{pmatrix} = \begin{pmatrix} \max\{0.1^n, 0.1^n\} \\ \max\{0.1, 0.1^n\} \end{pmatrix} = \begin{pmatrix} 0.1^n \\ 0.1 \end{pmatrix} \neq \begin{pmatrix} 0.1^n \\ 1 \end{pmatrix}$$

So, we have proven that $A \cdot \tilde{x} \neq b$ and therefore that there is no x such that $A \cdot x = b$.

d_i	m_i	e_i
$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$	(1)	$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$
$\begin{pmatrix} 0.1 & 1 & 1 \\ 0.1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0.1 & 1 \\ 0.1 & 1 \\ 0 & 1 \end{pmatrix}$
$\begin{pmatrix} 0.01 & 0 & 0.1 & 1 & 1 \\ 0.01 & 0 & 0.1 & 1 & 1 \\ 0 & 0.1 & 0 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0.1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0.1 & 1 \\ 0 & 0.1 & 1 \\ 0.1 & 0 & 1 \end{pmatrix}$
$\begin{pmatrix} 0 & 0.1 & 0.01 & 0 & 0.1 & 1 & 1 \\ 0 & 0.1 & 0.01 & 0 & 0.1 & 1 & 1 \\ 0 & 0 & 0 & 0.1 & 0 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0.1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0.1 & 1 \\ 0 & 0.1 & 1 \\ 0.1 & 0 & 1 \end{pmatrix}$

Table 4.1: Algorithm B applied to the automaton from Example 4.1.1, the i -th line ($i = 1 \dots 4$) of the table represents the intermediate results from the i -th iteration

Note that from the considerations above we can easily derive a procedure `findLinearCombination` as required by the algorithm. Using this procedure Algorithm B can be applied to weighted automata over l -monoids, as long as the required operations, i.e. \cdot , \sqcap , \sqcup and \rightarrow , are computable. However, the algorithm may not terminate for some l -monoids. For every distributive bounded lattice and every locally finite² l -monoid it will terminate though, since only finitely many different column vectors can appear in the e_i .

Example 4.3.6 *We now get back to the fuzzy automaton from Example 4.1.1 and apply Algorithm B to it. We start with $d_0 = e_0$, a 0×3 -matrix. Now, Tabelle 4.1 contains all further intermediate results computed by the algorithm. Note that $d_1 = e_1 = \alpha \cdot Fe_0$ is simply the last column of α .*

²A semiring is locally finite if each finitely generated subsemiring is finite.

$$\text{Next we obtain } Fe_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

$$d_2 = \alpha \cdot Fe_1 = \begin{pmatrix} 0.1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0.05 & 0 & 0.1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0.1 & 1 & 1 \\ 0.1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

The next step is to look for a representative e_2 for d_2 . For this, we apply the method **factorise** which amounts to checking for each column, starting from the first, whether it is a linear combination of the other columns. Obviously the second column, which is equal to the third, is a linear combination of the first and the third column. The largest solution to the corresponding equation multiplies each of the two columns with 1 and takes the maximum. No further columns can be eliminated and factorisation yields e_2 and m_2 in Tabelle 4.1.

In d_3 the first and fourth column can be eliminated, whereas in d_4 this is true for the first, second, third and sixth column. Hence, we obtain $e_3 = e_4$ and the algorithm terminates.

The row with the matrices m_i shows the redundant information that has been factored out. Again m_4 can be considered as a representative of α that can be used to decide language equivalence.

Looking at the result of the overall algorithm, e_4 , we can conclude: States 1 and 2 are equivalent, since their respective rows are equal, whereas state 3 is not equivalent to the others, since its row differs from the first two.

Now that we have seen how the algorithm works, we present an example where it does not terminate, again using the l -monoid from Example 2.2.20.

4.4 Conclusion

In order to conclude, we will first introduce the implementation and then summarise related work, both from the areas of coalgebra and weighted automata. Finally, we mention some ideas for future work.

Implementation: We have developed a generic implementation of the algorithms presented in this chapter using C#. Currently, Algorithm B is implemented and can be used for automata defined over arbitrary semirings, if the operations of the semiring and the method `findLinearCombination` are provided. For this, we have implemented a semiring generator that supports the definition of arbitrary semirings, including a method for finding linear combinations [Mik15, KKM17]. We have instantiated the algorithm to fields (using Gaussian elimination for equation solving), to distributive lattices, l -monoids and other algebraic structures. All examples in this section were verified with the implementation. This implementation formed the basis for the tool PAWS which contains implementations of most algorithms presented in this thesis and has been enhanced by my colleague Christina Mika in her Master’s thesis with a visual interface and a semiring generator to allow for convenient and flexible usability of the implementation for any semiring, cf. Section 8.1 for more details on PAWS.

Related Work: As already mentioned in the introduction, there are many closely related concepts and methods in the field of weighted automata and rational formal power series. Consequently, we do not claim much originality for the instantiation of our generic algorithm to weighted automata.

For weighted automata there exist minimisation algorithms for a special class that is known in the literature as deterministic weighted automata that, however, do not work in the more general case. These algorithms are based on the idea of redistributing the weights of transitions in a canonical way and applying the minimisation algorithm for deterministic automata (without weights) to the resulting automaton, where label and weight of each transition form a new transition label in the deterministic weight-free automaton. Mohri’s algorithm [Moh97, Moh09] is based on weight pushing, and is applicable whenever \mathbb{S} is zero-sum-free and weakly divisible. Eisner’s algorithm [Eis03] works whenever

S has multiplicative inverses. He remarks that his variation of weight pushing can also be applied to some semirings that do not have inverses, if they can be extended to ones that do have inverses. However, then, the minimal automaton might carry weights outside of S .

Enumerating words of increasing length and stopping when longer words do not generate a larger semimodule is an idea used by several authors [BLS06, BLS05, BR88], for semirings such as fields and principal ideal domains. The fact that this procedure is basically an instance of the proposed Algorithm B is not too surprising, considering – as we have shown before – that conjugacy and coalgebra homomorphisms for weighted automata are strongly related concepts. Since the reduction step in [BLS06] is a special instance of choosing a set of generators for a given semimodule, their proofs also show that Algorithm B always terminates for \mathbb{Z} . Skew-fields on the other hand have been considered in [FL97]

Droste and Kuske [DK13] describe when equivalence of two weighted automata is decidable, based on earlier results by Schützenberger [Sch61]. Again, their decidability result is close to our own, only they work with weighted automata with initial weights. They show that whenever S is a ring such that every subsemimodule generated by an automaton is finitely generated, language equivalence is decidable.

Ésik and Maletti have investigated proper semirings in [ÉM10] and have proven that language equivalence for weighted automata is decidable whenever the corresponding semiring is proper and effectively presentable. Furthermore, they investigated Noetherian semirings, i.e. semirings where every subsemimodule of a finitely generated S -semimodule is finitely generated. However, they do not give a concrete algorithm.

Additionally, there are several contributions on axiomatic treatment of language equivalence aimed at characterising weighted languages as the free algebras in certain classes of semirings. This line of work goes back to Morisaki and Sakai, establishing a characterisation of this kind for weighted automata with weights taken from a field [MS80]. Krob has extended this work to rings in [Kro91]. A thorough investigation of this line of work can be found in [BÉ93]. Ésik has further found characterisations as free algebras for weighted languages over $\mathbb{N} \cup \{\infty\}$ (together with Bloom, [BÉ09]), certain commutative ordered

semirings with a $*$ -operation (together with Kuich, [ÉK13]) and the languages of weighted tree automata [Ési11], extending the notion of conjugacy to weighted tree automata, as well.

Kiefer et al. [KMO⁺11] have investigated optimisations for the case of weighted automata with weights over the field \mathbb{R} , with applications to probabilistic automata. Their algorithm is a probabilistic optimisation of an algorithm that enumerates words of length at most n , where n is the number of states, and their weights. As far as we know this method can not easily be generalised to arbitrary semirings.

Language equivalence is not decidable for every semiring. For instance, it was shown that language equivalence is undecidable for the tropical semiring [Kro94, ABK11].

Therefore, procedures very similar to the instantiation of our Algorithm B to weighted automata have been described in several papers, usually restricted to certain types of semirings. This instantiation of Algorithm B to weighted automata is offering a general account on how to decide language equivalence for a wide array of semirings that makes use of linear combinations to shrink the intermediate results. In the next chapter we will discuss further optimisations for specific semirings that build upon this optimisation via linear combinations, but adds further optimisations in form of up-to techniques using congruence closures and simulations.

Future Work: For future work we plan to further investigate the issue of termination: are there more general criteria which guarantee termination for our algorithms? For this we will need a suitable notion of “finiteness” in a general categorical setting, for instance the notion of *finitely generated* or *locally presentable* [AR94].

Furthermore, when working with equivalence classes of arrows, it is necessary to find good representatives, in order to discard redundant information and make the representation more compact.

Recently, weighted automata over a more general structure than semirings, valuation monoids, have been studied. Valuation monoids consist of a commutative monoid, which intuitively acts as the addition in a semiring and a valuation function that maps sequences of monoid elements to single elements, acting as

the obvious projection on single-element sequences and maps sequences that contain the neutral element of the monoid operation to the neutral element. The valuation function takes the role of the multiplication in a semiring, yet even fewer conditions are required. While the neutral element of the addition is cancellative for the valuation function as it is true for semirings, the valuation function need not be associative and need not have a neutral element. Additionally, distributivity need not hold between the valuation function and the monoid operation. Valuation monoids have been studied for weighted automata for instance by Droste and Meinecke [DM12], where a weighted MSO logic was developed for weighted automata over valuation monoids (with some additional structure). It is an interesting challenge to analyse these more general weighted automata using a coalgebraically motivated framework and to find out whether the techniques developed in this chapter can be adapted. The same restrictions used by Droste and Meinecke, introducing a neutral element wrt. multiplication may be required for a coalgebraic treatment though, to ensure the existence of identity matrices.

More specifically, weighted automata have been extended to arbitrary, i.e. not necessarily distributive, bounded lattices, which can serve as interesting special cases of valuation monoids, cf. e.g. [DV12]. Note, that bounded lattices can be identified as valuation monoids by letting the supremum operation together with the bottom element form the monoid and taking the infimum operation as the valuation function. A major challenge wrt. an adaptation of the techniques developed in this chapter to bounded, non-distributive lattices is the lack of a useful notion of linear combination. When multiplication distributes over addition, matrix multiplication is associative, but this fails to hold when the structure under consideration is not distributive. A different notion of equivalence would therefore be required.

Chapter 5

Up-To Techniques for Weighted Systems

5.1 Introduction

In the previous Chapter 4, we have discussed how we can determine all language equivalent pairs of states in a weighted automaton via means of a generalised partition refinement algorithm. This approach corresponds to the well-known minimisation procedure for deterministic automata. However, this technique need not terminate for all semirings – which is a consequence of undecidability of language equivalence in the general case – and even if it terminates, for many semirings the algorithm does not perform very well. This is, why we will now have a look at an alternative approach, that is again motivated by techniques for weighted automata over the binary Boolean algebra, i.e. non-deterministic automata.

Language equivalence of deterministic automata can be checked by means of the bisimulation proof principle. For non-deterministic automata, this principle is sound but not complete: to use bisimulation, one first has to determinise the automaton, via the so-called powerset construction. Since the determinised automaton might be much larger than the original non-deterministic one, several algorithms [WDHR06, DR10, ACHV10, BP13] have been proposed to perform the determinisation on the fly and to avoid exploring a huge portion of states. Among these, the algorithm in [BP13] that exploits *up-to techniques* is particularly relevant for this chapter. Though in Chapter 4 we have already

explored an optimisation that may be regarded as up-to linear combination, here, we aim at up-to congruence, which subsumes up-to linear combination. Additionally, instead of a partition refinement algorithm, we will now focus on an on-the-fly-technique, that builds a language equivalence relation while exploring words of increasing length. It is worth noting that we do not aim to find all pairs of equivalent states (as in Chapter 4), but to determine if a given pair of states is language equivalent, instead.

Up-to techniques have been introduced by Robin Milner in his seminal work on CCS [Mil89b] and, since then, they have proven useful, if not essential, in numerous proofs about concurrent systems (see [PS11] for a list of references). According to the standard definition, a relation R is a bisimulation whenever two states x, y in R can simulate each other, resulting in a pair x', y' that is still in R . An up-to technique allows to replace the latter R by a larger relation $f(R)$ which contains more pairs and hence allows to cut off bisimulation proofs and work with much smaller relations.

Here, we focus on up-to techniques in a quantitative setting: weighted systems, especially weighted automata over arbitrary semirings. Some examples of up-to techniques for weighted systems already appeared in [BPPR14b] and [RBB⁺15], that study up-to techniques from the abstract perspective of coalgebras.

Although up-to techniques for weighted systems have already received some attention, their relevance for algorithms to perform behavioural analysis has never been studied properly. This is the main aim of this chapter: We give a uniform class of algorithms exploiting up-to techniques to solve the problems of equivalence, inclusion and universality, which, in the weighted setting, asks whether the weight of all words is below some given threshold. In particular we show how to implement these techniques and we perform runtime experiments.

The key ingredient to algorithmically exploit up-to techniques is a procedure to decide, given x, y, R as above, whether x, y belongs to $f(R)$. For a non-deterministic automaton (NFA) with state space S , the algorithm in [BP13] uses as sub-routine a rewriting system to check whether two sets of states $S, S' \in \mathcal{P}(X)$ – representing states of the determinised automaton – belong to $c(R)$, the congruence closure of R .

For NFA, the congruence closure is taken with respect to the structure of join semi-lattices $(\mathcal{P}(X), \cup, \emptyset)$, carried by the state space of a determinised

automaton. For weighted automata, rather than join semi-lattices, we need to consider the congruence closure for *semimodules*. Indeed, an analogue of the powerset construction for weighted automata results in a sort of “determinised automaton” (called in [BBB⁺12] linear weighted automaton) whose states are vectors with values in the underlying semiring. We have discussed this approach for general semirings in the previous Chapter 4.

Our first issue is to find a procedure to check whether two vectors belong to the congruence closure (with respect to semimodules) of a given relation. We face this problem for different semirings, especially rings and l -monoids. For l -monoids we adapt the rewriting procedure for the non-deterministic case [BP13] and show its confluence and termination, which guarantees a unique normal form as a representative for each equivalence class. Confluence holds in general and termination can be shown for certain semirings, such as the tropical semiring (also known as the $(\min, +)$ -semiring).

Reasoning up-to congruence is sound for language equivalence, but not for inclusion. For the latter, we need the precongruence closure that, in the case of l -monoids, can be checked with a simple modification of the rewriting procedure. Inspired by [ACHV10], we further combine this technique with a certain notion of weighted *similarity*, a preorder that entails language inclusion and can be computed in polynomial time.

We then show how to apply our up-to techniques to language equivalence and inclusion checks for weighted automata. For some interesting semirings, such as the tropical semiring, these problems are known to be undecidable [Kro94]. But based on the inclusion algorithm we can develop an algorithm which solves the universality (also called threshold) problem for the tropical semiring over the natural numbers. This problem is known to be PSPACE-complete and we give detailed runtime results that compare our up-to threshold algorithm with one previously introduced in [ABK11].

Throughout this chapter, we use the following notations and conventions:

- For notational convenience, we assume that for index sets X it holds that $X = \{1, 2, \dots, |X|\}$ and we write a vector v as a column vector.
- $M[x, y]$ denotes the (x, y) -th entry of a matrix M and $v[x]$ denotes the x -th entry of v .

$$\begin{array}{c}
\text{(REL)} \quad \frac{v \ R \ w}{v \ c(R) \ w} \quad \text{(REFL)} \quad \frac{}{v \ c(R) \ v} \quad \text{(SYM)} \quad \frac{v \ c(R) \ w}{w \ c(R) \ v} \\
\text{(TRANS)} \quad \frac{u \ c(R) \ v \quad v \ c(R) \ w}{u \ c(R) \ w} \quad \text{(SCA)} \quad \frac{v \ c(R) \ w}{v \cdot s \ c(R) \ w \cdot s} \quad \text{where } s \in \mathbb{S} \\
\text{(PLUS)} \quad \frac{v_1 \ c(R) \ v'_1 \quad v_2 \ c(R) \ v'_2}{v_1 + v_2 \ c(R) \ v'_1 + v'_2}
\end{array}$$

Table 5.1: Proof rules for the congruence closure

- We will always require l -monoids to be completely distributive, to ensure that we have a residuation operation.

5.2 Congruence Closure

As explained at the beginning of the chapter, the key ingredient for exploiting up-to techniques in Section 5.3 is an algorithmic procedure to check whether two vectors belong to the congruence closure of a given relation of vectors. In this section, we will formalise the problem and propose an algorithmic treatment for rings and l -monoids.

5.2.1 Problem Statement

Let X be a finite set and let \mathbb{S} be a semiring. A relation $R \subseteq \mathbb{S}^X \times \mathbb{S}^X$ is a *congruence* if it is an equivalence and *closed under linear combinations*, that is, for each $(v_1, v'_1), (v_2, v'_2) \in R$ and each scalar $s \in \mathbb{S}$, $(v_1 + v_2, v'_1 + v'_2) \in R$ and $(v_1 \cdot s, v'_1 \cdot s) \in R$. The *congruence closure* $c(R)$ of a relation R over a semiring \mathbb{S} is the smallest congruence $R' \subseteq \mathbb{S}^X \times \mathbb{S}^X$ such that $R \subseteq R'$. Alternatively, two vectors $v, v' \in \mathbb{S}^X$ are in $c(R)$ whenever this can be derived via the rules in Table 5.1.

Given a finite $R \subseteq \mathbb{S}^X \times \mathbb{S}^X$ and $v, w \in \mathbb{S}^X$, we aim to determine if $(v, w) \in c(R)$.

In [BP13], Bonchi and Pous presented a procedure to compute the congruence closure for the two-valued Boolean semiring $B = \{0, 1\}$. The purpose of this section is to generalise the procedure towards more general semirings, such as rings and l -monoids.

5.2.2 Congruence Closure for Rings

A simple case to start our analysis is the congruence closure of a ring. It is kind of folklore (see e.g. [Sta03, Bor09]) that a submodule¹ can be used to represent a congruence. In particular we write $[V]$ to denote the submodule generated by a set of vectors V .

Proposition 5.2.1 *Let \mathbb{L} be a ring and X be a finite set. Let $R \subseteq \mathbb{L}^X \times \mathbb{L}^X$ be a relation and let $(v, v') \in \mathbb{L}^X \times \mathbb{L}^X$ be a pair of vectors. We construct a generating set for a submodule of \mathbb{L}^X by defining $U_R = \{u - u' \mid (u, u') \in R\}$. Then $(v, v') \in c(R)$ iff $v - v' \in [U_R]$.*

In order to prove this proposition, we first define the following two functions that translate between the two sets R and U_R :

- $u: \mathcal{P}(\mathbb{L}^X \times \mathbb{L}^X) \rightarrow \mathcal{P}(\mathbb{L}^X)$ with $u(R) = \{v - v' \mid (v, v') \in R\}$.
- $r: \mathcal{P}(\mathbb{L}^X) \rightarrow \mathcal{P}(\mathbb{L}^X \times \mathbb{L}^X)$ with $r(U) = \{(v, v') \mid v - v' \in U\}$, where $U \subseteq \mathbb{L}^X$, i.e., U is a set of \mathbb{L} -vectors.

For the functions r and u we can prove some basic properties.

Lemma 5.2.2

- (i) *Let \mathbb{L} be a ring. Let $R \subseteq \mathbb{L}^X \times \mathbb{L}^X$ be a congruence. Then $u(R)$ is a module.*
- (ii) *Let \mathbb{L} be a ring. Let $U \subseteq \mathbb{L}^X$ be a module. Then $r(U)$ is a congruence.*

Proof:

- (i) We will show that $u(R)$ contains all vectors generated via linear combination from $u(R)$, making $u(R)$ a generating set for itself, i.e. a module.
 - Let $v'' \in u(R)$ then there must be $v, v' \in \mathbb{L}^X$ such that $(v, v') \in R$ and $v - v' = v''$. Since R is a congruence, it follows that $(v \cdot s, v' \cdot s) \in R$ for any $s \in \mathbb{L}$. This means that $v \cdot s - v' \cdot s \in u(R)$, distributivity now proves $(v - v') \cdot s \in u(R)$, i.e. $v'' \cdot s \in u(R)$.
 - Let $v''_1, v''_2 \in u(R)$. Then there must be $v_1, v'_1, v_2, v'_2 \in \mathbb{L}^X$ such that $v''_i = v_i - v'_i$ and $(v_i, v'_i) \in R$ for $i = 1, 2$. Since R is a congruence, it follows that $(v_1 + v_2, v'_1 + v'_2) \in R$. Thus, $(v_1 + v_2) - (v'_1 + v'_2) \in u(R)$.

¹A subsemimodule for a ring is called submodule.

Commutativity of addition yields $(v_1 - v'_1) + (v_2 - v'_2) \in u(R)$, i.e. $v''_1 + v''_2 \in u(R)$.

- (ii) • *Reflexivity*: Let any $v \in U$ be given, then $v \cdot 0 \in U$, so the 0-vector is in U . For any given $v \in \mathbb{L}^X$, $v - v = 0$, thus $(v, v) \in r(U)$.
- *Symmetry*: Let $(v, v') \in r(U)$, then $v - v' \in U$. Since U is a module, $(v - v') \cdot (-1) \in U$ and thus $-v + v' = v' - v \in U$, therefore $(v', v) \in r(U)$.
- *Transitivity*: Let $(v, v') \in r(U)$, $(v', v'') \in r(U)$, then $v - v' \in U$ and $v' - v'' \in U$. Since U is a module, $(v - v') + (v' - v'') \in U$ and thus $v - v'' \in U$. Therefore $(v, v'') \in r(U)$.
- *Addition*: Let $(v_1, v_2) \in r(U)$ and $(v'_1, v'_2) \in r(U)$, then $v_1 - v_2 \in U$ and $v'_1 - v'_2 \in U$. Therefore $(v_1 - v_2) + (v'_1 - v'_2) \in U$. Commutativity yields $(v_1 + v'_1) - (v_2 + v'_2) \in U$, i.e. $(v_1 + v'_1, v_2 + v'_2) \in r(U)$.
- *Multiplication*: Let $(v, v') \in r(U)$ and $s \in \mathbb{L}$. Then $v - v' \in U$. Since U is a module, $(v - v') \cdot s \in U$, distributivity yields $v \cdot s - v' \cdot s \in U$ and thus per definition $(v \cdot s, v' \cdot s) \in r(U)$.

□

Using these properties, we are now prepared to prove Proposition 5.2.1.

Proof (of Proposition 5.2.1): According to Lemma 5.2.2 we know that if R is a congruence, then $u(R)$ is a submodule and if U is a submodule then $r(U)$ is a congruence.

Observe that $R \subseteq R'$ implies $u(R) \subseteq u(R')$ via definition of u and $r(U) \subseteq r(U')$ whenever $U \subseteq U'$, by definition of r .

Observe furthermore that $r(u(R)) \subseteq c(R)$ holds, because if $(v_1, v_2) \in r(u(R))$, then there exists a $(v'_1, v'_2) \in R$ such that $v_1 - v_2 = v'_1 - v'_2$, hence $v'_1 - v_1 = v'_2 - v_2$. Now $(v'_1 - v_1, v'_1 - v_1) \in c(R)$ due to reflexivity and thus we obtain: $(v_1, v_2) + (v'_1 - v_1, v'_1 - v_1) = (v_1, v_2) + (v'_1 - v_1, v'_2 - v_2) = (v_1 + v'_1 - v_1, v_2 + v'_2 - v_2) = (v'_1, v'_2)$, hence $(v'_1, v'_2) \in c(R)$. Thus, $r(u(R)) \subseteq c(R)$, proving also that congruences are fixed points of the monotone function $r \circ u$, since $r(U)$ is always a congruence and for every congruence R it holds that $c(R) = R$.

Now we can observe that the module generated by a set of vectors is the smallest module that contains this set and the congruence closure of a relation is the smallest congruence closed relation containing that relation.

We will now show $r([U_R]) = r([u(R)]) = c(R)$, thus proving the statement of the proposition. We have $u(R) \subseteq u(c(R))$ and we know that $u(c(R))$ is a submodule from Lemma 5.2.2.(i), hence the submodule generated by $u(R)$ is included in $u(c(R))$, i.e. $[u(R)] \subseteq u(c(R))$. Therefore, $r([u(R)]) \subseteq r(u(c(R))) = c(R)$, and since Lemma 5.2.2.(ii) shows that r applied to a submodule yields a congruence and we have $R \subseteq r([u(R)]) \subseteq c(R)$, the second inclusion is indeed an equality. \square

This procedure directly yields a simple algorithm for a congruence check whenever we have a suitable algorithm to solve linear equations, e.g. for fields. If the ring is not a field, it might still be possible to embed it into a field. In this case we can for instance solve the language equivalence problem (Subsection 5.3.2) for weighted automata in the field and the results are also valid in the ring. Similarly, the procedure can be used for probabilistic automata which can be seen as weighted automata over the reals.

5.2.3 Embedding Semirings into Fields

This is an interesting property for the analysis of weighted automata. Assume a weighted automaton over a semiring that can be embedded into a field is given. Then one can embed the weights of the automaton into the field and decide language equivalence on the field. Since any pair of states is language equivalent in the field if and only if they are language equivalent in the semiring, this is a correct decision procedure for language equivalence. Note, however, that this embedding cannot be used to compute the congruence closure in the semiring, since factors that do not exist in the semiring may be required for some linear combinations.

Integral domains can be seen as sub-structures of fields and therefore, deciding language equivalence for weighted automata over integral domains is not only decidable but can be efficiently computed in just as many steps as there are states in the automaton. However, integral domains are special rings and weighted automata are defined over arbitrary semirings, so it is natural to ask, under which circumstances we can embed a semiring into a ring. We will give a necessary and sufficient condition for this. The construction uses the Grothendieck group of the additive monoid. Since some proofs in this section are rather technical and not important to the rest of this chapter, the interested

reader may be referred to Appendix B.1 for proofs that are missing in the main text.

We first need to show the following lemma:

Lemma 5.2.3 *Let $(\mathbb{S}, +, \cdot, 0, 1)$ be a semiring, then $(\mathbb{S} \times \mathbb{S}, +, \cdot, (0, 0), (1, 0))$ where $(s_1, s_2) + (s'_1, s'_2) = (s_1 + s'_1, s_2 + s'_2)$ and $(s_1, s_2) \cdot (s'_1, s'_2) = (s_1 \cdot s'_1 + s_2 \cdot s'_2, s_1 \cdot s'_2 + s_2 \cdot s'_1)$ is a semiring.*

Proposition 5.2.4 *Let \mathbb{S} be a semiring. Then \mathbb{S} is a subsemiring of a ring \mathbb{I} iff for all elements $s_1, s_2, s_3 \in \mathbb{S}$, $s_1 + s_2 = s_1 + s_3$ implies $s_2 = s_3$ (we say: $+$ is injective).*

Hence, in these cases, one can work in the ring the semiring embeds into. However, if one wants to be able to use Gaussian elimination for solving linear equations, it is necessary to have an integral domain. We will now see in which cases the super-ring of a semiring is an integral domain.

Proposition 5.2.5 *If a semiring \mathbb{S}*

- (i) *has injective $+$*
- (ii) *for each pair of elements $s_1, s_2 \in \mathbb{S}$ there always exists an element $\bar{s} \in \mathbb{S}$ such that $s_1 + \bar{s} = s_2$ or $s_2 + \bar{s} = s_1$*
- (iii) *and \mathbb{S} has no zero-divisors,*

then the super-ring \mathbb{I} of \mathbb{S} , which exists according to Proposition 5.2.4, is an integral domain, i.e. it also has no zero-divisors.

Proof: We first show that the only pairs equivalent to $(0, 0)$ are of the form (s, s) . Assume that $(s_1, s_2) \equiv (0, 0)$. This means that there exist s, s' such that $(s_1, s_2) + (s, s) = (0, 0) + (s', s') = (s', s')$. This implies $s_1 + s = s' = s_2 + s$ and from Condition (i) (injectivity of addition) we infer that $s_1 = s_2$.

Assume now that $(s_1, s_2) \cdot (s'_1, s'_2) \equiv (0, 0)$. According to Condition (ii), there exists an s such that $s_1 + s = s_2$ or $s_2 + s = s_1$. Hence in the first case $(s_1, s_2) + (0, 0) = (0, s) + (s_1, s_1)$ and hence $(s_1, s_2) \equiv (0, s)$. In the second case $(s_1, s_2) \equiv (s, 0)$. Assume $(s_1, s_2) \equiv (\bar{s}, 0)$ and $(s'_1, s'_2) \equiv (\bar{s}', 0)$ (the other cases are analogous). Then $(0, 0) \equiv (s_1, s_2) \cdot (s'_1, s'_2) \equiv (\bar{s}, 0) \cdot (\bar{s}', 0)$, i.e. there exists an element $s \in \mathbb{S}$ such that $(\bar{s}, 0) \cdot (\bar{s}', 0) = (s, s)$. Thus, $\bar{s} \cdot \bar{s}' = s$ and $0 = s$, hence $\bar{s} \cdot \bar{s}' = 0$. From Condition (iii) it follows that $\bar{s}_1 = 0$ or $\bar{s}'_1 = 0$, in either case, one of (s_1, s_2) and (s'_1, s'_2) is equivalent to $(0, 0)$. \square

5.2.4 Congruence Closure for l -Monoids

We now turn our attention towards l -monoids, to develop a technique to decide, whether a pair of vectors over an l -monoid belongs to the congruence closure of a relation (of pairs of vectors over this l -monoid). We will work towards adapting a rewriting technique developed by Bonchi and Pous [BP13]. So the general idea is to define a rewriting procedure based on a relation R that transforms any given vector into a (unique) normal form that has the property that any two given vectors are related by the congruence closure of R if and only if their normal forms coincide. We later want to use this rewriting technique to adapt the bisimulation proof technique up to congruence to weighted automata over l -monoids.

Rewriting and Normal Forms.

Our method to determine if a pair of vectors is in the congruence closure is to employ a rewriting algorithm that rewrites both vectors to a normal form. These coincide iff the vectors are related by the congruence closure. Note, that we will use the the l -monoid operations also on vectors, extending the operators component-wise.

Definition 5.2.6 (Rewriting and normal forms) *Let \mathbb{L} be an integral l -monoid and let $R \subseteq \mathbb{L}^X \times \mathbb{L}^X$ be a finite relation.*

We define a set of rewriting rules \mathcal{R} as follows: For each pair of vectors $(v, v') \in R$, we obtain two rewriting rules $v \mapsto v \sqcup v'$ and $v' \mapsto v \sqcup v'$.

A rewriting step works as follows: given a vector v and a rewriting rule $l \mapsto r$, we compute the residuum $l \rightarrow v$ and, provided $v \sqsubseteq (v \sqcup r \cdot (l \rightarrow v))$, the rewriting rule is applicable and v rewrites to $v \sqcup r \cdot (l \rightarrow v)$ (symbolically: $v \rightsquigarrow v \sqcup r \cdot (l \rightarrow v)$). A vector v is in normal form wrt. R , provided there exists no rule that is applicable to v .

Example 5.2.7 *In order to illustrate how rewriting works, we work in \mathbb{T} , set $X = \{1, 2\}$ (two dimensions) and take the relation $R = \left\{ \left(\begin{pmatrix} \infty \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \infty \end{pmatrix} \right) \right\} \subseteq \mathbb{T}^2 \times \mathbb{T}^2$, relating the two unit vectors, and the vector $v = \begin{pmatrix} \infty \\ 3 \end{pmatrix}$. This yields a rule $l = \begin{pmatrix} \infty \\ 0 \end{pmatrix} \mapsto r = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. We obtain $l \rightarrow v = 3$ and hence $v \rightsquigarrow v \sqcup r \cdot (l \rightarrow v) = \begin{pmatrix} \infty \\ 3 \end{pmatrix} \min \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix} + 3 \right) = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$.*

It is worth to observe that when \mathbb{L} is the binary Boolean semiring, the above procedure coincides with the one in [BP13]. There is an alternative characterisation of residuation for the use in the rewriting technique, which is equivalent to the one given above in terms of fixpoints, which we will use in some of the proofs in this section:

Lemma 5.2.8 *Let $(\mathbb{L}, \sqcup, \cdot, 0, 1)$ be an l -monoid and v, v' be n -dimensional \mathbb{L} -vectors. Then it holds that*

$$v \rightarrow v' = \bigsqcap \{v[i] \rightarrow v'[i] \mid 1 \leq i \leq n\}$$

Proof: We define $(v \Rightarrow v') := \bigsqcap \{v[i] \rightarrow v'[i] \mid 1 \leq i \leq n\}$.

First we will show that $v \Rightarrow v' \sqsubseteq v \rightarrow v'$. In order to prove this, we will show that $v \cdot (v \Rightarrow v') \leq v'$, i.e. $(v \Rightarrow v') \in \{\ell \in \mathbb{L} \mid v \cdot \ell \sqsubseteq v'\}$:

$$\begin{aligned} v \cdot (v \Rightarrow v') &= \begin{pmatrix} v[1] \\ v[2] \\ \vdots \\ v[n] \end{pmatrix} \cdot (v \Rightarrow v') = \begin{pmatrix} v[1] \cdot (v \Rightarrow v') \\ v[2] \cdot (v \Rightarrow v') \\ \vdots \\ v[n] \cdot (v \Rightarrow v') \end{pmatrix} \\ &\sqsubseteq \begin{pmatrix} v[1] \cdot (v[1] \rightarrow v'[1]) \\ v[2] \cdot (v[2] \rightarrow v'[2]) \\ \vdots \\ v[n] \cdot (v[n] \rightarrow v'[n]) \end{pmatrix} \sqsubseteq \begin{pmatrix} v'[1] \\ v'[2] \\ \vdots \\ v'[n] \end{pmatrix} = v' \end{aligned}$$

Next we need to show that $v \Rightarrow v' \sqsupseteq v \rightarrow v'$. It suffices to show that $v \Rightarrow v'$ is an upper bound of the set $\{\ell \in \mathbb{L} \mid v \cdot \ell \sqsubseteq v'\}$. Since $v \Rightarrow v'$ is the greatest lower bound of $\{v[i] \rightarrow v'[i] \mid 1 \leq i \leq n\}$, it is enough to show that every element of the first set and every element of the second set are in relation. Thus, take $\ell \in \mathbb{L}$ with $v \cdot \ell \sqsubseteq v'$ and an index i . Since $v \cdot \ell \sqsubseteq v'$ (component-wise), it holds that $v[i] \cdot \ell \sqsubseteq v'[i]$ for every i . Hence, $\ell \sqsubseteq v[i] \rightarrow v'[i]$.

□

In order to analyse the rewriting procedure, we need some monotonicity results concerning the operations at play in the rewriting algorithm.

Lemma 5.2.9 *In an l -monoid \mathbb{L} , for all $\ell, \ell', \ell_1, \ell_2, \ell_3$ the following hold:*

(i) *Whenever $\ell_1 \sqsubseteq \ell_2$ it follows that $\ell \cdot \ell_1 \sqsubseteq \ell \cdot \ell_2$ and $\ell_1 \cdot \ell \sqsubseteq \ell_2 \cdot \ell$.*

- (ii) For all matrices $M \in \mathbb{L}^{X \times Y}$ it holds that $v \sqsubseteq v'$ implies $Mv \sqsubseteq Mv'$.
- (iii) If \mathbb{L} is integral, $\ell \cdot \ell' \sqsubseteq \ell'$ and $\ell \cdot \ell' \sqsubseteq \ell$.
- (iv) $(\ell_1 \rightarrow \ell_2) \cdot \ell_3 \sqsubseteq \ell_1 \rightarrow (\ell_2 \cdot \ell_3)$.
- (v) If $\ell_2 \sqsubseteq \ell_3$ it follows that $(\ell_1 \rightarrow \ell_2) \sqsubseteq (\ell_1 \rightarrow \ell_3)$.
- (vi) $\ell_3 \sqsubseteq \ell_1 \rightarrow \ell_2 \iff \ell_1 \cdot \ell_3 \sqsubseteq \ell_2$.
- (vii) $\ell_1 \rightarrow (\ell_2 \sqcup \ell_3) \sqsupseteq (\ell_1 \rightarrow \ell_2) \sqcup (\ell_1 \rightarrow \ell_3)$.
- (viii) $\ell_1 \cdot (\ell_1 \rightarrow \ell_2) \sqsubseteq \ell_2$

Proof:

(i)

$$\ell \cdot \ell_2 = \ell \cdot (\ell_1 \sqcup \ell_2) = \ell \cdot \ell_1 \sqcup \ell \cdot \ell_2 \sqsupseteq \ell \cdot \ell_1$$

and

$$\ell_2 \cdot \ell = (\ell_1 \sqcup \ell_2) \cdot \ell = \ell_1 \cdot \ell \sqcup \ell_2 \cdot \ell \sqsupseteq \ell_1 \cdot \ell.$$

(ii) The second part follows directly, because

$$(Mv)[i] = \bigsqcup_{j \in Y} M[i, j] \cdot v[j] \sqsubseteq \bigsqcup_{v[j] \sqsubseteq v'[j]} M[i, j] \cdot v'[j] = (Mv')[i]$$

(iii) This follows directly from monotonicity, $\ell \cdot \ell' \sqsubseteq \top \cdot \ell' = 1 \cdot \ell' = \ell'$ and $\ell \cdot \ell' \sqsubseteq \ell \cdot \top = \ell \cdot 1 = \ell$.

(iv) We first compute:

$$(\ell_1 \rightarrow \ell_2) \cdot \ell_3 = \bigsqcup \{ \ell \in \mathbb{L} \mid \ell_1 \cdot \ell \sqsubseteq \ell_2 \} \cdot \ell_3 = \bigsqcup \{ \ell \cdot \ell_3 \mid \ell_1 \cdot \ell \sqsubseteq \ell_2 \}$$

and:

$$\ell_1 \rightarrow (\ell_2 \cdot \ell_3) = \bigsqcup \{ \ell \in \mathbb{L} \mid \ell_1 \cdot \ell \sqsubseteq \ell_2 \cdot \ell_3 \}$$

Now we obtain:

$$\ell_1 \cdot \ell \sqsubseteq \ell_2 \Rightarrow \ell_1 \cdot \ell \cdot \ell_3 \sqsubseteq \ell_2 \cdot \ell_3$$

and therefore $\{ \ell \cdot \ell_3 \mid \ell_1 \cdot \ell \sqsubseteq \ell_2 \} \subseteq \{ \ell \in \mathbb{L} \mid \ell_1 \cdot \ell \sqsubseteq \ell_2 \cdot \ell_3 \}$.

(v) Obviously, $\{ \ell \in \mathbb{L} \mid \ell_1 \cdot \ell \sqsubseteq \ell_2 \} \subseteq \{ \ell \in \mathbb{L} \mid \ell_1 \cdot \ell \sqsubseteq \ell_3 \}$ and therefore:

$$\ell_1 \rightarrow \ell_2 = \bigsqcup \{ \ell \in \mathbb{L} \mid \ell_1 \cdot \ell \sqsubseteq \ell_2 \} \sqsubseteq \bigsqcup \{ \ell \in \mathbb{L} \mid \ell_1 \cdot \ell \sqsubseteq \ell_3 \} = \ell_1 \rightarrow \ell_3$$

.

(vi) Whenever $\ell_3 \sqsubseteq \ell_1 \rightarrow \ell_2$, then $\ell_1 \cdot \ell_3 \sqsubseteq \ell_1 \cdot (\ell_1 \rightarrow \ell_2) = \ell_1 \cdot \bigsqcup \{ \ell \in \mathbb{L} \mid \ell_1 \cdot \ell \sqsubseteq \ell_2 \} = \bigsqcup \{ \ell_1 \cdot \ell \in \mathbb{L} \mid \ell_1 \cdot \ell \sqsubseteq \ell_2 \} \sqsubseteq \ell_2$.

Whenever $\ell_1 \cdot \ell_3 \sqsubseteq \ell_2$ we have that $\ell_1 \rightarrow \ell_2 = \bigsqcup \{ \ell \in \mathbb{L} \mid \ell_1 \cdot \ell \sqsubseteq \ell_2 \} \sqsupseteq \ell_3$, since ℓ_3 is an element of the set.

- (vii) Because of Lemma 5.2.9.(vi) it suffices to show that $\ell_1 \cdot ((\ell_1 \rightarrow \ell_2) \sqcup (\ell_1 \rightarrow \ell_3)) = \ell_1 \cdot (\ell_1 \rightarrow \ell_2) \sqcup \ell_1 \cdot (\ell_1 \rightarrow \ell_3) \sqsubseteq \ell_2 \sqcup \ell_3$.
- (viii) $\ell_1 \cdot (\ell_1 \rightarrow \ell_2) = \ell_1 \cdot \sqcup \{\ell' \mid \ell_1 \cdot \ell' \sqsubseteq \ell_2\} = \sqcup \{\ell_1 \cdot \ell' \mid \ell_1 \cdot \ell' \sqsubseteq \ell_2\} \sqsubseteq \sqcup \{\ell_2\} = \ell_2$

□

The rewriting relation satisfies some simple properties:

Lemma 5.2.10 *In an l -monoid \mathbb{L} , for all l -monoid vectors $v, v', w \in \mathbb{L}^X$, it holds that:*

- (i) *If $v \rightsquigarrow v'$ and $v \sqsubseteq w$, then $v' \sqsubseteq w$ or there exists $w' \in \mathbb{L}^X$ s.t. $w \rightsquigarrow w'$ and $v' \sqsubseteq w'$.*
- (ii) *Whenever $v \rightsquigarrow v'$, there exists a vector $u \in \mathbb{L}^X$ s.t. $v \sqcup w \rightsquigarrow u \sqsubseteq v' \sqcup w$ or $v \sqcup w = v' \sqcup w$.*

Proof:

- (i) Assume that $v \rightsquigarrow v'$, via rule $l \mapsto r$, and $v \sqsubseteq w$. Then $v' = v \sqcup r \cdot (l \rightarrow v) \sqsubseteq w \sqcup r \cdot (l \rightarrow w) =: w'$. Thus, either $w \rightsquigarrow w'$ or $w = w'$ and in this case $w \sqsupseteq v'$.
- (ii) Assume that $v \rightsquigarrow v'$, via rule $l \mapsto r$. Define $u := (v \sqcup w) \sqcup r \cdot (l \rightarrow (v \sqcup w)) \sqsupseteq (v \sqcup w) \sqcup r \cdot ((l \rightarrow v) \sqcup (l \rightarrow w)) = (v \sqcup r \cdot (l \rightarrow v)) \sqcup (w \sqcup r \cdot (l \rightarrow w)) \sqsupseteq v' \sqcup w$. The first inequality is due to Lemma 5.2.9.(vii).

Now either $v \sqcup w \rightsquigarrow u$ or $v \sqcup w = u \sqsupseteq v' \sqcup w$. Since we also have that $v \sqcup w \sqsubseteq v' \sqcup w$ due to extensiveness of rewriting, this implies $v \sqcup w = v' \sqcup w$.

□

We now have to prove the following three statements:

- (i) Our technique is sound, i.e. whenever two vectors have the same normal form wrt. R , they are in $c(R)$.
- (ii) Our technique is complete, i.e. whenever two vectors are in $c(R)$, they have the same normal form wrt. R .
- (iii) Our algorithm to compute normal forms terminates.

We will show (i) and prove that (ii) follows from (iii). Afterwards, we will discuss sufficient conditions and examples where (iii) holds.

Theorem 5.2.11 *Whenever there exists a vector \bar{v} , such that two vectors v_1, v_2 both rewrite to \bar{v} , i.e., $v_1 \rightsquigarrow^* \bar{v}$, $v_2 \rightsquigarrow^* \bar{v}$, then $(v_1, v_2) \in c(R)$.*

Proof: We will show that if v rewrites to v' via a rule $l \mapsto r$, then $(v, v') \in c(R)$.

Since $l \mapsto r$ is a rewriting rule, we have that $l = w$, $r = w \sqcup w'$ for $(w, w') \in R$ or $(w', w) \in R$. In both cases $w = w \sqcup w \sqsubset c(R) w \sqcup w'$ due to the definition of congruence closure, using rules (PLUS), (REL) and (REFL), as well as (SYM) in case $(w', w) \in R$. Hence, $l \sqsubset c(R) r$. This implies that $l \cdot (l \rightarrow v) \sqsubset c(R) r \cdot (l \rightarrow v)$ (SCA) and furthermore $v \sqcup l \cdot (l \rightarrow v) \sqsubset c(R) v \sqcup r \cdot (l \rightarrow v)$ (PLUS). Since $l \cdot (l \rightarrow v) \sqsubseteq v$ we have $v \sqcup l \cdot (l \rightarrow v) = v$ and hence $v \sqsubset c(R) v'$. \square

This concludes the proof of soundness, we will go on proving completeness.

Lemma 5.2.12 *Assume we have a rewriting system that always terminates, i.e. a Noetherian rewriting system. Then the local Church-Rosser property holds. That is, whenever $v \rightsquigarrow v_1$ and $v \rightsquigarrow v_2$, there exists a vector v' such that $v_1 \rightsquigarrow^* v'$ and $v_2 \rightsquigarrow^* v'$.*

Proof: Assume that $v \rightsquigarrow v_1$ and $v \rightsquigarrow v_2$. We set $v_0^a = v$, $v_1^a = v_1$ and consider a sequence of rewriting steps $v_1^a \rightsquigarrow v_2^a \rightsquigarrow \dots \rightsquigarrow v_n^a \not\rightsquigarrow$ that leads to the normal form v_n^a .

We now construct a sequence of vectors v_1^b, \dots, v_{n+1}^b where $v_1^b = v_2$, $v_i^b \rightsquigarrow v_{i+1}^b$ or $v_i^b = v_{i+1}^b$, and $v_{i+1}^b \sqsubseteq v_i^a$.

Given v_i^b with $i \geq 1$, Lemma 5.2.10.(i) inductively guarantees the existence of $v_{i+1}^b \sqsubseteq v_i^a$ with $v_i^b \rightsquigarrow v_{i+1}^b$, or $v_i^b \sqsubseteq v_i^a$. In the latter case we set $v_{i+1}^b = v_i^b$.

Since $v \rightsquigarrow^* v_{n+1}^b$ and v_n^a is a normal form, it must hold that $v_{n+1}^b \sqsubseteq v_n^a$. We also know from above that $v_{n+1}^b \sqsubseteq v_n^a$, hence $v_{n+1}^b = v_n^a$. Therefore, this is the vector v' which is reachable from both v_1 and v_2 and which proves the local Church-Rosser property. \square

If a rewriting system terminates and the local Church-Rosser property holds, the system is automatically confluent [DJ90]. In this case, every vector v is associated with a unique normal form, written $\Downarrow_{\mathcal{R}} v$ or simply $\Downarrow v$ where $v \rightsquigarrow^* \Downarrow v \not\rightsquigarrow$.

Furthermore, due to Lemma 5.2.10.(i) we know that \Downarrow is monotone, i.e., $v \sqsubseteq v'$ implies $\Downarrow v \sqsubseteq \Downarrow v'$. This also implies $\Downarrow(v \sqcup v') \sqsubseteq (\Downarrow v) \sqcup (\Downarrow v')$.

Lemma 5.2.13 *For all $v \in \mathbb{L}^X, \ell \in \mathbb{L}$ we have that, if $v \rightsquigarrow v'$, then $v \cdot \ell \rightsquigarrow v''$ for some $v'' \sqsupseteq v' \cdot \ell$ or $v \cdot \ell = v' \cdot \ell$. In particular, if rewriting terminates, we have $(\Downarrow v) \cdot \ell \sqsubseteq \Downarrow(v \cdot \ell)$.*

Proof: Assume that $v \rightsquigarrow v'$ via rule $l \mapsto r$. Hence $v' = v \sqcup r \cdot (l \rightarrow v)$. Let $v'' := v \cdot \ell \sqcup r \cdot (l \rightarrow v \cdot \ell)$. By adapting the proof of Lemma 5.2.9.(iv) to vectors we can show that $v'' \sqsupseteq v \cdot \ell \sqcup r \cdot (l \rightarrow v) \cdot \ell = (v \sqcup r \cdot (l \rightarrow v)) \cdot \ell = v' \cdot \ell$. Hence either $v \cdot \ell \rightsquigarrow v''$ or $v \cdot \ell = v''$. In the latter case $v \cdot \ell \sqsupseteq v' \cdot \ell$ (since $v'' \sqsupseteq v' \cdot \ell$), but also $v \cdot \ell \sqsubseteq v' \cdot \ell$ (since $v \sqsubseteq v'$). Hence $v \cdot \ell = v' \cdot \ell$.

Now assume that $v = v_0 \rightsquigarrow v' = v_1 \rightsquigarrow v_2 \rightsquigarrow \dots \rightsquigarrow v_n = \Downarrow v$. We construct a sequence of vectors w_i where $w_0 = v \cdot \ell$, $w_i \sqsupseteq v_i \cdot \ell$ and either $w_i \rightsquigarrow w_{i+1}$ or $w_i = w_{i+1}$.

Given $w_i \sqsupseteq v_i \cdot \ell$ and $v_i \rightsquigarrow v_{i+1}$. We know that one of the following two cases holds:

- there exists v'' such that $v_i \cdot \ell \rightsquigarrow v'' \sqsupseteq v_{i+1} \cdot \ell$: now, since $w_i \sqsupseteq v_i \cdot \ell$, we know due to Lemma 5.2.10.(i) that there exists w_{i+1} such that $w_i \rightsquigarrow w_{i+1} \sqsupseteq v'' \sqsupseteq v_{i+1} \cdot \ell$ or $w_i \sqsupseteq v''$. In the second case we set $w_{i+1} = w_i$.
- or $v_i \cdot \ell = v_{i+1} \cdot \ell$: again we set $w_{i+1} = w_i$ and obtain $w_{i+1} = w_i \sqsupseteq v_i \cdot \ell = v_{i+1} \cdot \ell$.

Hence, $w_n \sqsupseteq v_n = \Downarrow v$ and via monotonicity we obtain $\Downarrow(v \cdot \ell) = \Downarrow w_0 = \Downarrow w_n \sqsupseteq \Downarrow v_n = \Downarrow v$.

□

Now we have all the necessary ingredients to show that the technique is complete, provided the computation of a normal form terminates.

Theorem 5.2.14 *Assume that rewriting terminates. If $v \ c(R) \ v'$ then $\Downarrow v = \Downarrow v'$.*

Proof: It suffices to show that $\Downarrow v \sqsupseteq v'$ and $\Downarrow v' \sqsupseteq v$, because if $v \sqsubseteq \Downarrow v'$, then $\Downarrow v \sqsubseteq \Downarrow v'$ (\Downarrow is monotone and idempotent) and vice-versa. We prove this via rule induction (cf. rules in Table 5.1).

(Rel) If we find that $v \ c(R) \ v'$ because $v \ R \ v'$, then there are rules $v \mapsto v \sqcup v'$ and $v' \mapsto v \sqcup v'$.

Hence, v rewrites to $v \sqcup (v \sqcup v') \cdot (v \rightarrow v) \geq v \sqcup v'$, since $v \rightarrow v \sqsupseteq 1$ (or v can not be rewritten via this rule). Thus, either v is rewritten to a vector larger or equal v' or $v \sqsupseteq v'$ holds. We can conclude that $\Downarrow v \sqsupseteq v'$.

Analogously one can show $\Downarrow v' \sqsupseteq v$.

(Refl) If we find that $v \ c(R) \ v'$ because of reflexivity (i.e. $v = v'$), then trivially $\Downarrow v = \Downarrow v'$.

(Sym) If we find that $v \ c(R) \ v'$ because of symmetry, then we already know from the induction hypothesis that $\Downarrow v \sqsupseteq v'$ and $\Downarrow v' \sqsupseteq v$.

(Trans) If we find that $v_1 \ c(R) \ v_3$ because of transitivity, i.e. $v_1 \ c(R) \ v_2$ and $v_2 \ c(R) \ v_3$, we know from the induction hypothesis that $\Downarrow v_1 \sqsupseteq v_2$ and $\Downarrow v_2 \sqsupseteq v_3$ as well as $\Downarrow v_3 \sqsupseteq v_2$ and $\Downarrow v_2 \sqsupseteq v_1$. In particular, we have $v_1 \sqsubseteq \Downarrow v_2 \sqsubseteq \Downarrow v_3$ and $v_3 \sqsubseteq \Downarrow v_2 \sqsubseteq \Downarrow v_1$.

(Sca) If we find that $v \cdot \ell \ c(R) \ v' \cdot \ell$, because $v \ c(R) \ v'$, then $v \sqsubseteq \Downarrow v'$ and therefore, using Lemma 5.2.13

$$v \cdot \ell \sqsubseteq (\Downarrow v) \cdot \ell \sqsubseteq (\Downarrow v') \cdot \ell \sqsubseteq \Downarrow (v' \cdot \ell).$$

(Plus) If we find that $\bar{v} \sqcup v \ c(R) \ \bar{v}' \sqcup v'$ because of $\bar{v} \ c(R) \ \bar{v}'$ and $v \ c(R) \ v'$, then

$$v \sqcup \bar{v} \sqsubseteq (\Downarrow v') \sqcup (\Downarrow \bar{v}') \sqsubseteq \Downarrow (v' \sqcup \bar{v}'),$$

due to the monotonicity of \Downarrow .

□

Termination.

One technique to prove termination is given in Corollary 5.2.16: It suffices to show that the supremum of all the elements reachable via \rightsquigarrow is included in the congruence class. First we need the following result.

Proposition 5.2.15 *If $v \ c(R) \ \bar{v}$, then there exists a $v' \sqsupseteq v \sqcup \bar{v}$ such that $v \rightsquigarrow^* v'$.*

Proof: We prove this via rule induction (cf. rules in Table 5.1).

(Rel) If we find that $v \ c(R) \ \bar{v}$, because $v \ R \ \bar{v}$, then there are rules $v \mapsto v \sqcup \bar{v}$ and $\bar{v} \mapsto v \sqcup \bar{v}$. As in the proof of Theorem 5.2.14 we obtain that v rewrites to a vector larger or equal $v \sqcup \bar{v}$ in one step or that v itself has this property. Analogously for \bar{v} .

(Refl) If we find that $v \ c(R) \ \bar{v}$, because of reflexivity (i.e. $\bar{v} = v$), then no rewriting step is needed.

(Sym) If we find that $v \ c(R) \ \bar{v}$, because of symmetry, then we already know this from the induction hypothesis, because the property we want to prove is symmetric.

(Trans) If we find that $v_1 \ c(R) \ v_3$ because of transitivity, i.e. $v_1 \ c(R) \ v_2$ and $v_2 \ c(R) \ v_3$, we know inductively that there exist vectors v'_1, v'_2 such that $v_1 \rightsquigarrow^* v'_1 \sqsupseteq v_1 \sqcup v_2$, $v_2 \rightsquigarrow^* v'_2 \sqsupseteq v_2 \sqcup v_3$.

Now, due to Lemma 5.2.10.(ii), $v_1 \sqcup v_2 \rightsquigarrow^* u \sqsupseteq v_1 \sqcup v'_2 \sqsupseteq v_1 \sqcup v_2 \sqcup v_3 \sqsupseteq v_1 \sqcup v_3$. Furthermore, since $v'_1 \sqsupseteq v_1 \sqcup v_2$ we know from Lemma 5.2.10.(ii) that there exists a v''_1 such that $v_1 \rightsquigarrow^* v'_1 \rightsquigarrow^* v''_1 \sqsupseteq u$. Combined, we obtain $v_1 \rightsquigarrow^* v''_1 \sqsupseteq v_1 \sqcup v_3$.

For v_3 the proof is analogous.

(Sca) If we have $v_1 \cdot \ell \ c(R) \ v_2 \cdot \ell$, because $v_1 \ c(R) \ v_2$ then there exist vectors v'_1, v'_2 , such that $v_1 \rightsquigarrow^* v'_1 \sqsupseteq v_1 \sqcup v_2$ and $v_2 \rightsquigarrow^* v'_2 \sqsupseteq v_1 \sqcup v_2$. Thus, using Lemma 5.2.13, we can find v''_1, v''_2 such that

$$\begin{aligned} v_1 \cdot \ell \rightsquigarrow^* v''_1 \sqsupseteq v'_1 \cdot \ell \sqsupseteq (v_1 \sqcup v_2) \cdot \ell &= v_1 \cdot \ell \sqcup v_2 \cdot \ell \\ v_2 \cdot \ell \rightsquigarrow^* v''_2 \sqsupseteq v'_2 \cdot \ell \sqsupseteq (v_1 \sqcup v_2) \cdot \ell &= v_1 \cdot \ell \sqcup v_2 \cdot \ell \end{aligned}$$

(Plus) If we have $v_1 \sqcup v_2 \ c(R) \ v'_1 \sqcup v'_2$ because of $v_1 \ c(R) \ v'_1$ and $v_2 \ c(R) \ v'_2$, then $v_1 \rightsquigarrow^* v''_1 \sqsupseteq v_1 \sqcup v'_1$ and $v_2 \rightsquigarrow^* v''_2 \sqsupseteq v_2 \sqcup v'_2$ and we obtain with Lemma 5.2.10:

$$v_1 \sqcup v_2 \rightsquigarrow^* v \sqsupseteq v''_1 \sqcup v''_2 \sqsupseteq (v_1 \sqcup v'_1) \sqcup (v_2 \sqcup v'_2) = (v_1 \sqcup v_2) \sqcup (v'_1 \sqcup v'_2).$$

Analogously for v'_1, v'_2 .

□

If we assume that rule application is fair, we can guarantee that \bar{v} is eventually reached in every rewriting sequence.

Corollary 5.2.16 *If $v \in c(R) \sqcup \{\hat{v} \mid v \rightsquigarrow^* \hat{v}\}$, then the rewriting algorithm terminates, assuming that every rule that remains applicable is eventually applied.*

Proof: Take $\bar{v} = \sqcup \{\hat{v} \mid v \rightsquigarrow^* \hat{v}\}$. By Proposition 5.2.15 if $v \in c(R) \bar{v}$, then $v \rightsquigarrow^* \bar{v}$. Since \rightsquigarrow is irreflexive, $\bar{v} \not\rightsquigarrow$ and \bar{v} is in normal form.

If we assume that each rule that is applicable is applied at one point (or rendered unapplicable by other rule applications), it is sufficient to know that there exists one rewriting sequence reaching \bar{v} from v . If we decide to apply a rule, that was applied in this specific sequence, at a later point of time, either we have already exceeded the corresponding vector in the original rewriting sequence via other rule applications, or the rule can still be applied with the same or a greater multiplicand, leading again to a larger vector.

It follows, that every sequence of rewriting steps will eventually reach \bar{v} . \square

Termination for Specific l -Monoids. We now study the l -monoid $\mathbb{M} = ([0, 1], \max, \cdot, 0, 1)$ from Example 2.2.20 and show that the rewriting algorithm terminates for this l -monoid. For the proof we mainly use the pigeonhole principle and exploit the total ordering of the underlying lattice. Since \mathbb{M} is isomorphic to \mathbb{T} , we obtain termination for the tropical semiring as a corollary.

Theorem 5.2.17 *The rewriting algorithm terminates for the l -monoids \mathbb{M} and \mathbb{T} .*

Proof: We show this via contradiction. So we assume the algorithm does not terminate, i.e. there exists an infinite sequence of rewriting steps starting from a vector v . If that is the case, observe that there are some indices such that the corresponding entries in the vector increase infinitely often. We can assume that from the beginning, there are only indices that increase infinitely often or do not increase at all, because otherwise, we can apply rules until this is true and use the resulting vector as the new starting vector. Equivalently, we can assume that each rule is applied infinitely often, by applying rules until no rule that can only be applied finitely often can ever get applied anymore and then removing all these rules from the rule system.

We call the initial vector v and the rule system \mathcal{R} . The sequence of intermediate rewriting results is a sequence v_0, v_1, \dots where $v_i \rightsquigarrow v_{i+1}$ for all $i \in \mathbb{N}_0$ in

a single rewriting step. Taking a look at the history of a specific component $v[j]$ of v , we can observe that in each rewriting step applying a rule $l \mapsto r$, $v[j]$ either does not change or is rewritten to $\frac{r[j] \cdot v[j']}{l[j']}$ for a vector-index j' . In fact, we choose the index j' which minimizes that quotient. Inductively, we obtain that at any given rewriting step i ,

$$v_i[j] = \frac{r_n[j] \cdot r_{n-1}[j_{n-1}] \cdot \dots \cdot r_1[j_1] \cdot v[j_0]}{l_n[j_{n-1}] \cdot l_{n-1}[j_{n-2}] \cdot \dots \cdot l_1[j_0]}$$

where j_0, \dots, j_n are vector-indices and $l_1 \mapsto r_1, \dots, l_n \mapsto r_n$ are rules. The maximum index of rules is not the same as i , because only those rules are multiplied that really contributed to the value of $v_i[j]$ directly, thus, n might be smaller than i . Note, that we used the fact that multiplication with 1 in order to apply a rule cannot happen, since then the rule could never be applied again (in this case we say that the rule has been applied maximally). If a rule was used maximally instead, it would not necessarily contribute a factor of the type $\frac{r[\bar{i}]}{l[\bar{j}]}$. Note, that this representation is unique and we say $v_i[j]$ is based on $v[j_0]$, if it can be written as above.

Let N be the dimension of v . At any given time i , there are at most N different entries in v_i and each entry in v_{i+1} is obtained by multiplying one factor of the form $\frac{r[\bar{i}]}{l[\bar{j}]}$, where $l \mapsto r$ is a rule and \bar{i}, \bar{j} are vector-indices, with one of the entries in v_i , or is identical to the entry in v_i . After at most $N \cdot (N - 1) + 1$ steps, there must exist one vector-index j , such that there exist indices $i \leq i' < j' \leq i + N \cdot (N - 1) + 1$ where $v_{i'}[j]$ and $v_{j'}[j]$ are not identical and based on the same entry $\ell \in [0, 1]$ from v_i , i.e. $v_{i'}[j]$ can be written as

$$v_{i'}[j] = \frac{r_k[j] \cdot r_{k-1}[i_{k-1}] \cdot \dots \cdot r_1[i_1]}{l_k[i_{k-1}] \cdot \dots \cdot l_2[i_1] \cdot l_1[i_0]} \cdot \ell$$

and $v_{j'}[j]$ as

$$v_{j'}[j] = \frac{r'_h[j] \cdot r'_{h-1}[j_{h-1}] \cdot \dots \cdot r'_1[j_1]}{l_h[j_{h-1}] \cdot \dots \cdot l'_2[j_1] \cdot l'_1[j_0]} \cdot \ell$$

where k and h are at most $N \cdot (N - 1)$.

This can be proven as follows: Each vector v_i has at most N different entries, so there are at most N different entries from v_i that an entry in a vector v_j , $j \geq i$, can be based on. In each step, at least one entry of v_i is rewritten to something larger. Each of the N entries of v_i can be rewritten and increased in the process at most $N - 1$ times, without being based on the same element

from v_i twice (including the initial entry of v_i), due to the pigeonhole principle. Again, since in each rewriting step at least one entry gets changed and there are only N entries, after $N \cdot (N - 1) + 1$ rewriting steps, there must be at least one entry that was based on the same entry from v_i twice and increased in-between.

Thus, we have

$$v_{j'}[j] = \frac{r'_h[j] \cdot r'_{h-1}[j_{h-1}] \cdot \dots \cdot r'_1[j_1]}{l_h[j_{h-1}] \cdot \dots \cdot l'_2[j_1] \cdot l'_1[j_0]} \cdot \frac{l_k[i_{k-1}] \cdot \dots \cdot l_2[i_1] \cdot l_1[i_0]}{r_k[j] \cdot r_{k-1}[i_{k-1}] \cdot \dots \cdot r_1[i_1]} \cdot v_{i'}[j]$$

which means, $v_{j'}[j]$ can be obtained from $v_{i'}[j]$ via multiplication of at most $N \cdot (N - 1) + 1$ factors of the form $\frac{r[i]}{l[j]}$ and at most $N \cdot (N - 1) + 1$ factors of the form $\frac{l[j]}{r[i]}$. Also, since $v_{j'}[j] > v_{i'}[j]$, this multiplicand is larger than 1. Observe that due to finiteness of \mathcal{R} , there are only finitely many products of at most $N \cdot (N - 1) + 1$ factors of the form $\frac{r[i]}{l[j]}$ and at most $N \cdot (N - 1) + 1$ factors of the form $\frac{l[j]}{r[i]}$, so there is a least one such factor $\delta > 1$. Moreover, this construction works for each interval of size $N \cdot (N - 1)$. Dividing the whole history of rule applications into consecutive chunks of size $N \cdot (N - 1) + 1$ yields infinitely many intervals where in each interval at least one index increases by at least factor δ . Since the dimension of v is finite, there must be at least one index j which has this property infinitely often. That means that $v[j]$ is rewritten to something larger than $\delta^n \cdot v[j]$ for each $n \in \mathbb{N}_0$. However, the sequence $\langle \delta^n \rangle$ is not bounded, therefore $\delta^n \cdot v[j]$ is not bounded by 1, but that is a contradiction to the assumption that the rewriting never terminates.

Using this result, we can now go on to show that rewriting terminates for the tropical semiring as well.

We show this by proving that the tropical semiring $(\mathbb{R}_0^+ \cup \{\infty\}, \min, +\infty, 0)$ and $([0, 1], \max, \cdot, 0, 1)$ are isomorphic with an isomorphism that is compatible with the order and the multiplication, which ensures that any given transformation system in one of those semirings can do a transformation step iff the transformation system obtained by applying the isomorphism to each component of every rule as well as the vector under consideration can do one.

We use the bijection $f: \mathbb{R}_0^+ \cup \{\infty\} \rightarrow [0, 1], f(x) = 2^{-x}$ where we have extended the power to $-\infty$ via the natural definition $2^{-\infty} = 0$. Obviously, this function is bijective with the inverse being $f^{-1}(x) = -\log_2(x)$, where the base-2 logarithm is extended to 0 via $\log_2(0) = -\infty$. Hence, we only have to

prove that f respects the order of the lattices and that it is compatible with addition and multiplication.

- Let $\ell_1, \ell_2 \in \mathbb{R}_0^+ \cup \{\infty\}$ be given, then

$$\ell_1 \geq \ell_2 \Leftrightarrow -\ell_1 \leq -\ell_2 \Leftrightarrow 2^{-\ell_1} \leq 2^{-\ell_2} \Leftrightarrow f(\ell_1) \leq f(\ell_2).$$

Note, that the order is swapped between the two semirings, so the function indeed is an order-isomorphism.

- Let $\ell_1, \ell_2 \in \mathbb{R}_0^+ \cup \{\infty\}$ be given, then

$$f(\min\{\ell_1, \ell_2\}) = 2^{-\min\{\ell_1, \ell_2\}} = \max\{2^{-\ell_1}, 2^{-\ell_2}\} = \max\{f(\ell_1), f(\ell_2)\}$$

- Let $\ell_1, \ell_2 \in \mathbb{R}_0^+ \cup \{\infty\}$ be given, then

$$f(\ell_1 + \ell_2) = 2^{-(\ell_1 + \ell_2)} = 2^{-\ell_1} \cdot 2^{-\ell_2} = f(\ell_1) \cdot f(\ell_2)$$

□

These results provide an effective procedure for checking congruence closure over the tropical semiring. We will mainly apply them to weighted automata, but expect that they can be useful to solve other problems. For instance, in Section 5.5, we show an interesting connection to the shortest path problem.

Termination for Lattices. We next turn to lattices and give a sufficient condition for termination on lattices. Obviously, rewriting terminates for lattices for which the ascending chain condition holds (i.e., every ascending chain eventually becomes stationary), but one can go beyond that.

In this section, we assume a completely distributive lattice \mathbb{L} and a Boolean algebra \mathbb{B} such that the orders of \mathbb{L} and \mathbb{B} , as well as the infima coincide when evaluated on elements from \mathbb{L} . Suprema need not coincide. Thus, whenever there is ambiguity, we will add the index \mathbb{B} or \mathbb{L} to the operator. For the negation of a given $x \in \mathbb{B}$, we write $\neg x$.

One way to obtain such a Boolean algebra – in particular, one where suprema coincide as well – is via Funayama’s theorem, see [BGJ13]. This embedding is also discussed in Subsection 2.2.4.

We want to show that if \mathbb{L} approximates \mathbb{B} “well enough”, the rewriting algorithm terminates for \mathbb{L} .

First, we need to show some properties concerning residuation and negation.

Lemma 5.2.18 *Let $\ell_1, \ell_2 \in \mathbb{L}$, where \mathbb{L} is a lattice, then*

- (i) $\ell_1 \rightarrow_{\mathbb{L}} \ell_2 \supseteq \ell_2$
- (ii) $\ell_1 \rightarrow_{\mathbb{L}} \ell_2 \supseteq \lfloor \neg \ell_1 \rfloor$
- (iii) $\ell_1 \rightarrow_{\mathbb{L}} \ell_2 = \lfloor \ell_1 \rightarrow_{\mathbb{B}} \ell_2 \rfloor$
- (iv) $\lfloor \neg \ell_1 \rfloor \sqcup_{\mathbb{B}} \ell_2 \subseteq \ell_1 \rightarrow_{\mathbb{L}} \ell_2 \subseteq \neg \ell_1 \sqcup_{\mathbb{B}} \ell_2$
- (v) $\ell_1 \rightarrow_{\mathbb{L}} \ell_2$ can be written as $\ell_1^* \sqcup_{\mathbb{B}} \ell_2$ for an $\lfloor \neg \ell_1 \rfloor \subseteq \ell_1^* \subseteq \neg \ell_1$.

Proof:

- (i) Every lattice is integral, hence $\ell_1 \cdot \ell_2 \subseteq \ell_2$. Hence, ℓ_2 is an element of the set $\{\ell \mid \ell_1 \cdot \ell \subseteq \ell_2\}$, the supremum of which is $\ell_1 \rightarrow_{\mathbb{L}} \ell_2$. Thus, $\ell_1 \rightarrow_{\mathbb{L}} \ell_2 \supseteq \ell_2$.
- (ii) Per definition, $\lfloor \neg \ell_1 \rfloor \cap \ell_1 \subseteq \neg \ell_1 \cap \ell_1 = \perp_{\mathbb{B}}$ and, if there were an element $\ell \supset \perp_{\mathbb{L}}$ such that $\lfloor \neg \ell_1 \rfloor \supseteq \ell$ and $\ell_1 \supseteq \ell$, then this would be true in \mathbb{B} , too, and therefore such an ℓ cannot exist. Thus, in particular, $\lfloor \neg \ell_1 \rfloor \cap \ell_1 = \perp_{\mathbb{L}} \subseteq \ell_2$ and per definition of $\ell_1 \rightarrow_{\mathbb{L}} \ell_2$, this proves that $\lfloor \neg \ell_1 \rfloor \subseteq \ell_1 \rightarrow_{\mathbb{L}} \ell_2$.
- (iii) We observe that

$$\lfloor \ell_1 \rightarrow_{\mathbb{B}} \ell_2 \rfloor = \bigsqcup_{\mathbb{L}} \{\ell \in \mathbb{L} \mid \ell \subseteq \ell_1 \rightarrow_{\mathbb{B}} \ell_2\}$$

and

$$\ell_1 \rightarrow_{\mathbb{L}} \ell_2 = \bigsqcup_{\mathbb{L}} \{\ell \in \mathbb{L} \mid \ell_1 \cap \ell \subseteq \ell_2\}$$

We will show that both sets are equal:

- \subseteq :

$$\ell \subseteq (\ell_1 \rightarrow_{\mathbb{B}} \ell_2) \Rightarrow \ell_1 \cap \ell \subseteq \ell_1 \cap (\ell_1 \rightarrow_{\mathbb{B}} \ell_2)$$

But then:

$$\ell_1 \cap \ell \subseteq \ell_1 \cap (\ell_1 \rightarrow_{\mathbb{B}} \ell_2) \subseteq \ell_2$$

And thus $\ell \in \{\ell' \in \mathbb{L} \mid \ell_1 \cap \ell' \subseteq \ell_2\}$.

- \supseteq :

$$\ell_1 \cap \ell \subseteq \ell_2 \Rightarrow \ell \in \{\ell' \in \mathbb{L} \mid \ell_1 \cap \ell' \subseteq \ell_2\} \subseteq \{\ell' \in \mathbb{B} \mid \ell_1 \cap \ell' \subseteq \ell_2\}$$

Hence, ℓ is smaller or equal than the supremum of this set.

$$(iv) \quad \lfloor \neg \ell_1 \rfloor \sqcup_{\mathbb{B}} \ell_2 \stackrel{\text{Lemma 5.2.18.(i)}}{\sqsubseteq} \stackrel{\text{Lemma 5.2.18.(ii)}}{=} \ell_1 \rightarrow_{\mathbb{L}} \ell_2 \stackrel{\text{Lemma 5.2.18.(iii)}}{=} \lfloor \ell_1 \rightarrow_{\mathbb{B}} \ell_2 \rfloor = \lfloor \neg \ell_1 \sqcup_{\mathbb{B}} \ell_2 \rfloor \sqsubseteq \neg \ell_1 \sqcup_{\mathbb{B}} \ell_2$$

- (v) In this proof we are exclusively computing in \mathbb{B} , so we do not point out that $\sqcup = \sqcup_{\mathbb{B}}$. We will define $\ell_1^* := ((\ell_1 \rightarrow_{\mathbb{L}} \ell_2) \sqcap \neg \ell_2) \sqcup \lfloor \neg \ell_1 \rfloor$ and first prove that $\ell_1 \rightarrow_{\mathbb{L}} \ell_2 = \ell_1^* \sqcup \ell_2$ and then prove that $\lfloor \neg \ell_1 \rfloor \sqsubseteq \ell_1^* \sqsubseteq \neg \ell_1$.

For the first part of the proof:

$$\begin{aligned} \ell_1 \rightarrow_{\mathbb{L}} \ell_2 &\stackrel{\text{Lemma 5.2.18.(iv)}}{=} \ell_1 \rightarrow_{\mathbb{L}} \ell_2 \sqcup \lfloor \neg \ell_1 \rfloor \sqcup \ell_2 \\ &= ((\ell_2 \sqcup \neg \ell_2) \sqcap (\ell_1 \rightarrow_{\mathbb{L}} \ell_2)) \sqcup \lfloor \neg \ell_1 \rfloor \sqcup \ell_2 \\ &= ((\ell_1 \rightarrow_{\mathbb{L}} \ell_2) \sqcap \neg \ell_2) \sqcup ((\ell_1 \rightarrow_{\mathbb{L}} \ell_2) \sqcap \ell_2) \sqcup \lfloor \neg \ell_1 \rfloor \sqcup \ell_2 \\ &= ((\ell_1 \rightarrow_{\mathbb{L}} \ell_2) \sqcap \neg \ell_2) \sqcup \lfloor \neg \ell_1 \rfloor \sqcup \ell_2 \\ &= \ell_1^* \sqcup \ell_2 \end{aligned}$$

Now, obviously, $\lfloor \neg \ell_1 \rfloor \sqsubseteq ((\ell_1 \rightarrow_{\mathbb{L}} \ell_2) \sqcap \neg \ell_2) \sqcup \lfloor \neg \ell_1 \rfloor = \ell_1^*$, since $\lfloor \neg \ell_1 \rfloor$ is part of the supremum. It is only left to be shown that $((\ell_1 \rightarrow_{\mathbb{L}} \ell_2) \sqcap \neg \ell_2) \sqcup \lfloor \neg \ell_1 \rfloor \sqsubseteq \neg \ell_1$ holds as well. First, we observe, that $\lfloor \neg \ell_1 \rfloor \sqsubseteq \neg \ell_1$ per definition, so it suffices to show that $(\ell_1 \rightarrow_{\mathbb{L}} \ell_2) \sqcap \neg \ell_2 \sqsubseteq \neg \ell_1$. Moreover, it is true that $\neg \ell_1 = \bigsqcup \{\ell \in \mathbb{B} \mid \ell \sqcap \ell_1 = \perp\}$. Then a simple computation shows:

$$(\ell_1 \rightarrow_{\mathbb{L}} \ell_2) \sqcap \neg \ell_2 \sqcap \ell_1 = ((\ell_1 \rightarrow_{\mathbb{L}} \ell_2) \sqcap \ell_1) \sqcap \neg \ell_2 \sqsubseteq \ell_2 \sqcap \neg \ell_2 = \perp.$$

Thus,

$$(\ell_1 \rightarrow_{\mathbb{L}} \ell_2) \sqcap \neg \ell_2 \in \{\ell \in \mathbb{B} \mid \ell \sqcap \ell_1 = \perp\}$$

of which $\neg \ell_1$ is the supremum.

□

Theorem 5.2.19 *The approximation of an element $\ell \in \mathbb{B}$ in the lattice \mathbb{L} is defined as $\lfloor \ell \rfloor = \bigsqcup_{\mathbb{L}} \{\ell' \in \mathbb{L} \mid \ell' \sqsubseteq \ell\}$.*

Let \mathcal{R} be a rewriting system for vectors in \mathbb{L}^X . Whenever the set $L(l, x) = \{\ell \in \mathbb{L} \mid \lfloor \neg l[x] \rfloor \sqsubseteq \ell \sqsubseteq \neg l[x]\}$ is finite for all rules $(l \mapsto r) \in \mathcal{R}$ and all $x \in X$, rewriting terminates.

Proof: Lemma 5.2.18.(v) shows that each multiplicand $l \rightarrow_{\mathbb{L}} v$ can be written as supremum of $v[x]$ for an index $x \in X$ and an element l^* from the finite set

$L(l, x)$. This set is independent of v , the element from the set must however be chosen according to v . Therefore, each element we obtain in rewriting is built as infimum and supremum of finitely many elements from \mathbb{B} and – using conjunctive normal form – we obtain that we can only build finitely many different rewriting results from v . Therefore, rewriting terminates for every vector v . \square

Note, that $[\neg \ell] = [\ell \rightarrow_{\mathbb{B}} 0] = \ell \rightarrow_{\mathbb{L}} 0$ (Lemma 5.2.18). Hence, the theorem says that there should be only finitely many elements between the negation of an element in the lattice and the negation of the same element in the Boolean algebra. As a simple corollary we obtain that the rewriting algorithm terminates for all Boolean algebras.

We now provide an interesting example of an infinite lattice that satisfies the conditions of Theorem 5.2.19.

It is known that a bounded distributive lattice \mathbb{L} that satisfies the infinite distributive law is isomorphic to the lattice of clopen downwards-closed sets in its dual (Priestley) space, which has the prime ideals of \mathbb{L} as elements, ordered by inclusion. More importantly in the sequel, the lattice of clopen downward-closed sets in a Priestley space always is a bounded distributive lattice which can be embedded (as per Funayama’s theorem) into the Boolean algebra of all clopen sets in the Priestley space, preserving infimum and supremum. For a more detailed survey of Priestley spaces and the duality we use, cf. [DP02].

Example 5.2.20 *We now define a lattice by first giving a Priestley space. We work in the space of all natural numbers and infinity \mathbb{N}_{∞} . Open sets are all sets O that do not contain ∞ or that do contain ∞ , but where the complement set $\mathbb{N}_{\infty} \setminus O$ is finite. Then, the clopen sets are all sets that are finite and do not contain ∞ or that are infinite and contain ∞ but where the complement is finite (i.e. the co-finite sets).*

If we define a suitable order \leq on the elements of \mathbb{N}_{∞} to obtain a Priestley space, the space of all clopen downsets in \mathbb{N}_{∞} is a distributive lattice. We claim that the condition from Theorem 5.2.19 holds whenever for each element $\ell \in \mathbb{N}_{\infty}$ there are only finitely many larger elements in \mathbb{N}_{∞} (according to \leq).

To prove this, we choose an arbitrary clopen downset O and prove that $\neg O$ in \mathbb{B} , the Boolean algebra of all clopen sets of \mathbb{N}_{∞} , only contains finitely many elements not contained in $O \rightarrow_{\mathbb{L}} 0$. Observe that $\neg O = \mathbb{N}_{\infty} \setminus O$. Either $\infty \in O$,

or $\infty \in \neg O$.

- If $\infty \in O$, then there are only finitely many elements in $\neg O$. Since $O \rightarrow_{\mathbb{L}} 0$ cannot be smaller than \emptyset and there are only finitely many sets between the empty set and a finite set, there are at most finitely many sets between $O \rightarrow_{\mathbb{L}} 0$ and $\neg O$, with respect to inclusion.
- If $\infty \notin O$, O is finite. The number of sets that are between $O \rightarrow_{\mathbb{L}} 0$ and $\neg O$ is certainly finite, if there are only finitely many elements in $\neg O \setminus O \rightarrow_{\mathbb{L}} 0$. Moreover, $O \rightarrow_{\mathbb{L}} 0$ can be characterised as follows:

$$\begin{aligned} O \rightarrow_{\mathbb{L}} 0 &= \{n \in \mathbb{N}_{\infty} \mid \neg \exists n' \in O : n \geq n' \vee n' \geq n\} \\ &= \{n \in \mathbb{N}_{\infty} \setminus O \mid \neg \exists n' \in O : n > n'\} \end{aligned}$$

In the first step, we used that infimum and supremum are intersection and union, in the second step we use the fact that O is downward closed. Hence, the difference between $\neg O$ and $O \rightarrow_{\mathbb{L}} 0$ contains exactly those elements that are larger than an element in O . Since O only contains finitely many elements and for each element there exist only finitely many elements that are greater, there are only finitely many elements in $\neg O \setminus (O \rightarrow_{\mathbb{L}} 0)$.

Example 5.2.21 Let $\mathbb{L} = (\overline{\mathbb{R}}, \sup, \inf, -\infty, \infty)$ (or any other completely ordered lattice), where $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, \infty\}$, i.e. the real numbers closed with ∞ and $-\infty$. There cannot be a Boolean algebra embedding for \mathbb{L} such that there are at most finitely many elements between $\neg \ell$ and $\ell \rightarrow -\infty$ for all $\ell \in \mathbb{R}$. Observe that $\ell \rightarrow -\infty = -\infty$ and that there are infinitely many elements ℓ_i larger than ℓ , therefore for each $\ell < \ell_i < \ell_j$ we have $-\infty < \neg \ell_j < \neg \ell_i < \neg \ell$, proving that such a Boolean algebra cannot exist. However, rewriting still terminates, because of the special form $\ell \rightarrow \ell'$ has for arbitrary ℓ, ℓ' . If $\ell > \ell'$, then $\ell \rightarrow \ell' = \ell'$, otherwise $\ell \rightarrow \ell' = \infty$. Therefore, we can again find a conjunctive normal form over a finite amount of lattice elements – the ones in the rules and the one in the initial vector – to describe each intermediate rewriting result. Therefore, starting from one given vector, only finitely many different vectors can be obtained by rewriting and the algorithm necessarily terminates.

A similar argument holds for all lattices that are completely ordered (where finite suprema and infima are maxima and minima) or are a finite product of

completely ordered lattices (where the argument needs to be applied component-wise). The rewriting algorithm terminates for all such lattices and, provided they are infinite in size, none of them can be embedded in a Boolean algebra meeting the condition of Theorem 5.2.19.

5.3 Up-To Techniques for Weighted Automata

In this section we present applications of our congruence closure method. More specifically, we investigate weighted automata and present up-to techniques both for the language equivalence and the inclusion problem, which are variants of the efficient up-to based algorithm presented in [BP13]. For the tropical semiring we also give a procedure for solving the threshold problem, based on the language inclusion algorithm.

5.3.1 Coinduction and Up-to Techniques

The soundness of the algorithms in Section 5.3 can be proven in a clear way by exploiting coinduction and up-to techniques. In this subsection we shortly recall the essential results of the theory developed in [PS11]. We fix the lattice of relations over \mathbb{S}^X , $Rel_{\mathbb{S}^X}$, but the results expressed here hold for arbitrary complete lattices.

Given a monotone map $b: Rel_{\mathbb{S}^X} \rightarrow Rel_{\mathbb{S}^X}$, the Knaster-Tarski fixed-point theorem characterises the greatest fixed-point νb as the union of all post-fixed points of b :

$$\nu b = \bigcup \{R \subseteq \mathbb{S}^X \times \mathbb{S}^X \mid R \subseteq b(R)\}.$$

This immediately leads to the *coinduction proof principle*

$$\frac{\exists R, S \subseteq R \subseteq b(R)}{S \subseteq \nu b}$$

This coinduction proof principle allows to prove that $(v_1, v_2) \in \nu b$ holds by exhibiting a post-fixed-point R such that $\{(v_1, v_2)\} \subseteq R$. We call the post-fixed-points of b , *b-simulations*. For a monotone map $f: Rel_{\mathbb{S}^X} \rightarrow Rel_{\mathbb{S}^X}$, a *b-simulation up-to f* is a relation R such that $R \subseteq b(f(R))$. We say that the mapping f is *compatible with b* if the inclusion $f(b(R)) \subseteq b(f(R))$ holds for all

relations R . The following result from [PS11] justifies our interest in compatible up-to techniques.

Theorem 5.3.1 *If f is b -compatible and $R \subseteq b(f(R))$ then $R \subseteq \nu b$.*

The above theorem leads to the coinduction up-to principle

$$\frac{\exists R, S \subseteq R \subseteq b(f(R))}{S \subseteq \nu b}$$

Up-to techniques can be combined in a number of interesting ways. For a map $f: Rel_{\mathbb{S}^X} \rightarrow Rel_{\mathbb{S}^X}$, the n -iteration of f is defined as $f^{n+1} = f \circ f^n$ and $f^0 = id$, the identity function. The omega iteration is defined as $f^\omega(R) = \bigcup_{i=0}^\infty f^i(R)$. Given two relations R and S , we use $R \bullet S$ to denote their relational composition $R \bullet S = \{(x, z) \mid \exists y : (x, y) \in R \text{ and } (y, z) \in S\}$. For two functions, $f, g: Rel_{\mathbb{S}^X} \rightarrow Rel_{\mathbb{S}^X}$, we write $f \bullet g$ for the function mapping a relation R into $f(R) \bullet g(R)$.

The following result from [PS11] informs us that compatible up-to techniques can be composed resulting in other compatible techniques. This is helpful in two ways: When proving compatibility of new up-to techniques we can separate concerns by splitting them up into simpler up-to techniques, which may come with easier proofs for compatibility. Additionally, it also allows us to combine techniques that were found independently of each other.

Lemma 5.3.2 *The following functions are b -compatible:*

- id : the identity function;
- $f \circ g$: the composition of b -compatible functions f and g ;
- $\bigcup F$: the pointwise union of an arbitrary family F of b -compatible functions: $\bigcup F(R) = \bigcup_{f \in F} f(R)$;
- f^ω : the (omega) iteration of a b -compatible function f , defined as $f^\omega(R) = \bigcup_{i=0}^\infty f^i(R)$

Moreover, if $b(R) \bullet b(S) \subseteq b(R \bullet S)$ for all relations R, S

- $f \bullet g$: the relation composition of b -compatible functions f and g ;

is b -compatible.

With these results it is easy to prove the soundness of the algorithms discussed in the sequel.

5.3.2 Language Equivalence for Weighted Automata

We turn our attention towards weighted automata and the analysis of their languages.

We will use a different notation for weighted automata in this chapter than introduced in Chapter 2, because we will often need to refer to the submatrix of an automaton α that corresponds to an a -transition for a label $a \in A$. Thus, in this chapter, a *weighted automaton* over the semiring \mathbb{S} and alphabet A is a triple (X, o, t) where X is a finite set of states, $t = (t_a: X \rightarrow \mathbb{S}^X)_{a \in A}$ is an A -indexed set of transition functions and $o: X \rightarrow \mathbb{S}$ is the output function. Intuitively, $t_a(x)(y) = s$ means that the state x can make a transition to the state y with letter $a \in A$ and weight $s \in \mathbb{S}$ (sometimes written as $x \xrightarrow{a,s} y$). The functions t_a can be represented as $X \times X$ -matrices with values in \mathbb{S} and o as a row vector of dimension X . Given a vector $v \in \mathbb{S}^X$, we use $t_a(v)$ to denote the vector obtained by multiplying the matrix t_a by v and furthermore we write $o(v)$ to denote the scalar in \mathbb{S} obtained by multiplying the row vector o by the column vector v .

It is easy to see that this alternative way of writing weighted automata (X, o, t) is equivalent to the one previously introduced in Definition 2.3.4, where a weighted automaton over the state set X was defined to be an $X \times (A \times X + 1)$ -matrix α :

- o is the last column in α
- for each $a \in A$, the matrix t_a can be obtained by collecting the columns indexed by (x, a) , where $x \in X$.

This correspondence can serve as a translation in both ways.

A *weighted language* is a function $\varphi: A^* \rightarrow \mathbb{S}$, where A^* is the set of all words over A . We will use ε to denote the empty word and aw the concatenation of a letter $a \in A$ with the word $w \in A^*$. Every weighted automaton is associated with a function $\llbracket - \rrbracket: \mathbb{S}^X \rightarrow \mathbb{S}^{A^*}$ mapping each vector into its *accepted language*. For all $v \in \mathbb{S}^X$, $a \in A$ and $w \in A^*$, this is defined as

$$\llbracket v \rrbracket(\varepsilon) = o(v) \quad \llbracket v \rrbracket(aw) = \llbracket t_a(v) \rrbracket(w).$$

Two vectors $v_1, v_2 \in \mathbb{S}^X$ are called *language equivalent*, written $v_1 \sim v_2$, iff $\llbracket v_1 \rrbracket = \llbracket v_2 \rrbracket$.²

Recall that the problem of checking language equivalence in weighted automata for an arbitrary semiring is undecidable: for the tropical semiring this was shown by Krob in [Kro94]; the proof was later simplified in [ABK11]. However, for several semirings the problem is decidable, for instance for all (complete and distributive) lattices. For finite non-deterministic automata, i.e., weighted automata where the weights are taken from the binary Boolean semiring, Bonchi and Pous introduced in [BP13] the algorithm **HKC**. The name stems from the fact that the algorithm extends the procedure of Hopcroft and Karp [HK71] with congruence closure.

Algorithm 5.3.3 shows the extension of **HKC** to weighted automata over an arbitrary semiring: the code is the same as the one in [BP13], apart from the fact that, rather than exploring sets of states, the algorithm works with vectors in \mathbb{S}^X . The check at step 5 can be performed with the procedures discussed in Section 5.2.

²The notions of accepted language and language equivalence can be given for states rather than for vectors by embedding states $x \in X$ in the unit vectors $e_x \in \mathbb{S}^X$, to retrieve exactly the notion of language equivalence we have used so far, i.e. $\llbracket e_x \rrbracket(w) = L(w)(x)$. On the other hand, when weighted automata are given with an initial vector i – which is often the case in literature – one can define the language of an automaton just as $\llbracket i \rrbracket$.

Algorithm 5.3.3 Algorithm to check the equivalence of vectors $v_1, v_2 \in \mathbb{S}^X$ for a weighted automata (X, o, t) .

- **Input:** A weighted automaton $\alpha = (X, o, t)$ over a semiring \mathbb{S} and two initial vectors $v_1, v_2 \in \mathbb{S}^X$
- **Output:** True, if v_1 and v_2 are language equivalent, false otherwise

HKC (α, v_1, v_2)

```

1  $R := \emptyset$ ;  $todo := \emptyset$ 
2 insert  $(v_1, v_2)$  into  $todo$ 
3 while  $todo \neq \emptyset$  do
4   extract  $(v'_1, v'_2)$  from  $todo$ 
5   if  $(v'_1, v'_2) \in c(R)$  then continue
6   if  $o(v'_1) \neq o(v'_2)$  then return false
7   for all  $a \in A$ ,
8     insert  $(t_a(v'_1), t_a(v'_2))$  into  $todo$ 
9   insert  $(v'_1, v'_2)$  into  $R$ 
10 return true

```

Below we prove that the algorithm is sound and complete, but termination can fail in two ways: either the check at step 5 does not terminate, or the while loop at step 3 does not. For the tropical semiring we have seen that the check at step 5 can be effectively performed by rewriting (Theorem 5.2.17). Therefore, due to Krob's undecidability result, the while loop at step 3 may not terminate.

For (distributive) lattices, we have shown termination of rewriting under some additional constraints (Theorem 5.2.19); moreover the loop at step 3 will always terminate, because from a given finite set of lattice elements only finitely many lattice elements can be constructed using infimum and supremum Chapter 4.

To prove soundness of HKC, we introduce the notions of simulation and bisimulation up-to. Let $Rel_{\mathbb{S}^X}$ be the complete lattice of relations over \mathbb{S}^X and $b_1: Rel_{\mathbb{S}^X} \rightarrow Rel_{\mathbb{S}^X}$ be the monotone map defined for all $R \subseteq \mathbb{S}^X \times \mathbb{S}^X$ according to

$$b_1(R) = \{(v_1, v_2) \mid o(v_1) = o(v_2) \text{ and for all } a \in A, (t_a(v_1), t_a(v_2)) \in R\}$$

Definition 5.3.4 A relation $R \subseteq \mathbb{S}^X \times \mathbb{S}^X$ is a b_1 -simulation if $R \subseteq b_1(R)$, i.e., for all $(v_1, v_2) \in R$:

1. $o(v_1) = o(v_2)$
2. for all $a \in A$, $(t_a(v_1), t_a(v_2)) \in R$.

For a monotone map $f: Rel_{\mathbb{S}^X} \rightarrow Rel_{\mathbb{S}^X}$, a b_1 -simulation up-to f is a relation R such that $R \subseteq b_1(f(R))$.

It is easy to show (see e.g. [PS11]) that b_1 -simulation provides a sound and complete proof technique for \sim . On the other hand, not all functions f can be used as sound up-to techniques. The algorithm HKC exploits the monotone function $c: Rel_{\mathbb{S}^X} \rightarrow Rel_{\mathbb{S}^X}$ mapping each relation R to its congruence closure $c(R)$.

Proposition 5.3.5 Let $v_1, v_2 \in \mathbb{S}^X$. It holds that $v_1 \sim v_2$ iff there exists a b_1 -simulation R such that $(v_1, v_2) \in R$ iff there exists a b_1 -simulation up-to c R such that $(v_1, v_2) \in R$.

Proof: The key step in proving this claim consists in characterising \sim as νb_1 . This can easily be deduced from abstract results (see e.g. [BBB⁺12, HJ98]), but to keep the argument self-contained, a concrete proof would proceed as follows:

1. We will first prove that b_1 is a co-continuous function, i.e., for any index set I it holds that $b_1(\bigcap_I S_i) = \bigcap_I b_1(S_i)$ whenever $S_{i+1} \subseteq S_i$ holds for all $i \in I$:

Let any descending chain of relations S_1, S_2, \dots , where $S_i \supseteq S_{i+1}$ for all $i \in I$, be given.

- Let $(v_1, v_2) \in b_1(\bigcap_{i \in I} S_i)$ be given, then note that for all $i \in I$ it holds that $S_i \supseteq \bigcap_{i \in I} S_i$, and b_1 is obviously monotone. Therefore, it follows that $(v_1, v_2) \in b_1(S_i)$ for all $i \in I$ and thus $(v_1, v_2) \in \bigcap_{i \in I} b_1(S_i)$.
- Let $(v_1, v_2) \in \bigcap_{i \in I} b_1(S_i)$ be given, then $o(v_1) = o(v_2)$ and for all $a \in A$, all $i \in I$, it holds that $(t_a(v_1), t_a(v_2)) \in S_i$. Thus, it follows for all $a \in A$, that $(t_a(v_1), t_a(v_2)) \in \bigcap_{i \in I} S_i$. Therefore, $(v_1, v_2) \in b_1(\bigcap_{i \in I} S_i)$.

2. consequently, by the Kleene fixed-point theorem [DP02], it follows that $\nu b_1 = \bigcap_n b_1^n(\top)$ holds, where $b_1^0 = Id$ and $b_1^{n+1} = b_1 \circ b_1^n$;
3. prove, using induction, that for all n , $b_1^n(\top) = \{(v_1, v_2) \mid \llbracket v_1 \rrbracket(w) = \llbracket v_2 \rrbracket(w) \text{ for all words } w \in A^* \text{ up to length } n-1\}$:
 For $n = 0$, nothing is to prove, because the condition $\llbracket v_1 \rrbracket(w) = \llbracket v_2 \rrbracket(w)$ for all words $w \in A^*$ up to length -1 evaluates to true. For $n = 1$ this follows from the condition $o(v_1) = o(v_2)$. Now assume the claim holds for all $i \leq n$. Let $(v_1, v_2) \in b_1^{n+1}(\top)$, then for all $a \in A$ it must hold that $(t_a(v_1), t_a(v_2)) \in b_1^n(\top)$. The induction hypothesis then yields that $\llbracket t_a(v_1) \rrbracket(w) = \llbracket t_a(v_2) \rrbracket(w)$ for all words $w \in A^*$ up to length $n-2$ and thus $\llbracket v_1 \rrbracket(w) = \llbracket v_2 \rrbracket(w)$ for all words $w \in A^*$ up to length $n-1$.
4. conclude by 2 and 3 that $\nu b_1 = \{(v_1, v_2) \mid \llbracket v_1 \rrbracket = \llbracket v_2 \rrbracket \text{ for all words } w \in A^*\}$.

By coinduction, the first statement follows.

For the second statement we have to use coinduction up-to and prove b_1 -compatibility of c . The latter follows from abstract results [BPPR14b]. For a concrete proof, one first has to show that the following monotone maps are b_1 -compatible.

- *the constant reflexive function*: $r(R) = \{(x, x) \mid x \in S\}$: This is trivially true, in fact, even $r(b_1(R)) = b_1(r(R))$ holds.
- *the converse function*: $s(R) = \{(y, x) \mid (x, y) \in R\}$: Due to the symmetric definition of b_1 , it is easy to see that reversing the order in the pairs before or after applying b_1 does not make a difference.
- *the squaring function*: $t(R) = \{(x, z) \mid \exists y, (x, y) \in R \text{ and } (y, z) \in R\}$: We first compute

$$r(b_1(R)) = \{(x, z) \mid (x, y) \in b_1(R), (y, z) \in b_1(R)\},$$

$$b_1(r(R)) = b_1(\{(x, z) \mid (x, y) \in R, (y, z) \in R\}).$$

Now assume any x, z are given, such that $(x, z) \in b_1(r(R))$. By definition, there exists a y , such that $(x, y) \in R$ and $(y, z) \in R$ – i.e. $(x, z) \in \{(x', z') \mid (x', y') \in R, (y', z') \in R\}$. Then $o(x) = o(y)$ and $o(y) =$

$o(z)$, therefore, by transitivity of equality, $o(x) = o(z)$. Moreover, for $(x, y) \in b_1(R)$, it must be true that $t_a(x), t_a(y) \in R$ for all $a \in A$ and analogously $(t_a(y), t_a(z)) \in R$ can be shown to hold. Therefore, $(t_a(x), t_a(z)) \in \{(x', z') \mid (x', y') \in R, (y', z') \in R\}$, which concludes the proof that $(x, z) \in b_1(r(R))$.

- *the sum function:* $+(R) = \{(v_1 + v_2, v'_1 + v'_2) \mid (v_1, v_2) \in R \text{ and } (v'_1, v'_2) \in R\}$: We compute

$$+(b_1(R)) = \{(v_1 + v_2, v'_1 + v'_2) \mid (v_1, v'_1) \in b_1(R), (v_2, v'_2) \in b_1(R)\},$$

$$b_1(+(R)) = b_1(\{(v_1 + v_2, v'_1 + v'_2) \mid (v_1, v'_1) \in R, (v_2, v'_2) \in R\}).$$

Now assume $(v_1 + v_2, v'_1 + v'_2) \in +(b_1(R))$ is given, then $(v_1, v'_1) \in b_1(R) \subseteq R$, $(v_2, v'_2) \in b_1(R) \subseteq R$. Thus, $o(v_1) = o(v'_1)$ and $o(v_2) = o(v'_2)$, i.e. $o(v_1) + o(v_2) = o(v'_1) + o(v'_2)$. Moreover, since $(v_1, v'_1) \in b_1(R)$, it must hold for all $a \in A$ that $(t_a(v_1), t_a(v'_1)) \in R$ and analogously it can be shown that $(t_a(v_2), t_a(v'_2)) \in R$. Thus also $(t_a(v_1) + t_a(v_2), t_a(v'_1) + t_a(v'_2)) \in \{(\overline{v_1} + \overline{v_2}, \overline{v'_1} + \overline{v'_2}) \mid (\overline{v_1}, \overline{v'_1}) \in b_1(R), (\overline{v_2}, \overline{v'_2}) \in b_1(R)\}$, showing that $(v_1 + v_2, v'_1 + v'_2) \in b_1(+(R))$.

- *the scalar function:* $\cdot(R) = \{(v \cdot s, w \cdot s) \mid (v, w) \in R \text{ and } s \in \mathbb{S}\}$: We first compute

$$\cdot(b_1(R)) = \{(v \cdot s, w \cdot s) \mid (v, w) \in b_1(R), s \in \mathbb{S}\},$$

$$\begin{aligned} b_1(\cdot(R)) &= b_1(\{(v, w) \mid (v, w) \in \cdot(R)\}) \\ &= b_1(\{(v \cdot s, w \cdot s) \mid (v, w) \in R, s \in \mathbb{S}\}). \end{aligned}$$

Now assume any pair of vectors $(v \cdot s, w \cdot s) \in \cdot(b_1(R))$ is given. Then we can find

$$o(v) = o(w) \Rightarrow o(v \cdot s) = o(v) \cdot s = o(w) \cdot s = o(w \cdot s)$$

due to linearity of matrix multiplication. Moreover, since $(v, w) \in b_1(R)$ is true for all $a \in A$, it must hold that $(t_a(v), t_a(w)) \in R$, i.e.

$$(t_a(v) \cdot s, t_a(w) \cdot s) \in \{(v \cdot s, w \cdot s) \mid (v, w) \in R, s \in \mathbb{S}\},$$

concluding the proof that $(v \cdot s, w \cdot s) \in b_1(\cdot(R))$.

Also note, that $\text{ld}(R) \subseteq \cdot(R)$ using $s = 1$. Then, one can observe that $c = (r \cup s \cup t \cup + \cup \cdot)^\omega$ and conclude that c is b -compatible by Lemma 5.3.2. \square

With this result, it is easy to prove the correctness of the algorithm.

Theorem 5.3.6 *Whenever HKC terminates, it returns true iff $\llbracket v_1 \rrbracket = \llbracket v_2 \rrbracket$.*

Proof: Observe that $R \subseteq b_1(c(R) \cup \text{todo})$ is an invariant for the while loop at step 3.

If HKC returns *true* then *todo* is empty and thus $R \subseteq b_1(c(R))$, i.e., R is a b_1 -simulation up-to c . By Proposition 5.3.5, $v_1 \sim v_2$.

Whenever HKC returns false, it encounters a pair $(v'_1, v'_2) \in \text{todo}$ such that $o(v'_1) \neq o(v'_2)$. Observe that for all pairs $(v'_1, v'_2) \in \text{todo}$, there exists a word $w = a_1 a_2 \dots a_n \in A^*$ such that $v'_1 = t_{a_n}(\dots t_{a_2}(t_{a_1}(v_1)))$ and $v'_2 = t_{a_n}(\dots t_{a_2}(t_{a_1}(v_2)))$. Therefore $\llbracket v_1 \rrbracket(w) = \llbracket v'_1 \rrbracket(\varepsilon) = o(v'_1) \neq o(v'_2) = \llbracket v'_2 \rrbracket(\varepsilon) = \llbracket v_2 \rrbracket(w)$. \square

5.3.3 Language Inclusion

Whenever a semiring \mathbb{S} carries a partial order \sqsubseteq , one can be interested in checking language inclusion of the states of a weighted automata (X, o, t) . More generally, given $v_1, v_2 \in \mathbb{S}^X$, we say that the language of v_1 is included in the language of v_2 (written $v_1 \preceq v_2$), iff $\llbracket v_1 \rrbracket(w) \sqsubseteq \llbracket v_2 \rrbracket(w)$ for all $w \in A^*$.

The algorithm HKC can be slightly modified to check language inclusion, resulting in algorithm HKP: The only change when compared to HKC are the checks in steps 5 and 6.

For this algorithm, we use a function $p: \text{Rel}_{\mathbb{S}^X} \rightarrow \text{Rel}_{\mathbb{S}^X}$, which is the monotone function assigning to each relation R its precongruence closure $p(R)$.

The precongruence closure is defined as the closure of R under \sqsubseteq , transitivity and linear combination. That is, in the rules of Table 5.1 $c(R)$ is replaced by $p(R)$, rule (SYM) is removed and rule (REFL) is replaced by rule (ORD) on the right.

$$\text{(ORD)} \quad \frac{v \sqsubseteq v'}{v \ p(R) \ v'}$$

Algorithm 5.3.7 Algorithm to check the language inclusion of vectors $v_1, v_2 \in \mathbb{S}^X$ for a weighted automata (X, o, t) .

- **Input:** A weighted automaton $\alpha = (X, o, t)$ over a semiring \mathbb{S} and two initial vectors $v_1, v_2 \in \mathbb{S}^X$
- **Output:** True, if the language of v_1 is a sublanguage of the language of v_2 , false otherwise

HKP (α, v_1, v_2)

```

1  $R := \emptyset; \text{ todo} := \emptyset$ 
2 insert  $(v_1, v_2)$  into  $\text{todo}$ 
3 while  $\text{todo} \neq \emptyset$  do
4   extract  $(v'_1, v'_2)$  from  $\text{todo}$ 
5   if  $(v'_1, v'_2) \in p(R)$  then continue
6   if  $o(v'_1) \not\sqsubseteq o(v'_2)$  then return false
7   for all  $a \in A$ ,
8     insert  $(t_a(v'_1), t_a(v'_2))$  into  $\text{todo}$ 
9   insert  $(v'_1, v'_2)$  into  $R$ 
10 return true

```

The soundness of the modified algorithm can be proven in the same way as for HKC by replacing c by p and b_1 by b_2 : $Rel_{\mathbb{S}^X} \rightarrow Rel_{\mathbb{S}^X}$ defined for all $R \subseteq \mathbb{S}^X \times \mathbb{S}^X$ as

$$b_2(R) = \{(v_1, v_2) \mid o(v_1) \sqsubseteq o(v_2) \text{ and for all } a \in A, (t_a(v_1), t_a(v_2)) \in R\}.$$

However, the soundness of up-to reasoning is guaranteed only if \sqsubseteq is a precongruence, that is $p(\sqsubseteq)$ is contained in \sqsubseteq .

Lemma 5.3.8 $\nu b_2 = \sqsubseteq$.

Proof: The proof proceeds as for the first part of Proposition 5.3.5 by using \sqsubseteq in place of $=$ and b_2 in place of b_1 . \square

Lemma 5.3.9 If \sqsubseteq is a precongruence, the following monotone maps are b_2 -compatible:

- the constant ord function: $\sqsubseteq (R) = \{(v_1, v_2) \mid v_1 \sqsubseteq v_2\}$,
- the constant inclusion function: $\sqsubset (R) = \{(v_1, v_2) \mid v_1 \sqsubset v_2\}$,
- the squaring function: $t(R) = \{(v_1, v_3) \mid \exists v_2, (v_1, v_2) \in R \text{ and } (v_2, v_3) \in R\}$,
- the sum function: $+(R) = \{(v_1 + v_2, v'_1 + v'_2) \mid (v_1, v_2) \in R \text{ and } (v'_1, v'_2) \in R\}$,
- the scalar function: $\cdot (R) = \{(v \cdot s, w \cdot s) \mid (v, w) \in R \text{ and } s \in \mathbb{S}\}$.

Proof: Since \sqsubseteq is a precongruence, if $v_1 \sqsubseteq v_2$, then $o(v_1) \sqsubseteq o(v_2)$ and for all $a \in A$, $t_a(v_1) \sqsubseteq t_a(v_2)$. Which means that $\sqsubseteq \subseteq b_2(\sqsubseteq)$, that is, for all relations R , $\sqsubseteq (b_2(R)) \subseteq b_2(\sqsubseteq (R))$. This proves the first statement. The others are similar to the proof of b_1 -compatibility before. \square

Lemma 5.3.10 *p is compatible.*

Proof: Observe that by definition $p = (Id \cup \sqsubseteq \cup t \cup + \cup \cdot)^\omega$. The statement follows immediately by Lemma 5.3.2 and Lemma 5.3.9. \square

Note that we have not made use of the constant inclusion being b_2 -compatible here. However, it is useful, because it informs us that we use information about pairs of vectors, that are language included in each other. Therefore, we can initialise the relation R in our algorithm with any pairs of vectors, that are language included, known e.g. from a pre-computation.

Theorem 5.3.11 *Let \mathbb{S} be a semiring equipped with a precongruence \sqsubseteq . Whenever $\text{HKP}(v_1, v_2)$ terminates, it returns true if and only if $v_1 \sqsubset v_2$.*

Proof: For soundness, observe that the following is an invariant for the while loop at step 3.

$$R \subseteq b_2(p(R) \cup \text{todo})$$

If HKP returns *true* then *todo* is empty and thus $R \subseteq b_2(p(R))$, i.e., R is a b_2 -simulation up-to p . By Lemma 5.3.10, Theorem 5.3.1 and Lemma 5.3.8, $v_1 \sqsubset v_2$.

For completeness, we proceed in the same way as in Theorem 5.3.6. \square

In order for HKP to be effective, we need a procedure to compute p . When \mathbb{S} is an integral l -monoid, we can check $(v, v') \in p(R)$ via a variation of the congruence check, using a rewriting system as in Subsection 5.2.4.

Proposition 5.3.12 *Let \mathbb{L} be an integral l -monoid and let $R \subseteq \mathbb{L}^X \times \mathbb{L}^X$ be a relation.*

The set of rules \mathcal{R} is defined as $\{v' \mapsto v \sqcup v' \mid (v, v') \in R\}$.³ Rewriting steps are defined as in Definition 5.2.6. If the rewriting algorithm terminates, then for all $v, v' \in \mathbb{L}^X$, $(v, v') \in p(R)$ iff $\Downarrow v' \sqsupseteq v$ (where, as usual, $\Downarrow v'$ denotes the normal form of v').

Proof: This proof is very close in structure to the proofs for Theorem 5.2.11 and Theorem 5.2.14. We will use Lemma 5.2.12 and Lemma 5.2.13, because the claims and proofs for these lemmas can be copied verbatim for the asymmetric case, so we do not prove these claims again. This does not hold true for Theorem 5.2.11 and Theorem 5.2.14, though, where symmetry is relevant. We will now prove the two claims, adjusted to the non-symmetric case.

- *Whenever there exists a vector $v'_2 \sqsupseteq v_1$ such that v_2 rewrites to v'_2 via \mathcal{R} , i.e., $v_2 \rightsquigarrow_{\mathcal{R}}^* v'_2$, then $(v_1, v_2) \in p(R)$.* This is the analogue to Theorem 5.2.11.

We will show that if $v_2 \rightsquigarrow_{\mathcal{R}}^* v'_2$, then $(v'_2, v_2) \in p(R)$. Furthermore, $v_1 \sqsubseteq v'_2$ implies $v_1 p(R) v'_2$ due to rule (ORD). Transitivity then yields $(v_1, v_2) \in p(R)$.

Assume $v_2 \rightsquigarrow v'$ via a rule $l \mapsto r$, then $l = w'$, $r = w \sqcup w'$ where $(w, w') \in R$, according to definition of \mathcal{R} . Due to closure under linear combinations and idempotency of \sqcup , we have $w \sqcup w' p(R) w' \sqcup w' = w'$, i.e. $r p(R) l$. Therefore, due to closure under scalar multiplication, $r \cdot (l \rightarrow v_2) p(R) l \cdot (l \rightarrow v_2)$, due to closure under linear combination $v_2 \sqcup r \cdot (l \rightarrow v_2) p(R) v_2 \sqcup l \cdot (l \rightarrow v_2)$. Keeping in mind the definition of \rightarrow , we can observe that $l \cdot (l \rightarrow v_2) \sqsubseteq v_2$. Applying this and taking idempotency of the supremum into account again, we obtain $v_2 \sqcup l \cdot (l \rightarrow v_2) = v_2$, therefore $v' p(R) v_2$. Transitivity yields the claim for \rightsquigarrow^* .

³Whenever $v \leq v'$, the rule can be omitted, since it is never applicable.

- If $v \ p(R) \ v'$, then $\Downarrow v' \sqsupseteq v$. This is the analogue to Theorem 5.2.14. We perform a proof by structural induction on the modified derivation rules of Table 5.1 (where (SYM) is removed and (REFL) is replaced by rule (ORD)).

(Rel) If $v \ p(R) \ v'$ because vRv' then there exists a rewriting rule $(v' \mapsto v \sqcup v') \in \mathcal{R}$. Applying this rule shows that $\Downarrow v' \sqsupseteq v' \sqcup v \sqsupseteq v$.

(Ord) If $v \ p(R) \ v'$ because of the ordering rule, i.e. $v \sqsubseteq v'$, then $\Downarrow v' \sqsupseteq v' \sqsupseteq v$.

(Trans) If $v_1 \ p(R) \ v_3$ because of $v_1 \ p(R) \ v_2$ and $v_2 \ p(R) \ v_3$, then $\Downarrow v_3 \sqsupseteq v_2$ implies $\Downarrow v_3 \sqsupseteq \Downarrow v_2$. Furthermore $v_1 \ p(R) \ v_2$ inductively yields $\Downarrow v_2 \sqsupseteq v_1$ and transitivity therefore yields $\Downarrow v_3 \sqsupseteq v_1$.

(Sca) If $v \cdot \ell \ p(R) \ v' \cdot \ell$ because $v \ p(R) \ v'$, then $v \sqsubseteq \Downarrow v'$ and Lemma 5.2.13 yields $\Downarrow (v' \cdot \ell) \sqsupseteq (\Downarrow v') \cdot \ell \sqsupseteq v \cdot \ell$.

(Plus) If $\bar{v} \sqcup v \ p(R) \ \bar{v}' \sqcup v'$ because $\bar{v} \ p(R) \ \bar{v}'$ and $v \ p(R) \ v'$, then $v \sqcup \bar{v} \sqsubseteq (\Downarrow v') \sqcup (\Downarrow \bar{v}') \sqsubseteq \Downarrow (v' \sqcup \bar{v}')$, due to the monotonicity of \Downarrow .

□

Observe that Theorems 5.2.17 and 5.2.19 guarantee termination for certain specific l -monoids. In particular, termination for the tropical semiring will be pivotal henceforward.

5.3.4 Threshold Problem for Automata over the Tropical Semiring

Language inclusion for weighted automata over the tropical semiring is not decidable, because language equivalence is not: mutual language inclusion is the same as language equivalence.

However, the algorithm HKP that we have introduced in the previous section can be used to solve the so-called *threshold problem* over the tropical semiring of natural numbers $(\mathbb{N}_0 \cup \{\infty\}, \min, +, \infty, 0)$.

The problem is to check whether for a given threshold $T \in \mathbb{N}_0$, a vector of states of a weighted automaton $v \in (\mathbb{N}_0 \cup \{\infty\})^X$ satisfies the threshold T , i.e. $\llbracket v \rrbracket(w) \leq T$ for all $w \in A^*$.

Note, that this problem is also known as the *universality problem*: universality for non-deterministic automata can easily be reduced to a threshold check, by taking weight 0 for each transition of the automaton and setting $T = 0$ for the threshold.

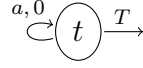


Figure 5.1: Added state for the threshold check

This problem – which is known to be **PSPACE**-complete [ABK11] – can be reduced to language inclusion by adding a new state t to the automaton, which has $o(t) = T$ as output and a 0 self-loop for each letter $a \in A$. This state then assigns to all words in A^* the weight T . Then we check whether the language of v includes the language of the unit vector e_t . If language inclusion does not hold, there must exist at least one word where the weight of v must be smaller than the weight of e_t , which is exactly T .

It is worth to note, that in $(\mathbb{N}_0 \cup \{\infty\}, \min, +, \infty, 0)$ the ordering \sqsubseteq is actually \geq , i.e. the reversed ordering on natural numbers. Therefore, in order to solve the threshold problem, we need to check whether the inclusion $e_t \preceq v$ holds, rather than the converse, which may appear to be the more natural choice at first sight.

Abstraction is not required for termination The algorithm HKP, when applied to an instance of the threshold problem will always terminate, without any additional requirements. In the original work [BKK17], we claimed that an abstraction is required to guarantee termination, but the rule ORD already guarantees termination without the need of an abstraction, using the following argument:

To show that the algorithm terminates, we observe that the following claim is a direct consequence of Dickson’s Lemma:

Every set of n -dimensional vectors over the tropical semiring that contains no pair of (different) vectors v, v' such that $v \sqsubseteq v'$, i.e. $v \geq v'$, is finite.

If the reader is unaware of Dickson’s Lemma, a direct proof of this claim may be found in Appendix B.2. Observe that for the threshold problem, the algorithm

HKP generates vector pairs where the right side always stays the same, so we can characterise a run of HKP as collecting vectors from a set of n -dimensional vectors, where no pair of different vectors v, v' with $v \geq v'$ exists – all other vectors are eliminated by the rule ORD. The constant right side is ignored in this view, instead we will solely focus on a projection to the first element of each pair.

This result can now be used to show termination, taking into account that the algorithm creates a sequence of such sets of vectors, that is strictly increasing and prunes them by removing all non-minimal vectors. So for each vector in iteration i it holds that this vector is also minimal in iteration $i + 1$, or replaced by a new, smaller vector. Each vector can only be replaced by a smaller vector finitely often. For entries which are natural numbers, this is obvious, for infinite entries, note, that after replacing them once by a smaller entry, the new entry also is a natural number. Now, we can take the union of all the sets of vectors obtained in the algorithm, pruned again to only include the minimal vectors and observe that there are only finitely many such vectors. This finite set must be reached after finitely many steps in the algorithm, because in each step, a change in the minimal vectors must occur, otherwise the algorithm terminates. Since only finitely many changes can occur, this is limiting the number of steps that the algorithm can take.

An abstraction to improve the run-time This reasoning certainly does not yield a very promising argument for termination speed of HKP when used to decide the threshold problem. However, it has already been observed in [ABK11] that it is a sound reasoning technique to replace every vector entry larger than the threshold T by ∞ . In order to reduce the number of vectors that need to be explored, it is sensible to apply this reasoning technique, even though it is not strictly required to guarantee termination. To formalise this result, we will first introduce an abstraction mapping \mathcal{A} and then state our modified algorithm:

Definition 5.3.13 *Let a threshold $T \in \mathbb{N}_0$ be given. We define the abstraction $\mathcal{A}: \mathbb{N}_0 \cup \{\infty\} \rightarrow \mathbb{N}_0 \cup \{\infty\}$ according to $\mathcal{A}(s) = s$ if $s \leq T$ and $\mathcal{A}(s) = \infty$ otherwise. The definition of the abstraction \mathcal{A} extends elementwise to vectors in $(\mathbb{N}_0 \cup \{\infty\})^X$.*

This abstraction can be incorporated into the algorithm **HKP**. Taking the abstraction into account, we call the algorithm obtained from **HKP** by changing steps 7 and 8 by applying the abstraction appropriately **HKP_A**. The algorithm **HKP_A** is given as follows:

Algorithm 5.3.14 Algorithm to check the language inclusion of vectors $v_1, v_2 \in \mathbb{S}^X$ for a weighted automata (X, o, t) .

- **Input:** A weighted automaton $\alpha = (X, o, t)$ over a semiring \mathbb{S} and two initial vectors $v_1, v_2 \in \mathbb{S}^X$
- **Output:** True, if the language of v_1 is a sublanguage of the language of v_2 , false otherwise

HKP_A(α, v_1, v_2)

```

1  $R := \emptyset; \text{ todo} := \emptyset$ 
2 insert  $(v_1, v_2)$  into todo
3 while todo  $\neq \emptyset$  do
4   extract  $(v'_1, v'_2)$  from todo
5   if  $(v'_1, v'_2) \in p(R)$  then continue
6   if  $o(v'_1) \not\sqsubseteq o(v'_2)$  then return false
7   for all  $a \in A$ 
8     insert  $(t_a(v'_1), \mathcal{A}(t_a(v'_2)))$  into todo insert  $(v'_1, v'_2)$  into  $R$ 
9 return true

```

Now to check whether a certain vector v satisfies the threshold of T , it is enough to run **HKP_A**(α, e_t, v) where e_t is the unit vector for t as defined above.

The soundness of the proposed algorithm can be shown in essentially the same way as for **HKP** but using a novel up-to technique to take care of the abstraction \mathcal{A} . For the completeness, we need the following additional result.

Lemma 5.3.15 For all vectors $v \in (\mathbb{N}_0 \cup \{\infty\})^X$ it holds that

- (i) $\mathcal{A}(t_a(\mathcal{A}(v))) = \mathcal{A}(t_a(v))$
- (ii) $\mathcal{A}(o(\mathcal{A}(v))) = \mathcal{A}(o(v))$.

Proof:

- (i) Observe that \mathcal{A} is increasing and thus, if $\mathcal{A}(t_a(v))[x] = \infty$, $\mathcal{A}(t_a(\mathcal{A}(v)))[x] = \infty$ necessarily holds, as well. Also note that $\mathcal{A}(t_a(v))[x] > T$ implies $\mathcal{A}(t_a(v))[x] = \infty$. Thus we can prove this lemma by showing the following: Let $I = \{x \in X \mid v[x] > T\}$, $x \in X$ and $a \in A$ such that $t_a(v)[x] \leq T$ be given, then $t_a(\mathcal{A}(v))[x] = t_a(v)[x]$.

All entries greater than T necessarily get mapped to ∞ .

We first compute:

$$t_a(\mathcal{A}(v))[x] = \bigsqcup \{\mathcal{A}(v)[y] \sqcap t_a[y, x] \mid y \in X\} = \min\{\mathcal{A}(v)[y] + t_a[y, x] \mid y \in X\}$$

and analogously

$$t_a(v)[x] = \min\{v[y] + t_a[y, x] \mid y \in X\}$$

Since we know that $t_a(v)[x] \leq T$, there must exist a $y \in X$ such that $v[y] + t_a[y, x] \leq T$. Observe that $v[x] > T$ and $t_a[y', x] \in \mathbb{N}_0$ for all $y' \in I$, therefore $y \notin I$. Thus

$$t_a(v)[x] = \min\{v[y] + t_a[y, x] \mid y \in X \setminus I\}$$

Now we can compute:

$$\begin{aligned} t_a(\mathcal{A}(v))[x] &= \min\{\mathcal{A}(v)[y] + t_a[y, x] \mid y \in X\} \\ &= \min\{\min\{\mathcal{A}(v)[y] + t_a[y, x] \mid y \in X \setminus I\}, \min\{\mathcal{A}(v)[y'] + t_a[y', x] \mid y' \in I\}\} \\ &= \min\{t_a(v)[x], \min\{\infty + t_a[y', x] \mid y' \in I\}\} = \min\{t_a(v)[x], \infty\} = t_a(v)[x] \end{aligned}$$

- (ii) First we show that if $o(v) \leq T$ it also holds that $o(\mathcal{A}(v)) = o(v)$. If $o(v) \leq T$ then $\min\{o[i] + v[i] \mid 1 \leq i \leq |X|\} \leq T$, so there is an index i where the minimum is reached and smaller than or equal T , i.e. $o[i] + v[i] \leq T$. Thus, since $+$ is increasing, $v[i] \leq T$ and therefore $\mathcal{A}(v[i]) = v[i]$. Since \mathcal{A} is also only increasing, we can conclude $o(\mathcal{A}(v)) = o(v)$.

It remains to be shown that $o(v) > T \Leftrightarrow o(\mathcal{A}(v)) > T$. Certainly, if $o(v) > T$ it must follow that $o(\mathcal{A}(v)) > T$, since $\mathcal{A}(v) \geq v$. So assume now, for the converse direction, that $o(\mathcal{A}(v)) > T$ holds. This means $\min\{o[i] + \mathcal{A}(v)[i] \mid 1 \leq i \leq |X|\} > T$ and assume $o(\mathcal{A}(v)) \neq o(v)$. Then there exists an index i where the minimum is reached, i.e. an index such that $o[i] + v[i] \leq o[j] + v[j]$ for all j . Since \mathcal{A} is only increasing the entries of a vector, it follows $o[i] + v[i] < o[j] + \mathcal{A}(v[j])$ for all j . Thus,

$o[i] + v[i] < o[i] + \mathcal{A}(v[i])$, i.e. $v[i] < \mathcal{A}(v[i])$. It follows that $\mathcal{A}(v[i]) = \infty$. Then, $v[i] > T$ and since $o[i] \geq 0$ it follows that $o(v) > T$.

□

Lemma 5.3.16 $p \bullet \sqsubseteq$ is b_2 -compatible, where $p: Rel_{\mathbb{S}^X} \rightarrow Rel_{\mathbb{S}^X}$ is the monotone function assigning to each relation R its precongruence closure.

Proof: By Lemma 5.3.9 and Lemma 5.3.10, \sqsubseteq and p are b_2 -compatible. It is easy to check that for all relations R, S , it holds that $b_2(R) \bullet b_2(S) \subseteq b_2(R \bullet S)$. Therefore, by Lemma 5.3.2, $p \bullet \geq$ is b_2 -compatible. □

Theorem 5.3.17 $HKP_{\mathcal{A}}(\alpha, e_t, v_1)$ always terminates. Moreover, $HKP_{\mathcal{A}}(\alpha, e_t, v_1)$ returns true iff $\llbracket v_1 \rrbracket(w) \leq T$ for all $w \in A^*$.

Proof: Termination is obvious, as there are only finitely many vectors with entries from the set $\{0, 1, 2, \dots, T, \infty\}$. We now prove soundness. Observe that the following is an invariant for the while loop at step 3.

$$R \subseteq b_2((p(R) \cup \text{todo}) \bullet \sqsubseteq)$$

since $\mathcal{A}(t_a(v'_1)) \sqsubseteq t_a(v'_1)$, i.e. $\mathcal{A}(t_a(v'_1)) \geq t_a(v'_1)$. If $HKP_{\mathcal{A}}$ returns true then *todo* is empty and thus $R \subseteq b_2(p(R) \bullet \sqsubseteq)$. By Lemma 5.3.16, Theorem 5.3.1 and Lemma 5.3.8, $e_t \preceq v_1$, i.e. $T \geq \llbracket v_1 \rrbracket(w)$ for all $w \in A^*$.

The converse implication is more elaborated than its analogous in Theorem 5.3.6. Assume that $HKP_{\mathcal{A}}$ yields false, then a vector v'_1 was found such that $o(v'_1) > T$. This means there exists a word $w = a_1 a_2 \dots a_n$ such that

$$\begin{aligned} v'_1 &= t_{a_n}(\mathcal{A}(t_{a_{n-1}}(\mathcal{A}(t_{a_{n-2}}(\dots \mathcal{A}(t_{a_1}(v_1)) \dots)))) \\ &\leq \mathcal{A}(t_{a_n}(\mathcal{A}(t_{a_{n-1}}(\mathcal{A}(t_{a_{n-2}}(\dots \mathcal{A}(t_{a_1}(v_1)) \dots)))) \end{aligned}$$

Now we can apply Lemma 5.3.15 and eliminate all inner \mathcal{A} -applications, yielding $v'_1 \leq \mathcal{A}(t_{a_n}(t_{a_{n-1}}(t_{a_{n-2}}(\dots (t_{a_1}(v_1)) \dots)))$. For easier reading we will now call $v' := t_{a_n}(t_{a_{n-1}}(t_{a_{n-2}}(\dots (t_{a_1}(v_1)) \dots)))$. Since $o(v'_1) > T$, certainly $o(\mathcal{A}(v')) > T$ due to transitivity. Since \mathcal{A} is increasing, $o(\mathcal{A}(v')) \leq \mathcal{A}(o(\mathcal{A}(v')))$, which is, according to Lemma 5.3.15, $\mathcal{A}(o(v'))$. Due to transitivity of $>$ we can conclude $\mathcal{A}(o(v')) > T$, i.e. $\mathcal{A}(o(v')) = \infty$. According to the definition of \mathcal{A} , it follows that $o(v') > T$, therefore w is indeed a witness for the automaton not to respect the threshold. □

5.3.5 Exploiting Similarity

For checking language inclusion of non-deterministic automata it is often convenient to precompute a *similarity* relation that allows to immediately skip some pairs of states [ACHV10]. This idea can be readapted to weighted automata over an l -monoid by using the following notion.

Definition 5.3.18 (Simulation) *Let (X, o, t) be a weighted automaton. A relation $R \subseteq \mathbb{S}^X \times \mathbb{S}^X$ on unit vectors is called a simulation relation whenever for all $(v, v') \in R$*

- (i) $o(v) \sqsubseteq o(v')$
- (ii) *for all $a \in A$, there exists a pair (u, u') that is a linear combination of vector pairs in R and furthermore $t_a(v) \sqsubseteq u$, $u' \sqsubseteq t_a(v')$.*

Similarity, written \preceq , is the greatest simulation relation.

Lemma 5.3.19 *Simulation implies language inclusion, i.e. $\preceq \subseteq \sqsubseteq$*

Proof:

We will prove this inductively, by showing that $\llbracket v \rrbracket(w) \sqsubseteq \llbracket v' \rrbracket(w)$ for all $w \in \Sigma^*$.

- **Induction start** ($|w| = 0$): In this case, $\llbracket v \rrbracket(w) = \llbracket v \rrbracket(\varepsilon) = o(v) \sqsubseteq o(v') = \llbracket v' \rrbracket(\varepsilon) = \llbracket v' \rrbracket(w)$.
- **Induction hypothesis:** For all words w , where $|w| \leq n$, it holds that $\llbracket v \rrbracket(w) \sqsubseteq \llbracket v' \rrbracket(w)$.
- **Induction step** ($n \rightarrow n+1$): Let w be given such that $|w| = n+1$. Then w can be written as $w = aw'$, $a \in \Sigma$, $w' \in \Sigma^*$. Since $(v, v') \in R$, there must exist (u, u') , $(v_1, v'_1), (v_2, v'_2), \dots, (v_m, v'_m) \in R$, $s_1, s_2, \dots, s_m \in \mathbb{L}$ such that $u = \sqcup \{v_i \cdot s_i \mid 1 \leq i \leq m\}$, $u' = \sqcup \{v'_i \cdot s_i \mid 1 \leq i \leq m\}$ and $t_a(v) \sqsubseteq u$, $u' \sqsubseteq t_a(v')$. Using monotonicity of \cdot and \sqcup (wrt. \sqsubseteq), we obtain the following two inequalities:

$$\llbracket v \rrbracket(w) = \llbracket v \rrbracket(aw') = \llbracket t_a(v) \rrbracket(w') \sqsubseteq \llbracket u \rrbracket(w')$$

$$\llbracket u' \rrbracket(w') \sqsubseteq \llbracket t_a(v') \rrbracket(w') = \llbracket v' \rrbracket(aw') = \llbracket v' \rrbracket(w)$$

Now we can apply the induction hypothesis, keeping in mind that $|w'| = n$: Since $\llbracket v_i \rrbracket(w') \sqsubseteq \llbracket v'_i \rrbracket(w')$ for all $1 \leq i \leq m$. Applying again monotonicity of \cdot and \sqcup , as well as the definition of u and u' , we get $\llbracket u \rrbracket(w') \sqsubseteq \llbracket u' \rrbracket(w')$. Finally, transitivity yields $\llbracket v \rrbracket(w) \sqsubseteq \llbracket v' \rrbracket(w)$.

□

There are other, matrix-based definitions for simulation used for weighted automata, but for our purpose we need a relational notion of simulation.

Similarity over an l -monoid can be computed with Algorithm 5.3.20. Even though the relation is not symmetric, the method is conceptually close to the traditional partition refinement algorithm to compute bisimilarity. Starting from the cross-product of all states, the algorithm first eliminates all pairs of states where the first state does not have a smaller-or-equal output than the second one and then continuously removes all pairs of states that do not meet the second requirement for a simulation relation, until the relation does not change anymore.

Algorithm 5.3.20 Algorithm to compute similarity (\preceq) for a weighted automaton $\alpha = (X, o, t)$.

- **Input:** A weighted automaton $\alpha = (X, o, t)$ over a semiring \mathbb{S}
- **Output:** R , the greatest simulation relation on α

SIM α

```

1  $R := \{(v, v') \in \mathbb{S}^X \times \mathbb{S}^X \mid v, v' \text{ are unit vectors}\}$ 
2  $R' := \emptyset$ 
3 for all  $(v, v') \in R$ 
4   if  $o(v) \not\sqsubseteq o(v')$  then  $R := R \setminus \{(v, v')\}$ 
5 while  $R \neq R'$ 
6    $R' := R$ 
7   for all  $a \in A$ 
8     for all  $(v, v') \in R$ 
9        $u := \sqcup \{v_1 \cdot (v_2 \rightarrow t_a(v')) \mid (v_1, v_2) \in R\}$ 
10      if  $t_a(v) \not\sqsubseteq u$  then  $R := R \setminus \{(v, v')\}$ 
11 return  $R$ 

```

We will now show the correctness of the algorithm to compute the bisimilarity.

Lemma 5.3.21 *SIM computes \preceq .*

Proof: First observe that, since simulations are closed under union, there exists a greatest simulation relation on unit vectors.

- Due to the nature of the algorithm, it necessarily terminates after finitely many steps: In the beginning, R contains only finitely many pairs of vectors and in each iteration, either some pairs are removed from R , or R does not change, but in the latter case the algorithm terminates.
- The result is always a simulation relation, this can be seen as follows: Let R be the result of a run of Algorithm 5.3.20 and $(v, v') \in R$. Then
 - It holds that $o(v) \sqsubseteq o(v')$, because otherwise, the pair (v, v') would have been removed from R in the first for all-loop.
 - Furthermore,

$$t_a(v) \sqsubseteq \bigsqcup \{v_1 \cdot (v_2 \rightarrow t_a(v')) \mid (v_1, v_2) \in R\} =: u$$

Moreover, for all $(v_1, v_2) \in R$, $v_2 \cdot (v_2 \rightarrow t_a(v')) \sqsubseteq t_a(v')$ holds per definition of residuation. Therefore,

$$u' := \bigsqcup \{v_2 \cdot (v_2 \rightarrow t_a(v')) \mid (v_1, v_2) \in R\} \sqsubseteq t_a(v')$$

holds, as well. This means that (u, u') is a linear combination of R -vectors and $t_a(v) \sqsubseteq u$, $u' \sqsubseteq t_a(v')$.

- Now we will show inductively, that there cannot exist any greater simulation relation. The algorithm starts with the full cross-product of unit vectors and removes all pairs (v, v') where $o(v) \sqsubseteq o(v')$ does not hold – meaning that these pairs cannot be in a simulation relation at all. Thus, before we enter the nested for all-loops, R is a superset of (or equal to) the greatest simulation relation on α . We will now show that, whenever a pair of vectors is removed in the nested for all-loops, it cannot be contained in a simulation relation, therefore proving that after each execution of the nested for all-loops, R retains the property to be a superset of (or equal to) the greatest simulation relation on α .

Assume that (v, v') is an element of the greatest simulation relation $R' = \{(v_1, v'_1), (v_2, v'_2), \dots, (v_n, v'_n)\}$. Then for all $a \in A$, there must exist multiplicands s_1, s_2, \dots, s_n such that $t_a(v) \sqsubseteq \sqcup\{v_i \cdot s_i \mid 1 \leq i \leq n\}$ and $\sqcup\{v'_i \cdot s_i \mid 1 \leq i \leq n\} \sqsubseteq t_a(v')$. From this it follows that for any given $1 \leq i \leq n$, it holds that $v'_i \cdot s_i \sqsubseteq t_a(v')$ and therefore $s_i \sqsubseteq \sqcup\{\ell \mid v'_i \cdot \ell \sqsubseteq t_a(v')\} = v'_i \rightarrow t_a(v')$, since it is included in the set. We can therefore compute, using $R' \sqsubseteq R$, which is the induction hypothesis:

$$\begin{aligned} u &= \sqcup\{\bar{v} \cdot (\bar{v}' \rightarrow t_a(v')) \mid (\bar{v}, \bar{v}') \in R\} \\ &\sqsupseteq \sqcup\{\bar{v} \cdot (\bar{v}' \rightarrow t_a(v')) \mid (\bar{v}, \bar{v}') \in R'\} \\ &= \sqcup\{v_i \cdot (v'_i \rightarrow t_a(v')) \mid 1 \leq i \leq n\} \\ &\sqsupseteq \sqcup\{v_i \cdot s_i \mid 1 \leq i \leq n\} \geq t_a(v) \end{aligned}$$

Thus, the if-condition $t_a(v) \not\sqsubseteq u$ in the second-to-last line does not evaluate to true, meaning that (v, v') will not be removed from R in the current iteration.

□

Lemma 5.3.22 *The runtime complexity of SIM when applied to an automaton over state set X and alphabet A is polynomial, assuming constant time complexity for all semiring operations (supremum, multiplication, residuation).*

Proof: Assuming all semiring operations (supremum, multiplication, residuation) consume constant time, the runtime of the algorithm can be analysed as follows: the for-loop in line 3 is executed $|X|^2$ many times, once for each element of R , which is initialised as all pairs of unit vectors of dimension $|X|$, of which there are exactly $|X|$ many. The while-loop in line 5 is executed until R remains constant for one iteration. Since R contains at most $|X|^2$ many elements in the beginning (if no pair was thrown out in the preceding for-loop), and in each step of the inner for all-loop, line, R remains either constant or a pair of vectors gets taken out of R . The for all-loop in line 7 is executed $|A|$ -times each time, and the inner for all-loop is executed $|R|$ -times. While line 10 only takes constant time, line 9 takes $|X| \cdot |R|$ many steps. At worst, the time taken by the whole while-loop in line 5 therefore is $|A| \cdot \sum_{i=1}^{|X|^2} ((|X|^2 - i)^2 \cdot |X|) \in \mathcal{O}(|A| \cdot |X|^7)$, if in each iteration exactly one pair of vectors gets removed from R . Obviously, the latter loop dominates the former, so the run time is in $\mathcal{O}(|A| \cdot |X|^7)$. □

Once \preceq is known, it can be exploited by **HKP** and **HKP_A**. To be completely formal in the proofs, it is convenient to define two novel algorithms – called **HKP'** and **HKP'_A** – which are obtained from **HKP** and **HKP_A** by changing step 5 to take simulation into account, using a function p' , where $p'(R)$ is defined for all relations R as $p'(R) = p(R \cup \preceq)$. In either algorithm, we replace the fifth line with the following line:

1 **if** $(v'_1, v'_2) \in p'(R)$ **then continue**

The following two results state the correctness of the two algorithms.

Lemma 5.3.23 *Let \mathbb{S} be a semiring equipped with a precongruence \sqsubseteq . Whenever **HKP'** (v_1, v_2) terminates, it returns true iff $v_1 \preceq v_2$.*

Proof: For soundness, we use the invariant $R \subseteq b_2(p'(R) \cup \text{todo})$, which allows to conclude that $R \subseteq b_2(p'(R))$. Now p' is not guaranteed to be compatible, but p'' defined for all relations R as $p''(R) = p(R \cup \preceq)$ is compatible by Lemma 5.3.9 and Lemma 5.3.2. By Lemma 5.3.19, $\preceq \subseteq \preceq$. By monotonicity of p , we have that $p'(R) = p(R \cup \preceq) \subseteq p(R \cup \preceq) = p''(R)$. By monotonicity of b_2 , $R \subseteq b_2(p''(R))$. By Theorem 5.3.1 and Lemma 5.3.8, $v_1 \preceq v_2$. For completeness, we proceed in the same way as in Theorem 5.3.6. \square

Lemma 5.3.24 ***HKP'_A** (e_t, v_1) always terminates. Moreover, **HKP'_A** (e_t, v_1) returns true iff $\llbracket v_1 \rrbracket(w) \leq T$ for all $w \in A^*$.*

Proof: For termination and completeness we reuse the same argument as in Theorem 5.3.17. For soundness we need to combine the proofs of Lemma 5.3.23 and of Theorem 5.3.17: we use the invariant $R \subseteq b_2((p'(R) \cup \text{todo}) \bullet \sqsubseteq)$, which allows to conclude that $R \subseteq b_2(p'(R) \bullet \sqsubseteq)$. Now p' is not guaranteed to be b_2 -compatible, but p'' , as defined in the proof of Lemma 5.3.23, is. By Lemma 5.3.2, $p'' \bullet \sqsubseteq$ is compatible, since \sqsubseteq is compatible. By monotonicity of p , we have that $p'(R) \bullet \sqsubseteq = p(R \cup \preceq) \bullet \sqsubseteq \subseteq p(R \cup \preceq) \bullet \sqsubseteq = p''(R) \bullet \sqsubseteq$. By monotonicity of b_2 , $R \subseteq b_2(p''(R) \bullet \sqsubseteq)$. By Theorem 5.3.1 and Lemma 5.3.8, $v_1 \preceq v_2$. \square

5.3.6 An Exponential Pruning

To illustrate the benefits of up-to techniques, we show an example where **HKP'_A** exponentially prunes the exploration space by exploiting the technique p' . We

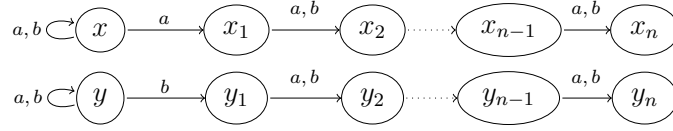


Figure 5.2: Class of examples where HKP_A' exponentially improves over ABK . Output weight is always 0, transition weight is always 1.

compare HKP_A' against ABK in Algorithm 5.3.25, that can be thought of as an adaptation of the algorithm proposed in [ABK11] to the notation used in this paper.

Algorithm 5.3.25 Algorithm to check whether a vector v_0 of a weighted automata (X, o, t) satisfies the threshold $T \in \mathbb{N}_0$

- **Input:** A weighted automaton $\alpha = (X, o, t)$ over a semiring \mathbb{S} , a threshold T and an initial vector $v_0 \in \mathbb{S}^X$
- **Output:** True, if v_0 respects the threshold T , false otherwise

$\text{ABK}(\alpha, T, v_0)$

```

1 todo := { $v_0$ }
2  $P := \emptyset$ 
3 while todo  $\neq \emptyset$ 
4     extract  $v$  from todo
5     if  $v \in P$  then continue
6     if  $o(v) \not\leq T$  then return false
7     for all  $a \in A$  insert  $\mathcal{A}(t_a(v))$  into todo
8     insert  $v$  into  $P$ 
9 return true

```

Consider the family of automata over the tropical semiring in Figure 5.2 and assume that $T = n$. By taking as initial vector $e_x \sqcup e_y$ (i.e., the vector mapping x and y to 0 and all the other states to ∞), the automaton clearly does not respect the threshold, but this can be observed only for words longer than n .

First, for ABK the runtime is exponential. This happens, since every word up to length n produces a different weight vector. For a word w of length m

state x_i has weight m iff the i -last letter of the word is a , similarly state y_i has weight m iff the i -last letter is b . All other weights are ∞ . For instance, the weights for word aab are given below.

x	x_1	x_2	x_3	x_4	\dots	y	y_1	y_2	y_3	y_4	\dots
3	∞	3	3	∞	\dots	3	3	∞	∞	∞	\dots

Now we compare this behaviour with HKP_A' . Observe that $x_i \preceq x$, $y_i \preceq y$ for all i . (Keep in mind that since the order is reversed, a lower weight simulates a higher weight.) Therefore, we obtain rewriting rules that allow to replace an ∞ -entry in x_i and y_i by m for all i (since both entries x and y are m , we can always apply this rule). In the example above this leads to a vector where every entry is 3.

Hence, it turns out that for all words of the same length, the corresponding vectors are all in the precongruence relation with each other – as they share the same normal form – and we only have to consider exactly one word of each length. Therefore, only linearly many words are considered and the runtime is polynomial.

5.4 Runtime Results for the Threshold Problem

We now discuss runtime results for the threshold problem for weighted automata over the tropical semiring of the natural numbers. We compare the following three algorithms: the algorithm without up-to technique (ABK), Algorithm 5.3.25, the algorithm that works up-to precongruence (HKP_A), explained in Subsection 5.3.4, and the algorithm that additionally exploits pre-computed similarity (HKP_A'), introduced in Subsection 5.3.5. This precomputation step is relatively costly and is included in the runtime results below.

We performed the following experiment: for certain values of $|X|$ (size of state set) and of T (threshold) we generated random automata. The alphabet size was randomly chosen between 1 and 5. For each pair of states and alphabet symbol, the probability of having an edge with finite weight is 90%. We chose this high number, since otherwise the threshold is almost never respected and the algorithms return false almost immediately due to absence of a transition

for a given letter. With our choice instead, the algorithms need many steps and the threshold is satisfied in 14% of the cases. In case the weight is different from ∞ , a random weight from the set $\{0, \dots, 10\}$ is assigned.

For each pair $(|X|, T)$ we generated 1000 automata. The runtime results can be seen in Table 5.2. We considered the 50%, 90% and 99% percentiles: the 50% percentile is the median and the 90% percentile means that 90% of the runs were faster and 10% slower than the time given in the respective field. Analogously for the 99% percentile.

Apart from the runtime we also measured the size of the relation R (or P in the case of ABK) and the size of the similarity \preceq (in case of HKP_A'). The program was written in C# and executed on an Intel Core 2 Quad CPU Q9550 at 2.83 GHz with 4 GB RAM, running Windows 10.

First note that, as expected, HKP_A and HKP_A' always produce much smaller relations than ABK. However, they introduce some overhead, due to rewriting for checking $p(R)$, and due to the computation of similarity, which is clearly seen for the 50% percentile. However, if we look at the larger parameters and at the 90% and 99% percentiles (which measure the worst-case performance), HKP_A and HKP_A' gain the upper hand in terms of runtime.

Note also that, while in the example above (Subsection 5.3.6) similarity played a large role, this is not the case for the random examples. Here similarity (not counting the reflexive pairs) is usually quite small. This means that similarity does not lead to savings, only in very few cases does the size of R decrease for HKP_A' . But this also means that the computation of \preceq is not very costly and therefore the runtime of HKP_A is quite similar to the runtime of HKP_A' . We believe that for weighted automata arising from concrete problems, the similarity relation will usually be larger and promise better runtimes. Note also that similarity is independent of the initial vector and the threshold. Consequently, if one wants to answer several threshold questions for the same automaton, it has to be computed only once.

We will now consider two additional examples, demonstrating the possible gain over the ABK algorithm. The first example is particularly simple, in this case, computing a simulation relation up-front does not yield any benefits. However, due to the small size of the example, the disadvantage of the additionally required computation is negligible either way. Therefore, in this case, we will

		Runtime (millisec.)			Size of R/P			Size of \preceq		
(X , T)	algo	50%	90%	99%	50%	90%	99%	50%	90%	99%
(3,10)	HKP_A'	2	8	20	5	14	33	0	2	4
	HKP_A	1	3	14	5	14	34	-	-	-
	ABK	1	3	13	6	28	92	-	-	-
(3,15)	HKP_A'	3	17	127	11	34	100	0	2	4
	HKP_A	2	16	126	11	34	100	-	-	-
	ABK	2	17	90	18	119	373	-	-	-
(3,20)	HKP_A'	6	65	393	18	70	174	0	2	4
	HKP_A	4	64	466	18	71	192	-	-	-
	ABK	5	79	315	55	364	825	-	-	-
(6,10)	HKP_A'	21	227	1862	18	106	302	0	2	12
	HKP_A	8	217	1858	19	106	302	-	-	-
	ABK	9	286	2045	40	693	2183	-	-	-
(6,15)	HKP_A'	90	2547	12344	65	353	750	0	2	11
	HKP_A	84	2560	12328	65	353	750	-	-	-
	ABK	88	4063	20987	346	3082	7270	-	-	-
(6,20)	HKP_A'	239	7541	59922	111	589	1681	0	3	11
	HKP_A	234	7613	60360	111	589	1681	-	-	-
	ABK	253	16240	103804	702	6140	14126	-	-	-
(9,10)	HKP_A'	274	9634	73369	98	582	1501	0	3	21
	HKP_A	236	9581	72833	99	582	1501	-	-	-
	ABK	232	17825	99332	536	6336	14956	-	-	-
(9,15)	HKP_A'	1709	71509	301033	256	1517	3319	0	3	19
	HKP_A	1681	70587	301018	256	1517	3319	-	-	-
	ABK	919	112323	515386	1436	14889	28818	-	-	-
(9,20)	HKP_A'	3885	168826	874259	407	2347	5086	0	3	20
	HKP_A	3838	168947	872647	407	2347	5086	-	-	-
	ABK	1744	301253	1617813	2171	22713	48735	-	-	-
(12,10)	HKP_A'	1866	93271	560824	247	1586	3668	0	7	31
	HKP_A	1800	92490	560837	251	1586	3668	-	-	-
	ABK	1067	189058	889949	1342	18129	37387	-	-	-
(12,15)	HKP_A'	5127	363530	1971541	423	3001	6743	0	7	35
	HKP_A	5010	362908	1968865	423	3001	6743	-	-	-
	ABK	1418	509455	2349335	1672	27225	55627	-	-	-
(12,20)	HKP_A'	15101	789324	3622374	744	4489	9027	0	6	32
	HKP_A	15013	787119	3623393	744	4489	9027	-	-	-
	ABK	4169	1385929	4773543	3297	43756	80712	-	-	-

Table 5.2: Runtime results on randomly generated automata

only compare the **ABK** algorithm to our algorithm HKP_A where R is initialised to the empty set.

Example 5.4.1 Consider the automaton depicted below with initial vector $(0, 0)$:



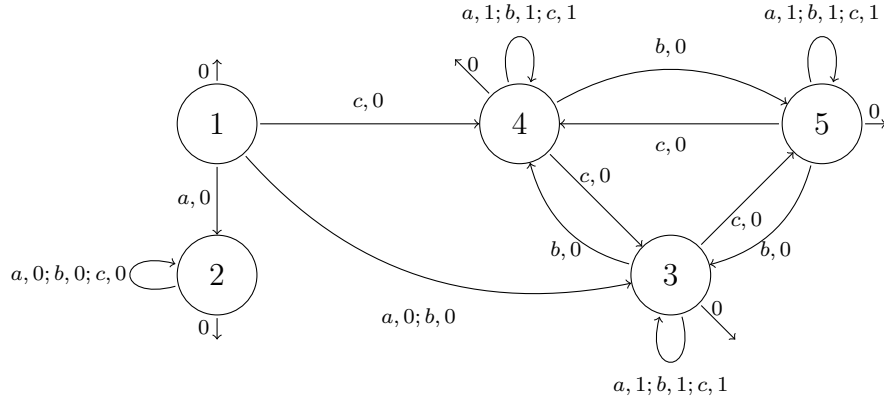
It is easy to see that this automaton does not respect any threshold, since for any threshold T , b^{T+1} is assigned the value $T + 1$. Our algorithm can observe this fact after considering only $3 \cdot T + 2$ many words, i.e. all those words (up to size $T + 1$) that contain only b s and at most one letter that is not a b at the end.

The corresponding vectors for any word that contains symbols other than b are dropped immediately in HKP_A , because they are subsumed by the corresponding vector consisting only of b 's. However, when performing the **ABK** algorithm, no such words can be ignored, but all different vectors must be considered. This renders only those words equivalent that contain the same numbers of a s, b s and c s.

As a result, the runtime of the **ABK** algorithm grows roughly quadratic in the threshold T , whereas it grows only roughly linearly in T for our algorithm HKP_A . For instance, for the threshold 100, we have recorded a runtime of our algorithm of less than 4 seconds, whereas the **ABK** algorithm took 49 seconds to terminate.

A more elaborate automaton, that also allows us to see a possible advantage in computing the simulation relation in advance will be detailed in the next example.

Example 5.4.2 Consider the following automaton:



Again, this automaton does not respect any threshold, when using the initial vector $(0, \infty, \infty, \infty, \infty)$, since for any threshold T , ba^{T+1} (as well as ca^{T+1}) is assigned the value $T + 1$. This automaton can be extended by increasing the size of the circle going from 3 over 4 to 5 by additional states. With increasing circle length, the number of words, that need to be explored in order to determine that the threshold is not respected, increases in the **ABK** algorithm.

However, it remains constant in our algorithm, since we can exploit the fact that the states on the circle can simulate each other. Provided the threshold is chosen large enough wrt. the circle length, we can observe how many words of any given length (larger than the number of states, shorter than the threshold) need to be explored by each algorithm. The following table shows how these numbers, as well as runtimes change when increasing the number of states in the circle:

	HKP_A'	HKP_A	ABK
Circle length 3, time / ms	4605	12091	107704
words per word-length	3	9	81
Circle length 4, time / ms	4484	12088	167628
words per word-length	3	9	126
Circle length 5, time / ms	5174	12890	357400
words per word-length	3	9	240
Circle length 6, time / ms	6657	12628	605130
words per word-length	3	9	315

The experiments were conducted with a threshold of 100. Note that there is

a significant speed up due to the up-to algorithm and also a gain to be observed from initialising R with the greatest simulation. However, with increasing circle length, the advantage of using the simulation decreases, since the computation of the simulation relation takes more time when the number of states increases.

However, the significance of the simulation relation is considerably greater when changing the initial vector to $(0, 0, \infty, \infty, \infty)$. In this case, any threshold is respected. Using the simulation relation, the algorithm can already conclude the positive result for the threshold after only exploring the empty word, the other two algorithms behave very similarly as with the initial vector $(0, \infty, \infty, \infty, \infty)$, though. For instance, with circle length 3, the time required by HKP_A' could, in our experiment, be reduced to 215 milliseconds. Note, that in this case, the time taken by the approach using the simulation relation is independent of the threshold T , whereas the runtime is linear in the value of T (i.e. exponential in its encoding) in the two other cases. For instance, if we increase the threshold to 200, the time used by the algorithm using the simulation relation remains constant at 204 milliseconds (the difference is just fluctuation), whereas our algorithm, when initialised with the empty set, takes 28228 milliseconds and the unoptimised reference algorithm takes 214546 milliseconds. Note, that the runtime was more than doubled in both cases. The results can be observed from the following table, where the algorithm is applied to the same automaton, but the threshold is increased stepwise from $T = 100$ up to $T = 500$.

	HKP_A'	HKP_A	ABK
$T=100$, time / ms	215	8087	65184
$T=200$, time / ms	204	28228	214546
$T=300$, time / ms	229	60813	456371
$T=400$, time / ms	304	102957	792559
$T=500$, time / ms	201	159925	1196630

The significant speed-up when using the initial vector $(0, 0, \infty, \infty, \infty)$ stems from the fact that state 2 simulates all other states. Therefore, for all $1 \leq i \leq 5$, (e_i, e_2) is in the relation R when the threshold algorithm is initialised. Linear combination of all these pairs yields the pair $(e_1 + e_2 + e_3 + e_4 + e_5, e_2) \in c'(R)$.

After adding $((\infty, \infty, \infty, \infty, \infty, 0), (0, 0, \infty, \infty, \infty))$ to R in the first step, we can drop all further vector pairs as follows: All vectors v that can be derived

in one or more steps from $(0, 0, \infty, \infty, \infty, \infty)$ (remember we add an additional state to the automaton to decide the threshold problem) still contain a 0 in the second component so, for the candidate $((\infty, \infty, \infty, \infty, \infty, 0), v)$ we can observe:

$$\begin{aligned} (\infty, \infty, \infty, \infty, \infty, 0) \ c'(R) \ (0, 0, \infty, \infty, \infty, \infty) &\sqsubseteq (e_1 + e_2 + e_3 + e_4 + e_5) \\ c'(R) \ (\infty, 0, \infty, \infty, \infty, \infty) &\sqsubseteq v \end{aligned}$$

So the speed-up overall depends on the simulation relation, closure under linear combination and closure under \sqsubseteq .

5.5 The Shortest Path Problem in Directed Weighted Graphs

The rewriting algorithm to compute the congruence closure over the tropical semiring is closely related to the Dijkstra algorithm to find the shortest paths in directed weighted graphs.

A *weighted directed graph* is a couple $G = (V, \text{weight})$ where $V = \{1, 2, \dots, n\}$ is the set of vertices and $\text{weight} : V \times V \rightarrow \mathbb{R}_0^+ \cup \{\infty\}$ is a function assigning to each pair of vertices v, w the weight to move from v to w . Intuitively, the weight is ∞ , if there is no way (no edge) to go from v directly to w .

Definition 5.5.1 (Rewriting System of a Graph) Let $G = (V, \text{weight})$ be a weighted, directed graph and e_i be the i -th unit vector of dimension $|V|$ in \mathbb{T} and fix v_i to be the vector representing the outgoing arrows from the i -th vertex, i.e. $v_i[j] = \text{weight}((i, j))$. The rewriting system associated to G is $\mathcal{R}_G = \{e_i \mapsto v_i \mid 1 \leq i \leq |V|\}$.

Proposition 5.5.2 For a graph G and vertex i , let $\Downarrow e_i$ be the normal form reached with \mathcal{R}_G . Then for all vertices j , $\Downarrow e_i[j]$ is the weight of the shortest path from i to j .

Proof:

- We first show inductively that for all vertices j it holds that $\Downarrow e_i[j] \leq d(i, j)$. We show this by induction over the length k of a shortest path from i to j

Induction Start For $k = 0$, it is clear that $j = i$ must hold and due to monotonicity of rewriting, we have $d(i, i) = 0 = e_i[i] \geq \Downarrow e_i[i]$

Induction Hypothesis For all vertices j where a shortest path of length at most k exists, it holds that $\Downarrow e_i[j] \leq d(i, j)$.

Induction Step Let j be a vertex such that there exists a shortest path of length $k + 1$ from i to j and let u be the predecessor on said path. Then $\Downarrow e_i(u) \leq d(i, u)$ and there exists a rule $e_u \mapsto v_u$. Since $\Downarrow e_i$ is a fixpoint we can deduce

$$\begin{aligned} \Downarrow e_i[j] &= \min\{\Downarrow e_i[j], (e_u \rightarrow \Downarrow e_i) + v_u[j]\} \\ &= \min\{\Downarrow e_i[j], \Downarrow e_i[u] + v_u[j]\} \\ &\leq \min\{\Downarrow e_i[j], d(i, j)\} \leq d(i, j) \end{aligned}$$

- The other way around, to show that $\Downarrow e_i[j] \geq d(i, j)$ for all vertices j , we argue that this property holds for all rewriting steps individually. So first we observe that $e_i[j] \geq d(i, j)$ for all vertices j is trivially true. Now, given any vector v such that $v[j] \geq d(i, j)$ for all vertices j , assume a rule $e_k \rightarrow v_k$ is applied. Then for all vertices j we can compute the result of the rule application as follows:

$$\begin{aligned} &\min\{(e_k \rightarrow v) + v_k[j], v[j]\} \\ &= \begin{cases} v[j] & \text{if } v[j] \leq (e_k \rightarrow v) + v_k[j] \\ (e_k \rightarrow v) + v_k[j] & \text{otherwise} \end{cases} \\ &\geq \begin{cases} d(i, j) & \text{if } v[j] \leq (e_k \rightarrow v) + v_k[j] \\ d(i, k) + v_k[j] & \text{otherwise} \end{cases} \geq d(i, j) \end{aligned}$$

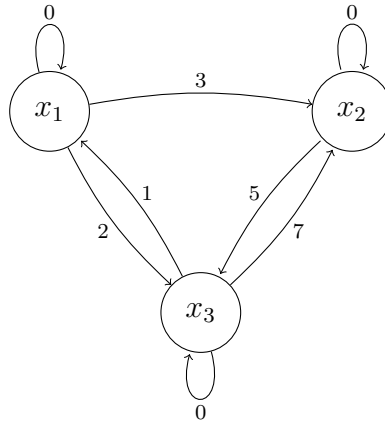
□

In order for rewriting to behave exactly like Dijkstra's algorithm, we need to always choose the rule with the smallest multiplicand that is applicable – i.e. the greatest one according to the order of the lattice. Choosing a rule corresponds to choosing a vertex to explore, determining the multiplicand corresponds to finding the weight, accumulated from the starting vertex to the vertex to be explored next, and applying the rule corresponds to updating the distances of all adjacent vertices that can be reached via a shorter path than the currently known shortest path.

Keep in mind that, while rewriting can be used to find shortest paths in a graph, it does not work the other way around, because for the shortest-path-interpretation we depend on a very specific kind of rewriting system, where each rule has a left hand side that is exactly a unit vector and for each unit vector there exists exactly one rule it is part of.

The following example shows the rewriting at work.

Example 5.5.3 Consider the following directed graph:



This graph corresponds to the following rule system:

$$\mathcal{R} = \left\{ \begin{pmatrix} 0 \\ \infty \\ \infty \end{pmatrix} \mapsto \begin{pmatrix} 0 \\ 3 \\ 2 \end{pmatrix}, \begin{pmatrix} \infty \\ 0 \\ \infty \end{pmatrix} \mapsto \begin{pmatrix} \infty \\ 0 \\ 5 \end{pmatrix}, \begin{pmatrix} \infty \\ \infty \\ 0 \end{pmatrix} \mapsto \begin{pmatrix} 1 \\ 7 \\ 0 \end{pmatrix} \right\}$$

Now we can, for instance, determine the weights of the shortest paths to all vertices starting in the vertex x_3 as follows:

$$\begin{pmatrix} \infty \\ \infty \\ 0 \end{pmatrix} \rightsquigarrow 1 + \begin{pmatrix} 0 \\ \infty \\ \infty \end{pmatrix} \min \begin{pmatrix} 1 \\ 7 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 7 \\ 0 \end{pmatrix} \rightsquigarrow 1 + \begin{pmatrix} 0 \\ 3 \\ 2 \end{pmatrix} \min \begin{pmatrix} 1 \\ 7 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \\ 0 \end{pmatrix}$$

Afterwards, no more rewriting rule is applicable.

Note, that rewriting is non-deterministic and we could have chosen another path. If we choose to apply the rule that has the smallest coefficient, we effectively simulate Dijkstra's algorithm.

We could have chosen to rewrite using 7 plus the second rule, first, but we would have then had to apply the first rule anyway and would have ended up with the same end result.

5.6 Conclusion and Future Work

In this work, we have investigated up-to techniques for weighted automata, including methods to determine the congruence closure for semimodules.

Related work: Related work on up-to techniques has already been discussed in the introduction. For the language equivalence problem for weighted automata we are mainly aware of the algorithm presented in [BLS06], which is a partition refinement algorithm and which already uses a kind of up-to technique: it can eliminate certain vectors which arise as linear combinations of other vectors. This algorithm also is very similar to the one presented in Chapter 4. The paper [UH14] considers simulation for weighted automata, but not in connection to up-to techniques.

Congruence closure for term rewriting has been investigated in [CLS96].

Our examples mainly involved the tropical semiring (and related semirings). Hence, there are relations to work by Aceto et al. [AEI03] who presented an equational theory for the tropical semiring and related semirings, as well as Gaubert et al. [GP97] who discuss several reasons to be interested in the tropical semiring and present solution methods for several types of linear equation systems.

Future work: As we have seen in the experiments on the threshold problem, our techniques greatly reduce the size of the relations. However, the reduction in runtime is less significant, which is due to the overhead of the computation of similarity and the rewriting procedure. There is still a substantial improvement for the worst-case running times (90% and 99% percentiles). So far, the algorithms, especially algorithm **SIM** for computing similarity, are not very sophisticated and we believe that there is further potential for optimisation.

Chapter 6

Algorithmic Issues and Applications for Conditional Transition Systems with Upgrades

6.1 Introduction

Software product lines (SPLs) are a software engineering method for managing a collection of similar software systems with common features. To ensure correctness of such systems in an efficient way, it is common to specify the behaviour of many products in a single transition system and provide suitable analysis methods based on model-checking or behavioural equivalences (see [CCP⁺12, FUB06, tBFGM16, GLS08, CCS⁺13, AFL15, CHS⁺10]).

A popular modelling technique for software product lines are featured transition systems (FTSs) [CCS⁺13], where transitions are annotated with features and a transition can only be taken when the corresponding feature is present in the product under consideration. An issue that has rarely been studied, is the notion of self-adaptivity for FTS [CCH⁺13], i.e., the view that features or products are not fixed a priori, but may change during run-time. Here we focus on adaptivity by performing upgrades.

In this context of adaptive SPLs, we are especially interested in behavioural equivalences, specifically in bisimilarity. The questions we ask are the following:

Given two states in the transition system specifying an SPL, are they equivalent for all products? Are they equivalent for a specific product? For which products are they equivalent? Such questions are relevant when we compare a system with its specification or want to modify a system in such a way that its observable behaviour is invariant. Furthermore, computing bisimilarity provides a minimisation technique for transition systems that are potentially very large.

To address these issues, we introduce the so-called conditional transition systems (CTSs) as a formal model expressive enough to incorporate upgrades in SPL. CTSs describe a more abstract view than FTSs: *transitions are labelled by conditions, corresponding to products, rather than features*. An equivalent view is to represent those transition systems by matrices (one for each label), where we assign to each pair of states x, y the set of products that allow a transition from x to y .

This exhibits three levels of abstraction: single features, products or conditions (sets of features) and sets of products. This is strongly reminiscent of propositional logic or Boolean algebra: features correspond to atomic propositions (X, Y, Z), products to complete conjunctions (e.g., $X \wedge \neg Y \wedge Z$) and sets of products to (equivalence classes of) formulae or Boolean expressions (e.g., $X \vee \neg Y$). All these different views can be helpful: The product view allows for compact and natural modelling, while the feature view can separate the concerns for different products when modelling and connects CTS to the well-known FTS. Finally, the Boolean view allows a compact representation of the fixed-point algorithm for bisimilarity checking based on matrix multiplication and BDDs.

The notion of an upgrade can be integrated into the above setting by switching from Boolean algebras to distributive lattices, where the upgrade ordering gives rise to the lattice order. Consequently, much of the theory and methods can be reused, including the matrix multiplication algorithm.

Our contributions are as follows. First, we make the different levels in the specification of SPLs explicit and give a theoretical foundation in terms of Boolean algebras and lattices. Second, we work out a theory of behavioural equivalences with corresponding games and algorithms catered towards conventional and adaptive SPLs. Third, we present our implementation based on binary decision diagrams (BDDs), which provide a compact encoding of a propositional formula

and also show how they can be employed in a lattice-based setting. Lastly, we show how the BDD-based matrix multiplication algorithm provides us with an efficient way to check bisimilarity, that can be vastly superior to the naive approach of checking all products, i.e., all feature combinations, separately.

This chapter is organised as follows. In Section 6.2 we formally introduce CTSs and conditional bisimilarity. Using the Birkhoff duality, it is shown in Section 6.3 that CTSs can be represented as lattice transition systems (LaTSs) whose transitions are labelled with the elements from a distributive lattice. Moreover, the bisimilarity introduced on LaTSs is shown to coincide with the conditional bisimilarity on the corresponding CTSs. In Section 6.4, we show how bisimilarity can be computed using a form of matrix multiplication. Section 6.5 focusses on the translation between an FTS and a CTS, and moreover, a BDD-based implementation of checking bisimilarity is laid out. Lastly, we conclude with a discussion on related work and future work in Section 6.6.

6.2 Conditional Transition Systems

The theory of conditional transition systems leans heavily on the theory of orders and (finite, distributive) lattices, as introduced in Section 2.2. Throughout the chapter, the reader is assumed to be familiar with the notions of lattices, partially ordered sets and the Birkhoff duality.

In this section, we introduce conditional transition systems to formally specify software product lines together with a notion of behavioural equivalence based on bisimulation. In [ABH⁺12], such transition systems were already investigated in a coalgebraic setting without an order on the conditions. Naturally, the CTS from [ABH⁺12] arise as a special case of our notion of CTS, when \leq is chosen as the discrete order $=$.

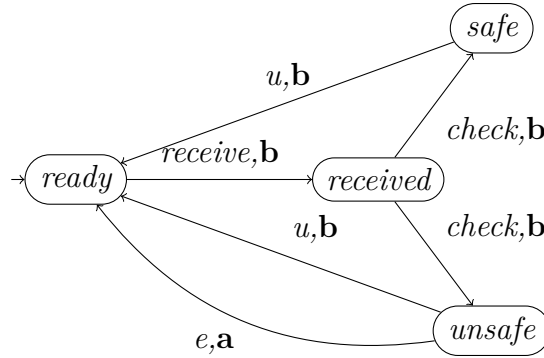
Definition 6.2.1 *Let (Φ, \leq) be a finite poset. Then, a conditional transition system (CTS) is a triple (X, A, f) consisting*

- *a set of states X*
- *a finite set A called the label alphabet and*
- *a function $f: X \times A \rightarrow (\Phi \rightarrow \mathcal{P}(X))$ mapping every pair in $X \times A$ to a monotone function of type $(\Phi, \leq) \rightarrow (\mathcal{P}(X), \supseteq)$.*

We call the elements of Φ the conditions of the CTS. As usual, we write $x \xrightarrow{a,\varphi} y$ whenever $y \in f(x, a)(\varphi)$.

Intuitively, a CTS evolves as follows. Before the system starts acting, it is assumed that all the conditions are fixed and a condition $\varphi \in \Phi$ is chosen arbitrarily which represents a selection of a valid product of the system (product line). Now all the transitions that have a condition greater than or equal to φ are activated, while the remaining transitions are inactive. Henceforth, the system behaves like a standard transition system; until at any point in the computation, the condition is changed to a smaller one (say, φ') signifying a selection of a valid, upgraded product. This, in turn, has the propelling effect in the sense that now (de)activation of transitions depends on the new condition φ' , rather than on the old condition φ . Note that, due to the monotonicity restriction, we have that $x \xrightarrow{a,\varphi} y$ and $\varphi' \leq \varphi$ imply $x \xrightarrow{a,\varphi'} y$. That is, active transitions remain active during an upgrade, but new transitions may become active.¹ Labelled Transition Systems arise as a special case, where $|\Phi| = 1$.

Example 6.2.2 Consider a simplified example (originally from [CCH⁺13]) of an adaptive routing protocol modelled as a CTS over the alphabet $A = \{\text{receive}, \text{check}, u, e\}$ graphically as follows:



The system consists of two products: the basic system with no encryption feature written as **b** and the advanced system with encryption feature written as **a**. The ordering on the sets of valid products is defined as the smallest poset

¹Monotonicity is important to ensure the duality between CTSs and lattice transition systems (see Section 6.3). In Subsection 6.4.5 we will weaken the requirement that transitions remain active after an upgrade by discussing a mechanism for deactivating transitions via priorities.

containing the relation $\mathbf{a} < \mathbf{b}$. Transitions that are present due to monotonicity are omitted.

Initially, the system is in state 'ready' and is waiting to receive a message. Once a message is received there is a check whether the system's environment is safe or unsafe, leading to non-deterministic branching. If the encryption feature is present, then the system can send an encrypted message (e) from the unsafe state only; otherwise, the system sends an unencrypted message (u) regardless of the state being 'safe' or 'unsafe'. Note that such a behaviour description can be easily encoded by a transition function. E.g., $f(\text{received}, \text{check})(\mathbf{b}) = \{\text{safe}, \text{unsafe}\}$ and $f(\text{received}, a)(x) = \emptyset$ (for $x \in \{\mathbf{a}, \mathbf{b}\}$ and $a \in A \setminus \{\text{check}\}$) specifies the transitions that can be fired from the received state to the (un)safe states.

Next, we turn our attention towards (strong) bisimulation relations for CTSs which consider the ordering of products in their transfer properties.

Definition 6.2.3 (Conditional Bisimulation) *Let (X, A, f) , (Y, A, g) be two CTSs over the same set of conditions (Φ, \leq) . For a condition $\varphi \in \Phi$, we define $f_\varphi(x, a) = f(x, a)(\varphi)$ to denote the traditional (A -)labelled transition system induced by a CTS (X, A, f) . We will sometimes call f_φ the instantiation of f to φ . Two states $x \in X, y \in Y$ are conditionally bisimilar under a condition $\varphi \in \Phi$, denoted $x \sim_\varphi y$, if there is a family of relations $R_{\varphi'}$ (for every $\varphi' \leq \varphi$) such that*

- (i) *each relation $R_{\varphi'}$ is a traditional bisimulation relation² between $f_{\varphi'}$ and $g_{\varphi'}$,*
- (ii) *whenever $\varphi' \leq \varphi''$, it holds that $R_{\varphi'} \supseteq R_{\varphi''}$, and*
- (iii) *R_φ relates x and y , i.e. $(x, y) \in R_\varphi$.*

To illustrate conditional bisimulation, we discuss our previous example of a CTS.

Example 6.2.4 *As an example of conditional bisimulation, consider the CTS illustrated in Example 6.2.2 where the condition \mathbf{b} of the transition*

²Since φ' is fixed, the traditional bisimulation relation only takes labels from A into account.

'received $\xrightarrow{\text{check}, \mathbf{b}}$ unsafe' is replaced by \mathbf{a} . Let ready_1 and ready_2 denote the initial states of the system before and after the above modification, respectively. Then, we find $\text{ready}_1 \sim_{\mathbf{a}} \text{ready}_2$; however, $\text{ready}_1 \not\sim_{\mathbf{b}} \text{ready}_2$. To see why the latter fails to hold, consider the smallest bisimulation relation $R_{\mathbf{b}}$ in the traditional sense between α and β under the condition \mathbf{b} that contains the pair $(\text{ready}_1, \text{ready}_2)$. The relation $R_{\mathbf{b}}$ has the following form:

$$R_{\mathbf{b}} = \{(\text{ready}_1, \text{ready}_2), (\text{received}_1, \text{received}_2), (\text{unsafe}_1, \text{safe}_2), (\text{safe}_1, \text{safe}_2)\}$$

Similarly, we can give the bisimilarity $R_{\mathbf{a}}$ between the instantiations of α and β to the condition \mathbf{a} :

$$R_{\mathbf{b}} = \{(\text{ready}_1, \text{ready}_2), (\text{received}_1, \text{received}_2), (\text{unsafe}_1, \text{unsafe}_2), (\text{safe}_1, \text{safe}_2)\}$$

Observe, that the states $\text{unsafe}_1, \text{safe}_2$ are bisimilar in the traditional sense, under the condition \mathbf{b} , i.e., $(\text{unsafe}_1, \text{safe}_2) \in R_{\mathbf{b}}$. This pair is required in any bisimulation relation that contains $(\text{ready}_1, \text{ready}_2)$, because otherwise, in the bisimulation game, after the necessary receive-steps in round 1, Player 1 can make a select-move from received_1 to unsafe_1 which cannot be answered by Player 2. However, the two states cannot be related by any traditional bisimulation relation under the condition \mathbf{a} , because from unsafe_1 an encrypted-transition is possible, which is not possible from the state safe_2 . Therefore, Condition 2 of Definition 6.2.3 is violated and the states $\text{ready}_1, \text{ready}_2$ can be seen not to be conditionally bisimilar, even though there are bisimilar for all possible instantiations.

Indeed, the two systems behave differently. In the first, it is possible to perform actions receive, check (arrive in state unsafe), do an upgrade, and send an encrypted message (e), which is not feasible in the second system, because the check transition forces the system to be in the state 'safe' before doing the upgrade. However, without upgrades, the above systems would be conditionally bisimilar for both products.

6.3 Lattice Transition Systems

Recall from Theorem 2.2.10 that there is a duality between partial orders and distributive lattices. In fact, as we will show below, this result can be lifted to

the level of transition systems as follows: a conditional transition system over a poset is equivalent to a transition system whose transitions are labelled by the downward closed subsets of the poset (cf. Theorem 6.3.2).

Definition 6.3.1 (Lattice Transition Systems) A lattice transition system (*LaTS*) over an alphabet A and a finite distributive lattice \mathbb{L} is a triple (X, A, α) consisting of a set of states X and a transition function $\alpha: X \times A \times X \rightarrow \mathbb{L}$.

Note that superficially, lattice transition systems resemble weighted automata [DKV09]. However, while in weighted automata the lattice annotations are seen as weights that are accumulated, in CTSs they play the role of guards that control which transitions can be taken. Furthermore, the notions of behavioural equivalence are quite different.

Given a CTS (X, A, f) over (Φ, \leq) , we can easily construct a LaTS over $\mathcal{O}(\Phi)$ by defining $\alpha(x, a, x') = \{\varphi \in \Phi \mid x' \in f(x, a)(\varphi)\}$ for $x, x' \in X, a \in A$. Due to monotonicity, $\alpha(x, a, x')$ is always downward-closed. Similarly, a LaTS can be converted into a CTS by using the Birkhoff duality and by taking the irreducibles as conditions.

Theorem 6.3.2 *The set of all CTSs over a set of ordered conditions Φ is isomorphic to the set of all LaTSs over the lattice whose elements are the downward-closed subsets of Φ .*

Proof: Given a set X , a partially ordered set (Φ, \leq) , and $\mathbb{L} = \mathcal{O}(\Phi)$, we define an isomorphism between the sets $(\Phi \xrightarrow{\text{mon.}} \mathcal{P}(X))^{X \times A}$ and $\mathcal{O}(\Phi)^{X \times A \times X}$. Consider the following function mappings $\eta: (\Phi \xrightarrow{\text{mon.}} \mathcal{P}(X))^{X \times A} \rightarrow \mathcal{O}(\Phi)^{X \times A \times X}$, $f \mapsto \eta(f)$ and $\eta': \mathcal{O}(\Phi)^{X \times A \times X} \rightarrow (\Phi \xrightarrow{\text{mon.}} \mathcal{P}(X))^{X \times A}$, $\alpha \mapsto \eta'(\alpha)$ defined as:

$$\begin{aligned}\eta(f)(x, a, x') &= \{\varphi \in \Phi \mid x' \in f_\varphi(x, a)\}, \\ \eta'(\alpha)(x, a)(\varphi) &= \{x' \mid \varphi \in \alpha(x, a, x')\}.\end{aligned}$$

We need to show that these functions actually map into their proclaimed domain, so we will show that $\eta(f)(x, a, x')$ is downward-closed and that $\eta'(\alpha)(x, a)$ is a monotone function.

Downward-closed Let $\varphi \in \eta(f)(x, a, x')$ and $\varphi' \leq \varphi$. By using these facts in the definition of $f_{\varphi'}$ we find $x' \in f_{\varphi'}(x, a)$, i.e., $\varphi' \in \eta(f)(x, a, x')$.

Anti-monotonicity Let $\varphi \leq \varphi'$ and $x' \in \eta'(\alpha)(x, a)(\varphi')$. Then by definition of η' we find $\varphi' \in \alpha(x, a, x')$. And by downward-closedness of $\alpha(x, a, x')$ we get $\varphi \in \alpha(x, a, x')$, i.e., $x' \in \eta'(\alpha)(x, a)(\varphi)$.

Now it suffices to show that η, η' are inverse to each other, because by the uniqueness of inverses we then have $\eta' = \eta^{-1}$. First, we show that $\eta' \circ \eta = \text{id}$:

$$x' \in f(x, a)(\varphi) \Leftrightarrow x' \in f_\varphi(x, a) \Leftrightarrow \varphi \in \eta(f)(x, a, x') \Leftrightarrow x' \in \eta'(\eta(f))(x, a)(\varphi) .$$

Analogously we can show that $\eta \circ \eta' = \text{id}$:

$$\begin{aligned} \varphi \in \alpha(x, a, x') &\Leftrightarrow x' \in \{x'' \mid \varphi \in \alpha(x, a, x'')\} \Leftrightarrow x' \in \eta'(\alpha)(x, a), (\varphi) \\ &\Leftrightarrow x' \in \eta'(\alpha)_\varphi(x, a) \Leftrightarrow \varphi \in \{\varphi' \in \Phi \mid x' \in \eta'(\alpha)_{\varphi'}(x, a)\} \Leftrightarrow \varphi \in \eta(\eta'(\alpha))(x, a, x') \end{aligned}$$

□

So every LaTS over a finite distributive lattice gives rise to a CTS in our sense (cf. Definition 6.2.1) and, since finite Boolean algebras are finite distributive lattices, conditional transition systems in the sense of [ABH⁺12] are CTSs in our sense as well. We chose the definition of a CTS using posets instead of the dual view using lattices, because this view yields a natural description to model the behaviour of a software product line, though when computing (symbolically) with CTSs we often choose the lattice view.

More importantly, the point is, by adopting the lattice view, conditional bisimulations can be computed more elegantly than the following brute-force approach:

- Instantiate a given CTS for each condition and apply the well-known algorithm for labelled transition systems to each element.
- For each condition φ that is not minimal, take the bisimulations of all smaller conditions than φ , intersect their respective bisimulations to obtain a new bisimulation for φ .
- Repeat until the relations R_φ do not change anymore.

Definition 6.3.3 (Lattice bisimulation) *Let (X, A, α) and (Y, A, β) be any two LaTSs over a lattice \mathbb{L} . A conditional relation R , i.e., a function of type $R: X \times Y \rightarrow \mathbb{L}$ is a lattice bisimulation for α, β if and only if the following transfer properties are satisfied.*

(i) For all $x, x' \in X$, $y \in Y$, $a \in A$, $\ell \in \mathcal{J}(\mathbb{L})$, whenever $x \xrightarrow{a, \ell} x'$ and $\ell \sqsubseteq R(x, y)$, there exists $y' \in Y$ such that $y \xrightarrow{a, \ell} y'$ and $\ell \sqsubseteq R(x', y')$.

(ii) Symmetric to (i) with the roles of x and y interchanged.

In the above, we write $x \xrightarrow{a, \ell} x'$, whenever $\ell \sqsubseteq \alpha(x, a, x')$.

For a condition $\varphi \in \Phi$, we have a transition $x \xrightarrow{a, \varphi} x'$ in the CTS if and only if there exists a transition $x \xrightarrow{a, \downarrow \varphi} x'$ in the corresponding LaTS. Consequently, they are denoted by the same symbol.

Theorem 6.3.4 *Let (X, A, f) and (Y, A, g) be any two CTSs over Φ . Two states $x \in X, y \in Y$ are conditionally bisimilar under a condition φ ($x \sim_{\varphi} y$) if and only if there is a lattice bisimulation R between the corresponding LaTSs such that $\varphi \in R(x, y)$.*

Proof: Let $x \in X, y \in Y$ be any two states in CTS (LaTS) $(X, A, f), (Y, A, g)$ over the conditions Φ $((X, A, \alpha), (Y, A, \beta))$ over the lattice $\mathcal{O}(\Phi \leq)$, respectively.

\Leftarrow Let $\varphi \in \Phi$ be a condition and let R be a lattice bisimulation relation such that $\varphi \in R(x, y)$. Then, we can construct a family of relations $R_{\varphi'}$ (for $\varphi' \leq \varphi$) as follows: $x R_{\varphi'} y \Leftrightarrow \varphi' \in R(x, y)$. For all other φ' , we set $R_{\varphi'} = \emptyset$. The downward-closure of $R(x, y)$ ensures that $R_{\varphi''} \subseteq R_{\varphi'}$ (for $\varphi', \varphi'' \leq \varphi$), whenever $\varphi' \leq \varphi''$.

Thus, it remains to show that every relation $R_{\varphi'}$ is a bisimulation. Let $x R_{\varphi'} y$ and $x' \in f_{\varphi'}(x, a)$. Then, $x \xrightarrow{a, \downarrow \varphi'} x'$. Since $\downarrow \varphi'$ is an irreducible in the lattice, $\downarrow \varphi' \subseteq R(x, y)$ and R is a lattice bisimulation, we find $y \xrightarrow{a, \downarrow \varphi'} y'$ and $\downarrow \varphi' \subseteq R(x', y')$, which implies $\varphi' \in R(x', y')$. That is, $y' \in g_{\varphi'}(y, a)$ and $x' R_{\varphi'} y'$. Likewise, the remaining symmetric condition of bisimulation can be proven.

\Rightarrow Let \sim_{φ} be a conditional bisimulation between the CTS $(X, A, f), (Y, A, g)$, for some $\varphi \in \Phi$. Then, construct a conditional relation: $R(x, y) = \{\varphi \mid x \sim_{\varphi} y\}$. Clearly, the set $R(x, y)$ is a downward-closed subset of Φ due to Definition 6.2.3(ii); i.e., an element in the lattice $\mathcal{O}(\Phi)$. Next, we show that R is a lattice bisimulation.

Let $x \xrightarrow{a, \downarrow \varphi'} x'$ and $\downarrow \varphi' \subseteq R(x, y)$. This implies $x' \in f_{\varphi'}(x, a)$ and $\varphi' \in R(x, y)$, hence $x \sim_{\varphi'} y$. So using the transfer property of traditional bisimulation, we obtain $y' \in g_{\varphi'}(y, a)$ and $x' \sim_{\varphi'} y'$. That is, $y \xrightarrow{a, \downarrow \varphi'} y'$ and $\varphi' \in R(x', y')$,

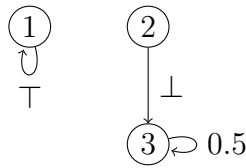
which implies $\downarrow \varphi' \subseteq R(x', y')$. Likewise, the symmetric condition of lattice bisimulation can be proven. \square

Incidentally, the order in \mathbb{L} also gives rise to a natural order on lattice bisimulations. Let $R_1, R_2: X \times Y \rightarrow \mathbb{L}$ be any two lattice bisimulations. We write $R_1 \sqsubseteq R_2$ if and only if $R_1(x, y) \sqsubseteq R_2(x, y)$ for all $x \in X, y \in Y$. As a result, taking the element-wise supremum of a family of lattice bisimulations is again a lattice bisimulation. Therefore, the greatest lattice bisimulation for a LaTS always exists, just like in the traditional case.

Lemma 6.3.5 *Let $R_i \in X \times Y \rightarrow \mathbb{L}, i \in I$ be lattice bisimulations for a pair of LaTSs (X, A, α) and (Y, A, β) . Then $\sqcup\{R_i \mid i \in I\}$ is a lattice bisimulation.*

Proof: Let $x, x' \in X, a \in A, y \in Y$ and $\ell \in \mathcal{J}(\mathbb{L})$ such that $\ell \sqsubseteq \sqcup_{i \in I} R_i(x, y)$ and $x \xrightarrow{a, \ell} x'$. Then, there is an index $i \in I$ such that $\ell \sqsubseteq R_i(x, y)$, since ℓ is an irreducible. Thus, there is a y' such that $y \xrightarrow{a, \ell} y'$ and $\ell \sqsubseteq R_i(x', y') \sqsubseteq \sqcup_{i \in I} R_i(x', y')$. Likewise, the symmetric condition when a transition emanates from y can be proven. \square

Coming back to our previous argument about the distinction between weighted automata and lattice transition systems, we will illustrate via a small example that lattice bisimilarity and weighted bisimilarity (in the sense of e.g. [Buc08]) do not coincide. Consider the lattice $\{\top, 0.5, \perp\}$ with the order $\perp \sqsubseteq 0.5 \sqsubseteq \top$ and the following LaTS over a single (unnamed) action:



Then 1 is lattice bisimilar to 2, as can be immediately observed: both, 1 and 2 cannot perform any action in product 0.5 unless they upgrade to the product \top , which in turn allows them to perform an arbitrary number of steps. No other irreducible elements exist. However, if the LaTS is considered as a weighted automaton, they are not bisimilar. It is even true that they are not language equivalent, since state 1 assigns the weight \top to all words, whereas all words of at least length 2 are assigned 0.5 by state 2.

6.3.1 Correspondence to Fitting's Bisimulation

Fitting [Fit02] has conducted work on characterising conditional relations as matrices, which is strongly related to lattice bisimulation. Remember that Boolean algebras are semirings, therefore we can use matrix operations like matrix multiplication \cdot . Additionally, for any matrix M , M^T denotes its transposed matrix. Fitting calls a matrix $R: X \times X \rightarrow \mathbb{B}$ (for any boolean algebra \mathbb{B}) for a transition matrix $\alpha: X \times X \rightarrow \mathbb{B}$ a bisimulation if and only if $R \cdot \alpha \sqsubseteq \alpha \cdot R$ and $R^T \cdot \alpha \sqsubseteq \alpha \cdot R^T$. By restricting ourselves to LaTSs over Boolean algebras and fixing our alphabet to be a singleton set, we can establish the following correspondence between Fitting's formulation of bisimulation and lattice bisimulation.

Lemma 6.3.6 *Let (X, α) be a LaTS over an atomic Boolean algebra \mathbb{B} . Then, a conditional relation $R: X \times X \rightarrow \mathbb{B}$ is a lattice bisimulation for α if and only if $R \cdot \alpha \sqsubseteq \alpha \cdot R$ and $R^T \cdot \alpha \sqsubseteq \alpha \cdot R^T$.*

Here we interpret α as a matrix of type $X \times X \rightarrow \mathbb{L}$ by dropping the occurrence of action labels.

Proof:

$\boxed{\Leftarrow}$ Let $R: X \times X \rightarrow \mathbb{B}$ be a conditional relation satisfying $R \cdot \alpha \sqsubseteq \alpha \cdot R$ and $R^T \cdot \alpha \sqsubseteq \alpha \cdot R^T$. Then, we need to show that R is a lattice bisimulation. Let $x \xrightarrow{\ell} y$ such that $\ell \in \mathcal{J}(\mathbb{B})$ and $\ell \sqsubseteq R(x, x')$. Then, we find $\ell \sqsubseteq \alpha(x, y)$. That is,

$$\ell \sqsubseteq R(x, x') \sqcap \alpha(x, y) = R^T(x', x) \sqcap \alpha(x, y) \sqsubseteq (R^T \cdot \alpha)(x', y) \sqsubseteq (\alpha \cdot R^T)(x', y).$$

By expanding the last term from above, we find that $\ell \sqsubseteq \alpha(x', y') \sqcap R^T(y', y)$, for some y' . Thus, $\ell \sqsubseteq \alpha(x', y')$ (which implies $x' \xrightarrow{\ell} y'$) and $\ell \sqsubseteq R(y, y')$. Similarly, the remaining condition when the transition emanates from x' can be verified using $R \cdot \alpha \sqsubseteq \alpha \cdot R$.

$\boxed{\Rightarrow}$ Let $R: X \times X \rightarrow \mathbb{B}$ be a lattice bisimulation. Then, we only prove $R \cdot \alpha \sqsubseteq \alpha \cdot R$; the proof of $R^T \cdot \alpha \sqsubseteq \alpha \cdot R^T$ is similar. Note that, for any $x, y' \in X$, we know that the element $(R \cdot \alpha)(x, y')$ can be decomposed into a set of atoms, since \mathbb{B} is an atomic Boolean algebra. Let $(R \cdot \alpha)(x, y') = \bigsqcup_i \ell_i$ for some index set I such that the ℓ_i are atoms or irreducibles in \mathbb{B} .

Furthermore, expanding the above inequality we get, for every $i \in I$ there is a state $y \in X$ such that $\ell_i \sqsubseteq R(x, y) \sqcap \alpha(y, y')$, since the ℓ_i are irreducibles. That is, for every $i \in I$ we have some state y such that $\ell_i \sqsubseteq R(x, y)$ and $\ell_i \sqsubseteq \alpha(y, y')$. Now using the transfer property of R we find some state x' such that $\ell_i \sqsubseteq \alpha(x, x')$ and $\ell_i \sqsubseteq R(x', y')$. Thus, for every $i \in I$ we find that $\ell_i \sqsubseteq (\alpha \cdot R)(x, y')$; hence, since $(\alpha \cdot R)(x, y')$ is an upper bound of all ℓ_i , $(R \cdot \alpha)(x, y') \sqsubseteq (\alpha \cdot R)(x, y')$. \square

6.4 Computation of Lattice Bisimulation

The goal of this section is to present an algorithm that computes the greatest lattice bisimulation between a given pair of LaTSs. In particular, we first characterise lattice bisimulation as a post-fixpoint of an operator F on the set of all conditional relations. Then, we show that this operator F is monotone with respect to the ordering relation \sqsubseteq ; thereby, ensuring that the greatest bisimulation always exists by applying the well-known Knaster-Tarski fixpoint theorem. Moreover, on finite lattices and finite sets of states, the usual fixpoint iteration based on the Kleene fixpoint theorem starting with the trivial conditional relation (i.e., the constant 1-matrix over \mathbb{L}) can be used to compute the greatest lattice bisimulation. Lastly, we give a translation of F in terms of matrices using a form of matrix multiplication found in the literature of residuated lattices [BK12] and database design [KB85]. We will also briefly discuss an extension of CTS that allows for deactivation of transitions on upgrading.

6.4.1 A Fixpoint Approach

Throughout this section, we let $\alpha: X \times A \times X \rightarrow \mathbb{L}$, $\beta: Y \times A \times Y \rightarrow \mathbb{L}$ denote any two LaTSs, \mathbb{L} denote a finite distributive lattice, and \mathbb{B} denote the Boolean algebra that this lattice embeds into.

Definition 6.4.1 *Recall the residuum operator (\rightarrow) on a lattice (cf. Definition 2.2.12) and define three operators $F, F_1, F_2: (X \times Y \rightarrow \mathbb{L}) \rightarrow (X \times Y \rightarrow \mathbb{L})$*

in the following way:

$$\begin{aligned}
F_1(R)(x, y) &= \prod_{a \in A, x' \in X} \left(\alpha(x, a, x') \rightarrow \left(\bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y')) \right) \right), \\
F_2(R)(x, y) &= \prod_{a \in A, y' \in Y} \left(\beta(y, a, y') \rightarrow \left(\bigsqcup_{x' \in X} (\alpha(x, a, x') \sqcap R(x', y')) \right) \right), \\
F(R)(x, y) &= F_1(R)(x, y) \sqcap F_2(R)(x, y).
\end{aligned}$$

Note that the above definition is provided for a distributive lattice, viewing it in classical two-valued Boolean algebra results in the well-known transfer properties of a bisimulation.

Theorem 6.4.2 *A conditional relation R is a lattice bisimulation if and only if R is a post-fixpoint of F , i.e., $R \sqsubseteq F(R)$.*

Proof:

\Leftarrow Let $R: X \times Y \rightarrow \mathbb{L}$ be a conditional relation over a pair of LaTS $(X, A, \alpha), (Y, A, \beta)$ such that $R \sqsubseteq F(R)$. Next, we show that R is a lattice bisimulation. For this purpose, let $\ell \in \mathcal{J}(\mathbb{L})$, $a \in A$. Furthermore, let $x \xrightarrow{a, \ell} x'$ (which implies $\ell \sqsubseteq \alpha(x, a, x')$) and $\ell \sqsubseteq R(x, y)$. From $R(x, y) \sqsubseteq F_1(R)(x, y)$ we infer $\ell \sqsubseteq F_1(R)(x, y)$. This means that $\ell \sqsubseteq \alpha(x, a, x') \rightarrow \left(\bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y')) \right)$. Since $\ell_1 \sqcap (\ell_1 \rightarrow \ell_2) \sqsubseteq \ell_2$, we can take the infimum with $\alpha(x, a, x')$ on both sides and obtain $\ell \sqsubseteq \ell \sqcap \alpha(x, a, x') \sqsubseteq \bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y'))$ (the first inequality holds since $\ell \sqsubseteq \alpha(x, a, x')$). Since ℓ is irreducible, there exists a y' such that $\ell \sqsubseteq \beta(y, a, y')$, i.e., $y \xrightarrow{a, \ell} y'$, and $\ell \sqsubseteq R(x', y')$.

Likewise, the remaining condition when a transition emanates from y can be proven.

\Rightarrow Let $R: X \times Y \rightarrow \mathbb{L}$ be a lattice bisimulation on $(X, A, \alpha), (Y, A, \beta)$. Then, we need to show that $R \sqsubseteq F(R)$, i.e., $R \sqsubseteq F_1(R)$ and $R \sqsubseteq F_2(R)$. We will only give the proof of the former inequality, the proof of the latter is analogous. To show $R \sqsubseteq F_1(R)$, it is sufficient to prove $\ell \sqsubseteq R(x, y) \Rightarrow \ell \sqsubseteq F_1(R)(x, y)$, for all $x \in X, y \in Y$ and all irreducibles ℓ . So let $\ell \sqsubseteq R(x, y)$, for some x, y . Next, simplify $F_1(R)$ as follows:

$$\begin{aligned}
F_1(R)(x, y) &= \prod_{a, x'} (\alpha(x, a, x') \rightarrow \bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y'))) \\
&= \prod_{a, x'} \left[\bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y')) \sqcup \neg \alpha(x, a, x') \right] \quad (\text{Lemma 2.2.18})
\end{aligned}$$

$$= \bigsqcap_{a,x'} \bigsqcup \{m \in \mathbb{L} \mid m \sqsubseteq \bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y')) \sqcup \neg\alpha(x, a, x')\}.$$

Thus, it is sufficient to show that $\ell \sqsubseteq \bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y')) \sqcup \neg\alpha(x, a, x')$, for any $a \in A, x' \in X$. We do this by distinguishing the following cases: either $\ell \sqsubseteq \neg\alpha(x, a, x')$ or $\ell \sqsubseteq \alpha(x, a, x')$. If the former holds (which corresponds to the case where there is no a -labelled transition under ℓ), then the result holds trivially. So assume $\ell \sqsubseteq \alpha(x, a, x')$. Recall, from above, that $\ell \sqsubseteq R(x, y)$ and R is a lattice bisimulation. Thus, there is a $y' \in Y$ such that $\ell \sqsubseteq \beta(y, a, y')$ and $\ell \sqsubseteq R(x', y')$; hence,

$$\ell \sqsubseteq \bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y')) \sqcup \neg\alpha(x, a, x') .$$

□

Next, it is easy to see that F is a monotone operator with respect to the ordering \sqsubseteq on \mathbb{L} , since the infimum and supremum are both monotonic, and moreover, the residuum operation is monotonic in the second component. Since we are only considering finite X, Y, \mathbb{L} , from the fact that F is monotonic it directly follows that F is co-continuous. As a result, we can use the following fixpoint iteration to compute the greatest bisimulation while working with finite lattices and finite sets of states.

Algorithm 6.4.3 Partition refinement algorithm to compute the bisimilarity between two CTSs (X, A, α) and (Y, A, β) .

Input: Two CTSs (X, A, α) and (Y, A, β) , i.e., the sets X, Y , and A are all finite.

Output: A matrix $R \subseteq \mathbb{L}^{X \times Y}$ that is the greatest bisimulation between α and β .

We define the following algorithm:

Step 0 Fix R_0 as $R_0(x, y) = 1$ for all $x \in X, y \in Y$.

Step $i + 1$ Compute $R_{i+1} = F(R_i)$ for all $i \in \mathbb{N}_0$

Termination When $R_n \sqsubseteq R_{n+1}$, return R_n as the greatest bisimulation from α to β .

If we assume $\alpha = \beta$, then it is not hard to see that the fixpoint iteration must stabilise after at most $|X|$ steps, since the R_i always induce equivalence

relations for all conditions φ and refinements regarding φ are immediately propagated to every $\varphi' \geq \varphi$. We will now formalise this result and show that it actually is true.

The crucial point is to prove that the intermediate results R_i induce equivalence relations $R_i[\varphi]$ for all $\varphi \in \Phi$. To prove this, we will heavily make use of the following Lemma:

Lemma 6.4.4 *Let $x, y \in X$ be given, then*

- $\varphi \in F_1(R)$ if and only if for all $\varphi' \leq \varphi$, and $x' \in X$ it holds that $\varphi' \notin \alpha(x, a, x')$ or there exists a y' such that $\varphi' \in \alpha(y, a, y')$ and $\varphi' \in R(x', y')$.
- $\varphi \in F_2(R)$ if and only if for all $\varphi' \leq \varphi$, and $y' \in X$ it holds that $\varphi' \notin \alpha(y, a, y')$ or there exists an x' such that $\varphi' \in \alpha(x, a, x')$ and $\varphi' \in R(x', y')$.

Proof: We will only prove the claim for F_1 , it can be proven analogously for F_2 . Using Lemma 2.2.18 and in particular the approximation operation $\lfloor \cdot \rfloor$, Definition 2.2.17, we can observe that

$$\begin{aligned} F_1(R)(x, y) &= \bigcap_{a \in A, x' \in X} \left(\alpha(x, a, x') \rightarrow_{\mathbb{L}} \left(\bigsqcup_{y' \in X} (\beta(y, a, y') \sqcap R(x', y')) \right) \right) \\ &= \left[\bigwedge_{a \in A, x' \in X} \left(\alpha(x, a, x') \rightarrow_{\mathbb{B}} \left(\bigvee_{y' \in X} (\beta(y, a, y') \wedge R(x', y')) \right) \right) \right] \\ &= \bigwedge_{a \in A, x' \in X} \left(\left[\alpha(x, a, x') \rightarrow_{\mathbb{B}} \left(\bigvee_{y' \in X} (\beta(y, a, y') \wedge R(x', y')) \right) \right] \right) \end{aligned}$$

Now, by definition of approximation, $\varphi \in \lfloor l \rfloor$ if and only if for all $\varphi' \leq \varphi$ it holds that $\varphi' \in l$. Applied to above formula, we get, for all $\varphi' \leq \varphi, a \in A, x' \in X$ the condition

$$\varphi' \in \left(\alpha(x, a, x') \rightarrow_{\mathbb{B}} \left(\bigvee_{y' \in Y} (\beta(y, a, y') \wedge R(x', y')) \right) \right)$$

The well-known characterisation of \rightarrow via negation and disjunction yields the expected result. \square

Definition 6.4.5 *Let $R: X \times X \rightarrow \mathbb{L}$ be a conditional relation, then we define, for all $\varphi \in \Phi$ the relation $R[\varphi] \subseteq X \times X$ according to*

$$R[\varphi] = \{(x, y) \mid \varphi \in R_i(x, y)\}$$

Lemma 6.4.6 *Let R_0, R_1, \dots, R_n be the sequence of conditional relations obtained via Algorithm 6.4.3 applied to (X, α) over \mathbb{L} , then for all $\varphi \in \Phi$, $R_i[\varphi]$ is an equivalence relation.*

Proof: We prove this via induction over the iteration i . For $R_0[\varphi]$ the claim is trivially true, since R_0 is the one-matrix and therefore $R_0[\varphi] = X \times X$ for all conditions $\varphi \in \Phi$. We can now assume that for all $\varphi \in \Phi$, $R_i[\varphi]$ is an equivalence relation and go on to show that then $R_{i+1}[\varphi]$ is an equivalence relation for all $\varphi \in \Phi$ as well. For this purpose, fix a $\varphi \in \Phi$, we can now check the three conditions of equivalence relations. In all instances, we will only show that the condition holds for F_1 , because the proofs for F_2 are completely analogous.

- *Reflexivity:* Let $x \in X$ be given arbitrarily. Via the induction hypothesis we obtain $(x, x) \in R_i[\varphi']$ for all $\varphi' \leq \varphi$. Let any $\varphi' \leq \varphi$ be given, to prove the claim that $(x, x) \in R_{i+1}[\varphi']$, we distinguish two cases. If $\varphi' \notin \alpha(x, a, x')$, then Lemma 6.4.4 shows that the claim is true. Now assume $\varphi' \in \alpha(x, a, x')$, in this case we can just choose $y' = x'$ to find $\varphi' \in \alpha(x, a, y')$. Due to the induction hypothesis, we also know that $\varphi' \in R_i(x', y') = R_i(x', x')$, due to reflexivity of $R_i[\varphi']$.
- *Symmetry:* Let $(x, y) \in R_{i+1}[\varphi]$ be given arbitrarily. We know that $(x, y) \in R_{i+1}[\varphi]$, thus, for all $\varphi' \leq \varphi$ it holds that $\varphi' \in F_2(R_i)(x, y)$, i.e.

$$\varphi' \in F_2(R_i)(x, y) = \bigwedge_{a \in A, y' \in Y} \alpha(y, a, y') \rightarrow \bigvee_{x' \in X} \alpha(x, a, x') \wedge R_i(x', y')$$

Using the induction hypothesis, we can observe that $R_i(x', y') = R_i(y', x')$ and thus also

$$\varphi' \in \bigwedge_{a \in A, y' \in Y} \alpha(y, a, y') \rightarrow \bigvee_{x' \in X} \alpha(x, a, x') \wedge R_i(y', x') = F_1(R_i)(y, x)$$

This concludes the proof of symmetry.

- *Transitivity:* Let any $(x, y) \in R_{i+1}[\varphi], (y, z) \in R_{i+1}[\varphi]$ be given, this means we know the following two things for all $\varphi' \leq \varphi$ and any $a \in A, x', y' \in X$:

1. $\varphi' \in \alpha(x, a, x') \rightarrow_{\mathbb{B}} \left(\bigvee_{y' \in X} (\beta(y, a, y') \wedge R(x', y')) \right)$
2. $\varphi' \in \alpha(y, a, y') \rightarrow_{\mathbb{B}} \left(\bigvee_{z' \in X} (\beta(z, a, z') \wedge R(x', y')) \right)$

Now choose any $a \in A, x' \in X$ such that $\varphi' \in \alpha(x, a, x')$. If no such (a, x') exists, then using Lemma 6.4.4 we have already proven transitivity. Using (1), there must exist a $y' \in X$ such that $\varphi' \in \alpha(y, a, y') \wedge R_i(x', y')$, i.e. $\varphi' \in \alpha(y, a, y')$ and $\varphi' \in R_i(x', y')$. Then, using (2), we must also find a $z' \in X$ such that $\varphi' \in \alpha(z, a, z') \wedge R_i(y', z')$, i.e. $\varphi' \in \alpha(z, a, z')$ and $\varphi' \in R_i(y', z')$. Finally, the induction hypothesis yields that transitivity of $R_i[\varphi']$ and thus, it follows from $\varphi' \in R_i(x', y')$ and $\varphi' \in R_i(y', z')$ that also $\varphi' \in R_i(x', z')$, proving transitivity. \square

Lemma 6.4.7 *If, for all $\varphi' \leq \varphi$, $F(R_i[\varphi']) = R_i[\varphi']$, then $F(F(R_i[\varphi'])) = R_i[\varphi']$ for all $\varphi' \leq \varphi$, i.e., if the fixpoint iteration does not change the relation for a φ and all smaller conditions, it becomes stationary.*

Proof: It holds that

$$\varphi \in F(F(R_i))(x, y) \Leftrightarrow \forall \varphi' \leq \varphi : \varphi' \in F(F(R_i))(x, y)$$

Let $\varphi' \leq \varphi$ such that $\varphi' \in F(F(R_i))(x, y)$. We will again only argue about F_1 and observe that the proof is completely analogously for F_2 . Then:

$$\begin{aligned} & \varphi' \in F_1(F(R_i))(x, y) \\ &= \bigwedge_{a \in A, x' \in X} \left[\alpha(x, a, x') \rightarrow \bigvee_{y' \in X} (\beta(y, a, y') \wedge F(R_i)(x', y')) \right] \\ &\Leftrightarrow \forall \varphi'' \leq \varphi' \forall (a, x') \in A \times X : \varphi'' \notin \alpha(x, a, x') \\ &\quad \vee \exists y' \in X : \varphi'' \in \beta(y, a, y') \wedge \varphi'' \in F(R_i)(x', y') \\ &\Leftrightarrow \forall \varphi'' \leq \varphi' \forall (a, x') \in A \times X : \varphi'' \notin \alpha(x, a, x') \\ &\quad \vee \exists y' \in X : \varphi'' \in \beta(y, a, y') \wedge \varphi'' \in R_i(x', y') \\ &\Leftrightarrow \varphi' \in \bigwedge_{a \in A, x' \in X} \left[\alpha(x, a, x') \rightarrow \bigvee_{y' \in X} (\beta(y, a, y') \wedge R_i(x', y')) \right] = F_1(R_i)(x, y) \end{aligned}$$

\square

Lemma 6.4.8 $R[\varphi] \subseteq R[\varphi']$ for all $\varphi' \leq \varphi$.

Proof: Let $(x, y) \in R[\varphi]$, then

$$\varphi \in R(x, y) \Rightarrow \varphi' \in R(x, y) \Rightarrow (x, y) \in R[\varphi']$$

due to $R(x, y)$ being downward-closed, since R has only values in \mathbb{L} . \square

Theorem 6.4.9 *The fixpoint iteration of Algorithm 6.4.3, applied to the system (X, α) terminates after at most $|X|$ steps.*

Proof: Let $\varphi \in \Phi$ be given arbitrarily, we show that the fixpoint iteration becomes stationary for φ after at most $|X|$ many iterations, then, from Lemma 6.4.7 it follows that the whole fixpoint iteration must become stationary after at most $|X|$ many steps, because the argument can be applied to all conditions independently of each other.

We consider the set

$$\Phi_i = \{\varphi' \leq \varphi \mid \exists \varphi'' \leq \varphi' : R_i[\varphi''] \neq R_{i+1}[\varphi'']\}$$

and show inductively that for all $\psi \in \Phi_i$, it holds that $R_{i+1}[\psi]$ has at least $i + 1$ equivalence classes. For $i = 0$ this is trivially true, because every equivalence relation contains at least one equivalence class. Now, assume the claim is true for i , and show that it also holds for $i + 1$. If $\psi \in \Phi_{i+1}$, then there exists a $\psi' \leq \psi$ such that $R_{i+1}[\psi'] \neq R_{i+2}[\psi']$. The induction hypothesis yields that $R_{i+1}[\psi']$ has at least $i + 1$ equivalence classes, therefore the equivalence relation $R_{i+2}[\psi] \subset R_{i+1}[\psi']$ must have at least $i + 2$ equivalence classes. Since $R_{i+2}[\psi] \subseteq R_{i+2}[\psi']$, $R_{i+2}[\psi]$ must also contain at least $i + 2$ equivalence classes. \square

6.4.2 Lattice Bisimilarity is Finer than Boolean Bisimilarity

We now show the close relation of the notions of bisimilarity for an LaTS defined over a finite distributive lattice \mathbb{L} and a Boolean algebra \mathbb{B} . As usual, let (X, A, α) and (Y, A, β) be any two LaTSs together with the restriction that the lattice \mathbb{L} embeds into the Boolean algebra \mathbb{B} . Moreover, let $F_{\mathbb{L}}$ and $F_{\mathbb{B}}$ be the monotonic operators as defined in Definition 6.4.1 over the lattice \mathbb{L} and the Boolean algebra \mathbb{B} , respectively. We say that R is an \mathbb{L} -bisimulation (resp. \mathbb{B} -bisimulation) whenever $R \sqsubseteq F_{\mathbb{L}}(R)$ (resp. $R \sqsubseteq F_{\mathbb{B}}(R)$).

Proposition 6.1

- (i) *If $R: X \times Y \rightarrow \mathbb{L}$, then $\lfloor F_{\mathbb{B}}(R) \rfloor = F_{\mathbb{L}}(R)$.*
- (ii) *Every \mathbb{L} -bisimulation is also a \mathbb{B} -bisimulation.*

- (iii) A \mathbb{B} -bisimulation $R: X \times Y \rightarrow \mathbb{B}$ is an \mathbb{L} -bisimulation, whenever all the entries of R are in \mathbb{L} .

Proof:

- (i) This follows directly from Lemma 2.2.18, allowing to move the approximations to the inside towards the implications and Lemma 2.2.18, allowing to approximate the implication in \mathbb{B} via the implication in \mathbb{L} .
- (ii) If R is a bisimulation in \mathbb{L} , then $F_{\mathbb{L}}(R) \supseteq R$. Since by definition, $\lfloor Q \rfloor \subseteq Q$ for all conditional relations Q and we have shown in (i) that $F_{\mathbb{L}}(R) = \lfloor F_{\mathbb{B}}(R) \rfloor$, we can conclude $F_{\mathbb{B}}(R) \supseteq \lfloor F_{\mathbb{B}}(R) \rfloor = F_{\mathbb{L}}(R) \supseteq R$. Thus, R is a \mathbb{B} -bisimulation.
- (iii) Clearly, $R \subseteq F_{\mathbb{B}}(R)$, because R is a \mathbb{B} -bisimulation. Since R has exclusively entries from \mathbb{L} , $F_{\mathbb{B}}(R) = \lfloor F_{\mathbb{B}}(R) \rfloor$, and finally (i) yields that $\lfloor F_{\mathbb{B}}(R) \rfloor = F_{\mathbb{L}}(R)$; thus, R is an \mathbb{L} -bisimulation.

□

However, even though the two notions of bisimilarity are closely related, they are not identical, i.e., it is not true that whenever a state x is bisimilar to a state y in \mathbb{B} that it is also bisimilar in \mathbb{L} (see Example 6.2.2 where we encounter a \mathbb{B} -bisimulation, which is not an \mathbb{L} -bisimulation).

6.4.3 Matrix Multiplication

An alternative way to represent a LaTS (X, A, α) is to view the transition function α as a family of matrices $\alpha_a: X \times X \rightarrow \mathbb{L}$ (one for each action $a \in A$), where the function α_a is defined as follows: $\alpha_a(x, x') = \alpha(x, a, x')$, for every $x, x' \in X$. We use standard matrix multiplication (where \sqcup is used for addition and \sqcap for multiplication), as well as a special form of matrix multiplication [BK12, KB85].

Definition 6.4.10 (\otimes -Multiplication) *Given an $X \times Y$ -matrix $U: X \times Y \rightarrow \mathbb{L}$ and a $Y \times Z$ -matrix $V: Y \times Z \rightarrow \mathbb{L}$, we define the \otimes -multiplication of U and V as follows:*

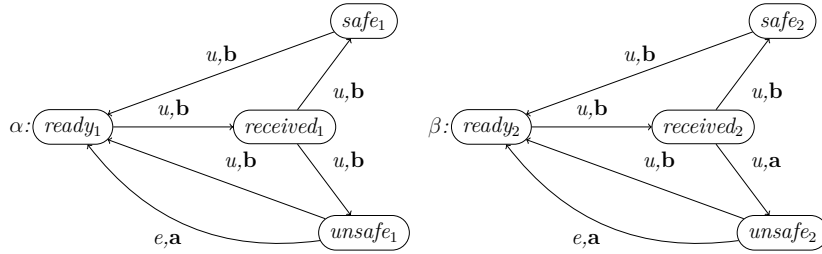
$$(U \otimes V)(x, z) = \bigsqcap_{y \in Y} \left(U(x, y) \rightarrow_{\mathbb{L}} V(y, z) \right) .$$

We obtain the following alternative characterisation of the fixpoint operator F .

Theorem 6.4.11 *Let $R: X \times Y \rightarrow L$ be a conditional relation between a pair of LaTSs (X, A, α) and (Y, A, β) . Then, $F(R) = \prod_{a \in A} ((\alpha_a \otimes (R \cdot \beta_a^T)) \sqcap (\beta_a \otimes (\alpha_a \cdot R)^T)^T$, where R^T denotes the transpose of a matrix R .*

To illustrate the fixpoint iteration and its representation via matrix multiplication, we consider a slightly simplified version of the routing protocol from previous examples.

Example 6.4.12 *Consider the following pair of systems:*



They can be represented by the following matrices, where the columns and rows refer to the states $ready_i$, $received_i$, $safe_i$, $unsafe_i$ (in that order), and the rows refer to the actions u, e (in that order):

$$\alpha = \begin{pmatrix} \emptyset & \{\mathbf{a}, \mathbf{b}\} & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} \\ \emptyset & \emptyset & \emptyset & \emptyset \\ \{\mathbf{a}, \mathbf{b}\} & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset \\ \{\mathbf{a}, \mathbf{b}\} & \emptyset & \emptyset & \emptyset \\ \{\mathbf{a}\} & \emptyset & \emptyset & \emptyset \end{pmatrix}, \beta = \begin{pmatrix} \emptyset & \{\mathbf{a}, \mathbf{b}\} & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}\} \\ \emptyset & \emptyset & \emptyset & \emptyset \\ \{\mathbf{a}, \mathbf{b}\} & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset \\ \{\mathbf{a}, \mathbf{b}\} & \emptyset & \emptyset & \emptyset \\ \{\mathbf{a}\} & \emptyset & \emptyset & \emptyset \end{pmatrix}$$

So, e.g. the entry $\{\mathbf{a}, \mathbf{b}\}$ in the fifth line, first column of α represents the information that it is possible to make a u step from state $safe_1$ to $ready_1$ under both, \mathbf{a} and \mathbf{b} .

Now we can iterate using the matrix multiplication algorithm as follows:

- We initiate the algorithm with the one-matrix.

$$R_0 = \begin{pmatrix} \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} \\ \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} \\ \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} \\ \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} \end{pmatrix}$$

- Since we have two actions, u, e , we need to consider two pairs of submatrices of α and β , where α_u can be obtained by taking all rows of odd index and α_e can be obtained by taking all rows of even index of α , analogously for β_u and β_e . Then F can be computed as $F(R) = (\alpha_u \otimes (R \cdot \beta_u^T)) \sqcap (\beta_u \otimes (\alpha_u \cdot R)^T)^T \sqcap (\alpha_e \otimes (R \cdot \beta_e^T)) \sqcap (\beta_e \otimes (\alpha_e \cdot R)^T)^T$, i.e we have

$$R_1 = F(R_0) = \begin{pmatrix} \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \emptyset \\ \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \emptyset \\ \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \{\mathbf{a}, \mathbf{b}\} & \emptyset \\ \emptyset & \emptyset & \emptyset & \{\mathbf{a}, \mathbf{b}\} \end{pmatrix}$$

$$R_2 = F(R_1) = \begin{pmatrix} \{\mathbf{a}, \mathbf{b}\} & \emptyset & \{\mathbf{a}, \mathbf{b}\} & \emptyset \\ \emptyset & \{\mathbf{a}\} & \emptyset & \emptyset \\ \{\mathbf{a}, \mathbf{b}\} & \emptyset & \{\mathbf{a}, \mathbf{b}\} & \emptyset \\ \emptyset & \emptyset & \emptyset & \{\mathbf{a}, \mathbf{b}\} \end{pmatrix}$$

$$R_3 = F(R_2) = \begin{pmatrix} \{\mathbf{a}\} & \emptyset & \emptyset & \emptyset \\ \emptyset & \{\mathbf{a}\} & \emptyset & \emptyset \\ \emptyset & \emptyset & \{\mathbf{a}, \mathbf{b}\} & \emptyset \\ \emptyset & \emptyset & \emptyset & \{\mathbf{a}, \mathbf{b}\} \end{pmatrix}$$

$$R_4 = F(R_3) = \begin{pmatrix} \{\mathbf{a}\} & \emptyset & \emptyset & \emptyset \\ \emptyset & \{\mathbf{a}\} & \emptyset & \emptyset \\ \emptyset & \emptyset & \{\mathbf{a}\} & \emptyset \\ \emptyset & \emptyset & \emptyset & \{\mathbf{a}\} \end{pmatrix} = F(R_4)$$

So the algorithm terminates after computing $R_5 = F(R_4) = R_4$. We can conclude that $x_1 \sim_{\mathbf{a}} x_2$ for all $x \in \{\text{ready}, \text{received}, \text{safe}, \text{unsafe}\}$ but no other pairs of states are bisimilar under \mathbf{a} and no pair of states is bisimilar under \mathbf{b} .

We end this section by making an observation on LaTS over a Boolean algebra. In a Boolean algebra, it is well known that the residuum operator can be eliminated by the negation and join operators. Thus, in this case, using only the standard matrix multiplication and (component wise) negation we get $U \otimes V = \neg(U \cdot (\neg V))$. Hence, a conditional relation $F(R)$ can be rewritten as:

$$F(R) = \prod_{a \in A} \left(\neg(\alpha_a \cdot \neg(R \cdot \beta_a^T)) \sqcap \neg(\neg(\alpha_a \cdot R) \cdot \beta_a^T) \right) .$$

This reduction is especially relevant to software product lines with no upgrade features.

6.4.4 Bisimulation Game

We will now show that conditional bisimulation gives rise to a bisimulation game that exactly characterises conditional bisimulation. This game is an adaptation of the bisimulation game for traditional labelled transition systems.

Definition 6.4.13 (Bisimulation Game) *Bisimulation can be characterised using the following game. Given two CTSs (X, A, f) and (Y, A, g) over a poset (Φ, \leq) , a state $x \in X$, a state $y \in Y$, and a condition $\varphi \in \Phi$, the bisimulation game is a round-based two-player game that uses both the CTSs as game boards. Let (x, y, φ) be a game instance indicating that there is a marked state in each of X and Y at any given time. The game progresses to the next game instance as follows:*

- *Player 1 is the first one to move. Player 1 can decide to make an upgrade, i.e., replace the condition φ by a smaller one (say $\varphi' \leq \varphi$, for some $\varphi' \in \Phi$).*
- *Player 1 can then choose either the marked state $x \in X$ or $y \in Y$ and must perform a transition $x \xrightarrow{a, \varphi'} x'$ or $y \xrightarrow{a, \varphi'} y'$ for some $a \in A$.*
- *Player 2 then has to simulate the last step, i.e., if Player 1 made a step $x \xrightarrow{a, \varphi'} x'$, Player 2 is required to make step $y \xrightarrow{a, \varphi'} y'$ and vice-versa.*
- *In turn, the new game instance is (x', y', φ') .*

Player 1 wins, if Player 2 cannot simulate the last step performed by Player 1. Player 2 wins, if the game never terminates or Player 1 cannot make another step.

So bisimulation is characterised as follows: Player 2 has a winning strategy for a game instance (x, y, φ) if and only if $x \sim_\varphi y$.

Lemma 6.4.14 *Given two CTSs (X, A, f) , (Y, A, g) and an instance (x, y, φ) of a bisimulation game, then whenever $x \sim_\varphi y$, Player 2 has a winning strategy for (x, y, φ) .*

Proof: The strategy of Player 2 can be directly derived from the family of CTS bisimulation relations $\{R_{\varphi'} \mid \varphi' \in \Phi\}$ where $(x, y) \in R_\varphi$. The strategy works inductively. Assume at any given point of time in the game, we have that the currently investigated condition is φ and $(x, y) \in R_\varphi$, where x and y are the currently marked states in X respectively Y . Then Player 1 upgrades to $\varphi' \leq \varphi$. Due to the condition on CTS bisimulations of reverse inclusion, we have $R_{\varphi'} \supseteq R_\varphi$, therefore $(x, y) \in R_{\varphi'}$. Then, when Player 1 makes a step $x \xrightarrow{a, \varphi'} x'$ in f , there must exist a transition $y \xrightarrow{a, \varphi'} y'$ in g such that $(x', y') \in R_{\varphi'}$ due to $R_{\varphi'}$ being a (traditional) bisimulation. Analogously, if Player 1 chooses a transition $y \xrightarrow{a, \varphi'} y'$ in g , there exists a transition $x \xrightarrow{a, \varphi'} x'$ in f for Player 2 such that $(x', y') \in R_{\varphi'}$. Hence, Player 2 will be able to react and establish the inductive condition again. In the beginning, the condition holds per definition. Thus, Player 2 has a winning strategy. □

We will now prove the converse, by explicitly constructing a winning strategy for Player 1, whenever two states are not in a bisimulation relation.

Lemma 6.4.15 *Given two CTSs A, B and an instance (x, y, φ) of a bisimulation game, then whenever $x \not\sim_\varphi y$, Player 1 has a winning strategy for (x, y, φ) .*

Proof: We consider the LaTSs which correspond to the CTSs A, B and compute the fixpoint by using the matrix multiplication algorithm, obtaining a sequence $R_0, R_1, \dots, R_n = R_{n+1} = \dots$ of lattice-valued relations $R_i: X \times Y \rightarrow \mathcal{O}(\Phi, \leq)$. Note, that instead of using exactly the matrix multiplication method, we can also use the characterization of Definition 6.3.3: whenever there exists a transition $x \xrightarrow{a, \varphi} x'$, for which there is no matching transition with $y \xrightarrow{y, \varphi} y'$

with $\varphi \in R_{i-1}(x', y')$, the condition φ and all larger conditions $\varphi' \geq \varphi$ have to be removed from $R_{i-1}(x, y)$ in the construction of $R_i(x, y)$.

We will now define $M^{\varphi'}(x, y) = \max\{i \in \mathbb{N}_0 \mid \varphi' \in R_i(x, y)\}$, where $\max \mathbb{N}_0 = \infty$. An entry $M^{\varphi'}(x, y) = \infty$ signifies that $x \sim_{\varphi'} y$, whereas any other entry $i < \infty$ means that x, y were separated under condition φ' at step i and hence $x \not\sim_{\varphi'} y$.

Now assume we are in a game situation with game instance (x, y, φ) where Player 1 has to make a step. We will show that if $M^\varphi(x, y) = i < \infty$, Player 1 can choose an upgrade $\bar{\varphi} \leq \varphi$, an action $a \in A$ and a step $x \xrightarrow{a, \bar{\varphi}} x'$ (or $y \xrightarrow{a, \bar{\varphi}} y'$) such that independently of the choice of the corresponding state y' , respectively x' , which Player 2 makes, $M^{\bar{\varphi}}(x', y') < i$.

For each $\varphi' \leq \varphi$ compute

$$\begin{aligned} \omega(\varphi') = \min\{ & \min_{a, x'} \{ \max_{y'} \{ M_n^{\varphi'}(x', y') \mid y \xrightarrow{a, \varphi'} y' \} \mid x \xrightarrow{a, \varphi'} x' \}, \\ & \min_{a, y'} \{ \max_{x'} \{ M_n^{\varphi'}(x', y') \mid x \xrightarrow{a, \varphi'} x' \} \mid y \xrightarrow{a, \varphi'} y' \} \} \end{aligned}$$

The formula can be interpreted as follows: The outer min corresponds to the choice of making a step in transition system A or B . The inner min corresponds to choosing the step that yields the best, i.e. lowest, guaranteed separation value and the max corresponds to the choice of Player 2 that yields the best, i.e. greatest, separation value for him.

Now choose a minimal condition $\bar{\varphi}$, such that $\omega(\bar{\varphi})$ is minimal for all $\varphi' \leq \varphi$. Player 1 now makes an upgrade from φ to $\bar{\varphi}$ and chooses a transition $x \xrightarrow{a, \bar{\varphi}} x'$ or $y \xrightarrow{a, \bar{\varphi}} y'$, such that the minimum in $\omega(\bar{\varphi})$ is reached. This means that Player 2 can only choose a corresponding successor state y' respectively x' such that $M^{\bar{\varphi}}(x', y') \leq \omega(\bar{\varphi})$.

Now it remains to be shown that $\omega(\bar{\varphi}) < i$, via contradiction: assume that $\omega(\bar{\varphi}) \geq i$. Since $\omega(\bar{\varphi})$ is minimal for all $\varphi' \leq \varphi$, we obtain $\omega(\varphi') \geq i$ for all $\varphi' \geq \varphi$. This implies that for each step $x \xrightarrow{a, \varphi'} x'$ there exists an answering step $y \xrightarrow{a, \varphi'} y'$, such that $M^{\varphi'}(x', y') \geq i$ (analogously for every step of y). The condition $M^{\varphi'}(x', y') \geq i$ is equivalent to $\varphi' \in R_i(x', y')$ and hence, we can infer that $\varphi' \in R_{i+1}(x, y)$. This also holds for $\varphi' = \varphi$, which is a contradiction to $M^\varphi(x, y) = i$.

In order to conclude, take two states x, y and a condition φ such that $x \not\sim_\varphi y$. Then $M^\varphi(x, y) = i < \infty$ and the above strategy allows Player 1 to

force Player 2 into a game instance $(x', y', \bar{\varphi})$ where $M^{\bar{\varphi}}(x', y') < M^{\varphi}(x, y)$. Whenever $M^{\varphi}(x, y) = 1$, Player 1 wins immediately, because then x allows a transition that y can not mimic or vice-versa, and Player 1 simply takes this transition. Therefore, we have found a winning strategy for Player 1. \square

To illustrate the game, we will show how Player 1 may act to prove that the states $ready_1$ and $ready_2$ from Example 6.2.4 are not bisimilar.

Example 6.4.16 *Consider the two systems from Example 6.2.4. A winning strategy for Player 1 starting in the states $ready_1$ and $ready_2$ and initial condition **b** could work as follows:*

- *Choose either $ready_1$ or $ready_2$, do not perform an upgrade and take the only possible transition labelled receive, leading to the state $received_1$, respectively $received_2$. Player 2 can only answer this step by doing the receive step in the other state. The new game situation is $(received_1, received_2, \mathbf{b})$.*
- *Player 1 now must choose state $received_1$ and perform no upgrade. Using the check transition, Player 1 does a step to the state $unsafe_1$. Player 2 again only has a single choice for his answering step, since no upgrade was performed and therefore, the transition to $unsafe_2$ is not available. Thus, Player 2 has to take the check transition to $safe_2$. So the new game situation is $(unsafe_1, safe_2, \mathbf{b})$.*
- *Now Player 1 can win the game by performing an upgrade to **a**, enabling Player 1 to do an e step from $unsafe_1$ to $ready_1$. Player 2 cannot answer this step, because in $safe_2$, no e transition is available.*

6.4.5 Deactivating Transitions

We will now introduce an extension, allowing to deactivate transitions when upgrading. This extension loses some of the mathematical elegance of the CTS without deactivation of transitions when upgrading, but in turn it is offering additional modelling opportunities while still allowing for a fixpoint iteration akin to the case without deactivation of transitions.

We have introduced conditional transition systems (CTS) as a modelling technique that allows to model a family of systems that are all based on a

common design, but with different actions available for different products. Products may be upgraded to advanced versions, activating additional transitions in the system. A change in the transition function can only be realised in one direction: by adding transitions which were previously not available, while all previously active transitions remain active.

However, this choice may not be the optimal choice in all cases, because sometimes an advanced version of a system may offer improved transitions over the base product. For instance, a free version of a system may display a commercial when choosing a certain transition, whereas a premium model may forego the commercial and offer the base functionality right away.

A practical motivation may be derived from our Example 6.2.2. In this transition system one may want to be able to model that in the unsafe state, the advanced version can only send an encrypted message, since we assume that the user is always interested in a secure communication, ensured either by a safe channel or by encryption. However, it is not an option to simply drop the unencrypted transition from the unsafe state with respect to the base version, because then, whenever the system encounters an unsafe state in the base version, the system will remain in a deadlock unless the user decides to perform an upgrade. We will solve such a situation as follows: we will add priorities that allow to deactivate the unencrypted transition in the presence of an encrypted transition.

In order to allow for deactivation of transitions when upgrading, we propose a slight variation of the definition of CTS/LaTS and the corresponding bisimulation relation.

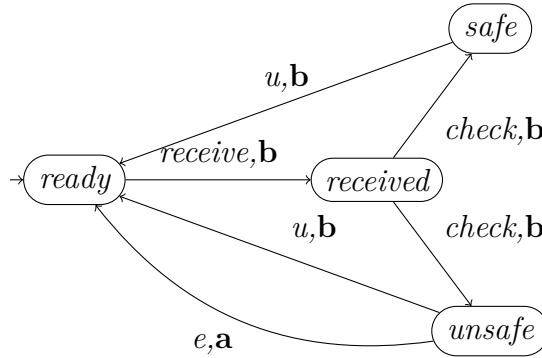
Definition 6.4.17 *A conditional transition system with action precedence is a triple $(X, (A, <_A), f)$, where (X, A, f) is a CTS and $<_A$ is a strict order on A .*

Intuitively, a CTS with action precedence evolves in a way very similar to standard CTS. Before the system starts acting, it is assumed that all the conditions are fixed and a condition $\varphi \in \Phi$ is chosen arbitrarily which represents a selection of a valid product of the system (product line). Now all the transitions that have a condition greater than or equal to φ are activated, while the remaining transitions are inactive. This is unchanged from standard CTS,

however, if from a state x there exist two transitions $x \xrightarrow{a,\varphi} x'$ and $x \xrightarrow{a',\varphi} x''$, where $a' > a$, i.e. a' takes precedence over a , then additionally $x \xrightarrow{a,\varphi} x'$ remains inactive. Henceforth, the system behaves like a standard transition system; until at any point in the computation, the condition is changed to a smaller one (say, φ') signifying a selection of a valid, upgraded product. Now (de)activation of transitions depends on the new condition φ' , rather than on the old condition φ . As before, active transitions remain active during an upgrade, *unless new active transitions appear that are exiting the same state and are labelled with an action of higher priority*.

In the sequel we will just write CTS for CTS with action precedence, since for the remainder of this section we will solely investigate this variation of CTS.

Example 6.4.18 We now refine our earlier Example 6.2.2 of an adaptive routing protocol modelled as a CTS over the alphabet $A = \{\text{receive}, \text{check}, u, e\}$ by adding an action precedence $<_A$ where $u <_A e$. The transition function remains unchanged, so does the visual representation:



The system retains two products: the basic system with no encryption feature written as **b** and the advanced system with encryption feature written as **a**. However, the action precedence changes the behaviour in the advanced version of the system. Under **a**, in state *unsafe*, it is not possible anymore to make a *u* (unencrypted) transition, since the alternative *e* (encrypted) transition has precedence over *u* and is available in state *unsafe* as well. Therefore, no non-deterministic choice may occur anymore in presence of *u* and *e* and the system model enforces that in *unsafe* environments, only encrypted messages may be sent. However, under product **b**, in state *unsafe*, the *u* transition remains active, since no *e* transition can take precedence over it. Similarly, also in state *safe*, for product **a**, the *u* transition remains active.

This changed interpretation of the behaviour of a CTS of course also has an effect on the bisimulation.

Definition 6.4.19 *Let $(X, (A, <_A), f)$, $(Y, (A, <_A), g)$ be two CTSs over the same set of conditions (Φ, \leq) . For a condition $\varphi \in \Phi$, we define $\bar{f}_\varphi(x, a)$ to denote the labelled transition system induced by a CTS $(X, (A, <_A), f)$ with action precedence, where*

$$\bar{f}_\varphi(x, a) = \{x' \mid x \xrightarrow{a, \varphi} x' \wedge \neg(\exists a' \in A, \exists x'' \in X : a' > a \wedge x \xrightarrow{a', \varphi} x'')\}.$$

Two states $x \in X, y \in Y$ are conditionally bisimilar (wrt. action precedence) under a condition $\varphi \in \Phi$, denoted $x \sim_\varphi^p y$, if there is a family of relations $R_{\varphi'}$ (for every $\varphi' \leq \varphi$) such that

(i) *each relation $R_{\varphi'}$ is a traditional bisimulation relation between $\bar{f}_{\varphi'}$ and $\bar{g}_{\varphi'}$,*

(ii) *whenever $\varphi' \leq \varphi''$, we have $R_{\varphi'} \supseteq R_{\varphi''}$, and*

(iii) *R_φ relates x and y , i.e. $(x, y) \in R_\varphi$.*

The definition of bisimilarity is analogous to traditional CTS but refers to the new transition system given by \bar{f} , which contains only the maximal transitions.

Lattice transition systems (LaTSs) can be extended in the same way, by adding an order on the set of actions and leaving the remaining definition unchanged. Disregarding deactivation, there still is a duality between CTSs and LaTSs. Now, in order to characterise bisimulation using a fixpoint operator, we can modify the operators F_1, F_2 and F to obtain G_1, G_2 and G , respecting the deactivation of transitions as follows.

Definition 6.4.20 *Let $(X, (A, <_A), \alpha)$ and $(Y, (A, <_A), \beta)$ be LaTSs (with ordered actions). Recall the residuum operator (\rightarrow) on a lattice and define three*

operators $G, G_1, G_2: (X \times Y \rightarrow \mathbb{L}) \rightarrow (X \times Y \rightarrow \mathbb{L})$ in the following way:

$$\begin{aligned}
& G_1(R)(x, y) \\
&= \bigsqcap_{a \in A, x' \in X} \left(\alpha(x, a, x') \rightarrow \left(\bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y')) \sqcup \bigsqcup_{a' > a, x'' \in X} \alpha(x, a', x'') \right) \right), \\
& G_2(R)(x, y) \\
&= \bigsqcap_{a \in A, y' \in Y} \left(\beta(y, a, y') \rightarrow \left(\bigsqcup_{x' \in X} (\alpha(x, a, x') \sqcap R(x', y')) \sqcup \bigsqcup_{a' > a, y'' \in Y} \beta(y, a', y'') \right) \right), \\
& G(R)(x, y) = G_1(R)(x, y) \sqcap G_2(R)(x, y).
\end{aligned}$$

Now, we need to show that we can characterise the new notion of bisimulations as post-fixpoints of this operator G . For the corresponding proof we will make use of the following observation:

Lemma 6.4.21 *Let $\mathbb{L} = \mathcal{O}(\Phi)$ for any finite partially ordered set (Φ, \leq) be a lattice that embeds into $\mathbb{B} = \mathcal{P}(\Phi)$. Take $\varphi \in \Phi$. Then, in order to show that $\varphi \in (l_1 \rightarrow l_2)$, for any given $l_1, l_2 \in \mathbb{L}$, it suffices to show that for all $\varphi' \leq \varphi$, $\varphi' \notin l_1$ or $\varphi' \in l_2$.*

Proof: We have already shown that $l_1 \rightarrow l_2 = \lfloor l_1 \rightarrow l_2 \rfloor = \lfloor \neg l_1 \sqcup l_2 \rfloor$. Now, if all $\varphi' \leq \varphi$ are not in l_1 , i.e. in $\neg l_1$, or in l_2 , then all $\varphi' \leq \varphi$ are in $\neg l_1 \sqcup l_2$. Therefore, $\downarrow \varphi \subseteq \neg l_1 \sqcup l_2$, and thus $\varphi \in l_1 \rightarrow l_2$. \square

Theorem 6.4.22 *Let $(X, (A, <_A), f)$, $(Y, (A, <_A), g)$ be two CTSs over (Φ, \leq) and $(X, (A, <_A), \alpha)$, $(Y, (A, <_A), \beta)$ over $\mathcal{O}(\Phi)$ be the corresponding LaTS. For any two states $x \in X, y \in Y$ it holds that $x \sim_\varphi^p y$ if and only if there exists a post-fixpoint $R: X \times Y \rightarrow \mathbb{L}$ of G ($R \sqsubseteq G(R)$) such that $\varphi \in R(x, y)$.*

Proof:

- Assume R is a post-fixpoint of G , i.e. $R \sqsubseteq G(R)$, let $x \in X$ and $y \in Y$ be given arbitrarily and $\varphi \in R(x, y)$. We define for each $\varphi' \leq \varphi$ a relation $R_{\varphi'}$ according to

$$(x', y') \in R_{\varphi'} \Leftrightarrow \varphi' \in R(x', y').$$

Since each set $R(x', y')$ is downward-closed for all $x' \in X, y' \in Y$, it holds that $R_{\varphi_1} \subseteq R_{\varphi_2}$ whenever $\varphi_1 \geq \varphi_2$. Moreover, since we assume $\varphi \in R(x, y)$, $(x, y) \in R_{\varphi'}$ must hold for all $\varphi' \leq \varphi$.

So we only need to show that all $R_{\varphi'}$ are traditional bisimulations for $\bar{f}_{\varphi'}$. For this purpose let x', y', φ' be given, such that $(x', y') \in R_{\varphi'}$. Moreover, let $a \in A$ and $x'' \in X$ be given such that $x'' \in \bar{f}_{\varphi'}(x', a)$ – if no such a and x'' exists then the first bisimulation condition is trivially true. For G_1 , it must be true that $\varphi' \in G_1(R)(x, y)$. Thus, φ' must also be contained in

$$\alpha(x', a, x'') \rightarrow \left(\bigsqcup_{y'' \in Y} (\beta(y', a, y'') \sqcap R(x'', y'')) \sqcup \bigsqcup_{a' > a, x''' \in X} \alpha(x', a', x''') \right).$$

Since we also know that $\varphi' \in \alpha(x', a, x'')$, because $x'' \in \bar{f}_{\varphi'}(x', a)$, it must be true that

$$\varphi' \in \left(\bigsqcup_{y'' \in Y} (\beta(y', a, y'') \sqcap R(x'', y'')) \sqcup \bigsqcup_{a' > a, x''' \in X} \alpha(x', a', x''') \right).$$

This is true, because $\psi \in l_1 \rightarrow l_2 \Leftrightarrow \psi \in [\neg l_1 \vee l_2] \Rightarrow \psi \in \neg l_1 \vee l_2$ (Lemma 6.4.21) and, if $\psi \in l_1$, hence $\psi \notin \neg l_1$, it follows that $\psi \in l_2$.

Per definition of $\bar{f}_{\varphi'}$, there exists no $a' > a$ such that $\bar{f}_{\varphi'}(x'', a') \neq \emptyset$. Therefore,

$$\varphi' \notin \bigsqcup_{a' > a, x''' \in X} \alpha(x', a', x''').$$

It follows that

$$\varphi' \in \bigsqcup_{y'' \in Y} \beta(y', a, y'') \sqcap R(x'', y'').$$

Then, there must exist at least one $y'' \in Y$ such that $\varphi' \in \beta(y', a, y'') \sqcap R(x'', y'')$. It follows that $\varphi' \in R(x'', y'')$, i.e. $(x'', y'') \in R_{\varphi'}$.

We will now show that $y'' \in \bar{g}_{\varphi'}(y', a)$, holds as well. Assume, to the contrary, that $y'' \notin \bar{g}_{\varphi'}(y', a)$, then, due to $\varphi' \in \beta(y', a, y'')$, there must exist an $a' > a$ and a $y''' \in Y$ such that $\varphi' \in \beta(y', a', y''')$. W.l.o.g. choose a' maximal. Since we required $(x', y') \in R_{\varphi'}$, it has to hold that $\varphi' \in G_2(R)(x', y')$. So in particular, φ' must be contained in

$$\beta(y', a', y''') \rightarrow \left(\bigsqcup_{x''' \in X} (\alpha(x', a', x''') \sqcap R(x''', y''')) \sqcup \bigsqcup_{a'' > a', y'''' \in Y} \beta(y', a'', y'''') \right)$$

Since we chose a' maximal, we know that $\varphi' \notin \bigsqcup_{a'' > a', y'''' \in Y} \beta(y', a'', y'''')$. Moreover, since $a' > a$ and $x'' \in \bar{f}_{\varphi'}(x', a)$, there exists no x''' such that $\varphi' \in \alpha(x', a', x''')$. Thus, φ' is not in the right side of the residuum, yet

it is in the left side of the residuum, therefore, it is not in the residuum. Thus, we can conclude $\varphi' \notin G_2(R)(x', y')$, which is a contradiction.

Thus, the first bisimulation condition is true. The second condition can be proven analogously, reversing the roles of G_2 and G_1 to find the answer step in $\bar{f}_{\varphi'}$.

- Now, assume the other way around, that a family R_{φ} of bisimulations from \bar{f}_{φ} to \bar{g}_{φ} exists such that for all states $x \in X$, $y \in Y$ and for all pairs of conditions $\varphi_1, \varphi_2 \in \Phi$ the expression $\varphi_1 \leq \varphi_2$ implies $R_{\varphi_1} \supseteq R_{\varphi_2}$. Moreover, let φ , $x \in X$ and $y \in Y$ be given such that $(x, y) \in R_{\varphi}$. We define $R: X \times Y \rightarrow \mathbb{L}$ according to

$$R(x, y) = \{\varphi' \mid (x, y) \in R_{\varphi'}\}.$$

Due to anti-monotonicity of the family of $R_{\varphi'}$ all entries in R are indeed lattice elements from $\mathcal{O}(\Phi, \leq)$. Moreover, by definition, $\varphi \in R(x, y)$. So it only remains to be shown that R is a post-fixpoint.

For this purpose, let $x' \in X$, $y' \in Y$ and $\varphi' \in \Phi$ be given, such that $\varphi' \in R(x', y')$. (If no such x', y', φ' exist, then R is the zero matrix (where all entries are \emptyset) and $R \sqsubseteq G(R)$ holds trivially.) We will now show that $\varphi' \in G_1(R)(x', y')$. The fact that $\varphi' \in G_2(R)(x', y')$ can be shown analogously. We need to show that

$$\varphi' \in \left(\alpha(x, a, x') \rightarrow \left(\bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y')) \sqcup \bigsqcup_{a' > a, x'' \in X} \alpha(x, a', x'') \right) \right)$$

for all $x'' \in X$ and $a \in A$.

We recall that $l_1 \rightarrow_{\mathbb{L}} l_2 = [l_1 \rightarrow_{\mathbb{B}} l_2] = [\neg l_1 \vee l_2]$ (Lemma 2.2.18) and show that whenever $\varphi' \in \alpha(x, a, x')$, it holds that $\varphi' \in \left(\bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y')) \sqcup \bigsqcup_{a' > a, x'' \in X} \alpha(x, a', x'') \right)$. We distinguish according to whether a is maximal such that $\varphi' \in \alpha(x, a, x'')$:

- There is no $a' > a$ such that $\varphi' \in \alpha(x, a', x'')$ for any $x'' \in X$:

Then there must exist a $y' \in Y$ such that $\varphi' \in \beta(y, a, y')$ and $(x', y') \in R_{\varphi'}$, i.e. $\varphi' \in R(x', y')$, because $R_{\varphi'}$ is a bisimulation and for all $\varphi'' \leq \varphi'$ we have $R_{\varphi''} \subseteq R_{\varphi'}$.

– There is an $a' > a$ such that $\varphi' \in \alpha(x, a, x'')$ for some $x'' \in X$:

Then $\varphi' \in \bigsqcup_{a' > a, x'' \in X} \alpha(x, a', x'')$.

So we have shown for all $\varphi' \in R(x', y')$ that $\varphi' \in \alpha(x, a, x')$ implies

$$\varphi' \in \left(\bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y')) \sqcup \bigsqcup_{a' > a, x'' \in X} \alpha(x, a', x'') \right),$$

i.e. we have

$$\varphi' \in \neg \alpha(x, a, x') \sqcup \left(\bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y')) \sqcup \bigsqcup_{a' > a, x'' \in X} \alpha(x, a', x'') \right)$$

in the Boolean algebra. Since $R(x', y')$ is a lattice element and therefore downward-closed, we can apply Lemma 6.4.21 and conclude that

$$\varphi' \in \left(\alpha(x, a, x') \rightarrow \left(\bigsqcup_{y' \in Y} (\beta(y, a, y') \sqcap R(x', y')) \sqcup \bigsqcup_{a' > a, x'' \in X} \alpha(x, a', x'') \right) \right)$$

in the lattice, concluding the proof.

□

Hence we can compute the bisimulation via a fixpoint iteration, as with LaTS without an ordering on the labels. Due to the additional supremum in the fixpoint operator, the matrix notation cannot be used anymore. However, since the additional supremum term can be precomputed for each pair of states $x \in X$ or $y \in Y$ and action $a \in A$, the performance of the algorithm should not be affected in a significant way.

Note that, different from the Boolean case, $l_1 \rightarrow (l_2 \sqcup l_3) \not\equiv (l_1 \rightarrow l_2) \sqcup l_3$, which is relevant for the definition of G . In fact, moving the supremum $\bigsqcup_{a' > a, x'' \in X} \alpha(x, a', x'')$ outside of the residuum would yield an incorrect notion of bisimilarity.

In addition, it may appear more convenient to drop the monotonicity requirement for transitions and to allow arbitrary deactivation of transitions, independently of their label. However, this would result in a loss of the duality property and as a result, the fixpoint algorithm that allows to compute the bisimilarity in parallel for all products would be rendered incorrect.

6.5 Application to Software Product Lines

6.5.1 Featured Transition Systems

A Software Product Line (SPL) is commonly described as “a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets [artifacts] in a prescribed way” [CN01]. The idea of designing a set of software systems that share common functionalities in a collective way is becoming prominent in the field of software engineering (cf. [MP14]). In this section, we show that featured transition system (FTS) – a well-known formal model that is expressive enough to specify an SPL – is a special instance of a CTS. We begin by giving the definition of an FTS (taken from [AFL15]).

Definition 6.5.1 *A featured transition system (FTS) over a finite set of features N is a tuple $\mathcal{F} = (X, A, T, \gamma)$, where X is a finite set of states, A is a finite set of actions and $T \subseteq X \times A \times X$ is the set of transitions. Finally, $\gamma: T \rightarrow \mathbb{B}(N)$ assigns a Boolean expression over N to each transition.*

FTSs are often accompanied by a so-called *feature diagram* [CCH⁺13, CHS⁺10, CCS⁺13], a Boolean expression $d \in \mathbb{B}(N)$ that specifies admissible feature combinations. Given a subset of features $C \subseteq N$ (called *configuration* or *product*) such that $C \models d$ and an FTS $\mathcal{F} = (X, A, T, \gamma)$, a state $x \in X$ can perform an a -transition to a state $y \in X$ in the configuration C , whenever $(x, a, y) \in T$ and $C \models \gamma(x, a, y)$.

It is easy to see that an FTS is a CTS, where the conditions are subsets of N satisfying d with the discrete order. Moreover, an FTS can also be seen as a special case of a LaTS due to Theorem 6.3.2 and $\mathcal{O}(\llbracket d \rrbracket, =) = \mathcal{P}(\llbracket d \rrbracket)$. Given an FTS $\mathcal{F} = (X, A, T, \gamma)$ and a feature diagram d , then the corresponding LaTS is (X, A, α) , where the function α is defined as follows: $\alpha(x, a, y) = \llbracket \gamma(x, a, y) \wedge d \rrbracket$, if $(x, a, y) \in T$; $\alpha(x, a, y) = \emptyset$, if $(x, a, y) \notin T$.

Furthermore, we can extend the notion of FTSs by fixing a subset of upgrade features $U \subseteq N$ that induces the following ordering on configurations $C, C' \in \llbracket d \rrbracket$:

$$C \leq C' \iff \forall f \in U (f \in C' \Rightarrow f \in C) \wedge \forall f \in (N \setminus U) (f \in C' \iff f \in C).$$

Intuitively, the configuration C can be obtained from C' by “switching” on one or several upgrade features $f \in U$. Notice that it is this upgrade ordering on configurations which gives rise to the partially ordered set of conditions in the definition of a CTS. Hence, in the sequel we will consider the lattice $\mathcal{O}(\llbracket d \rrbracket, \leq)$ (i.e., the set of all downward-closed subsets of $\llbracket d \rrbracket$).

6.5.2 BDDs as Models for Boolean Formulae

In this section, we discuss our implementation of the lattice bisimulation check using a special form of binary decision diagrams (BDDs) called *reduced and ordered binary decision diagrams* (ROBDDs). Our implementation can handle adaptive SPLs that allow upgrade features, using finite distributive lattices. Non-adaptive SPLs, based on Boolean algebras are a special case. BDD-based implementations of FTSs without upgrades have already been discussed in [CCP⁺12].

A *binary decision diagram* (BDD) is a rooted, directed, and acyclic graph which serves as a representation of a Boolean function. Every BDD has two distinguished terminal nodes 1 and 0, representing the logical constants *true* and *false*. The inner nodes are labelled by the atomic propositions of a Boolean expression $f \in \mathbb{B}(N)$ represented by the BDD, such that on each path from the root to the terminal nodes, every variable of the Boolean formula occurs at most once. Each inner node has exactly two distinguished outgoing edges called *high* and *low* representing the case that the atomic propositions of the inner node has been set to *true* or *false*. Given a BDD b for a Boolean expression $f \in \mathbb{B}(N)$ and a configuration $C \subseteq N$ (representing an evaluation of the atomic propositions), we can check whether $C \models b$ by following the path from the root node to a terminal node, where we choose the high-successor whenever the label of a node is in C and the low-successor otherwise. If we arrive at the terminal node labelled 1 we have established that $C \models f$, otherwise $C \not\models f$.

We will use a special class of BDDs, called *reduced and ordered BDDs* (ROBDDs) – full details in [And97] – in which the order of the variables occurring in the BDD is fixed and redundancy is avoided. If both the child

nodes of a parent node are identical, the parent node is dropped from the BDD and isomorphic parts of the BDD are merged. The advantage of ROBDDs is that two equivalent Boolean formulae are represented by exactly the same ROBDD (if the order of the variables is fixed). Furthermore, there are simple polynomial-time implementations for the basic operations – negation, conjunction, and disjunction. These are however sensitive to the ordering of atomic propositions and an exponential blow-up cannot be ruled out, but often it can be avoided.

We will give a short introduction to ROBDDs and describe how Boolean operators can be realised with ROBDDs.

First we will give an idea how Boolean formulae can be represented by ROBDDs and how to define ROBDDs used to represent formulae in a Boolean algebra:

Definition 6.5.2 (ROBDD)

- *A BDD b over a set of variables N is a node-labelled rooted directed acyclic graph. Each node of said graph has either outdegree of exactly two and is labelled by a variable in N or it has outdegree zero – the leaves of the BDD – and is labelled either 0 or 1. The successor nodes of any inner node v are called $\text{high}(v)$ and $\text{low}(v)$, respectively, and the root of the BDD is called $\text{root}(b)$. A BDD b can be written as an expression in one of the following forms: 0, or 1, or (f, b_1, b_0) , where $f \in N$ and b_0 and b_1 is the low- and high-successors, respectively.*
- *Given a configuration $C \subseteq N$, one can obtain the value of a BDD b over C by traversing the BDD starting from the root v as follows. For this purpose, we write $v(C)$ to denote the value of the tree represented by v under C .*
 - *if v is the 1-leaf, $v(C) = 1$*
 - *if v is the 0-leaf, $v(C) = 0$*
 - *if v is not a leaf we need to distinguish, if the label of v is in C or not. If it is in C , $v(C) = \text{high}(v)(C)$, otherwise $v(C) = \text{low}(v)(C)$*
- *If we put an order \leq on the variables and for each edge in a BDD b , it holds that the label of the successor nodes is less than the label of the*

predecessor node, we call b an OBDD (wrt \leq). An OBDD b is reduced, if it is minimal among all OBDDs for the same Boolean function that respect the same order, we then call b an ROBDD. An OBDD can be transferred into an ROBDD by repeatedly removing vertices where both successors are the same vertex and merging vertices that share a common label, as well as their high- and low-successors.

Moreover, if one implements ROBDDs as shared ROBDDs, where each ROBDD is represented as a node in one big graph, equivalence checking is particularly simple, because then, two formula are equal if and only if they are represented by the same node in the shared ROBDD. We will now give a brief overview over the operations to realise the three Boolean operators. Note that in this description we will distinguish between ROBDDs and the logical functions they represent explicitly.

- \neg : Given an ROBDD b representing a Boolean function f , the ROBDD representing $\neg f$ can be constructed by just flipping 0 and 1 nodes in b .
- \vee / \wedge : For those two binary operators, one recursive operation exists that can be used to implement them in time linear in the size of the two BDDs it gets applied to. Note, however, that the result is not reduced, an additional reduction step is needed afterwards. Ordering is being preserved by the operation, though.

Assume ROBDDs b_1 representing the logical formula f and b_2 , representing the logical formula g are given. Moreover, let an operation $\text{op} \in \{\vee, \wedge\}$ be given. We want to find an ROBDD b that represents the formula $f \text{ op } g$.

We will not describe the operation in detail, but only explain the general idea. Given a binary operation $\text{op} \in \{\vee, \wedge\}$, we apply, in a top-down manner, the following expansion law, which holds for any given variable x :

$$f \text{ op } g = \neg x \wedge (f|_{x=0} \text{ op } g|_{x=0}) \vee x \wedge (f|_{x=1} \text{ op } g|_{x=1})$$

For a logical formula f , we write $f|_{x=0}$ ($f|_{x=1}$), to denote the formula that arises if each occurrence of x is replaced by 0 (1), which amounts to a partial evaluation of f .

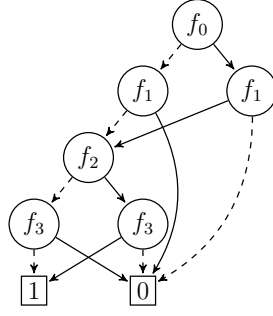
This expansion can be applied by taking into consideration that the high-successor of a node labelled by variable x representing a formula f is a node representing the formula $f|_{x=1}$ and the low successor represents the formula $f|_{x=0}$. We compute the desired ROBDD using the following recursive procedure: If $\text{root}(b_1)$ and $\text{root}(b_2)$ are terminal, the resulting BDD can directly be computed. If $\text{root}(b_1) = \text{root}(b_2)$, we create a node with label $\text{root}(b_1)$ and apply op recursively on $\text{low}(b_1)$ and $\text{low}(b_2)$ for its low successor, as well as $\text{high}(b_1)$ and $\text{high}(b_2)$ for its high successor. If $\text{root}(b_1) > \text{root}(b_2)$ or b_2 is a terminal node (analogously the other way around), we can apply the same inductive step with b_2 taking the part of both, $\text{low}(b_2)$ and $\text{high}(b_2)$, because then the value of b_2 is independent of $\text{root}(b_1)$.

Computing the ROBDD b that represents $f \text{ op } g$, $\text{op} \in \{\vee, \wedge\}$, in this manner takes time $\mathcal{O}(|b_1| \cdot |b_2|)$.

- Reduction: Given an OBDD b , one can reduce it via a recursive procedure that takes $\mathcal{O}(|b| \cdot \log |b|)$ time. We will not present the algorithm in detail here, the idea is to work through the BDD bottom-up and put an additional label on each node. For two nodes these labels should be identical if and only if the nodes represent the same Boolean function. For the terminal nodes, this works in the obvious way, for the other nodes, we can use a hash table and the labels of the succeeding nodes which have been computed earlier. When the labelling is complete, one can construct from the original OBDD an ROBDD in which one node per label exists and edges are being instated according to the now-labelled graph.

ROBDDs are commonly used as means to represent Boolean algebra elements because they give rise to efficient implementations of the operations \wedge , \vee and \neg .

Consider a Boolean formula f with $\llbracket f \rrbracket = \{\emptyset, \{f_2, f_3\}, \{f_0, f_1\}, \{f_0, f_1, f_2, f_3\}\}$ and the ordering on the atomic propositions as f_0, f_1, f_2, f_3 . Figure 6.1 shows the ROBDD for f , where the inner nodes, terminal nodes, and high (low) edges are depicted as circles, rectangles, and solid (dashed) lines, respectively.

Figure 6.1: BDD for a Boolean expression f .

Note that the elements of the Boolean algebra $\mathcal{P}(\mathcal{P}(N))$ *correspond exactly* to ROBDDs over N .

6.5.3 BDDs for Lattices

We now discuss how ROBDDs can also be used to specify and manipulate elements of the lattice $\mathcal{O}(\llbracket d \rrbracket, \leq)$. In particular, computing the infimum and the supremum in the lattice $\mathcal{O}(\llbracket d \rrbracket, \leq)$ is standard, since this lattice can be embedded into $\mathcal{P}(\mathcal{P}(N))$ and the infimum and supremum operations coincide in both structures. Therefore, it remains to represent the lattice elements and the residuum.

We say that an ROBDD b is *downward-closed* with respect to \leq (or simply, downward-closed) whenever the set of configurations $\llbracket b \rrbracket$ is downward-closed with respect to \leq . The following lemma characterises when an ROBDD b is downward closed. It follows from the fact that $F \in \mathcal{P}(\mathcal{P}(N))$ is downward-closed if and only if for all $C \in F, f \in U$ ($C \cup \{f\} \in F$).

Lemma 6.5.3 *Let $U \subseteq N$ be a set of upgrade features. An ROBDD b over N is downward-closed if and only if for each node labelled with an element of U , the low-successor implies the high-successor.*

Proof:

\Rightarrow Assume that $\text{low}(n) \models \text{high}(n)$ for all nodes n of b .

Let $C' \in \llbracket b \rrbracket$ and $C \leq C'$. Without loss of generality we can assume that $C = C' \cup \{f\}$ for some $f \in U$. (The rest follows from transitivity.) For the configuration C' there exists a path in b that leads to 1. We distinguish the following two cases:

- There is no f -labelled node on the path. Then the path for C also leads to 1 and we have $C \in \llbracket b \rrbracket$.
- If there is an f -labelled node n on the path, then C' takes the *low*-successor, C the *high*-successor of this node. Since $low(n) \models high(n)$, we obtain $\llbracket low(n) \rrbracket \subseteq \llbracket high(n) \rrbracket$. Hence, the remaining path for C , which contains the same features as the path for C' , will also reach 1.

\Leftarrow Assume by contradiction that $\llbracket b \rrbracket$ is downward-closed, but there exists a node n with $low(n) \not\models high(n)$ and $f = root(n) \in U$. Hence, there must be a path from the *low*-successor that reaches 1, but does not reach 1 from the *high*-successor. Prefix this with the path that reaches n from the root of b .

In this way we obtain two configurations $C = C' \cup \{f\}$, i.e., $C \leq C'$, where $C' \in \llbracket b \rrbracket$, but $C \notin \llbracket b \rrbracket$. This is a contradiction to the fact that $\llbracket b \rrbracket$ is downward-closed.

□

The next step is to compute a residuum in $\mathcal{O}(\llbracket d \rrbracket, \leq)$ by using the residuum operation of the Boolean algebra $\mathcal{P}(\mathcal{P}(N))$. For this, we first describe how to approximate an element of the Boolean algebra (equivalently represented as an ROBDD) in the lattice $\mathcal{O}(\mathcal{P}(N), \leq)$.

Algorithm 6.5.4 Approximation $\llbracket b \rrbracket$ of an ROBDD b in $\mathcal{O}(\mathcal{P}(N), \leq)$

Input: An ROBDD b over a set of features N and a set of upgrade features $U \subseteq N$.

Output: An ROBDD $\llbracket b \rrbracket$, which is the best approximation of b in the lattice.

$\llbracket b \rrbracket$

```

1 if  $b$  is a leaf then return  $b$ 
2 if  $root(b) \in U$  then
3   return  $build(root(b), \llbracket high(b) \rrbracket, \llbracket high(b) \rrbracket \wedge \llbracket low(b) \rrbracket)$ 
4 return  $build(root(b), \llbracket high(b) \rrbracket, \llbracket low(b) \rrbracket)$ 

```

In the above algorithm, for each non-terminal node that carries a label in U (line 3), we replace the *high*-successor with the conjunction of the *low* and the *high*-successor using the procedure described above. Since this might result in

a BDD that is not reduced, we apply the *build* procedure appropriately, which simply transforms a given ordered BDD into an ROBDD.

The result of the algorithm $\llbracket b \rrbracket$ coincides with the approximation $\lfloor b \rfloor$ of the ROBDD b seen as an element of the Boolean algebra $\mathcal{P}(\mathcal{P}(N))$ (Definition 2.2.17).

Lemma 6.5.5 *Let b be an ROBDD. Then $\llbracket b \rrbracket$ is downward-closed. Furthermore, $\llbracket b \rrbracket \models b$ and there is no other downward-closed ROBDD b' such that $\llbracket b \rrbracket \models b' \models b$. Hence, $\llbracket b \rrbracket = \lfloor b \rfloor$.*

Proof:

- We show that $\llbracket b \rrbracket$ as obtained by Algorithm 6.5.4 is downward-closed. This can be seen via induction over the number of different features occurring in the BDD b . If b only consists of a leaf node, then $\llbracket b \rrbracket$ is certainly downward-closed. Otherwise, we know from the induction hypothesis that $\llbracket \text{high}(b) \rrbracket, \llbracket \text{low}(b) \rrbracket$ are downward-closed. If $\text{root}(b) \notin U$, then $\llbracket b \rrbracket$ is downward-closed due to Lemma 6.5.3. If, however, $\text{root}(b) \in U$, then $\llbracket \text{high}(b) \rrbracket \wedge \llbracket \text{low}(b) \rrbracket$ is downward-closed (since downward-closed sets are closed under intersection). Furthermore $\llbracket \text{high}(b) \rrbracket \wedge \llbracket \text{low}(b) \rrbracket \models \llbracket \text{high}(b) \rrbracket$, i.e., the new *low*-successor implies the *high*-successor. That means that the condition of Lemma 6.5.3 is satisfied at the root and elsewhere in the BDD and hence, the resulting BDD $\llbracket b \rrbracket$ is downward-closed.
- First, from the construction where a *low*-successor is always replaced by a stronger *low*-successor, it is easy to see that $\llbracket b \rrbracket \models b$.

We now show that there is no other downward-closed ROBDD b' such that $\llbracket b \rrbracket \models b' \models b$: Assume to the contrary, that there exists such a downward-closed BDD b' . Hence, there exists a configuration $C \subseteq N$, such that $C \not\models \llbracket b \rrbracket$, $C \models b'$, $C \models b$. Choose C maximal wrt. inclusion.

Now we show that there exists a feature $f \in U$ such that $f \notin C$ and $C \cup \{f\} = C' \not\models b$. If this is the case, then $C' \leq C$ and $C' \not\models b'$, which is a contradiction to the fact that b' is downward-closed.

Consider the sequence $b = b_0, \dots, b_m = \llbracket b \rrbracket$ of BDDs that is constructed by the approximation algorithm (Algorithm 6.5.4), where the BDD structure

is upgraded bottom-up. We have $\llbracket b \rrbracket = b_m \models b_{m-1} \models \dots \models b_0 = b$, since in each newly constructed BDD for some node n $low(n)$ with $root(n) \in U$ is replaced by $high(n) \wedge low(n)$.

Since $C \models b$ and $C \not\models \llbracket b \rrbracket$, there must be an index k , such that $C \models b_k$, $C \not\models b_{k+1}$. Let n be the node that is modified in step k , where $root(n) = f \in U$. We must have $f \notin C$, since the changes concern only the *low*-successor and if $f \in C$, the corresponding path would take the *high*-successor and nothing would change concerning acceptance of C from b_k to b_{k+1} .

Now assume that $C' = C \cup \{f\} \models b$. This would be a contradiction to the maximality of C and hence $C \cup \{f\} \not\models b$, as required.

□

For each node in the BDD we compute at most one supremum, which is quadratic. Hence the entire run-time of the approximation procedure is at most cubic. Finally, we discuss how to compute the residuum in $\mathcal{O}(\llbracket d \rrbracket, \leq)$.

Proposition 6.2 *Let b_1, b_2 be two ROBDD which represent elements of $\mathcal{O}(\llbracket d \rrbracket, \leq)$, i.e., b_1, b_2 are both downward-closed and $b_1 \models d$, $b_2 \models d$.*

(i) $\lfloor \neg b_1 \vee b_2 \vee \neg d \rfloor \wedge d$ is the residuum $b_1 \rightarrow b_2$ in the lattice $\mathcal{O}(\llbracket d \rrbracket, \leq)$.

(ii) If d is downward-closed, then this simplifies to $b_1 \rightarrow b_2 = \lfloor \neg b_1 \vee b_2 \rfloor \wedge d$.

Here, the negation operation is the negation in the Boolean algebra $\mathcal{P}(\mathcal{P}(N))$.

Proof:

- (i) For this proof, we work with the set-based interpretation, which allows for four views, one on the Boolean algebra $\mathbb{B} = \mathcal{P}(\mathcal{P}(N))$, one on the lattice $\mathbb{L} = \mathcal{O}(\mathcal{P}(N), \leq)$, one of the Boolean algebra $\mathbb{B}' = \mathcal{P}(\llbracket d \rrbracket)$ and one on the lattice $\mathbb{L}' = (\mathcal{O}(\llbracket d \rrbracket), \leq')$ where $\leq' = \leq|_{\llbracket d \rrbracket \times \llbracket d \rrbracket}$. We will mostly argue in the Boolean algebra \mathbb{B} . When talking about downward-closed sets, we will usually indicate with respect to which order. Similarly, the approximation relative to \leq is written $\lfloor _ \rfloor$, whereas the approximation relative to \leq' is written $\lfloor _ \rfloor'$.

We can compute:

$$b_1 \rightarrow_{\mathbb{L}'} b_2 \equiv \lfloor \neg_{\mathbb{B}'} b_1 \vee b_2 \rfloor' \equiv \lfloor (\neg_{\mathbb{B}} b_1 \wedge d) \vee b_2 \rfloor'$$

To conclude the proof, we will now show that $\lfloor b \rfloor' \equiv \lfloor b \vee \neg d \rfloor \wedge d$ for any $b \in \mathbb{B}'$. We prove this via mutual implication.

- We show $\lfloor b \vee \neg d \rfloor \wedge d \models \lfloor b \rfloor'$:

$$\lfloor b \vee \neg d \rfloor \wedge d \models (b \vee \neg d) \wedge d \equiv (b \wedge d) \vee (\neg d \wedge d) \equiv b \wedge d \models b$$

Since $\lfloor b \vee \neg d \rfloor \wedge d$ implies d , it certainly is in \mathbb{B}' . We now show that it is downward-closed wrt. \leq' : we use an auxiliary relation \leq'' , which is the smallest partial order on \mathbb{B} that contains \leq' , i.e., \leq' extended to \mathbb{B} . We have $\leq'' \subseteq \leq$. Since $\lfloor b \vee \neg d \rfloor$ is an approximation, it is downward-closed wrt. \leq and hence downward-closed wrt. \leq' . Moreover, d is downward-closed relative to \leq'' (obvious by definition). Since the intersection of two downward-closed sets is again downward-closed, $\lfloor b \vee \neg d \rfloor \wedge d$ is downward-closed relative to \leq'' and since finally, downward-closure relative to \leq'' is the same as downward-closure relative to \leq' , provided we discuss an element from \mathbb{B}' , we can conclude that $\lfloor b \vee \neg d \rfloor \wedge d$ belongs to \mathbb{L}' .

From $\lfloor b \vee \neg d \rfloor \wedge d \in \mathbb{L}'$ and $\lfloor b \vee \neg d \rfloor \wedge d \models b$ it follows that $\lfloor b \vee \neg d \rfloor \wedge d \models \lfloor b \rfloor'$ by definition of the approximation.

- We show $\lfloor b \rfloor' \models \lfloor b \vee \neg d \rfloor \wedge d$:

Let any $C \in \mathcal{P}(N)$ be given, such that $C \in \llbracket \lfloor b \rfloor' \rrbracket$. We show that in this case $C \in \llbracket \lfloor b \vee \neg d \rfloor \wedge d \rrbracket$, which proves $\lfloor b \rfloor' \models \lfloor b \vee \neg d \rfloor \wedge d$. Let $\downarrow C$ be the downwards-closure of C wrt. \leq .

Since $\lfloor b \rfloor'$ must be downward-closed relative to \leq' , it holds that $\downarrow C \cap \llbracket d \rrbracket \subseteq \llbracket \lfloor b \rfloor' \rrbracket$. Disjunction with $\neg d$ on both sides yields $\downarrow C \subseteq \llbracket \lfloor b \rfloor' \vee \neg d \rrbracket \subseteq \llbracket b \vee \neg d \rrbracket$, since $c \models c \vee \neg d \equiv (c \wedge d) \vee \neg d$. The set $\downarrow C$ is downwards-closed wrt. \leq , so it is contained in the approximation relative to \leq of this set, i.e. $\downarrow C \subseteq \llbracket \lfloor b \vee \neg d \rfloor \rrbracket$. Thus, in particular, $C \in \llbracket \lfloor b \vee \neg d \rfloor \rrbracket$. Since $C \in \llbracket \lfloor b \rfloor' \rrbracket$, it follows that $C \in \llbracket d \rrbracket$, therefore we can conclude $C \in \llbracket \lfloor b \vee \neg d \rfloor \wedge d \rrbracket$.

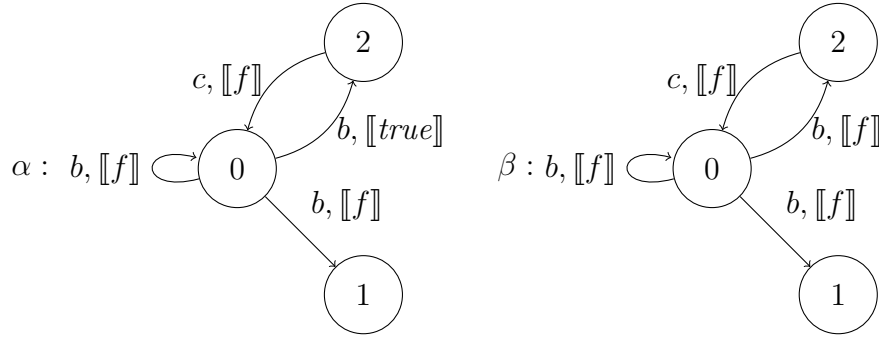


Figure 6.2: Components for α and β , where f is viewed as a Boolean expression indicating its presence.

Hence,

$$\lfloor (\neg_{\mathbb{B}} b_1 \wedge d) \vee b_2 \rfloor' \equiv \lfloor (\neg_{\mathbb{B}} b_1 \wedge d) \vee b_2 \vee \neg d \rfloor \wedge d \equiv \lfloor \neg_{\mathbb{B}} b_1 \vee b_2 \vee \neg d \rfloor \wedge d.$$

- (ii) Since d is downward-closed wrt. \leq , $d = \lfloor d \rfloor$, therefore, using Lemma 2.2.18, we obtain $\lfloor \neg b_1 \vee b_2 \vee \neg d \rfloor \wedge d \equiv \lfloor \neg b_1 \vee b_2 \vee \neg d \rfloor \wedge \lfloor d \rfloor \equiv \lfloor (\neg b_1 \vee b_2 \vee \neg d) \wedge d \rfloor \equiv \lfloor (\neg b_1 \vee b_2) \wedge d \vee \neg d \wedge d \rfloor \equiv \lfloor (\neg b_1 \vee b_2) \wedge d \rfloor \equiv \lfloor \neg b_1 \vee b_2 \rfloor \wedge \lfloor d \rfloor \equiv \lfloor \neg b_1 \vee b_2 \rfloor \wedge d.$

□

6.5.4 Implementation and Run-Time Results

We have implemented an algorithm that computes the lattice bisimulation relation based on the matrix multiplication (see Theorem 6.4.11) in a generic way. Specifically, this implementation is independent of how the irreducible elements are encoded, ensuring that no implementation details of operations such as matrix multiplication can interfere with the run-time results. For our experiments, we instantiated it in two possible ways: with bit vectors representing feature combinations and with ROBDDs as outlined above. Our results show a significant advantage when we use BDDs to compute lattice bisimilarity. The implementation is written in C# and uses the CUDD package by Fabio Somenzi via the interface PAT.BDD [NSL⁺12].

To show that the use of BDDs can potentially lead to an exponential gain in speed when compared to the naive bit vector implementation, we executed the algorithm on a family of increasingly larger LaTSs over an increasingly larger number of features, where all features are upgrade features. Hence, let F be

a set of features. The example we studied contains, for each feature $f \in F$, one disconnected component in both LaTSs that is depicted in Figure 6.2: the component for α on the left, the component for β is on the right. The only difference between the two is in the guard of the transition from state 0 to state 2.

The quotient of the times taken without BDDs and with BDDs is growing exponentially by a factor of about 2 for each additional feature (see the table below). Due to fluctuations, an exact rate cannot be given. By the eighteenth iteration (i.e. 18 features and copies of the basic component), the implementation using BDDs needed 17 seconds, whereas the version without BDDs took more than 96 hours. The nineteenth iteration exceeded the memory for the implementation without BDDs, but terminated within 22 seconds with BDDs.

Table 6.1 shows the run-time results (in milliseconds) for the computation of the largest bisimulation for our implementation on the family of CTSs.

It is true that the example is somewhat artificial, but it is hard to find a realistic case study where the number of features can be arbitrarily increased.

6.6 Conclusion, Related Work, Future Work

In this section, we have shown how CTSs can be equipped with an order on conditions to obtain systems whose behaviour can be upgraded by replacing the current condition by a smaller one. Corresponding verification techniques based on behavioural equivalences can be important for SPLs where an upgrade to a more advanced version of the same software should occur without unexpected behaviour. To this end, we proposed an algorithm, based on matrix multiplication, that allows to compute the greatest bisimulation of two given CTSs. Interestingly, the duality between lattices and downward-closed sets of posets, as well as the embedding into a Boolean algebra proved to be fruitful when developing it and proving its correctness.

There are two ways in which one can extend CTSs as a specification language: first, in some cases it makes sense to specify that an advanced version offers improved transitions with respect to a basic version. For instance, in our running example, allowing the router to send unencrypted messages in an unsafe environment is superfluous, because the advanced version always has

features	time(BDD) (in ms)	time(without BDD) (in ms)	$\frac{\text{time(without BDD)}}{\text{time(BDD)}}$
1	42	13	0.3
2	64	32	0.5
3	143	90	0.6
4	311	312	1.0
5	552	1128	2.0
6	1140	3242	2.8
7	1894	8792	4.6
8	1513	13256	8.8
9	1872	39784	21
10	3208	168178	52
11	5501	513356	93
12	7535	1383752	184
13	5637	3329418	591
14	6955	8208349	1180
15	11719	23700878	2022
16	15601	57959962	3715
17	18226	150677674	8267
18	17001	347281057	20427
19	22145	out of memory	—

Table 6.1: Runtime results on a family of CTS.

the encryption feature. Such a situation can be modelled in a CTS by adding a precedence relation over the set of actions, leading to the deactivation of transitions, which is worked out in Subsection 6.4.5. The second question is how to incorporate downgrades: one solution could be to work with a pre-order on conditions, instead of an order. This simply means that two conditions $\varphi \neq \psi$ with $\varphi \leq \psi$, $\psi \leq \varphi$ can be merged, since they can be exchanged arbitrarily. Naturally, one could study more sophisticated notions of upgrade and downgrade in the context of adaptivity.

As for the related work, literature on adaptive SPLs can be grouped into either empirical or formal approaches; however, below we focus only on the formal ones [CCH⁺13, GS13, DKB14, tBLLV15].

Cordy et al. [CCH⁺13] model an adaptive SPL using an FTS which encodes not only a product's transitions, but also how some of the features may change via the execution of a transition. In contrast, we encode adaptivity by requiring a partial order on the products of an SPL and its effect on behaviour evolution by the monotonicity requirement on the transition function. Moreover, instead of studying the model checking problem as in [CCH⁺13], our focus was on bisimilarity between adaptive SPLs.

In [GS13], the authors proposed methods to analyse reliability and energy consumption properties of an adaptive SPL, using Discrete Time Markov Chains (DTMC) and the model checker PARAM [HHWZ10].

Dubslaff et al. [DKB14] extended a Markov decision process (MDP) with annotated costs to analyse and verify nonfunctional requirements of an adaptive SPL, using the PRISM model checker.

In [tBLLLV15], a process calculus QFLan motivated by concurrent constraint programming was developed. Thanks to an in-built notion of a store, various aspects of an adaptive SPL such as (un)installing a feature and replacing a feature by another feature can be modelled at run-time by operational rules.

Behavioural equivalences such as (bi)simulation relations have already been studied in the literature of traditional SPLs. In [CCP⁺12], the authors proposed a definition of simulation relation between any two FTSs (without upgrades) to combat the state explosion problem by establishing a simulation relation between a system and its refined version. In contrast, the authors in [AFL15] used simulation relations to measure the discrepancy in behaviour caused by feature interaction, i.e., whether a feature that is correctly designed in isolation works correctly when combined with the other features or not.

(Bi)simulation relations on lattice Kripke structures were already studied by Kupferman and Lustig [KL10], although in the context of three-valued model checking (rather than on the analysis of adaptive SPLs). Disregarding the differences between transition systems and Kripke structures (i.e., forgetting the role of atomic propositions), the definition of bisimulation in [KL10] is quite similar to our Definition 6.4.1 (another similar formula occurs in [CCP⁺12]). However, in [KL10] the stronger assumption of finite distributive de Morgan algebras is used.³ Thus, the notion of bisimulation as in [KL10] and our notion

³A De Morgan algebra is a lattice \mathbb{L} with a negation operation $\neg : \mathbb{L} \rightarrow \mathbb{L}$ subject to

of bisimulation are not necessarily the same, since lattices arising from partial orders are in general not de Morgan algebras.

Lastly, Fitting [Fit02] studied bisimulation relations in the setting of unlabelled transition systems and gave an elegant characterisation of bisimulation when transition systems and the relations over states are viewed as matrices. We have shown that this corresponds to LaTS over Boolean algebras with a single alphabet symbol.

In summary, compared to related work, we are treating general distributive lattices that allow us to conveniently model and reason about upgrades.

$\bar{\bar{a}} = a$ and $a \leq b \Rightarrow \bar{a} \geq \bar{b}$. Note that the well known negation of Heyting algebras $\neg a$ does not necessarily yield a De Morgan algebra. As an example, consider the finite lattice from Example 2.2.6 in which we have $\neg\neg b = e \neq b$.

Chapter 7

Conditional Transition Systems Coalgebraically

7.1 Introduction

In Chapter 6, we have extended conditional transition systems (CTS), which were previously designed without upgrades in mind, to allow for a notion of upgrading. The original notion of CTS was introduced in [ABH⁺12] and has already been discussed in Chapter 2. In this chapter we want show how they fit into a coalgebraic framework.

Coalgebraic modelling of CTS is interesting for several reasons: first, it gives a non-trivial case study in coalgebra which demonstrates the generality of the approach. Second, it studies coalgebras in the category of partially ordered sets, respectively in Kleisli categories over this base category. We use the Birkhoff duality for distributive lattices to show equivalence of two Kleisli categories over two monads: the reader monad and the so-called lattice monad. This result can be of interest, independently of the coalgebraic theory. Third, we introduce a notion of upgrade into coalgebraic modelling.

The theory of coalgebras [Rut00] (see Section 2.4) allows uniform modelling and reasoning for a variety of state-based systems. For instance, (non)deterministic finite automata and weighted automata are the classical examples often studied in this context (see [Rut00] for more examples). Furthermore, coalgebraic modelling comes with the benefit of offering generic algorithms, capturing the core of algorithms that are similar across different types of automata. In

particular, through the final chain-based reasoning, one can compute quotients on automata up to a chosen notion of behavioural equivalence (such as strong bisimilarity or trace equivalence). A detailed discussion of algorithms based on the final chain can be found in Chapter 3.

A conditional transition system (CTS) [ABH⁺12], cf. also Chapter 6, is an extension of labelled transition system that is well suited to model software product lines [CN01], an emergent topic of research in the field of software engineering. In contrast to the commonly used featured transition systems [CCS⁺13], CTSs are not primarily concerned with the individual features of a software product, but mainly with the individual versions that may arise from the given feature combinations.

In CTSs [BKKS17], transitions are labelled with elements of a partially ordered set of conditions (Φ, \leq_Φ) , which can be viewed as products in the terminology of software product lines. This gives us a compact representation which merges the transition systems for many different products into one single structure.

Intuitively, CTSs evolve in two steps: first, a condition $\varphi \in \Phi$ is chosen at a given state; second, a transition is fired which is guarded by the chosen condition. Over the course of the run of a CTS, it can perform an operation called *upgrade* in which the system changes from a greater condition φ to a smaller condition $\varphi' \leq_\Phi \varphi$. This in turn activates additional transitions that may be taken in future steps. This model of CTS is discussed in Chapter 6. In [ABH⁺12], CTSs were defined without upgrades, i.e., $\leq_\Phi = =_\Phi$.

An interesting fact about a CTS is that there exists an equivalent model, called *lattice transition system* (LaTS), which allows for a more compact representation of a CTS using the lattice of downward-closed subsets of Φ (see Chapter 6 for more details). In essence, this can be viewed as a lifting of the well-known Birkhoff's representation theorem to the case of transition systems.

This paper aims at characterising CTS and LaTS coalgebraically. To this end, we define two monads, the reader monad and the lattice monad, which allow for modelling CTS and LaTS respectively – provided a matching functor is chosen – in their corresponding Kleisli categories. Note, that these functors are defined on the category **Poset**, so the reader monad we define here is not the same as the one on **Set**, defined in Chapter 2, though it is of course closely

related. We will show that these two categories are equivalent (in the categorical sense).

Our next aim is to characterise conditional bisimilarity using the notion of behavioural equivalence, a concept stemming from the theory of coalgebras. Roughly, two states of a system (modelled as a coalgebra) are *behaviourally equivalent* if and only if they are mapped to a common point by a coalgebra homomorphism. In this regard, capturing the right notion of behavioural equivalence (conditional bisimilarity in our case) depends on choosing the right choice of functor modelling CTSs. The usual powerset functor \mathcal{P} proves to be a viable choice for CTS without any upgrades, but we will see that it is not a viable choice to model CTS with upgrades. However, for a slight adaptation of the powerset functor, i.e., allowing $FX = \mathcal{P}(X \times \Phi)$ for a set of states X , behavioural equivalence indeed captures conditional bisimilarity. The idea is to record successor states together with the corresponding product versions, instead of just the possible successor states.

To conclude, we show that Algorithm C from Chapter 3, the minimisation algorithm based on the final chain construction plus factorisations [ABH⁺12], is applicable to the category under investigation and specify how it can be applied to CTS. CTS without upgrades have already been considered in [ABH⁺12], but applicability to CTS with upgrades is novel.

7.2 Preliminaries

Coalgebraic preliminaries In this chapter, our base category \mathbf{C} will be the well-known category of partially ordered sets denoted as \mathbf{Poset} . Formally, the objects of \mathbf{Poset} are pairs (X, \leq_X) , where X is a set and \leq_X is a partial order on X . Its arrows $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$ are all the order preserving functions from X to Y . Recall that \mathbf{Poset} is concrete over \mathbf{Set} as evident by the forgetful functor $U: \mathbf{Poset} \rightarrow \mathbf{Set}$ defined as: $U(X, \leq_X) = X$, for any object (X, \leq_X) , and $Uf = f$, for any arrow $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$.

Next, we recall the Kleisli category over the concrete category \mathbf{Poset} , which is just an instantiation of Definition 2.4.20 to the category \mathbf{Poset} .

Definition 7.2.1 (Kleisli Category as a Concrete Category) *Let (T, η, μ) be a monad on \mathbf{Poset} where η is the unit and μ the multiplication. Then*

its Kleisli category $\mathbf{Kl}(T)$ consists of partially ordered sets as objects (i.e. the objects of \mathbf{Poset}) and an arrow $X \rightarrow Y$ is an order-preserving function $X \rightarrow TY$. The identity arrow in $\mathbf{Kl}(T)$ is given by η_X and the composition of two arrows $f: X \rightarrow Y$, $g: Y \rightarrow Z$ in $\mathbf{Kl}(T)$ is given by $\mu_Z \circ Tg \circ f$. Kleisli categories on \mathbf{Poset} are concrete categories as evident by the concretisation functor $U(X, \leq_X) = \pi_1^2(T(X, \leq))$ on objects and $Uf = \mu_x \circ Tf$ on arrows $f: (X, \leq) \rightarrow (Y, \leq)$.

For the remainder of the chapter we will not make the usage of the concretisation functor explicit, but identify objects with their concretisation when necessary. Intuitively, Kleisli categories help in distinguishing between the visible effects of transitions in a system and those that can be characterised as side-effects instead. Effects of a transition that are intended to be observable when determining behavioural equivalence are encoded in a functor F on the Kleisli category, whereas the side effects are encoded via a monad T . This is motivated by the previous works in [HJS07, PT99], where Kleisli categories (on \mathbf{Set} rather than \mathbf{Poset}) are used to obtain (trace) language equivalence as behavioural equivalence instead of bisimulation, as demonstrated in Subsection 2.4.5.

In Definition 2.4.18 we have already discussed the well-known reader monad on \mathbf{Set} . We will now introduce a \mathbf{Poset} variant of the reader monad, which we will also simply call the reader monad. Throughout this chapter, whenever we use the name reader monad, we refer to the \mathbf{Poset} version.

Definition 7.2.2 (Reader Monad) *Let (Φ, \leq_Φ) be a finite partially ordered set. Then, for an object (X, \leq_X) and an arrow f in \mathbf{Poset} , we fix:*

$$(X, \leq_X)^\Phi = (\{C: (\Phi, \leq_\Phi) \rightarrow (X, \leq_X)\}, \leq_{X^\Phi}) \quad f^\Phi(C) = f \circ C,$$

where $C \leq_{X^\Phi} D$ if and only if $C(\varphi) \leq_X D(\varphi)$ for all $\varphi \in \Phi$. Furthermore, the unit $\nu_X: (X, \leq_X) \rightarrow (X, \leq_X)^\Phi$ is given as $\nu_X(x)(\varphi) = x$. Lastly, the multiplication $\zeta_X: ((X, \leq_X)^\Phi)^\Phi \rightarrow (X, \leq_X)^\Phi$ is defined as $\zeta_X(\hat{C})(\varphi) = \hat{C}(\varphi)(\varphi)$, for any $\hat{C} \in (X^\Phi)^\Phi$.

Proposition 7.2.3 *The reader monad $(_\!^\Phi, \nu, \zeta)$ is a monad on \mathbf{Poset} .*

Proof: Since order preservation is closed under function composition, it is clear that f^Φ is well-defined for an arrow f in \mathbf{Poset} . And the unit ν_X is a constant

function, so clearly it is order preserving. Furthermore, for the multiplication, let $\varphi \leq \varphi'$ and $\widehat{C}(\varphi) \leq \widehat{C}(\varphi')$. Then, $\widehat{C}(\varphi)(\varphi) \leq \widehat{C}(\varphi')(\varphi) \leq \widehat{C}(\varphi')(\varphi')$, showing that ζ_X is also order preserving. The rest of the proof obligations, namely that ν, M are actually natural transformations and that the unit and associative laws hold are standard, just like the well-known reader monad on **Set** as outlined in Definition 2.4.18. \square

As mentioned in the introduction, we will model LaTS as a coalgebra in the Kleisli category of a monad. One could try to simply use the monad mapping sets to arbitrary lattice-valued functions defined on objects as $TX = \mathbb{L}^X$ on objects and on arrows as $Tf(b)(y) = \bigsqcup_{f(x) \leq y} b(x)$, however, this would not be equivalent to the reader monad. Given a monotone function $f: \Phi \rightarrow X$, one would like to define a corresponding mapping $\bar{f}: X \rightarrow \mathbb{L}$ with $\mathbb{L} = \mathcal{O}(\Phi)$ and $\bar{f}(x) = \bigsqcup \{\varphi \in \Phi \mid f(\varphi) \leq x\}$. However, this does not result in a bijection, since some arrows $\bar{f}: X \rightarrow \mathbb{L}$ do not represent a monotone function $f: \Phi \rightarrow X$. Hence, we start by imposing restrictions on mappings \mathbb{L}^X and defining a suitable endofunctor in our base category **Poset**. The full definition of the lattice monad is as follows:

Definition 7.2.4 *Let \mathbb{L} be a complete lattice satisfying the join-infinite distributive law:*

$$\ell \sqcap \bigsqcup L = \bigsqcup \{\ell \sqcap \ell' \mid \ell' \in L\}, \quad (\text{for any } L \subseteq \mathbb{L}). \quad (\text{JID})$$

For an ordered set (X, \leq_X) , let $T(X, \leq_X) = (TX, \leq_{TX})$, where $TX = (X \rightarrow \mathbb{L})^$ is the set of all those functions $b: X \rightarrow \mathbb{L}$ satisfying the following restrictions¹:*

1. $\bigsqcup_{x \in X} b(x) = \top$
2. *For any $x, x' \in X$, we have $b(x) \sqcap b(x') = \bigsqcup \{b(y) \mid y \leq x, y \leq x'\}$.*
3. *For any join-irreducible element $\ell \in \mathcal{J}(\mathbb{L})$, we have*

$$\exists_{x \in X} \ell \leq_{\mathbb{L}} b(x) \wedge \forall_{x' \in X} (\ell \leq_{\mathbb{L}} b(x') \implies x \leq_X x') .$$

¹In case of finite \mathbb{L} , the Conditions 1 and 2 follow from Condition 3 and therefore need not be checked.

Furthermore, for any two functions $b, b' \in (X \rightarrow \mathbb{L})^*$ we let²

$$b \leq_{TX} b' \iff \forall_{x \in X} b'(x) \leq_{\mathbb{L}} b(x) .$$

For an order preserving function $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$, we fix

$$Tf(b)(y) = \bigsqcup_{f(x) \leq_Y y} b(x), \quad \text{for any } b \in (X \rightarrow \mathbb{L})^*, y \in Y .$$

Lastly, define two families of maps $\eta_X: X \rightarrow TX$ and $\mu_X: TTX \rightarrow TX$:

$$\eta_X(x)(x') = \begin{cases} \top & \text{if } x \leq x' \\ \perp & \text{otherwise} \end{cases},$$

$$\mu_X(\hat{B})(x) = \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} (\hat{B}(b) \sqcap b(x)), \quad \text{where } \hat{B} \in ((X \rightarrow \mathbb{L})^* \rightarrow \mathbb{L})^*.$$

Then (T, η, μ) is called the lattice monad.

Since the proof that the lattice monad is actually a monad is very technical and the techniques used do not play an important role in the remainder of the chapter, the proof is omitted here. The interested reader may consult Appendix C to find a full proof.

In order to understand this chapter, it is required to have an understanding of CTS, LaTS and their notion of bisimulation. If the reader has not read Chapter 6, it is recommended, to read Section 6.2 and Section 6.3, in particular the Definitions 6.2.1, 6.3.1 and 6.2.3 as well as Theorem 6.3.2 at this point.

7.3 Equivalence of Lattice Monad and Reader Monad

In this section, using the famous Birkhoff's representation theorem, we show that the Kleisli categories of the lattice monad and the reader monad are actually isomorphic.

Throughout this section, we fix (Φ, \leq_Φ) as a finite partially ordered set and \mathbb{L} is the set of all downward-closed subsets of Φ . Recall that every downward-closed subset of Φ can be represented by an element of \mathbb{L} . As a result, in the sequel, we do not distinguish between the elements of \mathbb{L} and Φ .

²Note the reversal of the order in the ordering of functions!

We start out by defining natural transformations $\alpha: T \Rightarrow _^\Phi$ and $\alpha^{-1}: _^\Phi \Rightarrow T$. Subsequently, we will show that the lattice monad and the reader monad are equivalent in the sense that, for any object (X, \leq_X) in **Poset**, we have $T(X, \leq_X) \cong (X, \leq_X)^\Phi$. Lastly, we will show that the induced Kleisli categories are isomorphic as well.

Definition 7.3.1 Define two families of arrows $\alpha_X: TX \rightarrow (X, \leq_X)^{(\Phi, \leq_\Phi)}$ and $\alpha_X^{-1}: (X, \leq_X)^{(\Phi, \leq_\Phi)} \rightarrow TX$ (for each ordered set X) in the following way:

$$\begin{aligned}\alpha_X(b)(\varphi) &= \min\{x \mid b(x)_\perp \geq \varphi\} \quad (\text{for every } b \in TX \text{ and } \varphi \in \Phi), \\ \alpha_X^{-1}(C)(x) &= \bigsqcup\{\varphi \mid C(\varphi) \leq_X x\} \quad (\text{for every } C \in (X, \leq_X)^{(\Phi, \leq_\Phi)} \text{ and } x \in X).\end{aligned}$$

Lemma 7.3.2 For an object in **Poset**, the corresponding functions α_X and α_X^{-1} are arrows in **Poset**.

Proof: Let (X, \leq_X) be an ordered set. To show that α_X is order preserving, let $b_2 \leq_{TX} b_1$, for some $b_1, b_2 \in TX$. I.e., $b_1(x) \leq_\perp b_2(x)$ for all $x \in X$ (remember the reversal of orders in the lattice monad T). Since $b_1(x) \leq_\perp b_2(x)$ for all $x \in X$, it must hold

$$\begin{aligned}b_1(x) \geq_\perp \varphi &\implies b_2(x) \geq_\perp \varphi \\ \implies \{x \mid b_1(x) \geq_\perp \varphi\} &\subseteq \{x \mid b_2(x) \geq_\perp \varphi\} \\ \implies \min\{x \mid b_1(x) \geq_\perp \varphi\} &\geq_X \min\{x \mid b_2(x) \geq_\perp \varphi\} \implies \alpha_X(b_1)(\varphi) \geq_X \alpha_X(b_2)(\varphi).\end{aligned}$$

And to show that α_X^{-1} is order preserving, let $C_1 \leq C_2$. Now, if $C_1(\varphi) \leq_X C_2(\varphi)$ (which is always true), then $C_2(\varphi) \leq_X x$ implies $C_1(\varphi) \leq_X x$, thus, $\alpha_X^{-1}(C_1)(x) \geq_\perp \alpha_X^{-1}(C_2)(x)$ for all x and, due to the reversal of the order in T , $\alpha_X^{-1}(C_1) \leq \alpha_X^{-1}(C_2)$. \square

Next, we show that the functions α_X and α_X^{-1} (for an ordered set (X, \leq_X)) are well defined. The next lemma proves this fact.

Lemma 7.3.3 Let (X, \leq_X) be an ordered set.

1. If $b \in TX$, then $\alpha_X(b) \in (X, \leq_X)^{(\Phi, \leq_\Phi)}$, i.e., $\alpha_X(b)$ is order preserving.
2. If $C \in (X, \leq_X)^{(\Phi, \leq_\Phi)}$, then $\alpha_X^{-1}(C) \in TX$.

Proof: For (1), let $\varphi \leq_\Phi \varphi'$. Then, we need to show that $\alpha_X(b)(\varphi) \leq_X \alpha_X(b)(\varphi')$, which follows directly from the fact that $\{x \mid \varphi' \leq_\perp b(x)\} \subseteq \{x \mid \varphi \leq_\perp b(x)\}$ holds.

For (2), let $C \in (X, \leq_X)^{(\Phi, \leq_\Phi)}$. Then, we need to show the three conditions for T :

Condition 1 Using the fact that $C \in (X, \leq_X)^{(\Phi, \leq_\Phi)}$ is a total function, we get

$$\bigsqcup_{x \in X} \alpha_X^{-1}(C)(x) = \bigsqcup_{x \in X} \bigsqcup_{C(\varphi) \leq_X x} \varphi = \bigsqcup \Phi = \top.$$

Condition 2 Let $x, x' \in X$. Then, using distributivity we find

$$\begin{aligned} \alpha_X^{-1}(C)(x) \sqcap \alpha_X^{-1}(C)(x') &= \bigsqcup_{C(\varphi) \leq_X x} \varphi \sqcap \bigsqcup_{C(\varphi') \leq_X x'} \varphi' \\ &= \bigsqcup_{C(\varphi) \leq_X x, C(\varphi') \leq_X x'} \varphi \sqcap \varphi'. \end{aligned}$$

Furthermore,

$$\bigsqcup_{\bar{x} \leq_X x, \bar{x} \leq_X x'} \alpha_X^{-1}(C)(\bar{x}) = \bigsqcup_{\bar{x} \leq_X x, \bar{x} \leq_X x'} \bigsqcup_{C(\varphi) \leq \bar{x}} \varphi = \bigsqcup_{C(\varphi) \leq_X x, C(\varphi) \leq_X x'} \varphi.$$

Next, we show that $\{\varphi \mid C(\varphi) \leq_X x, C(\varphi) \leq_X x'\} = \{\varphi \sqcap \varphi' \mid C(\varphi) \leq_X x, C(\varphi') \leq_X x'\}$ to complete the proof for this case. The direction ‘ \subseteq ’ is obvious, because $\varphi = \varphi \sqcap \varphi$. For the other direction, let $C(\varphi) \leq_X x$ and $C(\varphi') \leq_X x'$. Then, using order preservation of C we find $C(\varphi \sqcap \varphi') \leq_X C(\varphi) \leq_X x$. Likewise, $C(\varphi \sqcap \varphi') \leq_X C(\varphi') \leq_X x'$.

Condition 3 Let φ be a join irreducible element and let $x = C(\varphi)$. Then, we find that $\varphi \leq_{\mathbb{L}} \bigsqcup_{C(\varphi') \leq_X C(\varphi)} \varphi' = \alpha_X^{-1}(C)(x)$. Now suppose $\varphi \leq_{\mathbb{L}} \alpha_X^{-1}(C)(x')$, for some $x' \in X$. Then, $\varphi \leq_{\mathbb{L}} \bigsqcup_{C(\varphi') \leq_X x'} \varphi'$. Since φ is join-irreducible, we get $C(\varphi) \leq_X x'$ as required.

□

To show that α and α^{-1} induce isomorphisms between the two Kleisli categories, we will proceed to show that they are each other's inverses.

Lemma 7.3.4 *The transformations α and α^{-1} are each other's inverses.*

Proof: Let (X, \leq_X) be an ordered set. Then we distinguish the following cases.

- To show $\alpha_X^{-1} \circ \alpha_X(b) = b$, for $b \in TX$. Expanding $\alpha_X^{-1} \circ \alpha_X(b)$ we get

$$\begin{aligned} \alpha_X^{-1}(\alpha_X(b))(x) &= \bigsqcup \{\varphi \mid \alpha_X(b)(\varphi) \leq_X x\} \\ &= \bigsqcup \{\varphi \mid \min\{y \mid \varphi \leq_{\mathbb{L}} b(y)\} \leq_X x\}. \end{aligned}$$

Now, if $\varphi \leq_{\mathbb{L}} b(x)$, then $x \geq_X \min\{y \mid \varphi \leq_{\mathbb{L}} b(y)\}$ and thus $\varphi \leq_{\mathbb{L}} \alpha_X^{-1}(\alpha_X(b))(x)$. If, on the other hand, $\varphi \leq_{\mathbb{L}} \alpha_X^{-1}(\alpha_X(b))(x)$ holds, then there is a $y \in X$ such that $\varphi \leq_{\Phi} b(y)$ and $y \leq_X x$. Since b is order preserving, it follows that $\varphi \in b(x)$.

- To show $\alpha_X \circ \alpha_X^{-1}(C) = C$, for $C \in (X, \leq_X)^{(\Phi, \leq_{\Phi})}$. Expanding $\alpha_X \circ \alpha_X^{-1}(C)$ we get

$$\begin{aligned} \alpha_X(\alpha_X^{-1}(C))(\varphi) &= \min\{x \mid \alpha_X^{-1}(C)(x) \geq_{\mathbb{L}} \varphi\} \\ &= \min\{x \mid \bigsqcup\{\varphi' \mid C(\varphi') \leq_X x\} \geq_{\mathbb{L}} \varphi\}. \end{aligned}$$

Clearly, $C(\varphi) \leq_X x \Rightarrow \bigsqcup\{\varphi' \mid C(\varphi') \leq_X x\} \geq_{\mathbb{L}} \varphi \Rightarrow x \geq_X \min\{x \mid \bigsqcup\{\varphi' \mid C(\varphi') \leq_X x\} \geq_{\mathbb{L}} \varphi\}$.

If, on the other hand, $x \geq_X \alpha_X(\alpha_X^{-1}(C))(\varphi)$, i.e. $x \geq_X \min\{x \mid \bigsqcup\{\varphi' \mid C(\varphi') \leq_X x\} \geq_{\mathbb{L}} \varphi\}$, let $y := \min\{x \mid \bigsqcup\{\varphi' \mid C(\varphi') \leq_X x\} \geq_{\mathbb{L}} \varphi\}$. Then, $\bigsqcup\{\varphi' \mid C(\varphi') \leq_X y\} \geq_{\mathbb{L}} \varphi$ holds. Since φ is a join-irreducible element, we find $\varphi \leq_{\mathbb{L}} \varphi'$ and $C(\varphi') \leq_X y$, for some $\varphi' \in \Phi$. Lastly, order preservation of C and transitivity of \leq together yields $C(\varphi) \leq_X x$.

□

Henceforth, we will make heavy use the following notation, introduced in Definition 7.2.1: for an order preserving function $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$, we write f^{Φ} to denote the image of f under the reader monad, i.e., $f^{\Phi}(C) = f \circ C$, for any $C \in (X, \leq_X)^{(\Phi, \leq_{\Phi})}$.

Lemma 7.3.5 *The family α given in Definition 7.3.1 is a natural transformation.*

Proof: Given an order preserving function $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$, we need to show $\alpha_Y(T(f)(b))(\varphi) = f^{\Phi}(\alpha_X(b))(\varphi)$, for any $b \in TX$ and $\varphi \in \Phi$.

$$f^{\Phi}(\alpha_X(b))(\varphi) = f(\alpha_X(b))(\varphi) = f(\min\{x \mid b(x) \geq_{\mathbb{L}} \varphi\}) = \min\{f(x) \mid b(x) \geq_{\mathbb{L}} \varphi\}$$

In the last step we used monotonicity of f . On the other hand we have:

$$\alpha_Y(T(f)(b))(\varphi) = \min\{y \mid Tf(b)(y) \geq_{\mathbb{L}} \varphi\} = \min\{y \mid \bigsqcup\{b(z) \mid f(z) \leq_Y y\} \geq_{\mathbb{L}} \varphi\}$$

We will now show that both the expressions are equal.

- Let $f(x) \in \{f(x) \mid b(x) \geq_{\mathbb{L}} \varphi\}$, i.e. x be chosen such that $b(x) \geq_{\mathbb{L}} \varphi$. Then, $\bigsqcup\{b(z) \mid f(z) \leq_Y f(x)\} \geq \varphi$ and therefore $f(x) \in \{y \mid \bigsqcup\{b(z) \mid f(z) \leq_Y y\} \geq \varphi\}$. So, in particular, $f(x) \geq \min\{y \mid \bigsqcup\{b(z) \mid f(z) \leq_Y y\} \geq_{\mathbb{L}} \varphi\}$.
- Let $y \in \{y' \mid \bigsqcup\{b(z) \mid f(z) \leq_Y y'\} \geq_{\mathbb{L}} \varphi\}$. Since φ is join-irreducible, there must be a z such that $f(z) \leq_Y y$ and $b(z) \geq_{\mathbb{L}} \varphi$, i.e. $y \geq_Y \min\{f(x) \mid b(x) \geq_{\mathbb{L}} \varphi\}$.

□

Lemma 7.3.6 *The family α^{-1} given in Definition 7.3.1 is a natural transformation.*

Proof: Let $C \in X^{\Phi}$, $y \in Y$, $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$. We compute:

$$\begin{aligned} T(f)(\alpha_X^{-1}(C))(y) &= \bigsqcup\{\alpha_X^{-1}(C)(x) \mid f(x) \leq_Y y\} \\ &= \bigsqcup\{\bigsqcup\{\varphi \mid C(\varphi) \leq_X z, z \leq_X x\} \mid f(x) \leq_Y y\} = \bigsqcup\{\varphi \mid C(\varphi) \leq_X x, f(x) \leq_Y y\} \end{aligned}$$

as well as:

$$\begin{aligned} \alpha_Y^{-1}(f^{\Phi}(C))(y) &= \bigsqcup\{\varphi \mid f^{\Phi}(C)(\varphi) = x, x \leq_Y y\} \\ &= \bigsqcup\{\varphi \mid f(C(\varphi)) \leq_Y x, x \leq_Y y\} = \bigsqcup\{\varphi \mid f(C(\varphi)) \leq_Y y\}. \end{aligned}$$

We will now show that $\{\varphi \mid C(\varphi) \leq_X x, f(x) \leq_Y y\} = \{\varphi \mid f(C(\varphi)) \leq_Y y\}$, and therefore that their suprema are identical, too.

- Let $C(\varphi) \leq_X x$ and $f(x) \leq_Y y$, then, due to monotonicity of f , $f(C(\varphi)) \leq_Y f(x) \leq_Y y$.
- Let $f(C(\varphi)) \leq_Y y$. We define $x := C(\varphi)$ and observe that $f(x) = f(C(\varphi)) \leq_Y y$.

□

Now we want to show that the units are isomorphic as well.

Lemma 7.3.7 *Let (X, \leq_X) be an ordered set. Then, the following properties hold.*

1. For all $x, x' \in X$ we have $\alpha_X^{-1}(\nu_X(x))(x') = \eta_X(x)(x')$.
2. For all $x \in X$, $\varphi \in \Phi$ we have $\alpha_X(\eta_X(x))(\varphi) = \nu_X(x)(\varphi)$.

Proof: For (1), we derive

$$\begin{aligned} \alpha_X^{-1}(\nu_X(x))(x') &= \bigsqcup \{\varphi \mid \nu_X(x)(\varphi) \leq_X y, y \leq_X x'\} = \bigsqcup \{\varphi \mid \nu_X(x)(\varphi) \leq_X x'\} \\ &= \bigsqcup \{\varphi \mid x \leq_X x'\} = \begin{cases} \top & \text{if } x \leq_X x' \\ \perp & \text{otherwise} \end{cases} = \eta_X(x)(x'). \end{aligned}$$

For (2), we find $\alpha_X(\eta_X(x))(\varphi) = \min\{y \mid \eta_X(x)(y) \geq_{\mathbb{L}} \varphi\} = \min\{y \mid x \leq_X y\} = \nu_X(x)(\varphi)$. \square

In order to prove that the respective multiplications of our monads are isomorphic, we will investigate commutativity of two diagrams (drawn below in Lemma 7.3.8 and Lemma 7.3.9) that basically describe the translation process from one multiplication to the other and vice-versa. This is later used to prove that the corresponding Kleisli categories are isomorphic.

Lemma 7.3.8 *The following diagram commutes for all $g: Y \rightarrow T(Z)$:*

$$\begin{array}{ccccc} T(Y) & \xrightarrow{Tg} & TT(Z) & \xrightarrow{\mu_Z} & T(Z) \\ \alpha_Y \downarrow & & & & \downarrow \alpha_Z \\ Y^\Phi & & & & \\ g^\Phi \downarrow & & & & \\ T(Z)^\Phi & \xrightarrow{\alpha_Z^\Phi} & Z^{\Phi\Phi} & \xrightarrow{\zeta_Z} & Z^\Phi \end{array}$$

Proof: Let $b \in T(Y)$, $\varphi \in \Phi$. We compute:

$$\begin{aligned} (\alpha_Z \circ \mu_Z \circ Tg)(b)(\varphi) &= \min\{z \mid (\mu_Z \circ Tg)(b)(z) \geq_{\mathbb{L}} \varphi\} \\ &= \min\{z \mid \bigsqcup \{Tg(b)(d) \sqcap d(z) \mid d \in (Z \rightarrow \mathbb{L})^*\} \geq_{\mathbb{L}} \varphi\} \\ &= \min\{z \mid \bigsqcup \{b(y) \sqcap d(z) \mid d \in (Z \rightarrow \mathbb{L})^*, g(y) \leq d\} \geq_{\mathbb{L}} \varphi\}. \end{aligned}$$

as well as

$$\begin{aligned} (\zeta_Z \circ \alpha_Z^\Phi \circ g^\Phi \circ \alpha_Y)(b)(\varphi) &= (\alpha_Z^\Phi \circ g^\Phi \circ \alpha_Y)(b)(\varphi)(\varphi) \\ &= (\alpha_Z \circ g^\Phi \circ \alpha_Y)(b)(\varphi)(\varphi) = \min\{z \mid g^\Phi \circ \alpha_Y(b)(\varphi)(z) \geq_{\mathbb{L}} \varphi\} \\ &= \min\{z \mid g(\alpha_Y(b)(\varphi))(z) \geq_{\mathbb{L}} \varphi\} = \min\{z \mid g(\min\{y \mid \varphi \leq_{\mathbb{L}} b(y)\})(z) \geq_{\mathbb{L}} \varphi\}. \end{aligned}$$

It is sufficient to show that the following conditions are logically equivalent (for every $z \in Z$):

$$\varphi \leq_{\mathbb{L}} \bigsqcup_{g(y) \leq d, d \in TZ} (b(y) \sqcap d(z)) \iff \varphi \leq_{\mathbb{L}} g(\min\{y \mid \varphi \leq_{\mathbb{L}} b(y)\})(z).$$

$\boxed{\Leftarrow}$ Fix $\bar{y} = \min\{y \mid \varphi \leq_{\mathbb{L}} b(y)\}$, $d = g(\bar{y})$. Thus, $\varphi \leq_{\mathbb{L}} b(\bar{y}) \sqcap d(z) \leq_{\mathbb{L}} \bigsqcup_{g(y) \leq d, d \in TZ} (b(y) \sqcap d(z))$.

$\boxed{\Rightarrow}$ Assume $\varphi \leq_{\mathbb{L}} \bigsqcup_{g(y) \leq d, d \in TZ} (b(y) \sqcap d(z))$. Since φ is join-irreducible, we find that $\varphi \leq_{\mathbb{L}} b(y) \sqcap d(z)$, for some $y \in Y, d \in TZ$ such that $g(y) \leq_{TZ} d$. Let $\bar{y} = \min\{y \mid \varphi \leq_{\mathbb{L}} b(y)\}$. Then,

$$\bar{y} \leq_Y y \implies g(\bar{y}) \leq_{TZ} g(y) \implies g(y)(z) \leq_{\mathbb{L}} g(\bar{y})(z).$$

Moreover, from above, we have $\varphi \leq_{\mathbb{L}} d(z)$ and $g(y) \leq_{TZ} d$, i.e., $d(z) \leq_{\mathbb{L}} g(y)(z)$ – again, respecting the reversal of orders. Thus, from transitivity of $\leq_{\mathbb{L}}$ we obtain $\varphi \leq_{\mathbb{L}} g(\bar{y})(z)$. \square

Lemma 7.3.9 *The following diagram commutes for all $g: (Y, \leq_Y) \rightarrow (Z, \leq_Z)$ (Φ, \leq_{Φ}) :*

$$\begin{array}{ccccc} Y^{\Phi} & \xrightarrow{g^{\Phi}} & Z^{\Phi^{\Phi}} & \xrightarrow{\zeta_Z} & Z^{\Phi} \\ \alpha_Y^{-1} \downarrow & & & & \downarrow \alpha_Z^{-1} \\ T(Y) & & & & \\ T(g) \downarrow & & & & \\ T(Z^{\Phi}) & \xrightarrow{T(\alpha_Z^{-1})} & TT(Z) & \xrightarrow{\mu_Z} & T(Z) \end{array}$$

Proof: Let $g: Y \rightarrow Z^{\Phi}$, $C \in Y^{\Phi}$, $z \in Z$. We compute:

$$\begin{aligned} \alpha_Z^{-1} \circ \zeta_Z \circ g^{\Phi}(C)(z) &= \bigsqcup \{\varphi \mid \zeta_Z \circ g^{\Phi}(C)(\varphi) \leq_Z z\} \\ &= \bigsqcup \{\varphi \mid g^{\Phi}(C)(\varphi)(\varphi) \leq_Z z\} = \bigsqcup \{\varphi \mid g(C(\varphi))(\varphi) \leq_Z z\}, \end{aligned}$$

as well as:

$$\begin{aligned} &(\mu_Z \circ T(\alpha_Z^{-1}) \circ T(g) \circ \alpha_Y^{-1})(C)(z) \\ &= \bigsqcup \{(T(\alpha_Z^{-1}) \circ T(g) \circ \alpha_Y^{-1})(C)(b) \sqcap b(z) \mid b \in TZ\} \\ &= \bigsqcup \{T(g) \circ \alpha_Y^{-1}(C)(d) \sqcap b(z) \mid \alpha_Z^{-1}(D) \leq_{TZ} b, D \in Z^{\Phi}, b \in TZ\} \\ &= \bigsqcup \{\alpha_Y^{-1}(C)(y) \sqcap b(z) \mid g(y) \leq D, \alpha_Z^{-1}(D) \leq_{TZ} b, D \in Z^{\Phi}, b \in TZ\} \\ &= \bigsqcup \{\varphi \sqcap b(z) \mid C(\varphi) \leq_Y y, g(y) \leq D, \alpha_Z^{-1}(D) \leq_{TZ} b, D \in Z^{\Phi}, b \in TZ\}. \end{aligned}$$

So we need to prove that these two expressions are equal.

- Assume $\varphi \leq_{\Phi} \bigsqcup \{\varphi' \sqcap b(z) \mid C(\varphi') \leq_Y y, g(y) \leq D, \alpha_Z^{-1}(D) \leq_{TZ} b, b \in TZ\}$. Since φ is join-irreducible, we find some $b \in TZ$, $y \in Y$, and $D \in Z^{\Phi}$ such that $\alpha_Z^{-1}(D) \leq_{TZ} b$, $g(y) \leq D$, $C(\varphi') \leq_Y y$, $\varphi \leq_{\mathbb{L}} \varphi'$ and $b(z) \geq_{\mathbb{L}} \varphi$. Clearly, using order preservation of g, C , and transitivity of \leq , we get $g(C(\varphi)) \leq g(C(\varphi')) \leq g(y)$.

Moreover, $b(z) \leq_{\mathbb{L}} \alpha_Z^{-1}(D)(z)$ since $\alpha_Z^{-1}(D) \leq_{TZ} b$. Thus, $\varphi \leq_{\mathbb{L}} \alpha_Z^{-1}(D)(z)$. Since φ is join-irreducible, we find some φ'' such that $\varphi \leq_{\mathbb{L}} \varphi''$ and $D(\varphi'') \leq_Z z$. And using order preservation of D and transitivity we get $D(\varphi) \leq_Z z$.

Now using the above facts $g(C(\varphi)) \leq g(y)$, $g(y) \leq D$, and $D(\varphi) \leq_Z z$, we find the desired inequality: $g(C(\varphi))(\varphi) \leq_Z g(y)(\varphi) \leq_Z D(\varphi) \leq_U z$.

- Assume $\varphi \leq_{\mathbb{L}} \bigsqcup \{\varphi' \mid g(C(\varphi'))(\varphi') \leq_Z z\}$, then $g(C(\varphi))(\varphi) \leq_Z z$. Let $y := C(\varphi)$, then $C(\varphi) \leq_Y y$ per definition. Let $D = g(C(\varphi))$, then $g(y) = g(C(\varphi)) = D$. Further define $b = \alpha_Z^{-1}(D)$, then obviously we have $b \in (Z \rightarrow \mathbb{L})^*$ and per definition $\alpha_Z^{-1}(D) \leq b$. We now only have left to prove that with these definitions, $\varphi \leq_{\mathbb{L}} b(z)$ is true. We compute:

$$b(z) = \alpha_Z^{-1}(D)(z) = \alpha_Z^{-1}(g(C(\varphi)))(z) = \bigsqcup \{\varphi' \mid g(C(\varphi))(\varphi') \leq_Z z\}.$$

Since $g(C(\varphi))(\varphi) \leq_Z z$ and $\varphi \in \{\varphi' \mid g(C(\varphi))(\varphi') \leq_Z z\}$ hold; therefore, we get $\varphi \leq_{\mathbb{L}} \bigsqcup \{\varphi' \mid g(C(\varphi))(\varphi') \leq_Z z\} = b(z)$.

□

Using the equivalence of the lattice and the reader monads, we can establish an equivalence between the corresponding Kleisli categories as well.

Theorem 7.3.10 *The categories $\mathbf{Kl}(T)$ and $\mathbf{Kl}(_{}^{\Phi})$ are isomorphic via the mapping*

$$\bar{\alpha}: \mathbf{Kl}(T) \rightarrow \mathbf{Kl}(_{}^{\Phi}) \quad \bar{\alpha}^{-1}: \mathbf{Kl}(_{}^{\Phi}) \rightarrow \mathbf{Kl}(T)$$

defined as $\bar{\alpha}(f) = \alpha_X \circ f$, for every $f: X \rightarrow TY$ in $\mathbf{Kl}(_{}^{\Phi})$ and $\bar{\alpha}^{-1}(f) = \alpha_X^{-1} \circ f$, for every $f: X \rightarrow X^{\Phi}$ in $\mathbf{Kl}(_{}^{\Phi})$.

Proof: We have already shown that $\bar{\alpha}(\bar{\alpha}^{-1}(f)) = f$ and $\bar{\alpha}^{-1}(\bar{\alpha}(f)) = f$, so we only have to prove that $\bar{\alpha}$ and $\bar{\alpha}^{-1}$ are functors.

- We show that $\bar{\alpha}$ is a functor.

– $\bar{\alpha}$ preserves identities. Let $id_X: X \xrightarrow{\eta_X} T(X)$ then we can compute

$$\bar{\alpha}(id_X) = X \xrightarrow{\alpha_X \circ \eta_X} X^\Phi \underset{\text{Lemma 7.3.7, Item 2}}{=} X \xrightarrow{\nu_X} X^\Phi = Id_{\bar{\alpha}(X)}$$

– $\bar{\alpha}$ respects composition. Let $f: X \rightarrow Y, g: Y \rightarrow Z$, we compute:

$$\begin{aligned} \bar{\alpha}(g \circ f) &\underset{\text{in Set}}{=} \alpha_Z \circ \mu_Z \circ Tg \circ f \underset{\text{Lemma 7.3.8}}{=} \zeta_Z \circ \alpha_Z^\Phi \circ g^\Phi \circ \alpha_Y \circ f \\ &= \zeta_Z \circ \bar{\alpha}(g)^\Phi \circ \bar{\alpha}(f) \underset{\text{in Kleisli}}{=} \bar{\alpha}(g) \circ \bar{\alpha}(f) \end{aligned}$$

- We show that $\bar{\alpha}^{-1}$ is a Functor.

– $\bar{\alpha}^{-1}$ preserves identities. Let $id_X: X \xrightarrow{\nu_X} X^\Phi$ then we can compute

$$\bar{\alpha}^{-1}(id_X) = X \xrightarrow{\alpha_X^{-1} \circ \nu_X} TX \underset{\text{Lemma 7.3.7, Item 1}}{=} X \xrightarrow{\eta_X} TX = Id_{\bar{\alpha}^{-1}(X)}$$

– $\bar{\alpha}^{-1}$ respects composition. Let $f: X \rightarrow Y, g: Y \rightarrow Z$, we compute:

$$\begin{aligned} \bar{\alpha}^{-1}(g \circ f) &\underset{\text{in Set}}{=} \alpha_Z^{-1} \circ \zeta_Z \circ g^\Phi \circ f \\ &\underset{\text{Lemma 7.3.9}}{=} \mu_Z \circ T\alpha_Z^{-1} \circ T(g) \circ \alpha_Y^{-1} \circ f \\ &= \mu_Z \circ T\bar{\alpha}(g)^{-1} \circ \bar{\alpha}^{-1}(f) \underset{\text{in Kleisli}}{=} \bar{\alpha}^{-1}(g) \circ \bar{\alpha}^{-1}(f) \end{aligned}$$

□

When proving the equivalence of the reader monad and the lattice monad, the duality between complete finite distributive lattices and partially ordered sets plays an important role, namely when defining the isomorphisms α and α^{-1} . Considering there is a similar duality between complete distributive lattices (not necessarily finite) and clopen downsets on Priestley spaces – i.e. spaces of partially ordered sets with a topology that makes them totally order disconnected – one might wonder why we restrict to the finite variant.

To make a similar duality result work for the infinite case, we would need to restrict the lattices with additional assumptions, because not all bounded distributive lattices are isomorphic to a lattice of downsets of a partial order. This holds if additionally the JID and its dual hold and some notion of atomicity holds for the irreducibles in the lattice [DP02]. This however would make the definition of the lattice monad more complicated.

7.4 Modelling CTS without Upgrades using \mathcal{P}

Recall that once a condition is fixed by a CTS then it behaves like a traditional transition system (until another upgrade occurs). Thus, it is natural to consider the powerset functor to model the set of next possible states when the upgrade order is discrete. In this section, we investigate the finite version of the well-known powerset functor. This way of modelling CTS extends the approach in [ABH⁺12], where the set of actions A was fixed to be singleton. Note that the modelling in [ABH⁺12] could be adapted to incorporate non-singleton sets of actions without changing to the category **Poset**. This change of categories will only become important when we turn our attention towards systems with upgrades. Throughout this section, we assume that Φ is discretely ordered.

Definition 7.4.1 *The functor \mathcal{P} on **Poset** is defined according to:*

- On objects (X, \leq_X) we have $\mathcal{P}(X, \leq_X) = (\{Y \subseteq X \mid Y \text{ is finite}\}, \supseteq)$, so $\mathcal{P}(X, \leq_X)$ is the set of all finite subsets of X , ordered by inverse inclusion.
- Let $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$, then $\mathcal{P}(f): \mathcal{P}(X) \rightarrow \mathcal{P}(Y)$ is defined as $\mathcal{P}(f)(S) = \{f(s) \mid s \in S\}$, so $\mathcal{P}(f)(S)$ is the image of S under f .

It is easy to see that \mathcal{P} is an endofunctor on **Poset** (the proof is similar to the powerset functor on **Set**).

Now in order to extend \mathcal{P} to $\mathbf{Kl}(_\Phi)$, we need to find a distributive law $\lambda: \mathcal{P}(_\Phi) \Rightarrow \mathcal{P}(_\Phi)$. This extension then allows to define CTS in $\mathbf{Kl}(_\Phi)$.

Definition 7.4.2 *We define the distributive law $\lambda: \mathcal{P}(_\Phi) \Rightarrow \mathcal{P}(_\Phi)$ according to*

$$\lambda_X(S)(\varphi) = \{C(\varphi) \mid C \in S\}$$

where $\varphi \in \Phi$, $S \in \mathcal{P}(X^\Phi)$.

Note, that it is important here, that we only consider discrete orders for Φ . In fact, for a non-discrete order, λ fails to be a distributive law, because the components λ_X are not arrows in the category, since they are not necessarily monotone in Φ . To see this, consider the following counterexample: for the state set $X = \{x, y\}$, where $x \leq y$ and the set of conditions $\Phi = \{\varphi, \varphi'\}$, where $\varphi' \leq \varphi$, consider $S = \{C\}$, where $C(\varphi) = y$, $C(\varphi') = x$. Then C is a monotone function, but $\lambda_X(S)(\varphi') = \{C(\varphi')\} = \{x\} \not\supseteq \{y\} = \{C(\varphi)\} = \lambda_X(S)(\varphi)$.

As expected, we obtain the following result.

Theorem 7.4.3 *The mapping λ defined above is a distributive law.*

Proof: First, to show that λ_X is an arrow (for any (X, \leq_X)), since the conditions are discretely ordered, nothing has to be shown, the computation is straight forward.

Second, we show that λ is a natural transformation. Let $S \in \mathcal{P}(X^\Phi)$ and $\varphi \in \Phi$. Then,

$$\begin{aligned} \lambda_X(\mathcal{P}(f^\Phi)(S))(\varphi) &= \{C'(\varphi) \mid C' \in \mathcal{P}(f^\Phi)(S)\} \\ &= \{C'(\varphi) \mid C' \in \{f^\Phi(C) \mid C \in S\}\} \\ &= \{C'(\varphi) \mid C' \in \{[\varphi' \mapsto f(C(\varphi'))] \mid C \in S\}\} \\ &= \{f(x) \mid x \in \{C(\varphi) \mid C \in S\}\} \\ &= \{f(x) \mid x \in \lambda_X(S)(\varphi)\} = \mathcal{P}(f)(\lambda_X(S)(\varphi)) = \mathcal{P}(f)^\Phi(\lambda_X(S))(\varphi). \end{aligned}$$

Third, we show that the pentagonal law holds for λ , i.e., for all $S \in \mathcal{P}(X^{\Phi^\Phi})$, $\varphi \in \Phi$ we have $\lambda_X(\mathcal{P}(\zeta_X)(S))(\varphi) = \zeta_{\mathcal{P}(X)}(\lambda_{X^\Phi}(\lambda_{X^\Phi}(S)))(\varphi)$. We compute:

$$\begin{aligned} \lambda_X(\mathcal{P}(\zeta_X)(S))(\varphi) &= \{C(\varphi) \mid C \in \mathcal{P}(\zeta_X)(S)\} \\ &= \{C(\varphi) \mid C \in \{\zeta_X(C') \mid C' \in S\}\} \\ &= \{C(\varphi) \mid C \in \{[\varphi' \mapsto C'(\varphi')(\varphi')] \mid C' \in S\}\} \\ &= \{C(\varphi) \mid C \in \lambda_{X^\Phi}(S)(\varphi)\} \\ &= \lambda_{X^\Phi}(\lambda_{X^\Phi}(S))(\varphi)(\varphi) = \zeta_{\mathcal{P}(X)}(\lambda_{X^\Phi}(\lambda_{X^\Phi}(S)))(\varphi). \end{aligned}$$

Lastly, we show that the triangular law holds for λ , i.e., for all $S \in \mathcal{P}(X)$ and $\varphi \in \Phi$ we have $\lambda_X(\mathcal{P}(\nu_X)(S))(\varphi) = \nu_{\mathcal{P}(X)}(S)(\varphi)$. Note $\nu_{\mathcal{P}(X)}(S)(\varphi) = S$. Then,

$$\begin{aligned} \lambda_X(\mathcal{P}(\nu_X)(S))(\varphi) &= \{C(\varphi) \mid C \in \mathcal{P}(\nu_X)(S)\} \\ &= \{C(\varphi) \mid C \in \{\nu_X(x) \mid x \in S\}\} = \{C(\varphi) \mid C \in \{[\varphi' \mapsto x] \mid x \in S\}\} = S. \end{aligned}$$

□

To illustrate how the extended functor $\overline{\mathcal{P}}$ is thus defined, we compute, for all $f: (X, \leq_X) \rightarrow (Y, \leq_Y)^{(\Phi, \leq_\Phi)}$, $S \in \mathcal{P}(X)$ and $\varphi \in \Phi$:

$$\begin{aligned} \overline{\mathcal{P}}f(S)(\varphi) &= \lambda_X \mathcal{P}f(S)(\varphi) = \{C(\varphi) \mid C \in \mathcal{P}f(S)\} \\ &= \{C(\varphi) \mid C \in \{f(s) \mid s \in S\}\} = \{f(S)(\varphi) \mid s \in S\} \end{aligned}$$

Now to model CTS with action labels (the case when $|A| > 1$) we use a distributive law between the functor $_{}^A$ and $_{}^\Phi$. The following result is a straightforward exercise in currying; however, we present it for the sake of completeness (this proof also works in the case of non-discretely ordered conditions).

Definition 7.4.4 *We define, for any $\varphi \in \Phi$, $a \in A$, and $S \in (_{}^\Phi)^A$, the distributive law $\kappa: (_{}^\Phi)^A \Rightarrow (_{}^A)^\Phi$ according to*

$$\kappa_X(S)(\varphi)(a) = S(a)(\varphi).$$

Proposition 7.4.5 *The above mapping κ results in a distributive law.*

Proof: To see that κ is a natural transformation, let $\varphi \in \Phi$, $a \in A$ and $S \in (_{}^\Phi)^A$. Then

$$\kappa \circ ((S^\Phi)^A)(a)(\varphi) = (S^A)^\Phi(\varphi)(a) = (S^A)^\Phi \circ \kappa(a)(\varphi).$$

For the pentagonal law, we find $\kappa((\zeta_X^A(S))(\varphi)(a) = \zeta_X^A(S)(a)(\varphi) = S(a)(\varphi)(\varphi)$ and

$$\begin{aligned} (\zeta_{X^A}(\kappa_{X^\Phi}(\kappa_{X^\Phi}(S)))(\varphi))(a) &= (\kappa_{X^\Phi}(\kappa_{X^\Phi}(S))(\varphi)(\varphi))(a) \\ &= (\kappa_{X^\Phi}(S)(\varphi))(a)(\varphi) = S(a)(\varphi)(\varphi). \end{aligned}$$

Lastly, for the triangular law, we derive

$$\kappa_X(\nu_X^A(S))(\varphi)(a) = \kappa_X^A(S)(a)(\varphi) = S(a) = (\nu_{X^A}(S)(\varphi))(a).$$

□

The extended functor $_{}^{\bar{A}}$ acts as follows on arrows $f: X \rightarrow Y^\Phi$, where $S \subseteq X^A$, $\varphi \in \Phi$ and $a \in A$:

$$f^{\bar{A}}(S)(\varphi)(a) = \kappa_X \circ (f^A(S))(\varphi)(a) = f^A(S)(a)(\varphi) = f(S(a))(\varphi)$$

As a result, a given CTS (X, A, α) over $(\Phi, =_\Phi)$ can be seen as a coalgebra $\alpha: X \rightarrow \mathcal{P}(X^A)$ in $\mathbf{Kl}(_{}^\Phi)$ with $\leq_X = =_X$. Next, we prove that behavioural equivalence for coalgebras in $\mathbf{Kl}(_{}^\Phi)$ characterises conditional bisimilarity between the states of a system.

Theorem 7.4.6 *Given a CTS $\alpha: X \rightarrow \bar{\mathcal{P}}(X)^{\bar{A}}$ over a discretely ordered set of conditions Φ , two states $x, y \in X$ are bisimilar if and only if there exists a coalgebra homomorphism $f: (X \rightarrow \bar{\mathcal{P}}(X)^{\bar{A}}) \rightarrow (Y \rightarrow \bar{\mathcal{P}}(Y)^{\bar{A}})$ such that $f(x) = f(y)$.*

Proof:

- Given coalgebra homomorphism $f: (X, \alpha) \rightarrow (Y, \beta)$, we define the family of relations R_φ (for each $\varphi \in \Phi$) as: $R_\varphi = \{(x, y) \mid f(x)(\varphi) = f(y)(\varphi)\}$. Next, we show that R_φ satisfy the transfer property of a bisimulation relation. Let $xR_\varphi y$ and $x \xrightarrow{a, \varphi} x'$. Then

$$\begin{aligned} f(x')(\varphi) &\in (\overline{\mathcal{P}}f^{\overline{A}} \circ \alpha)(x)(a)(\varphi) = (\beta \circ f)(x)(a)(\varphi) \\ &= \beta(f(x)(\varphi))(a)(\varphi) = \beta(f(y)(\varphi))(a)(\varphi) = (\beta \circ f)(y)(a)(\varphi) \\ &= (\overline{\mathcal{P}}f^{\overline{A}} \circ \alpha)(y)(a)(\varphi) = \overline{\mathcal{P}}f^{\overline{A}}(\alpha(y)(a)(\varphi))(\varphi). \end{aligned}$$

I.e., there exists a $y' \in \alpha(y)(a)(\varphi)$ such that $f(y') = f(x')$, i.e., $(x', y') \in R_\varphi$. Therefore, the family $\{R_\varphi \mid \varphi \in \Phi\}$ is a conditional bisimulation, since we are considering the discrete order, we do not show any inclusion property for this family.

- Given a family of bisimulation relations $\{R_\varphi \mid \varphi \in \Phi\}$, we define the equivalence classes $[x]_\varphi = \{y \in X \mid (x, y) \in R_\varphi\}$ (where we distinguish $[x]_\varphi$ and $[x]_{\varphi'}$, even if the partitions coincide) and set $Y = \{[x]_\varphi \mid x \in X, \varphi \in \Phi\}$ with $\leq_Y = =_Y$. Now let $f: X \rightarrow Y$ as $f(x)(\varphi) = [x]_\varphi$ and define a coalgebra $\beta: Y \rightarrow \overline{\mathcal{P}}(Y)^{\overline{A}}$ as

$$\beta([x]_\varphi)(a)(\varphi) = \{[y]_\varphi \mid \exists_{x', y'} x' \in [x]_\varphi \wedge y' \in [y]_\varphi \wedge x' \xrightarrow{a, \varphi} y'\}.$$

Clearly, f and β are order preserving, since Φ is discretely ordered. Lastly, to show that f is a coalgebra homomorphism, it suffices to show that $\beta([x]_\varphi)(a)(\varphi) = \{[y]_\varphi \mid x \xrightarrow{a, \varphi} y\}$, for any x, a, φ . The direction ' \supseteq ' is obvious, so consider $[y]_\varphi \in \beta([x]_\varphi)(a)(\varphi)$. Then, we find some x', y' such that $x' \in [x]_\varphi \wedge y' \in [y]_\varphi \wedge x' \xrightarrow{a, \varphi} y'$. Now using the transfer property of bisimulation, we find some y'' such that $x \xrightarrow{a, \varphi} y''$ and $y'' \in [y']_\varphi$, i.e., $[x]_\varphi \xrightarrow{a, \varphi} [y]_\varphi$.

□

It is important to note, that this approach only works if the set of conditions is discretely ordered. In this case, the conditions can be regarded purely as side effects, because the condition is set in the beginning and can never be changed, over the run of a system. This changes, however, when the order is non-discrete, since then upgrades, initiated by the user, may occur, which requires the conditions to be observable to some capacity.

7.5 Modelling CTS with Upgrades in $\mathbf{Kl}(_\Phi)$

In the previous section, we argued that the functor \mathcal{P} is not fit to model CTS with a non-discrete order, since the distributive law required to extend it to $\mathbf{Kl}(_\Phi)$ is not a collection of arrows in case the order is non-discrete. The reason can intuitively be understood as follows: Using the functor \mathcal{P} , the version of the system is hidden as a side effect and therefore not considered in the behavioural equivalence – after all this is the purpose of modelling using a Kleisli category. However, while the versions can be seen as side effects in the absence of upgrades, this is no longer true when a CTS can perform an upgrade. Upgrades are monotone by nature, so in order to perform an upgrade, one must be aware of the current version of the system. Alternatively, one can observe, that in the characterisation via the conditional bisimulation game, an upgrade must be answered by an upgrade to the same system (see [BKKS17]).

Consequently, a functor that is fit to model CTS with upgrades has to carry information regarding the current version the system is operated with. To this end, we define a functor that is similar to \mathcal{P} , but collects successor states together with the products they can be reached in. The functor will again arise via an extension, but we will later also define the functor explicitly on the category.

Definition 7.5.1 *We define the **Poset** endofunctor $\mathcal{P}(_\times\Phi)$ as follows:*

Objects $((\mathcal{P}(X \times \Phi)), \subseteq)$, for an object (X, \leq_X) in **Poset**

Arrows For an arrow $f: X \rightarrow Y$ in **Poset**,

$$\mathcal{P}(f \times \Phi): (\mathcal{P}(X \times \Phi)) \rightarrow (\mathcal{P}(Y \times \Phi))$$

is a function

$$\mathcal{P}(f \times \Phi): (\mathcal{P}(X \times \Phi)) \rightarrow (\mathcal{P}(Y \times \Phi)),$$

$$\mathcal{P}(f \times \Phi)(p) = \{(f(x), \varphi) \mid (x, \varphi) \in p\},$$

for all $p \in (\mathcal{P}(X \times \Phi))$.

It is easy to see that this is an endofunctor:

Lemma 7.5.2 $\mathcal{P}(_ \times \Phi)$ is a functor.

Proof: It is well-known that inclusion is a partial order, so we will not show this. Also, where the order of the sets is not important in the following parts of the proof, we will omit them for ease of reading.

- $\mathcal{P}(_ \times \Phi)$ maps monotone functions to monotone functions. Let $f: X \rightarrow Y$ be a monotone function, and $p, q \in \mathcal{P}(X \times \Phi)$ such that $p \leq q$, i.e. $p \subseteq q$. Then $\mathcal{P}(f \times \Phi)(p) = \{(f(x), \varphi) \mid (x, \varphi) \in p\} \subseteq \{(f(x), \varphi) \mid (x, \varphi) \in q\} = \mathcal{P}(f \times \Phi)(q)$, i.e. $\mathcal{P}(f \times \Phi)(p) \leq \mathcal{P}(f \times \Phi)(q)$.
- $\mathcal{P}(_ \times \Phi)$ preserves identities. Let $p \in \mathcal{P}(X \times \Phi)$, then $\mathcal{P}(\text{id}_X \times \Phi)(p) = \{(\text{id}_X(x), \varphi) \mid (x, \varphi) \in p\} = p$.
- $\mathcal{P}(_ \times \Phi)$ respects composition. Let $f: X \rightarrow Y$, $g: Y \rightarrow Z$, and $p \in \mathcal{P}(X \times \Phi)$ then $\mathcal{P}(g \circ f \times \Phi)(p) = \{(g(f(x)), \varphi) \mid (x, \varphi) \in p\}$. On the other hand, $\mathcal{P}(g \times \Phi) \circ \mathcal{P}(f \times \Phi)(p) = \{(g(y), \varphi) \mid (x, \varphi) \in \mathcal{P}(f \times \Phi)(p)\} = \{(g(y), \varphi) \mid (x, \varphi) \in \{(f(x), \varphi) \mid (x, \varphi) \in p\}\} = \{(g(f(x)), \varphi) \mid (x, \varphi) \in p\} = \mathcal{P}(g \circ f \times \Phi)(p)$.

□

Now, we will show that this functor can be extended to $\mathbf{Kl}(_^\Phi)$ via a distributive law:

Definition 7.5.3 We define the distributive law $\iota: \mathcal{P}(_^\Phi \times \Phi) \Rightarrow \mathcal{P}(_ \times \Phi)^\Phi$ where $\iota_X: \mathcal{P}(X^\Phi \times \Phi) \Rightarrow \mathcal{P}(X \times \Phi)^\Phi$ is defined according to

$$\iota_X(S)(\varphi) = \{(f(\varphi'), \varphi') \mid (f, \varphi') \in S\}$$

for all $S \in \mathcal{P}(X^\Phi \times \Phi)$, $\varphi \in \Phi$.

Lemma 7.5.4 ι is a distributive law.

Proof: We need to show several parts:

- ι_X is an arrow: For this we need to show monotonicity in both arguments. So first, let $\varphi \in \Phi$ and $S, S' \in \mathcal{P}(X^\Phi \times \Phi)$ be given, such that $S \leq S'$, i.e. $S \subseteq S'$. Then,

$$\begin{aligned} \iota_X(S)(\varphi) &= \{(f(\varphi'), \varphi') \mid (f, \varphi') \in S\} \\ &\subseteq \{(f(\varphi'), \varphi') \mid (f, \varphi') \in S'\} = \iota_X(S')(\varphi). \end{aligned}$$

Monotonicity in the second argument is trivial, since ι_X is independent of the second argument, $\iota_X(S)(\varphi) = \iota_X(S)(\varphi')$ for all $S \in \mathcal{P}(X^\Phi \times \Phi)$ and all $\varphi, \varphi' \in \Phi$, not necessarily ordered.

- ι_X is a natural transformation: Let $S \in \mathcal{P}(X^\Phi \times \Phi)$ and $\varphi \in \Phi$, then we can compute:

$$\begin{aligned}
 & (\mathcal{P}(f \times X))^\Phi(\iota_X(S))(\varphi) \\
 &= \mathcal{P}(f \times X)(\iota_X(S))(\varphi) \\
 &= \{(f(x), \varphi') \mid (x, \varphi') \in \iota_X(S)(\varphi)\} \\
 &= \{(f(x), \varphi') \mid (x, \varphi') \in \{(g(\varphi'), \varphi') \mid (g, \varphi') \in S\}\} \\
 &= \{(f(g(\varphi')), \varphi') \mid (f, \varphi') \in S\}
 \end{aligned}$$

as well as:

$$\begin{aligned}
 & \iota_X(\mathcal{P}(f^\Phi \times \Phi))(S)(\varphi) \\
 &= \{(h(\varphi'), \varphi') \mid (h, \varphi') \in \mathcal{P}(f^\Phi \times \Phi)(S)\} \\
 &= \{(h(\varphi'), \varphi') \mid (h, \varphi') \in \{(f^\Phi(g), \varphi') \mid (g, \varphi') \in S\}\} \\
 &= \{(h(\varphi'), \varphi') \mid (h, \varphi') \in \{(f \circ g, \varphi') \mid (g, \varphi') \in S\}\} \\
 &= \{(f(g(\varphi')), \varphi') \mid (g, \varphi') \in S\}
 \end{aligned}$$

- The pentagonal law holds: Let $\varphi \in \Phi$ and $S \in \mathcal{P}(X^{\Phi^\Phi} \times \Phi)$, then we can compute:

$$\begin{aligned}
 & \iota_X(\mathcal{P}(\zeta_X \times \Phi)(S))(\varphi) = \{(f(\varphi'), \varphi') \mid (f, \varphi') \in \mathcal{P}(\zeta_X \times \Phi)(S)\} \\
 &= \{(f(\varphi'), \varphi') \mid (f, \varphi') \in \{(\zeta_X(g), \varphi') \mid (g, \varphi') \in S\}\} \\
 &= \{(g(\varphi')(\varphi'), \varphi') \mid (g, \varphi') \in S\}
 \end{aligned}$$

as well as:

$$\begin{aligned}
 & \zeta_{\mathcal{P}(X \times \Phi)}(\iota_X(\iota_X(S)))(\varphi) = \iota_X(\iota_X(S))(\varphi)(\varphi) \\
 &= \{(f(\varphi'), \varphi') \mid (f, \varphi') \in \iota_X(S)(\varphi)\} \\
 &= \{(f(\varphi'), \varphi') \mid (f, \varphi') \in \{(g(\varphi'), \varphi') \mid (g, \varphi') \in S\}\} \\
 &= \{(g(\varphi')(\varphi'), \varphi') \mid (g, \varphi') \in S\}
 \end{aligned}$$

- The triangular law holds:

$$\begin{aligned}
 & \iota_X(\mathcal{P}(\nu_X \times \Phi))(S)(\varphi) = \{(f(\varphi'), \varphi') \mid (f, \varphi') \in \mathcal{P}(\nu_X \times \Phi)(S)\} \\
 &= \{(f(\varphi'), \varphi') \mid (f, \varphi') \in \{(\nu_X(x), \varphi') \mid (x, \varphi') \in S\}\} \\
 &= \{(\nu_X(x)(\varphi'), \varphi') \mid (x, \varphi') \in S\} = \{(x, \varphi') \mid (x, \varphi') \in S\} = S
 \end{aligned}$$

By, definition $\nu_{\mathcal{P}(X \times \Phi)}(S)(\varphi) = S$, so the triangular law holds, as well. \square

Together with the extension of the functor $_{}^A$ for the set of actions A , we obtain the CTS functor F . To make further arguments easier, we give an explicit definition of the functor F we will use directly on $\mathbf{Kl}(_{}^\Phi)$. One can see that this definition is capturing exactly the functor constructed via the extension of $\mathcal{P}(_{} \times X)$ concatenated with the extension of $_{}^A$ via the following argument: Let $f: (X, \leq_X) \rightarrow (Y, \leq_Y)^{(\Phi, \leq_\Phi)}$, $S \in \mathcal{P}(X \times \Phi)$, $\varphi \in \Phi$, then we can compute:

$$\begin{aligned} \overline{\mathcal{P}(f \times \Phi)}(S)(\varphi) &= \iota_X \circ \mathcal{P}(f \times \Phi)(S)(\varphi) \\ &= \{(g(\varphi'), \varphi') \mid (g, \varphi') \in \mathcal{P}(f \times \Phi)(S)\} \\ &= \{(g(\varphi'), \varphi') \mid (g, \varphi') \in \{(f(x), \varphi') \mid (x, \varphi') \in S\}\} \\ &= \{(f(x)(\varphi'), \varphi') \mid (x, \varphi') \in S\} \end{aligned}$$

Definition 7.5.5 Consider the following mapping $F: \mathbf{Kl}(_{}^\Phi) \rightarrow \mathbf{Kl}(_{}^\Phi)$ defined according to:

Objects $F(X) = ((\mathcal{P}(X \times \Phi))^A, \leq_{FX})$, for an object (X, \leq_X) in **Poset**, where for all $g, h: (\mathcal{P}(X \times \Phi))^A$ it holds that $g \leq_{FX} h$ iff $g(a) \subseteq h(a)$ for all $a \in A$.

Arrows For an arrow $f: X \rightarrow Y$ in $\mathbf{Kl}(_{}^\Phi)$, $Ff: (\mathcal{P}(X \times \Phi))^A \rightarrow (\mathcal{P}(Y \times \Phi))^A$ is a function $Ff: (\mathcal{P}(X \times \Phi))^A \rightarrow ((\mathcal{P}(Y \times \Phi))^A)^\Phi$, where

$$Ff(p)(\varphi)(a) = \{(f(x)(\varphi'), \varphi') \mid (x, \varphi') \in p(a)\},$$

for all $p \in (\mathcal{P}(X \times \Phi))^A$, $\varphi \in \Phi$ and $a \in A$.

Now a CTS $(X, \Phi, A, \rightarrow)$ over a finite set of states X (that is ordered discretely), a finite set of actions A , and a partially ordered set of conditions Φ is modelled by the following arrow $\alpha: X \rightarrow FX$ in $\mathbf{Kl}(_{}^\Phi)$:

$$\alpha(x)(\varphi)(a) = \{(x', \varphi') \mid x \xrightarrow{a, \varphi'} x' \wedge \varphi' \leq \varphi\} .$$

As can be observed above, Ff is just a pure arrow, i.e. independent of the argument $\varphi \in \Phi$, so the definition of α must account for the filtering for the

next states. The modelling can be understood as follows: given a state x , an action a and a current version φ , we consider all possible upgrades $\varphi' \leq \varphi$ that could be performed and collect all states x' that can be reached under φ' . We then collect all those pairs of follow-up state and upgraded product. Note that not doing an upgrade, but continuing with the current version φ is permissible as well. From this intuition it can easily be seen that α is an arrow in $\mathbf{Kl}(_^\Phi)$, because we assume that the state set X is ordered discretely, rendering monotonicity in the first argument trivial, and α grows in its second argument φ . Indeed it is immediate that whenever $\varphi' \leq \varphi$,

$$\begin{aligned} \alpha(x)(\varphi')(a) &= \{(x', \varphi'') \mid x \xrightarrow{a, \varphi''} x' \wedge \varphi'' \leq \varphi'\} \\ &\subseteq \{(x', \varphi'') \mid x \xrightarrow{a, \varphi''} x' \wedge \varphi'' \leq \varphi\} = \alpha(x)(\varphi)(a) \end{aligned}$$

We now prove the main result of this section; namely, that behavioural equivalence coincides with conditional bisimulation.

Theorem 7.5.6 *Let $\alpha: X \rightarrow FX$ in $\mathbf{Kl}(_^\Phi)$ be a CTS where \leq_X is equality. Then, two states $x, x' \in X$ are conditionally bisimilar under $\varphi \in \Phi$ ($x \sim_\varphi y$) if and only if there exists a coalgebra homomorphism $f: (X, \alpha) \rightarrow (Y, \beta)$ (for some $\beta: Y \rightarrow FY$) in $\mathbf{Kl}(_^\Phi)$ such that $f(x)(\varphi) = f(x')(\varphi)$.*

Proof: $\boxed{\Leftarrow}$ We define a family of relations $R_\varphi \subseteq X \times X$ (for each $\varphi \in \Phi$) as follows: $x R_\varphi x'$ if and only if there *exists* a coalgebra homomorphism $f: (X, \alpha) \rightarrow (Y, \beta)$ in $\mathbf{Kl}(_^\Phi)$ such that $f(x)(\varphi) = f(x')(\varphi)$. We first show that each R_φ satisfies the transfer property of a traditional bisimulation relation. Let $x_1 R_\varphi x_2$ and $x_1 \xrightarrow{a, \varphi} x'_1$. Then, by the construction of R_φ we have some coalgebra homomorphism $f: (X, \alpha) \rightarrow (Y, \beta)$ in $\mathbf{Kl}(_^\Phi)$ with $f(x_1)(\varphi) = f(x_2)(\varphi)$. Thus, $Ff(\alpha(x_1)(\varphi))(\varphi) = \beta(f(x_1)(\varphi))(\varphi) = \beta(f(x_2)(\varphi))(\varphi) = Ff(\alpha(x_2)(\varphi))(\varphi)$. Then

$$(x'_1, \varphi) \in \alpha(x_1)(\varphi)(a) \implies (f(x'_1)(\varphi), \varphi) \in Ff(\alpha(x_2)(\varphi))(\varphi)(a).$$

I.e., there is some $(x'_2, \varphi') \in \alpha(x_2)(\varphi)(a)$ such that $(f(x'_2)(\varphi'), \varphi') = (f(x'_1)(\varphi), \varphi)$. Therefore, $\varphi = \varphi'$ and $f(x'_1)(\varphi) = f(x'_2)(\varphi)$. Hence, $x'_1 R_\varphi x'_2$.

Next, we need to show that, if $\varphi_2 \leq_\Phi \varphi_1$, then $R_{\varphi_1} \subseteq R_{\varphi_2}$. So let $x_1 R_{\varphi_1} x_2$ and $\varphi_2 \leq_\Phi \varphi_1$. Then, by the construction of R_{φ_1} we have some coalgebra homomorphism $f: (X, \alpha) \rightarrow (Y, \beta)$ in $\mathbf{Kl}(_^\Phi)$, such that $f(x_1)(\varphi_1) = f(x_2)(\varphi_1)$.

Thus, for any $a \in A$, we have

$$\{(f(x)(\varphi), \varphi) \mid \varphi \leq \varphi_1 \wedge x_1 \xrightarrow{a, \varphi} x\} = \{(f(x)(\varphi), \varphi) \mid \varphi \leq \varphi_1 \wedge x_2 \xrightarrow{a, \varphi} x\}.$$

Note, that $\{(f(x)(\varphi), \varphi) \mid \varphi \leq \varphi_2 \wedge x_1 \xrightarrow{a, \varphi} x\} = \{(f(x)(\varphi), \varphi) \mid \varphi \leq \varphi_2 \wedge x_2 \xrightarrow{a, \varphi} x\}$, since $\varphi_2 \leq \varphi_1$. Thus, $(Ff \circ \alpha)(x_1)(\varphi_2) = (Ff \circ \alpha)(x_2)(\varphi_2)$, i.e., $\beta(f(x_1)(\varphi_2)) = \beta(f(x_2)(\varphi_2))$. Now, taking into consideration that every coalgebra is a coalgebra homomorphism itself and that compositions of coalgebra homomorphisms are coalgebra homomorphisms, as well, $\beta \circ f$ is a coalgebra homomorphism witnessing $(x_1, x_2) \in R_{\varphi_2}$.

\Rightarrow Let R_φ (for $\varphi \in \Phi$) be a family of largest bisimulation relations (which exists, due to Lemma 6.3.5) on the given CTS such that $R_\varphi \subseteq R_{\varphi'}$, whenever $\varphi' \leq_\Phi \varphi$. Since the largest bisimulation relation is an equivalence relation on the set of states, we can define a function $g: X \times \Phi \rightarrow \mathcal{P}(X)$ according to $g(x, \varphi) = \{x' \mid x R_\varphi x'\}$. Moreover, we write $[x]_\varphi := (g(x, \varphi), \varphi)$ and define the set $Y = \{[x]_\varphi \mid x \in X, \varphi \in \Phi\}$, ordered by³ $\supseteq \times \leq_\Phi$. Now we can define a coalgebra $\beta: Y \rightarrow FY$ according to

$$\beta([x]_{\varphi_1})(\varphi_2)(a) = \{([y]_{\varphi'}, \varphi') \mid (y, \varphi) \in \alpha(x')(\varphi')(a), \varphi' \leq \varphi_1, \varphi' \leq \varphi_2, x R_{\varphi'} x'\}$$

as well as a matching coalgebra homomorphism $f: X \rightarrow Y$ where

$$f(x)(\varphi) = [x]_\varphi.$$

Now we need to show that β and f are arrows and that f is indeed a coalgebra homomorphism from α to β .

To show that β is an arrow, we show that it is well-defined and monotone in both arguments. We can observe that β is well-defined by showing that its definition is independent of the chosen representative x , so let $[x_1]_{\varphi_1} = [x_2]_{\varphi_1}$, i.e. $x_1 R_{\varphi_1} x_2$. Then, in the definition of β , it follows from $\varphi' \leq \varphi_1$ that $R_{\varphi'} \supseteq R_{\varphi_1}$, i.e. $x_1 R_{\varphi'} x_2$, and, using transitivity, $x_1 R_{\varphi'} x' \wedge x_1 R_{\varphi'} x_2 \Rightarrow x_2 R_{\varphi'} x'$.

Now, to show that β is monotone in the first argument, observe that $[\bar{x}]_{\varphi_1} \leq [x]_{\varphi'_1}$ implies $\varphi_1 \leq \varphi'_1$, and that $x R_{\varphi_1} \bar{x}$, i.e. $[\bar{x}]_{\varphi_1} = [x]_{\varphi_1}$. Thus, using transitivity in the definition of β , we can observe that whenever $\varphi' \leq \varphi_1$, then

³Since from $\varphi' \leq \varphi$ it follows that $g(x, \varphi') \supseteq g(x, \varphi)$, it is sufficient to just check the second component of the order.

also $\varphi' \leq \varphi'_1$, proving that $\beta([\bar{x}]_{\varphi_1})(\varphi_2)(a) = \beta([x]_{\varphi_1})(\varphi_2)(a) \subseteq \beta([x]_{\varphi'_1})(\varphi_2)(a)$ holds for all $\varphi_2 \in \Phi$.

Lastly, monotonicity of β in the second argument can be proven by observing that there is only one place, where the definition of β depends on the second argument, i.e. when checking that $\varphi' \leq \varphi_2$. Now, if $\varphi_2 \leq \varphi'_2$, by transitivity $\varphi' \leq \varphi'_2$, as well, proving that for all $[x]_{\varphi_1} \in Y$, $\beta([x]_{\varphi_1})(\varphi_2)(a) \subseteq \beta([x]_{\varphi_1})(\varphi'_2)(a)$ holds.

Monotonicity of f in the first argument is trivial, since X is discretely ordered, so we only need to show monotonicity in the second argument, but by definition of Y 's order and conditional bisimilarity, whenever $\varphi' \leq \varphi$, then $f(x)(\varphi') = [x]_{\varphi'}(\supseteq \times \leq_\Phi)[x]_\varphi = f(x)(\varphi)$.

So, it only remains to show that f is a coalgebra homomorphism. For this purpose, let any $x \in X$, $\varphi \in \Phi$, $a \in A$ be given, then compute

$$\begin{aligned} Ff \circ \alpha(x)(\varphi)(a) &= \{(f(y)(\varphi'), \varphi') \mid (y, \varphi') \in \alpha(x)(\varphi)(a)\} \\ &= \{([y]_{\varphi'}, \varphi') \mid (y, \varphi') \in \alpha(x)(\varphi)(a)\} \end{aligned}$$

as well as:

$$\begin{aligned} \beta \circ f(x)(\varphi)(a) &= \beta(f(x)(\varphi))(\varphi)(a) = \beta([x]_\varphi)(\varphi)(a) \\ &= \{([y]_{\bar{\varphi}}, \bar{\varphi}) \mid (y, \bar{\varphi}) \in \alpha(x')(\varphi')(a), \varphi' \leq \varphi, x R_{\varphi'} x'\} \end{aligned}$$

Since $R_{\varphi'}$ is reflexive, it is clear that $\beta \circ f(x)(\varphi)(a) \supseteq Ff \circ \alpha(x)(\varphi)(a)$ holds, so it remains to show that also $\beta \circ f(x)(\varphi)(a) \subseteq Ff \circ \alpha(x)(\varphi)(a)$ is true.

To show this, let $(y, \bar{\varphi}) \in \alpha(x')(\varphi')(a)$, $\varphi' \leq \varphi$ and $x R_{\varphi'} x'$ be given, then we need to show that there exists also a $(y', \bar{\varphi}) \in \alpha(x)(\varphi)(a)$ such that $y R_{\bar{\varphi}} y'$. By definition of α we can see that $(y, \bar{\varphi}) \in \alpha(x')(\varphi')(a)$ means that $\bar{\varphi} \leq \varphi'$. Since $x R_{\varphi'} x'$, by definition of conditional bisimulation, there must exist a $(y', \bar{\varphi}) \in \alpha(x)(\varphi')(a)$ such that $y' R_{\bar{\varphi}} y$ and due to monotonicity of α we can finally conclude $(y', \bar{\varphi}) \in \alpha(x)(\varphi)(a)$, concluding the proof that f is a coalgebra homomorphism from α to β . □

So F indeed offers a way of modelling CTS in the presence of upgrades. Note, however, that F -coalgebras exist that are not CTS themselves. As a consequence the above result is limited to systems modelled as described before.

7.6 Computing Behavioural Equivalence

In this section, we concentrate on algorithms to obtain a minimal CTS from a given CTS up to conditional bisimilarity. We will show that the final chain Algorithm C (using pseudo-factorisation, cf. Algorithm 3.3.8 and Definition 2.4.28) is applicable to the powerset functor \mathcal{P} , as well as the functor F (Definition 7.5.5).

For our purpose, we need to show that **Poset** is a reflective subcategory of $\mathbf{Kl}(_^\Phi)$, in order to construct a pseudo-factorisation. To this end, we first note that an arrow $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$ in **Poset** can be interpreted as a ‘pure’ arrow $f_\Phi: (X, \leq_X) \rightarrow (Y, \leq_Y)^\Phi$ in $\mathbf{Kl}(_^\Phi)$, simply by letting $f_\Phi(x)(\varphi) = f(x)$, for all $\varphi \in \Phi$. Clearly, the pure arrow f_Φ is order-preserving just because f is. Thus, **Poset** is a subcategory of $\mathbf{Kl}(_^\Phi)$.

Due to the Kleisli adjunction it is well-known that **Poset** is a coreflective subcategory of $\mathbf{Kl}(_^\Phi)$. Here we show that it is reflective as well.

Theorem 7.6.1 *The category **Poset** is a reflective subcategory of $\mathbf{Kl}(_^\Phi)$.*

Proof: For an object (X, \leq_X) in $\mathbf{Kl}(_^\Phi)$, we define its **Poset**-reflection $\rho_X: X \rightarrow (X \times \Phi)^\Phi$ as $\rho_X(x)(\varphi) = (x, \varphi)$. The set $X \times \Phi$ is ordered as follows: $(x, \varphi) \leq_{X \times \Phi} (x', \varphi') \iff x \leq_X x' \wedge \varphi \leq \varphi'$. Now for an arrow $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$ in $\mathbf{Kl}(_^\Phi)$, we need to find its ρ -reflection $f': (X \times \Phi) \rightarrow Y$. So let $f'(x, \varphi) = f(x)(\varphi)$, for any $x \in X$ and $\varphi \in \Phi$. To show that f' is order-preserving, let $x \leq_X x'$ and $\varphi \leq \varphi'$. Then, we find $f(x)(\varphi) \leq_Y f(x')(\varphi)$ and $f(x')(\varphi) \leq_Y f(x')(\varphi')$ since f is an arrow in $\mathbf{Kl}(_^\Phi)$. Thus, $f'(x, \varphi) = f(x)(\varphi) \leq_Y f(x')(\varphi') = f'(x', \varphi')$. Moreover, for any x, φ we have

$$f(x)(\varphi) = f'(x, \varphi) = f'(\rho_X(x)(\varphi)) = f'_\Phi(\rho_X(x)(\varphi))(\varphi) = f'_\Phi \circ \rho_X(x)(\varphi).$$

Now assume an order-preserving function $f'': X \times \Phi \rightarrow Y$ such that $f''_\Phi \circ \rho_X = f$. Then, for any x, φ we find that $f''(x, \varphi) = f(x)(\varphi) = f'(x, \varphi)$. Thus, $f' = f''$.

□

Following [ABH⁺12], a reflective subcategory that has an $(\mathcal{E}, \mathcal{M})$ -factorisation structure gives rise to a pseudo-factorisation structure (Definition 2.4.28) in the base category, which in turn can be used to compute behavioural equivalence, provided the functor meets some conditions as outlined in Chapter 3.

Definition 7.6.2 *We define a factorisation structure (Definition 2.4.26) for **Poset** in the following way:*

- An order-preserving function $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$ is in \mathcal{E} if and only if $f: X \rightarrow Y$ is surjective in **Set**, and \leq_Y is the smallest order satisfying $f(x_1) \leq_Y f(x_2)$ whenever $x_1 \leq_X x_2$. In other words, $y \leq_Y y'$ (for $y, y' \in Y$) if and only if there are $x_1, \dots, x_n, x'_1, \dots, x'_{n-1}$, such that $f(x_1) = y, f(x_n) = y', f(x_i) = f(x'_i)$ and $x'_i \leq_X x_{i+1}$ for all $i \in [1, n]$.
- An order-preserving function $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$ is in \mathcal{M} if and only if $f: X \rightarrow Y$ is injective in **Set**.

Lemma 7.6.3 *The tuple $(\mathcal{E}, \mathcal{M})$ from Definition 7.6.2 is a factorisation structure for **Poset**.*

Proof: Showing that the classes \mathcal{E} and \mathcal{M} are closed under composition with isomorphisms is standard. For instance, an isomorphism (order-preserving bijection) composed with an order-preserving surjection (injection) is again an order-preserving surjection (injection).

- An arrow $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$ can be factorised into an arrow $e: (X, \leq_X) \rightarrow (\text{Im}(f), \leq_{\text{Im}(f)})$ and an arrow $m: (\text{Im}(f), \leq_{\text{Im}(f)}) \rightarrow (Y, \leq_Y)$ just like in **Set**. Here, $\leq_{\text{Im}(f)}$ is the smallest order which makes the function e an order-preserving one, i.e., $y_1 \leq_{\text{Im}(f)} y_2$ if and only if there are $x_1, \dots, x_n, x'_1, \dots, x'_{n-1}$, such that $f(x_1) = y_1, f(x_n) = y_2$, and $x'_i \leq_X x_{i+1}, f(x_i) = f(x'_i)$, for all $i \in [1, n]$.

The relation $\leq_{\text{Im}(f)}$ is reflexive, because e is surjective and $\leq_{\text{Im}(f)}$ is transitive by construction. Lastly, antisymmetry of $\leq_{\text{Im}(f)}$ follows directly from \leq_Y .

- The unique diagonal property holds. Let $e: (X, \leq_X) \rightarrow (Y, \leq_Y) \in \mathcal{E}$, $m: (X', \leq_{X'}) \rightarrow (Y', \leq_{Y'}) \in \mathcal{M}$, $f: (Y, \leq_Y) \rightarrow (Y', \leq_{Y'})$, and $g: (X, \leq_X) \rightarrow (X', \leq_{X'})$, such that $f \circ e = m \circ g$ there exists a unique diagonal arrow $d: (Y, \leq_Y) \rightarrow (X', \leq_{X'})$ such that $d \circ e = g$ and $m \circ d = f$. Since any such diagonal arrow would also be a diagonal arrow in **Set**, uniqueness is clear, we only need to show the existence of the diagonal arrow. Define a function $d(y) = g(x) \iff e(x) = y$. Since e is surjective, d is well-defined. Now, we show that d is order-preserving. Let $a \leq b$, for some $a, b \in Y$. Then there are $x_1, \dots, x_n, x'_1, \dots, x'_{n-1}$ such that $x'_i \leq x_{i+1}, f(x_1) = a, f(x_n) = b, f(x_i) = f(x'_i)$. We know that, since g is

order-preserving, $g(x'_i) \leq g(x_{i+1})$ and want to show that for all $1 \leq i < n$ we have $g(x_i) = g(x'_i)$. Assume, on the contrary, that $g(x'_i) \neq g(x_i)$ for any i , then $m \circ g(x_i) \neq m \circ g(x'_i)$, because m is injective. Then, $f \circ e(x_i) \neq f \circ e(x'_i)$, but this is a contradiction, because $e(x_i) = e(x'_i)$. So, we can conclude: $d(a) = d(e(x_1)) = g(x_1) \leq g(x_n) = d(e(x_n)) = d(b)$.

□

Now that we have seen that there is a factorisation structure in **Poset** and that it can be lifted to $\mathbf{Kl}(_\Phi)$, we want to make explicit what it means to factorise in $\mathbf{Kl}(_\Phi)$.

Theorem 7.6.4 *Recall ρ_X from Theorem 7.6.1 and consider the following data:*

- An arrow $f: X \rightarrow Y$ given in $\mathbf{Kl}(_\Phi)$ with the set $Y_0 = \{f(x)(\varphi) \mid x \in X \wedge \varphi \in \Phi\}$.
- The function $e': X \rightarrow Y_0^\Phi$ defined as $e'(x)(\varphi) = f(x)(\varphi)$, for every $x \in X, \varphi \in \Phi$.
- The relation $\leq_{Y_0^\Phi}$ is the smallest order such that e' is order-preserving.
- The function $m: Y_0 \rightarrow Y$ defined as $m(y) = y$, for every $y \in Y_0$.

Then, $m_\Phi \circ e'$ is a pseudo-factorisation of f .

Proof: Given an arrow $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$ in $\mathbf{Kl}(_\Phi)$. Then, recall from the proof of Theorem 7.6.1, we have $\rho_X: (X, \leq_X) \rightarrow (X \times \Phi, \leq_{X \times \Phi})^\Phi$ and a unique order-preserving function $f': X \times \Phi \rightarrow Y$ such that $f = f'_\Phi \circ \rho_X$. Now factorising f' in **Poset** we get the decomposition $m \circ e$ with the functions $e: (X \times \Phi, \leq_{X \times \Phi}) \rightarrow (Y_0, \leq_{Y_0})$ and $m: (Y_0, \leq_{Y_0}) \rightarrow (Y, \leq_Y)$ defined in the obvious way. Note, that the order \leq_{Y_0} is the smallest order which makes e order-preserving. Then, interpreting these functions as pure arrows we get $e' = e_\Phi \circ \rho$. Thus, $m_\Phi \circ e' = f$. □

Algorithm C, where the pseudo-factorisation structure above is used for factorisation, cf. [ABH⁺12], is applicable, if the functor preserves the reflective subcategory and the class \mathcal{M} – it is obvious that \mathcal{M} maps to injective functions via the concretisation functor. We will now go on to show that the algorithm

is applicable to both CTS modelling techniques presented, starting with the variant without upgrades.

Lemma 7.6.5 *The functor \mathcal{P} (cf. Definition 7.4.1) preserves **Poset** and \mathcal{M} .*

Proof: To show that \mathcal{P} preserves **Poset**, let $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$ be an arrow in **Poset**. Then, we need to show that $\mathcal{P}(f_\Phi): \mathcal{P}(X) \rightarrow \mathcal{P}(Y)$ is a pure arrow. So let $U \subseteq X$ and $\varphi, \varphi' \in \Phi$. Then $\mathcal{P}(f_\Phi)(U)(\varphi) = \{f_\Phi(x)(\varphi) \mid x \in U\} = \{f_\Phi(x)(\varphi') \mid x \in U\} = \mathcal{P}(f_\Phi)(U)(\varphi')$.

Lastly, to show that \mathcal{P} preserves \mathcal{M} , let $m: X \rightarrow Y$ be an order-preserving injection. From above, we find $\mathcal{P}(m_\Phi)$ is an arrow in **Poset**. Let $\mathcal{P}(m_\Phi)(U)(\varphi) = \mathcal{P}(m_\Phi)(U')(\varphi)$. Then, we find $m(U) = m(U')$ and injectivity of m gives $U = U'$. Thus, $\mathcal{P}(m_\Phi)$ is also injective. \square

The next two lemmas show that Algorithm C (Algorithm 3.3.8) is applicable to compute the greatest conditional bisimilarity for the systems as F -coalgebras.

Lemma 7.6.6 *The functor F (cf. Definition 7.5.5) preserves **Poset** and \mathcal{M} .*

Proof:

Let $f: (X, \leq_X) \rightarrow (X, \leq_X)$ be an arrow in **Poset**. Then we need to show that $F(f_\Phi): F(X) \rightarrow F(Y)$ is a pure arrow. So let $U \subseteq X \times \Phi$ and $\varphi, \varphi' \in \Phi$, then

$$F(f_\Phi)(U)(\varphi) = \{(f_\Phi(x)(\tilde{\varphi}), \tilde{\varphi}) \mid (x, \tilde{\varphi}) \in U\} = F(f_\Phi)(U)(\varphi').$$

To show that F preserves \mathcal{M} , as well, let m be an order-preserving injection. As shown above, Fm_Φ is an arrow in **Poset**. To show that Fm_Φ is injective, let $p, q \in (\mathcal{P}(X \times \Phi))^A$ be given. If $Fm_\Phi(p) = Fm_\Phi(q)$, then for all $\varphi \in \Phi$ and $a \in A$, $Fm_\Phi(p)(\varphi)(a) = Fm_\Phi(q)(\varphi)(a)$. Thus, we compute

$$\begin{aligned} Fm_\Phi(p)(\varphi)(a) &= \{(m_\Phi(x)(\varphi'), \varphi') \mid (x, \varphi') \in p(a)\} \\ &= \{(m_\Phi(x)(\varphi'), \varphi') \mid (x, \varphi') \in q(a)\} = Fm_\Phi(q)(\varphi)(a). \end{aligned}$$

I.e., for all $(x, \varphi') \in p(a)$, there must exist a $(y, \varphi') \in q(a)$ such that $m_\Phi(x, \varphi') = m_\Phi(y, \varphi')$. Since m is injective, it follows $(x, \varphi') = (y, \varphi')$ and therefore $p(a) \subseteq q(a)$, and, analogously $q(a) \subseteq p(a)$. Thus, $q(a) = p(a)$ for all a , i.e. $q = p$. Hence, Fm_Φ is indeed injective. \square

Lemma 7.6.7 *The concretisation functor U for $\mathbf{Kl}(_^\Phi)$ maps \mathcal{M} to monos.*

Proof: Given an arrow $m: Y \rightarrow Z \in \mathcal{M}$ in $\mathbf{Kl}(_^\Phi)$ – note that m is then an injective function – and two functions $e, e' \in X \rightarrow Y^\Phi$. Then

$$\begin{aligned} Um \circ e &= Um \circ e' \Leftrightarrow \zeta_Z m^\Phi \circ e = \zeta_Z m^\Phi \circ e' \\ \Leftrightarrow \forall x \in X, \varphi \in \Phi : (\zeta_Z m^\Phi \circ e)(x)(\varphi) &= (\zeta_Z m^\Phi \circ e')(x)(\varphi) \\ \Leftrightarrow \forall x \in X, \varphi \in \Phi : m(e(x)(\varphi))(\varphi) &= m(e'(x)(\varphi))(\varphi) \\ \Leftrightarrow \forall x \in X, \varphi \in \Phi : e(x)(\varphi) &= e'(x)(\varphi) \Leftrightarrow e = e'. \end{aligned}$$

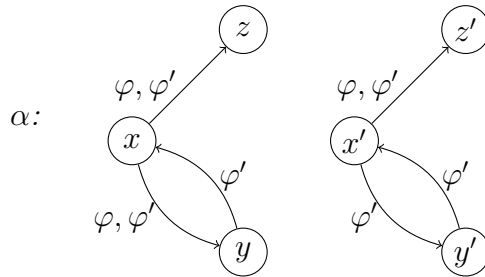
□

Thus, we can conclude that the algorithm from [ABH⁺12] is applicable, using the pseudo-factorisation structure we have derived. We now discuss a small example for the application of the minimisation algorithm from [ABH⁺12] using this pseudo-factorisation structure on $\mathbf{Kl}(_^\Phi)$ for the functor F .

Example 7.6.8 *Let $X = \{x, y, z, x', y', z'\}$, $|A| = 1$ and $\Phi = \{\varphi', \varphi\}$, with $\varphi' \leq_\Phi \varphi$. Let $X = \{x, y, z, x', y', z'\}$, $|A| = 1$ and $\Phi = \{\varphi', \varphi\}$, with $\varphi' \leq_\Phi \varphi$. We define the CTS α coalgebraically as follows:*

$$\begin{aligned} \alpha(x)(\varphi) &= \{(y, \varphi), (z, \varphi), (y, \varphi'), (z, \varphi')\} & \alpha(x)(\varphi') &= \{(y, \varphi'), (z, \varphi')\} \\ \alpha(y)(\bar{\varphi}) &= \{(x, \varphi')\} \text{ for } \bar{\varphi} \in \{\varphi, \varphi'\} & \alpha(y')(\bar{\varphi}) &= \{(x', \varphi')\} \text{ for } \bar{\varphi} \in \{\varphi, \varphi'\} \\ \alpha(x')(\varphi) &= \{(z', \varphi), (y', \varphi'), (z', \varphi')\} & \alpha(x')(\varphi') &= \{(y', \varphi'), (z', \varphi')\} \\ \alpha(\bar{z})(\bar{\varphi}) &= \emptyset \text{ for } \bar{\varphi} \in \{\varphi, \varphi'\}, \bar{z} \in \{z, z'\}. \end{aligned}$$

Graphically, the CTS can be depicted as follows:



To compute the behavioural equivalence, we start by taking the unique morphism $d_0: X \rightarrow 1$ into the final object of $\mathbf{Kl}(_^\Phi)$ that is $1 = \{\bullet\}$. At the i^{th} iteration, we obtain e_i via the pseudo-factorisation of $d_i = m_i \circ e_i$ and then we build $d_{i+1} = F e_i \circ \alpha$. These iterations are shown in the following tables. Note

that each table represents both, d_i and e_i , because the pseudo-factorisation just yields simple injections as monomorphisms, so d_i and e_i in each step only differ by their codomain.

d_0, e_0	x	y	z	x'	y'	z'
φ	•	•	•	•	•	•
φ'	•	•	•	•	•	•

d_1, e_1	x	y	z	x'	y'	z'
φ	$\{(\bullet, \varphi), (\bullet, \varphi')\}$	$\{(\bullet, \varphi')\}$	\emptyset	$\{(\bullet, \varphi), (\bullet, \varphi')\}$	$\{(\bullet, \varphi')\}$	\emptyset
φ'	$\{(\bullet, \varphi')\}$	$\{(\bullet, \varphi')\}$	\emptyset	$\{(\bullet, \varphi')\}$	$\{(\bullet, \varphi')\}$	\emptyset

d_2, e_2	x	y	z	x'	y'	z'
φ	blue	green	orange	black	green	orange
φ'	red	green	orange	red	green	orange

d_3, e_3	x	y	z	x'	y'	z'
φ	blue	green	orange	black	green	orange
φ'	red	green	orange	red	green	orange

In the tables for d_2/e_2 and d_3/e_3 we have used colours to code the entries, because the full notation for the entries would be too large to fit in the tables.

The codomains C_0, C_1, C_2 , and C_3 of e_0, e_1, e_2 , and e_3 (resp.) are given below (note that the colours in C_2 and C_3 indicate the colours in the tables above):

$$C_0 = \{\bullet\}$$

$$C_1 = \{\emptyset, \{(\bullet, \varphi')\}, \{(\bullet, \varphi), (\bullet, \varphi')\}\}$$

$$C_2 = \{\{(\emptyset, \varphi), (\emptyset, \varphi'), (\{(\bullet, \varphi')\}, \varphi')\}, \{\{(\emptyset, \varphi'), (\{(\bullet, \varphi')\}, \varphi')\}\}, \{\{(\{(\bullet, \varphi')\}, \varphi')\}, \emptyset, \{(\{(\bullet, \varphi')\}, \varphi), (\{(\bullet, \varphi')\}, \varphi'), (\emptyset, \varphi), (\emptyset, \varphi')\}\}$$

$$C_3 = \{\{(\{(\emptyset, \varphi), (\{(\{(\bullet, \varphi')\}, \varphi')\}, \varphi'), (\emptyset, \varphi')\}\}, \{\{(\emptyset, \varphi'), (\{(\{(\bullet, \varphi')\}, \varphi')\}, \varphi')\}\}, \{\{(\{(\{(\bullet, \varphi')\}, \varphi'), (\emptyset, \varphi')\}, \varphi')\}, \emptyset, \{(\{(\{(\bullet, \varphi')\}, \varphi')\}, \varphi), (\{(\{(\bullet, \varphi')\}, \varphi')\}, \varphi'), (\emptyset, \varphi), (\emptyset, \varphi')\}\}$$

each ordered by inclusion. By contrast, the codomain C'_i of d_i is defined as $C'_0 = C_0$, $C'_i = \mathcal{P}(C_i \times \Phi)$ for $i = 1, 2, 3$.

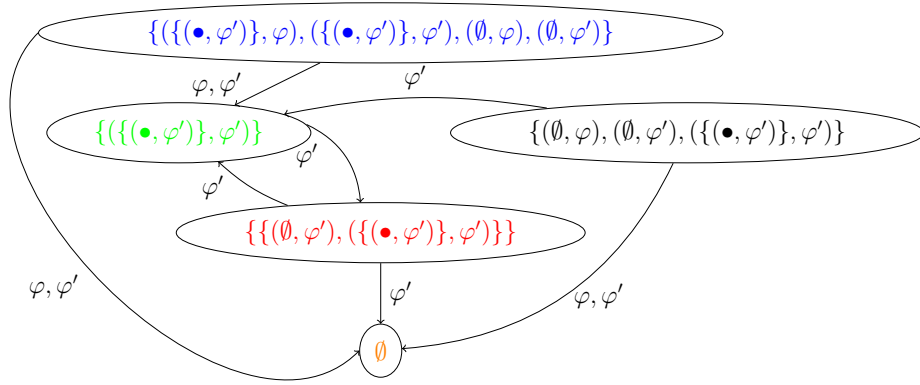
By comparing the columns for each state we can determine which states are bisimilar. The partitions are divided as follows (where X_i denote the entries at the i^{th} iteration):

$$\begin{aligned} X_0 &= \{\{x, y, z, x', y', z'\}\} & X_1 &= \{\{x, x'\}, \{y, y'\}, \{z, z'\}\} \\ X_2 &= X_3 = \{\{x\}, \{x'\}, \{y, y'\}, \{z, z'\}\}. \end{aligned}$$

To obtain the greatest conditional bisimulation from e_2 (or e_3), we need to compare individual entries of each table. We can identify the greatest bisimulation as $\{R_\varphi, R_{\varphi'}\}$, where (written as equivalence classes)

$$R_\varphi = \{\{z, z'\}, \{x\}, \{x'\}, \{y, y'\}\} \quad R_{\varphi'} = \{\{x, x'\}, \{y, y'\}, \{z, z'\}\}$$

Additionally, it is possible to derive the minimal coalgebra that was identified using the minimisation algorithm, which is of the form $(E_2, m_3 \circ \iota)$ where $\iota: E_2 \rightarrow F(E_2)$ is the arrow witnessing termination of the algorithm. The minimisation has the following form:



Note that, if there was no order on Φ , x and x' would be found equivalent under φ , because without upgrading, x and x' behave the same for φ : Both can do exactly one step, reaching either of y, z or z' , respectively, but in none of these states additional steps are possible in the product φ .

One can observe that both x and x' get mapped under φ' to the red state (second from bottom of the diagram), but under φ , the state x gets mapped to the blue state (top state in the diagram), whereas x' gets mapped to the black state (right-most state in the diagram).

Since we have seen that the Kleisli categories for the lattice monad and the reader monad, $\mathbf{Kl}(T)$ and $\mathbf{Kl}(_\Phi)$, are isomorphic, we want to characterise factorisation in $\mathbf{Kl}(T)$. We can factorise an arrow by converting it to a $\mathbf{Kl}(_\Phi)$ -arrow and factorising that arrow, then translating back to $\mathbf{Kl}(T)$. Since we have already seen that factorising in $\mathbf{Kl}(_\Phi)$ basically means to exclude all states from the codomain of the arrow that are not in the image of any pair of state

and alphabet symbol, this boils down to finding out when a state in a $\mathbf{Kl}(T)$ -arrow will be identified as redundant in $\mathbf{Kl}(_^\Phi)$. So let $f: (X, \leq_X) \rightarrow (Y, \leq_Y)$ be a $\mathbf{Kl}(T)$ -arrow, then $f(x)$ is a function $b: (Y \rightarrow \mathbb{L})^*$. An element $y \in Y$ will occur in the image of $f(x)$ in case there is an irreducible element $\varphi \in \mathcal{J}(\mathbb{L}) = \Phi$ such that y is the smallest element of Y such that $b(y) \geq \varphi$. This is true if $\sqcup\{b(y') \mid y' < y\} \neq b(y)$. So by factorising an arrow in $\mathbf{Kl}(T)$ we eliminate all states such that $\sqcup\{f(x)(y') \mid y' < y\} = b(y)$ for all $x \in X$.

Comparison of the Final Chain Algorithm and the Matrix Multiplication Algorithm

In Chapter 6, where we did not consider CTS and LaTS from a coalgebraic perspective, we also presented a fixed point algorithm to compute the greatest lattice bisimulation for an LaTS. The procedure works as follows (Algorithm 6.4.3):

Algorithm 7.6.9 *Let (X, A, \mathbb{L}, f) be a finite LaTSs over the finite distributive lattice $\mathbb{L} = \mathcal{O}(\Phi, \leq_\Phi)$, i.e., the sets X , and A are finite. We define a series of lattice-valued relations $R_i: X \times X \rightarrow \mathbb{L}$.*

Fix R_0 as $R_0(x, y) = 1$ for all $x, y \in X$. Then, compute $R_{i+1} = F_1(R_i) \sqcap F_2(R_i)$ for all $i \in \mathbb{N}_0$ until $R_n \sqsubseteq R_{n+1}$ for an $n \in \mathbb{N}_0$. F_1 and F_2 are defined according to:

$$F_1(R)(x, y) = \prod_{a \in A, x' \in X} \left(f(x, a, x') \rightarrow \left(\bigsqcup_{y' \in X} (f(y, a, y') \sqcap R(x', y')) \right) \right),$$

$$F_2(R)(x, y) = \prod_{a \in A, y' \in X} \left(f(y, a, y') \rightarrow \left(\bigsqcup_{x' \in X} (f(x, a, x') \sqcap R(x', y')) \right) \right).$$

Lastly, return R_n as the greatest bisimulation.

It can be shown that for two states x, y $x \sim_\varphi y$ iff $\varphi \in R_n(x, y)$.

As explained before, CTS and LaTS are dual models and can be translated directly into one another. Using the functor F combined with the lattice monad, we have shown how to define LaTS coalgebraically. The representation of an LaTS needs to be derived from the definition of CTS, passed through the isomorphism between the Kleisli categories for the reader monad and the lattice

monad: For any state x , action a and set of pairs $Y \subseteq (X \times \Phi)^A$ we obtain the coalgebra

$$f(x)(Y)(a) = \{\psi \in \Phi \mid \forall \varphi' \leq \psi, x' \in X : x \xrightarrow{a, \varphi'} x' \Rightarrow (x', \varphi') \subseteq Y(a)\}.$$

However, the final chain algorithm does not exactly replicate the fixed-point algorithm described above. Instead of $X \times X$ matrices over \mathbb{L} , the final chain algorithm yields – before factorisation – $X \times F^i 1$ matrices over \mathbb{L} , when translated via α^{-1} . Factorisation may shrink the second dimension, but it usually is not equal to X . We will now show that there is still a strong correspondence between the final chain algorithm and the fixed point computation: Both algorithms separate pairs of states at the same point in the computation and both algorithms terminate at the same time.

Lemma 7.6.10 *Let a CTS $\alpha : X \rightarrow FX$ over a finite set of states X and a finite ordered set of conditions Φ be given. Moreover, consider the dual LaTS $f : X \times X \rightarrow \mathcal{O}(\Phi)$. Then Algorithm 7.6.9 and Algorithm 3.3.8 terminate after the same number of iterations.*

Proof: Due to the special nature of the factorisation in Algorithm 3.3.8, we will always argue using d_k instead of e_k . We will first show that Algorithm 3.3.8 separates two states in iteration k if and only if Algorithm 7.6.9 does, as well. For that purpose we define the matrices

$$M_k(x, y) = \{\varphi \mid d_k(x)(\varphi) = d_k(y)(\varphi)\}.$$

Note, that per definition of F , this set is always downwards-closed for $k \geq 1$, and due to the d_0 codomain being the final object, it is also downwards-closed for $k = 0$. We can now prove that $R_k = M^k$ for all k , proving the first claim, that two states can only get separated in both algorithms at the same time. We prove this via induction.

- Let $k = 0$. Per definition, $R_0(x, y) = \Phi$ for all $x, y \in X$ and due to d_0 being the unique arrow into the final object, it must also hold that $M_0(x, y) = \Phi$ for all $x, y \in X$.
- Assume we have shown the claim for all $k' \leq k$.

- We will now show that the claim also holds for $k + 1$. For this we show mutual inclusion.

– Let $\varphi \in R_{k+1}(x, y)$ for any $\varphi \in \Phi$, $x, y \in X$. Then $R_k(x, y) \ni \varphi$ and – by definition of F_1 – for all $a \in A$, $\varphi' \leq \varphi$, $x' \in \alpha(x)(a)(\varphi')$ exists $y' \in \alpha(y)(a)(\varphi')$ such that $R(x', y') \ni \varphi'$ and – by definition of F_2 – vice-versa. The induction allows us to find $M^k(x, y) \ni \varphi$ and for all $a \in A$, $\varphi' \leq \varphi$, $x' \in \alpha(x)(a)(\varphi')$ there exists a $y' \in \alpha(y)(a)(\varphi')$ such that $d_k(x')(\varphi') = d_k(y')(\varphi')$ (and vice-versa). Therefore, we can compute:

$$\begin{aligned} d_{k+1}(x)(\varphi)(a) &= Fd_k \circ \alpha(x)(\varphi)(a) \\ &= \{(d_k(x')(\varphi'), \varphi') \mid \varphi' \leq \varphi, x' \in \alpha(x)(\varphi')(a)\} \\ &= \{(d_k(y')(\varphi'), \varphi') \mid \varphi' \leq \varphi, y' \in \alpha(y)(\varphi')(a)\} \\ &= Fd_k \circ \alpha(y)(\varphi)(a) = d_{k+1}(y)(\varphi)(a). \end{aligned}$$

Thus, $M^{k+1}(x, y) \ni \varphi$.

- Let $M^{k+1}(x, y) \ni \varphi$, i.e. $d_{k+1}(x)(\varphi')(a) = d_{k+1}(y)(\varphi')(a)$ for all $a \in A$, $\varphi' \leq \varphi$. Thus, by definition of d_{k+1} ,

$$\begin{aligned} &\{(d_k(x')(\varphi'), \varphi') \mid \varphi' \leq \varphi, x' \in \alpha(x)(\varphi')(a)\} \\ &= \{(d_k(y')(\varphi'), \varphi') \mid \varphi' \leq \varphi, y' \in \alpha(y)(\varphi')(a)\}. \end{aligned}$$

Therefore, for all $\varphi' \leq \varphi$, $a \in A$, $x' \in \alpha(x)(\varphi')(a)$, there must exist a $y' \in \alpha(y)(\varphi')(a)$ such that $d_k(x')(\varphi') = d_k(y')(\varphi')$ and vice-versa. Using the induction hypothesis we can therefore find, that $\varphi' \leq \varphi$, $a \in A$, $x' \in \alpha(x)(\varphi')(a)$, there must exist a $y' \in \alpha(y)(\varphi')(a)$ such that $\varphi' \in R_k(x', y')$, proving $\varphi \in F_1(R_k)(x, y)$ and vice-versa, proving $\varphi \in F_2(R_k)(x, y)$. Thus, $\varphi \in R_{k+1}(x, y)$.

So we have seen that both algorithms separate pairs of states in the same iteration. However, it still has to be shown that when no pairs of states are separated anymore, Algorithm 7.6.9 terminates. Per definition, Algorithm 3.3.8 terminates at that point, as well, so Algorithm 3.3.8 can mimic Algorithm 7.6.9. So, let d_n be such that $M_{n-1} = M_n$, and thus $R_{n-1} = R_n$, i.e. Algorithm 7.6.9 has terminated. Further, let x, y be given, such that $d_n(x)(\varphi) \leq d_n(y)(\varphi)$, i.e.

for all $a \in A$, $d_n(x)(\varphi)(a) \subseteq d_n(y)(\varphi)(a)$. Then

$$d_{n+1}(x)(\varphi)(a) = Fd_n \circ \alpha(x)(\varphi)(a) = \{(d_n(x')(\varphi'), \varphi') \mid (x', \varphi') \in \alpha(x)(\varphi) \wedge \varphi' \leq \varphi\}$$

and analogously

$$d_{n+1}(y)(\varphi)(a) = \{(d_n(y')(\varphi'), \varphi') \mid (y', \varphi') \in \alpha(y)(\varphi) \wedge \varphi' \leq \varphi\}.$$

We know that $d_n(x)(\varphi)(a) \subseteq d_n(y)(\varphi)(a)$, therefore, for $(x', \varphi') \in \alpha(x)(\varphi)$ there must exist a $(y', \varphi') \in \alpha(y)(\varphi)$ such that $d_{n-1}(x')(\varphi') = d_{n-1}(y')(\varphi')$. Since $M_{n-1} = M_n$, then it must also hold that $d_n(x')(\varphi') = d_n(y')(\varphi')$. Thus we can conclude $d_{n+1}(y)(\varphi)(a) \supseteq d_{n+1}(x)(\varphi)(a)$. Thus, Algorithm 3.3.8 terminates in step n as well. \square

7.7 Conclusion, Related Work and Future Work

We have seen that the Kleisli categories for the lattice monad and the reader monad are equivalent, providing an analogue for the Birkhoff duality between lattices and partially ordered sets. This duality also reflects the duality between conditional transition systems (CTS) and lattice transition systems (LaTS). We have investigated two different functors, \mathcal{P} and F , which can be used to model CTS without upgrades and general CTS, respectively, in such a way that behavioural equivalence is conditional bisimulation. The finite powerset functor \mathcal{P} does not lift in case the products are not discretely ordered, however, it is not surprising that this functor cannot be used to model CTS with upgrades. When considering upgrades, the individual products cannot be considered purely a side effect and can instead be observed, which leads to the requirement of making the products explicit in a way.

The Kleisli category for the reader monad has a pseudo-factorisation structure that makes it possible to use Algorithm C (Algorithm 3.3.8) to compute the greatest conditional bisimulation using a final chain-based algorithm for the functors \mathcal{P} and F .

Our work obviously stands in the tradition of the work in [ABH⁺12] and [KK14]. In a broader sense, the modelling technique of using Kleisli categories to obtain the 'right' notion of behavioural equivalence goes back to previous work in [HJS07, PT99], where non-deterministic branching of NFA was masked

by the use of a Kleisli category (over **Set** in this case) to obtain language equivalence as behavioural equivalence rather than bisimulation.

Modelling new types of systems and behaviours coalgebraically is an ongoing field of research, as evident by recent work for instance by Bonchi et al. on decorated traces [BBC⁺16], Hermanns et al. on probabilistic bisimulation [HKK14] or Latella et al. on labelled state-to-function transition systems [LMdV15].

We are currently working towards obtaining some of the more technical results in this chapter through universal constructions, instead of direct definitions to obtain more concise proofs. Additionally, we are interested in extending our duality result for the lattice and the reader monad appropriately. In particular, we are interested in extending the Birkhoff duality to capture precisely those – not necessarily finite – distributive lattices that are isomorphic to the downward closed sets of irreducible elements. It is known that this does not hold in general for all distributive lattices, though a more involved duality can still be found in [DP02] that covers all distributive lattices.

Chapter 8

Implementation, Future Work and Conclusion

8.1 PAWS: A Tool for the Analysis of Weighted Systems

As noted in Chapter 1, as well as the main chapters, throughout the thesis, all algorithms that were developed have also been implemented in *C#*. These implementations formed the basis for the tool PAWS, which has been developed together with my colleague Christina Mika.

A key feature of PAWS is its extensibility. The algorithms are parametrised over the semiring and it is therefore possible to use the algorithms PAWS offers, not only for the semirings that come pre-implemented, but also for newly generated semirings. For that purpose, PAWS offers ways of adding new semirings and executing algorithms for these semirings. All algorithms are implemented generically and can be used for various semirings or *l*-monoids, provided all necessary operations such as addition, multiplication, and solving linear equations are specified. Therefore, PAWS is equipped with a semiring generator that allows to generate new semirings that are not pre-implemented and to define weighted automata or conditional transition systems over these. Specifically, we have built several layers of automatisation, so that whenever one, e.g., defines a complete distributive lattice, it suffices to give a partial order, from which the lattice of downward-closed sets is generated [DP02] and all

operations are provided automatically. Building semirings from other semirings using cross products is almost completely automatised and modulo rings are automatised using Hensel liftings [DM09]. In addition, it is possible to add arbitrary semirings by providing code for the operations mentioned before. The semiring generator, as well as the implementation of the Hensel lifting is the main content of Christina Mika's Master's thesis [Mik15].

The problems PAWS solves and the corresponding algorithms and semirings are displayed in the following table:

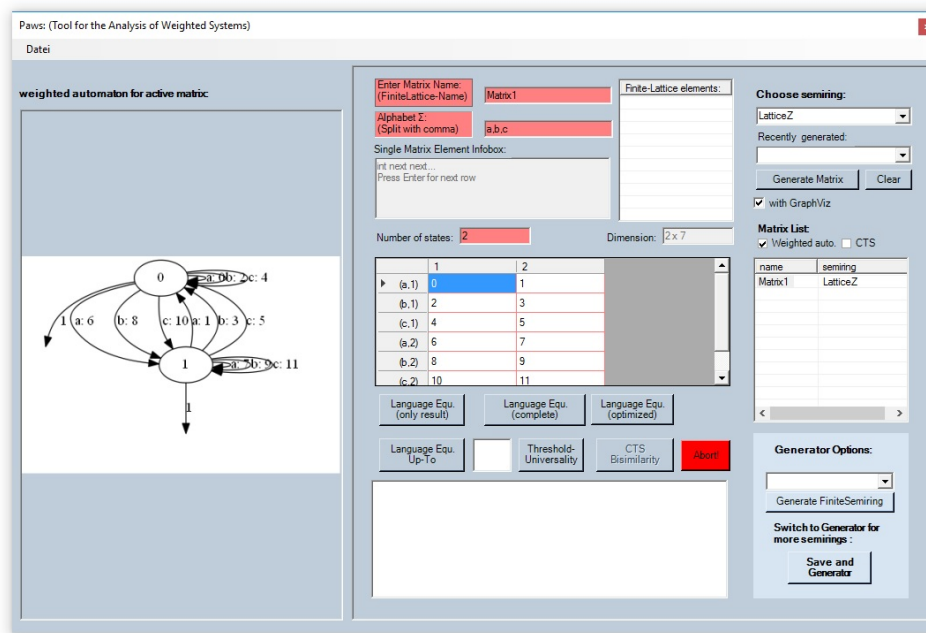
Problem	Algorithm	Semiring	Model
Language equivalence (all pairs)	<i>Language Equ.(Complete)</i>	any semiring	WA
Language equivalence (initial vectors)	<i>Language Equ.(Up-To)</i>	<i>l</i> -monoids, lattices	WA
Universality Problem	<i>Universality</i>	tropical semiring	WA
Conditional bisimilarity	<i>CTS Bisimilarity</i>	finite lattice	CTS

Where *Language Equ.(Complete)* is an implementation of Algorithm B (Algorithm 3.3.6, Algorithm 4.3.1), *Language Equ.(Up-To)* is an implementation of HKC (Algorithm 5.3.3), *Universality* is an implementation of HKP_A (Algorithm 5.3.14) and *CTS Bisimilarity* is the matrix multiplication algorithm for CTS (Algorithm 6.4.3).

PAWS offers two components:

- The *semiring generator* to build and provide the required semirings over which automata can be defined. This generator is used to generate semirings that cannot be obtained in a fully automated way and supports some automatic generations.
- The *analysis tool* that allows the user to choose a previously generated semiring, one of the semirings that come built-in with the PAWS or to build a finite lattice by inputting the dual partial order and then to define automata in a matrix representation over those semirings. Matrices are then interpreted as weighted automata or conditional transition systems (CTS) and can be used to compute language equivalence for weighted automata with two different approaches, decide the threshold problem for weighted automata over the tropical semiring of natural numbers or to compute the greatest bisimilarity of a CTS.

The analysis component is designed to offer generic algorithms applicable to numerous predefined or user-defined semirings. However, some of the algorithms can only be used with specific (types) of semirings. The most general algorithm is the language equivalence check, for which all semirings are eligible. Conditional transition systems are only defined over lattices, therefore, the bisimulation check is limited to lattice structures. However, the user still has the choice between two different ways of dealing with lattices: representing elements of the lattice as downward-closed sets of irreducibles via the Birkhoff duality [DP02], applicable to all finite distributive lattices, or representing them using binary decision diagrams (BDDs). The BDD variant is more restrictive and mainly designed for the application to CTS. Here, the irreducibles are required to be full conjunctions of features from a base set of features, ordered by the presence of distinct upgrade features. Lastly, the threshold check can only be performed over a single semiring, the tropical semiring over natural numbers.



The general workflow of the analysis tool is as follows:

- ▷ Choose a semiring
- ▷ Generate a matrix over this semiring, representing a weighted automaton or a conditional transition system

Alternatively: Choose the matrix from a list of matrices that have been generated previously

- ▷ Start the algorithm and provide – if necessary – additional input

Additional input comes in two forms: starting vectors and the threshold to be checked against in case of the threshold algorithm. Depending on the semiring of choice, questions regarding language equivalence might not be decidable, leading to non-termination of the corresponding procedure in PAWS. In order to deal with this problem and to allow abortion of an overlong computation, the actual computation is delegated to a separate thread that can be aborted by clicking a red button labelled “Abort” at any time. In that case all intermediate results are discarded.

Note, that only the two language equivalence-based algorithms can run into non-termination issues. For the CTS bisimulation check, as well as the threshold problem on the tropical semiring of natural numbers, termination is always guaranteed. However, the runtime of CTS bisimulation check can be doubly exponential in the number of features under consideration – because the lattice is the set of all possible configurations, which in turn are all possible conjunctions over the features. Using the BDD-based implementation of lattices – which is particularly suited to the needs of CTS modelling – this explosion is mitigated in many cases, but it can not be ruled out completely. On the other hand, the BDD-based modelling only allows for special lattices to be modelled, i.e. those that arise as the lattices constructed from a set of features and upgrade features, whereas the variant called *FiniteLattice* allows for arbitrary (finite, distributive) lattices to be represented. In this case, lattices are represented via the partial order of irreducible elements, using Birkhoff’s representation theorem [DP02].

In [KKM17] we have also given an overview over runtime results for the various algorithms that are implemented in PAWS. The most interesting result is, that using HKC for all pairs of states in a randomly generated automaton, one after another, can be expected to perform better than computing Algorithm B once, when considering the lattice of all integers with the standard order on integers.

8.2 Future Work

For each individual chapter, we have already detailed related work and future work. Thus, we will only note one more point of future research that is outside each individual topic. We have seen that weighted systems can be used to analyse quantitative aspects of system models. This can be used to express the likelihood or cost associated with a transition. In many cases, a system model is required to take these aspects into account, rather than just the labels of actions that can be performed.

Additionally, conditional transition systems offer means to model software product lines, where not a single piece of software is modelled, but a family, instead, that has a common base structure but deviating concrete feature sets. Using partial orders on the products, we can also capture a notion of upgrading between different products.

A natural question that arises here is, why would one only be interested in modelling a single system that takes e.g. performance into account? Or alternatively: Why would one be content with answering qualitative questions where software product lines are concerned? It is therefore a natural question whether there are means to combine the techniques developed for weighted automata and conditional transition systems, to answer questions about conditional weighted automata, i.e. families of weighted automata with a common base structure. Considering weighted automata are usually analysed for language equivalence rather than bisimulation, consequently, the problem we would like to consider is language equivalence in the following sense: Identify those products under which a given pair of states is language equivalent.

A focus on specific structures such as fields, where language equivalence is decidable and reasonably efficiently so, seems prudent, but, since the probabilistic case can be embedded into a field, one major field of interest in weighted automata can be covered this way. One general idea to approach this problem is to generalise the formal power series approach for weighted automata to take products into account. So one could write the weights of a weighted automaton as formal sums over the products and compute e.g. (a variant of) the algorithm up to congruence **HKC**. In each step of **HKC** one could prune the intermediate results by taking the upgrading structure into account: If for a condition φ a

word separating the initial vectors is found, one can conclude that the pair of vectors is also not language equivalent for all systems that can upgrade to φ . However, open questions in this regard include:

- Is a monotonicity requirement sensible as for CTS?
- Can techniques be developed that can yield a significant gain in performance by exploiting common structures among different products? This question is, of course, also tied to the matter of monotonicity requirements.

While this thesis is squarely focussed on behavioural equivalences, a natural extension of this is to discuss how *similar* pairs of states in a given state-based system are. Some coalgebraic groundwork for this has already been done by Henning Kerstan in his doctoral thesis [Ker16], where lifting on metrics was discussed, which could prove in particular fruitful for weighted automata. For CTS, in contrast, previous work, e.g. by Fahrenberg et al. [AFL15], on software product lines suggests, that contrasting the conditions under which two states are equivalent is a relevant metric for comparing states.

8.3 Conclusion

The overarching theme of this thesis was to extend known techniques for traditional state-based systems to ones that are enriched with weights or conditions. For this purpose, two main approaches were taken. On the one hand, coalgebraic techniques were adapted and refined, in order to identify commonalities between different systems and different notions of behaviour. On the other hand, we worked on improving our algorithmic results by employing various optimisations. The two approaches complement each other nicely. A coalgebraic analysis of behaviour allows to identify common structures in various kinds of system models and to develop core algorithms, that can be instantiated to many suitable classes of systems. A purely coalgebraic view abstracts from the specifics of a given class of systems though, which is handy when one is interested in core concepts, but may not be ideal when it comes to efficient decision procedures. Therefore, a more concrete perspective on specific system types is required to obtain well-performing algorithms for complex systems by

exploiting e.g. up-to techniques. Coalgebraic reasoning can still help to establish the ideas that form the basis for more specialised algorithmic approaches.

In consequence, on the coalgebraic side, we have developed a more generally applicable final chain algorithm – Algorithm A in Chapter 3 – that can for instance be used for weighted automata, conditional transition systems or HD-automata, but of course can also be applied to traditional systems such as (non-)deterministic automata or labelled transition systems. A collection of generic optimisation strategies for this approach have been proposed and evaluated. Notably, we have shown that we can rediscover techniques developed for various specific kinds of weighted automata, be it Schützenberger’s algorithm for weighted automata over a field, or conjugacy-based techniques (Chapter 4). Using a category that is strongly related to a Kleisli category to hide side effects, we managed to model weighted automata coalgebraically in such a way, that behavioural equivalence coincides with language equivalence, rather than a notion of bisimulation. For conditional transition systems (Chapter 7), we could capture the duality that lies at the core of the systems, in our coalgebraic model and retrieve an algorithm that is similar, yet not identical to the concrete approach we have developed. The proposed model for CTS is interesting in the sense that the conditions play a dual role of being involved in a side effect, hidden inside the Kleisli monad, and observable behaviour, as well. This stems from the fact that updating must be regarded as an observable action.

Beyond the coalgebraic view, we have developed up-to techniques for weighted automata that are designed in a modular way (Chapter 5), so that a procedure to decide congruence closure may be plugged in to the base algorithms to obtain an optimised algorithm to decide language equivalence for a new class of weighted automata. We specifically focussed on l -monoids, which are lattices with an additional monoid structure that distributes over suprema, to work out the algorithm in full for a broad class of semirings. Consequently, a concrete rewriting procedure for l -monoids has been developed and we demonstrated a potential exponential gain when compared to known techniques in various parameters. CTS, on the other hand, have been extended to take a notion of upgrades into account (Chapter 6). A fixpoint algorithm to decide (conditional) bisimulation, that is very reminiscent of the case of labelled transition systems has been developed and new BDD-based techniques for (special kinds of) lattices

rather than Boolean algebras have been proposed and shown to allow for an exponential speed-up when compared to a representation based on Birkhoff's duality.

Anhang A

Additional Proofs (Chapter 3)

The appendix consists of additional proofs that were omitted in the main text for one or more of the following reasons:

- The proof is rather technical and the technicalities involved do not contribute to the understanding of the main content of the respective chapter.
- The result is well-known, but we were not aware of a publication, where a corresponding proof can be found and referred to. For the sake of completion, the proof is given in the appendix.
- The result can be proven using previous work, but the previous result cannot be stated using the mathematical machinery of this thesis.

We start by showing that the semiring monad (Definition 3.4.11) is a monad. As stated in the main text, when the monad was defined, this is a standard result.

Lemma A.1 *(S, η, μ) , defined in Definition 3.4.11, is a monad.*

Proof:

- S preserves identities: Let $\text{id} : X \rightarrow X$ be the identity function on X , $a \in SX$ and $y \in X$, then

$$S\text{id}(a)(y) = \sum \{a(x) \mid x \in X, \text{id}(x) = y\} = \sum \{a(y)\} = a(y)$$

- S respects concatenation: Let $f : Y \rightarrow Z$, $g : X \rightarrow Y$, $a \in \mathbb{S}^X$ and $z \in Z$, then

$$\begin{aligned}
 Sf \circ g &= \sum \{a(x) \mid x \in X, f(g(x)) = z\} \\
 Sf \circ Sg(a)(z) &= \sum \{Sg(a)(y) \mid y \in Y, f(y) = z\} \\
 &= \sum \{\sum \{a(x) \mid x \in X, g(x) = y\} \mid y \in Y, f(y) = z\} \\
 &= \sum \{s(x) \mid x \in X, \exists y \in Y, f(y) = z \wedge g(x) = y\}
 \end{aligned}$$

We will conclude by showing that $x \in \{x \mid x \in X, f(g(x)) = z\} \Leftrightarrow x \in \{x \mid x \in X, \exists y \in Y, f(y) = z \wedge g(x) = y\}$:

If $x \in \{x \mid x \in X, f(g(x)) = z\}$, i.e. $f(g(x)) = z$, let $y = g(x)$, then $f(y) = z$, i.e. $x \in \{x \mid x \in X, \exists y \in Y, f(y) = z \wedge g(x) = y\}$.

If $x \in \{x \mid x \in X, \exists y \in Y, f(y) = z \wedge g(x) = y\}$ then there exists an $x \in X$ and a $y \in Y$, such that $g(x) = y$ and $f(y) = z$, i.e. $f(g(x)) = z$, therefore $x \in \{x \mid x \in X, f(g(x)) = z\}$.

- η is a natural transformation: Let $f : X \rightarrow Y$, then

$$\begin{aligned}
 (\eta_Y \circ f)(x)(y) &= \eta_Y(f(x))(y) = \begin{cases} 1 & \text{if } f(x) = y \\ 0 & \text{otherwise} \end{cases} \\
 Sf \circ \eta(x)(y) &= \sum \{\eta_X(x)(x') \mid x' \in X, f(x') = y\} \\
 &= \sum \{\eta_X(x)(x) \mid f(x) = y\} = \begin{cases} 1 & \text{if } f(x) = y \\ 0 & \text{otherwise} \end{cases}
 \end{aligned}$$

- μ is a natural transformation: Let $m \in \mathbb{S}^{\mathbb{S}^X}$, $f : X \rightarrow Y$ and $y \in Y$ be given, then

$$\begin{aligned}
 \mu_Y \circ (SSf(m))(y) &= \sum \{S(Sf(m))(g) \cdot g(y) \mid g \in \mathbb{S}^Y\} \\
 &= \sum \{\sum \{m(h) \mid h \in \mathbb{S}^X, Sf(h) = g\} \cdot g(y) \mid g \in \mathbb{S}^Y\} \\
 &= \sum \{m(h) \cdot Sf(h)(y) \mid h \in \mathbb{S}^X\} \\
 &= \sum \{m(h) \cdot \sum \{h(x) \mid x \in X, f(x) = y\} \mid h \in \mathbb{S}^X\} \\
 &= \sum \{m(h) \cdot h(x) \mid x \in X, f(x) = y, h \in \mathbb{S}^X\}
 \end{aligned}$$

and

$$\begin{aligned}
 Sf(\mu_X(m))(y) &= \sum \{\mu_X(m)(x) \mid x \in X, f(x) = y\} \\
 &= \sum \{\sum \{m(h) \cdot h(x) \mid h \in \mathbb{S}^X\} \mid x \in X, f(x) = y\} \\
 &= \sum \{m(h) \cdot h(x) \mid x \in X, f(x) = y, h \in \mathbb{S}^X\}
 \end{aligned}$$

- The unit law holds: Let $b : X \rightarrow \mathbb{S}^Y$ and $x \in X$. Then

$$\begin{aligned}\mu_X \circ \eta_{SX}(b)(x) &= \sum \{ \eta_{SX}(b)(f) \cdot f(x) \mid f \in \mathbb{S}^X \} \\ &= \sum \{ \eta_{SX}(b)(b) \cdot b(x) \} = 1 \cdot b(x) = b(x)\end{aligned}$$

as well as:

$$\begin{aligned}\mu_X \circ S(\eta_X)(b)(x) &= \sum \{ S\eta_X(b)(f) \cdot f(x) \mid f \in \mathbb{S}^X \} \\ &= \sum \{ \sum \{ b(y) \cdot f(x) \mid y \in X, \eta_X(y) = f \} \mid f \in \mathbb{S}^X \} \\ &= \sum \{ b(y) \cdot f(x) \mid \eta_X(y) = f, y \in X, f \in \mathbb{S}^X \} \\ &= \sum \{ b(y) \cdot \eta_X(y)(x) \mid y \in X \} = \sum \{ b(x) \cdot \eta_X(x)(x) \} = b(x)\end{aligned}$$

- The associative law holds: Let $x \in X$, $\hat{S} \in \mathbb{S}^{\mathbb{S}^X}$, then

$$\begin{aligned}\mu_X \circ S(\mu_X)(\hat{S})(x) &= \sum \{ S\mu_X(\hat{S})(f) \cdot f(x) \mid f \in \mathbb{S}^X \} \\ &= \sum \{ \sum \{ \hat{S}(g) \mid g \in \mathbb{S}^{\mathbb{S}^X}, \mu_X(g) = f \} \cdot f(x) \mid f \in \mathbb{S}^X \} \\ &= \sum \{ \sum \{ \hat{S}(g) \mid g \in \mathbb{S}^{\mathbb{S}^X}, \forall y \in X : \mu_X(g)(y) = f(y) \} \cdot f(x) \mid f \in \mathbb{S}^X \} \\ &= \sum \{ \hat{S}(g) \cdot \mu_X(g)(x) \mid g \in \mathbb{S}^{\mathbb{S}^X} \} \\ &= \sum \{ \hat{S}(g) \cdot \sum \{ g(f) \cdot f(x) \mid f \in \mathbb{S}^X \} \mid g \in \mathbb{S}^{\mathbb{S}^X} \} \\ &= \sum \{ \hat{S}(g) \cdot g(f) \cdot f(x) \mid f \in \mathbb{S}^X, g \in \mathbb{S}^{\mathbb{S}^X} \}\end{aligned}$$

as well as:

$$\begin{aligned}\mu_X \circ \mu_{SX}(\hat{S})(x) &= \sum \{ \mu_{SX}(\hat{S})(f) \cdot f(x) \mid f \in \mathbb{S}^X \} \\ &= \sum \{ \sum \{ \hat{S}(g) \cdot g(f) \mid g \in \mathbb{S}^{\mathbb{S}^X} \} \cdot f(x) \mid f \in \mathbb{S}^X \} \\ &= \sum \{ \hat{S}(g) \cdot g(f) \cdot f(x) \mid f \in \mathbb{S}^X, g \in \mathbb{S}^{\mathbb{S}^X} \}\end{aligned}$$

□

Anhang B

Additional Proofs (Chapter 5)

B.1 Proofs on the Embedding of Semirings

Here, we want to give two proofs which we have omitted in the main text for a lemma and a proposition we used to show under which circumstances a semiring can be embedded into a ring. As a reminder, we will restate the results before giving their respective proofs.

Lemma 5.2.3 *Let $(\mathbb{S}, +, \cdot, 0, 1)$ be a semiring, then $(\mathbb{S} \times \mathbb{S}, +, \cdot, (0, 0), (1, 0))$ where $(s_1, s_2) + (s'_1, s'_2) = (s_1 + s'_1, s_2 + s'_2)$ and $(s_1, s_2) \cdot (s'_1, s'_2) = (s_1 \cdot s'_1 + s_2 \cdot s'_2, s_1 \cdot s'_2 + s_2 \cdot s'_1)$ is a semiring.*

Proof: $(\mathbb{S} \times \mathbb{S}, +, (0, 0))$ is obviously a commutative monoid. The pair $(1, 0)$ really is the unit of multiplication, because given any $(s_1, s_2) \in \mathbb{S} \times \mathbb{S}$, it holds that $(s_1, s_2) \cdot (1, 0) = (s_1 \cdot 1 + s_2 \cdot 0, s_1 \cdot 0 + s_2 \cdot 1) = (s_1, s_2)$.

Now we need to show that \cdot is associative, let $(s_1, s_2), (s'_1, s'_2), (s''_1, s''_2) \in \mathbb{S} \times \mathbb{S}$ be given and compute:

$$\begin{aligned} & ((s_1, s_2) \cdot (s'_1, s'_2)) \cdot (s''_1, s''_2) = (s_1 \cdot s'_1 + s_2 \cdot s'_2, s_1 \cdot s'_2 + s_2 \cdot s'_1) \cdot (s''_1, s''_2) \\ &= ((s_1 \cdot s'_1 + s_2 \cdot s'_2) \cdot s''_1 + (s_1 \cdot s'_2 + s_2 \cdot s'_1) \cdot s''_2, \\ & \quad (s_1 \cdot s'_1 + s_2 \cdot s'_2) \cdot s''_2 + (s_1 \cdot s'_2 + s_2 \cdot s'_1) \cdot s''_1) \\ &= (s_1 \cdot s'_1 \cdot s''_1 + s_2 \cdot s'_2 \cdot s''_1 + s_1 \cdot s'_2 \cdot s''_2 + s_2 \cdot s'_1 \cdot s''_2, \\ & \quad s_1 \cdot s'_1 \cdot s''_2 + s_2 \cdot s'_2 \cdot s''_2 + s_1 \cdot s'_2 \cdot s''_1 + s_2 \cdot s'_1 \cdot s''_1) \\ &= (s_1 \cdot (s'_1 \cdot s''_1 + s'_2 \cdot s''_2) + s_2 \cdot (s'_1 \cdot s''_2 + s'_2 \cdot s''_1), \\ & \quad s_1 \cdot (s'_1 \cdot s''_2 + s'_2 \cdot s''_1) + s_2 \cdot (s'_1 \cdot s''_1 + s'_2 \cdot s''_2)) \\ &= (s_1, s_2) \cdot (s'_1 \cdot s''_1 + s'_2 \cdot s''_2, s'_1 \cdot s''_2 + s'_2 \cdot s''_1) = (s_1, s_2) \cdot ((s'_1, s'_2) \cdot (s''_1, s''_2)) \end{aligned}$$

Finally, we show that the distributive laws hold:

$$\begin{aligned}
& (s_1, s_2) \cdot ((s'_1, s'_2) + (s''_1, s''_2)) = (s_1, s_2) \cdot (s'_1 + s''_1, s'_2 + s''_2) \\
& = (s_1 \cdot (s'_1 + s''_1) + s_2 \cdot (s'_2 + s''_2), s_1 \cdot (s'_2 + s''_2) + s_2 \cdot (s'_1 + s''_1)) \\
& = (s_1 \cdot s'_1 + s_1 \cdot s''_1 + s_2 \cdot s'_2 + s_2 \cdot s''_2, s_1 \cdot s'_2 + s_1 \cdot s''_2 + s_2 \cdot s'_1 + s_2 \cdot s''_1) \\
& = (s_1 \cdot s'_1 + s_2 \cdot s'_2, s_1 \cdot s'_2 + s_2 \cdot s'_1) + (s_1 \cdot s''_1 + s_2 \cdot s''_2, s_1 \cdot s''_2 + s_2 \cdot s''_1) \\
& = (s_1, s_2) \cdot (s'_1, s'_2) + (s_1, s_2) \cdot (s''_1, s''_2) \\
\\
& ((s'_1, s'_2) + (s''_1, s''_2)) \cdot (s_1, s_2) = (s'_1 + s''_1, s'_2 + s''_2) \cdot (s_1, s_2) \\
& = ((s'_1 + s''_1) \cdot s_1 + (s'_2 + s''_2) \cdot s_2, (s'_1 + s''_1) \cdot s_2 + (s'_2 + s''_2) \cdot s_1) \\
& = (s'_1 \cdot s_1 + s''_1 \cdot s_1 + s'_2 \cdot s_2 + s''_2 \cdot s_2, s'_2 \cdot s_1 + s''_2 \cdot s_1 + s'_1 \cdot s_2 + s''_1 \cdot s_2) \\
& = (s'_1 \cdot s_1 + s'_2 \cdot s_2, s'_1 \cdot s_2 + s'_2 \cdot s_1) + (s''_1 \cdot s_1 + s''_2 \cdot s_2, s''_1 \cdot s_2 + s''_2 \cdot s_1) \\
& = (s'_1, s'_2) \cdot (s_1, s_2) + (s''_1, s''_2) \cdot (s_1, s_2)
\end{aligned}$$

□

Proposition 5.2.4 *Let $(\mathbb{S}, +, \cdot, 0, 1)$ be a semiring, then $(\mathbb{S} \times \mathbb{S}, +, \cdot, (0, 0), (1, 0))$ where $(s_1, s_2) + (s'_1, s'_2) = (s_1 + s'_1, s_2 + s'_2)$ and $(s_1, s_2) \cdot (s'_1, s'_2) = (s_1 \cdot s'_1 + s_2 \cdot s'_2, s_1 \cdot s'_2 + s_2 \cdot s'_1)$ is a semiring.*

Proof: Assume \mathbb{S} is a sub-semiring of a ring \mathbb{I} then, there must be an inverse element $-s_1 \in \mathbb{I}$, such that $s_1 + (-s_1) = 0$. Now, additionally assume $s_1 + s_2 = s_1 + s_3$, then, $s_3 = s_3 + s_1 + (-s_1) = (s_1 + s_3) + (-s_1) = (s_1 + s_2) + (-s_1) = s_2 + s_1 + (-s_1) = s_2$, so $s_1 + s_2 = s_1 + s_3$ implies $s_2 = s_3$.

For the other direction assume that $s_1 + s_2 = s_1 + s_3$ implies $s_2 = s_3$ for all $s_1, s_2, s_3 \in \mathbb{S}$.

We define a relation \equiv on $\mathbb{S} \times \mathbb{S}$ as follows:

$$\begin{aligned}
& (s_1, s_2) \equiv (s'_1, s'_2) \\
& \Leftrightarrow \exists s, s' \in \mathbb{S} : (s_1, s_2) + (s, s) = (s'_1, s'_2) + (s', s')
\end{aligned}$$

Note, that then for all $s \in \mathbb{S}$, $(s, s) \equiv (0, 0)$, which we will use in upcoming computations.

We will now prove that the relation \equiv is a congruence on the semiring $(\mathbb{S} \times \mathbb{S}, +, \cdot, (0, 0), (1, 0))$ from Lemma 5.2.3. First we need to prove that \equiv is an equivalence relation:

- \equiv is reflexive, because $(s_1, s_2) + (0, 0) = (s_1, s_2)$.

- \equiv is symmetric, because if $(s_1, s_2) \equiv (s'_1, s'_2)$ then there are $s, s' \in \mathbb{S}$ such that $(s, s) + (s_1, s_2) = (s'_1, s'_2) + (s', s')$, and since $=$ is symmetric, we also have $(s'_1, s'_2) \equiv (s_1, s_2)$.

- \equiv is transitive. Assume $(s_1, s_2) \equiv (s'_1, s'_2)$, i.e. there are $s, s' \in \mathbb{S}$ such that $(s_1, s_2) + (s, s) = (s'_1, s'_2) + (s', s')$, and $(s'_1, s'_2) \equiv (s''_1, s''_2)$, i.e. there are $\bar{s}, \bar{s}' \in \mathbb{S}$ such that $(s'_1, s'_2) + (\bar{s}, \bar{s}) = (s''_1, s''_2) + (\bar{s}', \bar{s}')$, then

$$\begin{aligned}
 (s_1, s_2) + (s + \bar{s}, s + \bar{s}) &= (s_1, s_2) + (s, s) + (\bar{s}, \bar{s}) \\
 &= (s'_1, s'_2) + (s', s') + (\bar{s}, \bar{s}) = (s'_1, s'_2) + (\bar{s}, \bar{s}) + (s', s') \\
 &= (s''_1, s''_2) + (\bar{s}', \bar{s}') + (s', s') = (s''_1, s''_2) + (s' + \bar{s}', s' + \bar{s}')
 \end{aligned}$$

Now we prove that \equiv is compatible with addition $(+)$: Assume $(s_1, s_2) \equiv (s'_1, s'_2)$, i.e. there are $s, s' \in \mathbb{S}$ such that $(s_1, s_2) + (s, s) = (s'_1, s'_2) + (s', s')$, and $(s''_1, s''_2) \equiv (s'''_1, s'''_2)$, i.e. there are $\bar{s}, \bar{s}' \in \mathbb{S}$ such that $(s''_1, s''_2) + (\bar{s}, \bar{s}) = (s'''_1, s'''_2) + (\bar{s}', \bar{s}')$. Then we can compute:

$$\begin{aligned}
 (s_1, s_2) + (s''_1, s''_2) &\equiv ((s_1, s_2) + (s''_1, s''_2)) + (s + \bar{s}, s + \bar{s}) \\
 &= (s_1 + s''_1, s_2 + s''_2) + (s + \bar{s}, s + \bar{s}) = (s_1 + s''_1 + s + \bar{s}, s_2 + s''_2 + s + \bar{s}) \\
 &= (s_1 + s + s''_1 + \bar{s}, s_2 + s + s''_2 + \bar{s}) = (s'_1 + s' + s''_1 + \bar{s}', s'_2 + s' + s''_2 + \bar{s}') \\
 &= (s'_1 + s''_1 + s' + \bar{s}', s'_2 + s''_2 + s' + \bar{s}') = (s'_1 + s''_1, s'_2 + s''_2) + (s' + \bar{s}', s' + \bar{s}') \\
 &= ((s'_1, s'_2) + (s''_1, s''_2)) + (s' + \bar{s}', s' + \bar{s}') \equiv (s'_1, s'_2) + (s''_1, s''_2)
 \end{aligned}$$

So $(s_1, s_2) + (s''_1, s''_2) \equiv (s'_1, s'_2) + (s''_1, s''_2)$. Finally, we also need to prove that \equiv is compatible with multiplication (\cdot) : $(s_1, s_2) \equiv (s'_1, s'_2)$, i.e. there are $s, s' \in \mathbb{S}$ such that $(s_1, s_2) + (s, s) = (s'_1, s'_2) + (s', s')$, and $(s''_1, s''_2) \equiv (s'''_1, s'''_2)$, i.e. there

are $\bar{s}, \bar{s}' \in \mathbb{S}$ such that $(s_1'', s_2'') + (\bar{s}, \bar{s}) = (s_1''', s_2''') + (\bar{s}', \bar{s}')$. Then we can compute:

$$\begin{aligned}
(s_1, s_2) \cdot (s_1'', s_2'') &= (s_2 \cdot s_2'' + s_1 \cdot s_1'', s_1 \cdot s_2'' + s_2 \cdot s_1'') \\
&\equiv (s_2 \cdot s_2'' + s_1 \cdot s_1'', s_1 \cdot s_2'' + s_2 \cdot s_1'') + (s \cdot s_2'', s \cdot s_2'') + (s \cdot s_1'', s \cdot s_1'') \\
&= (s_2 \cdot s_2'' + s \cdot s_2'' + s_1 \cdot s_1'' + s \cdot s_1'', s_1 \cdot s_2'' + s \cdot s_2'' + s_2 \cdot s_1'' + s \cdot s_1'') \\
&= ((s_2 + s) \cdot s_2'' + (s_1 + s) \cdot s_1'', (s_1 + s) \cdot s_2'' + (s_2 + s) \cdot s_1'') \\
&= ((s_2' + s') \cdot s_2'' + (s_1' + s') \cdot s_1'', (s_1' + s') \cdot s_2'' + (s_2' + s') \cdot s_1'') \\
&= (s_2' \cdot s_2'' + s' \cdot s_2'' + s_1' \cdot s_1'' + s' \cdot s_1'', s_1' \cdot s_2'' + s' \cdot s_2'' + s_2' \cdot s_1'' + s' \cdot s_1'') \\
&= (s_2' \cdot s_2'' + s_1' \cdot s_1'', s_1' \cdot s_2'' + s_2' \cdot s_1'') + (s' \cdot s_2'' + s' \cdot s_1'', s' \cdot s_2'' + s' \cdot s_1'') \\
&\equiv (s_2' \cdot s_2'' + s_1' \cdot s_1'', s_1' \cdot s_2'' + s_2' \cdot s_1'') \\
&\equiv (s_2' \cdot s_2'' + s_1' \cdot s_1'', s_1' \cdot s_2'' + s_2' \cdot s_1'') + (s_1' \cdot \bar{s}, s_1' \cdot \bar{s}) + (s_2' \cdot \bar{s}, s_2' \cdot \bar{s}) \\
&= (s_2' \cdot s_2'' + s_2' \cdot \bar{s} + s_1' \cdot s_1'' + s_1' \cdot \bar{s}, s_1' \cdot s_2'' + s_1' \cdot \bar{s} + s_2' \cdot s_1'' + s_2' \cdot \bar{s}) \\
&= (s_2' \cdot (s_2'' + \bar{s}) + s_1' \cdot (s_1'' + \bar{s}), s_1' \cdot (s_2'' + \bar{s}) + s_2' \cdot (s_1'' + \bar{s})) \\
&= (s_2' \cdot (s_2''' + \bar{s}') + s_1' \cdot (s_1''' + \bar{s}'), s_1' \cdot (s_2''' + \bar{s}') + s_2' \cdot (s_1''' + \bar{s}')) \\
&= (s_2' \cdot s_2''' + s_2' \cdot \bar{s}' + s_1' \cdot s_1''' + s_1' \cdot \bar{s}', s_1' \cdot s_2''' + s_1' \cdot \bar{s}' + s_2' \cdot s_1''' + s_2' \cdot \bar{s}') \\
&= (s_2' \cdot s_2''' + s_1' \cdot s_1''', s_1' \cdot s_2''' + s_2' \cdot s_1''') + (s_1' \cdot \bar{s}', s_1' \cdot \bar{s}') + (s_2' \cdot \bar{s}', s_2' \cdot \bar{s}') \\
&\equiv (s_2' \cdot s_2''' + s_1' \cdot s_1''', s_1' \cdot s_2''' + s_2' \cdot s_1''') = (s_1', s_2') \cdot (s_1''', s_2''')
\end{aligned}$$

Therefore, $\mathbb{I} := (\{[(s, s')]_{\equiv} \mid (s, s') \in S \times S\}, +, \cdot, [(0, 0)]_{\equiv}, [(1, 0)]_{\equiv})$ is a semiring. It is also a ring, because every $[(s, s')]_{\equiv}$ has an inverse $-[(s, s')]_{\equiv}$, namely $-[(s, s')]_{\equiv} := [(s', s)]_{\equiv}$, as seen via the following computation:

$$\begin{aligned}
[(s, s')]_{\equiv} + (-[(s, s')]_{\equiv}) &= [(s, s')]_{\equiv} + [(s', s)]_{\equiv} \\
&= [(s, s') + (s', s)]_{\equiv} = [(s + s', s' + s)]_{\equiv} = [(0, 0)]_{\equiv}
\end{aligned}$$

Up until now, we have not used injectivity of $+$. We only need this to finally prove that \mathbb{S} is a sub-semiring of \mathbb{I} . For this we need to show that for each $s, s' \in \mathbb{S}$, $(s, 0) \equiv (s', 0)$ implies $s = s'$. Assume, that $(s, 0) \equiv (s', 0)$, then there is $s'', s''' \in \mathbb{S}$ such that $(s + s'', s'') = (s''' + s', s')$, therefore $s' = s''$ and since $+$ is injective and $s + s'' = s''' + s'$, $s = s'''$. Furthermore, the set of elements of the form $(s, 0)$ is closed under addition and multiplication.

So, \mathbb{S} is indeed a sub-semiring of \mathbb{I} . □

B.2 Termination of HKP without Abstraction

When discussing termination of HKP, we made the following claim:

Lemma B.1 *Every set of n -dimensional vectors over the tropical semiring that contains no pair of (different) vectors v, v' such that $v \sqsubseteq v'$, i.e. $v \geq v'$, is finite.*

This can be shown using Dickson's Lemma, but to give a more direct proof in case the reader is unfamiliar with Dickson's Lemma, consider the following argument:

Proof: We can show this claim via induction over n .

If $n = 1$, then this is trivial, because every pair of elements in the tropical semiring is ordered. Now, assume we have shown the claim for all $m \leq n$, then we show that it also holds for $n + 1$ via contradiction. So assume there is an infinite set M of vectors of dimension $n + 1$, such that no pair of different vectors $v, v' \in M$ with $v \geq v'$ exists. Let $k := \min\{\tilde{v}[1] \mid \tilde{v} \in M\}$ and let one vector $\bar{v} \in M$ be given, such that $\bar{v}[1] = k$. Note that the tropical semiring is bounded from below (and the vector entries are natural numbers), so the minimum is defined. Then, for all vectors $\bar{v}' \in M$, it holds that $\bar{v}' \leq \bar{v} \Leftrightarrow \forall 1 < i \leq n + 1 : \bar{v}'[i] \leq \bar{v}[i]$. The induction hypothesis yields that there is no infinite set of n -dimensional vectors, such that there exists no pair of vectors v, v' , where $v \geq v'$. Thus, there must exist a pair of vectors v_1, v_2 in M such that $v_1 \geq v_2$ and $v_1 \neq v_2$. This is a contradiction, so the assumed infinite set M cannot exist, concluding the proof. \square

Anhang C

The Lattice Monad is a Monad (Chapter 7)

Here, we will prove that the lattice monad, Definition 7.2.4 is a monad. We will restate the definitions of the components of the monad in-place for easier reading.

Definition C.1 *Let \mathbb{L} be a complete lattice satisfying the join-infinite distributive law:*

$$\ell \sqcap \bigsqcup L = \bigsqcup \{\ell \sqcap \ell' \mid \ell' \in L\}, \quad (\text{for any } L \subseteq \mathbb{L}). \quad (\text{JID})$$

For an ordered set (X, \leq_X) , let $T(X, \leq_X) = (TX, \leq_{TX})$, where $TX = (X \rightarrow \mathbb{L})^$ is the set of all those functions $b: X \rightarrow \mathbb{L}$ satisfying the following restrictions:*

1. $\bigsqcup_{x \in X} b(x) = \top$
2. *For any $x, x' \in X$, we have $b(x) \sqcap b(x') = \bigsqcup \{b(y) \mid y \leq x, y \leq x'\}$.*
3. *For any join-irreducible element $\ell \in \mathcal{J}(\mathbb{L})$, we have*

$$\exists_{x \in X} \ell \leq_{\mathbb{L}} b(x) \wedge \forall_{x' \in X} (\ell \leq_{\mathbb{L}} b(x') \implies x \leq_X x') .$$

Furthermore, for any two functions $b, b' \in (X \rightarrow \mathbb{L})^$ we let¹*

$$b \leq_{TX} b' \iff \forall_{x \in X} b'(x) \leq_{\mathbb{L}} b(x) .$$

¹Note the reversal of the order in the ordering of functions!

Lastly, for an order preserving function $f : (X, \leq_X) \rightarrow (Y, \leq_Y)$, we fix

$$Tf(b)(y) = \bigsqcup_{f(x) \leq_Y y} b(x), \quad \text{for any } b \in (X \rightarrow \mathbb{L})^*, y \in Y.$$

It should be noted that Conditions 1 and 2 in the above definition are redundant in the case of finite distributive lattices in the following sense. Given a finite distributive lattice \mathbb{L} , an order preserving function $b : X \rightarrow \mathbb{L}$ satisfying Condition 3 also satisfies Condition 1. Moreover, Condition 3 and order preservation imply Condition 2 in case of finite lattices.

For Condition 1, we note $\top = \bigsqcup_{\ell \in \mathcal{J}(\mathbb{L})} \ell \leq_{\mathbb{L}} \bigsqcup_{\ell \leq b(x)} b(x_\ell) \leq_{\mathbb{L}} \bigsqcup_{x \in X} b(x)$. Here, $x_\ell = \min\{x \mid \ell \leq_{\mathbb{L}} b(x)\}$ whose existence is guaranteed by Condition 3.

For Condition 2, we first note that the inequality $\bigsqcup_{y \leq_X x, y \leq_X x'} b(y) \leq_{\mathbb{L}} b(x) \sqcap b(x')$ (for any $x, x' \in X$) is equivalent to demanding that the function b is order preserving. It is obvious that Condition 2 implies b is order preserving, simply let $x = x'$. On the other hand, if b is order preserving, then from $y \leq x$ and $y \leq x'$ it follows that $b(y) \leq b(x)$ and $b(y) \leq b(x')$. Therefore, $b(x) \sqcap b(x') = \bigsqcup\{\ell \mid \ell \leq b(x) \wedge \ell \leq b(x')\} \geq \bigsqcup\{b(y) \mid y \leq x \wedge y \leq x'\}$.

On the other hand, the inequality $\bigsqcup_{y \leq_X x, y \leq_X x'} b(y) \geq_{\mathbb{L}} b(x) \sqcap b(x')$ follows from Condition 3 *in the finite case*, because then

$$\begin{aligned} b(x) \sqcap b(x') &= \bigsqcup\{\ell \in \mathcal{J}(\mathbb{L}) \mid \ell \leq b(x) \wedge \ell \leq b(x')\} \\ &= \bigsqcup\{b(x_\ell) \mid b(x_\ell) \leq b(x) \wedge b(x_\ell) \leq b(x')\}. \end{aligned}$$

Lastly, Condition 3 is required to rule out the cases when X has an infinite descending chain $x_1 \geq x_2 \geq x_3 \cdots$ such that $\ell \leq_{\mathbb{L}} b(x_i)$ (for all $x_i \in X$). This in-turn is necessary to construct a corresponding function $C_b \in X^{\mathcal{J}(\mathbb{L})}$ for a given $b \in TX$ such that $C_b(\ell) = \min\{x \mid \ell \leq_{\mathbb{L}} b(x)\}$ (for every $\ell \in \mathcal{J}(\mathbb{L})$), which is required to prove isomorphism of the reader monad and the lattice monad we are now constructing.

Next we prove that T is well-defined. Henceforth, we omit the index from the ordering relation, whenever it is clear from the context.

Lemma C.2 *Every function in $(X \rightarrow \mathbb{L})^*$ is order preserving.*

Proof: Let $x \leq x'$, for some $x, x' \in X$. Then,

$$b(x) \sqcap b(x') = \bigsqcup_{x'' \leq x, x'' \leq x'} b(x'') = \bigsqcup_{x'' \leq x} b(x'') = b(x).$$

Thus, $b(x) = b(x) \sqcap b(x')$, i.e., $b(x) \leq b(x')$. □

Lemma C.3 *If $f : X \rightarrow Y$ is order preserving and $b \in TX$, then $Tf(b) \in T(Y)$.*

Proof: We need to show that $Tf(b)$ satisfies the three conditions of Definition C.1. For Condition 1, we find $\bigsqcup_{y \in Y} Tf(b)(y) = \bigsqcup_{y \in Y} \bigsqcup_{f(x) \leq y} b(x) = \bigsqcup_{x \in X} b(x) = \top$.

For Condition 2, let $y, y' \in Y$. We need to show that $Tf(b)(y) \sqcap Tf(b)(y') = \bigsqcup_{\bar{y} \leq y \wedge \bar{y} \leq y'} Tf(b)(\bar{y})$. By expanding the right hand side, we get

$$\bigsqcup_{\bar{y} \leq y \wedge \bar{y} \leq y'} Tf(b)(\bar{y}) = \bigsqcup_{\bar{y} \leq y \wedge \bar{y} \leq y'} \bigsqcup_{f(x) \leq \bar{y}} b(x) = \bigsqcup_{f(x) \leq y \wedge f(x) \leq y'} b(x).$$

For Condition 3, let $\ell \in \mathcal{J}(\mathbb{L})$. Then, there is a minimal element x_0 satisfying $\ell \leq b(x_0)$. Clearly, $\ell \leq b(x_0) \leq \bigsqcup_{f(x) \leq f(x_0)} b(x) = Tf(b)(f(x_0))$. Now let $\ell \leq Tf(b)(y)$. Then, $\ell \leq \bigsqcup_{f(x) \leq y} b(x)$. I.e., $f(x) \leq y$ and $\ell \leq b(x)$, for some $x \in X$. But by Condition 3, we get $x_0 \leq x$. And by order preservation of f we get $f(x_0) \leq f(x) \leq y$; thus, $Tf(b)(f(x_0)) \leq Tf(b)(y)$. \square

Lemma C.4 *If $f : (X, \leq_X) \rightarrow (Y, \leq_Y)$ is order preserving, then so is Tf .*

Proof: Let $f : (X, \leq_X) \rightarrow (Y, \leq_Y)$ be an order preserving function. Then, we want to prove that $Tf : T(X, \leq_X) \rightarrow T(Y, \leq_Y)$ is also order preserving. Let $b \leq_{TX} b'$, for some $b, b' \in (X \rightarrow \mathbb{L})^*$. Then, we have $b'(x) \leq_{\mathbb{L}} b(x)$, for any $x \in X$. Thus, we have

$$Tf(b')(y) = \bigsqcup_{f(x) \leq_Y y} b'(x) \leq_{\mathbb{L}} \bigsqcup_{f(x) \leq_Y y} b(x) = Tf(b)(y).$$

Hence, Tf is order preserving. \square

Now we prove that T is actually a functor.

Lemma C.5 *The mapping T as in Definition C.1 is an endofunctor.*

Proof: We start by showing the preservation of identities:

$$(T\text{id})(b)(y) = \bigsqcup_{\text{id}(x) \leq y} b(x) = \bigsqcup_{x \leq y} b(x) = \bigsqcup_{x=y} b(x) = b(y) = \text{id}_T(b)(y).$$

To show that T respects concatenation of arrows, we derive:

$$\begin{aligned} (Tf \circ Tg)(b)(z) &= Tf(Tg(b))(z) = \bigsqcup_{f(y) \leq z} Tg(b)(y) = \bigsqcup_{f(y) \leq z} \bigsqcup_{g(x) \leq y} b(x), \\ T(f \circ g)(b)(z) &= \bigsqcup_{(f \circ g)(x) \leq z} b(x). \end{aligned}$$

Now we will prove that the sets over which the supremum is taken are identical, making the suprema themselves identical.

- If $f(g(x)) \leq z$, then let $y := g(x)$, then automatically $y \geq g(x)$ and additionally $f(y) \leq z$.
- If $y \geq g(x)$ and $z \geq f(y)$, then $z \geq f(y) \geq f(g(x))$ due to monotonicity of f .

□

Next, we define a unit and a multiplication associated with our functor T , which in turn give rise to the *lattice monad*.

Definition C.6 We define two families of maps $\eta_X : X \rightarrow TX$ and $\mu_X : TT X \rightarrow TX$:

$$\eta_X(x)(x') = \begin{cases} \top & \text{if } x \leq x' \\ \perp & \text{otherwise} \end{cases},$$

$$\mu_X(\hat{B})(x) = \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} (\hat{B}(b) \sqcap b(x)), \quad \text{where } \hat{B} \in ((X \rightarrow \mathbb{L})^* \rightarrow \mathbb{L})^*.$$

In the remainder, we show that the families of functions η and μ are well-defined resulting in natural transformations $\eta: \text{Id} \Rightarrow T$ and $\mu: TT \Rightarrow T$, respectively.

Lemma C.7 The family η_X defined in Definition C.6 is a natural transformation.

Proof: We first show that $\eta_X \in TX$, for any ordered set X . Condition 1 follows directly $\bigsqcup_{x' \in X} \eta_X(x)(x') = \top$. For Condition 2, let $x', x'' \in X$, then we find

$$\begin{aligned} \bigsqcup \{ \eta_X(x)(\bar{x}) \mid \bar{x} \leq x' \wedge \bar{x} \leq x'' \} &= \bigsqcup \{ \top \mid x \leq \bar{x} \wedge \bar{x} \leq x' \wedge \bar{x} \leq x'' \} \\ &= \eta_X(x)(x') \sqcap \eta_X(x)(x''). \end{aligned}$$

Lastly, for Condition 3, given a join-irreducible element $\ell \in \mathcal{J}(\mathbb{L})$, we have $\ell \leq \top = \eta_X(x)(x)$. Moreover, if $\ell \leq \eta_X(x)(x')$, then clearly we have $x \leq x'$ per definition of η_X .

We also need to show that η_X is an arrow in **Poset**. For that purpose let $x \leq x'$ and an arbitrary $\bar{x} \in X$ be given, then

$$\eta_X(x)(\bar{x}) = \begin{cases} \top & \text{if } x \leq \bar{x} \\ \perp & \text{otherwise} \end{cases} \geq \begin{cases} \top & \text{if } x' \leq \bar{x} \\ \perp & \text{otherwise} \end{cases} = \eta_X(x')(\bar{x}).$$

Due to the reversal of orders in TX , this actually means $\eta_X(x) \leq \eta_X(x')$, so η_X is order preserving.

Next, it remains to verify that the function η_X satisfies the naturality square. So let $f : (X, \leq_X) \rightarrow (Y, \leq_Y)$. Then, we find

$$Tf(\eta_X(x))(y) = \bigsqcup_{f(x') \leq_Y y} \eta_X(x)(x') = \bigsqcup_{f(x') \leq_Y y \wedge x \leq_X x'} \top = \bigsqcup_{f(x) \leq_Y y} \top = \eta_Y(f(x))(y).$$

□

Lemma C.8 *The family μ_X defined in Definition C.6 is a natural transformation.*

Proof: First we show that for any $\hat{B} \in ((X \rightarrow \mathbb{L})^* \rightarrow \mathbb{L})^*$, we have $\mu_X(\hat{B}) \in TX$. To see that Condition C.1(1) holds, we expand $\bigsqcup_{x \in X} \mu_X(\hat{B})(x)$ and derive:

$$\begin{aligned} \bigsqcup_{x \in X} \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} (\hat{B}(b) \sqcap b(x)) &= \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} \bigsqcup_{x \in X} (\hat{B}(b) \sqcap b(x)) = \\ \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} \hat{B}(b) \sqcap \left(\bigsqcup_{x \in X} b(x) \right) &= \top. \end{aligned}$$

Using Condition C.1(1) on \hat{B} and b individually.

For Condition C.1(2), the direction $\bigsqcup_{\bar{x} \leq x \wedge \bar{x} \leq x'} \mu_X(\hat{B})(\bar{x}) \leq_{\mathbb{L}} \mu_X(\hat{B})(x) \sqcap \mu_X(\hat{B})(x')$ is straightforward, because $\hat{B}(b) \sqcap b(\bar{x}) \leq \hat{B}(b) \sqcap b(x)$ whenever $\bar{x} \leq x$, for any $b \in TX$, since b is order preserving. For the other direction, we first note by applying the JID law twice that $\mu_X(\hat{B})(x) \sqcap \mu_X(\hat{B})(x') = \bigsqcup_{b, b' \in (X \rightarrow \mathbb{L})^*} (\hat{B}(b) \sqcap b(x) \sqcap \hat{B}(b') \sqcap b'(x'))$. It is sufficient to show that for any $b, b' \in TX$ we have $\hat{B}(b) \sqcap b(x) \sqcap \hat{B}(b') \sqcap b'(x') \leq_{\mathbb{L}} \bigsqcup_{\bar{x} \leq x \wedge \bar{x} \leq x'} \mu_X(\hat{B})(\bar{x})$. So, let $b, b' \in TX$. Then we derive

$$\begin{aligned} \hat{B}(b) \sqcap \hat{B}(b') \sqcap b(x) \sqcap b'(x') &\stackrel{(2)}{=} \left(\bigsqcup_{\bar{b} \leq b \wedge \bar{b} \leq b'} \hat{B}(\bar{b}) \right) \sqcap b(x) \sqcap b'(x') && \text{(JID law)} \\ &= \bigsqcup_{\bar{b} \leq b \wedge \bar{b} \leq b'} (\hat{B}(\bar{b}) \sqcap b(x) \sqcap b'(x')) && \text{(Def. of } \leq_{TX}) \\ &\leq_{\mathbb{L}} \bigsqcup_{\bar{b} \leq b \wedge \bar{b} \leq b'} (\hat{B}(\bar{b}) \sqcap \bar{b}(x) \sqcap \bar{b}(x')) && \text{(Def. C.1(2))} \\ &= \bigsqcup_{\bar{b} \leq b \wedge \bar{b} \leq b'} \left(\hat{B}(\bar{b}) \sqcap \bigsqcup_{\bar{x} \leq x \wedge \bar{x} \leq x'} \bar{b}(\bar{x}) \right) && \text{(JID law)} \\ &= \bigsqcup_{\bar{b} \leq b \wedge \bar{b} \leq b'} \bigsqcup_{\bar{x} \leq x \wedge \bar{x} \leq x'} (\hat{B}(\bar{b}) \sqcap \bar{b}(\bar{x})) \end{aligned}$$

$$\begin{aligned}
&= \bigsqcup_{\bar{x} \leq x \wedge \bar{x} \leq x'} \bigsqcup_{\bar{b} \leq b \wedge \bar{b} \leq b'} (\widehat{B}(\bar{b}) \sqcap \bar{b}(\bar{x})) \\
&\leq_{\mathbb{L}} \bigsqcup_{\bar{x} \leq x \wedge \bar{x} \leq x'} \bigsqcup_{\bar{b} \in TX} (\widehat{B}(\bar{b}) \sqcap \bar{b}(\bar{x})) = \bigsqcup_{\bar{x} \leq x \wedge \bar{x} \leq x'} \mu_X(\widehat{B})(\bar{x}).
\end{aligned}$$

Lastly for Condition C.1(3), let $\ell \in \mathbb{L}$ be a join-irreducible element. Since $\widehat{B} \in ((X \rightarrow \mathbb{L})^* \rightarrow \mathbb{L})^*$, there is a minimal $b_0 \in (X \rightarrow \mathbb{L})^*$ such that $\ell \leq \widehat{B}(b_0)$. I.e., for any $b \in (X \rightarrow \mathbb{L})^*$ with $\ell \leq \widehat{B}(b)$ we have $b(x) \leq b_0(x)$, for all $x \in X$ – again, note that this holds due to the reversal of orders.

Moreover, there is a minimal $x_0 \in X$, such that $\ell \leq b_0(x_0)$, since $b_0 \in (X \rightarrow \mathbb{L})^*$. Then, we find $\ell \leq \widehat{B}(b_0) \sqcap b_0(x_0)$, i.e., $\ell \leq \mu_X(\widehat{B})(x_0)$. Now let $x \in X$ such that $\ell \leq \mu_X(\widehat{B})(x)$. Thus, it remains to be shown that $\mu_X(\widehat{B})(x_0) \leq \mu_X(\widehat{B})(x)$. To see this, we first observe that $\ell = \ell \sqcap \mu_X(\widehat{B})(x) = \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} (\ell \sqcap \widehat{B}(b) \sqcap b(x))$ using the JID law. And since ℓ is join-irreducible, we find $\ell \leq \widehat{B}(b) \sqcap b(x)$ for some $b \in (X \rightarrow \mathbb{L})^*$. Clearly, $\ell \leq b(x) \leq b_0(x)$, (since $b_0 \leq b$) and $\ell \leq \widehat{B}(b)$. Thus, by minimality of x_0 , we find $x_0 \leq x$; hence, $b(x_0) \leq b(x)$.

We further need to show that μ_X is an arrow in **Poset**. For that purpose let $\widehat{B} \leq \widehat{B}'$ and any $x \in X$ be given and compute

$$\mu_X(\widehat{B})(x) = \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} \widehat{B}(b) \sqcap b(x) \geq \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} \widehat{B}'(b) \sqcap b(x) = \mu_X(\widehat{B}')(x)$$

because of the reversal of order: If $\widehat{B} \leq \widehat{B}'$, then $\widehat{B}(b) \geq \widehat{B}'(b)$ for all $b \in (X \rightarrow \mathbb{L})^*$. Now, again because of the reversal of orders – note that $\mu_X(\widehat{B}), \mu_X(\widehat{B}') \in (X \rightarrow \mathbb{L})^*$ – it follows that $\mu_X(\widehat{B}) \leq \mu_X(\widehat{B}')$, so μ_X is order preserving.

Now it remains to show that the μ_X satisfies the naturality square. Let $\widehat{B} \in ((X \rightarrow \mathbb{L})^* \rightarrow \mathbb{L})^*$ and $y \in Y$. Then, we find

$$Tf(\mu_X(\widehat{B}))(y) = \bigsqcup \{ \widehat{B}(b) \sqcap b(x) \mid f(x) \leq y \wedge b \in TX \}.$$

Furthermore,

$$\begin{aligned}
(\mu_Y \circ TTf(\widehat{B}))(y) &= \bigsqcup \{ TTf(\widehat{B})(c) \sqcap c(y) \mid c \in TY \} \\
&= \bigsqcup \left\{ \left(\bigsqcup_{Tf(b) \leq c} \widehat{B}(b) \right) \sqcap c(y) \mid c \in TY \right\} \quad (\text{JID law}) \\
&= \bigsqcup \{ \widehat{B}(b) \sqcap c(y) \mid c \in TY \wedge Tf(b) \leq c \} \\
&= \bigsqcup \{ \widehat{B}(b) \sqcap c(y) \mid c \in TY \wedge \forall_{y' \in Y} c(y') \leq Tf(b)(y') \}.
\end{aligned}$$

Suppose there are $b \in TX, c \in TY$ such that $\forall_{y'} c(y') \leq Tf(b)(y')$. Thus, $c(y) \leq Tf(b)(y) = \bigsqcup_{f(x) \leq y} b(x)$. I.e., $\hat{B}(b) \sqcap c(y) \leq \hat{B}(b) \sqcap \bigsqcup_{f(x) \leq y} b(x)$. Thus, $(\mu_Y \circ TTf(\hat{B}))(y) \leq Tf(\mu_X(\hat{B}))(y)$. The other direction is straightforward, take $c = Tf(b)$. \square

Now that we know that η and μ are well-defined natural transformations, we investigate whether the so-called *unit* and *associative* law of a monad hold.

Lemma C.9 *The associative law holds for μ , i.e. $\mu_X \circ T(\mu_X) = \mu_X \circ \mu_{TX}$.*

Proof: Let $x \in X$ and $\hat{C} \in TTTX$. We first compute $\mu_X \circ \mu_{TX}$ (below $b \in TX, \hat{B} \in TTX$):

$$\begin{aligned}
\mu_X(\mu_{TX}(\hat{C}))(x) &= \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} (\mu_{TX}(\hat{C})(b) \sqcap b(x)) \\
&= \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} \left(\bigsqcup_{\hat{B} \in ((X \rightarrow \mathbb{L})^* \rightarrow \mathbb{L})^*} (\hat{C}(\hat{B}) \sqcap \hat{B}(b)) \sqcap b(x) \right) \quad (\text{JID law}) \\
&= \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} \bigsqcup_{\hat{B} \in ((X \rightarrow \mathbb{L})^* \rightarrow \mathbb{L})^*} (\hat{C}(\hat{B}) \sqcap \hat{B}(b) \sqcap b(x)) \\
&= \bigsqcup_{\hat{B} \in ((X \rightarrow \mathbb{L})^* \rightarrow \mathbb{L})^*} \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} (\hat{C}(\hat{B}) \sqcap \hat{B}(b) \sqcap b(x)) \quad (\text{JID law}) \\
&= \bigsqcup_{\hat{B} \in ((X \rightarrow \mathbb{L})^* \rightarrow \mathbb{L})^*} \hat{C}(\hat{B}) \sqcap \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} (\hat{B}(b) \sqcap b(x)) \\
&= \bigsqcup_{\hat{B} \in ((X \rightarrow \mathbb{L})^* \rightarrow \mathbb{L})^*} (\hat{C}(\hat{B}) \sqcap \mu_X(\hat{B})(x)).
\end{aligned}$$

Furthermore, we compute $\mu_X \circ T(\mu_X)$ as follows:

$$\begin{aligned}
\mu_X(T(\mu_X)(\hat{C}))(x) &= \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} (T(\mu_X(\hat{C}))(b) \sqcap b(x)) \\
&= \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} \left(\left(\bigsqcup_{\mu_X(\hat{B}) \leq b} \hat{C}(\hat{B}) \right) \sqcap b(x) \right) \quad (\text{JID law}) \\
&= \bigsqcup_{b \in (X \rightarrow \mathbb{L})^*} \bigsqcup_{\mu_X(\hat{B}) \leq b} (\hat{C}(\hat{B}) \sqcap b(x)).
\end{aligned}$$

Now, we observe that $\hat{C}(\hat{B}) \sqcap b(x) \leq \hat{C}(\hat{B}) \sqcap \mu_X(\hat{B})(x)$, for any $b \in (X \rightarrow \mathbb{L})^*$ satisfying $b(x') \leq \mu_X(\hat{B})(x')$ (for any $x' \in X$). Thus, $\mu_X(T(\mu_X)(\hat{C}))(x) \leq \mu_X(\mu_{TX}(\hat{C}))(x)$. For the other direction, take $b = \mu_X(\hat{B})$; thus, $\mu_X(\mu_{TX}(\hat{C}))(x) \leq \mu_X(T(\mu_X)(\hat{C}))(x)$. \square

Lemma C.10 *The unit law holds, i.e., $\mu_X \circ \eta_{TX} = id_{TX} = \mu_X \circ T(\eta_X)$.*

Proof: We first expand the left hand side

$$\begin{aligned}\mu_X(\eta_{TX})(b)(y) &= \bigsqcup \{ \eta_{TX}(b)(d) \sqcap d(y) \mid d \in (X \rightarrow \mathbb{L})^* \} \\ &= \bigsqcup \{ d(y) \mid b \leq d \} = \bigsqcup \{ d(y) \mid \forall x \in X : b(x) \geq d(x) \} = b(y).\end{aligned}$$

Furthermore, expanding the right hand side we get

$$\begin{aligned}\mu_X(T\eta_X)(b)(y) &= \bigsqcup \{ T\eta_X(b)(d) \sqcap d(y) \mid d \in (X \rightarrow \mathbb{L})^* \} \\ &= \bigsqcup \{ b(x) \sqcap d(y) \mid d \in (X \rightarrow \mathbb{L})^*, \eta_X(x) \leq d \} \\ &= \bigsqcup \{ b(x) \sqcap d(y) \mid d \in (X \rightarrow \mathbb{L})^*, \forall x' \in X : \eta_X(x)(x') \geq d(x') \} \\ &= \bigsqcup \{ b(x) \mid x \leq y \} = b(y).\end{aligned}$$

□

So we have proven the main result of this section.

Corollary C.11 *The tuple (T, η, μ) is a monad on **Poset**.*

References

- [ABH⁺12] Jiří Adámek, Filippo Bonchi, Mathias Hülsbusch, Barbara König, Stefan Milius, and Alexandra Silva. A coalgebraic perspective on minimization and determinization. In *Proc. of FOSSACS '12*, pages 58–73. Springer, 2012. LNCS/ARCoSS 7213.
- [ABK11] Shaull Almagor, Udi Boker, and Orna Kupferman. What’s decidable about weighted automata? In *Proc. of ATVA '11*, pages 482–491. Springer, 2011. LNCS 6996.
- [ACHV10] Parosh A. Abdulla, Yu-Fang Chen, Lukáš Holík, and Tomáš Vojnar. When simulation meets antichains (on checking language inclusion of NFAs). In *Proc. of TACAS '10*, pages 158–174. Springer, 2010. LNCS 6015.
- [AEI03] Luca Aceto, Zoltán Ésik, and Anna Ingólfssdóttir. Equational theories of tropical semirings. *Theoretical Computer Science*, 298(3):417 – 469, 2003. Foundations of Software Science and Computation Structures.
- [AFL15] Jo M. Atlee, Uli Fahrenberg, and Axel Legay. Measuring behaviour interactions between product-line features. In *Proc. of Formalise '15*, pages 20–25, Piscataway, NJ, USA, 2015. IEEE Press.
- [AHS90] Jiří Adámek, Horst Herrlich, and George E. Strecker. *Abstract and Concrete Categories - The Joy of Cats*. Wiley, 1990.
- [AK95] Jiří Adámek and Vaclav Koubek. On the greatest fixed point of a set functor. *Theoretical Computer Science*, 150:57–75, 1995.

- [And97] Henrik R. Andersen. An introduction to binary decision diagrams. *Course Notes*, 1997.
- [AR94] Jiří Adámek and Jiří Rosický. *Locally Presentable and Accessible Categories*, volume 189 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1994.
- [Bai96] Christel Baier. Polynomial time algorithms for testing probabilistic bisimulation and simulation. In *Proc. of CAV '96*, pages 50–61. Springer, 1996. LNCS 1102.
- [BBB⁺12] Filippo Bonchi, Marcello M. Bonsangue, Michele Boreale, Jan J. M. M. Rutten, and Alexandra Silva. A coalgebraic perspective on linear weighted automata. *Inf. Comput.*, 211:77–105, 2012.
- [BBC⁺16] Filippo Bonchi, Marcello Bonsangue, Georgiana Caltais, Jan J. M. M. Rutten, and Alexandra Silva. A coalgebraic view on decorated traces. *Mathematical Structures in Computer Science*, 26(7):1234–1268, 2016.
- [BÉ93] Stephen L. Bloom and Zoltán Ésik. *Iteration Theories: The Equational Logic of Iterative Processes*. EATCS Monographs on Theoretical Computer Science. Springer, 1993.
- [BÉ09] Stephen L. Bloom and Zoltán Ésik. Axiomatizing rational power series over natural numbers. *Information and Computation*, 207(7):793 – 811, 2009.
- [BGJ13] Guram Bezhanishvili, David Gabelaia, and Mamuka Jibladze. Funayama’s theorem revisited. *Algebra universalis*, 70(3):271–286, 2013.
- [BJ72] Thomas S. Blyth and Melvin F. Janowitz. *Residuation Theory*. Pergamon Press, 1972.
- [BK12] Radim Belohlavek and Jan Konecny. Row and column spaces of matrices over residuated lattices. *Fundam. Inf.*, 115(4):279–295, December 2012.

- [BK17] Harsh Beohar and Sebastian Küpper. On path-based coalgebras and weak notions of bisimulation. In *Proc. of CALCO '17*, 2017. LIPIcs Vol. 72, to appear.
- [BKK13] H.J. Sander Bruggink, Barbara König, and Sebastian Küpper. Concatenation and other closure properties of recognizable languages in adhesive categories. In *Proc. of GT-VMT '13 (Workshop on Graph Transformation and Visual Modeling Techniques)*, volume 58 of *Electronic Communications of the EASST*, 2013.
- [BKK15] H. J. Sander Bruggink, Barbara König, and Sebastian Küpper. Robustness and closure properties of recognizable languages in adhesive categories. *Sci. Comput. Program.*, 104:71–98, 2015.
- [BKK17] Filippo Bonchi, Barbara König, and Sebastian Küpper. Up-to techniques for weighted systems. In *Proc. of TACAS '17, Part I*, pages 535–552. Springer, 2017. LNCS 10205.
- [BKKS17] Harsh Beohar, Barbara König, Sebastian Küpper, and Alexandra Silva. Conditional transition systems with upgrades. In *Proc. of TASE '17 (Theoretical Aspects of Software Engineering)*, 2017. to appear.
- [BKK⁺ed] Harsh Beohar, Barbara König, Sebastian Küpper, Alexandra Silva, and Thorsten Wißmann. Conditional transition systems coalgebraically. submitted. arXiv:1612.05002.
- [BLS05] Marie-Pierre Béal, Sylvain Lombardy, and Jacques Sakarovitch. On the equivalence of \mathbb{Z} -automata. In *Proc. of ICALP '05*, pages 397–409. Springer, 2005. LNCS 3580.
- [BLS06] Mariel-Pierre Béal, Sylvain Lombardy, and Jacques Sakarovitch. Conjugacy and equivalence of weighted automata and functional transducers. In *Proc. of CSR '06*, pages 58–69. Springer, 2006. LNCS 3967.
- [BMS13] Marcello Bonsangue, Stefan Milius, and Alexandra Silva. Sound and complete axiomatizations of coalgebraic language equivalence. *ACM Transactions on Computational Logic*, 14(1), 2013.

- [Bor09] Michele Boreale. Weighted bisimulation in linear algebraic form. In *Proc. of CONCUR '09*, pages 163–177. Springer, 2009. LNCS 5710.
- [BP13] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *Proc. of POPL '13*, pages 457–468. ACM, 2013.
- [BPPR14a] Filippo Bonchi, Daniela Petrisan, Damien Pous, and Jurriaan Rot. Coinduction up to in a fibrational setting. CoRR abs/1401.6675, 2014.
- [BPPR14b] Filippo Bonchi, Daniela Petrisan, Damien Pous, and Jurriaan Rot. Coinduction up-to in a fibrational setting. In Thomas A. Henzinger and Dale Miller, editors, *Proc. of CSL-LICS '14*, pages 20:1–20:9. ACM, 2014.
- [BR88] Jean Berstel and Christophe Reutenauer. *Rational Series and their Languages*. Springer, 1988.
- [Buc08] Peter Buchholz. Bisimulation relations for weighted automata. *Theoretical Computer Science*, 393(1):109 – 123, 2008.
- [CCH⁺13] Maxime Cordy, Andreas Classen, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model checking adaptive software with featured transition systems. In *Assurances for Self-Adaptive Systems*, volume 7740 of *LNCS*, pages 1–29. Springer, 2013.
- [CCP⁺12] Maxime Cordy, Andreas Classen, Gilles Perrouin, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Simulation-based abstractions for software product-line model checking. In *ICSE*, pages 672–682, 2012.
- [CCS⁺13] Andreas. Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans. Softw. Eng.*, 39(8):1069–1089, August 2013.

- [CG79] Raymond A. Cuninghame-Green. *Minimax algebra*. Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, 1979.
- [CHS⁺10] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proc. of ICSE'10*, pages 335–344, NY, USA, 2010. ACM.
- [CLS96] David Cyrluk, Patrick Lincoln, and Natarajan Shankar. On Shostak’s decision procedure for combinations of theories. In *Proc. of CADE-13*, pages 463–477. Springer-Verlag, 1996.
- [CN01] Paul C. Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, 1990.
- [DK13] Manfred Droste and Dietrich Kuske. Weighted automata. In Jean-Eric Pin, editor, *Automata: from Mathematics to Applications*. European Mathematical Society, 2013. to appear.
- [DKB14] Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. Probabilistic model checking for energy analysis in software product lines. In *Proc. of MODULARITY '14*, pages 169–180. ACM, 2014.
- [DKV09] Manfred Droste, Werner Kuich, and Heiko Vogler, editors. *Handbook of Weighted Automata*. Springer, 2009.
- [DM09] Abhijit Das and C. E. Veni Madhavan. *Public-Key Cryptography: Theory and Practice*. Pearson Education, 2009. pp. 295-296.
- [DM12] Manfred Droste and Ingmar Meinecke. Weighted automata and weighted {MSO} logics for average and long-time behaviors. *Information and Computation*, 220-221:44 – 59, 2012.

- [DP02] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 2002.
- [DR10] Laurent Doyen and Jean-François Raskin. Antichain Algorithms for Finite Automata. In *Proc. of TACAS*, pages 2–22. Springer, 2010. LNCS 6015.
- [DV12] Manfred Droste and Heiko Vogler. Weighted automata and multi-valued logics over arbitrary bounded lattices. *Theoretical Computer Science*, 418:14 – 36, 2012.
- [Eis03] Jason Eisner. Simpler and more general minimization for weighted finite-state automata. In *Proc. of HLT-NAACL '03, Volume 1*, pages 64–71. Association for Computational Linguistics, 2003.
- [ÉK01] Zoltán Ésik and Werner Kuich. A generalization of Kozen’s axiomatization of the equational theory of the regular sets. In *Words, Semigroups, and Transductions - Festschrift in Honor of Gabriel Thierrin*, pages 99–114, 2001.
- [ÉK13] Zoltán Ésik and Werner Kuich. Free inductive k -semialgebras. *Journal of Logic and Algebraic Programming*, 82(3–4):111–122, 2013.
- [ÉM10] Zoltán Ésik and Andreas Maletti. Simulation vs. equivalence. In *Proc. of FCS '10*, pages 119–124. CSREA Press, 2010.
- [Ési11] Zoltán Ésik. Multi-linear iterative k - Σ -semialgebras. *ENTCS*, 276:159–170, 2011.
- [Fit02] Melvin Fitting. Bisimulations and Boolean vectors. In *Advances in Modal Logic*, pages 97–126, 2002.
- [FL97] Marianne Flouret and Éric Laugerotte. Noncommutative minimization algorithms. *Information Processing Letters*, 64(3):123–126, 1997.
- [FMT05] Gianluigi Ferrari, Ugo Montanari, and Emilio Tuosto. Coalgebraic minimization of HD-automata for the π -calculus using poly-

morphic types. *Theoretical Computer Science*, 331(2–3):325–365, February 2005.

- [FUB06] Dario Fischbein, Sebastian Uchitel, and Victor Braberman. A foundation for behavioural conformance in software product line architectures. In *Proc. of ROSATEA '06*, pages 39–48. ACM, 2006.
- [GLS08] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Modeling and model checking software product lines. In *Proc. of FMOODS'08*, volume 5051 of *LNCS*, pages 113–131. Springer, 2008.
- [GP97] Stéphane Gaubert and Max Plus. Methods and applications of $(\max, +)$ linear algebra. In *Proc. of STACS '97*, pages 261–282. Springer Berlin Heidelberg, 1997. LNCS 1200.
- [GS13] Carlo Ghezzi and Amir M. Sharifloo. *Dealing with Non-Functional Requirements for Adaptive Systems via Dynamic Software Product-Lines*, pages 191–213. Springer, 2013.
- [HHWZ10] Ernst M. Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. Param: A model checker for parametric Markov models. In *Proc. of CAV '10*, pages 660–664. Springer, 2010.
- [HJ98] Claudio Hermida and Bart Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. Comput.*, 145(2):107–152, 1998.
- [HJS07] Ichiro Hasuo, Bart Jacobs, and Ana Sokolova. Generic trace semantics via coinduction. *Logical Methods in Computer Science*, 3(4:11):1–36, 2007.
- [HK71] John E. Hopcroft and Richard M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report TR 114, Cornell University, 1971.
- [HKK14] Holger Hermanns, Jan Krcál, and Jan Kretínský. Probabilistic bisimulation: Naturally on distributions. *CoRR*, abs/1404.5084, 2014.

- [HMRU00] John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.
- [Hop71] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.
- [HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation*. Addison Wesley, Reading, Massachusetts, 1979.
- [KB85] Ladislav J. Kohout and Wyllis Bandler. Relational-product architectures for information processing. *Inf. Sci.*, 37(1-3):25–37, December 1985.
- [Ker16] Henning Kerstan. *Coalgebraic Behavior Analysis – From Qualitative to Quantitative Analyses*. PhD thesis, Universität Duisburg-Essen, Fakultät für Ingenieurwissenschaften, Abteilung für Informatik und Angewandte Kognitionswissenschaft, 2016.
- [KK14] Barbara König and Sebastian Küpper. Generic partition refinement algorithms for coalgebras and an instantiation to weighted automata. In *Proc. of TCS '14, IFIP AICT*, pages 311–325. Springer, 2014. LNCS 8705.
- [KK16] Barbara König and Sebastian Küpper. A generalized partition refinement algorithm, instantiated to language equivalence checking for weighted automata. *Soft Computing*, pages 1–18, 2016.
- [KKM17] Barbara König, Sebastian Küpper, and Christina Mika. Paws: A tool for the analysis of weighted systems. In *Proc. of QAPL '17 (International Workshop on Quantitative Aspects of Programming Languages and Systems)*, 2017. to appear.
- [KL10] Orna Kupferman and Yoad Lustig. Latticed simulation relations and games. *International Journal of Foundations of Computer Science*, 21(02):167–189, 2010.

- [KMO⁺11] Stefan Kiefer, Andrzej S. Murawski, Joel Ouaknine, Bjoern Wachter, and James Worrell. Language equivalence for probabilistic automata. In *Proc. of CAV '11*, pages 526–540. Springer, 2011. LNCS 6806.
- [Kro91] Daniel Krob. Expressions rationnelles sur un anneau. In *Topics in Invariant Theory: Séminaire d'Algèbre P. Dubreil et M.-P. Malliavin 1989–1990 (40ème Année)*, pages 215–243. Springer, 1991.
- [Kro94] Daniel Krob. The equality problem for rational series with multiplicities in the tropical semiring is undecidable. *International Journal of Algebra and Computation*, 4(3):405–425, 1994.
- [LMdV15] Diego Latella, Mieke Massink, and Erik P. de Vink. Bisimulation of labelled state-to-function transition systems coalgebraically. *Logical Methods in Computer Science*, 11(4), 2015.
- [LS89] Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing (preliminary report). In *Proc. of POPL '89*, pages 344–352. ACM, 1989.
- [Mik15] Christine Mika. Ein generisches Werkzeug für Sprachäquivalenz bei gewichteten Automaten. Master's thesis, Universität Duisburg-Essen, November 2015.
- [Mil89a] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [Mil89b] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [ML71] Saunders Mac Lane. *Categories for the working mathematician*. Graduate texts in mathematics. Springer-Verlag, New York, 1971.
- [Moh97] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23:269–311, 1997.

- [Moh09] Mehryar Mohri. Weighted automata algorithms. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*, pages 213–254. Springer, 2009.
- [MP14] Andreas Metzger and Klaus Pohl. Software product line engineering and variability management: Achievements and challenges. In *Proc. of FOSE’14*, pages 70–84, New York, NY, USA, 2014. ACM.
- [MS80] Masato Morisaki and Ko Sakai. A complete axiom system for rational sets with multiplicity. *Theoretical Computer Science*, 11(1):79–92, 1980.
- [NSL⁺12] Truong K. Nguyen, Jun Sun, Yang Liu, Jin S. Dong, and Yan Liu. Improved BDD-based discrete analysis of timed systems. In *Proc. of FM’12*, volume 7436 of *LNCS*, pages 326–340. Springer, 2012.
- [Par81] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer, 1981.
- [PS11] Damien Pous and Davide Sangiorgi. Enhancements of the coinductive proof method. In Davide Sangiorgi and Jan J. M. M. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, 2011.
- [PT99] John Power and Daniele Turi. A coalgebraic foundation for linear time semantics. In Martin Hofmann, Dusko Pavlović, and Giuseppe Rosolini, editors, *Proc. 8th CTCS Conf.*, volume 29 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.
- [Rah09] George Rahonis. Fuzzy languages. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*, pages 481–517. Springer, 2009.
- [RBB⁺15] Jurriaan Rot, Filippo Bonchi, Marcello Bonsangue, Damien Pous, Jan J. M. M. Rutten, and Alexandra Silva. Enhanced coalgebraic bisimulation. *Mathematical Structures in Computer Science*, pages 1–29, 2015.

- [Rut00] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249:3–80, 2000.
- [Sak09] Jacques Sakarovitch. Rational and recognisable power series. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*, pages 105–174. Springer, 2009.
- [Sch61] Marcel-Paul Schützenberger. On the definition of a family of automata. *Information and Control*, 4(2–3):245–270, 1961.
- [Sta03] Eugene W. Stark. On behaviour equivalence for probabilistic i/o automata and its relationship to probabilistic bisimulation. *Journal of Automata, Languages and Combinatorics*, 8(2):361–395, 2003.
- [Sta09] Sam Staton. Relating coalgebraic notions of bisimulation. In *Proc. of CALCO '09*, pages 191–205. Springer, 2009. LNCS 5728.
- [tBFGM16] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *Journal of Logical and Algebraic Methods in Programming*, 85(2):287 – 315, 2016.
- [tBLLLV15] Maurice H. ter Beek, Axel Legay, Alberto Lluch-Lafuente, and Andrea Vandin. Statistical analysis of probabilistic models of software product lines with quantitative constraints. In *Proc. of SPLC '15*, pages 11–15. ACM, 2015.
- [UH14] Natsuki Urabe and Ichiro Hasuo. Generic forward and backward simulations III: Quantitative simulations by matrices. In *Proc. of CONCUR '14*, pages 451–466. Springer, 2014. LNCS/ARCoSS 8704.
- [WDHR06] Martin De Wulf, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. of CAV '06*, pages 17–30. Springer, 2006. LNCS 4144.

- [Wor05] James Worrell. On the final sequence of a finitary set functor.
Theor. Comput. Sci., 338(1-3):184–199, June 2005.

Index

- F (functor for CTS), 260
- F (operator for CTS), 202
- F (operator for weighted automata), 104
- F_1 (operator for CTS), 202
- F_2 (operator for CTS), 202
- G (operator for conditional bisimilarity with action precedence), 218
- G_1 (operator for conditional bisimilarity with action precedence), 218
- G_2 (operator for conditional bisimilarity with action precedence), 218
- S (monad), 85
- T (lattice functor), 243
- \equiv^X on arrows, 56
- \equiv^X on matrices, 106
- \leq^X on arrows, 56
- \leq^X on matrices, 106
- $\mathbf{Coalg}(F)$ (category), 46
- \mathbf{H} (category), 94
- $\mathbf{Kl}(S)$, 85
- $\mathbf{M}(S)$, 82
- $\mathbf{M}(S)$ (as Kleisli category), 85
- \mathbf{Poset} (category), 36
- \mathbf{Rel} (category), 36
- \mathbf{Set} (category), 35
- \mathcal{P} (on \mathbf{Poset}), 253
- $Path_\sim$ (functor), 77
- \otimes -multiplication, 209
- l -monoid, 23
- PAWS, 277
- Algorithm A, 59
- Algorithm A (language equivalence for weighted automata), 111
- Algorithm B, 61
- Algorithm B (language equivalence for weighted automata), 114
- Algorithm C, 62
- Algorithm D, 66
- antisymmetry, 12
- approximation (of Boolean algebra elements in a lattice), 22
- approximation (ROBDD), 229
- arrow, 34
- associative, 8
- associative (category), 34
- associative law (monad), 43
- BDD, 224
- behavioural equivalence, 48
- behavioural equivalence as a post-fixpoint, 58
- Birkhoff's representation theorem, 18
- bisimulation (CTS without upgrades), 34
- bisimulation (CTS), 195
- bisimulation (LaTS), 198

- bisimulation (LTS), 31
- bisimulation game (CTS), 212
- bisimulation game (LTS), 32
- Boolean algebra, 19
- Boolean expressions, 17
- bounded lattice, 15
- branching bisimulation (coalgebra), 78
- category, 34
- category of matrices, 82
- class of representatives for weighted automata, 89
- coalgebra, 46
- coalgebra homomorphism, 46
- codomain, 34
- commutative, 8
- commute (diagram), 42
- complete lattice, 15
- component (of a natural transformation), 42
- composition functor, 41
- composition of functors, 41
- composition of natural transformations, 43
- concrete category, 40
- conditional bisimulation, 195
- conditional bisimulation (CTS with action precedence), 218
- conditional transition system (coalgebra), 260
- conditional transition system (with upgrades), 193
- conditional transition system (without Upgrades), 33
- configuration, 223
- congruence closure, 136
- congruence closure (proof rules), 136
- congruence closure (rings), 137
- conjugacy, 108
- coslice category, 56
- CTS (coalgebra), 260
- CTS (with upgrades), 193
- CTS (without Upgrades), 33
- CTS with action precedence, 216
- deactivation of upgrades, 216
- deterministic automaton, 25
- deterministic automaton (coalgebra), 71
- distributive lattice, 16
- distributive law (from $(_\Phi)^A$ to $(_^A)^\Phi$), 255
- distributive law (monads), 45
- distributive laws (semiring), 9
- domain, 34
- downward-closed (ROBDD), 228
- downward-closed sets, 17
- embedding (of a finite distributive lattice into a Boolean algebra), 21
- embedding (of semirings into rings), 140
- embedding of a semiring into a field, 140
- endofunctor, 38
- epimorphism, 37
- equivalence of arrows, 56
- equivalence relation, 12
- extension of functors, 45
- factorisation structure on **Poset**, 264

- factorisation structures, 52
- faithful functor, 40
- feature diagram, 223
- featured transition system, 223
- field, 9
- final chain, 51
- Final Chain Algorithm A, 59
- Final Chain Algorithm B, 61
- Final Chain Algorithm C, 62
- Final Chain Algorithm D, 66
- final object, 38
- fixpoint algorithm (conditional bisimilarity), 202
- free monoid, 25
- FTS, 223
- functor, 38
- generator / generating set, 10
- Heyting algebra, 19
- history dependent automaton (coalgebra), 94
- identity functor, 39
- infimum, 14
- instantiation of CTS, 195
- inverse arrow, 37
- irreducible elements, 17
- isomorphic categories, 41
- isomorphism, 37
- JID, 243, 295
- join-infinite distributive law, 243, 295
- join-irreducible elements, 17
- kernel of a function, 96
- kernel of a named function, 96
- Kleisli category, 44
- Kleisli category of matrices, 85
- Kleisli category over **Poset**, 241
- labelled transition system (coalgebra), 71
- labelled transition system (LTS), 30
- language (deterministic automata), 26
- language (non-deterministic automata), 26
- language (weighted automaton), 29
- language equivalence (weighted automaton), 29
- language equivalence (weighted automaton, alternative characterisation), 107
- LaTS, 197
- lattice, 15
- lattice bisimulation, 198
- lattice homomorphism, 16
- lattice monad, 243
- Lattice Transition System, 197
- LTS (coalgebra), 71
- matrix, 10
- matrix multiplication algorithm (for conditional bisimilarity), 210
- maximum, 14
- minimum, 14
- monad, 43
- monoid, 8
- monomorphism, 37
- monotone function (order-preserving function), 12
- morphism, 34

- multiplication (lattice monad), 243
- multiplication (monad), 43
- named function, 94
- named set, 94
- natural isomorphism, 42
- natural transformation, 42
- non-deterministic automaton, 26
- non-deterministic automaton (coalgebra), 47
- non-deterministic automaton in $\mathbf{KI}(\mathcal{P})$, 50
- normal forms, 141
- object, 34
- order on arrows, 56
- order-preserving function (monotone function), 12
- partial order, 12
- partially ordered set, 12
- partition refinement algorithm (LTS bisimilarity), 32
- partition refinement algorithm for conditional bisimilarity, 204
- path, 77
- poset, 12
- powerset functor (\mathbf{Set}), 39
- powerset monad (\mathbf{Set}), 44
- preorder on arrows, 56
- preorder on named functions, 96
- product, 223
- pseudo-factorisation, 53
- pseudo-factorisation (for $\mathbf{KI}(_^\Phi)$), 266
- reader monad (\mathbf{Poset}), 242
- reader monad (\mathbf{Set}), 43
- reflective subcategory, 53
- reflexivity, 12
- relation, 12
- residuum, 19
- rewriting, 141
- rewriting system for a graph, 187
- ring, 9
- ROBDD, 225
- routing protocol (CTS example), 194
- routing protocol with action precedence (CTS example), 217
- semimodule, 10
- semiring, 9
- similarity (weighted automaton), 175
- simulation (weighted automaton), 175
- software product line, 223
- SPL, 223
- split-mono, 58
- subsemimodule, 10
- supremum, 14
- symmetry, 12
- transitivity, 12
- tropical semiring, 9
- unique up to isomorphism, 38
- unit (lattice monad), 243
- unit (monad), 43
- unit law (monad), 43
- upgrade features, 223
- weighted automaton, 29
- weighted automaton (coalgebra), 82
- winning strategy for player 1 (bisimulation game for CTS), 213

winning strategy for player 2 (bisimulation game for CTS), 213