

Automatic Construction of Implicative Theories for Mathematical Domains

DISSERTATION

zur Erlangerung des akademischen Grades
Doctor of Philosophy (Ph. D.)

vorgelegt

der Fakultät Informatik
der Technischen Universität Dresden

von

Dipl.-Math. Artem Revenko
geboren am 11. Juni 1986 in Moskau

Gutachter: Prof. Dr. Bernhard Ganter
Technische Universität Dresden

Prof. Dr. Gernot Salzer
Technische Universität Wien

Eingereicht am: 22.06.2015
Tag der Disputation: 21.08.2015

Preface

According to Wikipedia, logic can be considered as “the use and study of valid reasoning”. In mathematics and computer sciences one is usually interested in *formal* logic, i.e. “the study of inference with purely formal content. An inference possesses a purely formal content if it can be expressed as a particular application of a wholly abstract rule, that is, a rule that is not about any particular thing or property.” The *formality* of the content under logical consideration allows one for examining the reasoning without references to any particular entities. Hence, formal logic offers a uniform approach to reasoning in all possible domains.

An attempt to formalize a domain and bring it to logical discourse leads to an idea of *logical theories*, i.e. a set of axioms describing the domain. We can classify the logical theories with respect to the expressive power of the logical language used to construct the theory. Although very expressive languages are being successfully investigated, and modern computer system are able to arguably efficiently reason using very expressive languages, the composition of queries to the theories and the interpretation of the results of reasoning are left to humans. Thereby, the most intuitive and easy to understand axioms are of a great value. However, the most simple theory is the empty theory. Hence, a good trade-off between *expressiveness* and *complexity* of a language should be kept. Arguably the most useful theories should mimic the way people organize knowledge in their minds. With this in mind we focus on logical theories containing only *implications*, i.e. logical connectives corresponding to the rule of causality “if ... then ...”. We call these theories *implicative theories*. Such theories not only correspond to our requirements, but also have certain theoretical advantages (e.g. polynomial complexity of standard reasoning tasks).

Another advantage of implicative theories is that they can be constructed from data – a collection of facts about particular entities from a domain. However, if the data is not complete, the obtained theory may contain invalid implication. Namely, some implications may be violated by further facts. Entities violating the implications are called *counter-examples*. This suggests an idea of constructing implicative theories

through refinement of existing theories via adding new data.

Moreover, any implicative theory may be compactly represented through its (*implication*) *basis*. All the valid implications of the implicative theory are logical consequences of the basis. In [GD86] an implication basis minimal in cardinality was introduced. Therefore, it is possible to use this basis representation and look for counter-examples to the implications from the basis. The iterative procedure of computing an implication basis from examples, looking for a counter-example, and refining the data with a newly found counter-example is called *Attribute Exploration* (AE) [GW99b].

The mathematical domains are probably the most suitable domains for performing AE, because the mathematical statements, if decidable, are either true or false. Moreover, we may try to generate the desired counter-examples algorithmically.

Examples of using AE or similar ideas include applications in toxicology [BDF⁺03], chemistry [KS05], internet marketing [IK08], constructing the hierarchy of classes in software [ST98, GV05], software debugging [CDFR08, Rev13a], and optimization of the procedure of software testing [Str15].

Structure of Current Study

In the current study we use the framework of *Formal Concept Analysis* [GW99b]. We represent the data in form of a triple (G, M, I) called a *formal context*; the set G is called the set of *objects*; M – the set of *attributes*; I – (binary) relation between objects and attributes.

The study contains a theoretical introduction (Chapter 1) and descriptions of two investigations about automation of the explorations of knowledge in two domains: parametric expressibility of logical functions and interrelations between algebraic identities. The study is presented in a form that allows to consider the two investigations independently. If the reader is only interested in one of the investigations, the reader may completely ignore the other investigation (respective chapter) without losing any information essential for the understanding of the interesting investigation. Hence, only common concepts and notations essential for understanding both investigations are introduced in Chapter 1.

The domains of application of AE are chosen such that the respective implicative theory is of interest to experts from the respective domain. The *goal of both investigations* is to explore the implicative theory of the domain, i.e. find all valid implications (in basis form) over selected attributes (in Chapter 2 the number of

attributes is not fixed).

Below we briefly describe the material of the chapters. For a more thorough description we refer the reader to the summaries at the beginning of each chapter.

In Chapter 2 we are interested in “minimal” (with respect to parametric expressibility) logical functions. Namely, we are busy with the question “which and how many functions do we need to be able to describe all possible functional p-clones – classes of functions closed under parametric expressibility?” In this case objects and attributes are the same functions and the role of I is played by the commutation relation between functions. The standard AE is not suitable for such investigation, therefore an extension is introduced and investigated. Moreover, two different competing approaches to finding counter-examples are introduced and implemented. The elaborated methods and tools allow for successful completion of the exploration.

In Chapter 3 the exploration of algebraic identities is described. The classes of structures defined by algebraic identities – equational classes – are under investigation for several centuries. We only consider identities of size up to 5. It turns out that it is not possible to complete the construction of the implicative theory without considering infinite structures. The structure of possible infinite counter-examples is investigated and, based on this investigation, a method for finding these counter-examples is introduced and implemented. The exploration is successfully finished. A similar investigation was started independently a few years earlier by Dr Peter Kestler [Kes13]. However, the current investigation has different objectives, namely, the automation of the exploration. The methods and tools of this investigation are sufficiently different. Further remarks may be found in Section 3.7. The investigation and some preliminary results were announced in [Pe13], however, the essential part is first presented in the current study.

The present study is in a large part based on two articles [Rev14, Rev15]. Further concepts and ideas have been published in [KR15, RK12, Rev13b]. The main results of the articles are embedded into this study, which represents them coherently in a more wholistic way. The results of the current study were communicated at international conferences (ICFCA 2014, submitted to CLA 2015), at international workshops (AAA 89, EPCL Workshop 2013), and at research seminars (Institut für Algebra TU Dresden, Knowledge Systems group TU Dresden, Theory and Logic group TU Wien, School of Data Analysis and Artificial Intelligence HSE Moscow).

The program code written for the execution of the current investigations is stored at <https://github.com/artreven> in three repositories:

auto_ae: Automatic execution of AE.

commuting_functions: Automatic exploration of p-indecomposable function (see Chapter 2).

bunny Automatic exploration of equational classes (see Chapter 3).

Acknowledgements

I would like to express my gratitude to Professor Bernhard Ganter for giving me a chance of conducting this study, motivating and supporting me during the work, careful proofreading, and mental presence and attention to the project. I would also like to thank Professor Bernhard Ganter for introducing me to his scientific school.

I would like to express my gratitude to Professor Gernot Salzer for introducing me to the research topic, for his support and assistance, attention to the work, careful proofreading, and emotional generosity.

I would like to express my gratitude to Professor Sergei O. Kuznetsov for his effort in organizing the possibility of conducting this study, attention to the work, constant support, and tutorship.

I would like to thank Professor Reinhard Pöschel for his attention to the work. I would like to thank Dr Mikhail Roshchin for giving me a flavor of applied value of logical and mathematical studies, for his trust and support. I would like to thank Alexandre Albano for discussions and proofreading. I would like to thank the whole FCA community for inspiring and friendly atmosphere, especially at the conferences. I would like to thank the current and former members of the institute of algebra at TU Dresden for providing a productive scientific environment.

I would like to thank the organizers of European PhD Program in Computational Logic for giving me an opportunity to communicate my research, obtain international experience, and broaden my horizons in the field of computational logic.

I would like to thank Jelena Goborova for her patience, support, assistance, motivation, trust, and care. I would like to thank my parents Oxana Fedorova and Victor Revenko for all their support and care.

I would like to thank DAAD and Siemens AG for the financial support during the project.

Contents

Preface	i
1 Preliminaries	1
1.1 Lattice Theory	3
1.2 Formal Concept Analysis	5
1.2.1 Lattice Diagrams	7
1.3 Logic	8
1.3.1 First-order Logic	8
1.3.2 Attribute Logic	11
1.4 Practical Implementation Remarks	17
2 Lattice of P-Clones	20
2.1 Expressibility of Functions	23
2.1.1 Compositional Expressibility	23
2.1.2 Parametric Expressibility	24
2.2 Context of Commuting Functions	25
2.3 Finding Counter-examples	26
2.3.1 Strategy: Satisfy Premise	29
2.3.2 Strategy: Violate Conclusion	34
2.3.3 Comparison	37
2.4 Exploration of the Lattice of P-clones	47
2.4.1 Object-Attribute Exploration	51
2.4.2 Implicatively Closed Subcontexts	57
2.5 Results	63
2.6 Conclusion	64
2.7 Remarks	65
Appendix to Chapter 2	67
3 Implicative Theory of Algebraic Identities	72
3.1 Algebras and Algebraic Identities	75

3.2	Context of Algebras and Identities	76
3.2.1	Identities as Attributes	77
3.3	Finding Counter-examples	81
3.3.1	Necessity of Infinite Algebras	82
3.3.2	Finding Infinite Counter-examples	85
3.4	Exploration of Algebraic Identities	94
3.5	Results	94
3.6	Conclusion	95
3.7	Remarks	96
	Appendix to Chapter 3	97
	Conclusion	98
	Description of Attachments	100
	Symbols and Conventions	102
	Index	106
	Bibliography	110

List of Figures

1.1	Concept lattice of the context from Figure 1.6	8
1.2	Dualities between contexts, implication bases, and concept lattices ¹	13
1.3	Scheme of Attribute Exploration	14
1.4	Initial context of convex quadrangles $\mathbb{K}_{\square}^{(0)}$ and its implication basis.	15
1.5	Context of convex quadrangles $\mathbb{K}_{\square}^{(1)}$ with a quadrangle with a right angle and its implication basis	16
1.6	Final context of convex quadrangles $\mathbb{K}_{\square}^{(2)}$ and its implication basis	16
2.1	Function compatible with relation $f \triangleleft \rho$	23
2.2	Functions f and h do not commute	25
2.3	Constraints on decision variables of f from commutation with h	28
2.4	New assignment to decision variable	29
2.5	Illustration to Algorithm 2.5	34
2.6	Counter-examples to $H \rightarrow j, H^{\perp} \leq \overline{j^{\perp}} $	41
2.7	Counter-examples to $H \rightarrow j, H^{\perp} \geq \overline{j^{\perp}} $	42
2.8	Counter-examples to $H \rightarrow j, H^{\perp} \geq \overline{j^{\perp}} $	43
2.9	Counter-examples to $H \rightarrow j, H^{\perp} \geq \overline{j^{\perp}} $	44
2.10	Context \mathbb{K}_{F_4} containing $f_0^u, f_1^u, f_2^u, f_3^u, f_8^b, f_{14}^b, f_{212}^t$	49
2.11	Concept lattice of the context \mathbb{K}_{F_4} from Figure 2.10	52
2.12	Context $\mathbb{K}_0^{(3)}$ of function on domain A_3 containing $f_{3,0}^u, f_{3,1}^u, f_{3,12015}^b$	53
2.13	Concept lattice of the context $\mathbb{K}_0^{(3)}$ from Figure 2.12	54
2.14	Possible commutations of g_1 and g_2 from Proposition 2.4.10	55
2.15	Possible commutations of g_1 and g_2 from Proposition 2.4.11	55
2.16	Partitioning of the context $\mathbb{K}_{F_k^p}$ of all p-indecomposable functions	57
2.17	Context $\mathbb{K}_{F_2^p \cup \{g_1, g_2, g_3\}}$ from Example 2.4.13	58
2.18	Context $\mathbb{K}_{F_2^p \cup \{g_1, g_2, g_4\}}$ from Example 2.4.13	59
2.19	Context $\mathbb{K}_{\{f_0^u, f_1^u, f_{212}^t, f_3^u, g_1, g_2, g_3\}}$ from Example 2.4.14	59
2.20	Functions $f_0^u, f_1^u, f_{212}^t, f_3^u$ are first-order reducible for $\mathbb{K}_{\{f_{150}^t, f_{14}^b, f_8^b\}}$	62
2.21	Concept lattice of the context of functions $f_0^u, f_1^u, f_2^u, f_3^u, f_6^b, f_8^b, f_{14}^b, f_{212}^t$	66

List of Tables

2.1	Decision variables for a function f of arity 1 over a domain $\{0,1\}$. . .	27
2.3	Finding binary counter-example to $\{f_2^u, f_3^u\} \rightarrow f_8^b$ with violate_conclusion	37
2.2	Finding binary counter-example to $\{f_2^u, f_3^u\} \rightarrow f_8^b$ with satisfy_premise	40
2.4	Function f_0^u	67
2.5	Function f_1^u	67
2.6	Function f_2^u	67
2.7	Function f_3^u	68
2.8	Function f_8^b	68
2.9	Function f_{13}^b	68
2.10	Function f_{14}^b	68
2.11	Function f_{150}^t	69
2.12	Function f_{212}^t	69
2.13	Function f_{232}^t	70
2.14	Function $f_{3,0}^u$	70
2.15	Function $f_{3,1}^u$	70
2.16	Function $f_{3,12015}^b$	71
2.17	Function $f_{3,756}^b$	71
3.1	A counter-example \mathcal{A} to $\{x \equiv a * (\triangleright x)\} \rightarrow x \equiv \triangleright(a * x)$	83
3.2	Decision variables in finding infinite algebra \mathcal{A}	88
3.3	Algebra \mathcal{A}_1 from Example 3.3.18	92
3.4	Algebra \mathcal{A}_2 from Example 3.3.18	92
3.5	A counter-example from Example 3.3.19	93
3.6	A counter-example from Example 3.3.20	94
3.7	Pairwise non-equivalent identities of size at most 5.	97

List of Algorithms

2.1	satisfy_premise	31
2.2	f _commutes_with	31
2.3	extend_ f	32
2.4	backtrack_ f	32
2.5	violate_conclusion	35
3.1	check_identity	77
3.2	generate_terms	80
3.3	find_algebra	89
3.4	exists_next_ \mathcal{A}	89
3.5	discover_constraints	90

Chapter 1

Preliminaries

1.1	Lattice Theory	3
1.2	Formal Concept Analysis	5
1.2.1	Lattice Diagrams	7
1.3	Logic	8
1.3.1	First-order Logic	8
1.3.2	Attribute Logic	11
1.4	Practical Implementation Remarks	17

Chapter Summary

Formal Concept Analysis (FCA) is extensively used in the current investigation as the main toolbox for derivation, analysis, and representation of knowledge. Moreover, methods developed within the current study methods add to the methodology of FCA.

In the current chapter the methodology of FCA as well as the basics of logic are presented in a concise manner. All terms and concepts necessary for understanding the main matter are introduced here. The foundations of theories are only presented up to the extent that is necessary to make the current work self-contained. This presentation may not be sufficient for a deeper understanding of the respective theories, therefore we provide references at the beginning of the corresponding sections.

As historically FCA stems from the idea of reshaping the lattice theory towards applications [Wil05] the current chapter starts with presenting the basics of lattice theory. Next follow the basics of FCA itself, followed by the introduction to attribute logic and an active learning technique called “Attribute Exploration”, which will be the main technique for exploring implicative theories.

1.1 Lattice Theory

For further reading we refer the reader to [Bir67, Grä03, DP02].

Let S and N be sets.

Definition 1.1.1. A *binary relation* R between S and N is a set of pairs (s, n) , where $s \in S$ and $n \in N$. In other words $R \subseteq S \times N$, where $S \times N$ is the cartesian product of S and N . We use the notation sRn to indicate that $(s, n) \in R$.

For $R \subseteq S \times S$ we say that R is a binary relation *on* S .

Definition 1.1.2. A binary relation R on S is called a (*partial*) *order* if for all $x, y, z \in S$ the following conditions are satisfied:

1. xRx (reflexivity);
2. xRy and $x \neq y \Rightarrow \text{not } yRx$ (antisymmetry);
3. xRy and $yRz \Rightarrow xRz$ (transitivity).

A set S with a (partial) order \leq defined on S is called a (*partially*) *ordered set* or *poset*, denoted (S, \leq) .

Definition 1.1.3. The *greatest element* of a poset (S, \leq) is an element $x \in S$ such that for all $s \in S : s \leq x$. The greatest element may not exist. The *least element* is defined dually.

Definition 1.1.4. Let (S, \leq) be a poset and $A \subseteq S$. The *greatest lower bound* (or *infimum*) of A is the greatest element $s \in S$ such that for all $a \in A : s \leq a$ (denoted $s = \bigwedge A$), if it exists. The *least upper bound* (*supremum*) is defined dually (denoted $\bigvee A$).

If $A = \{x, y\}$ then we use the infix notation $x \wedge y$ and $x \vee y$ to denote the infimum and the supremum of A , respectively.

Definition 1.1.5. A poset $\mathcal{L} = (L, \leq)$ is called a *lattice* if for all $x, y \in L$ $x \vee y$ and $x \wedge y$ exist.

Remark. In this work all lattices are assumed to be finite and nonempty. If $x \vee y$ and $x \wedge y$ exist for all x, y of a finite nonempty lattice L then $\bigwedge A$ and $\bigvee A$ exist for all $A \subseteq L$. Therefore, any finite nonempty lattice is a *complete lattice*.

Chapter 1 Preliminaries

Definition 1.1.6. Let $x \leq y$ for $x, y \in L$. The element x is called the *lower neighbor* of y if there does not exist an element $z \in L$ such that $z \neq y, z \neq x, x \leq z \leq y$; we also say that y *covers* x . Let $\mathbb{0}$ and $\mathbb{1}$ be the least and the greatest elements of \mathcal{L} , respectively. An element $x \in L$ is called an *atom* if it has the only lower neighbor $\mathbb{0}$; *coatom* is defined dually.

Definition 1.1.7. A subset F of L is called a *filter* if

- $F \neq \emptyset$;
- $x \wedge y \in F$ for all $x, y \in F$;
- $y \in F$ if $x \leq y$ for some $x \in F$.

The least filter containing the element x is called the *principal filter* generated by x . The principal filter of x is denoted by $\uparrow x$

Definition 1.1.8. A subset I of L is called an *ideal* if

- $I \neq \emptyset$;
- $x \vee y \in I$ for all $x, y \in I$;
- $y \in I$ if $y \leq x$ for some $x \in I$.

The least ideal containing the element x is called the *principal ideal* generated by x . The principal ideal of x is denoted by $\downarrow x$.

Definition 1.1.9. For an element $v \in L$ we define:

$$v_* = \bigvee \{x \in L \mid x < v\}; \quad (1.1a)$$

and

$$v^* = \bigwedge \{x \in L \mid v < x\}. \quad (1.1b)$$

The element v is called *supremum-irreducible* if $v \neq v_*$, i.e. v is not representable as the supremum of strictly smaller elements. Likewise, the element v is called *infimum-irreducible* if $v \neq v^*$.

Definition 1.1.10. A set $P \subseteq S$ is called *supremum-dense* in S if for all $s \in S$ there exists $P_* \subseteq P$ such that $s = \bigvee P_*$, i.e. every element s of S is a supremum of some elements of P . The *infimum-dense* subsets are defined dually.

For a finite lattice L the set of all supremum-irreducible elements is supremum-dense in L .

1.2 Formal Concept Analysis

For further reading we refer the reader to [GW99b].

Let G and M be sets and let $I \subseteq G \times M$ be a binary relation between G and M .

Definition 1.2.1. The triple $\mathbb{K} := (G, M, I)$ is called a (*formal*) *context*. The set G is called the set of *objects*. The set M is called the set of *attributes*. A context (G_*, M_*, I_*) such that $G_* \subseteq G$, $M_* \subseteq M$, and $I_* = I \cap G_* \times M_*$ is called a *subcontext* of \mathbb{K} , denoted $(G_*, M_*, I_*) \leq \mathbb{K}$.

Consider mappings $\varphi: \mathcal{P}(G) \rightarrow \mathcal{P}(M)$ and $\psi: \mathcal{P}(M) \rightarrow \mathcal{P}(G)$:

$$\varphi(X) := \{m \in M \mid gIm \text{ for all } g \in X\},$$

$$\psi(A) := \{g \in G \mid gIm \text{ for all } m \in A\}.$$

Mappings φ and ψ define a *Galois connection* between $(\mathcal{P}(G), \subseteq)$ and $(\mathcal{P}(M), \subseteq)$, i.e. $\varphi(X) \subseteq A \Leftrightarrow \psi(A) \subseteq X$. Usually, instead of φ and ψ a single notation $(\cdot)'$ is used. For $X, Y \subseteq G$ we have $(X \cup Y)' = \{m \in M \mid gIm \text{ for all } g \in X \cup Y\} = \{m \in M \mid gIm \text{ for all } g \in X\} \cap \{m \in M \mid gIm \text{ for all } g \in Y\} = X' \cap Y'$, hence, $(X \cup Y)' = X' \cap Y'$. Similarly, for $A, B \subseteq M$ we have $(A \cup B)' = A' \cap B'$.

For any $X, Y \subseteq G$, $A, B \subseteq M$ one has

1. $X \subseteq Y \Rightarrow Y' \subseteq X'$;
2. $A \subseteq B \Rightarrow B' \subseteq A'$.

The operator $(\cdot)''$ is a *closure operator*, i.e. for any $Z, Z_1, Z_2 \subseteq M$ or $Z, Z_1, Z_2 \subseteq G$ the following holds:

1. $Z \subseteq Z''$;
2. $Z'' = Z''''$ (actually $Z' = Z'''$);
3. $Z_1 \subseteq Z_2 \Rightarrow Z_1'' \subseteq Z_2''$.

The set Z'' is called the *closure* of Z in \mathbb{K} . If $Z'' = Z$ then we call Z *closed*.

Definition 1.2.2. A *formal concept* C of a formal context (G, M, I) is a pair (X, A) , $X \subseteq G$, $A \subseteq M$ such that $X' = A$ and $A' = X$. The set X is called the *extent* of C and is denoted by $\text{ext}(C)$, and the set A is called the *intent* of C and is denoted by $\text{int}(C)$.

We also speak about the intent (extent) of a single object g (attribute m) and instead of $\{g\}'$ ($\{m\}'$) we write g' (m').

Chapter 1 Preliminaries

Definition 1.2.3. For a context (G, M, I) , a concept $C_1 = (X, A)$ is a *subconcept* of a concept $C_2 = (Y, B)$ (denoted $C_1 \leq C_2$) if $X \subseteq Y$ or, equivalently, $B \subseteq A$. This defines an order on formal concepts.

The set of all formal concepts $\mathfrak{B}(G, M, I) = \{(X, A) \in \mathcal{P}(G) \times \mathcal{P}(M) \mid A' = X, X' = A\}$ together with the order \leq is called the *concept lattice*, denoted $\underline{\mathfrak{B}}(G, M, I)$. It can be shown that the set of all intents, denoted $\text{Int}(G, M, I)$, is exactly the set of all closed subsets of M . Dually, all extents, denoted $\text{Ext}(G, M, I)$, are all closed subsets of G . therefore, the set of all intents (extents) of a context forms a *closure system*, i.e. if I_1, I_2 are intents of \mathbb{K} then $I_1 \cap I_2$ is also an intent of \mathbb{K} . The set of all extents of (G, M, I) order by set inclusion is dually isomorphic to the set of all intents of (G, M, I) ordered by set inclusion.

The main theorem of FCA consists in the following.

Theorem 1.2.4. *Let (G, M, I) be a context. A concept lattice $\underline{\mathfrak{B}}(G, M, I)$ is a lattice. Let J be an index set. For any*

$$\{(X_j, A_j) \mid j \in J\} \subseteq \mathfrak{B}(G, M, I)$$

suprema and infima are given by

$$\begin{aligned} \bigwedge_{j \in J} (X_j, A_j) &= \left(\bigcap_{j \in J} X_j, \left(\bigcup_{j \in J} A_j \right)'' \right); \\ \bigvee_{j \in J} (X_j, A_j) &= \left(\left(\bigcup_{j \in J} X_j \right)'', \bigcap_{j \in J} A_j \right). \end{aligned}$$

Moreover, an arbitrary complete lattice $\mathcal{L} = (L, \leq)$ is isomorphic to $\underline{\mathfrak{B}}(G, M, I)$ iff there exist mappings $\gamma : G \rightarrow L$ and $\nu : M \rightarrow L$ such that

1. $\gamma(G)$ is supremum-dense in L , $\nu(M)$ is infimum-dense in L ;
2. $\gamma(g) \leq \nu(m)$ iff gIm .

For $g \in G$ we suppose $\gamma(g) = (g'', g')$; (g'', g') is called the *object concept* of g . Likewise, for $m \in M$ we suppose $\nu(m) = (m', m'')$; (m', m'') is called the *attribute concept* of m .

Definition 1.2.5. A context (G, M, I) is called *clarified*, if for any objects $g, h \in G$ from $g' = h'$ it always follows that $g = h$ and, correspondingly, $m' = n'$ implies $m = n$ for all $m, n \in M$.

Definition 1.2.6. An object g is called *reducible* in a clarified context $\mathbb{K} := (G, M, I)$ iff there exists $X \subseteq G \setminus g$ such that $g' = X'$. Reducible attributes are defined dually.

A clarified context without reducible objects is called *reduced*, the procedure of eliminating reducible objects is called *reduction*.

In what follows we introduce other types of reducibility, therefore, we refer to this type of reducibility as *plain* reducibility. If an object is not reducible we call it *irreducible*. The concept lattice $\mathfrak{B}(G, M, I)$ is isomorphic to the concept lattice of the same context after reducing. Note that a new object g with intent g' is going to be reducible in the context (G, M, I) if $g' \in \text{Int}(G, M, I)$.

Reducible objects neither contribute to any implication basis (see Definition 1.3.14) nor to the concept lattice, therefore, if one is only interested in the implicative theory or in the concept lattice of the context reducible objects can be neglected.

1.2.1 Lattice Diagrams

As stated in Theorem 1.2.4, any concept lattice is a lattice. Posets and lattices in particular can be represented by diagrams, a simple and widely used type of diagrams is ordered diagrams [BFR72].

An ordered diagram represents a finite poset in the form of a drawing of its transitive reduction. For a poset (S, \leq) one represents each element of S as a vertex in the plane and draws a line segment or curve that goes upward from x to y whenever y covers x . These curves may cross each other but must not touch any vertices other than their endpoints. Such a diagram, with labeled vertices, uniquely determines its order. Although ordered diagrams were originally devised as a technique for making drawings of posets by hand, they have more recently been created automatically using graph drawing techniques [Zsc07, Fre04].

Regarding the labeling of the diagrams we may formulate the following rules. Every object and attribute concept is labeled by the respective object or attribute. Consider a formal concept (X, A) corresponding to node C of a labeled diagram of concept lattice $\mathfrak{B}(G, M, I)$. An object g belongs to X iff the node with label g lies in the principal ideal $\downarrow C$, m belongs to A iff node with label m lies in the principal filter $\uparrow C$. One can verify correctness of the above labeling.

Lattice diagrams are useful for explicit knowledge representation only as long as they are “readable”. As the number of formal concepts grow the readability of the

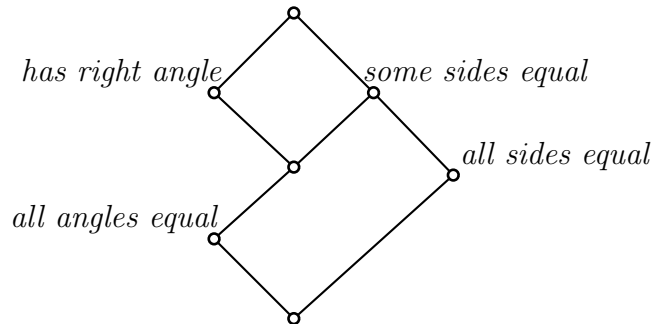


Figure 1.1: Concept lattice of the context from Figure 1.6

lattice decreases. The number of concepts of a formal context may be exponentially large in the size of the context [AdL13]. In the examples in the following chapters lattices are sometimes presented in order to give an optical overview of the data. However, the context obtained in Chapter 2 has 2986 formal concepts, and it is practically inconvenient to work with the diagram of the whole concept lattice of this context.

An example of an ordered diagram is presented in Figure 1.1, only attributes are labeled.

1.3 Logic

1.3.1 First-order Logic

For a thorough study of first-order logic we refer the reader to the following works: [Bar82, Chu96, AS07].

We will only use fragments of first-order logic in this work: the propositional fragment and the universally quantified fragment with equality, but without relations (equational logic). In this subsection we introduce some general notions and notations of first-order logic. More specific definitions will be given in subsequent chapters.

Definition 1.3.1. The alphabet λ of a formal language L of first-order logic consists

Chapter 1 Preliminaries

of the following symbols.

1. Function symbols (possibly indexed): f, g, \dots ;
2. variables (possibly indexed): w, x, y, z, \dots ;
3. relation symbols (possibly indexed): ρ, \dots ;
4. logical connectives: \neg, \wedge ;
5. equality symbol \equiv ;
6. quantifier: \forall ;
7. punctuation: “(”, “)”, “,”, “:”, and “.”.

Each relation symbol ρ and each function symbol f is associated with a natural number called its *arity*, denoted by $ar(\rho)$ and $ar(f)$, respectively. Relation and function symbols with arity 1 (2, 3) are called *unary* (*binary*, *ternary*, respectively). We sometimes use superscripts to indicate the arity of a symbol: $f^{(2)}, f^t$, where “ t ” stands for “ternary”. Function symbols with arity 0 are called *nullary* or *constants*. For constants we use notation a, b, \dots .

A first-order language L is characterized by its alphabet λ and the mapping ar . We use the notation $L = (\lambda, ar)$.

Definition 1.3.2. Let $L = (\lambda, ar)$ be a formal language. A finite string t of symbols from λ is an *L-term* if either

1. t is a variable x , or
2. t is $f(t_1, t_2, \dots, t_{ar(f)})$ for L -terms $t_1, t_2, \dots, t_{ar(f)}$.

Note that constants are L -terms.

If the language is clear from the context, we will say “term” instead of “ L -term”.

Definition 1.3.3. A finite string ϕ of symbols from λ is an *L-formula* if either

1. ψ is $t_1 \equiv t_2$ for L -terms t_1, t_2 ,
2. ψ is $\rho(t_1, t_2, \dots, t_{ar(\rho)})$ for L -terms $t_1, t_2, \dots, t_{ar(\rho)}$,
3. ψ is $\neg\alpha$ for an L -formula α ,
4. ψ is $\alpha \wedge \beta$ for L -formulae α, β ,
5. ψ is $\forall x : \alpha$ for an L -formula α and a variable x .

Formulae corresponding to the first two clauses are called *atomic*.

Chapter 1 Preliminaries

The derived connectives are:

- $\psi \vee \varphi$ for $\neg(\neg\psi \wedge \neg\varphi)$,
- $\psi \Rightarrow \varphi$ for $\neg\psi \vee \varphi$,
- $\psi \Leftrightarrow \varphi$ for $(\psi \Rightarrow \varphi) \wedge (\varphi \Rightarrow \psi)$,
- $\exists x : \psi$ for $\neg(\forall x : (\neg\psi))$.

We will use $\text{TM}(L)$ to denote the set of all L -terms, $\text{Fml}(L)$ to denote the set of all L -formulae, and $\text{Vbl}(L)$ or X to denote the set of all variables.

Definition 1.3.4. Let $L = (\lambda, ar)$ be a formal language. An L -structure \mathcal{A} consists of a set A , called the *universe* (or *carrier*, or *domain*) of \mathcal{A} , an $ar(\rho)$ -ary relation $\rho^{\mathcal{A}}$ on A for each relation symbol ρ , and an $ar(f)$ -ary function $f^{\mathcal{A}}$ on A for each function symbol f .

We call an L -structure \mathcal{A} finite iff A is finite. Otherwise, we call it infinite. $\rho^{\mathcal{A}}$ and $f^{\mathcal{A}}$ are called the *interpretations* of ρ and f , respectively.

Definition 1.3.5. A function $h : \text{Vbl}(L) \rightarrow A$ is called a *variable assignment function into \mathcal{A}* .

The function $\bar{h} : \text{TM}(L) \rightarrow A$, called the *term assignment function generated by h* , is the extension of h to **terms** $\text{TM}(L)$.

Definition 1.3.6. We say \mathcal{A} *satisfies ψ under the assignment h* , denoted $\mathcal{A} \models \psi[h]$, iff

1. ψ is $t_1 \equiv t_2$ for L -terms t_1, t_2 and $\bar{h}(t_1) = \bar{h}(t_2)$;
2. ψ is $\rho(t_1, \dots, t_{ar(\rho)})$ for L -terms $t_1, \dots, t_{ar(\rho)}$ and $(\bar{h}(t_1), \dots, \bar{h}(t_{ar(\rho)})) \in \rho^{\mathcal{A}}$;
3. ψ is $\neg\alpha$ for an L -formula α and $\mathcal{A} \not\models \alpha[h]$, i.e. not $\mathcal{A} \models \alpha[h]$;
4. ψ is $\alpha \wedge \beta$ for L -formulae α and β and both $\mathcal{A} \models \alpha[h]$ and $\mathcal{A} \models \beta[h]$;
5. ψ is $\forall x : \alpha$ for an L -formula α and a variable x and $\mathcal{A} \models \alpha[h]$ no matter what is $h(x) \in A$.

Definition 1.3.7. A formula ψ is said to be *valid* in \mathcal{A} or \mathcal{A} *satisfies ψ* , denoted $\mathcal{A} \models \psi$, if $\mathcal{A} \models \psi[h]$ holds for all variable assignments h . If ψ is valid in every L -structure then ψ is called *valid*.

Let Σ be a set of formulae. We say that \mathcal{A} satisfies Σ if it satisfies every formula in Σ .

Definition 1.3.8. If for all L -structures \mathcal{A} we have that if $\mathcal{A} \models \Sigma$ then $\mathcal{A} \models \psi$ then we say that Σ *logically implies* ψ , denoted $\Sigma \Vdash \psi$.

An *inference rule* for a formal language L is a certain pair of a set of logical formulae Γ and a formula ψ , denoted $\frac{\Gamma}{\psi}$, such that $\Gamma \Vdash \psi$.

Definition 1.3.9. For the alphabet of propositional logic, containing proposition symbols P_0, P_1, \dots , connectives \neg, \wedge , and parentheses, we define a *propositional interpretation* s as function that assigns to every proposition symbol P one of the values from a two-valued domain A_2 . We use either the domain $\{0, 1\}$ or the domain $\{\text{True}, \text{False}\}$.

The **formulae** of propositional logic are defined by using the appropriate subset of clauses from Definition 1.3.3. Every propositional interpretation s can be extended to the interpretation \bar{s} over the set of all **formulae** of propositional logic:

1. $\bar{s}(P) = s(P)$ for every proposition symbol P ;
2. $\bar{s}(\neg\psi) = \text{True}$ iff $\bar{s}(\psi) = \text{False}$;
3. $\bar{s}(\psi_1 \wedge \psi_2) = \text{True}$ iff $\bar{s}(\psi_1) = \text{True}$ and $\bar{s}(\psi_2) = \text{True}$.

If $\bar{s}(\psi) = \text{True}$ we will also write $s \models \psi$.

1.3.2 Attribute Logic

For a deeper study of attribute contextual logic we suggest the work [GW99a].

Definition 1.3.10. An (*attribute*) *implication* of $\mathbb{K} = (G, M, I)$ is defined as a pair (A, B) , where $A, B \subseteq M$, written $A \rightarrow B$. A is called the *premise*, B is called the *conclusion* of the implication $A \rightarrow B$.

Definition 1.3.11. The implication $A \rightarrow B$ is *respected* by a set of attributes N if $A \not\subseteq N$ or $B \subseteq N$. We say that the implication is *respected* by an object g if it is respected by g' .

We consider objects $g \in G$ as propositional interpretations of attribute implication. In order to do so we associate a proposition P_m with every attribute m and a propositional interpretation s_g with every object g . Propositions are interpreted as follows: $s_g(P_m) = \text{True}$ iff $m \in g'$. Usually we use m instead of P_m .

To every set of attributes $A = \{m_1, m_2, \dots, m_{|A|}\}$ corresponds a conjunction of its attributes $\bigwedge A = m_1 \wedge m_2 \wedge \dots \wedge m_{|A|}$. Therefore, $\bar{s}_g(\bigwedge A) = \text{True}$ iff $A \subseteq g'$. We

Chapter 1 Preliminaries

write $g(A)$ and $g(m)$ instead of $\bar{s}_g(\bigwedge A)$ and $s_g(m)$, respectively. We also define an interpretation for negation, namely, $g(\neg m) = \text{True}$ iff $m \notin g'$. Now we can use other derived connectives as defined in the previous section. One can easily verify that $g(\bigwedge A \Rightarrow \bigwedge B) = \text{True}$ iff $A \rightarrow B$ is respected by g . If an object g does not satisfy an implication $(A \rightarrow B)$ then we say that g *violates* $(A \rightarrow B)$ or g is a *counter-example* to $(A \rightarrow B)$. We call objects satisfying an implication the *models* of the implication.

Definition 1.3.12. The implication $A \rightarrow B$ *holds* (is *valid*) in \mathbb{K} if it is respected by all g , $g \in G$, i.e. every object that has all the attributes from A also has all the attributes from B ($B' \subseteq A'$).

New valid implications can be obtained using the Armstrong inference rules:¹

$$\frac{}{A \rightarrow A} \quad , \quad \frac{A \rightarrow B}{A \cup C \rightarrow B} \quad , \quad \frac{A \rightarrow B, B \cup C \rightarrow D}{A \cup C \rightarrow D}$$

Definition 1.3.13. A *unit implication* is an implication with only one attribute in its conclusion, i.e. it is of the form $A \rightarrow \{b\}$, where $A \subseteq M$, $b \in M$. Every implication $A \rightarrow B$ can be regarded as a conjunction of unit implications $\bigwedge \{A \rightarrow b \mid \{b\} \in B\}$ ($g \models (A \rightarrow B)$ iff $g \models \bigwedge \{A \rightarrow \{b\} \mid b \in B\}$). For the purposes of the current study it suffices to consider solely unit implications². We omit brackets in the conclusions of unit implications.

We call a set of implications closed under the application of the Armstrong rules an *implicative theory*. To every implicative theory corresponds a unique up to reducibility formal context and vice versa. Therefore, implicative theories and formal contexts are dual up to reducibility of objects in formal contexts.

Definition 1.3.14. An *implication basis* of a context \mathbb{K} is a set of implications $\mathfrak{L}_{\mathbb{K}}$, from which any valid in \mathbb{K} implication can be deduced by the Armstrong rules and none of the proper subsets of $\mathfrak{L}_{\mathbb{K}}$ has this property.

A basis minimal in the number of implications was defined in [GD86] and is known as the *canonical implication basis*. We use the canonical implication basis, however, our investigations can also be performed using another implication basis.

Figure 1.2 depicts three different ways to represent data.

¹Originally the list of rules contained more rules [Arm74]. All the rules that are not presented here can be derived from the presented ones.

²As long as one is not concerned with the number of implications one can consider only unit implications without loss of generality.

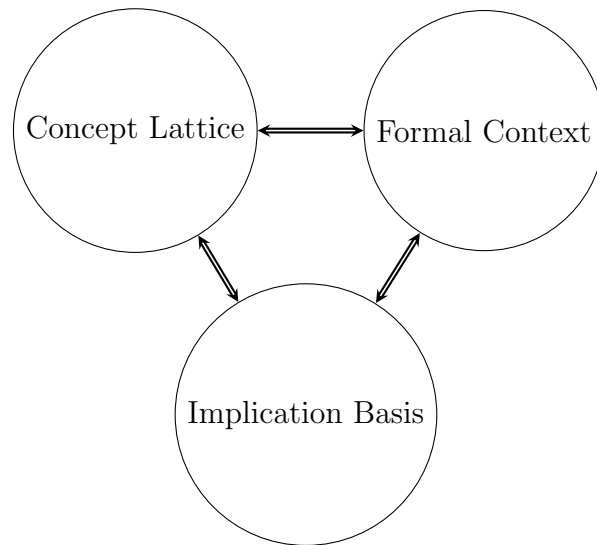


Figure 1.2: Dualities between contexts, implication bases, and concept lattices³

Attribute Exploration (AE) consists in iterations of the following steps until stabilization: computing the implication basis of a context, finding counter-examples to implications, updating the context with counter-examples as new objects, and recomputing the basis (Figure 1.3). AE has been successfully used for investigations in many mostly analytical areas of research. For example, in [KPR06] AE is used for studying Boolean algebras, in [Dau00] lattice properties are studied, in [RK12] function properties are studied.

To start AE it is necessary to have:

1. an initial context;
2. a procedure for finding counter-examples.

Additionally it is possible to make use of a prover for proving implications. In parallel to finding counter-examples it is possible to make an attempt to find a proof.

We illustrate AE with an example about convex quadrangles.

Example 1.3.15. Our formal contexts consist of a subset of the following set of convex quadrangles:

³Note that we can not reconstruct the names of the objects and the reducible objects from concept lattices and implication bases.

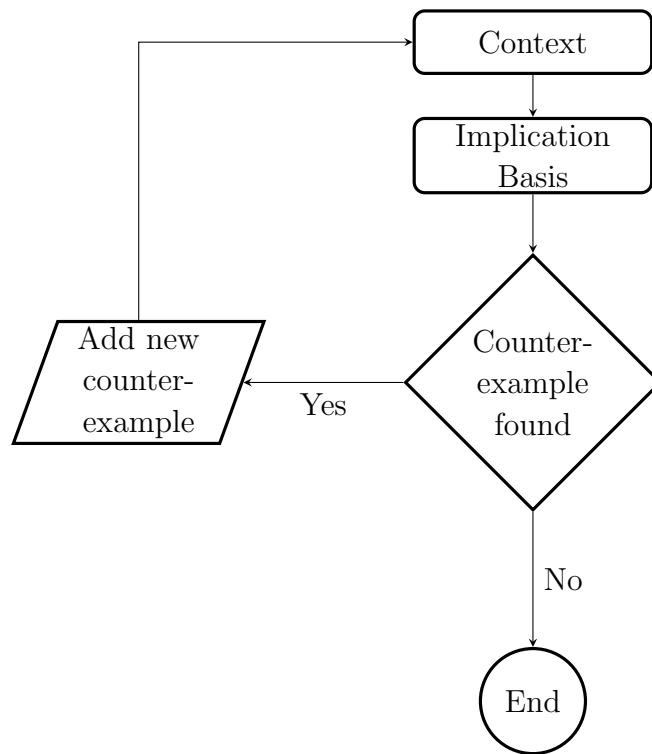


Figure 1.3: Scheme of Attribute Exploration

Chapter 1 Preliminaries

- – a square;
- ▭ – a rectangle;
- ◁ – a quadrangle;
- ◇ – a diamond;
- ▭ – a parallelogram;
- ◁ – a quadrangle with a right angle;
- ◻ – a trapezium with a right angle and two equal sides (left and top);

the following attributes: “all sides equal”, “some sides equal”, “has right angle”, “all angles equal”, and the relation indicating if a quadrangle has respective attribute. The attribute “some sides equal” requires at least two equal sides in a quadrangle. In this example we start with only four objects, see the context $\mathbb{K}_{\square}^{(0)}$ in Figure 1.4. The canonical basis of the context $\mathbb{K}_{\square}^{(0)}$ is in the right side of Figure 1.4.







	some sides equal	has right angle	all sides equal	all angles equal
□	×	×	×	×
▭	×	×		×
◁				
◇	×		×	
▭	×			

1. all sides equal \rightarrow some sides equal;
2. all angles equal \rightarrow some sides equal, has right angle;
3. has right angle \rightarrow some sides equal, all angles equal.

Figure 1.4: Initial context of convex quadrangles $\mathbb{K}_{\square}^{(0)}$ and its implication basis.








Implication 3 from Figure 1.4 is violated by a quadrangle with a right angle. On this step this object is added to the context. The new context $\mathbb{K}_{\square}^{(1)}$ and its implication basis is presented in Figure 1.5.

Implication 3 from Figure 1.5 is violated by a trapezium with a right angle and two equal sides. The new context $\mathbb{K}_{\square}^{(2)}$ and its implication basis is presented in Figure 1.6.

	some sides equal	has right angle	all sides equal	all angles equal
	×	×	×	×
	×	×		×
				
	×		×	
	×			
		×		

1. all sides equal \rightarrow some sides equal;
2. all angles equal \rightarrow some sides equal, has right angle;
3. has right angle, some sides equal \rightarrow all angles equal.

Figure 1.5: Context of convex quadrangles $\mathbb{K}_{\square}^{(1)}$ with a quadrangle with a right angle and its implication basis

	some sides equal	has right angle	all sides equal	all angles equal
	×	×	×	×
	×	×		×
				
	×		×	
	×			
		×		
	×	×		

1. all sides equal \rightarrow some sides equal;
2. all angles equal \rightarrow some sides equal, has right angle;
3. has right angle, all sides equal \rightarrow all angles equal.

Figure 1.6: Final context of convex quadrangles $\mathbb{K}_{\square}^{(2)}$ and its implication basis

All implications from the basis in Figure 1.6 are valid statements in geometry, therefore, no counter-examples exists. AE is finished, the final implication basis represents the implicative theory of the given attributes.

1.4 Practical Implementation Remarks

Algorithms for Computing Canonical Bases

An overview of the algorithms for the construction of the canonical implication bases lies outside of the scope of the current work. For a thorough survey we refer the reader to [KO02]. Here, we classify the algorithms with respect to one criterion only, namely whether an algorithm is able to output implications from the basis during the construction of the basis. Some algorithms, like `NextClosure` [Gan10], output implications from the final implication basis while still computing the basis (other implications), whereas other algorithms, like `IncrementalAlgorithm` [OD03], can only output the whole basis. In the investigation we use the two mentioned algorithms. `NextClosure` usually takes longer to compute the whole basis, however, it is able to output a significant part of the basis before `IncrementalAlgorithm` has computed the whole basis.

If every counter-example is able to significantly change the canonical basis then it makes sense to use `NextClosure` and output implications until the first counter-example is found. Afterwards, this counter-example is added to the context and the computation of the basis is restarted. Otherwise, if we are interested in finding all counter-examples for the current basis, we use `IncrementalAlgorithm`.

Finding Counter-examples

The methods for finding counter-examples are the essential part of constructing the implicative theory. However, these methods may be slow or even semi-decisive (i.e. do not terminate if no counter-example exists). Therefore, in order to be more flexible we use a time constraint for these methods. If needed, we may first use a small constraint and afterwards, if no counter-examples are found, we increase the time constraint.

Description of Algorithms

All the algorithms in the current work are presented using the same notations. Some structures (**assignments**) are global meaning that the changes to the values of these structures inside function calls are not discarded. Every such structure is explicitly mentioned in the description of the algorithms. We use mixed notation, borrowing from natural language (written *in italics*, for example, *not empty*), mathematical notations (written in usual way), and programming notations. For data instances, like tuples, lists, dictionaries, etc., we use **sans serif**; for functions – **typewriter symbols**; for built-in symbols – **bold symbols**. We use `True` and `False` to denote the propositional values. In order to access attributes or methods of some instance we use dot notations, for example, for a set `s` the notation `s.add(a)` denotes the method of adding a new element `a` to the set `s`.

The symbol “`==`” denotes the equality check. The symbol “`←`” is used for assignments. The square brackets “[”, “]” are used:

1. to denote list,
2. as list or tuple indices to access the corresponding element (if used after the name of a list),
3. as dict keys to access the corresponding values (if used after the name of a dict).

Big O Notation and Complexity

For every algorithm we want to “measure” its response to the change of the “size” of the input. Therefore, we analyze the *complexity* of the algorithm. We estimate the behavior of the algorithm through the complexity in the worst case. In order to do this we use the “Big O” notation. This notation characterizes functions according to their growth rates. A description of a function in terms of big O notation only provides an upper bound on the growth rate of the function. For a thorough description see, e.g., [Knu97].

Decision Variables

In describing the computational models for solving problems and developing algorithms we use the notion of *decision variable*. One should understand a decision variable as something that we have control of, and we use this control in order to

Chapter 1 Preliminaries

find a solution to the problem. Decision variables, in contrast to input and output, are not fixed by the problem statement. On the contrary, the choice of the decision variables is a part of finding a solution to the problem. The output of the algorithm should be uniquely reconstructible from the given assignments to the decision variables. For example, in the task of solving Sudoku a natural choice of decision variables would be the values in respective squares. The solution is uniquely reconstructible through these values.

Chapter 2

Automatized Construction of The Lattice of Parametrically Closed Classes of Functions on Three-valued Domain

2.1	Expressibility of Functions	23
2.1.1	Compositional Expressibility	23
2.1.2	Parametric Expressibility	24
2.2	Context of Commuting Functions	25
2.3	Finding Counter-examples	26
2.3.1	Strategy: Satisfy Premise	29
2.3.2	Strategy: Violate Conclusion	34
2.3.3	Comparison	37
2.4	Exploration of the Lattice of P-clones	47
2.4.1	Object-Attribute Exploration	51
2.4.2	Implicatively Closed Subcontexts	57
2.5	Results	63
2.6	Conclusion	64
2.7	Remarks	65
	Appendix to Chapter 2	67

Chapter Summary

The expressibility of functions is a major topic in mathematics and has a long history of investigation. The interest is explainable: when one aims at investigating any kind of functional properties, which classes of functions should one consider? If a function f is expressible through a function h then it often means that f inherits properties of h and should not be treated separately. Moreover, if h in turn is expressible through f then both have similar or even the same properties. Therefore, partition with respect to expressibility is meaningful and can be the first step in the investigation of functions.

With the development of electronics and logical circuits a new question arises: if one wants to be able to express all possible functions, which minimal set of functions should one have at hands? One of the first investigations in this direction was carried out in [Pos42]. There all the Boolean classes of functions closed under expressibility are found and described. Afterwards many important works were dedicated to related problems such as the investigation of the structure of the lattice of functional classes, see for example [Yab60, Ros70]. However, it is known that the lattice of classes of functions closed under expressibility is in general uncountably infinite. In [Kuz79] a more general type of functional expressibility was introduced – parametric expressibility. A significant advantage of this type of expressibility is that for any finite domain A_k the lattice of all classes closed under parametric expressibility classes of functions (p-clones) is finite [BW87]. However, finding this lattice is an extremely complex task. For $k = 2$ the lattice of p-clones was known. For $k = 3$ in a thorough and tedious investigation [Dan77] it was proved that a system of 197 functions suffices to construct the lattice of all p-clones. This investigation carried out without the use of computers lead to a PhD degree.

In this chapter we introduce, develop, and investigate the methods and tools for automating the exploration of the lattice of p-clones. Therefore, this chapter “applied” to A_3 can be seen as complementing the work [Dan77] where a proof of the results obtained with the tools described in this chapter can be found. Namely, in this chapter we answer the question **how** to find all the p-clones, whereas in [Dan77] it is proved that certain functions allow us to construct the desired lattice. The presented methods and tools are extensible to larger domains as well.

In Section 2.1 we give a deeper and more formal introduction to the topic. In Section 2.2 we describe the context of commuting functions and give some general notes.

In the current chapter we face the necessity of finding **finite** counter-examples to implications over functions. Therefore we face the task of violating implicative constraints. We introduce two methods for solving the task and investigate and compare the resulting algorithms in Section 2.3. If the reader is not interested in the practical aspects of the investigation and in the algorithms we suggest to skip this section.

In Section 2.4 the necessary extension of AE is discussed. At the end of the chapter the results are discussed. Some remarks and an appendix can be found after the discussion of results.

Contributions

- New original approach to exploring the lattice of p-clones introduced;
- Two approaches to finding finite counter-examples are introduced;
- The corresponding algorithms are described, compared, implemented;
- An extension of the standard exploration procedure is introduced and investigated;
- The whole procedure is implemented and executed; the obtained results confirm the previously known results;
- It is proved that for certain starting conditions the desired lattice will necessarily be eventually discovered.

2.1 Expressibility of Functions

2.1.1 Compositional Expressibility

Consider a finite set A_k with k elements. Consider functions $f : A_k^{ar(f)} \rightarrow A_k$, the set of all possible functions over A_k is denoted U_k (when k is clear from the context or not important, we may omit the subscript). The particular functions $p_n^k(x_1, \dots, x_n) = x_k$ are called the *projections*. The set of all projections is denoted by Pr . We also use (\mathbf{x}) instead of (x_1, \dots, x_n) .

Let $H \subseteq U_k$. We say that f is *compositionally expressible* through H (denoted $f \leq H$) if the following condition holds:

$$f(\mathbf{x}) \equiv h(j_1(\mathbf{x}), \dots, j_{ar(h)}(\mathbf{x})), \quad (2.1)$$

for some $h, j_1, \dots, j_{ar(h)} \in H \cup Pr$. Note that the functions j_i need *not* essentially depend on all variables (\mathbf{x}) .

We say that an n -ary function f is *compatible* with (or *preserves*) an m -ary relation ρ (denoted $f \triangleleft \rho$) if for any tuples $(x_{11}, x_{12}, \dots, x_{1m}), \dots, (x_{n1}, x_{n2}, \dots, x_{nm}) \in \rho$ we have $(f(x_{11}, \dots, x_{n1}), \dots, f(x_{1m}, \dots, x_{nm})) \in \rho$ (see Figure 2.1). We say that a set of functions H is compatible with a set of relations R if for every $f \in H$ and for every $\rho \in R : f \triangleleft \rho$.

f	f	\dots	
x_{11}	x_{12}	\dots	$\in \rho$
x_{21}	x_{22}	\dots	$\in \rho$
\dots	\dots	\dots	$\in \rho$
$f(x_{11}, x_{21}, \dots)$	$f(x_{12}, x_{22}, \dots)$	\dots	$\in \rho$

Figure 2.1: Function compatible with relation $f \triangleleft \rho$

The set of all relations compatible with all the functions from H is denoted by $Inv(H)$; for a finite set of relations R the set of all functions preserving all relations from R is denoted by $Pol(R)$. It is known that if $f \leq H$ then f is compatible with all relations from $Inv(H)$ ([BKKR69]). Therefore, in order to *separate* f from H , i.e. show that f is not expressible through H , one has to find a relation ρ such that $H \triangleleft \rho$, but $f \not\triangleleft \rho$.

A *functional clone* is a set of functions containing all projections and closed under compositions. The set of all functional clones over a domain of size $k = 2$ forms a countably infinite lattice [Pos42]. However, if $k > 2$ then the set of all functional classes is uncountable [YM59].

2.1.2 Parametric Expressibility

Let $H \subseteq U_k$ and for any $i \in [1, m] : t_i, s_i \in H \cup Pr$. We say that $f \in U_k$ is *parametrically expressible* through H (denoted $f \leq_p H$) if the following condition holds:

$$f(\mathbf{x}) = y \iff \exists \mathbf{w} \bigwedge_{i=1}^m t_i(\mathbf{x}, \mathbf{w}, y) = s_i(\mathbf{x}, \mathbf{w}, y). \quad (2.2)$$

The notation $J \leq_p H$ means that every function from J is parametrically expressible through H . A *parametric clone* (or *p-clone*) is a set of functions closed under parametric expressibility and containing all projections. We consider a special relation f^\bullet of arity $ar(f) + 1$ on A_k called the *graph* of function f . f^\bullet consists of the tuples of the form $(\mathbf{x}, f(\mathbf{x}))$. We say that the functions f and h *commute*, denoted $f \perp h$, if the identity

$$f(h(x_{11}, \dots, x_{1m}), \dots, h(x_{n1}, \dots, x_{nm})) \equiv h(f(x_{11}, \dots, x_{n1}), \dots, f(x_{1m}, \dots, x_{nm})),$$

holds, where $ar(f) = n$ and $ar(h) = m$. It is easy to see that $f \perp h$ holds iff h is compatible with f^\bullet iff f is compatible with h^\bullet . For a set of functions H we write $f \perp H$ to denote that for all $h \in H : f \perp h$. The commutation property is commutative, i.e. $f \perp h$ iff $h \perp f$. See also an example in Figure 2.2. We investigate the commutation property in more details in Section 2.3. Now we only note that checking if two functions f and h commute takes $O(k^{ar(f)*ar(h)})$ operations in the worst case.

The *centralizer* of H is defined by $H^\perp = \{g \in U_k \mid g \perp H\}$. Note that $H^\perp = Pol(H^\bullet)$. In [Kuz79] it is shown that if $f \leq_p H$ then $f \perp H^\perp$.

A function f is called *p-indecomposable* if each system H parametrically equivalent to $\{f\}$ (i.e. $f \leq_p H$ and $H \leq_p f$) contains a function parametrically equivalent to f . Hence, for each p-indecomposable function there exists a class of p-indecomposable functions that are parametrically equivalent to it. From each such class we take only one representative (only one p-indecomposable function) and collect them in a set of p-indecomposable functions denoted by F_k^p . A p-clone H cannot be represented

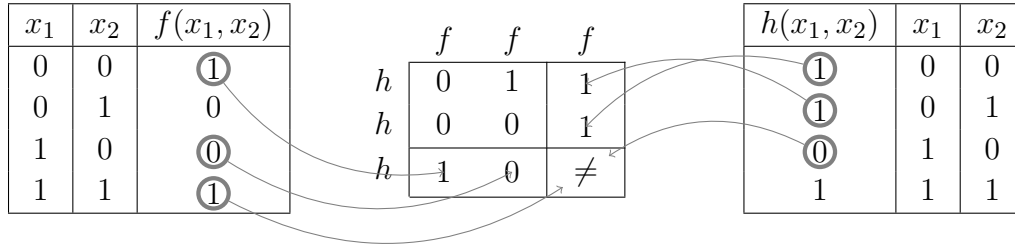


Figure 2.2: Functions f and h do not commute

as an intersection of p-clones strictly containing H if and only if there exists a p-indecomposable function f such that $H = f^{\perp\perp}$. Hence, in order to construct the lattice of all p-clones it suffices to find all p-indecomposable functions. The lattice of all p-clones for any finite k is finite [BW87], hence, F_k^p is finite.

In [BW87] it is proved that it suffices to consider p-indecomposable functions of arity at most k^k , however, the authors conjecture that the actual arity should be equal to k for $k \geq 3$. The conjecture is still open. Nevertheless, thanks to results reported in [Dan77], we know that the conjecture holds for $k = 3$.

2.2 Context of Commuting Functions

The knowledge about the commutation properties of a finite set of functions $F \subseteq U_k$ can be represented as a formal context $\mathbb{K}_F = (F, F, \perp_F)$, where a pair $(f_1, f_2) \in F^2$ belongs to the relation \perp_F iff $f_1 \perp f_2$. Note that the relation \perp_F is symmetric, hence, the objects and the attributes of the context are the same elements and we call them *entities*.

The goal of this chapter is to develop methods and algorithms for constructing the lattice of all p-clones on A_3 . As already noted, for the purpose of constructing the lattice of p-clones it suffices to find all p-indecomposable functions F_k^p . The set of supremum-irreducible elements of the lattice of p-clones is exactly the set $\{f^{\perp\perp} \mid f \in F_k^p\}$.

For any domain of size k there exist k^{k^k} functions of arity k . Therefore, to compute the context of all commuting functions \mathbb{K}_{U_k} one has to perform $O(k^{k^k} * k^{k^k} * k^{k^2})$ operations (taking into consideration only functions of arity k and the cost of commutation check in the worst case). For $k = 3$ we count about 10^{30} operations.

Therefore, already for $k = 3$ a brute-force solution is infeasible.¹ In what follows we introduce a solution inspired by FCA and based on AE. We intend to apply AE to commuting functions. For this purpose we investigate the possibilities of finding counter-examples to implications over functions from U_k . However, as the number of attributes is not fixed, the success of applying AE is not guaranteed, i.e. it is not guaranteed that the complete lattice of p-clones will eventually be discovered using AE.

2.3 Finding Counter-examples

In this section we look for a method that takes an implication over functions and outputs a function that is a counter-example to the implication.

Before developing a computational method to solve the problem we introduce new notations to be able to present statements in a compact and convenient form.

We introduce a numbering on functions. To a binary function f we assign a number $num_f = f(k, k)f(k, k - 1) \dots f(1, 1)$ in base k . For the general case of n -ary f the number num_f in decimal representation is equal to

$$num_f = \sum_{a_1, \dots, a_n \in A_k^n} f(a_1, \dots, a_n) * k^{a_1 * k^{n-1} + \dots + a_n * k^0}.$$

We use superscripts \cdot^u for unary, \cdot^b for binary, and \cdot^t for ternary functions.

Example 2.3.1. For the binary function f such that $f(0, 0) = 0, f(0, 1) = 0, f(1, 0) = 0, f(1, 1) = 1$ (see Table 2.8) the number is $num_f = 0 * 2^0 + 0 * 2^1 + 0 * 2^2 + 1 * 2^3 = 8$, therefore, we denote the function as f_8^b .

Let $(H \rightarrow j)$ be an implication, where $H \subseteq U_k$ is a finite set of functions and $j \in U_k$ is a function. We look for a function f such that $f \perp H, f \not\leq j$. We suppose that the arity of the desired function f is fixed. The decision variables in the problem are the outputs of f , therefore, there are k^n decision variables. In order to distinguish the decision variables from the outputs of the constructed function (although they define one another, they are not the same) we introduce a slightly different notation. We call the set of decision variables V_f . We understand this set as an array of variables

¹Of course one can use dualities, but it does not give a feasible solution as well as there exist only $k!$ dualities.

Chapter 2 Lattice of P-Clones

and we use indices to access entries of the array and write these indices in square brackets, for example, $f(0, 1)$ is the same as $V_f[0, 1]$. In what follows the symbols f and V_f are used interchangeably.

Table 2.1: Decision variables for a function f of arity 1 over a domain $\{0,1\}$

x	$f(x)$
0	$f(0) = V_f[0]$
1	$f(1) = V_f[1]$

Given two functions f and h over a domain A_k checking if they commute requires traversing over all possible matrices

$$M = \begin{pmatrix} m_{1,1} & \dots & m_{1,ar(f)} \\ \dots & \dots & \dots \\ m_{ar(h),1} & \dots & m_{ar(h),ar(f)} \end{pmatrix}, \quad m_{ij} \in A_k.$$

There exist $k^{ar(h)*ar(f)}$ such matrices.

For $i \in [1, ar(f)]$ we denote $m_{\bullet,i} = (m_{1,i}, m_{2,i}, \dots, m_{ar(h),i})$ and for $j \in [1, ar(h)]$ we denote $m_{j,\bullet} = (m_{j,1}, m_{j,2}, \dots, m_{j,ar(f)})$. In these notations for every matrix M it is necessary to check the condition (see Figure 2.3)

$$f(h(m_{\bullet,1}), \dots, h(m_{\bullet,ar(f)})) = h(f(m_{1,\bullet}), \dots, f(m_{ar(h),\bullet})) \quad (2.3)$$

We denote the left hand side of (2.3) $f(Mh)$ or $V_f[Mh]$, i.e.

$$f(Mh) := f(h(m_{\bullet,1}), \dots, h(m_{\bullet,ar(f)})),$$

the right hand side by $(fM)h$, i.e.

$$(fM)h := h(f(m_{1,\bullet}), \dots, f(m_{ar(h),\bullet})).$$

The functions f and h evaluate the condition (2.3) to True or False. We introduce a new propositional symbol $fMh := (f(Mh) = (fM)h)$. We understand (Mh) as a tuple of outputs of h when applied “vertically” (see Figure 2.3) on matrix M . Therefore,

$$(Mh) := (h(m_{\bullet,1}), \dots, h(m_{\bullet,ar(f)})).$$

Similarly, we understand (fM) as a tuple of outputs of f applied “horizontally” (note that the position of \bullet changes)

$$(fM) := (f(m_{1,\bullet}, \dots, m_{ar(h),\bullet})).$$

When function has a tuple as its argument then the postfix notation is the same as the prefix notation, i.e. $(fM)h := h(fM)$. To confirm that two functions commute one has to check the validity of one condition like (2.3) per matrix.

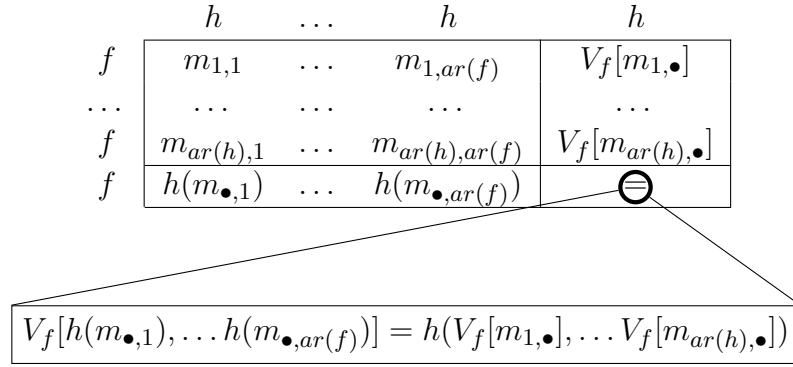


Figure 2.3: Constraints on decision variables of f from commutation with h

Example 2.3.2. In order to commute with f_8^b (see Table 2.8) the values of a unary function f (see Table 2.1) have to satisfy the following constraints:

1. $V_f[0] = f_8^b(V_f[0], V_f[0]);$
2. $V_f[0] = f_8^b(V_f[0], V_f[1]);$
3. $V_f[0] = f_8^b(V_f[1], V_f[0]);$
4. $V_f[1] = f_8^b(V_f[1], V_f[1]).$

We rewrite the condition (2.3) as a constraint on decision variables (see Figure 2.3)

$$\begin{aligned} \forall m_{1,1}, \dots, m_{ar(f),ar(h)} \in A_k : \\ V_f[h(m_{\bullet,1}), \dots, h(m_{\bullet,ar(h)})] = h(V_f[m_{1,\bullet}], \dots, V_f[m_{ar(f),\bullet}]). \end{aligned} \quad (2.4)$$

In what follows we write $\forall M \in A_k^{ar(f)*ar(h)}$ instead of $\forall m_{1,1}, \dots, m_{ar(h),ar(f)} \in A_k$. Therefore, $f \perp h$ holds iff the functions f and h evaluate the condition

$$\bigwedge_{M \in A_k^{ar(f)*ar(h)}} fMh \text{ to True.}$$

We can expand the commutation constraints in the implication and formalize the task. We look for an assignment to the decision variables V_f such that they satisfy:

$$(\forall h \in H. \forall M \in A_k^{ar(f)*ar(h)} : V_f M h) \wedge (\exists M \in A_k^{ar(f)*ar(j)} : \neg V_f M j). \quad (2.5)$$

We introduce two strategies to attack the problem of finding counter-examples.

2.3.1 Strategy: Satisfy Premise

The first strategy is to start from the premise of the implication ($H \rightarrow j$), find functions satisfying the premise H , and afterwards find those of them that do not satisfy the conclusion j . In order to satisfy H a function f has to commute with all functions from H .

As before, let $(\mathbf{x}) = (x_1, \dots, x_{ar(f)})$.

Proposition 2.3.3. *Let $f, h \in U_k$ and $f \perp h$. Fix $f(\mathbf{x}) = a$, $a \in A_k$. Then $f(h(x_1, \dots, x_1), \dots, h(x_{ar(f)}, \dots, x_{ar(f)})) = h(a, \dots, a)$.*

Proof. By direct check of (2.3) □

	h	\dots	h	h
V_f	x_1	\dots	$x_{ar(f)}$	a
\dots	\dots	\dots	\dots	\dots
V_f	x_1	\dots	$x_{ar(f)}$	a
V_f	$[h(x_1, \dots, x_1) \dots h(x_{ar(f)}, \dots, x_{ar(f)})]$			$= h(a, \dots, a)$

Figure 2.4: New assignment to decision variable

As follows from Proposition 2.3.3 after an assignment to a decision variable the constraints on other decision variables may become explicit, i.e. an equality with a single variable, see Figure 2.4. In general, we have two possibilities:

1. $(h(x_1, \dots, x_1), \dots, h(x_{ar(f)}, \dots, x_{ar(f)})) = \mathbf{x}$. Then f may commute with h only if $h(a, \dots, a) = a$.
2. $(h(x_1, \dots, x_1), \dots, h(x_{ar(f)}, \dots, x_{ar(f)})) \neq \mathbf{x}$. In this case we obtain $f(h(x_1, \dots, x_1), \dots, h(x_{ar(f)}, \dots, x_{ar(f)})) = h(a, \dots, a)$.

We proceed until all decision variables are assigned and the constraints arising from the premise are satisfied. After that we check if the found total function f commutes with j . If $f \perp j$ then we return to the previously assigned decision variable and try a new assignment satisfying the premise, if any. We continue until we find such f that $f \not\perp j$.

The algorithm in Algorithm 2.1 implements the described ideas. We give a precise and compact description of the algorithm.

The function `satisfy_premise` systematically explores the search space in a depth first manner. The arity of f can be seen as a parameter of the algorithm. Due to an observation described in Proposition 2.3.3 some branches may be discarded. When checking the violation of conclusion, i.e. $f \not\perp j$, no pruning is performed due to the fact that a matrix M such that $\neg fMj$ has to **exist**. Therefore, we cannot discard the possibility that f does not commute with j until f is defined for all inputs, i.e. f is a total function. The approach starting from violating the conclusion is presented in Subsection 2.3.2.

For all the algorithms in this section we use two global structures:

1. The constructed function f is global. f is extended only in `extend_f` in Algorithm 2.3 and reduced only in `backtrack_f` in Algorithm 2.4.
2. The stack `assignments` is global. `assignments` stores tuples of the form `(input, possible_outputs)`, where for each `input` f is defined and `possible_outputs` have not yet been tried for $f(\text{input})$. `assignments` is modified in `extend_f` in Algorithm 2.3 and in `backtrack_f` in Algorithm 2.4.

At the start f is initialized with the totally undefined function (Line 1). In Lines 2 - 3 the function f is defined for a new input. The function `backtrack_f` returns `None` if the search space is exhausted, hence, in Line 4 we check that the possible assignments are not yet exhausted.

The function `f_commutates_with` makes calls to `extend_f`, hence, in Line 5 the function f may be modified. However, only “forced” assignments are made, i.e. assignments necessary to guarantee that f commutes with H . After entering the **if** clause in Line 6 the algorithm either returns or continues the execution from Line 4, hence, if the possible assignments are exhausted after the call to `backtrack_f` in Line 9 then the **while** loop ends and `None` is returned. If f is not total after checking the commutation with H then we pick some other input for which f is not yet defined and continue (Lines 12 - 13). At last, if f is not able to commute with H with any further assignments, we call `backtrack_f` in Line 15.


```

Input:  $H \rightarrow j$  ( $H \subseteq U_k, j \in U_k$ ),  $ar(f)$ 
Output: Function  $f$  such that  $\forall h \in H. \forall M \in A^{ar(f)*ar(h)} : fMh$  and
 $\exists M \in A^{ar(f)*ar(j)} : \neg fMj$ .

1  $f \leftarrow$  everywhere undefined function of arity  $ar(f)$ 
2 input  $\leftarrow$   $\mathbf{x} \in A^{ar(f)}$  such that  $f(\mathbf{x})$  is not defined
3 extend_f(input,  $A$ )
4 while input  $\neq$  None: // only if assignments are not yet exhausted
5     if  $f\_commutes\_with(H, \text{input})$ :
6         if  $f$  is total:
7             if  $f \not\vdash j$ :
8                 return  $f$ 
9             else:
10                input  $\leftarrow$  backtrack_f()
11        else:
12            input  $\leftarrow$   $\mathbf{x} \in A^{ar(f)}$  such that  $f(\mathbf{x})$  is not defined
13            extend_f(input,  $A$ )
14    else:
15        input  $\leftarrow$  backtrack_f()
16 return None

```

Algorithm 2.1: satisfy_premise

```

Input:  $H \subseteq U_k$ , input  $\in A^{ar(f)}$ 
Output: a Boolean value: for all  $M$  containing input,  $\forall h \in H : fMh$ ?

1 for  $h$  in  $H$ :
2      $M_s \leftarrow \{M \in \mathbb{M}_{f, \text{input}}^{(ar(h))}\}$  // input in M
3     for  $M$  in  $M_s$ :
4         new_input  $\leftarrow$   $Mh$ 
5         output  $\leftarrow$   $(fM)h$ 
6         if  $f(\text{new\_input})$  is not defined:
7             extend_f(new_input, {output})
8             if not  $f\_commutes\_with(H, \text{new\_input})$ :
9                 return False
10        elif  $f(\text{new\_input}) \neq \text{output}$ :
11            return False
12 return True

```

Algorithm 2.2: $f_commutes_with$

Chapter 2 Lattice of P-Clones

The function `f_commutates_with` presented in Algorithm 2.2 checks if it is possible to find an assignment to the decision variables such that f commutes with all functions from H . $\mathbb{M}_{f,\mathbf{x}}^{(s)}$ denotes $(m_{\bullet,1}, \dots, m_{\bullet,s})$, $s \in \mathbb{N}$ such that there exists at least one $i \in [1, s] : m_{\bullet,i} == \mathbf{x}$. For each function $h \in H$ a set of matrices \mathbf{Ms} is constructed; this set contains only such matrices M that f is defined on each row of M and at least one row of M is equal to the tuple `input`. For each matrix M we have three possibilities:

1. $f(Mh)$ is not defined (Lines 6 - 9). In this case we set $f(Mh) := (fM)h$ and recursively call `f_commutates_with` on the newly defined input.
2. $f(Mh) \neq (fM)h$ (Lines 10 - 11). In this case we cannot make new assignments such that f commutes with h , hence, we return `False`.
3. $f(Mh) = (fM)h$. No action is needed, we continue.

Input: `input` $\in A^{ar(f)}$, `outputs` $\subseteq A$

- 1 choose output from `outputs`
- 2 $f(\text{input}) \leftarrow \text{output}$
- 3 `assignments.push(input, (outputs \ {output}))`

Algorithm 2.3: `extend_f`

The function `extend_f` presented in Algorithm 2.3 makes a new assignment to f and modifies the stack assignments.

Output: `latest_input` $\in A^{ar(f)}$

- 1 **if** `assignments` *is not empty*:
- 2 `latest_input, outputs` \leftarrow `assignments.pop()`
- 3 **if** `outputs` *is not empty*:
- 4 `extend_f(latest_input, outputs)`
- 5 **return** `latest_input`
- 6 **else**:
- 7 **delete** $f(\text{latest_input})$
- 8 **return** `backtrack_f(f, assignments)`
- 9 **return** `None`

Algorithm 2.4: `backtrack_f`

The function `backtrack` is presented in Algorithm 2.4. The purpose of the function is to backtrack the assignments, i.e. reassign decision variables. In Line 2 the latest

entry of the `assignments` is taken. The entry consists of the latest input for which f has been defined and the possibly valid outputs that were not tried yet. Note that the stack `assignments` is modified, because the latest entry is popped.

If the possible outputs are not empty then we reassign the output of f and push the tuple of input and outputs back to stack inside `extend_f` in Line 4. If no outputs are left to try we delete the previously assigned output of f in Line 7 and make a recursive call in Line 8.

Algorithm is sound In `satisfy_premise` it is checked that $f \perp H$ and $f \not\leq j$ before returning f . Therefore, if f is output then it satisfies the desired condition.

Algorithm is complete The ordering in `assignments` guarantees that the search branch is only exhausted if a matrix M was found such that for some $h \in H : \neg fMh$. The only case when not all assignments for a certain decision variable are tried is when the decision variable is fixed in `f_commutates_with` in Line 7. But in this case it is guaranteed that no valid assignment is missed. Therefore, the algorithm is complete.

Algorithm terminates As we always proceed in the search space never returning to the already tried assignments and the search space is obviously finite, the algorithm eventually terminates.

The computational complexity of `backtrack_f` in the worst case is dependent on the size of `assignments`. The number of keys in `assignments` is equal to the number of decision variables, hence equal to $k^{ar(f)}$. In the worst case we have to iterate ones over each key, hence the complexity is bounded by $O(k^{ar(f)})$.

Let $m = \max(\{ar(h) \mid h \in H\})$. In the inner loop of `f_commutates_with` there are at most $k^{ar(f)*m}$ possible matrices M . In the outer loop the functions iterates over all possible functions from H , i.e. at most $|H|$ repetitions. Computing fMh takes $O(|M|)$ operations, hence $O(ar(f) * m)$ in the worst case. As the values never get backtracked in this function there are at most $k^{ar(f)}$ possible recursive calls (one for each decision variable). Therefore, the overall complexity of `f_commutates_with` is $O(|H| * k^{ar(f)*m} * ar(f) * m * \underbrace{k^{ar(f)}}_{\text{recursive calls}})$.

In function `satisfy_premise` in the worst case it is necessary to iterate over all possible functions of the given arity and to check if each function commutes with the conclusion (consider the implication $\emptyset \rightarrow f_2^u$). Therefore, the

worst case complexity is bounded by $O(k^{k^{ar(f)}} * (\text{complexity of } f_commutes_with) * (\text{complexity of } backtrack_f) * \underbrace{k^{ar(f)*ar(j)}}_{f \perp j?})$, which is equal to

$$O(k^{k^{ar(f)}+ar(f)*(m+1+ar(j))+1} * m * |H|). \quad (2.6)$$

Note that the only dependence of the complexity on j arises from the final commutation check.

2.3.2 Strategy: Violate Conclusion

The second strategy is to start from the conclusion of the implication $(H \rightarrow j)$, find functions violating the conclusion j , and afterwards find those of them that satisfy the premise H . In order to violate the conclusion, i.e. $f \not\perp j$, there has to exist M such that $\neg fMj$.

Proposition 2.3.4. *Let $f(a_1, \dots, a_{ar(f)}) = b$. If there exists a matrix $M \in A_k^{ar(f)*ar(j)}$ such that $(Mj) = (a_1, \dots, a_{ar(f)})$ and $(fM)j \neq b$ then $f \not\perp j$.*

Proof. As $f(Mj) \neq (fM)j$ then $f \not\perp j$. □

Remark. The converse does not hold.

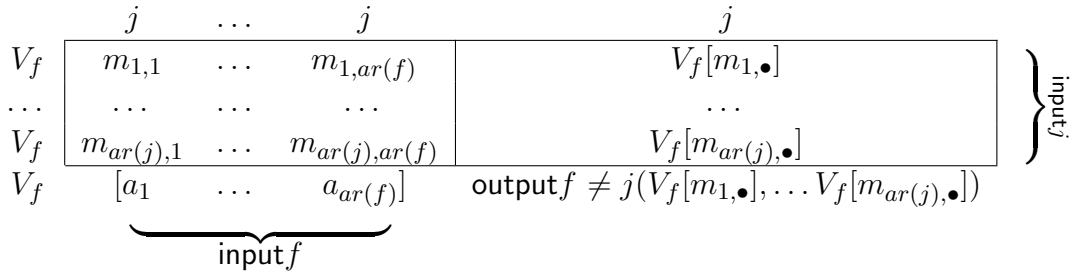


Figure 2.5: Illustration to Algorithm 2.5

The algorithm `violate_conclusion` in Algorithm 2.5 exploits the idea behind Proposition 2.3.4. The algorithm insures that f does not commute with j from the beginning. This is done through proper assignments to $a_1, \dots, a_{ar(f)}$, (fM) , and $f(Mj)$ for some $M \in A^{ar(f)*ar(j)}$, see Figure 2.5. A simple modification of `satisfy_premise` is used to finish the construction. The modification is denoted by

satisfy_premise*; it takes a partial function f and the premise H and outputs a function commuting with H . The only difference is that **satisfy_premise*** makes assignments to only those decision variables that were not assigned at the time of the call.

```

Input:  $H \rightarrow j$  ( $H \subseteq U_k, j \in U_k$ ),  $ar(f)$ 
Output: Function  $f$  such that  $\forall h \in H. \forall M \in A^{ar(h)*ar(f)} : fMh$  and
 $\exists M \in A^{ar(j)*ar(f)} : \neg fMj$ .

1 for input  $f$ , Ms such that (for all  $M \in Ms : Mj = \text{input}f$ ):
2   for output  $f$  in  $A$ :
3     extend_  $f$ (input  $f$ , {output  $f$ })
4     if  $f$ _commutes_with( $H$ , input  $f$ ):
5        $f_{\text{copy}} \leftarrow \text{copy}(f)$ 
6       for  $M$  in Ms:
7         input  $j \leftarrow \{ \text{input}j \in A^{ar(j)} \mid j(\text{input}j) \neq f(\text{input}f) \text{ and}$ 
           it is possible to assign  $fM := \text{input}j \}$ 
8         for input  $j$  in input  $j$ :
9           for row in  $M$  and  $i$  in input  $j$ :
10            extend_  $f$ (row, { $i$ })
11             $f \leftarrow \text{satisfy\_premise}^*(f, H)$ 
12            if  $f == \text{None}$ :
13               $f \leftarrow f_{\text{copy}}$ 
14            else:
15              return  $f$ 
16         else:
17           backtrack_  $f$ () //  $f$  becomes totally undefined
18 return None

```

Algorithm 2.5: violate_conclusion

The algorithm starts from the values $a_1, \dots, a_{ar(f)}$ in Figure 2.5 denoted in the pseudocode by **input** f , Line 1. This is later used as input to f . To each **input** f correspond a set of matrices $Ms \subseteq \mathbb{M}_f^{ar(f)}$ such that for every $M \in Ms : (Mj) = \text{input}f$.

In the second **for** cycle in Line 2 the algorithm tries different assignments to $V_f[\text{input}f]$, **output** f in the pseudocode, see also Figure 2.5. In Line 4 it is checked that f could commute with H with the already performed assignment. For this purpose we use the function f _commutes_with already described in Algorithm 2.2. If it is not possible to commute with H we backtrack f in Line 17 and try the next

assignment to $V_f[\text{input}f]$.

In Line 5 we save a copy of f that we may need later. After that for each M from \mathbf{Ms} we construct a set of possible $\text{input}j$ such that $(fM)j \neq f(Mj)$, i.e. f does not commute with j . The name $\text{input}j$ stems from the fact that each tuple is used as arguments of j afterwards, see also Figure 2.5. There are two things to take care about:

1. For some inputs f is already defined. Hence, for n -th row of M and for n -th element i of $\text{input}j$ we have that $f(\text{row}) \neq i$ then such $\text{input}j$ should be discarded.
2. If n -th row of M is equal to k -th row of M than n -th element of $\text{input}j$ should be equal to k -th element of $\text{input}j$.

In Lines 9 - 10 the new values $\text{input}j$ are assigned to outputs of f . As the numbers of rows in M is equal to the arity of j , i.e. to the size of $\text{input}j$, we can iterate over them in one cycle. Then the algorithm attempts to satisfy the premise via finishing the construction of partially defined f in Line 11. If successful then f is returned in Line 15. Otherwise we return to the previous function f_{copy} in Line 13.

Algorithm is sound In `violate_conclusion` before checking the satisfaction of premise it is guaranteed that $f \not\perp j$ via making one of the appropriate assignments. Afterwards, f is made total in such a way that $f \perp H$. Therefore, if f is output then it satisfies the premise and violates the conclusion.

Algorithm is complete The iteration over all possible pairs $(\text{input}f, \mathbf{Ms})$, outputs of f , and $\text{input}j$ guarantees that no possible assignment violating the conclusion is missed. The completeness of `satisfy_premise` and `f_commutates_with` is already discussed above. Therefore, the algorithm is complete.

Algorithm terminates As we always proceed in the search space never returning to the already tried assignments and the search space is obviously finite, the algorithm eventually terminates.

The complexity of `violate_conclusion` is determined by the complexity of preparatory steps, i.e. preparing appropriate $\text{input}f$ and \mathbf{Ms} in Line 1, and the complexity of the four **for** cycles multiplied by the complexity of `satisfy_premise*` and intermediary functions. The complexity of preparatory steps is $O(k^{\text{ar}(j)*\text{ar}(f)} * k^{\text{ar}(f)})$ and is smaller than the complexity of respective cycles, therefore, it does not add

up to the overall complexity.

The number of pairs ($\text{input } f, \mathbf{Ms}$) is bounded by $O(k^{ar(f)})$; the number of possible output f 's is k . The complexity of `f_commutates_with` is, as before, $O(|H| * k^{ar(f)*m} * ar(f) * m * k^{ar(f)})$; let c_{com} denote this value. There are up to $O(k^{ar(j)*ar(f)})$ many matrices in \mathbf{Ms} ; the size of `inputs j` is bounded by $O(k^{ar(j)})$. In the cycle of extending f we make at most $ar(j)$ assignments. The complexity of `satisfy_premise*` is computed in the same manner as of `satisfy_premise`, except for the factor $k^{ar(f)*ar(j)}$, which is absent due to the fact that there is no need to check if f and j commute; moreover, some outputs of f are already defined at the time of the call. Let c_{sp*} denote the complexity of `satisfy_premise*`. Therefore, the overall worst case complexity is bounded by

$$O(k^{1+ar(f)+ar(j)+ar(j)*ar(f)} * c_{com} * (ar(j) + c_{sp*})). \quad (2.7)$$

2.3.3 Comparison

In this subsection we compare the strategies and the performance of the corresponding algorithms. We start with presenting two examples demonstrating the execution of both algorithms on the same implication $\{f_2^u, f_3^u\} \rightarrow f_8^b$. We use the domain $\{0, 1\}$.

Example 2.3.5. Construction of a counter-example to implication $\{f_2^u, f_3^u\} \rightarrow f_8^b$ using `satisfy_premise` is described in Table 2.2. In usual notations $f_2^u(x) = x$, $f_3^u(x) = 1$, $f_8^b(x, y) = x \wedge y$. The found counter-example is $f_{14}^b(x, y) = x \vee y$.

Example 2.3.6. Construction of a counter-example to implication $\{f_2^u, f_3^u\} \rightarrow f_8^b$ using `violate_conclusion` is represented in Table 2.3. As previously, $f_2^u(x) = x$, $f_3^u(x) = 1$, $f_8^b(x, y) = x \wedge y$. The found counter-example is $f_{13}^b(x, y) = (y \rightarrow x)$.

Table 2.3: Finding binary counter-example to $\{f_2^u, f_3^u\} \rightarrow f_8^b$ with `violate_conclusion`

#	input f	output f	f after <code>f_commutates_with</code>	M	correct_ assignments
1.1.1	(0, 1)	1	$f(0, 1) = 1$ $f(1, 1) = 1$	$\begin{pmatrix} 0, 1 \\ 1, 1 \end{pmatrix}$	\emptyset
1.1.2				$\begin{pmatrix} 1, 1 \\ 0, 1 \end{pmatrix}$	\emptyset
Continued on next page					

Table 2.3 – continued from previous page

#	input f	output f	f after f -commutes-with	M	correct-assignments
1.1.3				$\begin{pmatrix} 0, 1 \\ 0, 1 \end{pmatrix}$	\emptyset
1.2.1		0	$f(0, 1) = 0$ $f(1, 1) = 1$	$\begin{pmatrix} 0, 1 \\ 1, 1 \end{pmatrix}$	\emptyset
1.2.2				$\begin{pmatrix} 1, 1 \\ 0, 1 \end{pmatrix}$	\emptyset
1.2.3				$\begin{pmatrix} 0, 1 \\ 0, 1 \end{pmatrix}$	\emptyset
2.1.1	(1, 0)	1	$f(1, 0) = 1$ $f(1, 1) = 1$	$\begin{pmatrix} 1, 0 \\ 1, 1 \end{pmatrix}$	\emptyset
2.1.2				$\begin{pmatrix} 1, 1 \\ 1, 0 \end{pmatrix}$	\emptyset
2.1.3				$\begin{pmatrix} 1, 0 \\ 1, 0 \end{pmatrix}$	\emptyset
2.2.1		0	$f(1, 0) = 0$ $f(1, 1) = 1$	$\begin{pmatrix} 1, 0 \\ 1, 1 \end{pmatrix}$	\emptyset
2.2.2				$\begin{pmatrix} 1, 1 \\ 1, 0 \end{pmatrix}$	\emptyset
2.2.3				$\begin{pmatrix} 1, 0 \\ 1, 0 \end{pmatrix}$	\emptyset
3.1.1	(0, 0)	1	$f(0, 0) = 1$ $f(1, 1) = 1$	$\begin{pmatrix} 0, 0 \\ 1, 1 \end{pmatrix}$	\emptyset
3.1.3				$\begin{pmatrix} 0, 1 \\ 1, 0 \end{pmatrix}$	$\left\{ \begin{matrix} (0, 1), \\ (1, 0), \\ (0, 0) \end{matrix} \right\}$

Continued on next page

Table 2.3 – continued from previous page

#	input f	output f	f after $f_commutes_with$	M	correct_assignments
	Output:	$f_{13}^b :=$	$\begin{bmatrix} f(0, 0) = 1 \\ f(0, 1) = 0 \\ f(1, 0) = 1 \\ f(1, 1) = 1 \end{bmatrix}$		

As follows from examples, the algorithms explore search space in different manners and, therefore, may arrive at different solutions. Note that the case of valid implication (no counter-examples exist) is more challenging for the algorithms as it is necessary to iterate over all functions that possibly fit (i.e. either all functions satisfying premise or all functions violating the conclusion).

For a set of functions H we use the notation H^\perp to denote the set $\{f \in U_k \mid f \perp H\}$. Consider an implication $(H \rightarrow j)$. The function `satisfy_premise` in Algorithm 2.1 iterates over all functions that satisfy the premise of an implication. Therefore, it is more efficient in the case the set of functions satisfying the premise is smaller than the set of functions violating the conclusion. In Figure 2.6 the case of small set of functions satisfying the premise H^\perp is represented. It is possible that the function `violate_conclusion` finds a counter-example faster only because the first violating assignment contains a counter-example. However, if there does not exist a counter-example (see Figure 2.7) then `satisfy_premise` outperforms `violate_conclusion` on nearly all examples.

The function `violate_conclusion` in Algorithm 2.5 is more efficient in the case the set of functions satisfying the premise is larger than the set of functions violating the conclusion. Figure 2.8 represents the case when a counter-example exists, Figure 2.9 represents the case when there exist no counter-examples. It is worth noting that on simple implications `satisfy_premise` may outperform `violate_conclusion` even in the case no counter-examples exist, see in Figure 2.9, due to the preparatory steps in `violate_conclusion`. However, as implications become more complex (the size of domain and the number of functions in the premise increase) the function `violate_conclusion` clearly outperforms `satisfy_premise`.

The size of M s for each `input f` (see Algorithm 2.5 Line 1) is dependent on the number of matrices M such that $(Mj) = \text{input } j$. In the worst case the number is $A_k^{ar(j)}$.

Table 2.2: Finding binary counter-example to $\{f_2^u, f_3^u\} \rightarrow f_8^b$ with `satisfy_premise`

#	new_input	f	assignments
1	(0,0)	$f(0,0) = 0$	(0,0): [1]
		In <code>f_commutates_with</code> : from commutation with f_3^u over (0,0) we obtain an assignment $f(1,1) = 1$.	
2	(0,1)	$f(0,0) = 0$ $f(1,1) = 1$ $f(0,1) = 0$	(0,0) : [1] (1,1) : [] (0,1) : [1]
		In <code>f_commutates_with</code> : commutes with both function on defined inputs.	
3	(1,0)	$f(0,0) = 0$ $f(1,1) = 1$ $f(0,1) = 0$ $f(1,0) = 0$	(0,0) : [1] (1,1) : [] (0,1) : [1] (1,0) : [1]
		In <code>f_commutates_with</code> : commutes with both function on defined inputs. Commutates with $f_8^b \Rightarrow$ <code>backtrack_f</code> . In <code>backtrack_f</code> : $f(1,0) = 1$.	
4	(1,0)	$f(0,0) = 0$ $f(1,1) = 1$ $f(0,1) = 0$ $f(1,0) = 1$	(0,0) : [1] (1,1) : [] (0,1) : [1] (1,0) : []
		In <code>f_commutates_with</code> : commutes with both function on defined inputs. Commutates with $f_8^b \Rightarrow$ <code>backtrack_f</code> . In <code>backtrack_f</code> : $f(0,1) = 1, f(1,0)$ undefined.	
5	(1,0)	$f(0,0) = 0$ $f(1,1) = 1$ $f(0,1) = 1$ $f(1,0) = 0$	(0,0) : [1] (1,1) : [] (0,1) : [] (1,0) : [1]
		In <code>f_commutates_with</code> : commutes with both function on defined inputs. Commutates with $f_8^b \Rightarrow$ <code>backtrack_f</code> . In <code>backtrack_f</code> : $f(1,0) = 1$.	
	Output:	$f_{14}^b :=$	$\begin{bmatrix} f(0,0) = 0 \\ f(1,1) = 1 \\ f(0,1) = 1 \\ f(1,0) = 1 \end{bmatrix}$

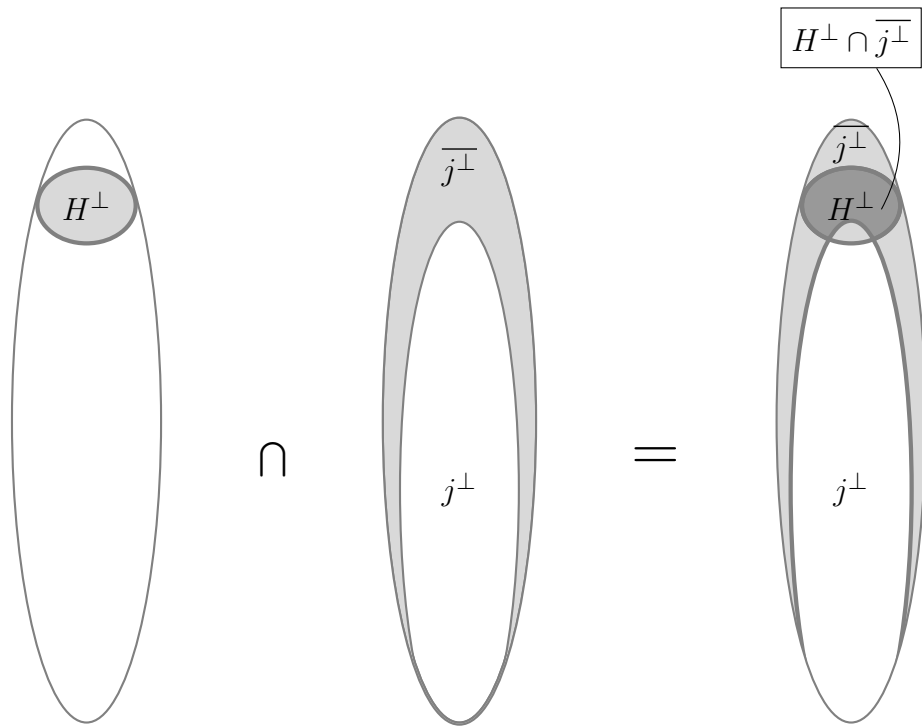


Figure 2.6: Counter-examples to $H \rightarrow j, |H^\perp| \leq |\overline{j^\perp}|$

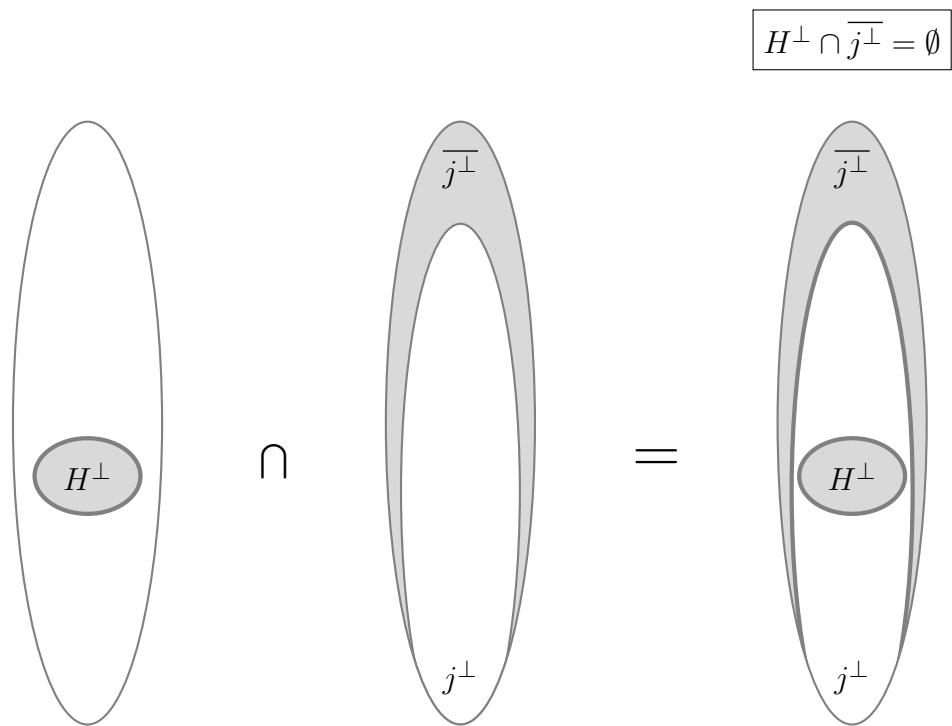


Figure 2.7: Counter-examples to $H \rightarrow j, |H^\perp| \geq |\overline{j^\perp}|$

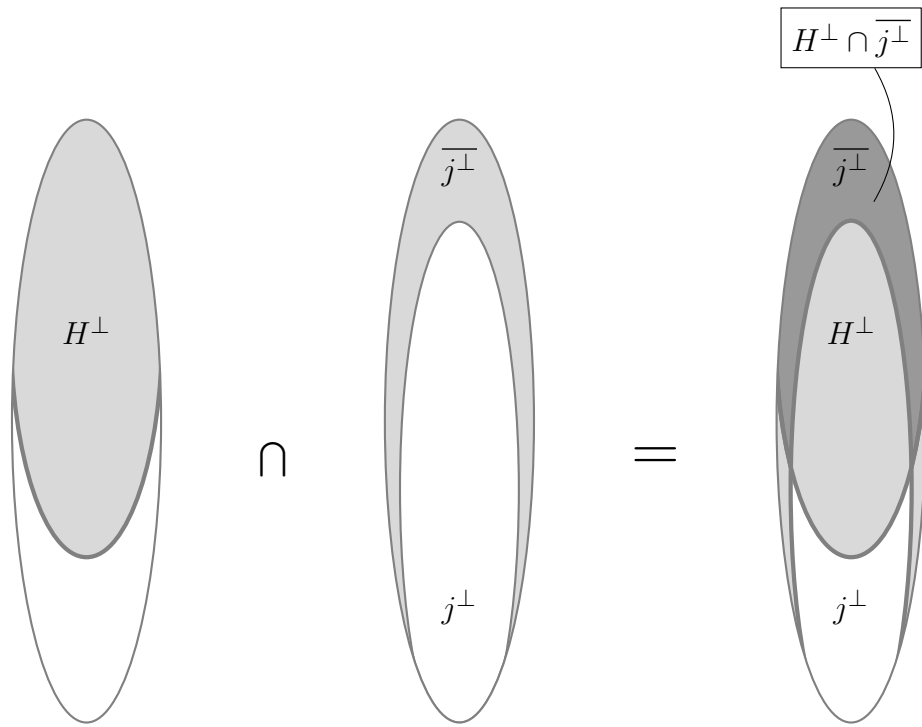


Figure 2.8: Counter-examples to $H \rightarrow j, |H^\perp| \geq |\overline{j^\perp}|$

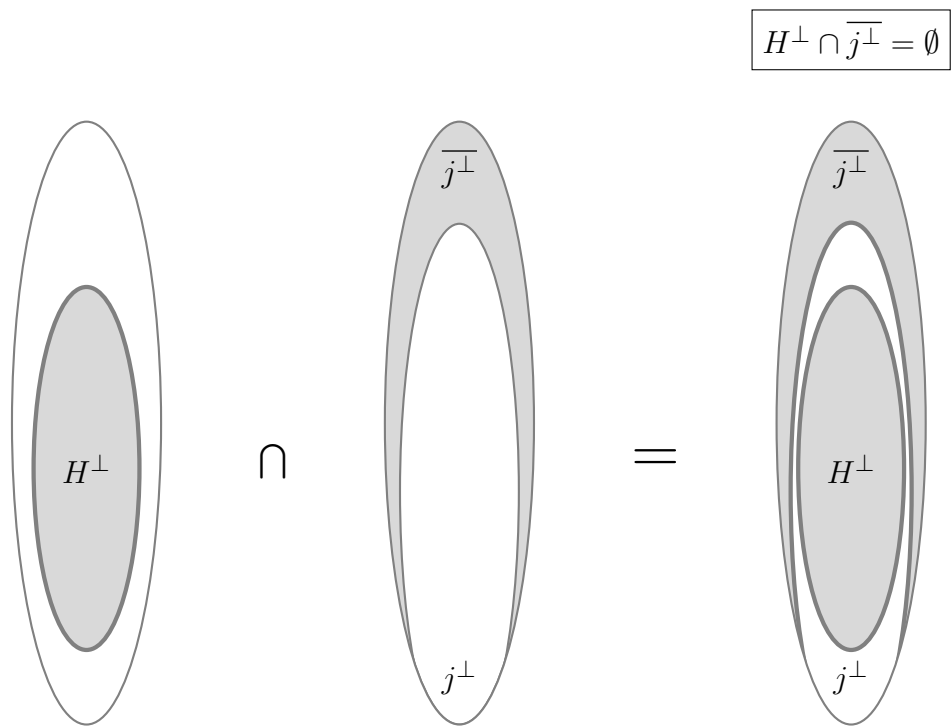


Figure 2.9: Counter-examples to $H \rightarrow j, |H^\perp| \geq |\overline{j^\perp}|$

Therefore, in the worst case the number of possible `correct_assignments` depends on $ar(j)$ exponentially. Hence, the efficiency of `violate_conclusion` significantly depends on the arity of the conclusion.

Next we present several examples of the executions of algorithms for finding counter-examples for $k = 2$. All the involved functions are presented in the appendix to the current chapter. The given time is the average for 5 runs.

Example 2.3.7. Implication $f_2^u, f_3^u \rightarrow f_8^b$.

`satisfy_premise`: f_{14}^b , 0.0022s.

`violate_conclusion`: f_{13}^b , 0.0016s.

The process of constructing counter-examples to this implication with both algorithms is presented in Example 2.3.5 and in Example 2.3.6. Neither unary functions in the premise nor binary function in the conclusion constrain the sets H^\perp and $\overline{j^\perp}$ significantly. Therefore, the comparable execution times of both algorithms is to be expected.

Example 2.3.8. Implication $f_1^u, f_2^u, f_3^u \rightarrow f_8^b$.

`satisfy_premise`: f_{232}^t , 0.0044s.

`violate_conclusion`: f_{212}^t , 0.0037s.

The difference to the previous example is that the only counter-examples are of arity 3.

Example 2.3.9. Implication $f_1^u, f_2^u, f_8^b \rightarrow f_0^u$.

`satisfy_premise`: no counter-examples, 0.0046s.

`violate_conclusion`: no counter-examples, 0.00036s.

No counter-examples exist. The input arity was 3.

The conclusion consists of a unary function, therefore, it is to be expected that $\overline{j^\perp}$ is small. Hence, `violate_conclusion` gives better performance.

Example 2.3.10. Implication $f_1^u, f_2^u, f_8^b \rightarrow f_{14}^b$.

`satisfy_premise`: no counter-examples, 0.0056s.

`violate_conclusion`: no counter-examples, 0.016s.

No counter-examples exist. The input arity was 3.

In this case the conclusion is not restrictive, however, it is easy to satisfy the premise, hence `satisfy_premise` performs better. It is worth noting that this result is specific for $k = 2$, because the set of all ternary functions on a domain of size 2 is small, therefore it is easy to iterate over them.

Example 2.3.11. Implication $f_2^u, f_{14}^b, f_{150}^t \rightarrow f_8^b$.

`satisfy_premise`: no counter-examples, 0.012s.

`violate_conclusion`: no counter-examples, 0.013s.

No counter-examples exist. The input arity was 3.

The conclusion is not restrictive. However, there is a ternary function in the premise that is difficult to satisfy. Hence, the performance of algorithms is comparable. This result is also specific for $k = 2$, because closing under commutation may take more time than simple iteration over all possible functions.

Example 2.3.12. Implication $f_2^u, f_8^b, f_{150}^t \rightarrow f_{232}^t$.

`satisfy_premise`: no counter-examples, 0.026s.

`violate_conclusion`: no counter-examples, 0.21s.

No counter-examples exist. The input arity was 3.

A ternary function in the conclusion prevents `violate_conclusion` from executing fast.

Next follow examples of functions on A_3 , these functions have an additional subscript “3, ”. These examples only serve the purpose of illustrating the difference in performance between `satisfy_premise` and `violate_conclusion` on A_3 .

Example 2.3.13. Implication $f_{3,15951}^b, f_{3,15663}^b \rightarrow f_{3,7625403765063}^t$.

`satisfy_premise`: no counter-examples, 4.7s.

`violate_conclusion`: no counter-examples, 19.6s.

No counter-examples exist. The input arity was 3.

A ternary function in the conclusion prevents `violate_conclusion` from executing fast.

Example 2.3.14. Implication $f_{3,8}^u \rightarrow f_{3,18}^u$.

`satisfy_premise`: no counter-examples, more than 4037s.

`violate_conclusion`: no counter-examples, 0.0072s.

No counter-examples exist. The input arity was 3.

The conclusion consists of a unary function, therefore it is expectable that $\overline{j^\perp}$ is small. Hence, `violate_conclusion` gives better performance.

2.4 Exploration of the Lattice of P-clones

In order to start the attribute exploration we generate some initial context. In an extreme case the initial context may contain all functions from U_k ; then no further exploration is necessary. However, practically it does not make sense as it is infeasible to produce such a context straight-forwardly. Therefore, we require that the initial context should be easy to generate. A good approach in this sense is to take all unary functions on a given domain A_k . There exist only k^k unary functions on A_k , therefore the context containing all unary functions is of modest size for a small k .

Example 2.4.1 (Attribute Exploration of Boolean P-clones). As the initial set of functions we take all unary Boolean functions, there exist exactly four.

Initial Step The initial context \mathbb{K}_{F_1} of all unary function and its implication basis.

	f_0^u	f_1^u	f_2^u	f_3^u
f_0^u	×		×	
f_1^u		×	×	
f_2^u	×	×	×	×
f_3^u			×	×

1. $\emptyset \rightarrow f_2^u$;
2. $f_2^u, f_3^u, f_1^u \rightarrow f_0^u$;
3. $f_2^u, f_3^u, f_0^u \rightarrow f_1^u$.

Chapter 2 Lattice of P-Clones

We proceed with examining implications until we find a counter-example².

$\emptyset \rightarrow f_2^u$: no counter-examples;

$f_2^u, f_3^u, f_1^u \rightarrow f_0^u$: no counter-examples;

$f_2^u, f_3^u, f_0^u \rightarrow f_1^u$: a counter-example found – f_8^b .

Second Step We add f_8^b to the context and obtain the context \mathbb{K}_{F_2} containing $f_0^u, f_1^u, f_2^u, f_3^u, f_8^b$.

	f_0^u	f_1^u	f_2^u	f_3^u	f_8^b
f_0^u	×		×		×
f_1^u		×	×		
f_2^u	×	×	×	×	×
f_3^u			×	×	×
f_8^b	×		×	×	×

Implications of the context \mathbb{K}_{F_2} :

$\emptyset \rightarrow f_2^u$: no counter-examples;

$f_2^u, f_3^u \rightarrow f_8^b$: a counter-example found – f_{14}^b .

Third Step We add f_{14}^b to the context and obtain the context \mathbb{K}_{F_3} containing $f_0^u, f_1^u, f_2^u, f_3^u, f_8^b, f_{14}^b$.

	f_0^u	f_1^u	f_2^u	f_3^u	f_8^b	f_{14}^b
f_0^u	×		×		×	×
f_1^u		×	×			
f_2^u	×	×	×	×	×	×
f_3^u			×	×	×	×
f_8^b	×		×	×	×	
f_{14}^b	×		×	×		×

Implications of the context \mathbb{K}_{F_3} :

²We use `NextClosure` to compute the basis and output implications one by one until we find the first counter-examples, see Section 1.4. Therefore, only a part of the basis is examined and presented in the lists below.

Chapter 2 Lattice of P-Clones

	f_0^u	f_1^u	f_2^u	f_3^u	f_8^b	f_{14}^b	f_{212}^t
f_0^u	×		×		×	×	×
f_1^u		×	×				×
f_2^u	×	×	×	×	×	×	×
f_3^u			×	×	×	×	×
f_8^b	×		×	×	×		
f_{14}^b	×		×	×		×	
f_{212}^t	×	×	×	×			

Figure 2.10: Context \mathbb{K}_{F_4} containing $f_0^u, f_1^u, f_2^u, f_3^u, f_8^b, f_{14}^b, f_{212}^t$

$\emptyset \rightarrow f_2^u$: no counter-examples;

$f_2^u, f_{14}^b, f_1^u \rightarrow f_3^u$: no counter-examples;

$f_2^u, f_{14}^b, f_1^u \rightarrow f_8^b$: no counter-examples;

$f_2^u, f_{14}^b, f_1^u \rightarrow f_0^u$: no counter-examples;

$f_2^u, f_8^b, f_2^u \rightarrow f_3^u$: no counter-examples;

$f_2^u, f_8^b, f_1^u \rightarrow f_0^u$: no counter-examples;

$f_2^u, f_8^b, f_1^u \rightarrow f_{14}^b$: no counter-examples;

$f_2^u, f_3^u, f_1^u \rightarrow f_8^b$: a counter-example found – f_{212}^t .

Fourth Step We add f_{212}^t to the context and obtain the context \mathbb{K}_{F_4} containing $f_0^u, f_1^u, f_2^u, f_3^u, f_8^b, f_{14}^b, f_{212}^t$.

Implications of the context \mathbb{K}_{F_4} :

$\emptyset \rightarrow f_2^u$: no counter-examples;

$f_2^u, f_{14}^b, f_{212}^t \rightarrow f_8^b$: no counter-examples;

$f_2^u, f_8^b, f_{212}^t \rightarrow f_{14}^b$: no counter-examples;

$f_2^u, f_8^b, f_{14}^b \rightarrow f_{212}^t$: no counter-examples;

$f_2^u, f_3^u, f_{212}^t \rightarrow f_8^b$: no counter-examples;

$f_2^u, f_3^u, f_{212}^t \rightarrow f_{14}^b$: no counter-examples;

$f_2^u, f_{14}^b, f_1^u \rightarrow f_3^u$: no counter-examples;

$f_2^u, f_{14}^b, f_1^u \rightarrow f_8^b$: no counter-examples;
 $f_2^u, f_{14}^b, f_1^u \rightarrow f_0^u$: no counter-examples;
 $f_2^u, f_{14}^b, f_1^u \rightarrow f_{212}^t$: no counter-examples;
 $f_2^u, f_8^b, f_1^u \rightarrow f_3^u$: no counter-examples;
 $f_2^u, f_8^b, f_1^u \rightarrow f_0^u$: no counter-examples;
 $f_2^u, f_8^b, f_1^u \rightarrow f_{212}^t$: no counter-examples;
 $f_2^u, f_8^b, f_1^u \rightarrow f_{14}^b$: no counter-examples;
 $f_2^u, f_3^u, f_1^u \rightarrow f_0^u$: no counter-examples;
 $f_2^u, f_0^u, f_{212}^t \rightarrow f_8^b$: no counter-examples;
 $f_2^u, f_0^u, f_{212}^t \rightarrow f_{14}^b$: no counter-examples;
 $f_2^u, f_3^u, f_0^u, f_{212}^t, f_{14}^b, f_8^b \rightarrow f_1^u$: no counter-examples;
 $f_2^u, f_0^u, f_1^u \rightarrow f_3^u$: no counter-examples.

At this point no counter-examples exist and the attribute exploration terminates. Not all classes of p-indecomposable functions were found, therefore, exploration is not successful.

The difference between the considered example and the usual settings for AE is the varying number of attributes. Indeed, if the set of attributes is fixed and all the irreducible objects are found, the implicative theory of data is guaranteed to be discovered. The maximal number of irreducible objects in the context is limited to $2^{|M|}$, where $|M|$ is the size of the set of attributes.

In order to find all the classes of p-indecomposable functions we need to take into account that the set of objects and the set of attributes grow simultaneously. We extend the procedure with an intermediate step of finding those functions that are not counter-examples to a valid implication of the current context, but alter the current lattice when added to the context. The simplest extension would allow to find such new functions that alter the current lattice on their own (that is we look for only one new function at a time). Note that in order for the new function not to be a counter-example to some valid implication its intent restricted to functions from the context should be equal to an existing intent from $\mathfrak{B}(F, F, \perp)$. Hence, the extension of AE finds a single new function altering the concept lattice when no counter-examples exist. We call this new procedure *object-attribute exploration*.

2.4.1 Object-Attribute Exploration

We now describe which commuting properties a new function $g \notin F$ should possess in order to alter the concept lattice of the original context $\mathbb{K} = (F, F, \perp)$ despite the fact that the intent of g is equal to an intent from $\mathfrak{B}(F, F, \perp_F)$.

To distinguish between binary relations on different sets of functions we use subscripts. The commutation relation on F is denoted by \perp_F , i.e. $\perp_F = \{(h, j) \in F^2 \mid h \perp j\}$. The context with the new function $(F \cup g, F \cup g, \perp_{F \cup g})$ is denoted by $\mathbb{K}_{F \cup g}$. The derivation operator for the context $\mathbb{K}_{F \cup g}$ is denoted by $(\cdot)^{\perp_{F \cup g}}$.

Proposition 2.4.2. *Let $C \in \mathfrak{B}(F, F, \perp)$ such that $\text{ext}(C) \not\subseteq \text{int}(C)$. Let $g \notin F$ be a function such that $g^{\perp_{F \cup g}} \cap F = \text{int}(C)$.*

g is irreducible in $\mathbb{K}_{F \cup g} \Leftrightarrow g \perp g$.

Proof. As $\text{ext}(C) \not\subseteq \text{int}(C)$ and for all $f \in F \setminus \text{int}(C) : g \not\perp f$ it follows that $g \not\perp \text{ext}(C)$. We prove the contrapositive statement: g is reducible in $\mathbb{K}_{F \cup g} \Leftrightarrow g \not\perp g$.

\Leftarrow As $g \not\perp g$ we have $g^{\perp_{F \cup g}} = \text{int}(C) = \text{ext}(C)^{\perp_{F \cup g}}$. Therefore, g is reducible.

\Rightarrow As g is reducible we obtain $g^{\perp_{F \cup g}} = H^{\perp_{F \cup g}}$ for some $H \subseteq F$. Fix this H . As $H^{\perp_{F \cup g}} = \text{int}(C)$ we have $H^{\perp_{F \cup g} \perp_{F \cup g}} = \text{ext}(C)$. Suppose $H \subseteq \text{int}(C)$, then $H^{\perp_{F \cup g} \perp_{F \cup g}} \subseteq \text{int}(C)^{\perp_{F \cup g} \perp_{F \cup g}} = \text{int}(C)$. As $H^{\perp_{F \cup g} \perp_{F \cup g}} = \text{ext}(C)$ and $\text{ext}(C) \not\subseteq \text{int}(C)$ we arrive at a contradiction. Therefore, $H \not\subseteq \text{int}(C)$. Hence, $g \not\perp H$, therefore, $g \notin H^{\perp_{F \cup g}}$, hence, $g \notin g^{\perp_{F \cup g}}$. □

Corollary 2.4.3. *If g is irreducible in $\mathbb{K}_{F \cup g}$ and $g \not\perp g$ then $\text{ext}(C) \rightarrow g$ holds in $\mathbb{K}_{F \cup g}$.*

Proof. As $g^{\perp_{F \cup g}} = \text{int}(C) \cup \{g\}$ and $\text{ext}(C)^{\perp_{F \cup g}} = \text{int}(C)$ we have $\text{ext}(C)^{\perp_{F \cup g}} \subset g^{\perp_{F \cup g}}$, therefore, $\text{ext}(C) \rightarrow g$. □

Example 2.4.4. The lattice of the context after the fourth step is presented in Figure 2.11. All the concepts are labeled. The labels starting from C are just names. The labels containing a function f denote the object concept generated by f if the label is below the concept and the attribute concept if the label is above the concept. The concept $((f_0^u)^{\perp_{F_4}}, (f_0^u)^{\perp_{F_4} \perp_{F_4}})$ is additionally labeled by C_* .

The function f_6^b commutes only with the functions f_2^u, f_0^u from the context \mathbb{K}_{F_4} in Figure 2.10. Therefore, $(f_6^b)^{\perp_{F_4}} = \text{int}(C_*)$. Moreover, $\text{ext}(C_*) =$

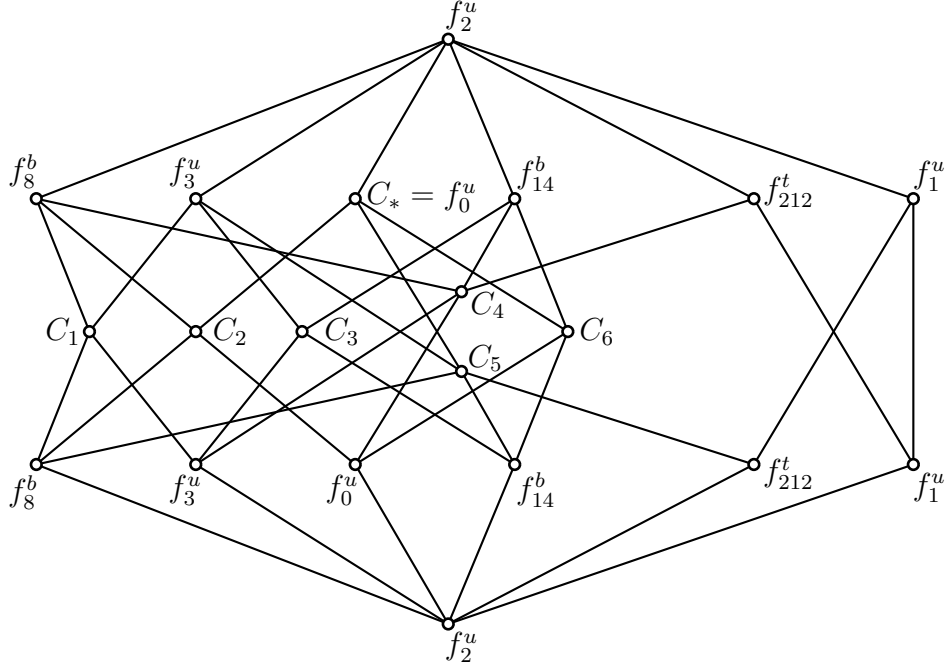


Figure 2.11: Concept lattice of the context \mathbb{K}_{F_4} from Figure 2.10

$\{f_2^u, f_0^u, f_8^b, f_{14}^b, f_{212}^t\} \not\subseteq \{f_2^u, f_0^u\} = \text{int}(C_*)$ and $f_6^b \perp f_6^b$, see Figure 2.11. As follows from Proposition 2.4.2, f_6^b is irreducible in $\mathbb{K}_{F_4 \cup f_6^b}$.

For the lattice of the context after adding f_6^b see Item 3 in Section 2.7.

The statement dual to Proposition 2.4.2 holds as well.

Proposition 2.4.5. *Let $C \in \mathfrak{B}(F, F, \perp_F)$ such that $\text{ext}(C) \subseteq \text{int}(C)$. Let $g \in U_k, g \notin F$ be a function such that $g^{\perp_{F \cup g}} \cap F = \text{int}(C)$.*

g is irreducible in $\mathbb{K}_{F \cup g} \Leftrightarrow g \not\perp g$.

Proof. As $\text{ext}(C) \subseteq \text{int}(C)$ and $g \perp \text{int}(C)$ then $g \perp \text{ext}(C)$. We prove the contrapositive statement: g is reducible in $\mathbb{K}_{F \cup g} \Leftrightarrow g \perp g$.

\Leftarrow As $g \perp g$ and $g \perp \text{ext}(C)$ we have $\text{ext}(C)^{\perp_{F \cup g}} = \text{int}(C) \cup \{g\} = g^{\perp_{F \cup g}}$. Hence, g is reducible.

\Rightarrow As g is reducible we obtain $g^{\perp_{F \cup g}} = H^{\perp_{F \cup g}}$ for some $H \subseteq F$. Fix this H . As $g \perp \text{int}(C)$ we have $H \perp \text{int}(C)$, hence, $H \subseteq \text{ext}(C)$. As $g \perp \text{ext}(C)$ we have $g \perp H$, hence, $g \in H^{\perp_{F \cup g}}$, therefore, $g \in g^{\perp_{F \cup g}}$ and $g \perp g$.

□

Chapter 2 Lattice of P-Clones

	$f_{3,0}^u$	$f_{3,1}^u$	$f_{3,12015}^b$
$f_{3,0}^u$	×		×
$f_{3,1}^u$		×	×
$f_{3,12015}^b$	×	×	

Figure 2.12: Context $\mathbb{K}_0^{(3)}$ of function on domain A_3 containing $f_{3,0}^u, f_{3,1}^u, f_{3,12015}^b$

Corollary 2.4.6. *If g is irreducible in $\mathbb{K}_{F \cup g}$ and $g \perp g$ then $g \rightarrow \text{ext}(C)$ holds in $\mathbb{K}_{F \cup g}$.*

Proof. As $g^{\perp_{F \cup g}} = \text{int}(C)$ and $\text{ext}(C)^{\perp_{F \cup g}} = \text{int}(C) \cup \{g\}$ we have $g^{\perp_{F \cup g}} \subset \text{ext}(C)^{\perp_{F \cup g}}$, therefore, $g \rightarrow \text{ext}(C)$. \square

In order to distinguish reducibility in the old context \mathbb{K}_F and in the new context $\mathbb{K}_{F \cup g}$ we introduce a new notion.

Definition 2.4.7. A function g is *first-order irreducible for \mathbb{K}_F* if it is reducible in \mathbb{K}_F , but irreducible in $\mathbb{K}_{F \cup g}$. A function g is *first-order reducible for \mathbb{K}_F* if it is reducible for \mathbb{K}_F and reducible in $\mathbb{K}_{F \cup g}$.

Remember that we call g plainly irreducible if it is irreducible in $(F \cup g, F, \perp_F \cup \{(g, f) \mid f \in F, f \perp g\})$. Hence, if function is first-order reducible for \mathbb{K}_F then it is also plainly reducible in \mathbb{K}_F . Note that g is plainly irreducible in \mathbb{K}_F iff g is a counter-example to some implication valid in \mathbb{K}_F .

Next we present an example with functions from U_3 . As before, we add 3 in the subscript of every function.

Example 2.4.8. The context under consideration $\mathbb{K}_0^{(3)}$ is presented in Figure 2.12, the lattice of the context $\mathbb{K}_0^{(3)}$ is presented in Figure 2.13. The implication basis of $\mathbb{K}_0^{(3)}$ is empty, therefore, there exist no plainly irreducible functions. The function $f_{3,756}^b$ has the following commuting properties: $f_{3,756}^b \perp \{f_{3,0}^u, f_{3,12015}^b\}$ and $f_{3,756}^b \not\perp f_{3,1}^u$. Moreover, $f_{3,756}^b \not\perp f_{3,756}^b$ and for the corresponding concept C holds $\text{ext}(C) = \{f_{3,0}^u\} \subset \{f_{3,0}^u, f_{3,12015}^b\} = \text{int}(C)$. As follows from Proposition 2.4.5, the function $f_{3,756}^b$ is first-order irreducible for $\mathbb{K}_0^{(3)}$.

Corollary 2.4.9. *Let $C \in \mathfrak{B}(F, F, \perp_F)$, $g \in U_k, g \notin F$, and g be first-order reducible for \mathbb{K}_F .*

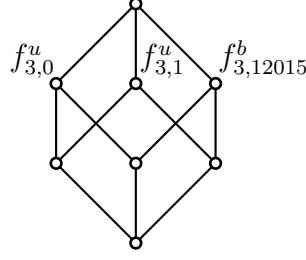


Figure 2.13: Concept lattice of the context $\mathbb{K}_0^{(3)}$ from Figure 2.12

$$\text{ext}(C) \perp g \quad \Leftrightarrow \quad g \perp g.$$

Proof. Follows from Propositions 2.4.2 and 2.4.5 and the fact that $\text{ext}(C) \perp g \Leftrightarrow \text{ext}(C) \subseteq \text{int}(C)$. \square

There remains a possibility that a union of sets of reducible functions is irreducible. We proceed with the simplest case when there are only two sets each containing a single first-order reducible function for the current context. We prove several propositions about such pairs of first-order reducible functions. The consequences of these propositions and several examples are investigated more deeply in Section 2.4.2.

We consider a context \mathbb{K}_F and new functions $g_1, g_2 \in U_k$, $g_1, g_2 \notin F$. We denote $\{g_1, g_2\}$ by G . As in the case with one function, for $i \in \{1, 2\}$: g_i is not a counterexamples to a valid implication iff $g_i^{\perp_{F \cup G}} \cap F \in \text{Int}(F, F, \perp_F)$. We denote the corresponding intents by $\text{int}(C_1)$ and $\text{int}(C_2)$, meaning that $g_i^{\perp_{F \cup G}} = \text{int}(C_i)$ for $i \in \{1, 2\}$.

Proposition 2.4.10. *Let $C_1, C_2 \in \mathfrak{B}(F, F, \perp_F)$ and $g_1, g_2 \notin F$ be first-order reducible for \mathbb{K}_F . Suppose $g_1 \perp g_2$.*

Both g_1, g_2 are irreducible in $\mathbb{K}_{F \cup G} \Leftrightarrow \text{ext}(C_1) \not\subseteq \text{int}(C_2)$.

Proof. As g_1 is irreducible it holds that $g_1^{\perp_{F \cup G}} \neq \text{ext}(C_1)^{\perp_{F \cup G}}$. From Corollary 2.4.9 follows that $g_1 \in \text{ext}(C_1)^{\perp_{F \cup G}}$ iff $g_1 \in g_1^{\perp_{F \cup G}}$. Therefore, $\text{ext}(C_1)^{\perp_{F \cup G}} = g_1^{\perp_{F \cup G}} \setminus \{g_2\}$. Hence, $\text{ext}(C_1) \not\subseteq g_2$, hence, $\text{ext}(C_1) \not\subseteq \text{int}(C_2)$. Similarly for g_2 , $\text{ext}(C_2) \not\subseteq \text{int}(C_1)$. \square

Chapter 2 Lattice of P-Clones

	g_1	g_2
g_1		\times
g_2	\times	

	g_1	g_2
g_1	\times	\times
g_2	\times	

	g_1	g_2
g_1	\times	\times
g_2	\times	\times

Figure 2.14: Possible commutations of g_1 and g_2 from Proposition 2.4.10

	g_1	g_2
g_1		
g_2		

	g_1	g_2
g_1	\times	
g_2		

	g_1	g_2
g_1	\times	
g_2		\times

Figure 2.15: Possible commutations of g_1 and g_2 from Proposition 2.4.11

The possible commutation properties of g_1 and g_2 corresponding to Proposition 2.4.10 are presented in Figure 2.14.

Proposition 2.4.11. *Let $C_1, C_2 \in \mathfrak{B}(F, F, \perp_F)$ and $g_1, g_2 \notin F$ be first-order reducible for \mathbb{K}_F . Suppose $g_1 \not\perp g_2$.*

Both g_1, g_2 are irreducible in $\mathbb{K}_{F \cup G} \iff \text{ext}(C_1) \subseteq \text{int}(C_2)$.

Proof. As g_1 is irreducible it holds that $g_1^{\perp_{F \cup G}} \neq \text{ext}(C_1)^{\perp_{F \cup G}}$. From Corollary 2.4.9 follows that $g_1 \in \text{ext}(C_1)^{\perp_{F \cup G}}$ iff $g_1 \in g_1^{\perp_{F \cup G}}$. Therefore, $\text{ext}(C_1)^{\perp_{F \cup G}} = g_1^{\perp_{F \cup G}} \cup \{g_2\}$. Hence, $\text{ext}(C_1) \perp g_2$, hence, $\text{ext}(C_1) \subseteq \text{int}(C_2)$. By the properties of derivation operators, $\text{ext}(C_2) \subseteq \text{int}(C_1)$. \square

The possible commutation properties of g_1 and g_2 corresponding to Proposition 2.4.11 are presented in Figure 2.15.

The functions mentioned in Propositions 2.4.11 and 2.4.10 can be called *second-order irreducible for \mathbb{K}_F* . In the next proposition we show that it is not necessary to look for three functions at once in order to find all p-indecomposable functions. Therefore, we do not need to define third-order irreducibility.

Here we use the notation: for $I \subseteq \{1, 2, 3\}$: $L_I = \{g_i \mid i \in I\}$. We omit the curly brackets in I , i.e. $L_{\{1,2\}} = L_{12} = \{g_1, g_2\}$.

Proposition 2.4.12. *Let $G = \{g_1, g_2, g_3\}$ be a set of functions such that $G \cap F = \emptyset$ and for $i \in \{1, 2, 3\}$: $g_i^{\perp_{F \cup G}} \cap F = \text{int}(C_i)$. If not all functions from G are reducible*

Chapter 2 Lattice of P-Clones

in $\mathbb{K}_{F \cup G}$ then there exists $L \subset G$ such that not all functions from L are reducible in $\mathbb{K}_{F \cup L}$.

Proof. Let g_1 be reducible in $\mathbb{K}_{F \cup L_{12}}$ and in $\mathbb{K}_{F \cup L_{13}}$. Then there exists $H \subseteq F \cup \{g_2\} : H^{\perp_{F \cup L_{12}}} = g_1^{\perp_{F \cup L_{12}}}$ and $J \subseteq F \cup \{g_3\} : J^{\perp_{F \cup L_{13}}} = g_1^{\perp_{F \cup L_{13}}}$. Fix these H and J . If either g_2 is irreducible in $\mathbb{K}_{F \cup L_2}$ or g_3 is irreducible in $\mathbb{K}_{F \cup L_3}$ then the proposition is proved. Therefore, we can assume that they are reducible in corresponding context. Hence, without loss of generality, we can assume that $H, J \subseteq F$ (i.e. $H \cap G = J \cap G = \emptyset$). Note that

$$g_1^{\perp_{F \cup G}} = g_1^{\perp_{F \cup L_{13}}} \cup g_1^{\perp_{F \cup L_{12}}} = J^{\perp_{F \cup L_{13}}} \cup H^{\perp_{F \cup L_{12}}}. \quad (2.8)$$

Let $g_3 \in H^{\perp_{F \cup G}}$. Then $g_3 \perp H$. As $g_3^{\perp_{F \cup G}} \cap F = \text{int}(C_3)$ we obtain $H \subseteq \text{int}(C_3)$. Moreover, as $\text{int}(C_3)$ is an intent in \mathbb{K}_F we have $H^{\perp_{F \cup L_2}} \subseteq \text{int}(C_3)$. As $g_1^{\perp_{F \cup G}} \cap F = H^{\perp_{F \cup L_2}} = J^{\perp_{F \cup L_3}} = \text{int}(C_1)$ we have $J^{\perp_{F \cup L_3}} \subseteq \text{int}(C_3)$ and, by properties of closure operators, $J \subseteq \text{int}(C_3)$. Therefore, $g_3 \perp J$ and $g_3 \in J^{\perp_{F \cup G}}$. Similarly, if $g_2 \in J^{\perp_{F \cup G}}$ then $g_2 \in H^{\perp_{F \cup G}}$. Hence,

$$H^{\perp_{F \cup L_{12}}} \cup J^{\perp_{F \cup L_{13}}} = H^{\perp_{F \cup G}} \cup J^{\perp_{F \cup G}}. \quad (2.9)$$

Combining (2.8) and (2.9) we obtain $g_1^{\perp_{F \cup G}} = H^{\perp_{F \cup G}} \cup J^{\perp_{F \cup G}}$. Therefore, $g_1^{\perp_{F \cup G}} = (H \cap J)^{\perp_{F \cup G}}$. Hence, g_1 is reducible in $\mathbb{K}_{F \cup G}$ and we arrive at a contradiction with initial assumption.

Therefore, if g_1, g_2 are in $\mathbb{K}_{F \cup L_{12}}$ then at least g_1 is irreducible in $\mathbb{K}_{F \cup L_{13}}$. If g_3 is reducible in $\mathbb{K}_{F \cup L_{13}}$ then g_1 is reducible in $\mathbb{K}_{F \cup L_1}$. Otherwise, both g_1, g_3 are irreducible in $\mathbb{K}_{F \cup L_{13}}$. \square

Suppose that a context \mathbb{K}_F contains all p-indecomposable functions, however, the task is to prove this fact, i.e. that no further p-indecomposable functions exist. Suppose it has been checked that no counter-examples exist and every single function g is first-order reducible for \mathbb{K}_F . According to the above propositions it is necessary to look for exactly two functions at once in order to prove the desired statement. Therefore, in order to complete the proof for every $C_1, C_2 \in \mathfrak{B}(\mathbb{K}_F)$ one has to find all the functions g_1, g_2 such that $g_1^{\perp_{F \cup g_1}} \cap F = \text{int}(C_1)$ and $g_2^{\perp_{F \cup g_2}} \cap F = \text{int}(C_2)$ and then check if g_1 commutes with g_2 . Therefore one has to check the commutation property between all functions (if the context indeed contains all p-indecomposable functions). As already discussed, this task is infeasible. This result is discouraging. However, having the knowledge about the final result in some cases we can guarantee that all p-indecomposable functions will be found even without looking for two functions at once.

2.4.2 Implicatively Closed Subcontexts

As Example 2.4.1 suggests, during the exploration of p-clones one can discover such a subcontext of functions that no further function is a counter-example to existing implications. We shall say that such a subcontext is *implicatively closed*, meaning that all implications valid in this subcontext are valid in the final context as well. Analysis of similar constructions can be found in [Gan07], however, the authors are interested in partitions where both parts are implicatively closed and in relations between these parts. Here the complementary subcontext is not necessarily implicatively closed.

In order to guarantee the discovery of all p-indecomposable functions (success of exploration) it suffices to find a subcontext such that it is neither implicatively closed nor contained in any other implicatively closed subcontext. Suppose the context $\mathbb{K}_F = (F, F, \perp_F)$, $F \subseteq U_k$ is discovered. As earlier, we denote the context of all p-indecomposable functions on U_k by $\mathbb{K}_{F_k^p}$. Let $S = F_k^p \setminus F$. It would be desirable to be able to guarantee the discovery of functions S by considering only the discovered part of relation \perp_F and the part \perp_{FS} ($= \perp_{SF}^{-1}$), see Figure 2.16. Unfortunately, as the next example shows, in general it is not possible.

	F	S
F	\perp_F	\perp_{FS}
S	\perp_{SF}	\perp_S

Figure 2.16: Partitioning of the context $\mathbb{K}_{F_k^p}$ of all p-indecomposable functions

Remark. The function f_2^u is the identity function. The presence of this function in the context does not modify the concept lattice, therefore we may not include it in the context. In what follows this function is not included in the context anymore in order to make the figures more compact.

Example 2.4.13. Consider the context in Figure 2.17. The context contains all p-indecomposable functions from U_2 and three additional objects g_1, g_2, g_3 . Functions having commutation properties as g_1, g_2, g_3 do not exist. However, if functions with these commutation properties existed then the functions g_1, g_2 would not be counter-examples to any implication valid in $\mathbb{K}_{F_2^p \cup g_3}$. Note that g_3 is a counter-example to an implication valid in $\mathbb{K}_{F_2^p}$. Therefore, the subcontext containing functions $F_2^p \cup g_3$ would be implicatively closed. Moreover, it is even closed with respect to finding first-order irreducible functions as g_1 is reducible in $\mathbb{K}_{F_2^p \cup \{g_1, g_3\}}$ and g_2 is reducible in $\mathbb{K}_{F_2^p \cup \{g_2, g_3\}}$.

	f_0^u	f_1^u	f_{14}^b	f_8^b	f_{212}^t	f_{150}^t	f_3^u	g_3	g_1	g_2
f_0^u	×		×	×	×	×		×	×	
f_1^u		×			×	×				
f_{14}^b	×		×				×	×		
f_8^b	×			×			×			×
f_{212}^t	×	×					×	×		
f_{150}^t	×	×				×	×			
f_3^u			×	×	×	×	×	×	×	×
g_3	×		×		×		×			
g_1	×						×			×
g_2				×			×		×	×

Figure 2.17: Context $\mathbb{K}_{F_2^p \cup \{g_1, g_2, g_3\}}$ from Example 2.4.13

However, if instead of g_3 we consider the function g_4 , which differs from g_3 only in that g_4 commutes with both g_1 and g_2 (see Figure 2.18), then the subcontext containing $F_2^p \cup g_4$ is neither implicatively closed nor contained in any implicatively closed subcontext of the context $\mathbb{K}_{F_2^p \cup \{g_1, g_2, g_4\}}$. The difference between g_3 and g_4 is contained in \perp_S in Figure 2.16. Therefore, in general it is not possible to guarantee the discovery of functions S without considering \perp_S .

Even if all functions from S are counter-examples to implications valid in $\mathbb{K}_{F_k^p}$ one cannot guarantee that all functions S will eventually be discovered by means of object-attribute exploration. We presented a corresponding example.

Example 2.4.14. In the context $\mathbb{K}_{\{f_0^u, f_1^u, f_{212}^t, f_3^u, g_1, g_2, g_3\}}$ in Figure 2.19 the objects g_1, g_2, g_3 are counter-examples to the implication $\{f_0^u, f_1^u\} \rightarrow f_3^u$ valid in

Chapter 2 Lattice of P-Clones

	f_0^u	f_1^u	f_{14}^b	f_8^b	f_{212}^t	f_{150}^t	f_3^u	g_4	g_1	g_2
f_0^u	×		×	×	×	×		×	×	
f_1^u		×			×	×				
f_{14}^b	×		×				×	×		
f_8^b	×			×			×			×
f_{212}^t	×	×					×	×		
f_{150}^t	×	×				×	×			
f_3^u			×	×	×	×	×	×	×	×
g_4	×		×		×		×		×	×
g_1	×						×	×		×
g_2				×			×	×	×	×

Figure 2.18: Context $\mathbb{K}_{F_2^p \cup \{g_1, g_2, g_4\}}$ from Example 2.4.13

$\mathbb{K}_{\{f_0^u, f_1^u, f_{212}^t, f_3^u\}}$. However, the subcontext $\mathbb{K}_{\{f_0^u, f_1^u, f_{212}^t, f_3^u, g_3\}}$ is implicatively closed and g_1, g_2 are first-order reducible for it.

	f_0^u	f_1^u	f_{212}^t	f_3^u	g_3	g_1	g_2
f_0^u	×		×		×	×	×
f_1^u		×	×		×	×	×
f_{212}^t	×	×		×			
f_3^u			×	×			
g_3	×	×			×		
g_1	×	×					×
g_2	×	×				×	

Figure 2.19: Context $\mathbb{K}_{\{f_0^u, f_1^u, f_{212}^t, f_3^u, g_1, g_2, g_3\}}$ from Example 2.4.14

Definition 2.4.15. Let \mathbb{K}_H be a context, $\mathbb{K}_F \subseteq \mathbb{K}_H$, $S = H \setminus F$. An object $s \in S$ is called an *essential counter-example* for \mathbb{K}_F if there exists an implication Imp valid in \mathbb{K}_F such that

1. s is a counter-example to Imp ;
2. there does not exist an object $p \in S \setminus \{s\}$ such that p is a counter-example to Imp .

It is clear that all the essential counter-examples will necessarily be added to the context during the exploration. The next proposition suggests how one can check if a counter-example is essential or not.

Proposition 2.4.16. *A function s is an essential counter-example for \mathbb{K}_F iff there exists $h \in (s^{\perp U_k} \cap F)^{\perp F \perp F} \setminus (s^{\perp U_k} \cap F)$ such that for all $p \in S \setminus \{s\}$ such that $(s^{\perp U_k} \cap F) \subseteq (p^{\perp U_k} \cap F)$ we have $h \in (p^{\perp U_k} \cap F)$.*

Proof. We start from the latter part. As $h \in (s^{\perp U_k} \cap F)^{\perp F \perp F} \setminus (s^{\perp U_k} \cap F)$ the implication $(s^{\perp U_k} \cap F) \rightarrow h$ is valid in \mathbb{K}_F . Moreover, for any $p \in S \setminus \{s\}$ if $(s^{\perp U_k} \cap F) \subseteq (p^{\perp U_k} \cap F)$ then $g \in (p^{\perp U_k} \cap F)$, hence, p is not a counter-example to the implication. Therefore, s is an essential counter-example.

On the other hand, if s is an essential counter-example then there exists such an implication $A \rightarrow h$ that $A \subseteq (s^{\perp U_k} \cap F)$ and $h \notin s^{\perp U_k} \cap F$. However, as the implication is valid in \mathbb{K}_F we have $h \in (s^{\perp U_k} \cap F)^{\perp F \perp F} \setminus (s^{\perp U_k} \cap F)$. Moreover, as s is essential we have for any $p \in S \setminus \{s\}$ such that $(s^{\perp U_k} \cap F) \subseteq (p^{\perp U_k} \cap F)$: $h \in (p^{\perp U_k} \cap F)$, because otherwise p would be a counter-example to the implication. \square

Therefore one can guarantee that the essential counter-examples will be discovered while considering only \perp_{FS} from Figure 2.16.

In the context $\mathbb{K}_{F_3^p}$ there are several pairs of functions (f_1, f_2) such that they commute with the same functions except that one commutes with itself and the other does not commute with itself. These functions cannot be essential counter-examples, because they are counter-examples to the same implications, if any. However, if they are the only counter-examples to some valid implication then these functions will eventually be discovered by object-attribute exploration.

Proposition 2.4.17. *Let $s_1, s_2 \in S$ such that $s_2 \not\leq s_1$ and $s_1^{\perp U_k} = s_2^{\perp U_k} \cup \{s_2\}$. If there exists a valid in \mathbb{K}_F implication Imp such that the counter-examples are exactly $s_1, s_2 \in S$ then s_1 is first-order irreducible for $\mathbb{K}_{F \cup s_2}$ and s_2 is first-order irreducible for $\mathbb{K}_{F \cup s_1}$.*

Proof. **s_1 in $\mathbb{K}_{F \cup s_2}$.** As Imp is valid in \mathbb{K}_F the set $s_2^{\perp F \cup s_1}$ is closed in \mathbb{K}_F . Therefore, as follows from Proposition 2.4.2 for the object concept of s_2 ($\text{ext}(C_{s_2}) \not\subseteq \text{int}(C_{s_2})$), the function s_1 ($s_1 \perp s_1$) is first-order irreducible.

s_2 in $\mathbb{K}_{F \cup s_1}$. As Imp is valid in \mathbb{K}_F the set $s_1^{\perp F \cup s_2}$ is closed in \mathbb{K}_F . Therefore, as follows from Proposition 2.4.5 for the object concept of s_1 ($\text{ext}(C_{s_1}) \subseteq \text{int}(C_{s_1})$), the function s_2 ($s_2 \not\leq s_1$) is first-order irreducible.

□

The next proposition states that the discovery of p-decomposable (reducible in final context) functions does not have an impact on the overall success of the procedure.

Proposition 2.4.18. *Consider \mathbb{K}_F . Let R denote all the functions that are not p-indecomposable. Let $H = F_k^p \setminus F$. If $h \in H$ is first-order irreducible for \mathbb{K}_F then h is first-order irreducible for $\mathbb{K}_{F \cup R}$.*

Proof. Note that for all $r \in R : r^{\perp_{U_k}} = \bigcap \{f^{\perp_{U_k}} \mid f \in J\} = J^{\perp_{U_k}}$ for some $J \subseteq F_k^p$.

The function h is first-order irreducible in \mathbb{K}_F , hence, $h^{\perp_{U_k}} = \text{int}(C_F)$ for some $C_F \in \mathfrak{B}(F, F, \perp_F)$ and from Corollary 2.4.9 we have $\text{ext}(C_F) \perp h \Leftrightarrow h \perp h$. In $\mathbb{K}_{F \cup R}$ we decompose the intent of the corresponding concept C in two parts: $\text{int}(C) = \text{int}(C_F) \cup \text{int}(C_R)$.

1. Suppose $\text{ext}(C_F) \not\perp h$. Hence, $\text{ext}(C) \not\perp h$. Moreover, as h is first-order irreducible for \mathbb{K}_F we have $h \perp h$. Therefore, h is first-order irreducible for $\mathbb{K}_{F \cup R}$ as well.
2. Suppose $\text{ext}(C_F) \perp h$. Hence, either there exists a counter example to some valid in $\mathbb{K}_{F \cup R}$ implication or $\text{ext}(C) \perp h$. If no counter-example exists then $\text{ext}(C_R) \perp h$. Moreover, as h is first-order irreducible for \mathbb{K}_F we have $h \not\perp h$. Therefore, h is first-order irreducible for $\mathbb{K}_{F \cup R}$ as well.

□

Finding all implicatively closed partitions of a context is a complex task. The following proposition allows for speeding up the process.

Proposition 2.4.19. *Let \mathbb{K}_H be a context, $\mathbb{K}_F \subseteq \mathbb{K}_H$. Let Imp be a valid in \mathbb{K}_F implication such that $J \subseteq H \setminus F$ is the set of all counterexamples to Imp . Each context \mathbb{K}_L such that $\mathbb{K}_F \subseteq \mathbb{K}_L \subseteq \mathbb{K}_H$ and $L \cap J = \emptyset$ is not implicatively closed.*

Proof. The implication Imp is valid in \mathbb{K}_L . However, J contains counter-examples to this implication, therefore, \mathbb{K}_L is not implicatively closed. □

We have investigated different types of reducibilities and we have shown that there is no need to define third-order irreducible functions. However, the task of finding second-order irreducible functions is infeasible. Fortunately, it is not only feasible to find plainly irreducible functions, but also first-order irreducible functions. Moreover, if it would be possible to prove that the functions undiscovered so far are

	f_{150}^t	f_{14}^b	f_8^b	f_0^u	f_1^u	f_{212}^t	f_3^u
f_{150}^t	×			×	×		×
f_{14}^b		×		×			×
f_8^b			×	×			×
f_0^u	×	×	×	×		×	
f_1^u	×				×	×	
f_{212}^t				×	×		×
f_3^u	×	×	×			×	×

Figure 2.20: Functions $f_0^u, f_1^u, f_{212}^t, f_3^u$ are first-order reducible for $\mathbb{K}_{\{f_{150}^t, f_{14}^b, f_8^b\}}$.

not second-order irreducible then we can guarantee that all the p-indecomposable functions will eventually be discovered.

Example 2.4.20 (Partition of the context of Boolean p-indecomposable functions.). After an investigation of Boolean p-indecomposable functions we are able to find an implicatively closed subcontext $\mathbb{K}_{\{f_{150}^t, f_{14}^b, f_8^b\}}$ such that all the undiscovered p-indecomposable functions are second-order irreducible, Figure 2.20.

However, if we start the exploration of Boolean functions from all Boolean unary functions then we will eventually discover all Boolean p-indecomposable functions.

We use the same idea for discovering all p-indecomposable functions on A_3 . Namely, we take all unary functions as the starting point. Thanks to earlier investigation in [Dan77] we know the final context. When we investigate all possible implicatively closed partitions such that the implicatively closed subcontext contains all unary functions we find the following:

- We start with 27 unary functions, 26 of them are p-indecomposable;
- After adding all essential counter-examples we obtain 147 functions, Proposition 2.4.16 is used;
- After using Proposition 2.4.17 we obtain 155 functions;
- There remain 42 functions to be discovered. By direct check we find that there does not exist an implicatively closed subcontext containing the 155 functions mentioned above such that all the undiscovered functions are second-order irreducible.

Hence, if we start from all unary functions on A_3 all the functions F_3^p will eventually

be discovered.

2.5 Results

In the previous sections the exploration process is described and investigated. As the result of this investigation we obtain that for the execution of exploration of p-indecomposable functions it is necessary to be able to

1. Find counter-examples to the implications of the form $H \rightarrow j$, $F \subseteq U_k, j \in U_k$;
2. For a context \mathbb{K}_F find a new function $h \in U_k$ such that $h^{\perp_{U_k}} \cap F = \text{int}(C)$, $C \in \mathfrak{B}(F, F, \perp_F)$ and either if $\text{ext}(C) \subseteq \text{int}(C)$ then $h \not\perp h$ or if $\text{ext}(C) \not\subseteq \text{int}(C)$ then $h \perp h$.

Note that it is possible to rewrite the second task in an implicative manner. Let $\mathbb{K}_F = (F, F, \perp_F)$, $C \in \mathfrak{B}(F, F, \perp_F)$. We use Boolean disjunction \vee . We extended the class of considered implications by allowing disjunctions in conclusion. For a context $\mathbb{K} = (G, M, I)$ for $A, B \subseteq M$ we say that an implication $A \rightarrow \bigvee B$ is *not respected/violated* by an object $x \in G$ iff the implication evaluates to false, i.e. g has all the attributes A and **none** of the attributes B . We also say that such x is a *counter-example* to the implication $A \rightarrow \bigvee B$. Hence, x is a counter-example iff $A \subseteq x'$ and $B \cap x' = \emptyset$. Otherwise, x *respects* the implication $A \rightarrow \bigvee B$. If we want an object x to be reducible in \mathbb{K} and have the same attributes as in the intent of a concept C then we look for a counter-example to the implication $\text{int}(C) \rightarrow \bigvee (M \setminus \text{int}(C))$. Indeed, x is a counter-example if it has attributes $\text{int}(C)$ and does not have all other attributes. Therefore, we rewrite the task of finding first-order irreducible functions in implicative manner with a disjunction in conclusion. A simple modification of the algorithm `satisfy_premise` creates an iterator over all counter-examples to $\text{int}(C) \rightarrow \bigvee (M \setminus \text{int}(C))$. Afterwards we check if there is a function in iterator that commutes/not commutes with itself.

Using all the methods and algorithms presented in this chapter it was possible to find all p-indecomposable functions on A_3 . Here we note that the two algorithms `satisfy_premise` and `violate_conclusion` were launched for every single implication in parallel and the process waited for the first algorithm to finish. As already mentioned after finding the first counter-example the further implications from the basis were not analyzed; the found counter-example was added to the context and the basis was recomputed.

The experiment was conducted three times starting from different initial contexts, all three times the exploration was successful. The exploration presented in the attachment to the current chapter took 207 steps (understanding a step as the process of finding the implication basis and trying to find counter-example to each implication from the basis).

2.6 Conclusion

In this chapter methods and tools for exploration of the lattice of p-clones are developed and investigated. Two algorithms for finding counter-examples to implications over functions on A_k are introduced, investigated, and compared. It is shown that none of the algorithms uniformly outperforms the other. Therefore, the best results are obtained when using both algorithms simultaneously. Afterwards a modification of the attribute exploration process is introduced. The modification is particularly suitable for discovering p-indecomposable functions. The resulting procedure is further investigated. It is shown that despite several discouraging result about the new exploration process it is possible to guarantee the success on A_3 if all the unary functions are discovered at some point.

2.7 Remarks

1. Time taken by the whole procedure for $k = 3$ is around two weeks. It is difficult to estimate the dependence between the time of the whole exploration and k as the time depends essentially on the number of p-indecomposable functions. Moreover, it also depends on the particular order of finding functions. When the context \mathbb{K}_F is small ($|F|$ is small) then there normally exist a lot of counter-examples. If the algorithm is “lucky”, i.e. it finds a p-indecomposable function out of all possible counter-examples, then it will definitely need less time than an “unlucky” algorithm. Finding such “lucky” algorithms would be an interesting topic for further investigation.

We have to take into account that the commutativity check takes $O(k^{ar_1*ar_2})$. Hence, if $ar_1 = ar_2 = k = 4$ then the commutativity check takes $\frac{4^{4*4}}{3^{3*3}} \approx 218200$ times more operations. We assume that in average the luckiness of the algorithm and the number of p-indecomposable function depend on the total number of function U_k

- linearly. Then the new exploration time would be (several weeks) multiplied by $\frac{|U_4|}{|U_3|}$. Therefore, the new exploration time would be about 10^{31} weeks. This is totally infeasible. However, the assumption of linear growth is very pessimistic.
 - logarithmically. Therefore, the new exploration time would be about 10^7 weeks. With further optimizations this time may be reduced and become feasible.
2. Complexity of `satisfy_premise` may be further investigated to find a better bound. If it is necessary to iterate over all possible functions of given arity then the closure under commutation is trivial. On the other hand, if closing under commutation requires many operations then many functions are skipped. The investigation of the interplay of these two factors may give a better bound on the complexity. However, this investigation is in connection with the problem of finding p-indecomposable functions in general. Indeed, if we knew which assignments arise from commutation with functions F then we would have a deeper insight into the p-expressible through F functions.
 3. The lattice of context $\mathbb{K}_{F_4 \cup f_6^b}$ is presented in Figure 2.21. The only new concept is $(f_6^{b \perp_{F_4 \cup f_6^b}}, f_6^{b \perp_{F_4 \cup f_6^b}})$, a lower neighbor of C_* . The intent of the new concept consists exactly of the intent of C_* and f_6^b . For a more thorough description

see Subsection 2.4.1.

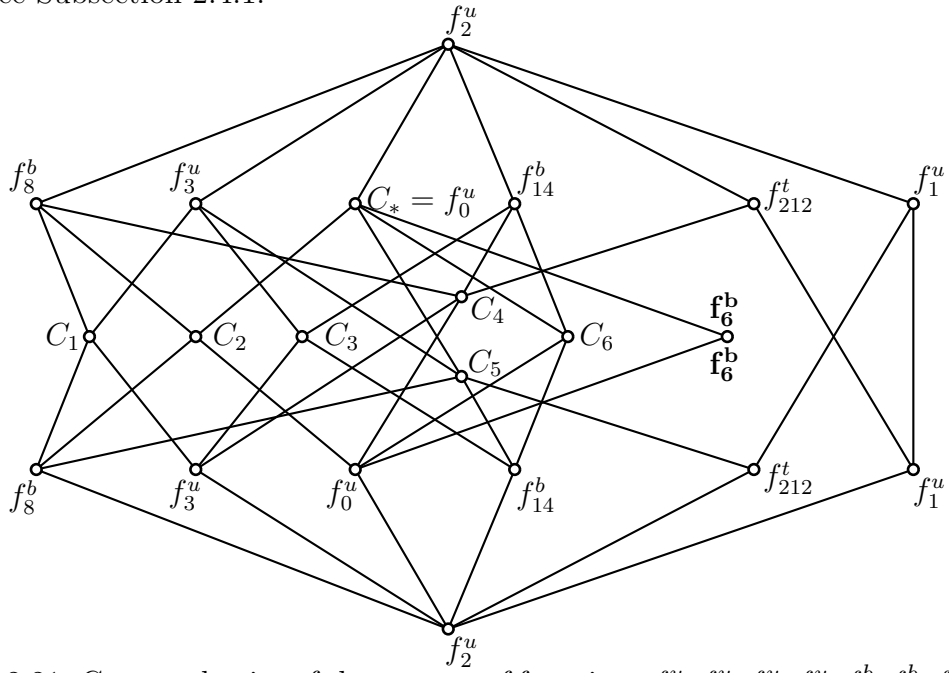


Figure 2.21: Concept lattice of the context of functions $f_0^u, f_1^u, f_2^u, f_3^u, f_6^b, f_8^b, f_{14}^b, f_{212}^t$

4. In general if f is a new first-order irreducible function and for some concept C we have that $\text{int}(C) = f^{\perp F}$ then the new concept $(f^{\perp F \cup f}, f^{\perp F \cup f})$ is either the upper neighbor (if $f \not\perp f$) or the lower neighbor (if $f \perp f$) of the concept C .

Appendix to Chapter 2

Table 2.4: Function f_0^u

$$f_0^u =$$

x	$f(x)$
0	0
1	0

Table 2.5: Function f_1^u

$$f_1^u =$$

x	$f(x)$
0	1
1	0

Table 2.6: Function f_2^u

$$f_2^u =$$

x	$f(x)$
0	0
1	1

Chapter 2 Lattice of P-Clones

Table 2.7: Function f_3^u

$$f_3^u =$$

x	$f(x)$
0	1
1	1

Table 2.8: Function f_8^b

$$f_8^b =$$

x_1	x_2	$f(x_1, x_2)$
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.9: Function f_{13}^b

$$f_{13}^b =$$

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	0
1	0	1
1	1	1

Table 2.10: Function f_{14}^b

$$f_{14}^b =$$

x_1	x_2	$f(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	1

Table 2.11: Function f_{150}^t

$$f_{150}^t =$$

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Table 2.12: Function f_{212}^t

$$f_{212}^t =$$

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table 2.13: Function f_{232}^t

$$f_{232}^t =$$

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 2.14: Function $f_{3,0}^u$

$$f_{3,0}^u =$$

x	$f(x)$
0	0
1	0
2	0

Table 2.15: Function $f_{3,1}^u$

$$f_{3,1}^u =$$

x	$f(x)$
0	1
1	0
2	0

Table 2.16: Function $f_{3,12015}^b$

$$f_{3,12015}^b =$$

x_1	x_2	$f(x_1, x_2)$
0	0	0
0	1	0
0	2	0
1	0	1
1	1	1
1	2	1
2	0	1
2	1	2
2	2	1

Table 2.17: Function $f_{3,756}^b$

$$f_{3,756}^b =$$

x_1	x_2	$f(x_1, x_2)$
0	0	0
0	1	0
0	2	0
1	0	1
1	1	0
1	2	0
2	0	1
2	1	0
2	2	0

Chapter 3

Automatized Construction of Implicative Theory of Algebraic Identities of Size up to 5

3.1	Algebras and Algebraic Identities	75
3.2	Context of Algebras and Identities	76
3.2.1	Identities as Attributes	77
3.3	Finding Counter-examples	81
3.3.1	Necessity of Infinite Algebras	82
3.3.2	Finding Infinite Counter-examples	85
3.4	Exploration of Algebraic Identities	94
3.5	Results	94
3.6	Conclusion	95
3.7	Remarks	96
	Appendix to Chapter 3	97

Chapter Summary

Algebraic identities describe different classes of algebraic structures (equational classes) and therefore play one of the central roles in algebra. The field of research that studies common patterns of algebraic structures is called universal algebra [Bir35]. As noted in [Tay79]: “The role of algebraic equations was pronounced from the start”. The study of equational classes is essential for mathematics.

A central question about equational classes is the following: if a class satisfies a given set of identities which other identities are necessarily satisfied by all the members of the class? The strength and importance of equational deduction can be well appreciated from the words from [CT51]: “it has even been shown that every problem concerning the derivability of a mathematical statement from a given set of axioms can be reduced to the problem of whether an equation is identically satisfied in every relation algebra. One could thus say that, in principle, the whole of mathematical research can be carried out by studying identities in the arithmetic of relation algebras.” It is well known that in general it is not possible to decide if an identity is deducible from a given set of identities, see e.g. [Tar41]. Even for a finite set of identities this question can be undecidable [Per67, p. 179], [Tay79, p. 28]. However, there are special classes of identities for which the question is decidable, for example, groups [Deh11]. The modern field of science called automated theorem proving has made a big progress in equational deduction (as a part of deduction in first order logic). However, equational deduction is semidecidable in general, meaning that it is not always possible to say if the answer is negative, i.e. when an identity does not hold. As a counterpart of automatic theorem provers, automatic model finders are also actively developed. However, modern tools focus on finite models.

Deductibility is not at all the only question of interest about equational classes. As pointed out in [BS81, Recent Developments and Open Problems] finding (finite) bases for equational theories and classification of equational classes are current research activities as well. For the purpose of solving these two questions in a given set of identities one could find all possible interrelations between identities inside this set (implicative theory of identities). Up to now no automated knowledge processing algorithm was offered to automatize the research of the implicative theory of a given set of identities.

Automating AE for the exploration of identities and making it efficient raises a number of unique challenges. For example, though only 70 identities of size up to 5

are under investigation in the current chapter, it turns out that it is not possible to finish the investigation considering only finite counter-examples.

An investigation similar to the current one was successfully carried out in [Kes13], however, the automation of the procedure (finding infinite counter-examples, checking satisfaction of identities in infinite algebras, finding proofs for identities) as well as an extension of the used methods for a more general case were not intended.

In Section 3.1 we present some general definitions and standard results from the field of universal algebra. None of these is new. For more information we refer the reader to [BS81, HM88], and [Bar82, Chapters A.1, A.2].

In Section 3.2 we consider the context of algebras and identities. In particular, we describe an algorithm for checking the satisfaction of identities in finite algebras and an algorithm for finding non-equivalent identities of a given size.

The most significant results of the current chapter are concentrated in Section 3.3. Namely, in Subsection 3.3.1 we introduce and prove the criteria for the necessity of infinite counter-examples. In Subsection 3.3.2 we investigate the structure of infinite counter-examples and introduce an algorithm for finding them. At the end of the chapter results are presented and discussed, conclusion is given.

Contributions

- An algorithm for finding non-equivalent identities is introduced and implemented;
- The conditions for the necessity of infinite counter-examples are introduced and proved;
- The structure of infinite counter-examples is investigated;
- A computational model and an algorithm for generating infinite algebras satisfying a set of identities but not satisfying a given identity is developed and implemented;
- The results of a successful exploration are discussed.

3.1 Algebras and Algebraic Identities

Consider a first-order formal language L consisting of variables X and a set of function symbols Φ with arities $ar(\Phi)$. When we refer to L -terms instead of $TM(L)$ we write $T_\Phi(X)$ to explicitly mention the function symbols and variables. For a term $p \in T_\Phi(X)$ the *size* or *length* $l[p]$ is the number of all occurrences of function symbols and variables in p .

An *identity* of Φ over X is an expression of the form

$$p \equiv q,$$

where $p, q \in T_\Phi(X)$.

The *size* or *length* $l[p \equiv q]$ of an identity $p \equiv q$ is the sum of the sizes of both terms, i.e. $l[p \equiv q] := l[p] + l[q]$.

The set of identities is a subset of the set of first-order L -formulae.

A (Φ) -*algebra* is an L -structure for the formal language $L = (X, \Phi, ar(\Phi))$. An algebra \mathcal{A} is *finite* if its universe A is finite, otherwise \mathcal{A} is *infinite*. We call the interpretation $p^{\mathcal{A}}$ of a term $p \in T_\Phi(X)$ a *term operation*. For $f \in \Phi$ a term operation $f^{\mathcal{A}}$ is called a *fundamental operation* of \mathcal{A} .

Satisfaction of identities in algebras plays a crucial role for the investigation. Terms $p, q \in T_\Phi(X)$ are called *pairwise equivalent* if the identity $p \equiv q$ is satisfied in every (Φ) -algebra. We call two identities $p_1 \equiv q_1$ and $p_2 \equiv q_2$ *pairwise equivalent* if they are satisfied in the same algebras, i.e. if $p_1 \equiv q_1 \Vdash p_2 \equiv q_2$ and $p_2 \equiv q_2 \Vdash p_1 \equiv q_1$.

Equational Classes, Free Algebras, Locally Finite Algebras

Definition 3.1.1 (Subalgebra). Let \mathcal{A} and \mathcal{B} be two Φ -algebra. \mathcal{B} is a *subalgebra* of \mathcal{A} if $B \subseteq A$ and every fundamental operation $f^{\mathcal{B}}$ is a restriction of the corresponding operation $f^{\mathcal{A}}$ on B .

For $S \subseteq A$ we say that \mathcal{B} is a *subalgebra of \mathcal{A} generated by S* if B is equal to the smallest set containing S and closed under all fundamental operations of \mathcal{A} .

Definition 3.1.2 (Homomorphism). Let \mathcal{A} and \mathcal{B} be two Φ -algebra. A mapping $\varphi : \mathcal{A} \rightarrow \mathcal{B}$ is called a *homomorphism* from \mathcal{A} to \mathcal{B} if

$$\varphi f^{\mathcal{A}}(a_1, \dots, a_n) = f^{\mathcal{B}}(\varphi a_1, \dots, \varphi a_n).$$

Definition 3.1.3 (Equational class). For a set of identities Σ the class of all algebras $\text{Mod}(\Sigma)$ such that in every algebra from $\text{Mod}(\Sigma)$ every identity from Σ is satisfied is called an *equational class*. The set of identities that hold in each algebra from an equational class \mathfrak{E} is denoted by $\text{Eq}(\mathfrak{E})$. The equational class $E(\mathcal{A})$ is the equational class $\text{Mod}(\text{Eq}(\{\mathcal{A}\}))$.

Definition 3.1.4 (Free algebra in equational class). Let E be an equational class. An algebra $F_E(X)$ is called the *free algebra in E , freely generated by X* if every mapping of X into any algebra from E extends to a homomorphism of $F_E(X)$ into that algebra.

Definition 3.1.5 (Locally finite algebra). An algebra \mathcal{A} is called *locally finite* iff every subalgebra of \mathcal{A} generated by finitely many elements is finite.

Theorem 3.1.6. *Let \mathcal{A} be an algebra. \mathcal{A} is locally finite iff $F_{E(\mathcal{A})}(X)$ is finite for any X such that $|X| < |\mathbb{N}|$.*

3.2 Context of Algebras and Identities

In the current investigation we are interested in $\Phi = (*, \triangleright, a)$ with arities $ar(*) = 2$, $ar(\triangleright) = 1$, $ar(a) = 0$. Alternatively we refer to these function symbols via $f^{(2)}, f^{(1)}, f^{(0)}$ with corresponding arities in superscripts. For the chosen Φ the Φ -algebras are called *bunnies* (from Binary, Unary, Nullary).

Several important classes of algebras can be defined using all or several function symbols from the chosen signature. Such classes include finite and infinite algebras.

Example 3.2.1. A groupoid is an algebra $(A, *)$. A groupoid satisfying associativity $(x * y) * z \equiv x * (y * z)$ is called a semigroup. If in a semigroup exists an identity element a , i.e. $a * x \equiv x * a \equiv x$, the semigroup is called a monoid. A group is a monoid with inverse elements $x * (\triangleright x) \equiv (\triangleright x) * x \equiv a$.

We will use a set of identities M_{id} as attributes of a formal context $\mathbb{K}_{\text{B}} = (G_{\text{B}}, M_{\text{id}}, I)$ and algebras as objects. An algebra \mathcal{A} is in relation I with an identity $id \in M_{\text{id}}$ iff $\mathcal{A} \models id$.

In order to obtain a finite set of identities we constrain the size of identities to be not larger than five. Because of this constraint it suffices to use only three variables

$X = \{x, y, z\}$. Our goal is to find all equational classes that can be defined by these identities, i.e. we aim at building the lattice of all classes that are closed under deduction. In order to do this we need to determine if an implication over identities holds. Therefore we either prove that the identities in the conclusion follow from the identities in the premise (in the sense of equational logic or first-order logic) or present a counter-example. For the purpose of deduction we use the automated theorem prover **Prover9**, for finding finite models we use **Prover9**'s counterpart **Mace4** [McC10]. However, in general the equational theory involving identities from above (and some others) is undecidable, because of, for example, having the identity $x * x \equiv x$ [BHSS89, pp. 34–36].

3.2.1 Identities as Attributes

An algorithm for checking the satisfaction of identities in finite algebras arises from the definitions. Note that the algorithm is not appropriate for checking the satisfaction of an identity in an infinite algebra if this infinite algebra satisfies this identity as the algorithm would require an infinite number of operations.

Input: $\mathcal{A} = (A, \Phi)$, $p, q \in T_{\Phi}(X)$.
Output: Is an identity $p \equiv q$ satisfied in the algebra \mathcal{A} ?

```

1 for map in  $A^X$ :
2    $a_1, \dots, a_n \leftarrow \text{map}(x_1), \dots, \text{map}(x_n)$ 
3   if not  $p^{\mathcal{A}}(a_1, \dots, a_n) == q^{\mathcal{A}}(a_1, \dots, a_n)$ :
4     return False
5 return True

```

Algorithm 3.1: check_identity

Generating pairwise non-equivalent identities

In order to generate identities we start from generating terms of limited length and afterwards pair them and gather the pairs in a set of identities. Having this set we filter out pairwise equivalent identities. What is left is the desired set of pairwise nonequivalent identities of given length.

The algorithm for generating terms exploits the idea that a term of fixed signature is uniquely determined by three tuples:

1. Variables **vars**;

2. Orders of application of function symbols **fs_order**;
3. Types of function symbols **fs_types**.

The variables are ordered in the tuple **vars**. The nullary functions require special handling in this representation. For the nullary function we always use a “placeholder”. The nullary functions are always performed first. A function with a minimal corresponding order is applied next. If two functions have the same order the result should be independent from the order of their applications.

The idea of processing these tuple into a term is best illustrated with an example.

Example 3.2.2. The input consists of three tuples:

vars := (x, x) ;
fs_order := $(0, 2, 1)$;
fs_types := $(f^{(0)}, f^{(2)}, f^{(1)})$.

We process the input “in place”, i.e. we use the input tuples to produce the output. The final result is constructed in the tuple **vars**. First the functions with corresponding order 0 is applied, i.e. the nullary function. The intermediate result is:

vars := $(f^{(0)}, x)$;
fs_order := $(2, 1)$;
fs_types := $(f^{(2)}, f^{(1)})$.

Next the functions with corresponding order 1 is applied – the unary function on second argument. The intermediate result is:

vars := $(f^{(0)}, f^{(1)}(x))$;
fs_order := (2) ;
fs_types := $(f^{(2)})$.

In the last step the binary function is applied. The result is $f^{(2)}(f^{(0)}, f^{(1)}(x))$ or $a * (\triangleright x)$.

The algorithm for generating terms of a given length s and over given set of variables X is presented in Algorithm 3.2. The desired output of the algorithm is the set of terms $T^{(s)}(X) \subseteq T(X)$ such that for all $p \in T^{(s)}(X) : l[p] = s$.

The function $\#$ returns the length of its argument, for example, $\#((x, x)) = 2$. The function $N_{\text{fs_types}}$ returns the number of occurrences of its argument in the tuple

fs_types. In Line 3 the algorithm produces all possible tuples **fs_types** – cartesian power of function symbols $\{f^{(2)}, f^{(1)}, f^{(0)}\}$.

In Line 4 the algorithm sorts out those tuples **fs_types** that will not produce a term of desired length s . For example, the tuple $(f^{(2)}, f^{(2)}, f^{(0)})$ cannot produce a term of length 4 as we already have 3 functions and we will not manage to produce a valid term with only 1 additional variable. We obtain a condition on the length of variables and the number of occurrences of different functions in the constructed term. The number $\#(\mathbf{vars})$ represents the number of occurrences of variables in the constructed term. This number is fixed first, before the actual tuple **vars** is constructed. On the one hand, from the constraint on the size of the term we obtain that $\#(\mathbf{vars})$ is equal to $s - N_{\mathbf{fs_types}}(f^{(2)}) - N_{\mathbf{fs_types}}(f^{(1)})$. On the other hand, from the definition of terms as valid applications of functions (Definition 1.3.2) we obtain that $\#(\mathbf{vars})$ is equal to $N_{\mathbf{fs_types}}(f^{(2)}) + 1$. Therefore, we obtain the condition:

$$2 \times N_{\mathbf{fs_types}}(f^{(2)}) + N_{\mathbf{fs_types}}(f^{(1)}) + 1 = s.$$

In Line 5 we fix $\#(\mathbf{vars})$. In Lines 6 - 9 for each possible order of application of functions **fs_types** we do the following. If the arity of function is zero then the corresponding order of application is changed to 0 in Line 9 so that this function is performed first. The function `Term.tuples2term` in Line 11 produces a string of applications of functions as shown in Example 3.2.2. Before adding the tuples to the result, in Line 12 we check that the tuples are not equivalent to some term that is already added to **result**. This is performed through comparing respective functional strings. The set **result** may already contain an equivalent term due to the fact that orders of nullary functions were changed and any variable could be a placeholder for the nullary function.

Algorithm is sound, complete, and terminates. The algorithm terminates as there are only finite cycles. The algorithm produces only the desired terms and finds all of them.

As follows from the description the algorithm does some overwork due to producing possible tuples that are later filtered out in Lines 4 and 12. Nevertheless, it works efficiently enough to be able to produce (in minutes on a standard modern computer) terms of much larger sizes than we would need in this chapter (at least up to 8). In the practical usage in Line 2 we may change $[1, s]$ to $[k, s]$ such that $k = \lceil \frac{s+1}{2} \rceil$, because for **size_fs** in $[1, k]$ we will not find any **fs_types** that would produce a term of given length.

```

Input:  $s \in \mathbb{N}$ ,  $X$ .
Output:  $T^{(s)}(X)$  such that  $\forall p \in T^{(s)}(X) : l[p] = s$ .

1 result  $\leftarrow \emptyset$ 
2 for size_fs in  $[1, s]$ :
3   for fs_types in  $\{f^{(2)}, f^{(1)}, f^{(0)}\}^{\text{size\_fs}}$ :
4     if  $2 \times N_{\text{fs\_types}}(f^{(2)}) + N_{\text{fs\_types}}(f^{(1)}) + 1 == s$ :
5        $\#(\text{vars}) \leftarrow N_{\text{fs\_types}}(f^{(2)}) + 1$ 
6       for fs_order in permutations( $[1, \text{size\_fs}]$ ):
7         for i in  $[0, \text{size\_fs}]$ :
8           if fs_types[i] ==  $f^{(0)}$ :
9             fs_order[i]  $\leftarrow 0$ 
10        for vars in  $X^{\#(\text{vars})}$ :
11          term  $\leftarrow \text{Term.tuples2term}(\text{vars}, \text{fs\_order}, \text{fs\_types})$ 
12          if not term in result:
13            result.add(term)
14 return result

```

Algorithm 3.2: generate_terms

In order to generate identities of size s it is necessary to combine terms p and q such that $l[p] + l[q] = s$. However, it remains to filter out the pairwise equivalent identities. The following propositions explore some classes of pairwise equivalent identities.

Proposition 3.2.3. *If p is of the form $x, x \in X$ and q does not contain any occurrences of x then $p \equiv q$ is equivalent to $x \equiv a$.*

Proof. Let two assignment functions \bar{h}_0, \bar{h}_1 differ only in that $\bar{h}_0(x) \neq \bar{h}_1(x)$. As q does not contain x we have $\bar{h}_0(q) = \bar{h}_1(q)$, hence $\bar{h}_0(x) = \bar{h}_1(x)$, contradiction. Hence, the identity $p \equiv q$ may only be satisfied if there does not exist \bar{h}_0, \bar{h}_1 such that $\bar{h}_0(x) \neq \bar{h}_1(x)$. Hence, for any assignment function \bar{h} the value $\bar{h}(x)$ is equal to the only element of the domain a^A . \square

Hence, it suffices to consider only one identity from the class described in Proposition 3.2.3, we take the identity $x \equiv y$.

Proposition 3.2.4. *If p is of the form $\triangleright x, x \in X$ and q is of the form $\triangleright t, t \in T(X)$ such that t does not contain any occurrences of x then $p \equiv q$ is equivalent to $\triangleright x \equiv \triangleright a$.*

Proof. Let two assignment functions \bar{h}_0, \bar{h}_1 differ only in that $\bar{h}_0(x) \neq \bar{h}_1(x)$. As t does not contain x we have $\bar{h}_0(t) = \bar{h}_1(t)$. Therefore, $\triangleright^{\mathcal{A}}(\bar{h}_0(x)) = \triangleright^{\mathcal{A}}(\bar{h}_1(x)) = \triangleright^{\mathcal{A}}(\bar{h}_0(t))$. As we can take further assignment functions \bar{h}_2, \dots that only differ in the assignment for x , we obtain the condition that for all $b, c \in A : \triangleright^{\mathcal{A}}(b) = \triangleright^{\mathcal{A}}(c)$. Hence, the output of $\triangleright^{\mathcal{A}}$ is constant and equal to $\triangleright^{\mathcal{A}}(a^{\mathcal{A}})$. \square

Out of the identities described in Proposition 3.2.4 we take the identity $\triangleright x \equiv \triangleright a$.

Of course if the only difference between two identities is in the names of variables (e.g. “ $x \equiv x * y$ ” and “ $y \equiv y * z$ ”) then they are pairwise equivalent, hence, filtered out.

The functions implementing the stated ideas are contained in the software libraries written in order to perform the current investigation. The algorithm takes the size of identities and the number of variables as input arguments and produces the desired set of identities. The algorithm is able to produce identities of arbitrary size, these identities may be used for further investigations.

The produced set of pairwise non-equivalent identities of size up to five consists of 70 identities, this set M_{id} is listed in Table 3.7. In [Kes13] it was shown on examples that these identities are indeed pairwise non-equivalent.

3.3 Finding Counter-examples

The task of finding counter-examples to an implication $P \rightarrow c$, where P is a set of identities and c is an identity, consists in finding a bunny \mathcal{A} satisfying P and not satisfying c , i.e. $\mathcal{A} \models P, \mathcal{A} \not\models c$.

In practice it is usually easier to work with finite algebras. For example, it is possible to directly check the satisfaction of an identity by iterating over all possible assignment functions. Moreover, in the current investigation the automatic model finder **Mace4** [McC10] was used. However, as was shown already in [Kes13], it is not possible to reach our goal considering only finite algebras. Modern automatic model finders like **Mace4** [McC10], **E-Darwin** [BFdNT07] and **Paradox** [CS] are only designed for finding finite models.

3.3.1 Necessity of Infinite Algebras

Next we prove the necessity of infinite counter-examples, however, we use other methods than in [Kes13]. These methods allow us to introduce criteria and investigate the structure of infinite counter-examples.

Proposition 3.3.1. *If in algebra \mathcal{A} and for some $n \in \mathbb{N}$ there exists an infinite set of term operations $\{p_i(x_1, \dots, x_n)\}_{i \in I}$ then $F_{E(\mathcal{A})}(X)$ is infinite.*

This proposition follows instantly from the definition of $F_{E(\mathcal{A})}(X)$. From this proposition and Theorem 3.1.6 we obtain

Corollary 3.3.2. *If $\{p_i(x_1, \dots, x_n)\}_{i \in I}$ is infinite then \mathcal{A} is not locally finite.*

Next follows a criterion for the necessity of infinite counter-examples.

Lemma 3.3.3. *Let Y be a tuple of variables. If there exist unary terms p, q and terms r, r^* such that*

$$\mathcal{A} \models x \equiv pq(x), \quad pr(Y) \equiv pr^*(Y),$$

$$\mathcal{A} \models r(Y) \not\equiv r^*(Y)$$

then \mathcal{A} is not locally finite.

Proof. Consider $Q_n := \{q^i(x)\}_{i \leq n}$, $Q := \{q^i(x)\}_{i \in \mathbb{N}}$. Proof by induction.

1. Assume $q(x) \equiv x$. Hence, $p(x) \equiv x$. Hence, $r(Y) \equiv r^*(Y)$, contradiction. Therefore, $q(x) \not\equiv x$.
2. Assume all terms in Q_{n-1} are pairwise non-equivalent and there exists $k < n$: $q^n(x) \equiv q^k(x)$.
 - a) Assume $q^n(x) \equiv x$. Hence, $x \equiv p^n(x)$. Hence, $r(Y) \equiv p^{n-1}p(r(Y)) \equiv p^{n-1}p(r^*(Y)) \equiv r^*(Y)$. Therefore, $r(Y) \equiv r^*(Y)$, we come to a contradiction. Hence, $q^n(x) \not\equiv x$.
 - b) Assume $q^n(x) \equiv q^k(x)$. We have $p^k q^k(x) \equiv x$ and $p^k q^n(x) \equiv q^l(x)$, $l = n - k$. Hence, $x \equiv q^l(x)$ and we come to a contradiction to the initial assumption about Q_{n-1} . Hence, $q^n(x) \not\equiv q^k(x)$.

Hence, $q^n(x) \not\equiv q^k(x)$.

Therefore, the set of terms Q is infinite, hence, \mathcal{A} is not locally finite. \square

Table 3.1: A counter-example \mathcal{A} to $\{x \equiv a * (\triangleright x)\} \rightarrow x \equiv \triangleright(a * x)$

$$m *^{\mathcal{A}} n := \begin{cases} 0 *^{\mathcal{A}} n = n - 1, & \text{if } n \geq 1; \\ m *^{\mathcal{A}} n = 0, & \text{else;} \end{cases}$$

$$\triangleright^{\mathcal{A}} n := n + 1;$$

$$a^{\mathcal{A}} := 0.$$

Corollary 3.3.4. *If there exist two unary terms p, q such that $\mathcal{A} \models x \equiv pq(x)$ and $\mathcal{A} \models x \not\equiv qp(x)$ then \mathcal{A} is not locally finite.*

Proof. Consider $r(x) := x$, $r^*(x) := qp(x)$. Hence, $pr(x) = p(x)$, $pr^*(x) = pqp(x) \equiv p(x)$, therefore, $pr(x) \equiv pr^*(x)$. Now we can use Lemma 3.3.3. \square

Proposition 3.3.5. *Let \mathcal{A} be a bunny. If \mathcal{A} is a counter-example to the implication*

$$\{x \equiv a * (\triangleright x)\} \rightarrow x \equiv \triangleright(a * x) \quad (3.1)$$

then \mathcal{A} is not locally finite.

Proof. Consider $p(x) = a * x$, $q(x) = \triangleright x$, and Corollary 3.3.4. \square

Therefore, all finite algebras satisfy Implication (3.1).

Example 3.3.6. A counter-example to Implication (3.1) is the infinite algebra \mathcal{A} presented in Table 3.1.

Although $\mathcal{A} \models x \equiv a * (\triangleright x)$, the function \triangleright is surjective, but not bijective. This is only possible if the universe is infinite.

Several other implications over our identities have only infinite counter-examples.

Proposition 3.3.7. *Let \mathcal{A} be a bunny. If \mathcal{A} is a counter-example to the implication*

$$\{x \equiv \triangleright(a * x)\} \rightarrow x \equiv a * (\triangleright x) \quad (3.2)$$

then \mathcal{A} is not locally finite.

Proof. Consider $p(x) = \triangleright x$, $q(x) = a * x$, and Corollary 3.3.4. \square

Proposition 3.3.8. *Let \mathcal{A} be a bunny. If \mathcal{A} is a counter-example to the implication*

$$\{a \equiv \triangleright a, a \equiv \triangleright(x * a), x \equiv \triangleright(x * x)\} \rightarrow a \equiv x * a \quad (3.3)$$

then \mathcal{A} is not locally finite.

Proof. Consider $p(x) = \triangleright x$, $q(x) = x * x$, $r = a$, $r^*(x) = x * a$ and Lemma 3.3.3. \square

Proposition 3.3.9. *Let \mathcal{A} be a bunny. If \mathcal{A} is a counter-example to the implication*

$$\{\triangleright a \equiv \triangleright(\triangleright a), x \equiv \triangleright(x * x)\} \rightarrow a \equiv \triangleright a \quad (3.4)$$

then \mathcal{A} is not locally finite.

Proof. Consider $p(x) = \triangleright x$, $q(x) = x * x$, $r = \triangleright a$, $r^* = a$ and Lemma 3.3.3. \square

Proposition 3.3.10. *Let \mathcal{A} be a bunny. If \mathcal{A} is a counter-example to the implication*

$$\{a \equiv \triangleright(x * a), x \equiv \triangleright(x * x), \triangleright x \equiv a * x\} \rightarrow \triangleright a \equiv x * a \quad (3.5)$$

then \mathcal{A} is not locally finite.

Proof. Consider $p(x) = \triangleright x$, $q(x) = x * x$, $r = \triangleright a$, $r^*(x) = x * a$. We have $pr^*(x) = \triangleright(x * a) = a$, $pr = \triangleright(\triangleright a) = \triangleright(a * a) = a$. Now we can use Lemma 3.3.3. \square

Proposition 3.3.11. *Let \mathcal{A} be a bunny. If \mathcal{A} is a counter-example to the implication*

$$\{a \equiv \triangleright(\triangleright a), x \equiv \triangleright(x * x)\} \rightarrow \triangleright a \equiv a * a \quad (3.6)$$

then \mathcal{A} is not locally finite.

Proof. Consider $p(x) = \triangleright x$, $q(x) = x * x$, $r = p(a) = \triangleright a$, $r^* = q(a) = a * a$. We have $pr = \triangleright(\triangleright a) = a$, $pr^* = \triangleright(a * a) = a$. Now we can use Lemma 3.3.3. \square

As the experiment proves, Propositions 3.3.5, 3.3.7, 3.3.8, 3.3.9, 3.3.10, 3.3.11 describe all possible **classes** of implications over M_{id} that have only infinite counter-examples. Each class of implications contains the respective implication, e.g. Implication (3.6), and implications obtained by adding identities to the premise of implications, e.g. the implication

$$\{a \equiv \triangleright(\triangleright a), x \equiv \triangleright(x * x), \triangleright x \equiv a * x\} \rightarrow \triangleright a \equiv a * a \quad (3.7)$$

is in the class of Implication (3.6). However, after adding certain identities to the premise an implication may become valid. Such implications, of course, do not have any counter-examples and do not belong to the described classes. All other invalid implications over M_{id} may be violated by finite algebras.

3.3.2 Finding Infinite Counter-examples

Now, after we have identified that infinite counter-examples are necessary and described classes of implications that have only infinite counter-examples, we investigate possibilities of finding these infinite counter-examples. This investigation starts from the analysis of their structure.

Structure of Infinite Counter-examples

As follows from the Downward Löwenheim-Skolem Theorem [Löv15], [Bar82, Chapter A.1], it suffices to consider only models over a countable universe A .

Let

$$\begin{aligned} T_{fin} &= \Sigma \cup \{pr \equiv pr^*\}, \\ T_{fin}^* &= T_{fin} \cup \{r \not\equiv r^*\}, \end{aligned}$$

where Σ is a set of identities. The terms p and q are, as before, some unary terms, $p, q, r, r^* \in T(X)$.

Let $ar(r) = ar(r^*) = n$. For an algebra \mathcal{A} we fix the tuple $C_v \in A^n$ such that $r^A(C_v) \neq r^{*A}(C_v)$, if it exists. Denote $c := r^A(C_v)$, $c^* := r^{*A}(C_v)$, then $c \neq c^*$. For $b \in A$, $B \subseteq A$ we introduce the following notations:

$$\begin{aligned} q^{-\infty}(b) &= \{x \mid \exists n \in \mathbb{N} : (q^A)^n(x) = b\}, \\ q^{-\infty}(B) &= \cup \{q^{-\infty}(b) \mid b \in B\}, \\ q^\infty(b) &= \{(q^A)^n(b) \mid n \in \mathbb{N}\}, \\ q^\infty(B) &= \cup \{q^\infty(b) \mid b \in B\}. \end{aligned}$$

Lemma 3.3.12. *If $\mathcal{A} \models T_{fin}^* \cup \{x \equiv pq(x)\}$ then there do not exist $b, b^* \in A$: $q^A(b) = c$, $q^A(b^*) = c^*$.*

Proof. Assume the opposite. Then $b = p^A q^A(b) = p^A(c)$ and $b^* = p^A q^A(b^*) = p^A(c^*)$. As $pr(Y) \equiv pr^*(Y)$, we have $p^A(c) = p^A(c^*)$. Therefore $b = p^A(c) = p^A(c^*) = b^*$. But then $c = q^A(b) = q^A(b^*) = c^*$ yields a contradiction to the assumption $c \neq c^*$. \square

Therefore, in \mathcal{A} at least one of the elements c, c^* does not have a preimage under q^A . Without loss of generality let c have no preimage under q^A , i.e. there is no $a \in A$ such that $q^A(a) = c$. Consider the set $q^\infty(c)$.

Proposition 3.3.13. *Under the conditions of Lemma 3.3.12, the set $q^\infty(c)$ is infinite.*

Proof. 1. $(q^A)^n(c) \neq c$. Otherwise $q^A(\hat{c}) = c$, $\hat{c} = (q^A)^{n-1}(c)$, hence, \hat{c} is a preimage of c under q^A , contradiction.
 2. $(q^A)^n(c) \neq (q^A)^k(c)$. Otherwise $(q^A)^{n-k}(c) = c$, and we arrive at the first case.

Therefore, $q^\infty(c)$ is an infinite subset of the universe. \square

Note that as $\mathcal{A} \models x \equiv pq(x)$ then for all $n \in \mathbb{N}$: $\mathcal{A} \models pq^n(x) \equiv q^{n-1}(x)$.

Let $C = \{d \in \{c, c^*\} \mid \nexists b \in A : q^A(b) = d\}$.

Remark. One can observe that in order to satisfy $x \equiv pq(x)$ in algebra \mathcal{A} the term operation q^A has to be injective and the term operation p^A has to be surjective [Kes13, pp. 83-85].

The next lemma states that a subuniverse $B \subseteq A$ can only be finite if $C \cap q^{-\infty}(B) = \emptyset$.

Lemma 3.3.14. *If $T_{fin} \cup \{x \equiv pq(x)\}$ has a finite nontrivial model and $T_{fin}^* \cup \{x \equiv pq(x)\}$ is satisfiable then there exists an algebra \mathcal{A} such that $\mathcal{A} \models T_{fin}^* \cup \{x \equiv pq(x)\}$ and*

$$\forall B \subseteq A : (q^\infty(B) \text{ is finite}) \Rightarrow \forall d \in C : d \notin q^{-\infty}(B).$$

Proof. Take $B \subseteq A$ such that $q^\infty(B)$ is finite. Now suppose there exists $d \in C$: $d \in q^{-\infty}(B)$. Therefore, there exists $n > 0$ such that $(q^A)^n(d) = b$, $b \in B$. From Proposition 3.3.13 we know that $q^\infty(C)$ is infinite, therefore, $q^\infty(b)$ is infinite as well, contradiction. \square

Corollary 3.3.15. *For all $b \in A \setminus C$ there exist a unique preimage under q^A in A . For all $b \in A \setminus \{p^A(d) \mid d \in C\}$ there exist a unique preimage under p^A in A and for each $d \in C$ there exists $E \subseteq A$ such that for all $e \in E$: $p^A(e) = p^A(d)$.*

Generating Infinite Algebras

Denote by \mathcal{A}_{fin} the biggest finite subalgebra of \mathcal{A} (if it exists) and the corresponding subuniverse by A_{fin} . Consider an implication $P \rightarrow c$, $P \subseteq M_{id}$, $c \in M_{id}$. We look for an infinite counter-example to this implication, i.e. an infinite algebra \mathcal{A} such that $\mathcal{A} \models P$, $\mathcal{A} \not\models c$. As the experiment proves for the chosen identities, if an implication

has only infinite counter-examples we can always find a counter-example \mathcal{A} such that for $B \subseteq A$ if $C \cap q^{-\infty}(B) = \emptyset$ then $q^\infty(B)$ is finite. See also Remark 2. It is natural to suggest that the decision variables of the problem describe the outputs of the fundamental operations $*^{\mathcal{A}} = (f^{(2)})^{\mathcal{A}}$, $\triangleright^{\mathcal{A}} = (f^{(1)})^{\mathcal{A}}$, $a^{\mathcal{A}} = (f^{(0)})^{\mathcal{A}}$. We make a natural requirement that the number of decision variables should be finite. As the universe A is infinite we can not assign a decision variable to each respective output as we would need an infinite number of decision variables. In order to overcome this obstacle we use different domains for different decision variables. The decision variables that describe the outputs of fundamental operations over the infinite set $q^\infty(C)$ may not only take values from A , but also expressions of special type.

We enumerate the infinite universe A and work with the indices of the elements. Next follows the description of the enumeration we use. The size of the set C is at most 2, we enumerate elements of this and consider an ordered set $(C, <)$ such that the order $<$ corresponds to the natural order on indices of the elements. As A_{fin} and $q^\infty(C)$ have an empty intersection and for $k, n \in \mathbb{N}, k \neq n$ and $c \in C : (q^{\mathcal{A}})^n(c) \neq (q^{\mathcal{A}})^k(c)$ then each element of the set $q^\infty(C)$ is uniquely represented by $(q^{\mathcal{A}})^n(c), n \in \mathbb{N}$.

We denote the size of A_{fin} by S . We use the enumeration induced by $c \in C$ and $n \in \mathbb{N}$. Namely, for least element c_1 of the set C and some $n \in \mathbb{N}$ the value $(q^{\mathcal{A}})^n(c_1)$ corresponds to the element $b_{2n+S} \in A$. For the second element c_2 (if it exists) of the set C and some $n \in \mathbb{N}$ the value $(q^{\mathcal{A}})^n(c_2)$ corresponds to the element $b_{2n+S+1} \in A$.

Example 3.3.16. Let $q = x * a, p = \triangleright x$. Let $c_1 \in C$ and $A_{fin} = \emptyset$. We use the respective number 1 instead of c_1 . Then we may assign $b_1 *^{\mathcal{A}} a^{\mathcal{A}} := b_3, b_3 *^{\mathcal{A}} a^{\mathcal{A}} := b_5$, and so on, i.e. $b_n *^{\mathcal{A}} a^{\mathcal{A}} := b_{n+2}$. The values of $p^{\mathcal{A}}(n)$ for $n \geq 3$ would be defined by $\triangleright^{\mathcal{A}} b_n := b_{n-2}$.

In what follows we use simply n instead of b_n .

We use an arbitrary enumeration of the set A_{fin} . Note that as the size of identities is limited to 5 there are at most three nested functions in the terms in Table 3.7. Hence, A_{fin} of an infinite algebra \mathcal{A} satisfying the identities $a \equiv \triangleright(\triangleright(\triangleright a))$ and $a \not\equiv \triangleright a$ has at least three elements. It was experimentally proved that for the chosen identities it suffices to consider the finite subuniverse A_{fin} of the size not larger than 3.

We associate the universe A with the set of natural numbers \mathbb{N} . We fix $a^{\mathcal{A}} = 0$. This choice does not reduce the generality of the approach as it is always possible to choose a different ordering of the values of the universe. In these settings we arrive at 16 decision variables for the problem of finding an infinite counter-example, see Table 3.2.

Table 3.2: Decision variables in finding infinite algebra \mathcal{A}

x	$\triangleright^{\mathcal{A}}$	$*^{\mathcal{A}}$	0	1	2	n
0	$\triangleright^{\mathcal{A}}0$	0	$0 *^{\mathcal{A}} 0$	$0 *^{\mathcal{A}} 1$	$0 *^{\mathcal{A}} 2$	$0 *^{\mathcal{A}} n$
1	$\triangleright^{\mathcal{A}}1$	1	$1 *^{\mathcal{A}} 0$	$1 *^{\mathcal{A}} 1$	$1 *^{\mathcal{A}} 2$	$1 *^{\mathcal{A}} n$
2	$\triangleright^{\mathcal{A}}2$	2	$2 *^{\mathcal{A}} 0$	$2 *^{\mathcal{A}} 1$	$2 *^{\mathcal{A}} 2$	$2 *^{\mathcal{A}} n$
n	$\triangleright^{\mathcal{A}}n$	n	$n *^{\mathcal{A}} 0$	$n *^{\mathcal{A}} 1$	$n *^{\mathcal{A}} 2$	$n *^{\mathcal{A}} n$

The pseudocode of the algorithm for finding infinite counter-examples is presented in Algorithm 3.3. The function `discover_constraints` takes identities from the premise P as the input and outputs `constraints` and a list of decision variables `required_vars`. The `constraints` represent equality constraints on the outputs of fundamental operations of \mathcal{A} . The `required_vars` are the decision variables that are needed in order to check the satisfaction of identities P . The function `discover_constraints` is presented in Algorithm 3.5.

After identifying the constraints, the global list `assignments` of possible assignments to the required decision variables is prepared in Line 2. The function `product` produces the cartesian product of the arguments; the function `domain` returns the set of possible assignments to its argument. The function `domain` returns different possible domains of values for different outputs. Namely, if a considered output arises from an assignment of a variable x over A_{fin} then possible outputs range only over A_{fin} . Otherwise, if a variable x is assigned a value from $q^\infty(C)$ then the corresponding output may get a value either from A_{fin} or from $q^\infty(C)$. As already noted, in practice we use indices to represent values from $q^\infty(C)$, hence, the respective domain for an output would include values $n, n + 1, n + 2, n - 1, n - 2$.

Further on the function `find_algebra` proceeds with making next assignment to decision variables (outputs of fundamental operations of \mathcal{A}) satisfying the discovered constraints, Line 3. This is performed with the help of the function `exists_next_A` presented in Algorithm 3.4. If such an assignment exists then in Line 4 the function `find_algebra` checks if the conclusion is violated and, if so, returns the found algebra \mathcal{A} . If the desired assignment does not exist then the search space is exhausted without finding counter-examples, hence `None` is returned.

The function `exists_next_A` finds the next assignments that comply with `constraints`. First in Line 1 the function checks if there are any assignments left to try. If so, the required decision variables get the next assignment in Line 2. The global list `assignments` is changed. Next if the resulting algebra \mathcal{A} satisfies the con-

Input: $P \rightarrow c$ ($P \subseteq M_{\text{id}}, c \in M_{\text{id}}$).
Output: \mathcal{A} such that $\mathcal{A} \models P$, $\mathcal{A} \not\models c$.

```

1 (required_vars, constraints) ← discover_constraints(P)
2 assignments ← product([domain(var) | var in required_vars])
3 while exists_next_A(required_vars, constraints):
4     if  $\mathcal{A} \not\models c$ :
5         return  $\mathcal{A}$ 
6 return None

```

Algorithm 3.3: find_algebra

straints then the functions returns **True**. Otherwise it continues to the next possible assignment. If all possible assignments are exhausted, the function returns **False**.

Input: required_vars, constraints.
Output: a Boolean value: does \mathcal{A} satisfying constraints exist?

```

1 while assignments is not empty:
2     required_vars ← assignments.pop()
3     if  $\mathcal{A}$  satisfies constraints:
4         return True
5 return False

```

Algorithm 3.4: exists_next_A

The function `discover_constraints` takes the identities from the premise and constructs the sets `required_variables` and `constraints`. Assignments of variables X to values of $A \cup \{n\}$ are taken. With the help of these assignments terms of each identity are evaluated in \mathcal{A} . The terms are set equal, hence, certain constraints over outputs of fundamental operations of \mathcal{A} are discovered. See Example 3.3.17.

Example 3.3.17. Consider the identity $x \equiv \triangleright(a * x)$. From the considered identity we obtain the following constraints on fundamental operations of \mathcal{A} and different variable assignment functions h . `constraints` contain the following bindings:

$$\begin{aligned}
 h_0(x) = 0: & \quad 0 = \triangleright^{\mathcal{A}}(0 *^{\mathcal{A}} 0); \\
 h_1(x) = 1: & \quad 1 = \triangleright^{\mathcal{A}}(0 *^{\mathcal{A}} 1); \\
 h_2(x) = 2: & \quad 2 = \triangleright^{\mathcal{A}}(0 *^{\mathcal{A}} 2); \\
 h_3(x) = n: & \quad n = \triangleright^{\mathcal{A}}(0 *^{\mathcal{A}} n).
 \end{aligned}$$

Chapter 3 Implicative Theory of Algebraic Identities

The list `required_vars` consists of the decision variables $0 *^A 0$, $0 *^A 1$, $0 *^A 2$, $0 *^A n$, $\triangleright^A(0 *^A 0)$, $\triangleright^A(0 *^A 1)$, $\triangleright^A(0 *^A 2)$, $\triangleright^A(0 *^A n)$. Note that the last 4 decision variables are defined through the first 4.

In Line 4 the number of different variables occurring in $p \equiv q$ is counted and assigned to k . This is used in Line 5 to define an appropriate tuple \mathbf{a} of values. In Lines 6 - 7 we take all subterms s from the identity $p \equiv q$ and check if the corresponding evaluation $s^A(\mathbf{a})$ is already added to `required_variables`. As we take all subterms it may happen that another required variable is nested in the subterm. However, when the nested subterms get assigned the term operation $s^A(\mathbf{a})$ will be “resolved” to an application of a fundamental operation to some values from A , hence, the term operation will be “resolved” to a decision variable. In Line 9 a new constraint for each \mathbf{a} is added.

Input: P .
Output: `required_vars`, `constraints`.

```

1 required_vars ← ∅
2 constraints ← ∅
3 for  $p \equiv q$  in  $P$ :
4    $k \leftarrow$  the number of different variables in  $p \equiv q$ 
5   for  $\mathbf{a}$  in  $(A_{fin} \cup \{n\})^k$ :
6     for  $s$  in  $p, q$ :
7       if not  $s^A(\mathbf{a})$  in required_vars:
8         required_vars.add( $s^A(\mathbf{a})$ )
9         constraints.add( $p^A(\mathbf{a}) = q^A(\mathbf{a})$ )
10 return (required_vars, constraints)

```

Algorithm 3.5: `discover_constraints`

Algorithm is sound The satisfaction of premise is guaranteed via discovering the constraints that arise from the respective identities. As follows from the investigation of the universe A these constraints guarantee the satisfaction of identities from premise. The violation of the conclusion is explicitly checked.

Algorithm is complete The completeness of the algorithm follows from the fact that it suffices to consider algebras with the given structure of A . In the function `exists_next_A` the algorithm iterates over all possible algebras of this type.

Algorithm terminates The function `exists_next_A` always proceeds in the finite search space.

The complexity of the whole algorithm depends on the complexity of `exists_next_A`, the complexity of `discover_constraints`, and the complexity of checking the violation of conclusion.

The complexity of `discover_constraints` is determined by the complexity of three **for** cycles. Let k_{max} denote the maximal number of different variables used in identities (as before, $k_{max} \leq 3$). Let m denote the maximum level of nesting in terms of identities (as before, $m \leq 3$). The number of possible tuples \mathbf{a} is bounded by $O(|A_{fin}|^{k_{max}})$, hence, the overall complexity is bounded by $O(|P|*|A_{fin}|^{k_{max}*m})$.

The number of outputs of the fundamental operations of \mathcal{A} is $O(|A_{fin}|^2)$. The number of possible assignments to each output is $O(|A_{fin}|)$. Hence, there exist $O(|A_{fin}|^{|A_{fin}|^2})$ possible assignments. Assigning to the required variables can be done in the number of operations proportional to the size of the list of required variables, hence, $O(|A_{fin}|^2)$. To check each constraint from `constraints` we need at most $O(|A_{fin}|)$ operations, the same for checking the violation of the conclusion. Hence, the overall complexity of the algorithm is

$$O(|A_{fin}|^{|A_{fin}|^2+3+|P|+|P|*|A_{fin}|^{k_{max}*m}}).$$

Now we proceed with an example of running the function `find_algebra`.

Example 3.3.18. Consider the implication $\{x \equiv \triangleright(x * x), \triangleright a \equiv \triangleright(\triangleright a)\} \rightarrow a \equiv \triangleright a$. The function `discover_constraints` returns the following results:

1. $0 = \triangleright^{\mathcal{A}}(0 *^{\mathcal{A}} 0)$;
2. $1 = \triangleright^{\mathcal{A}}(1 *^{\mathcal{A}} 1)$;
3. $2 = \triangleright^{\mathcal{A}}(2 *^{\mathcal{A}} 2)$;
4. $n = \triangleright^{\mathcal{A}}(n *^{\mathcal{A}} n)$;
5. $n = \triangleright^{\mathcal{A}}(n *^{\mathcal{A}} n)$;
6. $\triangleright^{\mathcal{A}}0 = \triangleright^{\mathcal{A}}(\triangleright^{\mathcal{A}}0)$.

The first assignment to decision variables in function `exists_next_A` leads to the algebra \mathcal{A}_1 presented in Table 3.3. However, $\mathcal{A}_1 \models a \equiv \triangleright a$. Hence, next appropriate assignment is taken, see Table 3.4. The algebra \mathcal{A}_2 from Table 3.4 does not satisfy the identity $a \equiv \triangleright a$ ($\triangleright^{\mathcal{A}_2}0 = 2$), hence, this algebra is the desired counter-example.

Table 3.3: Algebra \mathcal{A}_1 from Example 3.3.18

$$m *^{\mathcal{A}_1} n := \begin{cases} 0 *^{\mathcal{A}_1} 0 = 1; \\ 1 *^{\mathcal{A}_1} 1 = 2; \\ 2 *^{\mathcal{A}_1} 2 = 3; \\ n *^{\mathcal{A}_1} n = n + 1, & \text{if } n \geq 3; \\ m *^{\mathcal{A}} n = 0, & \text{else.} \end{cases}$$

$$\triangleright^{\mathcal{A}_1} n := \begin{cases} \triangleright^{\mathcal{A}_1} 0 = 0; \\ \triangleright^{\mathcal{A}_1} 1 = 0; \\ \triangleright^{\mathcal{A}_1} 2 = 1; \\ \triangleright^{\mathcal{A}_1} n = n - 1, & \text{if } n \geq 3. \end{cases}$$

$$a^{\mathcal{A}_1} := 0.$$

Table 3.4: Algebra \mathcal{A}_2 from Example 3.3.18

$$m *^{\mathcal{A}_2} n := \begin{cases} 0 *^{\mathcal{A}_2} 0 = 1; \\ 1 *^{\mathcal{A}_2} 1 = 3; \\ 2 *^{\mathcal{A}_2} 2 = 4; \\ n *^{\mathcal{A}_2} n = n + 2, & \text{if } n \geq 3; \\ m *^{\mathcal{A}} n = 0, & \text{else.} \end{cases}$$

$$\triangleright^{\mathcal{A}_2} n := \begin{cases} \triangleright^{\mathcal{A}_2} 0 = 2; \\ \triangleright^{\mathcal{A}_2} 1 = 0; \\ \triangleright^{\mathcal{A}_2} 2 = 21; \\ \triangleright^{\mathcal{A}_2} n = n - 2, & \text{if } n \geq 3. \end{cases}$$

$$a^{\mathcal{A}_2} := 0.$$

Table 3.5: A counter-example from Example 3.3.19

$$m *^{\mathcal{A}} n := \begin{cases} 0 *^{\mathcal{A}} 0 = 2; \\ 1 *^{\mathcal{A}} 0 = 1; \\ 2 *^{\mathcal{A}} 0 = 2; \\ m *^{\mathcal{A}} 0 = 1, & \text{if } m \geq 3; \\ m *^{\mathcal{A}} 1 = 0; \\ m *^{\mathcal{A}} n = n + 1, & \text{if } n \geq 2; \\ m *^{\mathcal{A}} n = 0, & \text{else.} \end{cases}$$

$$\triangleright^{\mathcal{A}} n := \begin{cases} \triangleright^{\mathcal{A}} 0 = 1; \\ \triangleright^{\mathcal{A}} 1 = 0; \\ \triangleright^{\mathcal{A}} 2 = 0; \\ \triangleright^{\mathcal{A}} n = n - 1, & \text{if } n \geq 3. \end{cases}$$

$$a^{\mathcal{A}} := 0.$$

Examples of Infinite Counter-examples Below several infinite counter-examples that were found during the investigation are presented.

Example 3.3.19. Implication: $\{a \equiv \triangleright(\triangleright a), a \equiv \triangleright(a * a), a \equiv \triangleright(x * a), x \equiv \triangleright(a * x), x \equiv x * (\triangleright x), x \equiv \triangleright(y * x), x \equiv x, a \equiv x * (\triangleright a), x \equiv \triangleright(x * x), a \equiv a * (\triangleright a)\} \rightarrow \{\triangleright a \equiv a * a\}$.

Compare with Implication (3.6).

A counter-example is the infinite algebra \mathcal{A} presented in Table 3.5.

Example 3.3.20. Implication: $\{a \equiv \triangleright(a * a), a \equiv \triangleright(\triangleright(\triangleright a)), a \equiv \triangleright(x * a), x \equiv \triangleright(a * x), x \equiv x, x \equiv x * (\triangleright a), a \equiv a * (\triangleright a)\} \rightarrow \{x \equiv a * (\triangleright x)\}$. Compare with Implication (3.2).

A counter-example is the infinite algebra \mathcal{A} presented in Table 3.6.

Table 3.6: A counter-example from Example 3.3.20

$$m *^{\mathcal{A}} n := \begin{cases} 0 *^{\mathcal{A}} n = n + 1, & \text{if } n \geq 2; \\ m *^{\mathcal{A}} 0 = 2; \\ m *^{\mathcal{A}} 1 = m; \\ m *^{\mathcal{A}} n = 0, & \text{else.} \end{cases}$$

$$\triangleright^{\mathcal{A}} n := \begin{cases} \triangleright^{\mathcal{A}} 0 = 1; \\ \triangleright^{\mathcal{A}} 1 = 2; \\ \triangleright^{\mathcal{A}} 2 = 0; \\ \triangleright^{\mathcal{A}} n = n - 1, & \text{if } n \geq 3. \end{cases}$$

$$a^{\mathcal{A}} := 0.$$

3.4 Exploration of Algebraic Identities

In AE of algebraic identities two methods for finding counter-examples were used: finite counter-examples were found using **Mace4**, infinite counter-examples were found Algorithm 3.3. Moreover, before finding counter-examples the program **Prover9** made an attempt to prove implications. If a proof for an implication was found the attribute exploration proceeds with the next implication. The initial context contained all irreducible algebras over a universe of size 2.

3.5 Results

Attribute Exploration of the algebraic identities from Table 3.7 was run on a computer with Intel Core i5 1.6GHz×4 processor and 6 Gb of RAM running Linux Ubuntu 12.10 x64. In order to speed up the procedure the time limits for finding each separate counter-example were changed in the process of exploration, this information is contained in the attachment to the current chapter in file “progress.txt”.

Attribute Exploration took 30 steps (understanding a step as the process of finding the implication basis and trying to find counter-example to each implication from the basis). 1771 finite and 3179 infinite counter-examples were found during the exploration. After reduction 626 finite algebras and 1529 infinite algebras were left in the context. All 4398 unit implications from the canonical basis were proved by *Prover9*. Results of investigation [Kes13] were repeated using only software tools. The exploration took around two weeks.

Note that the procedure was not optimized to perform the exploration in the shortest possible time as this was not the objective of current investigation. For example, choosing more appropriate time limits for finding counter-examples can speed up the process significantly. In another experiment the final context was explored in around 78 hours.

3.6 Conclusion

The aim of the current chapter was to elaborate methods and tools in order to automatically discover and prove the implicative theory of algebraic identities of size up to 5. It turns out that it is necessary to be able to find infinite counter-examples. The structure of such infinite algebras is investigated and, based on this investigation, a computational model is introduced. The computational model is capable of finding all the needed infinite counter-examples. The developed software satisfies all the introduced requirements and is able to produce results within several days on a modern computer.

3.7 Remarks

1. Our approach is different from that of Dr. Kestler in [Kes13] insofar as Dr. Kestler considered only injectivity and surjectivity to prove the absence of finite counter-examples.¹ For special term operations he stated they are necessarily injective but not surjective, and vice versa. But our characterization is more precise. Indeed, consider Implication (3.6). Suppose $a^{\mathcal{A}} = 0$ and $0 *^{\mathcal{A}} 0 = 0$, $n *^{\mathcal{A}} n = n + 1$ ($q^{\mathcal{A}}(x) = x *^{\mathcal{A}} x$). $x \equiv \triangleright(x * x)$ yields $\triangleright^{\mathcal{A}} 0 = 0$. The term operation $x *^{\mathcal{A}} x$ is injective, but not surjective. However, there exists no term operation $\triangleright^{\mathcal{A}} x$ such that $\triangleright^{\mathcal{A}} 0 \neq 0 *^{\mathcal{A}} 0$. Therefore, not just any injective and not surjective term operation $q^{\mathcal{A}}(x) = x *^{\mathcal{A}} x$ can lead to a counter-example.

Another example: consider Implication (3.5) and $a^{\mathcal{A}} = 0$, $q^{\mathcal{A}}(0) = 0 *^{\mathcal{A}} 0 = 1$, $q^{\mathcal{A}}(1) = 1 *^{\mathcal{A}} 1 = 0$, $q^{\mathcal{A}}(n) = n + 1$. Such $q^{\mathcal{A}}(x)$ is injective and not surjective, but no matter how you define the other operations, there is no counter-example with such $q^{\mathcal{A}}$.

2. Whether the statement converse to Lemma 3.3.14 holds is an open question. Namely, it is not clear if it always suffices to consider only the algebras with the described structure of universe. However, as was proven by experiment, for the chosen identities these algebras are sufficient.

¹Besides the automation of the whole investigation.

Appendix to Chapter 3

Table 3.7: Pairwise non-equivalent identities of size at most 5.

	$x \equiv a * x;$	$a \equiv \triangleright(x * a);$	$x \equiv (\triangleright y) * x;$
	$x \equiv x * a;$	$a \equiv \triangleright(x * x);$	$x \equiv \triangleright(a * x);$
	$x \equiv x * x;$	$a \equiv \triangleright(x * y);$	$x \equiv \triangleright(x * a);$
Size 2:	$x \equiv x * y;$	$\triangleright a \equiv \triangleright(\triangleright a);$	$x \equiv \triangleright(x * x);$
$x \equiv x;$	$x \equiv y * x;$	$\triangleright a \equiv \triangleright(\triangleright x);$	$x \equiv \triangleright(x * y);$
$x \equiv y;$	Size 5:	$\triangleright a \equiv a * a;$	$x \equiv \triangleright(y * x);$
Size 3:	$a \equiv \triangleright(\triangleright(\triangleright a));$	$\triangleright a \equiv a * x;$	$\triangleright x \equiv \triangleright(\triangleright x);$
$a \equiv \triangleright a;$	$a \equiv \triangleright(\triangleright(\triangleright x));$	$\triangleright a \equiv x * a;$	$\triangleright x \equiv a * a;$
$a \equiv \triangleright x;$	$a \equiv a * (\triangleright a);$	$\triangleright a \equiv x * x;$	$\triangleright x \equiv a * x;$
$x \equiv \triangleright x;$	$a \equiv a * (\triangleright x);$	$\triangleright a \equiv x * y;$	$\triangleright x \equiv a * y;$
Size 4:	$a \equiv (\triangleright a) * a;$	$x \equiv \triangleright(\triangleright(\triangleright x));$	$\triangleright x \equiv x * a;$
$a \equiv \triangleright(\triangleright a);$	$a \equiv (\triangleright a) * x;$	$x \equiv a * (\triangleright x);$	$\triangleright x \equiv x * x;$
$a \equiv \triangleright(\triangleright x);$	$a \equiv x * (\triangleright a);$	$x \equiv x * (\triangleright a);$	$\triangleright x \equiv x * y;$
$a \equiv a * a;$	$a \equiv x * (\triangleright x);$	$x \equiv x * (\triangleright x);$	$\triangleright x \equiv y * a;$
$a \equiv a * x;$	$a \equiv x * (\triangleright y);$	$x \equiv x * (\triangleright y);$	$\triangleright x \equiv y * x;$
$a \equiv x * a;$	$a \equiv (\triangleright x) * a;$	$x \equiv y * (\triangleright x);$	$\triangleright x \equiv y * y;$
$a \equiv x * x;$	$a \equiv (\triangleright x) * x;$	$x \equiv (\triangleright a) * x;$	$\triangleright x \equiv y * z.$
$a \equiv x * y;$	$a \equiv (\triangleright x) * y;$	$x \equiv (\triangleright x) * a;$	
$\triangleright a \equiv \triangleright x;$	$a \equiv \triangleright(a * a);$	$x \equiv (\triangleright x) * x;$	
$x \equiv \triangleright(\triangleright x);$	$a \equiv \triangleright(a * x);$	$x \equiv (\triangleright x) * y;$	

Conclusion

In the frames of the current project general approaches to automatic constructions of implicative theories for mathematical domains are investigated on two applications. The methodology of investigation is based on discovering knowledge from (counter-)examples – the procedure of Attribute Exploration. The relevant software tools for performing Attribute Exploration automatically are implemented and analyzed. The implementation is independent of the domain of application, hence, may be used for further explorations.

In both applications we succeeded in automatizing the process of exploration of the implicative theories. This goal is achieved thanks to both the pragmatical approach of Attribute Exploration and discoveries in the respective domains of application. The structure of the attributes in the investigated domain is the key to developing efficient methods of finding counter-examples – the core of Attribute Exploration.

In the current study the methods and algorithms for finding both finite and infinite counter-examples are investigated and developed. In the case of infinite counter-examples the preliminary knowledge about the structure of the desired solution is essential for finding the counter-examples. In the case of finite counter-examples it was necessary to develop competitive algorithms implementing different computational strategies. The parallel run of the algorithms allows for finding counter-examples in limited time.

In the case of p -indecomposable functions it is necessary to extend the procedure of Attribute Exploration and to investigate the extended version. It turned out that in case of a growing number of attributes it is difficult to state any assumptions about the overall success of the exploration. However, we succeed in the exploration of p -indecomposable functions on the three-valued domain, and the developed tools and methods may be used for an exploration on even larger domains.

The two diverse application domains favourably illustrate different possible usage patterns of Attribute Exploration – in one case the number of attributes is fixed, however, counter-examples are infinite, in the other case the number of attributes

Conclusion

grow, but the counter-examples are finite. The elaborated approaches may be further developed and used not only for constructing the complete implicative theories, but also for a more widespread problem of finding counter-examples to certain implications.

The choice of mathematical domains as domains of application is justified by the fact that the mathematical statements considered in the project are either true or false. Moreover, it is possible to generate the desired counter-examples algorithmically. Real life is usually more complex, the truth of many statements is argued and in order to find a counter-example it is necessary to involve experts from the respective domain. The current investigation (together with methods for finding mistakes in object intents [KR15, Rev13b]) may be seen as the first step to elaborating a system for organizing the knowledge and assisting experts from a domain in working with this knowledge.

Description of Attachments

The current study is accompanied by attachments presenting the result of the conducted experiments.

Attachment to Chapter 2

Attachment to Chapter 2 is contained in the folder “Ch_PClones”. The folder contains the following files:

final_cxt.txt File contains the final context in .txt format.

final_cxt.cxt File contains the final context in .cxt format.

progress.txt File contains the report about the progress of the experiment. Information about the number of the step, number of objects and attributes in the current context, number of processed implication and information about time is presented. Moreover, new function and reason for finding it (either an implication violated by the new function or respective concept from the current lattice) are presented. The format of representation of functions is compatible with arguments to Python scripts, i.e. if respective package is imported by the Python interpreter than the representation will be understood by the interpreter and the desired object will be returned. In implications the symbol “=>” separates premise from conclusion.

step(*number*)[ces/new_objs].txt File contains information about the respective step of the exploration. Files with names ending with “ces.txt” stand for the steps of finding counter-examples, files ending with “new_objs.txt” stand for the steps of finding new first-order irreducible objects. Time spent on each implication or concept is also presented. Each file contains the current context and information about processing implications or concepts of the current lattice. Time spent on each implication or concept is given.

Attachment to Chapter 3

Attachment to Chapter 3 is contained in the folder “Ch_Identities”. The structure of the folder is very similar to the structure of “Ch_PClones”, except for an additional folder “proofs”. The folder “Ch_Identities” contains:

final_cxt.txt File contains the final context in .txt format.

final_cxt.cxt File contains the final context in .cxt format.

progress.txt File contains the report about the progress of the experiment. Information about the number of the step, number of objects and attributes in the current context, number of processed implication and information about time is presented. Moreover, new function and reason for finding it (either an implication violated by the new function or respective concept from the current lattice) are presented. The format of representation of functions is compatible with arguments to Python scripts, i.e. if respective package is imported by the Python interpreter than the representation will be understood by the interpreter and the desired object will be returned. In implications the symbol “=>” separates premise from conclusion.

step(number)ces.txt File contains information about the respective step of the exploration. Each file contains the current context and information about processing implications from the current implication basis. Time spent on each implication is given.

The folder “Ch_Identities/proofs” contains:

imp(number).in File contains an implication represented as an input to Prover9.

imp(number).prover9.out File contains the output of Prover9 with a proof of the respective implication.

basis.txt File contains the canonical implication basis of the final context.

Symbols and Conventions

The following conventions are used:

- As usual, $s \in S$ indicates that s belongs to S , $P \subseteq S$ indicates that P is contained in or equal to S .
- The statement $P \subset S$ is equivalent to $P \subseteq S$ and $P \neq S$.
- We usually represent sets in curly brackets and separate elements via commas, e.g. $S = \{s_1, s_2, \dots\}$. If a set consists of a single element we will usually omit the curly brackets, e.g. $\{s_1\}$ is the same as s_1 . If ambiguity is excluded we sometimes omit the curly brackets even if a set contains more than one element.
- The symbol \mathbb{N} denotes the set of natural numbers (including 0).
- For $k, n \in \mathbb{N}, k \leq n$ the set $\{k, k+1, \dots, n\}$ is written as $[k, n]$.
- $|S|$ denotes the cardinality of set S .
- We use the notation $A_k, k \in \mathbb{N}$ to indicate that A_k has exactly k elements; instead of the elements of A_k we will usually use their indices, i.e. $A_k = \{0, 1, \dots, k-1\}$ and $|A_k| = k$.
- Cartesian product of sets A and B , denoted $A \times B$, is the set of all pairs $\{(a, b) \mid a \in A, b \in B\}$.
- For a set A and a number $n \in \mathbb{N}$ we use the notation A^n to denote the n -th cartesian power of A , i.e. $A^n = \underbrace{A \times A \times \dots \times A}_n$.
- Notation $f : A \rightarrow B$ denotes a mapping f from A to B .
- For two sets A, X the notation A^X denotes the set of all possible mappings from X to A , i.e. $A^X = \{f : X \rightarrow A\}$.
- A function f of arity $ar(f) \in \mathbb{N}$ on A is a mapping $f : A^{ar(f)} \rightarrow A$.
- If S is a set, $\mathcal{P}(S)$ denotes the set of all subsets of S , i.e. the powerset.

Symbols and Conventions

- A relation R of arity $ar(R) \in \mathbb{N}$ on A is a subset of $A^{ar(R)}$.
- We denote n consecutive applications of a unary term p by p^n . For unary terms p, q we denote $p(q(x))$ by $pq(x)$.
- A restriction of a function $f : A^{ar(f)} \rightarrow A$ to a set $B \subseteq A$ is a function $f|_B : B^{ar(f)} \rightarrow B$ such that for all $b \in B$: $f|_B(b) = f(b)$.
- Let $f : A \rightarrow B$ and $b = f(a)$. The element b is called the image of a under f . The element a is called the preimage of b under f .
- The notation $A \Rightarrow B$ is used to substitute the (natural language) expression “from A follows B ”.
- The expression “iff” is used as a shorthand for “if and only if”.
- In mathematical expressions “.” is to be interpreted as “such that”.
- The notation $A \Leftrightarrow B$ is used to substitute the (natural language) expression “ A is equivalent to B ”.
- $A \cap B$ denotes the intersection of sets A and B , i.e. the set of all elements contained in both A and B ; $A \cup B$ denotes the union of sets A and B , i.e. the set of all elements contained in A or B ; $A \setminus B$ denotes the difference of A and B , i.e. the set of all elements which are in A but not in B .

Index

- algebra, *see also* logic, L -structure
 - free, 76
 - locally finite, 76
- arity, 9
- assignment function
 - term, 10
 - variable, 10
- atom, 4
- Attribute Exploration, 13
- attributes, 5

- binary relation, 3
- bunny, 76

- clone
 - functional, 24
 - p-, parametric, 24
- closure, 5
 - operator, 5
 - system, 6
- commutation, 24
- compatible, 23
- complexity of algorithm, 18
- concept
 - attribute, 6
 - lattice, 6
 - object, 6
- consequence, *see* logic, logically imply
- constants, 9
- counter-example, i, 12

- essential, 58

- decision variable, 18
- dense
 - infimum-, 4
 - supremum-, 4

- element
 - greatest, 3
 - least, 3
- equational class, 73, 76
- expressible
 - compositionally-, 23
 - parametrically-, 24

- formal concept, 5
 - extent, 5
 - intent, 5
 - order, 6
- formal context, ii, **5**
 - reduced, 7
- fundamental operation, 75

- Galois connection, 5
- graph of function, 24

- homomorphism, 75

- identity, *see also* logic, L -formula
 - pairwise equivalent, 75
 - satisfaction, *see* logic, L -formula, satisfaction

Index

- size, 75
- implication, i, 10
 - attribute, 11
 - basis, 12
 - canonical, 13
 - conclusion, 11
 - premise, 11
 - respected, 11
 - satisfaction, 12
 - unit, 12
 - validity, 12
 - violation, 12
- implicative theory, i, **12**
- infimum, 3
- irreducible, 7
 - first-order, 53
 - infimum-, 4
 - plainly, 7
 - second-order, 55
 - supremum-, 4
- lattice, 3
 - concept-, 6
 - filter, 4
 - ideal, 4
- logic
 - alphabet, 8
 - attribute, 11
 - first order language, 9
 - inference rule, 11
 - L -formula, 9
 - atomic, 10
 - satisfaction, 10
 - validity, 10
 - logically imply, 11
 - L -structure, 10
 - finite, 10
 - infinite, 10
 - interpretation, 10
 - universe, 10
 - L -term, 9
 - size, 75
 - propositional, 11
 - formula, 11
 - interpretation, 11
 - values, 11
 - theory, i
- neighbor
 - lower, 4
 - upper, 4
- Object-Attribute Exploration, 50
- objects, 5
 - reducible, 7
- order, 3
- ordered diagram, 7
- p -indecomposable function, 24
- parametrically equivalent, 24
- preserve, 23
- projections, 23
- subalgebra, 75
- subcontext, 5
 - implicatively closed, 57
- success of exploration, ii, 26
- supremum, 3
- symbols
 - function, 9
 - relation, 9
 - variable, 9
- term, *see* logic, L -term
 - operation, 75
 - pairwise equivalent, 75
- universe, *see* logic, L -structure, uni-verse

Bibliography

- [AdL13] A. Albano and A.P. do Lago. A convexity upper bound for the number of maximal bicliques of a bipartite graph. *Discrete Applied Mathematics*, 2013.
- [Arm74] W.W. Armstrong. Dependency structures of data base relationships. In *IFIP Congress*, pages 580–583, 1974.
- [AS07] J. Adler and J. Schmid. Introduction to mathematical logic. *University of Bern*, 2007.
- [Bar82] J Barwise. *Handbook of mathematical logic*. Elsevier, 1982.
- [BDF⁺03] V.G. Blinova, D.A. Dobrynin, V.K. Finn, S.O. Kuznetsov, and E.S. Pankratova. Toxicology analysis by means of JSM-method. *Bioinformatics*, 19:1201–1207, 2003.
- [BFdNT07] P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing finite models by reduction to function-free clause logic. *Journal of Applied Logic*, 2007.
- [BFR72] K.A. Baker, P.C. Fishburn, and F.S. Roberts. Partial orders of dimension 2. *Networks*, 2(1):11–28, 1972.
- [BHSS89] H.-J. Bürkert, A. Herold, and M. Schmidt-Schauss. On equational theories, unification, and (un)decidability. *Journal of Symbolic Computation*, 8(1):3–49, 1989.
- [Bir35] G. Birkhoff. On the structure of abstract algebras. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 31, pages 433–454. Cambridge Univ Press, 1935.
- [Bir67] G. Birkhoff. *Lattice Theory*. Am. Math. Soc., Providence, RI, 3rd edition, 1967.
- [BKKR69] V.G. Bodnarchuk, L.A. Kaluzhnin, V.N. Kotov, and B.A. Romov. Galois theory for post algebras. i-ii. *Cybernetics*, 5(3):243–252, 1969.

Bibliography

- [BS81] S. Burris and H.P. Sankappanavar. *A course in universal algebra*, volume 78. Springer-Verlag New York, 1981.
- [BW87] S. Burris and R. Willard. Finitely many primitive positive clones. *Proceedings of the American Mathematical Society*, 101(3):427–430, 1987.
- [CDFR08] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux. Formal concept analysis enhances fault localization in software. In R. Medina and S. Obiedkov, editors, *ICFCA*, volume 4933 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2008.
- [Chu96] A. Church. *Introduction to Mathematical Logic*, volume 13. Princeton University Press, 1996.
- [CS] K. Claessen and N. Sörensson. Paradox 1.0. <http://www.cs.miami.edu/~tptp/CASC/19/SystemDescriptions.html#Paradox---1.0>.
- [CT51] L.H. Chin and A. Tarski. *Distributive and modular laws in the arithmetic of relation algebras*, volume 1. University of California Press, 1951.
- [Dan77] A.F. Danil’chenko. Parametric expressibility of functions of three-valued logic. *Algebra and Logic*, 16(4):266–280, 1977.
- [Dau00] F. Dau. Implications of properties concerning complementation in finite lattices. In: *Contributions to General Algebra 12 (D. Dorninger et al., eds.), Proceedings of the 58th workshop on general algebra “58. Arbeitstagung Allgemeine Algebra”, Vienna, Austria, June 3-6, 1999, Verlag Johannes Heyn, Klagenfurt*, pages 145–154, 2000.
- [Deh11] M. Dehn. Über unendliche diskontinuierliche Gruppen. *Mathematische Annalen*, 71(1):116–144, 1911.
- [DP02] B.A. Davey and H.A. Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.
- [Fre04] R. Freese. Automated lattice drawing. In *Concept Lattices*, pages 112–127. Springer, 2004.
- [Gan07] B. Ganter. Relational galois connections. *Formal Concept Analysis*, pages 1–17, 2007.
- [Gan10] B. Ganter. Two basic algorithms in concept analysis. In L. Kwuida and B. Sertkaya, editors, *Formal Concept Analysis*, volume 5986 of *Lecture Notes in Computer Science*, pages 312–340. Springer Berlin Heidelberg, 2010.

Bibliography

- 2010.
- [GD86] J.-L. Guigues and V. Duquenne. Familles minimales d'implications informatives résultant d'un tableau de données binaires. *Math. Sci. Hum.*, 24(95):5–18, 1986.
- [Grä03] G. Grätzer. *General Lattice Theory*. Springer, 2003.
- [GSW05] B. Ganter, G. Stumme, and R. Wille, editors. *Formal Concept Analysis, Foundations and Applications*, volume 3626 of *Lecture Notes in Computer Science*. Springer, 2005.
- [GV05] R. Godin and P. Valtchev. Formal concept analysis-based class hierarchy design in object-oriented software development. In Ganter et al. [GSW05], pages 304–323.
- [GW99a] B. Ganter and R. Wille. Contextual attribute logic. In *Proceedings of the 7th International Conference on Conceptual Structures: Standards and Practices*, ICCS '99, pages 377–388, London, UK, UK, 1999. Springer-Verlag.
- [GW99b] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.
- [HM88] D. Hobby and R. McKenzie. *The structure of finite algebras*. American Mathematical Society Providence, 1988.
- [IK08] D.I. Ignatov and S.O. Kuznetsov. Concept-based recommendations for internet advertisement. In: *Belohlavek R., Kuznetsov S.O., Eds., Proc. 6th International Conference on Concept Lattices and Their Applications (CLA 2008)*, pages 157–167, 2008.
- [Kes13] P. Kestler. *Strukturelle Untersuchungen eines Varietätenverbandes von Gruppoiden mit unärer Operation und ausgezeichnetem Element*. PhD thesis, TU Bergakademie, Freiberg, 2013.
- [Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.
- [KO02] S.O. Kuznetsov and S.A. Obiedkov. Comparing performance of algorithms for generating concept lattices. *Journal of Experimental & Theoretical Artificial Intelligence*, 14(2-3):189–216, 2002.
- [KPR06] L. Kwuida, C. Pech, and H. Reppe. Generalizations of boolean algebras. an attribute exploration. *Mathematica Slovaca*, 56(2):145–165, 2006.

Bibliography

- [KR15] Sergei O. Kuznetsov and Artem Revenko. Interactive error correction in implicative theories. *International Journal of Approximate Reasoning*, 63:89 – 100, 2015.
- [KS05] S.O. Kuznetsov and M.V. Samokhin. Learning closed sets of labeled graphs for chemical applications. In: *Proc. 15th Conference on Inductive Logic Programming (ILP 2005), Lecture Notes in Artificial Intelligence (Springer)*, 2635:190–208, 2005.
- [Kuz79] A.V. Kuznetsov. Means for detection of nondeducibility and inexpressibility. *Logical Inference*, pages 5–33, 1979.
- [Löw15] L. Löwenheim. Über Möglichkeiten im Relativkalkül. *Mathematische Annalen*, 76(4):447–470, 1915.
- [McC10] W. McCune. Prover9 and mace4. <http://www.cs.unm.edu/~mccune/prover9/>, 2005–2010.
- [OD03] S. Obiedkov and V. Duquenne. Incremental construction of the canonical implication basis. *Proc. of JIM 2003, Metz, France*, 2003.
- [Per67] P. Perkins. Unsolvable problems for equational theories. *Notre Dame Journal of Formal Logic*, 8(3):175–185, 1967.
- [Pos42] E.L. Post. *The two-valued iterative systems of mathematical logic*. Princeton University Press, 1942.
- [Rev13a] A. Revenko. Debugging program code using implicative dependencies. In *FCA4AI 2013*, pages 27–39. CEUR, 2013.
- [Rev13b] A. Revenko. Detecting mistakes in binary data tables. *Automatic Documentation and Mathematical Linguistics*, 47(3):102–110, 2013.
- [Rev14] A. Revenko. Automated construction of implicative theory of algebraic identities of size up to 5. In Cynthia Vera Glodeanu, Mehdi Kaytoue, and Christian Sacarea, editors, *Formal Concept Analysis*, volume 8478 of *Lecture Notes in Computer Science*, pages 188–202. Springer International Publishing, 2014.
- [Rev15] A. Revenko. Finding p-indecomposable function: FCA approach. In *CLA 2015*, submitted 2015.
- [RK12] A. Revenko and S.O. Kuznetsov. Attribute exploration of properties of functions on sets. *Fundamenta Informaticae*, 115(4):377–394, 2012.
- [Ros70] I. Rosenberg. *Über die funktionale Vollständigkeit in den mehrwertigen*

Bibliography

- Logiken: Struktur der Funktionen von mehreren Veränderlichen auf endlichen Mengen.* Academia, 1970.
- [ST98] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. *SIGSOFT Softw. Eng. Notes*, 23(6):99–110, November 1998.
- [Str15] F. Strok. Applying FCA toolbox to software testing. *arXiv preprint arXiv:1505.01367*, 2015.
- [Tar41] A. Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(3):73–89, 1941.
- [Tay79] W. Taylor. *Equational logic*. University of Houston, Department of Mathematics, 1979.
- [Wil05] R. Wille. Formal concept analysis as mathematical theory of concepts and concept hierarchies. In Ganter et al. [GSW05], pages 1–33.
- [Yab60] S.V. Yablonsky. *Functional Constructions in K-valued Logic*. U.S. Joint Publications Research Service, 1960.
- [YM59] Yu.I. Yanov and A.A. Muchnik. On the existence of k-valued closed classes that have no bases. *Doklady Akademii Nauk SSSR*, 127:44–46, 1959.
- [Zsc07] C. Zschalig. An fdp-algorithm for drawing lattices. In P. W. Eklund, J. Diatta, and M. Liquiere, editors, *Proceedings of the Fifth International Conference on Concept Lattices and Their Applications, CLA 2007, Montpellier, France, October 24-26, 2007*, volume 331 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.
- [Pe13] А. Ревенко. *Построение имплекативных зависимостей для аналитического описания предметных областей и обнаружения ошибок в данных*. PhD thesis, НИУ Высшая школа экономики, 2013.