DEPARTAMENTO DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE ABERTA

UNIVERSIDADE
AbERTA
www.uab.pt

Computing Congruences and Endomorphisms for Algebras of Type $(2^m, 1^n)$

Rui Miguel Barradas Pereira

PhD Thesis in Computational Algebra

Supervised by:
João Araújo (Universidade Nova de Lisboa / Universidade Aberta)
Wolfram Bentz (Universidade Aberta)
Luis Sequeira (Universidade de Lisboa)

April 2022

Acknowledgements

I would like to thank my supervisor João Araújo, for his support and guidance during the last years. I also thank the support of my co-supervisors Wolfram Bentz, whose support in the review of the results and resulting documentation was unvaluable, and Luís Sequeira whose support in identifying some last minute problems was especially helpful.

I thank also the collaboration of Choiwah Chow that contributed with the implementation of the automorphism algorithm for the **CREAM** package and was always very prompt in providing support when needed.

Finally I thank my wife, Ana, and my children Guilherme and Rafael for their patience and support during these last years.

UNIVERSIDADE
AbERTA
www.uab.pt

DECLARAÇÃO DE INTEGRIDADE

STATEMENT OF INTEGRITY

Declaro ter atuado com integridade na elaboração da presente dissertação/tese. Confirmo que em todo o trabalho conducente à sua elaboração não recorri à prática de plágio ou a qualquer outra forma de falsificação de resultados.

Mais declaro que tomei conhecimento integral do Regulamento Disciplinar da Universidade Aberta, publicado no Diário da República, 2.ª série, n.º 215, de 6 de novembro de 2013.

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.

I further declare that I have fully acknowledged Disciplinary Regulations of the Universidade Aberta (regulation published in the official journal Diário da República, 2.ª série, N.º 215, de 6 de novembro de 2013).

Universidade Aberta, 06 de Abril de 2022

Nome completo/Full name: Rui Miguel Barradas Pereira

Assinatura/Signature:

[illegible]

manuscrita ou digital / handwritten or digital

Resumo

Atualmente a principal aplicação de software para semigrupos é a package de GAP chamada Semigroups [29], em articulação com a package Smallsemi [13]. O GAP [17] é um sistema e linguagem de programação para álgebra discreta computacional. Embora estas packages ofereçam muitas opções de cálculo e forneçam uma biblioteca de todos os semigrupos até o tamanho 8, várias operações de semigrupos importantes não estão disponíveis. Em parte, isto deve-se ao facto de a arquitetura subjacente ser voltada para a teoria de grupos e os semigrupos frequentemente requerem técnicas algorítmicas que são mais próximas das empregadas na Álgebra Universal do que daquelas usadas em grupos.

Existem maneiras de construir álgebras a partir das existentes (produtos diretos etc.), mas a operação inversa é crítica: decompor uma dada álgebra em outras menores. Este tipo de decomposição é especialmente importante em semigrupos, pois mesmo a estrutura de semigrupos muito pequenos pode ser muito obscura.

Até agora não existe uma ferramenta computacional geral para decompor semigrupos. Por exemplo, o GAP já contém código para encontrar todas as congruências de classes muito particulares de semigrupos, mas está muito longe de fornecer um método geral. A situação é ainda pior em relação aos endomorfismos. O objetivo da package **CREAM** (Algebra ConGRuences, Endomorphisms and AutomorphisMs) é resolver esta situação implementando algoritmos eficientes para calcular congruências, endomorfismos e automorfismos de álgebras do tipo $(2^m, 1^n)$. Vários algoritmos gerais (como em [15]) serão adaptados para o contexto da teoria de semigrupos e mais geralmente para álgebras do tipo $(2^m, 1^n)$ cobrindo assim grupos e semigrupos mas também álgebras unárias e também loops, campos, anéis, semi-anéis, álgebras de Lie, MV-álgebras, Meadows, álgebras de lógica, etc. Estes algoritmos serão implementados em GAP. Isso incluirá as seguintes questões:

- Dada uma álgebra finita A do tipo $(2^m, 1^n)$, encontrar todas as congruências de A, pelo menos tão rápido quanto os programas existentes (ou seja, o código produzido será geral e pelo menos tão rápido quanto o código que existe para classes específicas de álgebras apenas);
- Dada uma álgebra finita A do tipo $(2^m, 1^n)$, encontrar todos os endomorfismos de A; em particular, o código deve calcular efetivamente os automorfismos de A, uma

ferramenta que essencialmente existe apenas para grupos.

Uma álgebra universal é uma estrutura algébrica que consiste num conjunto A e numa coleção de operações sobre A. Uma operação n-aria sobre A é uma função que tem como entrada um n-tuplo de elementos de A e retorna um elemento de A. Neste contexto só estão a ser consideradas operações com aridade 1 ou 2, i.e. operações unárias e binárias. Uma álgebra do tipo $(2^m, 1^n)$ é uma álgebra universal com m operações binárias e n unárias.

Este uso genérico da package **CREAM** depende de teoremas de álgebra universal. Isto tem custos, pois teoremas de tipos específicos de álgebras (e.g. grupos, semigrupos, etc) não podem ser usados para reduzir o espaço de procura e melhorar a performance. Canon e Holt [10], usaram vários algoritmos inteligentes e teoremas específicos de grupos para produzir código mais rápido que a package **CREAM** a calcular automorfismos de grupos com ordens maiores. Analogamente, Mitchell et al. [29] usaram teoremas da teoria de semigrupos para calcular eficazmente automorfismos e congruências de semigrupos completamente 0-simples. Mas estes estão entre os poucos casos de código GAP disponível que é mais rápido que a package **CREAM** que é de uso mais generalizado. Para a maioria de outras classes de álgebras de tipo $(2^m, 1^n)$ a package **CREAM** é mais rápida a calcular auto[endo]morfismos/congruências. Um dos algoritmos principais implementado na package é o algoritmo de Freese descrito em [15] que calcula congruências principais para uma álgebra universal.

Para chegar a esta performance a package **CREAM** usa uma mistura de algoritmos standard de álgebra universal juntamente com ferramentas de procura eficiente por modelos finitos de fórmulas de primeira ordem e a implementação de partes de código em C.

Em geral, calcular congruências de álgebras é uma tarefa difícil. Existem vários teoremas descritivos para diferentes tipos de álgebra e algumas ferramentas computacionais para calcular congruências para álgebras finitas, mas geralmente essas ferramentas são aplicáveis a um conjunto muito específico e estreito de álgebras. O nosso objetivo era fornecer uma ferramenta eficaz para calculá-los para álgebras finitas do tipo $(2^m, 1^n)$ abrangendo um conjunto muito amplo de álgebras, incluindo a maioria dos tipos e classes de álgebra atualmente estudados. O algoritmo usado foi o algoritmo de Freese cuja eficiência depende em muito da representação das álgebras e congruências. Em especial a representação de partições/congruências como um array é determinante na eficiência do cálculo das congruências principais uma vez que permite uma junção de blocos muito rápida, uma das operações mais utilizadas durante a execução do algoritmo de congruências. Além disso a representação usada não limita o âmbito geral das álgebras suportadas.

No âmbito deste doutoramento vários algoritmos para o cálculo de automorfismos foram testados e a conclusão foi que o algoritmo mais promissor foi um algoritmo que usa invariantes das operações das álgebras para limitar os possíveis automorfismos da álgebra. Este algoritmo é usado na package Loops cuja implementação foi usada como guia tendo o seu âmbito sido expandido para suportar não só magmas mas qualquer álgebra do tipo $(2^m, 1^n)$. A implementação final dos algoritmos de automorfismos usando a abordagem de invariantes foi contribuída para a package **CREAM** por Choiwah Chow com base nas conclusões e no trabalho realizado no âmbito deste doutoramento.

Dada a falta de referências sobre algoritmos para calcular endomorfismos, o algoritmo para calcular endomorfismos usado na package **CREAM** foi definido usando teoremas básicos de álgebra universal como o teorema do homomorfismo, para relacionar congruências, álgebras quocientes, subálgebras e endomorfismos. O algoritmo foi desenvolvido usando os algoritmos de congruências e automorfismos implementados, e uma aplicação denominada **MACE4**. O **MACE4** [27] é uma aplicação de linha de comando que procura modelos finitos de fórmulas de primeira ordem.

Embora o GAP seja adequado para prototipagem e implementação rápida de algoritmos, o código resultante não é muito rápido devido ao facto de ser uma linguagem interpretada. Reescrever o código em C permitiu melhorias surpreendentes que alcançaram uma melhoria de até 670 vezes do código GAP para o código C.

Além disso, a integração com o **MACE4** permite combinar a eficiência de algoritmos como [15], com um amplo conjunto de possibilidades fornecidas por uma ferramenta eficiente na procura por modelos finitos de fórmulas de primeira ordem, dando uma flexibilidade muito grande à package **CREAM**.

A package **CREAM** é em média 20 vezes mais rápida no cálculo de congruências dos tipos de semigrupos muito limitados que são suportados pela função **Congruence-sOfSemigroups** da package **Semigroups** que é a função mais abrangente em GAP para o cálculo de congruências. A única aplicação que possui um âmbito semelhante em termos de álgebras suportadas é a interface de linha de comando **UACalc**, mas faz isso em jython, sem beneficiar do ecossistema existente na plataforma GAP. Quando comparado com o **UACalc**, o package **CREAM** é consistentemente mais de 3 vezes mais rápido.

Relativamente a automorfismos e endomorfismos, a comparação com outras bibliotecas/aplicativos é muito difícil, uma vez que o suporte é limitado principalmente a grupos e outras estruturas algébricas intimamente relacionadas com grupos. Para essas álgebras, o desempenho das packages GAP **Loops** e **Sonata** são comparáveis e às vezes melhores do que a package **CREAM**. Essas bibliotecas por vezes contam com o uso de teoremas específicos para essas álgebras. Mas apenas a package **CREAM** suporta a maioria das álgebras estudadas em álgebra convencional e moderna.

Dada a importância das congruências, automorfismos e endomorfismos para o estudo de estruturas algébricas, espera-se que a package **CREAM** com seu desempenho e versatilidade possa ser uma ferramenta útil para a comunidade GAP e um amplo grupo de matemáticos.

O código resultante está disponível como o pacote GAP **CREAM** que está totalmente documentado.

Palavras-chave: Congruências, Morfismos, Universal, Álgebra, GAP

Summary

While there are efficient algorithms to decompose very particular classes of semigroups, groups and quasigroups, there are no similar facilities available for the more general algebraic structures such as algebras of type $(2^m, 1^n)$, with an arbitrary number of binary and unary operations. In this thesis, its presented the GAP package **CREAM** (Algebra CongRuences, Endomorphisms and AutomorphisMs) that provides efficient algorithm implementations to calculate congruences, endomorphisms and automorphisms of algebras of type $(2^m, 1^n)$ covering groups and semigroups but also unary algebras, and loops, fields, rings, semi-rings, MV-algebras, Meadows, algebras of logic, etc.

An universal algebra is an algebraic structure consisting of a set A together with a collection of operations on A. An n-ary operation on A is a function that takes n-tuple of elements from A and returns a single element of A. In the current scope are only considered operations with arity of 1 or 2, i.e. unary and binary operations. An algebra of type $(2^m$,$1^n)$ is a universal algebra with m binary and n unary operations.

This general applicability of the **CREAM** package relies on universal algebra theorems. This comes with a cost since theorems on specific types of algebras (e.g. groups, semigroups, etc) cannot be used to reduce the search space and enhance performance. Canon and Holt, using a mix of smart algorithms and specific group theorems, produced fast code to compute automorphisms of groups that is faster than **CREAM** on large orders. Similarly, Mitchell et al. used semigroup theory theorems to compute in a very effective way automorphisms and congruences of completely 0-simple semigroups. But these are among the few cases of known GAP code that are faster than our general purpose package **CREAM**. For most other classes of algebras of type $(2^m, 1^n)$, **CREAM** is faster computing auto[endo]morphisms/congruences. One core algorithm implemented in the package is Freese's algorithm [15] that calculates principal congruences for a universal algebra.

To get this performance, **CREAM** uses a mixture of standard universal algebra algorithms together with tools that can search efficiently for finite models of first-order formulas and the implementation of parts of the code in C.

Keywords: Congruences, Morphisms, Universal, Algebra, GAP

Contents

List of Figures

List of Tables

Acronyms

CPU	Central Process Unit
CREAM	algebra CongRuences, Endomorphisms and AutomorphisMs
FIFO	First In, First Out
GAP	Groups, Algorithms, Programming
GUI	Graphical User Interface
MV	Many Valued logic
PhD	Doctor of Philosophy
SONATA	System Of Nearrings And Their Applications
UA	Universal Algebra

Chapter 1

Introduction

Currently the main software application for semigroups is the GAP package Semigroups [29], in connection with the package Smallsemi [13]. GAP [17] is a system and computer language for computational discrete algebra. While these packages offer many calculation options, and provide a library of all semigroups up to size 8, several important semigroup operators are not available. This is partly because the underlying architecture is geared towards group theory and semigroups often require algorithmic techniques that are closer to those employed in Universal Algebra than to those used in groups.

There are ways of building algebras from existing ones (direct products, etc.), but critical is the inverse operation: decomposing a given algebra into smaller ones. This type of decompositions is especially important in semigroups as even the structure of very small semigroups might be very obscure.

Computing congruences is important since congruences have close ties to the structure of algebras - for example, the subdirectly irreducible algebras are precisely the ones that possess a unique minimal nontrivial congruence; the use of congruences for the decomposition in direct products is an example of the relation between congruences' properties and direct reducibility. The connection between congruences and endomorphisms is well known and its importance in modern algebra can hardly be overestimated. Tame Congruence Theory [19] emerged as a powerful tool to study the structure of finite algebras, and continues to provide a big leverage in the study of locally finite varieties. These examples point to the critical importance of the study of congruences in algebra.

Investigation of automorphism groups of mathematical structures is one of the classical algebraic problems. A cornerstone was the work of Evariste Galois, but its impact goes far beyond. In the words of P. J. Cameron [9]:

In the famous Erlanger Programme, Klein proposed that geometry is the study of symmetry; more precisely, the geometric properties of an object are those which

1. INTRODUCTION

are invariant under all automorphisms of the objects. (...) According to Artin, "the investigation of symmetries of a given mathematical structure has always yielded the most powerful results."

These ideas, expressed by Klein in 1872 [23] and by Artin in 1957 [6], give some insight on why the computation of automorphisms of various mathematical structures has been such an attractive topic for so many decades.

As natural numbers are products of prime numbers, finite algebras are compositions of simpler algebras. Two of the most frequent compositions are direct products and sub-direct products. Tools to decompose a given algebra are therefore very important.

The Krohn-Rodes theorem is an important decomposition theorem for decomposition of finite automata and finite semigroups [32] with applications in biology and biochemical systems, artificial intelligence, finite-state physics, psychology, game theory and other domains. It is the analogue of the Jordan-Hölder Theorem for groups. One of the fundamental ingredients of this theorem are divisors.

Sub-algebras are the most basic and omnipresent objects in algebra. Sub-algebras are so important that some have their own names. For example, in groups we have: Hall, Frattini, center, Cartan, Fitting, normal, centralizer, fixed-point, etc. In rings, we have: ideal, center, centralizer, kernel, Gaussian integers, principal ideals, etc. A tool to compute sub-algebras (subgroups, subrings, etc) has been one of the most frequent requests in the GAP Forum. However calculating SubAlgebras is very demanding.

This makes the case in favour of a general centralized and effective GAP tool that computes congruences, automorphisms, endomorphisms, isomorphisms, monomorphisms, epimorphisms, divisors, sub-algebras and more for algebras of type $(2^m, 1^n)$, i.e. a finite universal algebra with m binary and n unary operations.

So far, there is no general computational tool to decompose algebras. For example, GAP already contains code to find all the congruences of semigroups, but it is very far from providing a general method other types of algebras. The situation is even worse regarding endomorphisms. The aim of the project is to address this situation. In this project several general algorithms (as in [15]) will be adapted to support algebras of type $(2^m, 1^n)$ thus covering not only semigroups but also unary algebras, loops, fields, rings, semi-rings, Lie algebras, MV-algebras, Meadows, algebras of logic, etc. These algorithms will be implemented in GAP. These will include the following questions:

- Given a finite algebra A of type $(2^m, 1^n)$, find all congruences of A, at least as fast as the existing programs (that is, the code produced will be general and at least as fast as the code that exists for particular classes of algebras only);

- Given a finite algebra A of type $(2^m, 1^n)$, find all endomorphisms of A; in particular the code should effectively compute the automorphisms of A, a tool that essentially only exists for groups.

The result of this work was **CREAM**[5] that stands for "Algebra **C**ong**R**uences, **E**ndomorphisms and **A**utomorphisi**M**s" and is a GAP package with fast methods for calculating congruences, automorphisms, endomorphisms, isomorphisms, monomorphisms, epimorphisms, divisors, sub-algebras and more for algebras of type $(2^m, 1^n)$, i.e., a finite universal algebra with m binary and n unary operations.

Algebras of type $(2^m, 1^n)$ cover the majority of the algebras used in practice and it is a very rare occurrence that an algebraic structure actually uses ternary or higher arity operations (see [8]).

Handling the most popular classes of algebras with a good performance makes **CREAM** a useful tool to almost all algebraists. Nevertheless, this needed to be achieved without using theorems specific to some types of algebras. Only theorems from Universal Algebra.

To achieve this, the most time consuming parts of the algorithms were re-written in C to improve the performance of the code, for example taking advantage of data structures allowing the fast manipulation of data among many other techniques. Also **CREAM** was integrated with **MACE4** benefiting of one of the fastest applications in its class that is under a continuous improvement process including artificial intelligence techniques that will benefit the performance of **CREAM** in the future. As far as we know this is the first time that this toll is integrated with GAP.

The resulting code is available as the fully documented GAP package **CREAM**.

After this introduction, the core of this thesis is divided in three parts. Chapter 2 reviews the mathematical background underlying this work. Chapter 3 discusses the several aspects of the computational implementation of the package **CREAM**. Chapter 4 describes application of the work done in this thesis. After these three core chapters, there is a small conclusion on Chapter 5. As Appendices this thesis includes the **CREAM** package manual and the article "CREAM: a Package to Compute [Auto, Endo, Iso, Mono, Epi]-morphisms, Congruences, Divisors and More for Algebras of Type $(2^n, 1^n)$"[5].

Chapter 2, "Mathematical Background", is divided in four sections covering the definitions of algebras of type $(2^n, 1^n)$, the definition of congruence and a discussion of the state of the art in terms of algorithms to calculate congruences, automorphisms and endomorphisms. In the last section, other algebras that are mentioned in examples or used in tests are defined.

Chapter 3, “Computational Implementation”, is divided in five sections covering the representation of data structures in the package **CREAM**, the algorithms implemented in **CREAM**, a description **CREAM** that points to the **CREAM** manual in the appendix A, a description of **MACE4** and finally the perfomance discussion of the package **CREAM**.

Chapter 4, “Applications”, is divided in six sections covering the application of **CREAM** in determination of monolithic algebras, to small semigroups, small groups, special groups, semigroups and monoids and particular classes of algebras defined axiomatically. Finally there is a section going through future improvements to **CREAM**.

Finally Chapter 5, “Conclusions”, goes through the main conclusions from this work.

Chapter 2

Mathematical Background

Definitions adapted from [8] are used throughout this section.

2.1 Algebras of Type (2^m,1^n)

The **CREAM** package applies to Universal Algebras of Type (2^m,1^n)[8] which includes a wide range of algebraic structures. Starting with the definition of Universal Algebra, different types and classes of algebras are defined in this section.

For A, a nonempty set and $n > 0$, A^n is the set of n-tuples of elements from A. An n-ary operation on A is a function $f : A^n \to A$. The arity of f is n.

A language (or type) of algebras is a set $\mathcal{F}$ of function symbols such that a nonnegative integer n is assigned to each member f of $\mathcal{F}$. This integer is called the arity (or rank) of f, and f is said to be an n-ary function symbol.

Definition 2.1.1 (Universal Algebra)**.** *If $\mathcal{F}$ is a language (or type) of algebras then an algebra **A** of type $\mathcal{F}$ is an ordered pair (A, F) where A is a nonempty set and F is a family of finitary operations on A indexed by the language $\mathcal{F}$ such that corresponding to each n-ary function symbol f in F there is an n-ary operation f^A on A.*

For the scope of this package are only considered operations with arity of 1 or 2, ie. unary and binary operations.

Definition 2.1.2 (Algebra of type (2^m,1^n))**.** *An algebra of type (2^m,1^n) is a universal algebra with m binary and n unary operations.*

This type of algebras do not include nullary operations for the sake of simplicity. This will not impact the calculation of congruences but might impact the calculation of endomorphisms. For instance, the endomorphisms of a monoid without taking into consideration the nullary operation (0) will be calculated in the sense of semigroup endomorphisms and not in the sense of monoid endomorphisms. If the monoid has more than one idempotent, there might be additional semigroup endomorphisms that are not monoid endomorphisms.

Two algebras are of the same type if the number of operations of each arity is the same. This does not mean that algebras of the same type belong to the same algebra class. For example, an algebra of type (2^1) is not necessarily a semigroup.

A table was proposed by the British mathematician Arthur Cayley to represent the binary operation of a group. In his honor, these are now known as Cayley Tables.

A Cayley table can be used to define a binary operation on a finite set A.

Table 2.1: Cayley Table

	1	2	3
1	1	2	3
2	2	3	1
3	3	1	2

Analogously to a Cayley table, a row vector can be used to represent a unary operation on a finite set A.

Table 2.2: Row Vector

1	2	3
3	1	2

An Algebra of type (2^m,1^n) can be defined by a set of Cayley tables and Row vectors representing binary and unary operations respectively.

Definition 2.1.3 (Magma)**.** *A Magma $(A, \cdot)$ is an algebraic structure consisting of a set A together with 1 binary operation. No other properties are imposed.*

Magmas can also be refered as Binar or Groupoid.

One of the main and most studied algebras classes are Semigroups.

Definition 2.1.4 (Semigroups)**.** *A Semigroup $(A, \cdot)$ is an algebraic structure consisting of a set A together with 1 binary operation, such that the following hold:*

$$\forall x, y, z \in A: \quad (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad \textit{(associativity)}$$

A semigroup is a Universal Algebra of type (2^1) in which the binary operation is associative. At times, if there is no cause for confusion, the operation symbol may be omitted, writing a *semigroup A* instead of a *semigroup* $(A, \cdot)$.

Many algebras, like monoids and groups, also have a binary associative operation, and often thought of as semigroups (with additional operations).

Definition 2.1.5 (Monoids)**.** *A Monoid* $(A, \cdot, 1)$ *is an algebraic structure consisting of a set A together with 1 binary operation and 1 nullary operation (constant), such that the following hold:*

$$\forall x, y, z \in A: \quad (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad \textit{(associativity)}$$
$$\forall x \in A: \quad 1 \cdot x = x = x \cdot 1 \quad \textit{(identity)}$$

1 *is the identity element of A.*

A Monoid is a Semigroup with identity and a Universal Algebra of type (2^1, 0^1) in which the binary operation is associative and where the nullary operation maps to the identity.

Since the Monoid constant 1 derives from the operation $\cdot$ and the Monoid axioms, a Monoid is often considered as a algebraic structure consisting of a set A together with 1 binary operation $(A, \cdot)$. As mentioned before representing monoids without the nullary operation might produce additional endomorphisms.

An ideal of a semigroup A is a set $I \subseteq A$ for which $AI \cup IA \subseteq I$.

Groups are one of the most studied algebraic structures since its introduction by Évariste Galois in the 1830s.

Definition 2.1.6 (Groups)**.** *A Group* $(A, \cdot, ^{-1}, 1)$ *is an algebraic structure consisting of a set A together with 1 binary operation, 1 unary operation and a 1 nullary operation (constant), such that the following hold:*

$$\forall x, y, z \in A: \quad (x \cdot y) \cdot z = x \cdot (y \cdot z) \quad \textit{(associativity)}$$
$$\forall x \in A: \quad 1 \cdot x = x = x \cdot 1 \quad \textit{(identity)}$$
$$\forall x \in A: \quad x \cdot x^{-1} = 1 = x^{-1} \cdot x \quad \textit{(inverse)}$$

1 *is the identity element of A.*

A Group is a Semigroup with identity and inverse and a Universal Algebra of type (2^1, 1^1, 0^1) in which the binary operation is associative, has an identity and inverse.

Since the Group unary operation $^{-1}$ and the constant 1 derive from the operation $\cdot$ and the Group axioms, a Group is often considered as a algebraic structure consisting of a set A together with 1 binary operation $(A, \cdot)$

A binary operation $\cdot$ on a set A is *commutative* if the following holds:

$$\forall x, y \in A : \quad x \cdot y = y \cdot x \quad \text{(commutativity)}$$

A Commutative Group is also called an Abelian Group.

Definition 2.1.7 (Lattices). *A Lattice $(A, \sqcup, \sqcap)$ is an algebraic structure consisting of a set A together with 2 binary operations called join and meet, such that the following hold:*

$$\begin{gathered}
\forall x, y \in A : \quad x \sqcup y = y \sqcup x \quad \wedge \quad x \sqcap y = y \sqcap x \quad \textit{(commutativity)} \\
\forall x \in A : \quad x \sqcup x = x \quad \wedge \quad x \sqcap x = x \quad \textit{(idempotency)} \\
\forall x, y, z \in A : \quad x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z \quad \wedge \quad x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z \quad \textit{(associativity)} \\
\forall x, y \in A : \quad x \sqcup (x \sqcap y) = x \quad \wedge \quad x \sqcap (x \sqcup y) = x \quad \textit{(absorption)}
\end{gathered}$$

Lattices can also be viewed as ordered sets in a standard way: we stipulate that $x \leq y$ if and only if $x \sqcup y = y$ (or, equivalently, if and only if $x \sqcap y = x$); then $(A, \leq)$ is a poset, and furthermore, each two element subset $\{x, y\}$ has a least upper bound equal to $x \sqcup y$ and a greatest lower bound equal to $x \sqcap y$. Conversely, if $(A, \leq)$ is a partially ordered set such that each two element set $\{x, y\}$ has both a least upper bound and a greatest lower bound, we obtain a lattice, in the sense of Definition 2.1.7, by defining the join operation as the least upper bound and the meet as the greatest lower bound.

The congruences of an algebra (see 2.2) form a lattice.

Other algebras that were used in the scope of testing or examples of the usage of **CREAM** to calculate congruences, automorphims and endomorphism are presented in section 2.4.

2.2 Congruences

Congruences are one of the main topics of this PhD work. Congruences is a unifying concept for quotient algebras, subalgebras and homomorphisms.

A congruence of an algebra is an equivalence relation on the underlying set of an algebra that is compatible with all algebraic operations [8].

Definition 2.2.1 (Congruence). *Let A be an algebra of type $\mathcal{F}$ and θ an equivalence relation on A. Then θ is a congruence on an algebra A if it satisfies $\mu(a_1, a_2, ..., a_n)\theta\mu(b_1, b_2, ..., b_n)$ for every n-ary operation $\mu \in \mathcal{F}$ and all elements $a_1, ..., a_n, b_1, ..., b_n$ such that $a_i\theta b_i$ for each $i = 1, ..., n$.*

The set of all congruences on an algebra A is denoted by $Con\ A$.

An equivalence relation on A, such as a congruence, partitions the set A into equivalence classes or blocks each made up of related elements.

Definition 2.2.2 (Partition). *A partition of a set A is a collection of non-empty subsets of A, such that every element of A is included in exactly one subset. Each subset in a partition is called a block.*

For example, $\{\{1, 3\}, \{2, 4\}\}$ is a 2-block partition of the set $\{1, 2, 3, 4\}$.

Let A be a set and θ an equivalence on A. For each $a \in A$, we denote by a/θ the equivalence class of a and by A/θ we denote the set of all equivalence classes:

$$a/\theta = \{\, a' \in A \mid a\theta a' \,\}$$
$$A/\theta = \{\, a/\theta \mid a \in A \,\}$$

To each congruence θ on A, there corresponds a *natural map* from A to A/θ, that maps each element of A to its equivalence class.

Definition 2.2.3 (Quotient Algebra). *On the set A/θ an algebra of type $\mathcal{F}$ is defined as follows:*

$\mu^{A/\theta}(a_1/\theta, a_2/\theta, ..., a_n/\theta) = \mu^A(a_1, a_2, ..., a_n)/\theta$ for every n-ary operation $\mu^A \in \mathcal{F}$.

This is well-defined since θ is a congruence of the algebra A and hence compatible with its operations.

Definition 2.2.4 (Principal Congruence). *Given an algebra A of type $\mathcal{F}$ with $x, y \in A$, the principal congruence generated by (x, y), which is denoted by $Cg^A(x, y)$ is the smallest congruence θ such that $x\theta y$.*

The congruences of A form a lattice with universe $Con\ A$, where the meet operation is the intersection of the input congruences and the join is the smallest congruence containing both input congruences.

Homomorphisms are closely related with congruences.

Definition 2.2.5 (Homomorphism)**.** *If A, B are two algebras of the same type, then a function $f : A \to B$ is a* homomorphism *from A to B if $f(\mu^A(a_1, a_2, ..., a_n)) = \mu^B(f(a_1), f(a_2), ..., f(a_n))$ for every n-ary operation μ^A, μ^B and $a_1, \ldots a_n \in A$. Here μ^A and μ^B stand for the two operations of A and B that are indexed equally by the (common) index scheme.*

A homomorphism from an algebra to itself is called an *endomorphism*.

If $f : A \to B$ is a bijective homomorphism then it is called a *isomorphism*. If an *isomorphism* exists between two algebras, then the two algebras are said to be *isomorphic*.

If f is both an *endomorphism* and an *isomorphism* then f is referred to as an automorphism.

If f is a homomorphism of an algebra A, then the image of $B = f(A)$ is a subalgebra of A and is also isomorphic to a quotient algebra of A, namely, it is isomorphic to A/θ, where θ is the kernel of f:

Definition 2.2.6 (Kernel)**.** *Let A, B be two algebras of the same type and $f : A \to B$ be a* homomorphism *from A to B. The kernel of f is:*

$$ker(f) = \{(x, y) \in A^2 : f(x) = f(y)\}$$

For each homomorphism $f : A \to B$, $ker(f)$ is an equivalence relation and is also a congruence on A.

Conversely, every congruence on A can be realized as the kernel of a suitable homomorphism:

If A is an algebra and $\theta \in Con\ A$, the natural map $f : A \to A/\theta$ defined by $f(x) = x/\theta$ is a homomorphism. Furthermore, $ker(f) = \theta$.

Theorem 2.2.1 (Homomorphism Theorem)**.** *If A, B are two algebras of the same type and the function $f : A \to B$ is an* onto homomorphism *from A to B then there is a unique isomorphism $g : A/ker(f) \to B$ satisfying $f = g \circ h$, where h is the natural homomorphism $h : A \to A/ker(f)$*

Figure 2.1: Homomorphism Theorem

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "h"] & B \\
A/ker(f) \arrow[ur, "g"] &
\end{tikzcd}$$

This relation between congruences, quotient algebras, subalgebras and endomorphisms allows us, having the congruences of an algebra, to calculate all endomorphisms.

2.3 State of the Art

In general, computing congruences of algebras is a difficult task. There are several descriptive theorems for different types of algebra and there are some computational tools to compute these objects for finite objects but generally these tools are applicable to a very specific and narrow set of algebras. Our goal was to provide an effective tool to compute them for finite algebras of type $(2^m, 1^n)$ encompassing a very wide set of algebras including most of the currently studied algebra types and classes.

Fast algorithms exist to find the congruences of groups and semigroups. For example, with GAP [17], a system for computational discrete algebra, the package semigroups [29] includes the function **CongruencesOfSemigroup** that returns the congruences of a given semigroup. This function only works for a limited set of semigroups that belong to the classes Simple, Brandt, Group, Zero Simple or Rectangular Band and is only highly efficient for Rees Matrix semigroups and Rees Zero Matrix semigroups. This function relies on specific techniques and congruence representations such as linked triples and others that are specific to some types of semigroups.

Not much literature was found regarding algorithms to calculate congruences for generic universal algebras. The Ralph Freese's algorithm described in [15] provides such an algorithm and was selected as basis for this PhD work. The Freese algorithm provides not only very fast algorithms based on strong mathematical background but also relies on data structures to encode partitions and algebra operations that be accessed fast and easily manipulated. The underlying data structures are arrays allowing that each element can accessed directly using pointers. For the representation of partitions (that represent also congruences) the encoding proposed is not only an array but also takes into consideration the performance implication of the encoding of the partition in the array.

During this work a second article [11] was analysed and its algorithms for calculating congruences were compared with the Freese's algorithm. One distinguishing feature of this approach is that this algorithm generates a closure system graph $G(\mathcal{C})$ that has as vertexes all unordered pairs $P(A)$ of the algebra A and edges from vertex $\{x, y\}$ to $\{z, w\}$ if there is an operation $f \in \mathcal{F}$ such that $\{z, w\} = \{f(x), f(y)\}$. The algorithm described here is very efficient in calculating this graph avoiding the repetition of the execution of algebra operations.

The complexity of these algorithms is detailed in the section 3.5.2 and the Freese algorithm is shown to have a complexity of $O(|\mathcal{F}|n^3)$ while the complexity of the graph algorithm is calculated to be bounded by $O(log_2(n^2)n^4)$. This makes both algorithms comparable if $|\mathcal{F}|$ grows proportional to $n\log(n^2)$, with an advantage for the Freese algorithm for slower growth and an advantage for the graph algorithm otherwise.

The algorithm chosen as the basis of this work was the Freese's algorithm and given the efficiency of its underlying data structures it was not considered changing it for this initial version of **CREAM**. However, there is the possibility that precalculating a transition graph between the unordered pairs and generating the congruences from this graph could result in some performance gains. This is one possibility that is listed as possible future improvement for the **CREAM** package.

Running on a different platform, jython, UACalc [16] also implements the Freese's algorithm that is the basis for the calculation of congruences described here. UACalc Command Line Interface was used as reference in terms of performance.

There are numerous papers concerning the automorphism groups of particular classes of algebras, for example, Schreier [34] and Mal'cev [26] described all automorphisms of the semigroup of all mappings from a set to itself. Similar results have been obtained for various other structures such as orders, equivalence relations, graphs, and hypergraphs; see the survey papers [31] and [30]. More examples are provided, among others, by Gluskĭn [18], Araújo and Konieczny [3], [4], and [2], Fitzpatrick and Symons [14], Levi [21] and [20], Liber [22], Magill [25], Schein [33], Sullivan [35], and Šutov [36].

In GAP, special properties of groups are used to implement the function *AutomorphismGroup* to efficiently find the automorphism group of a given group. However, this specialized method, while extremely efficient, works only on groups and not on any other more general algebraic structures, such as quasigroups, semigroups and magmas. Likewise, the Loops package [24] in GAP provides another version of the function *AutomorphismGroup* to compute all the automorphisms of a given quasigroup.

The Loops package algorithm uses invariants on group and quasigroup operations to narrow the possible automorphisms to a manageable set that can be easily tested.

In the scope of this PhD several automorphism algorithms were tested and the conclusion was that the invariants algorithm was the most promising. The final implementation of the automorphism algorithms using the invariants approach was contributed to **CREAM** by Choiwah Chow based on conclusions and work done in the scope of this PhD.

In the automorphisms algorithm the Loops package was used as guideline for the implementation of the **CreamAutomorphims** adapting the algorithm to support any magma allowing an easy adaptation to support any algebra of type (2^m,1^n).

Regarding endomorphisms the literature found was very scarce and a single implementation for groups and near-rings was found in the GAP library SONATA [1]. SONATA implements the function *Endomorphisms* that can calculate endomorphisms for groups and nearings by calculating the endomorphisms of the group reduct of the nearing and returning those that are also nearing endomorphisms. This function is very efficient for the algebras it supports.

Given the lack of references on algorithms to calculate endomorphisms, the endomorphism algorithm used in the **CREAM** package was defined using basic universal algebra theorems such as the homomorphism theorem to relate congruences, quotient algebras, subalgebras and endomorphisms. The algorithm was developed using the implemented congruences and automorphisms algorithms and an application that searches for finite models of first order formulas called **MACE4**.

2.4 Other Algebras

In this section we present a set of algebras that were used in the scope of testing or examples of the usage of **CREAM** to calculate congruences, automorphims and endomorphisms. While their definition is not required for understanding the theoretical background of this work, they are defined here for the sake of completeness.

Simple Semigroups have specific characteristics that makes the use of specific algorithms to calculate congruences possible.

Definition 2.4.1 (Simple Semigroup). *A Semigroup A is simple if it has no Ideals other than A.*

A Semigroup A with 0 contains an element $0 \in A$ such that $0x = x0 = 0$ for any $x \in A$.

Definition 2.4.2 (0-Simple Semigroup). *A Semigroup A is 0-simple if it has no Ideals other than $\{0\}$ and A.*

An element x of a set A equipped with a binary operator $\cdot$ is said to be idempotent if:

$$x \cdot x = x.$$

An idempotent $x \in A$ is said to be primitive if it is non-zero and there is no other non-zero idempotent $i \in A$ such that $ix = xi = i$.

A Completely 0-Simple Semigroup is a 0-Simple Semigroup that contains a primitive idempotent.

Viktor Valdimirovich Wagner in 1952 and Gordon Preston in 1954, introduced the notion of Inverse Semigroups.

Definition 2.4.3 (Inverse Semigroups). *A semigroup is called an Inverse Semigroup if every element x has a unique inverse x^{-1}, such that:*

$$xx^{-1}x = x \wedge x^{-1}xx^{-1} = x^{-1}.$$

Brandt semigroups were introduced by Heinrich Brandt in 1927.

Definition 2.4.4 (Brandt Semigroup). *A Brandt semigroup is completely 0-simple inverse semigroup.*

The Rees matrix semigroups were introduced by David Rees in 1940 that are used for building new semigroups from existing ones.

Definition 2.4.5 (Rees matrix semigroup). *A Rees matrix semigroup $(A, \cdot)$ is an algebraic structure consisting of a set A together with 1 binary operation. Given sets I and J, S a Group and $P = (p_{ij})_{j \in J, i \in I}$ a $\mid J \mid \times \mid I \mid$ matrix with entries in S, the Rees matrix semigroup is $(A, \cdot)$, where $A = J \times S \times I$ and $(i, s, j) \cdot (k, t, l) = (i, sp_{jk}t, l)$.*

Another interesting type of semigroups that will be refered in this work are Rectangular Bands.

Definition 2.4.6 (Rectangular Band). *A Rectangular Band is a semigroup such that the following hold:*

$$\forall x, y \in A : \quad xyx = x.$$

A bipartition of degree n is an equivalence relation on the set $I \cup J$, where $I = \{i_1, ..., i_n\}$ and $J = \{j_1, ..., j_n\}$.

Given 2 bipartitions of degree n, α and β, their product or composition $\alpha \cdot \beta$ can be described as: let $K = \{k_1, ..., k_n\}$, α' be obtained from α by changing every point $j \in J$ to K, β' be obtained from β by changing every point $i \in I$ to K. Θ is the equivalence relation $\alpha' \cup \beta'$ in $I \cup J \cup K$. $\alpha \cdot \beta$ is defined as the bipartition $\Theta \cap ((I \cup J) \times (I \cup J))$.

Definition 2.4.7 (Partition Monoid). *A Partition Monoid $(A, \cdot)$ is an algebraic structure consisting of a set A together with 1 binary operation where A is every possible bipartition of degree n and $\cdot$ is the composition of the 2 bipartitions.*

In the scope of a bipartition the relation $<$ is defined such than $i_k < i_l$, $j_k < j_l$, $i_k < j_l$, $j_k < i_l$ whenever $k < l$.

A bipartion x is called planar if there aren't distinct blocks $A, B \in x, a, a' \in A, b, b' \in B$ where $a < b < a' < b'$.

Definition 2.4.8 (Planar Partition Monoid). *A Planar Partition Monoid $(A, \cdot)$ is an algebraic structure consisting of a set A together with 1 binary operation where A is every possible planar bipartition of degree n and $\cdot$ is the composition of the 2 bipartitions.*

The Planar Partition Monoid is the subsemigroup of the Partition Monoid consisting of those bipartitions that are planar.

Definition 2.4.9 (Partial Brauer Monoid). *A Brauer Monoid $(A, \cdot)$ is an algebraic structure consisting of a set A together with 1 binary operation where A is every possible bipartition of degree n where blocks have size 2 or 1 and $\cdot$ is the composition of the 2 bipartitions.*

The Partial Brauer Monoid is the subsemigroup of the Partition Monoid consisting of those bipartitions where blocks have size 2 or 1.

Definition 2.4.10 (Brauer Monoid). *A Brauer Monoid $(A, \cdot)$ is an algebraic structure consisting of a set A together with 1 binary operation where A is every possible bipartition of degree n where blocks have size 2 and $\cdot$ is the composition of the 2 bipartitions.*

The Brauer Monoid is the subsemigroup of the Partial Brauer Monoid consisting of those bipartitions where blocks have size 2.

Definition 2.4.11 (Jones Monoid). *A Jones Monoid $(A, \cdot)$ is an algebraic structure consisting of a set A together with 1 binary operation where A is every possible planar bipartition of degree n where blocks have size 2 and $\cdot$ is the composition of the 2 bipartitions.*

The Jones Monoid is the subsemigroup of the Brauer Monoid consisting of those bipartitions that are planar.

Definition 2.4.12 (Motzkin Monoid). *A Motzkin Monoid $(A, \cdot)$ is an algebraic structure consisting of a set A together with 1 binary operation where A is every possible planar bipartition of degree n where blocks have size 1 or 2 and $\cdot$ is the composition of the 2 bipartitions.*

The Motzkin Monoid is the subsemigroup of the Partial Brauer Monoid consisting of those bipartitions that are planar.

Definition 2.4.13 (Gossip Monoid). *A Gossip Monoid $(A, \cdot)$ is an algebraic structure consisting of a set A together with 1 binary operation where A is all d by d boolean matrices that define an equivalence relation and $\cdot$ is the multiplication of the 2 matrices.*

Definition 2.4.14 (Symmetric Group). *The Symmetric Group of degree n is the group of all possible permutation on an n-element set with binary operation being the composition of the permutations.*

Definition 2.4.15 (Distributive Lattice). *A Lattice $(A, \sqcup, \sqcap)$ is* distributive *if it satisfies:*

$$\forall x, y, z \in A: \quad x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$$

Definition 2.4.16 (Near Distributive Lattice). *A Near Distributive Lattice $(A, \sqcup, \sqcap)$ is an algebraic structure consisting of a set A together with 2 binary operations called join and meet, such that the following hold:*

$$\begin{array}{rl} & (A, \sqcup, \sqcap) \textit{ is a Lattice} \\ \forall x, y \in A: & x \sqcap (y \sqcup z) = x \sqcap (y \sqcup (x \sqcap (z \sqcup (x \sqcap y)))) \\ \forall x, y \in A: & x \sqcup (y \sqcap z) = x \sqcup (y \sqcap (x \sqcup (z \sqcap (x \sqcup y)))) \end{array}$$

Definition 2.4.17 (Almost Distributive Lattice). *An Almost Distributive Lattice $(A, \sqcup, \sqcap)$ is an algebraic structure consisting of a set A together with 2 binary operations called join and meet, such that the following hold:*

$$\begin{array}{rl} & (A, \sqcup, \sqcap) \textit{ is a Near Distributive Lattice} \\ \forall x, y, v, u \in A: & v \sqcap (u \sqcup (x \sqcap (y \sqcup (x \sqcap z)))) \leq u \sqcup ((x \sqcap (y \sqcup (x \sqcap z))) \sqcap (v \sqcup (x \sqcap y) \sqcup (x \sqcap z))) \\ \forall x, y, v, u \in A: & v \sqcup (u \sqcap (x \sqcup (y \sqcap (x \sqcup z)))) \geq u \sqcap ((x \sqcup (y \sqcap (x \sqcup z))) \sqcup (v \sqcap (x \sqcup y) \sqcap (x \sqcup z))) \end{array}$$

$\leq$ *and* $\geq$ *being the underlying partial orders resulting from* $\sqcup$ *and* $\sqcap$).

Definition 2.4.18 (Bounded Lattice)**.** *A Bounded Lattice* $(A, \sqcup, \sqcap, 0, 1)$ *is an algebraic structure consisting of a set* A *together with 2 binary operations and 2 nullary operations (constants), such that the following hold:*

$$\begin{aligned} &(A, \sqcup, \sqcap) \textit{ is a Lattice} \\ \forall x \in A: \quad & 0 \sqcup x = x \\ \forall x \in A: \quad & 1 \sqcup x = 1 \end{aligned}$$

Definition 2.4.19 (Modular Lattice)**.** *A Modular Lattice* $(A, \sqcup, \sqcap)$ *is an algebraic structure consisting of a set* A *together with 2 binary operations, such that the following hold:*

$$\begin{aligned} &(A, \sqcup, \sqcap) \textit{ is a Lattice} \\ \forall x, y, z \in A: \quad & ((x \sqcap z) \sqcap y) \sqcup z = (x \sqcap z) \sqcup (y \sqcap z) \quad \textit{(modular identity)} \end{aligned}$$

Definition 2.4.20 (Lattice-Ordered Monoid)**.** *A Lattice-Ordered Monoid* $(A, \sqcup, \sqcap, *, 1)$ *is an algebraic structure consisting of a set* A *together with 3 binary operations and a nullary operation (constant), such that the following hold:*

$$\begin{aligned} &(A, \sqcup, \sqcap) \textit{ is a Lattice} \\ &(A, *, 1) \textit{ is a Monoid} \\ \forall x, y, z \in A: \quad & x * (y \sqcup z) = (x * y) \sqcup (x * z) \\ \forall x, y, z \in A: \quad & (x \sqcup y) * z = (x * z) \sqcup (y * z) \end{aligned}$$

A Commutative Lattice-Ordered Monoid is one where the monoid operation $*$ is commutative.

Definition 2.4.21 (Distributive Lattice Ordered Semigroup)**.** *A Distributive Lattice Ordered Semigroup* $(A, \sqcup, \sqcap, *)$ *is an algebraic structure consisting of a set* A *together with 3 binary operations, such that the following hold:*

$$\begin{aligned} &(A, \sqcup, \sqcap) \textit{ is a Distributive Lattice} \\ &(A, *) \textit{ is a Semigroup} \\ \forall x, y, z \in A: \quad & x * (y \sqcup z) = (x * y) \sqcup (x * z) \\ \forall x, y, z \in A: \quad & (x \sqcup y) * z = (x * z) \sqcup (y * z) \end{aligned}$$

Definition 2.4.22 (Complemented Lattice)**.** *A Complemented Lattice* $(A, \sqcup, \sqcap, 0, 1)$ *is an algebraic structure consisting of a set* A *together with 2 binary operations and 2 nullary operations (constants), such that the following hold:*

$$\begin{aligned} &(A, \sqcup, \sqcap, 0, 1) \textit{ is a Bounded Lattice} \\ \forall x \exists y \in A: \quad & (x \sqcup y = 1) \wedge (x \sqcap y = 0) \quad \textit{(complement)} \end{aligned}$$

Definition 2.4.23 (Boolean Algebra)**.** *A Boolean Algebra* $(A, \sqcup, \sqcap, \neg, 0, 1)$ *is an algebraic structure consisting of a set* A *together with 2 binary operations, 1 unary operation and 2 nullary operations (constants), such that the following hold:*

$$\begin{gathered}(A, \sqcup, \sqcap, 0, 1) \textit{ is a Bounded Distributive Lattice}\\ \forall x \in A: \quad x \sqcap \neg x = 0\\ \forall x \in A: \quad x \sqcup \neg x = 1\end{gathered}$$

Definition 2.4.24 (Ockham Algebra)**.** *A Ockham Algebra $(A, \sqcup, \sqcap, ', 0, 1)$ is an algebraic structure consisting of a set A together with 2 binary operations, 1 unary operation and 2 nullary operations (constants), such that the following hold:*

$$\begin{gathered}(A, \sqcup, \sqcap, 0, 1) \textit{ is a Bounded Distributive Lattice}\\ \forall x, y \in A: \quad (x \sqcap y)' = x' \sqcup y'\\ \forall x, y \in A: \quad (x \sqcup y)' = x' \sqcap y'\\ 0' = 1\\ 1' = 0\end{gathered}$$

Definition 2.4.25 (Ortholattice)**.** *A Ortholattice $(A, \sqcup, \sqcap, ', 0, 1)$ is an algebraic structure consisting of a set A together with 2 binary operations, 1 unary operation and 2 nullary operations (constants), such that the following hold:*

$$\begin{gathered}(A, \sqcup, \sqcap, 0, 1) \textit{ is a Bounded Lattice}\\ \forall x \in A: \quad x \sqcup x' = 1\\ \forall x \in A: \quad x \sqcap x' = 0\\ \forall x, y \in A: \quad (x \sqcap y)' = x' \sqcup y'\\ \forall x, y \in A: \quad (x \sqcup y)' = x' \sqcap y'\end{gathered}$$

Definition 2.4.26 (Bilattice)**.** *A Bilattice $(A, \sqcup, \sqcap, \oplus, \otimes, \neg)$ is an algebraic structure consisting of a set A together with 4 binary operations and an unary operation, such that the following hold:*

$$\begin{gathered}(A, \sqcup, \sqcap) \textit{ is a Lattice}\\ (A, \oplus, \otimes) \textit{ is a Lattice}\\ \forall x, y \in A: \quad (\neg(x \sqcup y) = \neg x \sqcap \neg y) \wedge (\neg\neg x = x)\\ \forall x, y \in A: \quad (\neg(x \oplus y) = \neg x \oplus \neg y) \wedge (\neg(x \otimes y) = \neg x \otimes \neg y)\end{gathered}$$

Definition 2.4.27 (MV-algebras)**.** *An MV-algebra $(A, +, \neg, 0)$ is an algebraic structure consisting of a set A together with 1 binary operation, 1 unary operation and 1 nullary operation (constant), such that the following hold:*

$$\begin{gathered}(A, +, 0) \textit{ is Commutative Monoid}\\ \forall x \in A: \quad \neg\neg x = x\\ \forall x \in A: \quad x + \neg 0 = \neg 0\\ \forall x, y \in A: \quad \neg(\neg x + y) + y = \neg(\neg y + x) + x\end{gathered}$$

A MV-algebra (Multivalued Logic Algebra) is a Universal Algebra of type $(2^1, 1^1, 0^1)$.

Definition 2.4.28 (Quasigroup). *A Quasigroup $(A, *, \backslash, /)$ is an algebraic structure consisting of a set A together with 3 binary operations, such that the following hold:*

$$\begin{aligned} \forall x, y \in A: &\quad y = x * (x \setminus y) \\ \forall x, y \in A: &\quad y = x \setminus (x * y) \\ \forall x, y \in A: &\quad y = (y/x) * x \\ \forall x, y \in A: &\quad y = (y * x)/x \end{aligned}$$

$\backslash$ *and* $/$ *are normally called left and right divisions.*

A Quasigroup is a Universal Algebra of type (2^3).

Since the Quasigroup operations $\backslash$ and $/$ that are derived from the operation $*$ and the Quasigroup axioms, a Quasigroup is often considered as an algebraic structure consisting of a set A together with 1 binary operation $(A, *)$.

Definition 2.4.29 (Loop). *A Loop $(A, *, \backslash, /, 1)$ is an algebraic structure consisting of a set A together with 3 binary operations and 1 nullary operation (constant), such that the following hold:*

$$\begin{gathered} (A, *, \backslash, /) \text{ is a Quasigroup} \\ \forall x \in A: \quad 1 \cdot x = x = x \cdot 1 \quad \text{(identity)} \end{gathered}$$

A Loop is a Quasigroup with identity and a Universal Algebra of type (2^3, 0^1).

Since the Loop operations $\backslash$, $/$ and constant 1 derive from the operation $*$ and the Loop axioms, a Loop is often considered as a algebraic structure consisting of a set A together with 1 binary operation $(A, *)$.

Definition 2.4.30 (Ring). *A Ring $(A, +, *, -, 0)$ is an algebraic structure consisting of a set A together with 2 binary operations, 1 unary operation and 1 nullary operation (constant), such that the following hold:*

$$\begin{gathered} (A, +, -, 0) \text{ is an Abelian Group} \\ (A, *) \text{ is a Semigroup} \\ \forall x, y, z \in A: \quad x * (y + z) = (x * y) + (x * z) \quad \text{(left distributivity)} \\ \forall x, y, z \in A: \quad (y + z) * x = (y * x) + (z * x) \quad \text{(right distributivity)} \end{gathered}$$

A Ring is a Universal Algebra of type ($2^2, 1^1, 0^1$).

Since the Ring operation $-$ and constant 0 are derived from the operation $+$ and the Ring axioms, a Ring is often considered as a algebraic structure consisting of a set A together with 2 binary operations $(A, +, *)$

Definition 2.4.31 (Ring with Identity)**.** *A Ring with Identity* $(A, +, *, -, 0, 1)$ *is an algebraic structure consisting of a set A together with 2 binary operations, 1 unary operation and 2 nullary operation (constant), such that the following hold:*

$$\begin{aligned} &(A, +, *, -, 0) \textit{ is a Ring} \\ \forall x \in A: \quad & x * 1 = x \wedge 1 * x = x \quad \textit{(identity)} \end{aligned}$$

A Ring with Identity is a Universal Algebra of type $(2^2, 1^1, 0^2)$.

Definition 2.4.32 (Regular Ring)**.** *A Regular Ring* $(A, +, *, -, 0, 1)$ *is an algebraic structure consisting of a set A together with 2 binary operations, 1 unary operation and 2 nullary operation (constant), such that the following hold:*

$$\begin{aligned} &(A, +, *, -, 0, 1) \textit{ is a Ring with Identity} \\ \forall x \exists y \in A: \quad & x * y * x = x \quad \textit{(pseudo-inverse)} \end{aligned}$$

A Regular Ring is a Universal Algebra of type $(2^2, 1^1, 0^2)$.

A Commutative Ring is a Ring where the $*$ operation is commutative.

Definition 2.4.33 (Boolean Ring)**.** *A Boolean Ring* $(A, +, *, -, 0, 1)$ *is an algebraic structure consisting of a set A together with 2 binary operations, 1 unary operation and 2 nullary operation (constant), such that the following hold:*

$$\begin{aligned} &(A, +, *, -, 0) \textit{ is a Commutative Ring} \\ &(A, +, *, -, 0, 1) \textit{ is a Ring with Identity} \\ \forall x \in A: \quad & x * x = x \end{aligned}$$

A Boolean Ring is a Universal Algebra of type $(2^2, 1^1, 0^2)$.

Thus, a Boolean Ring is a commutative ring with identity where the multiplication operation is idempotent.

Definition 2.4.34 (Semiring)**.** *A Semiring* $(A, +, *)$ *is an algebraic structure consisting of a set A together with 2 binary operations, such that the following hold:*

$$\begin{aligned} &(A, +) \textit{ is a Commutative Semigroup} \\ &(A, *) \textit{ is a Semigroup} \\ \forall x, y, z \in A: \quad & x * (y + z) = (x * y) + (x * z) \quad \textit{(left distributivity)} \\ \forall x, y, z \in A: \quad & (y + z) * x = (y * x) + (z * x) \quad \textit{(right distributivity)} \end{aligned}$$

A Semiring is a Universal Algebra of type (2^2).

Definition 2.4.35 (Idempotent Semiring)**.** *A Idempotent Semiring* $(A, +, *)$ *is an algebraic structure consisting of a set* A *together with 2 binary operations, such that the following hold:*

$$\begin{gathered}(A, +, *) \textit{ is a Semiring} \\ \forall x \in A : \quad x + x = x\end{gathered}$$

An Idempotent Semiring is a Universal Algebra of type (2^2).

Definition 2.4.36 (Meadows)**.** *A Meadow* $(A, +, *, -, ^{-1}, 0, 1)$ *is an algebraic structure consisting of a set* A *together with 2 binary operations, 2 unary operations and 2 nullary operations (constants), such that the following hold:*

$$\begin{gathered}(A, +, *, -, 0) \textit{ is a Commutative Ring} \\ \forall x \in A : \quad 1 * x = x \quad \textit{(identity)} \\ \forall x \in A : \quad (x^{-1})^{-1} = x \\ \forall x \in A : \quad x * (x * x^{-1}) = x\end{gathered}$$

A Meadow is a commutative Ring with a multiplicative identity element and a total multiplicative inverse operation and is a Universal Algebra of type ($2^2, 1^2, 0^2$).

Since the Meadows operations $-$, $^{-1}$ and constants 0 and 1 are derived from the operations $+$, $*$ and the Meadows axioms, a Meadow is often considered as a algebraic structure consisting of a set A together with 2 binary operations $(A, +, *)$

Definition 2.4.37 (BCI-algebras)**.** *A BCI-algebra* $(A, *, 0)$ *is an algebraic structure consisting of a set* A *together with 1 binary operation and 1 nullary operation (constant), such that the following hold:*

$$\begin{gathered}\forall x, y, z \in A : \quad ((x * y) * (x * z)) * (z * y) = 0 \\ \forall x, y \in A : \quad (x * (x * y)) * y = 0 \\ \forall x \in A : \quad x * x = 0 \\ \forall x, y \in A : \quad ((x * y = 0) \wedge (y * x = 0)) \Rightarrow (x = y) \\ \forall x \in A : \quad (x * 0 = 0) \Rightarrow (x = 0)\end{gathered}$$

A BCI-algebras is a Universal Algebra of type ($2^1, 0^1$).

Chapter 3

Computational Implementation

3.1 Representation

The representation of algebras and congruences has a significant impact on the efficiency of the algorithms implemented. Especially the representation of partitions/congruences as an array is determinant for the efficiency in the calculation of congruences since it allows for a very fast join of blocks, one of the most used operations during the execution of the congruences algorithm.

3.1.1 Algebra of Type (2^m,1^n)

The package **CREAM** represents a finite Universal Algebra as a list of operations. An operation of arity 1 is represented by a vector, while an operation of arity 2 is represented by a square matrix. The underlying set A is implicitly specified by the vector or matrix sizes, which need to agree in a valid representation. If this dimension is d, the algebra is defined on the set $A = \{1, \ldots, d\}$. The particular case of $\mathcal{F} = \{\}$, an algebra without operations, is not supported by this implicit representation. Since such algebras are not interesting mathematically these algebras are being excluded from this representation and are not considered valid algebras.

The vector or matrix describes the corresponding operation by listing all images in the obvious way. The following is an example of a representation for an algebra with a unary and a binary operation.

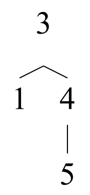
```
[ [3, 1, 2],
  [ [1, 2, 3],
    [2, 3, 1],
    [3, 1, 2] ] ]
```

3.1.2 Partition

One of the key starting points in the representation of algebras is the representation of partitions. The Freese algorithm [15] performance depends on the way partitions are represented internally. In [15], Freese goes back to 1960's and 1970's findings in computer science described in [37]. Several approaches were tested and the most efficient way to represent a block of a partition was found to be a tree. Hence, the representation of a partition would then be a forest where a forest is a disjoint union of trees. The forest is physically encoded in the computer memory by an array, whose size will be the size of the set A.

In a tree, each node or leaf has a parent node except for the top nodes or roots. For example the block $\{1, 3, 4, 5\}$ could be represented by the tree below where 3 is the root block and 1 and 5 are the leafs.

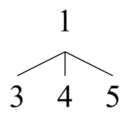
Figure 3.1: Possible tree representation of a block [1,3,4,5]



This isn't the single representation of this block. Parts of the algorithms will work with any representation of a block while others will require the block representation to be unique.

A shallow tree is a tree in which all non-root nodes are connected directly to the top root element. To obtain a unique representation, we adopted the convention that the trees used in representing a partition are shallow and that the smallest element in a block will be the root node. A partition representation that respects the above conventions will be called a normalized (representation of a) partition. The normalized representation of the block $\{1, 3, 4, 5\}$ would be:

Figure 3.2: Normalized tree representation of a block [1,3,4,5]



Since a partition is a set of disjoint blocks it can be represented as a set of disjoint trees that hence can be called a forest.

A partition such as $\{\{1,3,4,5\},\{2,6\}\}$ will be represented by 2 trees such as:

Figure 3.3: Forest representation of a congruence [[1,3,4,5][2,6]]

When representing the forest as an array each position of the array represents a node and the value of the node points to its parent node. The top nodes should have a value indicating that the node is the root of the tree. A negative number is used to indicate that a node is a root node and the absolute value of this number is the number of elements of the block.

There are two main reasons for this to be the most efficient representation of partitions. On one side, the tree representation allow the joining of 2 blocks to be very fast, since after the determination of the root node of each of the blocks to join, it is sufficient to change the value of one of the root nodes to point to the other to do the join. On the other hand, the use of an array allows direct access to any node and a fast transversing of the tree up to the root node when determining the root node.

Using these conventions, the encoding of the above partition will have the form:

```
[[1, 3, 4, 5], [2, 6]]  -  [-4, -2, 1, 1, 1, 2]
```

The encoding of a partition into singletons will have the form:

```
[[1], [2], [3], [4], [5], [6]]  -  [-1, -1, -1, -1, -1, -1]
```

The encoding of a partition with just one block is:

```
[[1, 2, 3, 4, 5, 6]]  -  [-6, 1, 1, 1, 1, 1]
```

Other examples are:

```
[[1, 6], [2], [3, 5], [4]] - [-2, -1, -2, -1, 3, 1]
[[1, 3, 5], [2, 6], [4]] - [-3, -2, 1, -1, 1, 2]
[[1, 2, 5, 6], [3, 4]] - [-4, 1, -2, 3, 1, 1]
```

3.1.3 Congruences

Every equivalence relation and hence every congruence corresponds to a partition of the underlying set, and hence congruences can be represented by partitions as detailed in the previous section.

3.1.4 Homomorphism, Endomorphism, Isomorphism and Automorphism

To represent an endomorphism of an algebra of size n, an n-sized array can be used.

In this array representation the position i of the array contains the image of i in this mapping. Therefore the mapping $f(i) = a_i$ for an algebra of size 6 is represented as:

```
[a1, a2, a3, a4, a5, a6]
```

The identity mapping on an algebra A of order 6 would be represented by:

```
[1, 2, 3, 4, 5, 6]
```

If this mapping is bijective, such an automorphism, then all values in the array will be different and less or equal than n, such as:

```
[ 1, 2, 6, 5, 4, 3 ]
[ 1, 3, 2, 5, 4, 6 ]
[ 1, 3, 6, 4, 5, 2 ]
[ 1, 6, 2, 4, 5, 3 ]
[ 1, 6, 3, 5, 4, 2 ]
```

In case of non-injective mappings, such as endomorphisms repeated values can appear in the array:

```
[ 1, 1, 1, 1, 1, 1 ]
[ 1, 2, 2, 1, 1, 2 ]
[ 1, 3, 3, 1, 1, 3 ]
[ 1, 6, 6, 1, 1, 6 ]
```

3.2 Algorithms

In this section, several algorithms will be presented using pseudocode. This pseudocode tries to be as language agnostic as possible but in some cases uses the syntax of GAP or C to convey certain concepts.

One of these cases are array and lists were the GAP syntax is used for representing and accessing these structures. To represent the element in the position index in a uni-dimensional array or list we use `array[index]` and in the case of bi-dimensional arrays or list of lists we use `array[index1][index2]`. To represent/create an array or list with 2 elements we use `[element1, element2]`.

3.2.1 Congruences algorithm

The starting point for the computation of congruences are the algorithms described in [15] to calculate the smallest congruence containing a given partition Θ of a finite algebra A; in particular this is used to compute the congruence generated by a pair of elements $(a, b) \in A \times A$, called the *principal congruence* generated by $\{a, b\}$. From this base algorithm all congruences of the algebra A are generated in an efficient way. We used [15] as it is the fastest general algorithm we know; some optimizations were introduced taking into account that we only deal with operations of arity at most 2, and we take advantage of C to get an implementation faster than the original implementations made by Freese and his collaborators; finally, we make it available through GAP and hence fully compatible with the many other resources in the system. Of course, for our goals, the computation of congruences is the necessary intermediate step towards the computing of proper endomorphisms of general algebras (of the considered types).

3.2.1.1 Partition Functions

Several functions to manipulate partitions play an important role in the algorithm to calculate principal congruences and will be described in the following section.

The function **SizeOfSet** takes a partition as argument and returns the number of elements of the set underlying to the partition.

The function **NumberOfElements** takes an array as argument and returns the number of elements of the array.

The function **NumberOfFunctions** takes an algebra as argument and returns the number of functions of the algebra.

The function **CreamSizeAlgebra** takes an algebra as argument and returns the number of elements of the set underlying to the algebra.

The function **CreamRootBlock** is an operation that returns the root node for a node i. The root node of a node is itself if the value of the node representation is negative or points to itself. If the value of the node representation points to a different node then that node is the parent node. This algorithm could be run recursively until reaching the root node but it is implemented iteratively for better performance. This operation will work both for normalized and non-normalized partitions.

Before returning, the node parent is set to the found root node in order to make the tree representing the partition as shallow as possible avoiding that in future calls the algorithm needs to transverse several nodes to reach the root node.

Algorithm 1 CreamRootBlock (i, partition)

```
$j \leftarrow i$
while $partition[j] \geq 0$ do
  $j \leftarrow partition[j]$
end while
if $i \neq j$ then
  $partition[i] \leftarrow j$
end if
return $j$
```

The function **CreamJoinBlocks** is an operation that joins the blocks of x and y in a partition. This operation will work both for normalized and non-normalized partitions and the resulting partition will not necessarily be normalized.

In order to keep the tree representing the partition as shallow as possible the root node of the merged block will be the root node of the block with more elements originally.

Algorithm 2 CreamJoinBlocks (x, y, partition)

```
$r \leftarrow CreamRootBlock(x, partition)$
$s \leftarrow CreamRootBlock(y, partition)$
**if** $r \neq s$ **then**
   **if** $partition[r] < partition[s]$ **then**
      $partition[r] \leftarrow partition[r] + partition[s]$
      $partition[s] \leftarrow r$
   **else**
      $partition[s] \leftarrow partition[r] + partition[s]$
      $partition[r] \leftarrow s$
   **end if**
**end if**
```

The **CreamNumberOfBlocks** function returns the number of blocks of a partition. Given the encoding of partitions, we simply count the number of positions of the array that have negative values.

Algorithm 3 CreamNumberOfBlocks (partition)

```
$nblocks \leftarrow 0$
$dimension \leftarrow SizeOfSet(partition)$
**for** $i = 1$ to $dimension$ **do**
   **if** $partition[i] < 0$ **then**
      $nblocks \leftarrow nblocks + 1$
   **end if**
**end for**
**return** $nblocks$
```

The **CreamNormalizePartition** function normalizes the partition by making it shallow and having the smallest element of each block the root node.

Algorithm 4 CreamNormalizePartition (partition)

```
$dimension \leftarrow SizeOfSet(partition)$
for $i = 1$ to $dimension$ do
    $r \leftarrow CreamRootBlock(i, partition)$
    if $r \geq i$ then
        $partition[i] \leftarrow -1$
        if $r > i$ then
            $partition[r] \leftarrow i$
        end if
    else
        $partition[r] \leftarrow partition[r] - 1$
    end if
end for
```

The **CreamJoinPartition** function joins two partitions, i.e. the smallest partition containing both input partitions.

Algorithm 5 CreamJoinPartition (partition1, partition2)

```
$dimension \leftarrow SizeOfSet(partition1)$
for $i = 1$ to $dimension$ do
    CreamJoinBlocks($i$,CreamRootBlock ($i$,$partition1$),$partition2$)
end for
```

The **CreamComparePartitions** function compares normalized partitions. Returns 0 if the partitions are equal. Returns -1 or 1 if partition1 > partition2 or partition1 < partition2 . Partition order is equivalent to the underlying list order. Its purpose is to allow binary search in sets of partitions.

Algorithm 6 CreamComparePartitions (partition1, partition2)

```
$dimension \leftarrow SizeOfSet(partition)$
**for** $i = 1$ to $dimension - 1$ **do**
    **if** $partition1[i] > partition2[i]$ **then**
        **return** $-1$
    **end if**
    **if** $partition1[i] < partition2[i]$ **then**
        **return** $1$
    **end if**
**end for**
**return** $0$
```

3.2.1.2 Calculating Principal Congruences

The base algorithm in [15] calculates the smallest congruence containing Θ, a partition of a finite algebra A. The simplest case where Θ only contains one nontrivial block and this block has only two elements, is used as a starting point.

The algorithm works with unary algebra operations but in what concerns this algorithm an n-ary algebra operation can be converted into a set of unary operations. For the purpose of this work, algebras with operation arity up to 2 are considered. Therefore, binary algebra operations need to be converted to unary operations.

For simplicity it will be assumed that A is an algebra of size $|A|$ and type (2^1) with the binary operation μ. This can be easily extended to multiple binary operations. To convert this operation to unary operations, x is fixed to each argument of μ, and for each $x \in A$ will result in:

$$\mu_x(y) = \mu(y,x)$$
$$\mu'_x(y) = \mu(x,y)$$

The equivalence relation θ on A is a congruence if and only if it is compatible with all $2|A|$ unary operations $\mu_x(y)$, $\mu'_x(y)$ for all $x \in A$.

3. COMPUTATIONAL IMPLEMENTATION

Suppose θ is a congruence and that $a\theta b$ with $a, b \in A$, then for every x:

$$\mu_x(a) = \mu(a,x)\theta\mu(b,x) = \mu_x(b)$$
$$\mu'_x(a) = \mu(x,a)\theta\mu(x,b) = \mu'_x(b)$$

Conversely, suppose that θ is compatible with all $\mu_x(y)$, $\mu'_x(y)$ for all $x \in A$, and that $a_1\theta b_1$, $a_2\theta b_2$ with $a_1, a_2, b_1, b_2 \in A$. Then:

$$\mu(a_1, a_2) = \mu_{a2}(a_1)\theta\mu_{a2}(b_1) = \mu(b_1, a_2) = \mu'_{b1}(a_2)\theta\mu'_{b1}(b_2) = \mu(b_1, b_2)$$

Therefore for the purpose of congruence calculation, a binary operation $\mu(x, y)$ with $x, y \in A$ can be converted into $2|A|$ unary operations, where $|A|$ is the size of the algebra:

$$\mu_x(y) = \mu(y, x) \text{ and } \mu'_x(y) = \mu(x, y)$$

There will be one unary operation for each line and column of the operation's multiplication matrix. Duplicate unary operations that result from this conversion can be removed from the set of unary operations.

For instance for the algebra:

```
[ [ [2, 1, 1],
    [1, 2, 2],
    [1, 3, 2] ] ]
```

the following unary operations would be considered;

```
[ [2, 1, 1],
  [1, 2, 2],
  [1, 3, 2],
  [1, 2, 3] ]
```

Notice that 2 duplicate unary operations are removed in this particular case.

The principal congruence algorithm will take as input the algebra and a pair of elements of A that will constitute the one nontrivial block with 2 elements to be used as a starting point.

Algorithm 7 CreamPrincipalCongruence (algebra,InitialPair)

```
$PairList \leftarrow [InitialPair]$
$partition \leftarrow SingletonPartition()$
$partition \leftarrow CreamJoinBlocks(partition, InitialPair[1], InitialPair[2])$
$NFuncs \leftarrow NumberOfFunctions(algebra)$
**while** $PairList \neq empty$ **do**
    $Pair \leftarrow$ Get and remove element from $PairList$
    **for** $i = 1$ to $NFuncs$ **do**
        $f \leftarrow algebra(i)$
        $r \leftarrow CreamRootBlock(partition, f(Pair[1]))$
        $s \leftarrow CreamRootBlock(partition, f(Pair[2]))$
        **if** $r \neq s$ **then**
            $partition \leftarrow CreamJoinBlocks(partition, r, s)$
            Add $[r, s]$ to $PairList$
        **end if**
    **end for**
**end while**
**return** $partition$
```

Let A be an algebra of type (1^f) with f unary operations μ_p and θ an equivalence relation on A. The algorithm starts by joining the initial pair (a, b) in the same block and creates a new pair $(r_i(\mu_p(a)), r_i(\mu_p(b)))$ for every operation where $r_i(x)$ is the root of x according to the variable $partition$ at step i. If the elements of the new pair are in different blocks, the blocks are joined, the new pair is added to the list of pairs to be processed and the cycle is repeated. Otherwise, if elements of the new pair are already in the same block no new pair will be added to the pair list. The algorithm terminates when every pair in the list is already in the same block, running out of new pairs and returning the current partition.

If θ is a congruence on an algebra A and $x, y \in A$ then by definition:

$$x\theta y \Rightarrow \mu_p(x)\theta\mu_p(y), \text{ for every } \mu_p \text{ where } 0 < p \leq f$$

and the principal congruence $Cg(a, b)$ is the smallest congruence such that $a\theta b$.

The first step to prove the correctness of the algorithm is to prove that the termination condition of the algorithm's main cycle is reached, the algorithm terminates and returns a partition.

Initially the $partition$ variable contains a partition where every block has a single element. The starting pair (a, b) is added to a FIFO list and the blocks of a and b are joined.

Let $r_i(x)$ be the root of x according to the variable $partition$ at step i of the algorithm for any $x \in A$.

On each subsequent step i a pair (u, v), $u, v \in A$, is removed from the list, and a new pair $(r_i(\mu_p(u)), r_i(\mu_p(v)))$ is created for each μ_p. If $r_i(\mu_p(u)) = r_i(\mu_p(v))$ then $\mu_p(u)$ and $\mu_p(v)$ are already in the same block of the partition and no new pair is added to the pair list. Otherwise the blocks are joined and the pair $(r_i(\mu_p(u)), r_i(\mu_p(v)))$ is added to the pair list.

The algorithm stops when the pair list is empty. This termination condition is guaranteed to occur since at most $n - 2$ joins can be done before every pair (u, v) is removed from the list and $r_i(\mu_p(u)) = r_i(\mu_p(v))$ is true and in this case the pair is removed from the pair list and no new pair is added, progressively reducing the list size until the list is exhausted.

Let θ' be the partition's underlying equivalence relation when the algorithm stops and let θ be the principal congruence $Cg(a, b)$.

The correctness of the algorithm requires that the correctness of the output is proven. One of the conditions to prove the correctness of the algorithm is that the resulting partition is a congruence. To prove that θ' is a congruence it's needed to prove that if $u\theta'v$ then $\mu_p(u)\theta'\mu_p(v)$ for all μ_p.

At the start of algorithm, every element starts as a singleton in the partition being its own root. The first pair whose blocks are joined in the same block is the initial pair (a, b) and the algorithm generates pairs such as $(r_i(\mu_p(a)), r_i(\mu_p(b)))$. This means $a\theta'b$ and later at a step i the pair $(r_i(\mu_p(a)), r_i(\mu_p(b)))$ blocks are joined and $r_i(\mu_p(a))\theta' r_i(\mu_p(b))$. Since obviously $\mu_p(a)\theta' r_i(\mu_p(a))$ and $r_i(\mu_p(b))\theta'\mu_p(b)$ then $\mu_p(a)\theta'\mu_p(b)$ for any μ_p, proving that the conditions for θ' to be a congruence hold for the initial pair (a, b).

Lemma 3.2.1. *If $u\theta'v$ then $\mu_p(u)\theta'\mu_p(v)$ and the conditions for θ' to be a congruence hold.*

Proof. In each join of the pair (u_1, v_1) with blocks $\{u_1, ..., u_p\}$ and $\{v_1, ..., v_q\}$, the block $\{u_1, ..., u_p, v_1, ..., v_q\}$ is created and $u_1\theta'...\theta'u_p\theta'v_1\theta'...\theta'v_q$. This means that $u_1\theta'v_1$ and at some later step i the new pair $(r_i(\mu_p(u_1)), r_i(\mu_p(v_1)))$ is created, its blocks are joined and $r_i(\mu_p(u_1))\theta' r_i(\mu_p(v_1))$. Assuming that from previous pairs it's already proven that $\mu_p(u_1)\theta'...\theta'\mu_p(u_p)$ and $\mu_p(v_1)\theta'...\theta'\mu_p(v_q)$ then this step proves also that $\mu_p(u_1)\theta'...\theta'\mu_p(u_p)\theta'\mu_p(v_1)\theta'...\theta'\mu_p(v_q)$. Since our assumption holds at start of the algorithm then it will hold for every subsequent step. This proves that if $u\theta'v$ then $\mu_p(u)\theta'\mu_p(v)$ proving that the conditions for θ' to be a congruence hold after each step of the algorithm. ☐

One additional condition for the correctness of the algorithm is that θ' contains $\theta = Cg(a, b)$. At the start of the algorithm the initial pair (a, b) blocks are joined by

$CreamJoinBlocks(a, b)$ in the same block and (a, b) will be the initial pair added to the pair list that will be processed with each operation. Since the initial pair will always be in the same block of the partition and $a\theta' b$ then θ' contains $\theta = Cg(a, b)$.

At this point it remains to be proved that $\theta' \subseteq \theta = Cg(a, b)$.

To prove that $\theta' = \theta$ it remains to prove that for every pair u and v in the same block of the variable $partition$ then $u\theta v$. This is self-evident for the initial pair (a, b) since $a\theta b$ and at the start of the algorithm the blocks of a and b are joined and $\{a, b\}$ is the only non-singleton block in $partition$. Moreover the pair added to the list is (a, b) which obviously lies in θ. Remains to prove that after each subsequent step this remains true taking as assumption that this holds for previous steps.

Lemma 3.2.2. *If u and v are in the same block of the variable $partition$ at step i then $u\theta v$.*

Proof. Assuming that at a step i that the $partition$ blocks are contained in θ then in each join with the pair (u_1, v_1) the blocks $\{u_1, ..., u_p\}$ and $\{v_1, ..., v_q\}$ are contained in θ and the resulting block from the join will be contained in θ if $u_1\theta v_1$. Considering the algorithm then $u_1 = r_i(\mu_p(x))$ and $v_1 = r_i(\mu_p(y))$ where x and y come from the pair list and are in the same block since for them to get into the pair list their blocks would have to be merged beforehand. Under the initial assumption, if x and y are in the same block then $x\theta y$. Since θ is a congruence if $x\theta y$ then $\mu_p(x)\theta\mu_p(y)$. Also given that $r_i(\mu_p(x))$ and $\mu_p(x)$ are by definition in the same block then $r_i(\mu_p(x))\theta\mu_p(x)$ and analogously $r_i(\mu_p(y))\theta\mu_p(y)$. This means that since $x\theta y$ then $r_i(\mu_p(x)) = u_1\theta v_1 = r_i(\mu_p(y))$ and the joined block will be contained in θ. Since our assumption holds at start of the algorithm and holds after every step then if the elements of a pair are in the same block of the variable $partition$ and then the pair lies in θ. As the lemma holds at every step i, it also holds when the algorithm terminates □

Since θ' is a congruence containing the principal congruence and every pair in the same block lies in the principal congruence then θ' is the principal congruence and $\theta' = \theta$.

To calculate all the principal congruences this function is called for all pairs of elements of A.

Algorithm 8 CreamAllPrincipalCongruences (algebra)

```
$dimension \leftarrow CreamSizeAlgebra(algebra)$
$allPrincipalCongruences \leftarrow []$
for $i = 1$ to $dimension - 1$ do
    for $j = i + 1$ to $dimension$ do
        $congruence \leftarrow CreamPrincipalCongruence(algebra, [i, j])$
        $AddCongruence(allPrincipalCongruences, congruence)$
    end for
end for
return $allPrincipalCongruences$
```

As described in [15] this algorithm is very efficient showing a moderate growth of execution time with $|A|$. Apart from the algorithm itself, there are a few implementation options that make this implementation of the algorithm especially efficient.

One of these aspects is the use of arrays to encode partitions, which allow for a constant and very fast random access to each partition element.

This is combined with the balancing and collapsing of the trees representing the blocks in the partition that are baked into the **CreamJoinBlocks** and **CreamRootBlock** algorithms which try to ensure that the trees are as shallow as possible during the execution of the **CreamPrincipalCongruence** algorithm.

In **CreamJoinBlocks** the tree is balanced by keeping as root of the joined block the root of the bigger original block.

In **CreamRootBlock** the tree is collapsed by setting the root node of the element on which the function is called to its return value avoiding having to transverse several nodes of the tree on future calls.

Both of these implementation details allow for shallower trees representing the blocks that will make **CreamRootBlock** faster given that fewer nodes will have to be transversed to determine the root node of a node.

This is especially important since **CreamRootBlock** is the most called function when calculating a principal congruence of an algebra.

3.2.1.3 Calculating All Congruences

Having all the principal congruences, the remaining congruences can be obtained by combining the principal congruences.

It is well known that congruences of an algebra are naturally ordered by inclusion and form a lattice. All the congruences can be obtained as joins of principal congruences.

This allows an efficient way to compute all the congruences. The principal congruences are stored in an ordered set. In a separate ordered set, all congruences are gathered. This is initialized with the set of principal congruences. For each congruence in this set, the joins with principal congruences are computed, and the resulting new congruences are added to the set (eliminating duplicates).

Algorithm 9 CreamAllCongruences (algebra)

```
$allPrincipalCongruences \leftarrow CreamAllPrincipalCongruences(algebra)$
$allCongruences \leftarrow allPrincipalCongruences$
$i \leftarrow 1$
**while** $i < NumberOfElements(allCongruences)$ **do**
  **for** $j = 1$ to $NumberOfElements(allPrincipalCongruences)$ **do**
    **if** $\neg isContained(allPrincipalCongruences[j], allCongruences[i])$ **then**
      $congruence \leftarrow JoinPartition(allCongruences[i], allPrincipalCongruences[j])$

      $AddCongruence(allCongruences, congruence)$
    **end if**
  **end for**
  $i \leftarrow i + 1$
**end while**
**return** $allCongruences$
```

The join is only done if the principal congruence is not contained in the congruence that is being joined with. If this is the case, the congruence resulting of the join will be the containing congruence. This can be optimized by keeping the information about which principal partitions were joined to reach a congruence.

3.2.2 Automorphism algorithm

In the scope of this PhD several automorphism algorithms were tested and the conclusion was that the invariants algorithm, similar to what is used in the Loops package, was the most promising. Due to the need to focus other parts of the **CREAM** Package this line of work was stopped and its conclusions were passed to Choiwah Chow that used this knowledge to make the final implementation of the automorphism algorithms using the invariants approach. This implementation was contributed to **CREAM**.

In the automorphisms algorithm, the Loops package was used as guideline for the implementation of the **CreamAutomorphims** function by tweaking the algorithm to accept any magma as input allowing an easy adaptation to support any algebra of type (2^m,1^n).

The invariants algorithm uses characteristics of each element that are invariants under automorphism to reduce the search space for automorphisms. One invariant under automorphism is idempotency, meaning that an automorphism maps idempotents to idempotents and therefore identifying idempotents allows to reduce the search space for automorphisms by searching only for mappings where idempotents map to idempotents and non-idempotents map only to non-idempotents.

Some simple invariants are:

- Distinct Row Elements
- Distinct Column Elements
- Number of Idempotents in a Row
- Number of Idempotents in a Column
- Number of Elements Commuting
- Number of Appearances in Cayley Table

Although based on work done in this PhD the code of the automorphism algorithm included in the **CREAM** Package is not part of this PhD work. This algorithm is described in [5].

3.2.3 Endomorphisms algorithm

This section deals with non-automorphism endomorphisms. Automorphisms are dealt with in section 3.2.2.

MACE4 is used for the calculation of endomorphisms. **MACE4** [27] is a program that searches for finite models of first-order formulas. The use of **MACE4** in **CREAM** is further described in section 3.4. The classic approach to calculate endomorphisms using **MACE4** would be to initiate a search with the algebra's operations encoded in **MACE4**. The algebra:

```
[ [ [ 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 2, 3, 1 ] ] ]
```

would be encoded in **MACE4** as:

```
fa1(0,0)=0. fa1(0,1)=0. fa1(0,2)=0. fa1(0,3)=0. fa1(0,4)=0. fa1(0,5)=0.
fa1(1,0)=0. fa1(1,1)=0. fa1(1,2)=0. fa1(1,3)=0. fa1(1,4)=0. fa1(1,5)=0.
fa1(2,0)=0. fa1(2,1)=0. fa1(2,2)=0. fa1(2,3)=0. fa1(2,4)=0. fa1(2,5)=0.
fa1(3,0)=0. fa1(3,1)=0. fa1(3,2)=0. fa1(3,3)=0. fa1(3,4)=0. fa1(3,5)=0.
fa1(4,0)=0. fa1(4,1)=0. fa1(4,2)=0. fa1(4,3)=0. fa1(4,4)=0. fa1(4,5)=0.
fa1(5,0)=0. fa1(5,1)=0. fa1(5,2)=0. fa1(5,3)=1. fa1(5,4)=2. fa1(5,5)=0.
```

In this encoding $fa1$ is the algebra's operation. Normally **MACE4** input will have one clause per line but here we are showing different clauses in each line for compactness. Please notice that **MACE4** is zero-based unlike GAP that is one-based.

In a zero-based language, the initial element of a sequence is 0 while in one-based languages the initial element of a sequence is 1. The impact in representing algebras is that in an algebra of size 6 is represented with elements $\{0, 1, 2, 3, 4, 5\}$ in a zero-based language and with elements $\{1, 2, 3, 4, 5, 6\}$ in a one-based language

To the algebra operations one clause for each operation will be added such as:

`f(fa1(x))=fa1(f(x)).` , if fa_1 is a unary operation or

`f(fa1(x,y))=fa1(f(x),f(y)).` , if fa_1 is a binary operation.

This classic approach while very effective and fast for low order algebras needs further optimizations for high order algebras in which the congruences of the algebra are used to limit the **MACE4** search space for endomorphisms.

To calculate all the endomorphisms of an algebra of high order, the following steps are followed:

1. Calculate the congruences of algebra A;
2. For each congruence R calculate A/R (except for the singleton congruence);
3. Calculate the subalgebras of A that are isomorphic to A/R (using **MACE4**);
4. For each pair subalgebra/congruence derive the corresponding endomorphism;
5. Calculate the automorphisms of A (that will also be endomorphisms).

The congruences of A are calculated with the algorithms described in 3.2.1.

3. COMPUTATIONAL IMPLEMENTATION

For each congruence R (except for the singleton congruence that will be dealt later), the A/R operation can be obtained just by substituting the values in A operations by its root blocks. For the previously shown algebra the congruence is:

```
[ [ 1, 2, 3], [4], [5, 6] ] - [ -3, 1, 1, -1, -2, 5 ]
```

and the A/R operation is:

```
1 x 1 = 1
1 x 4 = 1
1 x 5 = 1
4 x 1 = 1
4 x 4 = 1
4 x 5 = 1
5 x 1 = 1
5 x 4 = 1
5 x 5 = 1
```

This operation can be encoded in **MACE4** as:

```
fa1(a0,a0)=a0.
a0!=a3.
fa1(a0,a3)=a0.
a0!=a4.
fa1(a0,a4)=a0.
fa1(a3,a0)=a0.
fa1(a3,a3)=a0.
a3!=a4.
fa1(a3,a4)=a0.
fa1(a4,a0)=a0.
fa1(a4,a3)=a0.
fa1(a4,a4)=a0.
```

Running **MACE4** with the definition of the algebra operation and the above encoding of A/R as assumptions we get 16 possible models each one of them corresponding to a different endomorphism of the algebra A. One of these models would be:

```
interpretation( 6, [number = 1,seconds = 0], [
    function(a0, [0]),
    function(a3, [1]),
    function(a4, [2]),
    function(fa1(_,_), [
        0,0,0,0,0,0,
        0,0,0,0,0,0,
        0,0,0,0,0,0,
        0,0,0,0,0,0,
        0,0,0,0,0,0,
        0,0,0,1,2,0])]).
```

This means that this model corresponds to the mapping $\{1 \to 1, 4 \to 2, 5 \to 3\}$ which can be represented as:

```
[ 1, , , 2, 3, ]
```

The missing mappings of the endomorphism can be easily deduced from the congruence since elements of the same block must map to the same element. In this example the mapping of 2 and 3 needs to be the same as 1 and the mapping of 6 needs to be the same as 5 and the missing mappings would then be $\{2 \to 1, 3 \to 1, 6 \to 3\}$ yielding the endomorphism:

```
[ 1, 1, 1, 2, 3, 3]
```

Repeating this process for all congruences and for all models obtained with **MACE4** from each congruence, all the algebra's endomorphisms (except for the automorphisms) can be obtained.

Finally, for the singleton congruence the associated endomorphisms would be also automorphisms and in this case is more efficient to use the algorithm described in 3.2.2 than the one using **MACE4**.

3.2.4 Monomorphism algorithm

The monomorphism algorithm implemented in **CREAM** uses invariants in a similar way to the automorphism algorithm.

Invariants can be used to speed up the process of finding a monomorphism from one magma to another, or from one algebra to another. If f is an injective homomorphism from a magma A to a magma B, then it is a isomorphism from A to B restricted to the range of f. Thus, the same ideas of applying invariants in constructing automorphisms can be used for constructing monomorphisms. Specifically, an monomorphism f can map an element $a \in A$ to $b \in B$ only if a and b have the same invariant vector. This greatly reduces the search space for monomorphisms between A and B.

Again and although it was based on work done in this PhD the code of the monomorphism algorithm included in the **CREAM** Package is not part of this PhD work. This algorithm is described in [5].

The **CREAM** package includes the functions **CreamExistsMonomorphism** (returning true if a monomorphism exists) and **CreamOneMonomorphism** (returning one monomorphism) and **CreamAllMonomorphisms** (returning all monomorphisms)

3.2.5 Epimorphism algorithm

CreamAllEpimorphisms(A_1, A_2) returns all epimorphisms from A_1 to A_2, provided the algebras are compatible. The algorithm first finds all congruences φ of A_1 such that A_1/φ and A_2 are isomorphic. For each such φ, the corresponding epimorphisms are obtained by composing the quotient map with an isomorphism to A_2 and all automorphisms of A_2.

For efficiency, a size check is implemented before searching for isomorphisms and the automorphisms of A_2 are only calculated once.

Algorithm 10 CreamAllEpimorphisms(algebra1, algebra2)

```
$allCongruences \leftarrow CreamAllCongruences(algebra1)$
$dimension1 \leftarrow CreamSizeAlgebra(algebra1)$
$dimension2 \leftarrow CreamSizeAlgebra(algebra2)$
$autoList \leftarrow []$
$epiList \leftarrow []$
for all $cong$ in $allConguences$ do
  if $dimension2 = CreamNumberOfBlocks(cong)$ then
    $[qalgebra, mapToQalgebra] \leftarrow QuotientAlgebraFromCongruence(algebra1, cong)$

    $iso \leftarrow IsomorphismAlgebras(qalgebra, algebra2)$
    if not $iso = fail$ then
      if autoList = [] then
        $autoList \leftarrow CreamAutomorphisms(algebra2)$
      end if
      for all $auto$ in $autoList$ do
        $epi \leftarrow []$
        for $j = 1$ to $dimension1$ do
          $epi[j] \leftarrow auto[iso[mapToQalgebra[i]]]$
        end for
        Add $epi$ to $epiList$ if not already in the list
      end for
    end if
  end if
end for
return $epiList$
```

In this section only the function **CreamAllEpimorphisms** is addressed but the **CREAM** package also includes the functions **CreamExistsEpimorphism** (returning true if an epimorphism exists) and **CreamOneEpimorphism** (returning one epimorphism).

3.2.6 SubUniverses algorithm

The SubUniverses algorithm returns a list of all underlying sets of all subalgebras of the input algebra. It first generates all 1-generated subalgebras, then iteratively expands each i-generated algebra by adding another generator.

For performance enhancement, these expansions are limited to one element from each orbit of the algebra's automorphism group. At the end of each cycle, the remaining $(i+1)$-generated subuniverses are obtained by applying the automorphism group. The algorithm stops when an iteration produces only one subuniverse with the same size of the algebra.

The algorithm relies on the **SubUniverseFromElement** routine, which calculates the subalgebra generated by a subalgebra and an additional element. This routine assumes that the input subuniverse is closed under the algebra's operations. The algorithm adds the element to the updated subuniverse, calculates all directly generated additional elements, and places them in a temporary list. These elements are then filtered for those that are already in the subuniverse, and the remaining elements are put into a second list. Elements from the second list are then added to the subuniverse iteratively.

The function **SizeSubUniverse** takes a subuniverse as argument and returns the number of elements of the subset underlying to the subuniverse.

Algorithm 11 CreamAllSubUniverses(algebra)

```
$dimension \leftarrow CreamSizeAlgebra(algebra)$
$autoList \leftarrow CreamAutomorphisms(algebra)$
**for** $i = 1$ to $dimension$ **do**
    $sigma[i] \leftarrow []$
    **if** $i = 1$ **then**
        $sigmaMinus \leftarrow [[]]$
    **else**
        $sigmaMinus \leftarrow sigmaExpanded[i-1]$
    **end if**
    **for all** $cSigma$ in $sigmaMinus$ **do**
        $elemList \leftarrow [1..dimension]$
        Remove all elements in $cSigma$ from $elemList$
        **while** $NumberOfElements(elemList) <> 0$ **do**
            $j \leftarrow elemList[1]$
            $sUniverse \leftarrow SubUniverseFromElement(algebra, cSigma, j)$
            Add $sUniverse$ to $sigma[i]$ if not already in the list
            $autoEs \leftarrow$ the orbit of $j$ under $autoList$
            Remove all elements in $autoEs$ from $elemList$
        **end while**
    **end for**
    $sigmaExpanded[i] \leftarrow []$
    **for all** $sUniverse$ in $sigma[i]$ **do**
        $autoUniverses \leftarrow$ the orbit of $sUniverse$ under $autoList$
        Add all elements of $autoUniverses$ to $sigmaExpanded[i]$ if not already in the list
    **end for**
    **if** $NumberOfElements(sigma[i]) = 1$ and $SizeSubUniverse(sigma[i][1]) = dimension$ **then**
        $break$
    **end if**
**end for**
$sUniverses \leftarrow \cup_{j=1}^{i} sigmaExpended[j]$
**return** $sUniverses$
```

Algorithm 12 SubUniverseFromElement(algebra, initSubUniverse, element)

```
$dimension \leftarrow CreamSizeAlgebra(algebra)$
$elemList \leftarrow [element]$
$newSubUniverse \leftarrow initSubUniverse$
$nElem \leftarrow NumberOfElements(elemList)$
**while** $nElem > 0$ **do**
  $currentElem \leftarrow elemList[nElem]$
  Remove element in position $nElem$ from $elemList$
  $newElemList \leftarrow []$
  **for all** operations $op$ of the $algebra$ **do**
    **if** $op$ is binary **then**
      Add $op[currentElem][currentElem]$ to $newElemList$ if not already in the list
      **for all** $subUniverseElem$ in $newSubUniverse$ **do**
        Add $op[currentElem][subUniverseElem]$ to $newElemList$ if not already in the list
        Add $op[subUniverseElem][currentElem]$ to $newElemList$ if not already in the list
      **end for**
    **else**
      Add $op[currentElem]$ to $newElemList$ if not already in the list
    **end if**
  **end for**
  Add $currentElem$ to $newSubUniverse$ if not already in the list
  $elemList \leftarrow elemList \cup (newElemList \setminus newSubUniverse)$
  $nElem \leftarrow NumberOfElements(elements)$
  **if** $SizeOfSubUniverse(newSubUniverse) + nElem = dimension$ **then**
    $newSubUniverse \leftarrow [1..dimension]$
    $nElem \leftarrow 0$
  **end if**
**end while**
**return** $newSubUniverse$
```

3.2.7 DivisorUniverses algorithm

CreamAllDivisorUniverses(A_1, A_2) checks if A_1 has a divisor that is isomorphic to A_2, where A_1 and A_2 are compatible algebras. Its return is a list of all pairs (B, φ), such that B is a subuniverse of A_1, φ is a congruence of the subalgebra B, and B/φ is isomorphic to A_2.

The algorithm first calculates the subalgebras of A_1, then their quotients and then checks those for isomorphisms to A_2, in each case using the corresponding algorithms of the

CREAM package. A size condition is used to prune unnecessary calculations.

Algorithm 13 CreamAllDivisorUniverses(algebra1, algebra2)

```
$dimension \leftarrow CreamSizeAlgebra(algebra2)$
$subUniverseList \leftarrow CreamSubUniverses(algebra1)$
$divisorList \leftarrow []$
for all $subUniverse$ in $subUniverseList$ do
    if $dimension <= SizeOfSubUniverse(subUniverse)$ then
        $subAlgebra \leftarrow CreamSubUniverse2Algebra(algebra1, subUniverse)$
        $congList \leftarrow CreamAllCongruences(subAlgebra)$
        $rcongList \leftarrow []$
        for all $cong$ in $congList$ do
            if $dimension = CreamNumberOfBlocks(cong)$ then
                Add $cong$ to $rcongList$ if not already in the list
            end if
        end for
        for all $cong$ in $rcongList$ do
            $[qAlgebra, mapToQalgebra] \leftarrow QuotientAlgebraFromCongruence(subAlgebra, cong)$

            if $CreamAreAlgebrasIsomorphic(qAlgebra, algebra2)$ then
                Add $[subUniverse, cong]$ to $divisorList$ if not already in the list
            end if
        end for
    end if
end for
return $divisorList$
```

In this section only the function **CreamAllDivisorUniverses** is addressed but the **CREAM** package also includes the functions **CreamExistsDivisor** (returning true if a divisor between the algebras exists) and **CreamOneDivisorUniverse** (returning one divisor between the algebras).

3.2.8 DirectlyReducible algorithm

CreamAllDirectlyReducible(A) looks for two (non-trivial) commuting congruences ϕ, ψ of A, whose meet is trivial and whose join is $A \times A$. If those exist, A is isomorphic to the direct product $A/\phi \times A/\psi$.

Because all algebras are finite, it suffices to check that the congruences have a trivial meet, and that the corresponding product algebra has the correct size. The algorithm uses numerical constraints to avoid unnecessary calculations.

The return of the algorithm is a list of pairs of congruences. These pairs should be consider as sets, i.e. if (α, β) is listed the (equally valid) pair (β, α) is not listed separately.

Algorithm 14 CreamAllDirectlyReducible(algebra)

```
$dimension \leftarrow CreamSizeAlgebra(algebra)$
$congList \leftarrow CreamAllCongruences(algebra)$
$pCongList \leftarrow []$
for all $cong$ in $congList$ do
  $cNBlocks \leftarrow CreamNumberOfBlocks(cong)$
  if $cNBlocks <> 1$ and $cNBlocks <> dimension$ and $IsInt(dimension/cNBlocks)$ then
    Add $cong$ to $pCongList$
  end if
end for
$pairs \leftarrow []$
for all 2-element subsets $\{i, j\}$ of $pCongsList$ do
  $iNBlocks \leftarrow CreamNumberOfBlocks(i)$
  $jNBlocks \leftarrow CreamNumberOfBlocks(j)$
  if $iNBlocks * jNBlocks = dimension$ then
    $cMeet \leftarrow MeetPartition(i, j)$
    $cMeetNBlocks \leftarrow CreamNumberOfBlocks(cMeet)$
    if $cMeetNBlocks = dimension$ then
      Add $[i, j]$ to $pairs$
    end if
  end if
end for
return $pairs$
```

In this section only the function **CreamAllDirectlyReducible** is addressed but the **CREAM** package also includes the functions **CreamExistsDirectlyReducible** (returning true if there are commuting congruences R, T where A/R x A/T is isomorphic to A) and **CreamOneDirectlyReducible** (returning one pair of commuting congruences R, T where A/R x A/T is isomorphic to A).

3.3 The CREAM Package

CREAM stands for “Algebra **C**ong**R**uences, **E**ndomorphisms and **A**uto**M**orphisms”. **CREAM** is a GAP Package developed in GAP and C. GAP stands for “**G**roups, **A**lgorithms, **P**rogramming” and is a system for computational discrete algebra, with particular emphasis on Computational Group Theory. A detailed description of the **CREAM** Package is presented in the Appendix A.

3.4 MACE4

MACE4 [27] is a command line application that searches for finite models of first-order formulas and statements.

MACE4 is used by the package **CREAM** but was not developed or changed in the scope of the work of this PhD. Despite not being part of the PhD this section about **MACE4** was added to contextualize its use in the package **CREAM**.

MACE4 uses the same formulas and statements syntax as Prover9 [28]. **MACE4** takes into account two sets of clauses: assumptions and goals. Goal clauses are negated by **MACE4**.

MACE4 can be used either to find models that verify a set of clauses or to find counterexamples.

The domain size can be set and the search can be limited by time or number of models or the search will be done until exhausting all possible models.

The **MACE4** version being used is an optimized rebuild done by Carlos Sousa and Choiwah Chow in the scope of their PhD thesis work that can be found in `https://gitlab.com/cfmsousa/prover9`.

MACE4 is used internally by **CREAM** for 3 purposes:

- Generating algebras of a specific type from axioms (e.g group, semigroup, etc).
- Determining whether 2 algebras are isomorphic.
- Determining the endomorphisms of an algebra.

The use of **MACE4** for the determination of endomorphisms is described in section 3.2.3.

MACE4 can be used to search for algebras of defined types within a domain. For instance we can determine all groups of a determined order by adding as **MACE4** clauses the group axioms.

```
(x * y) * z = x * (y * z). % associativity
1 * x = x. % left identity
x * 1 = x. % right identity
x' * x = 1. % lef inverse
x * x' = 1. % right inverse
```

For a domain of size 4, 3 models are returned.

```
interpretation( 4, [number = 1,seconds = 0], [
    function(*(_,_), [
        1,0,3,2,
        0,1,2,3,
        3,2,1,0,
        2,3,0,1]),
    function('(_), [0,1,2,3])]).
interpretation( 4, [number = 2,seconds = 0], [
    function(*(_,_), [
        1,0,3,2,
        0,1,2,3,
        3,2,0,1,
        2,3,1,0]),
    function('(_), [0,1,3,2])]).
interpretation( 4, [number = 3,seconds = 0], [
    function(*(_,_), [
        3,0,1,2,
        0,1,2,3,
        1,2,3,0,
        2,3,0,1]),
    function('(_), [2,1,0,3])]).
```

Some of these models might be isomorphic. To determine whether 2 algebras corresponding models returned by **MACE4** we need to determine whether a bijective mapping exists between the algebras that preserves the operations of the algebraic structures.

This can be done by defining the algebras' operations as **MACE4** clauses adding clauses for a structure preserving bijective mapping between them. In the following example this was done for the second and the third models returned by **MACE4**.

```
fs(0,0)=1.fs(0,1)=0.fs(0,2)=3.fs(0,3)=2.
fs(1,0)=0.fs(1,1)=1.fs(1,2)=2.fs(1,3)=3.
fs(2,0)=3.fs(2,1)=2.fs(2,2)=0.fs(2,3)=1.
fs(3,0)=2.fs(3,1)=3.fs(3,2)=1.fs(3,3)=0.

ft(0,0)=3.ft(0,1)=0.ft(0,2)=1.ft(0,3)=2.
ft(1,0)=0.ft(1,1)=1.ft(1,2)=2.ft(1,3)=3.
ft(2,0)=1.ft(2,1)=2.ft(2,2)=3.ft(2,3)=0.
ft(3,0)=2.ft(3,1)=3.ft(3,2)=0.ft(3,3)=1.

f(fs(x,y)) = ft(f(x),f(y)).
f(x) = f(y) -> x = y.
```

There is one bijective mapping $f(x)$ between these algebras:

```
interpretation( 4, [number = 1,seconds = 0], [
    function(f(_), [3,1,0,2]),
    function(fs(_,_), [
        1,0,3,2,
        0,1,2,3,
        3,2,0,1,
        2,3,1,0]),
    function(ft(_,_), [
        3,0,1,2,
        0,1,2,3,
        1,2,3,0,
        2,3,0,1])]).
```

The fact that there is a bijective mapping between the groups means that the 2 groups are isomorphic.

When this is done for the first 2 groups:

```
fs(0,0)=1.fs(0,1)=0.fs(0,2)=3.fs(0,3)=2.
fs(1,0)=0.fs(1,1)=1.fs(1,2)=2.fs(1,3)=3.
fs(2,0)=3.fs(2,1)=2.fs(2,2)=1.fs(2,3)=0.
fs(3,0)=2.fs(3,1)=3.fs(3,2)=0.fs(3,3)=1.

ft(0,0)=1.ft(0,1)=0.ft(0,2)=3.ft(0,3)=2.
ft(1,0)=0.ft(1,1)=1.ft(1,2)=2.ft(1,3)=3.
ft(2,0)=3.ft(2,1)=2.ft(2,2)=0.ft(2,3)=1.
ft(3,0)=2.ft(3,1)=3.ft(3,2)=1.ft(3,3)=0.

f(fs(x,y)) = ft(f(x),f(y)).
f(x) = f(y) -> x = y.
```

No models are found.

```
============================== DOMAIN SIZE 4 =========================

============================== STATISTICS ============================

For domain size 4.

Current CPU time: 0.00 seconds (total CPU time: 0.00 seconds).
Ground clauses: seen=64, kept=60.
Selections=2, assignments=8, propagations=38, current_models=0.
Rewrite_terms=195, rewrite_bools=56, indexes=0.
Rules_from_neg_clauses=2, cross_offs=20.

============================== end of statistics =====================

User_CPU=0.00, System_CPU=0.06, Wall_clock=0.

Exiting with failure.

Process 26688 exit (exhausted) Wed Jun 23 10:20:26 2021
The process finished Wed Jun 23 10:20:26 2021
```

This means that these 2 groups are not isomorphic.

3.5 Performance

3.5.1 Improving Performance

The code calculating congruences was originally developed in **GAP**. This allowed a fast prototyping of the algorithms and a rapid detection and correction of issues.

While **GAP** is adequate for fast prototyping and implementation of algorithms the resulting code is not very fast because it is an interpreted language. **GAP** supports what is known internally as kernel modules that allow the development of C code that can be called within **GAP**. This code is compiled and generally shows a much better performance that the interpreted **GAP** code.

In order to have the best balance between performance gain and development effort the code was profiled in order to identify which functions were taking more CPU time. Prioritizing the implementation in C of those functions would give us the biggest performance gains with minimal development effort.

Please note that some of the following transcripts were obtained before the final names for the functions and package were defined. Due to this, the function names are different from the current ones.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (4))];;
gap> ProfileFunctions(internalPartitionRoot, blockJoin, internalMinimalCongrue$
gap> allCongruences(algebra);;
gap> DisplayProfile();
  count  self/ms  chld/ms  stor/kb  chld/kb  package  function
   5922      104       84        0        0  Algebr*  blockJoin
    282       76      172        0        0  Algebr*  normalizePartition
1659681    16316       52        0        0  Algebr*  internalRootPartition
    276     3616    16512      353     3152  Algebr*  internalMinimalCongru*
      1       12    20132        6     3514  Algebr*  internalAllMinimalCon*
      1        0    20160        0     3522  Algebr*  allCongruences
      1        0    20160        0     3522  Algebr*  internalAllCongruences
              36              3162                    OTHER
           20160              3522                    TOTAL
```

From the profiling data, the most consuming functions revealed to be:

- **CreamRootBlock** (shown as **internalRootPartition**) – 1659681 runs spending a total of 16316 ms (∼81%) runtime
- **CreamPrincipalCongruence** (shown as **minimalCongruence**) – 276 runs spending a total of 3616 ms (∼18%) runtime
- **CreamJoinBlocks** (shown as **blockJoin**) – 5922 runs spending a total of 104 ms (<1%) runtime
- **CreamNormalizePartition** (shown as **normalizePartition**) – 282 runs spending a total of 76 ms (<1%) runtime

The most time consuming functions are **CreamRootBlock** and **CreamPrincipalCongruence** that account for nearly 99% of the runtime and these were rewritten in C. The time consumption of **CreamJoinBlocks** and **CreamNormalizePartition** is negligible (<1%) but since they are called inside **CreamPrincipalCongruence** they were rewritten in C also.

The performance results of running the code with S_4 and S_5 before this rewrite can be seen in transcript below.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (4))];;
gap> allCongruences(algebra);;time;
484
gap> algebra:= [MultiplicationTable(SymmetricGroup (5))];;
gap> allCongruences(algebra);;time;
481000
```

The calculation of the congruences of S_4 took 484ms to calculate while the calculation of the congruences of S_5 took 481000s.

The performance results of running the code with S_4 and S_5 after this rewrite can be seen in transcript below.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (4))];;
gap> allCongruences(algebra);;time;
16
gap> algebra:= [MultiplicationTable(SymmetricGroup (5))];;
gap> allCongruences(algebra);;time;
9636
```

This yielded 16ms for the calculation of the congruences of S_4 and 9636ms for S_5

These rewrites achieved a staggering improvement yielding a 30-fold improvement (from 484ms to 16ms) in calculating the congruences for S_4 and a 50-fold improvement (from 481000ms to 9636ms) in calculating the congruences for S_5.

The following step in terms of the performance improvement process was to rewrite the full **CreamAllCongruences** and **CreamAllPrincipalCongruences** in C. After this full rewrite further improvements were obtained.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (4))];;
gap> CreamAllCongruences (algebra);;time;
4
gap> algebra:= [MultiplicationTable(SymmetricGroup (5))];;
gap> CreamAllCongruences (algebra);;time;
716
```

With this full rewrite a total 120-fold improvement (from 484ms to 4ms) was obtained for S_4 and a 670-fold improvement (from 481000ms to 716ms) for S_5.

3.5.2 Congruences algorithm performance

The analysis of the theoretical time complexity of the Freese algorithm was done based on the paper data [15] and compared with the graph algorithm's [11] complexity analysed in section 2.3.

Considering unary algebras, for the sake of simplicity, the analysis of the Freese algorithm presents a complexity of $O(|\mathcal{F}|n)$ where $n = |A|$, to calculate a principal congruence $Cg^A(x, y)$. Taking into account that there will be $\binom{n}{2} = \frac{n(n-1)}{2}$ unordered pairs for which to calculate $Cg^A(x, y)$ then all principal congruences can be calculated in a maximum of $O(|\mathcal{F}|n^3)$.

For the graph algorithm the complexity of calculating the graph is indicated to be $O(|\mathcal{F}|n^2)$. At first look the graph algorithm has a lesser complexity, growing slower with n but after running the graph algorithm it would still be necessary to build the congruences from the graph. Even if some optimizations from the Freese algorithm are used to build the congruences from the graph the time complexity of doing this would be bounded by $O(log_2(n^2)n^4)$. This makes both algorithms comparable if $|\mathcal{F}|$ grows proportional to $n\log(n^2)$, with an advantage for the Freese algorithm for slower growth and an advantage for the graph algorithm otherwise.

3. COMPUTATIONAL IMPLEMENTATION

One of the advantages of the Freese's algorithm is the use of data structures that facilitate the traversing of the graph of unordered pairs without having to build the graph explicitly. Moreover, the main time consumption in the Freese's algorithm is the congruence generation and special attention was taken to the performance of the algorithms building partitions (congruences) from unordered pairs.

The performance of the congruence algorithm implementation in **CREAM** was tested against different congruence implementations. For these tests the algebras were considered as universal algebras of type (2^1) even in cases were these algebras can have definitions with different types.

The **semigroup GAP** package includes the function **CongruencesOfSemigroup** that returns the congruences for a semigroup. This function does not work for all semigroups and in the case of the semigroups listed in the **smallsemi** package we determined that it works for the semigroups that belong to one of the following classes (see section 2.4 for definitions on these algebras):

- Simple
- Brandt
- Group
- Zero Simple
- Rectangular Band

For these semigroups we've run the **CongruencesOfSemigroup** function and the **CreamAllCongruences** 5 times and calculated the average runtime after removing the best and worst runtimes getting the following results:

Table 3.1: Performance Comparison between **CongruencesOfSemigroup** and **CREAM** on calculating all congruences.

Size	Number of Algebras	CongruencesOfSemigroup(ms)	CreamAllCongruences(ms)
1	1	7	< 1
2	3	20	< 1
3	4	29	< 1
4	7	41	< 1
5	9	53	< 1
6	8	71	< 1
7	12	153	4
8	14	484	35
1-8	58	858	40

So although the number of algebras for which **CongruencesOfSemigroup** works is very limited even for those algebras **CreamAllCongruences** is more than 20 times faster.

During testing it was found that there is a very specific type of semigroup where the function **CongruencesOfSemigroup** achieves significatively better results than **CreamAllCongruences**. These are the Rees Matrix Semigroups or the Rees Zero Matrix Semigroups. These groups are Simple Semigroups or Zero Simple Semigroups. But to achieve this kind of performance the semigroup needs to be created using the GAP functions **ReesMatrixSemigroup** or **ReesZeroMatrixSemigroup**. If otherwise, the semigroup is created from its multiplication table while being isomorphic to a Rees Matrix Semigroup or a Rees Zero Matrix Semigroup the performance will be similar to what was found above. See in the following table the average runtime for 10 runs with random Rees Matrix Semigroups:

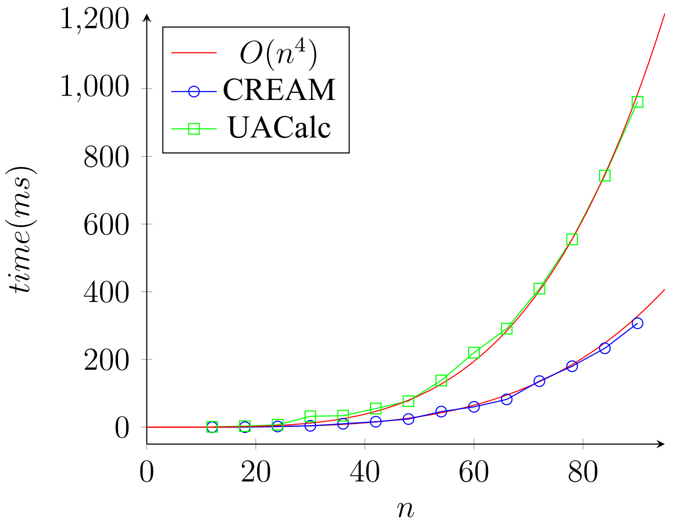
Table 3.2: Performance Comparison on calculating all congruences of Rees Matrix Semigroup.

Size	CongruencesOfSemigroup (ReesMatrixSemigroup) (ms)	CongruencesOfSemigroup (MultiplicationTable) (ms)	CreamAllCongruences (ms)	UACalc (ms)
12	1	15	< 1	1
18	2	28	< 1	3
24	5	55	2	7
30	1	92	4	32
36	4	139	10	34
42	11	203	16	55
48	8	296	24	77
54	5	416	46	138
60	5	553	60	220
66	1	671	82	291
72	9	873	136	409
78	2	1 098	180	555
84	5	1 341	233	743
90	4	1 622	307	961

For the Rees Matrix Semigroups or the Rees Zero Matrix Semigroups the function **CongruencesOfSemigroup** takes advantage of its particular structure to apply in a very efficient way the use of the linked triple algorithm but this can only be used in a very restrict setting and with a very limited set of semigroups.

UACalc is the implementation of the Freeze algorithm done under his supervision in java and jython. It includes a GUI version in Java and a command line interface in jython. The performance of the GUI version is poor and we will not be looking into its performance. On the other hand the command line interface has a performance that is comparable to **CreamAllCongruences**. The same random Rees Matrix Semigroups that were used with **CongruencesOfSemigroup** and **CreamAllCongruences** were written into files readable by **UACalc** and the congruences were calculated in **UACalc**. **CreamAllCongruences** is consistently more than 3 times faster than **UACalc**. For these algebras with one binary operation the runtime rises roughly proportionally to n^4 (where n is the size of the algebra) or t^2 (being t the number cells of the multiplication matrix of the algebra operation).

Figure 3.4: Performance Comparison on calculating all congruences of Rees Matrix Semigroup



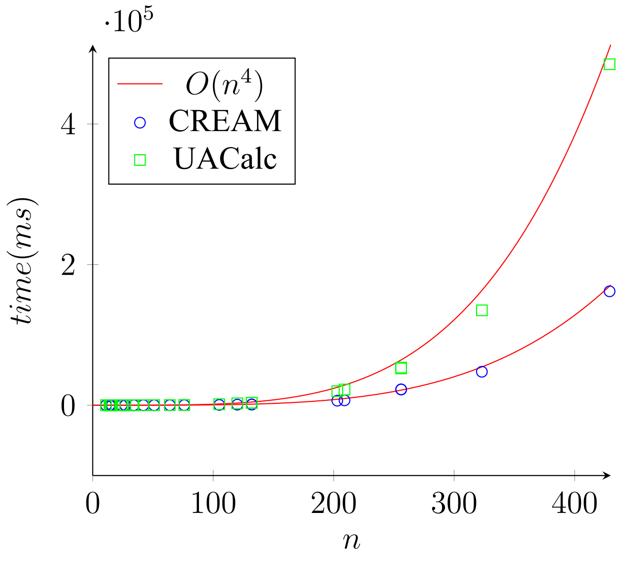
To further compare the performance of **CreamAllCongruences** with **UACalc** several specific types of Semigroups, Monoids and Groups were run.

Table 3.3: Performance Comparison between CREAM and UACalc

Algebra	Size	Number of Congruences	CreamAllCongruences (ms)	UACalc (ms)
GossipMonoid(3)	11	84	1	2
PlanarPartitionMonoid(2)	14	9	1	1
JonesMonoid(4)	14	9	< 1	1
BrauerMonoid(3)	15	7	< 1	2
PartitionMonoid(2)	15	13	< 1	1
FullPBRMonoid(1)	16	167	3	4
SymmetricGroup(4)	24	4	3	10
FullTransformationSemigroup(3)	27	7	3	14
FullTransformationMonoid(3)	27	7	3	7
SymmetricInverseMonoid(3)	34	7	6	25
JonesMonoid(5)	42	6	12	36
MotzkinMonoid(3)	51	10	23	73
PartialTransformationMonoid(3)	64	7	63	223
PartialBrauerMonoid(3)	76	16	117	351
BrauerMonoid(4)	105	19	430	1 420
SymmetricGroup(5)	120	3	812	2 700
PlanarPartitionMonoid(3)	132	10	1 094	3 490
JonesMonoid(6)	132	10	950	3 444
PartitionMonoid(3)	203	16	6 500	19 979
SymmetricInverseMonoid(4)	209	11	6 970	22 318
FullTransformationSemigroup(4)	256	11	22 529	53 557
FullTransformationMonoid(4)	256	11	22 385	52 361
MotzkinMonoid(4)	323	11	47 593	134 945
JonesMonoid(7)	429	7	161 987	485 004

The performance for these algebras is mostly similar to what was found for the Rees Matrix Semigroups with a bigger variance in the results in the sense that the average of **CreamAllCongruences** over **UACalc** is between 3 and 3,5 times but depending on the algebras we get improvements spanning from 2 times to 4,5 times. The runtime also rises roughly proportionally to n^4 (although with more variance).

Figure 3.5: Performance Comparison between CREAM and UACalc



3.5.3 Automorphisms algorithm performance

The automorphism algorithm included in the **CREAM** Package was developed by Choiwah Chow and is not part of this PhD work. Its performance is described in [5].

3.5.4 Endomorphisms algorithm performance

To test the performance of the algorithms to calculate endomorphisms, the algorithms were run with specific types of Semigroups, Monoids and Groups and using the classic and congruence algorithms running with **MACE4**. As described in 3.2.3 the classic algorithm uses only **MACE4** to calculate endomorphisms while the congruence algorithm uses the algebra congruences to limit the search space of **MACE4**. The classic algorithm is used in **CREAM** for algebras of size up to 60 while the congruence algorithm is used for algebras of size above 60. The only **GAP** function to calculate endomorphisms that came to our attention is the function Endomorphisms from the package **SONATA**. This function can calculate endomorphisms for a very narrow set of algebras, namely groups and near-rings.

The groups in the list algebras were also run with **SONATA**.

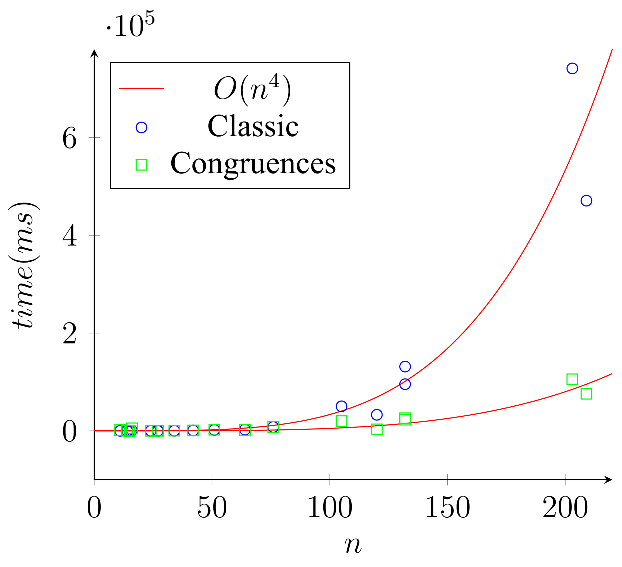
As for congruences, in these tests the algebras were considered as universal algebras of type (2^1) even in cases were these algebras can have definitions with different types.

Table 3.4: Performance Comparison between endomorphisms calculation algorithms

Algebra	Size	Number of Endomorphisms	Classic (ms)	Congruences (ms)	SONATA (ms)
GossipMonoid(3)	11	66	30	1 635	NA
PlanarPartitionMonoid(2)	14	72	56	233	NA
JonesMonoid(4)	14	72	45	232	NA
BrauerMonoid(3)	15	28	38	160	NA
PartitionMonoid(2)	15	89	50	313	NA
FullPBRMonoid(1)	16	1 426	585	5 134	NA
SymmetricGroup(4)	24	58	101	142	166
FullTransformationSemigroup(3)	27	40	116	364	NA
FullTransformationMonoid(3)	27	40	112	360	NA
SymmetricInverseMonoid(3)	34	54	274	504	NA
JonesMonoid(5)	42	113	746	777	NA
MotzkinMonoid(3)	51	98	1 972	2 400	NA
PartialTransformationMonoid(3)	64	138	2 586	1 963	NA
PartialBrauerMonoid(3)	76	165	7 897	7 796	NA
BrauerMonoid(4)	105	274	50 164	19 875	NA
SymmetricGroup(5)	120	146	32 861	2 979	201
PlanarPartitionMonoid(3)	132	393	95 460	25 803	NA
JonesMonoid(6)	132	393	131 472	22 889	NA
PartitionMonoid(3)	203	687	741 441	105 421	NA
SymmetricInverseMonoid(4)	209	282	470 901	75 940	NA

In this test the classic algorithms not using congruences is faster than using congruences for algebras up to size 60 but for larger algebras its runtime raises very fast and the algorithm using congruences becomes faster.

Figure 3.6: Endomorphisms performance



For groups the **SONATA** package achieves very good results but covering a very narrow set of algebras.

Considering these results the **CreamEndomorphisms** function uses the classic algorithm for algebras with size up to 60 and the congruences algorithm for 60 or larger algebras.

Chapter 4

Applications

4.1 Monolithic Algebras

A monolith is a congruence that is contained in every other congruence except the identity congruence. An algebra with a monolith is said to be a monolithic algebra.

As an example the semigroup:

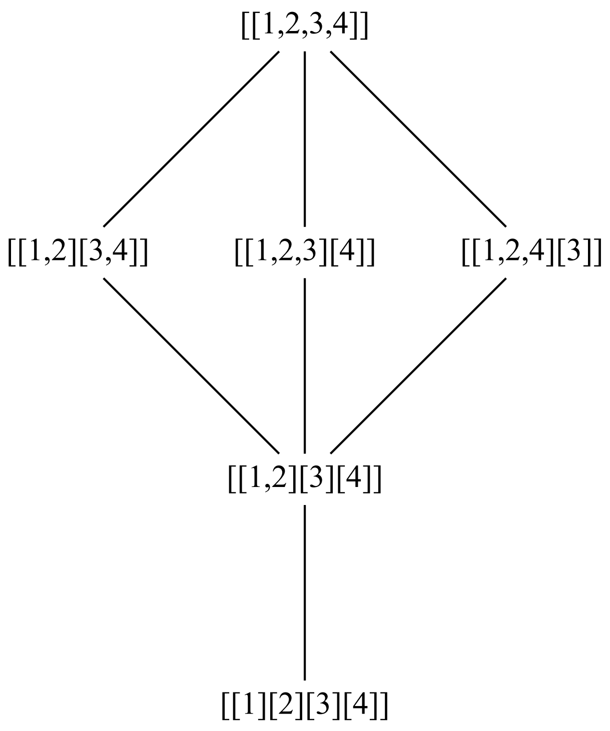
```
[ [ [ 1, 1, 1, 1 ],
    [ 1, 1, 1, 1 ],
    [ 1, 1, 1, 1 ],
    [ 1, 1, 2, 1 ] ] ]
```

is monolithic:

```
gap> algebra := [RecoverMultiplicationTable(4,2)];
[ [ [ 1, 1, 1, 1 ], [ 1, 1, 1, 1 ], [ 1, 1, 1, 1 ], [ 1, 1, 2, 1 ] ] ]
gap> CreamAllCongruences(algebra);
[ [ -1, -1, -1, -1 ], [ -2, 1, -1, -1 ], [ -2, 1, -2, 3 ], [ -3, 1, 1, -1 ],
  [ -3, 1, -1, 1 ], [ -4, 1, 1, 1 ] ]
gap> CreamIsAlgebraMonolithic(algebra);
true
```

This can be confirmed visually if the congruences of this semigroup are depicted as a graph and it can be seen that the congruence [[1,2][3][4]] is contained in every other congruence except for the singleton congruence confirming that the semigroup is monolithic.

Figure 4.1: Congruences of a monolithic algebra



On the other hand the semigroup:

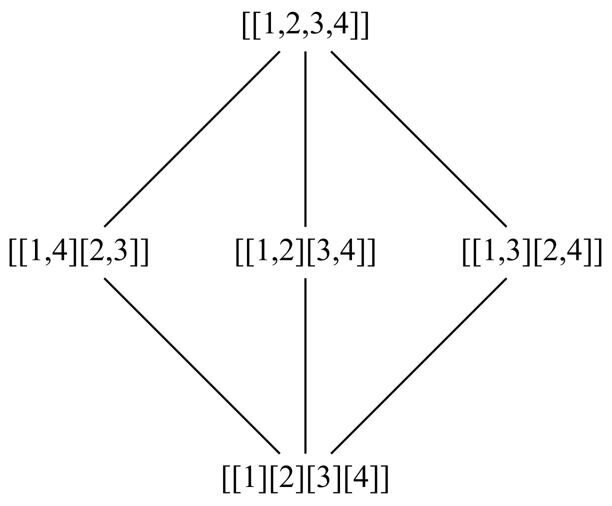
```
[ [ [ 1, 2, 3, 4 ],
    [ 2, 1, 4, 3 ],
    [ 3, 4, 1, 2 ],
    [ 4, 3, 2, 1 ] ] ]
```

is not monolithic:

```
gap> algebra := [RecoverMultiplicationTable(4,7)];
[ [ [ 1, 2, 3, 4 ], [ 2, 1, 4, 3 ], [ 3, 4, 1, 2 ], [ 4, 3, 2, 1 ] ] ]
gap> CreamAllCongruences(algebra);
[ [ -1, -1, -1, -1 ], [ -2, 1, -2, 3 ], [ -2, -2, 2, 1 ], [ -2, -2, 1, 2 ],
  [ -4, 1, 1, 1 ] ]
gap> CreamIsAlgebraMonolithic(algebra);
false
```

Moreover, looking at the congruences graph it can be seen that none of the congruences except for the singleton congruence is contained in every other congruence confirming that the semigroup isn't monolithic.

Figure 4.2: Congruences of a non-monolithic algebra



It is simple to calculate whether an algebra in monolithic having the list of minimal congruences and a function that compares two partitions and calculates whether one is contained in the other. One congruence is contained in another if every element in same block in the first congruence will also be in the same block in second congruence.

The **CreamContainedPartition** function returns whether a partition is contained in another. A partition is contained in another if each of its blocks is contained in a block of the other partition. This function is used internally to determine whether an Algebra is monolithic or not. This algorithm assumes that the partitions are normalized.

Algorithm 15 CreamContainedPartition (partition1, partition2)

```
$dimension \leftarrow SizeOfSet(partition1)$
for $i = 1$ to $dimension$ do
  if $partition1[i] < 0$ then
    if $partition2[i] < 0$ then
      $block2 \leftarrow i$
    else
      $block2 \leftarrow partition2[i]$
    end if
    for $j = i + 1$ to $dimension$ do
      if $partition1[j] = i$ and not $block2 = partition2[j]$ then
        return $false$
      end if
    end for
  end if
end for
return $true$
```

Having this we can calculate the minimal congruences and compare them, eliminating those that contain other congruences. If a single congruence is returned then this congruence is contained in every other congruence (and the algebra is monolithic). This is done with the **CreamIsAlgebraMonolithic** function.

Algorithm 16 CreamIsAlgebraMonolithic (algebra)

```
$partitions \leftarrow CreamAllPrincipalCongruences(algebra)$
$npartitions \leftarrow NumberOfElements(partitions)$
**if** $npartitions > 1$ **then**
  $i \leftarrow 2$
  **repeat**
    **if** $CreamContainedPartition(partitions[1], partitions[i])$ **then**
      $Remove(partitions, i)$
    **else if** $CreamContainedPartition(partitions[i], partitions[1])$ **then**
      $partitions[1] \leftarrow partitions[i]$
      $Remove(partitions, i)$
      $i \leftarrow 2$
    **else**
      $i \leftarrow i + 1$
    **end if**
  **until** $NumberOfElements(partitions) < i$
**end if**
$npartitions \leftarrow NumberOfElements(partitions)$
**if** $npartitions > 1$ **then**
  **return** $false$
**else**
  **return** $true$
**end if**
```

Some examples of the use of the package can be seen next:

```
gap> algebra := [RecoverMultiplicationTable(6,1)];
[ [ [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ],
      [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ] ] ]
gap> CreamAllPrincipalCongruences(algebra);
[ [ -1, -1, -1, -1, -2, 5 ], [ -1, -1, -1, -2, 4, -1 ],
  [ -1, -1, -1, -2, -1, 4 ], [ -1, -1, -2, 3, -1, -1 ],
  [ -1, -1, -2, -1, 3, -1 ], [ -1, -1, -2, -1, -1, 3 ],
  [ -1, -2, 2, -1, -1, -1 ], [ -1, -2, -1, 2, -1, -1 ],
  [ -1, -2, -1, -1, 2, -1 ], [ -1, -2, -1, -1, -1, 2 ],
  [ -2, 1, -1, -1, -1, -1 ], [ -2, -1, 1, -1, -1, -1 ],
  [ -2, -1, -1, 1, -1, -1 ], [ -2, -1, -1, -1, 1, -1 ],
  [ -2, -1, -1, -1, -1, 1 ] ]
gap> CreamIsAlgebraMonolithic(algebra);
false
gap> algebra := [RecoverMultiplicationTable(6,19)];
[ [ [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ],
      [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 2, 1, 1 ], [ 1, 1, 2, 1, 1, 1 ] ] ]
```

```
gap> CreamAllPrincipalCongruences(algebra);
[ [ -2, 1, -1, -1, -1, -1 ], [ -2, 1, -1, -1, -2, 5 ],
  [ -2, 1, -1, -2, 4, -1 ], [ -2, 1, -1, -2, -1, 4 ],
  [ -2, 1, -2, 3, -1, -1 ], [ -2, 1, -2, -1, 3, -1 ],
  [ -2, 1, -2, -1, -1, 3 ], [ -3, 1, 1, -1, -1, -1 ],
  [ -3, 1, -1, 1, -1, -1 ], [ -3, 1, -1, -1, 1, -1 ],
  [ -3, 1, -1, -1, -1, 1 ] ]
gap> CreamIsAlgebraMonolithic(algebra);
true
```

4.2 Small Semigroups

For all small semigroups up to size 6, all its congruences and endomorphisms were calculated and are provided as an annex to this thesis. It was calculated also whether the semigroups were monolithic (this was calculated up to size 8).

For these test runs the algebras were considered as universal algebras of type (2^1) even in cases where these algebras can have definitions with different types.

Table 4.1: Determination of all monolithic semigroups up to size 8

Size	**Number of semigroups**	**Number of monoliths**	**Performance (ms)**
1	1	0	0
2	4	4	0
3	18	7	0
4	126	16	8
5	1 160	103	108
6	15 973	1 823	1 712
7	836 021	149 020	127 356
8	1 843 120 128	48 438 046	462 897 348

All automorphisms for semigroups up to size 7 were also calculated separately and the automorphism group and its ID was calculated for each of these semigroups. The group ID is the identification of a group in the Small Group textbfGAP library [7].

4. APPLICATIONS

The **CREAM** library doesn't calculate the automorphisms of an algebra as an automorphism group but as a list of bijective mappings but there is an easy way to calculate the automorphism group from the list of mappings. To do this all the mappings are converted into permutation using the **PermList** function and each of these permutations is used as generator for the group. Having this we can get the Group Id using the **IdGroup** function. An example of this process is shown below:

```
gap> ##################################################
gap> # Small Semigroup size 7 id 836016
gap> ##################################################
gap> algebra := [RecoverMultiplicationTable(7,836016)];
[ [ [ 1, 2, 3, 3, 2, 2, 2 ], [ 2, 3, 1, 1, 3, 3, 3 ], [ 3, 1, 2, 2, 1, 1, 1 ],
      [ 3, 1, 2, 2, 1, 1, 1 ], [ 2, 3, 1, 1, 4, 4, 4 ],
      [ 2, 3, 1, 1, 4, 4, 4 ], [ 2, 3, 1, 1, 4, 4, 4 ] ] ]
gap> auto := CreamAutomorphisms (algebra);
[ [ 1 .. 7 ], [ 1, 2, 3, 4, 5, 7, 6 ], [ 1, 2, 3, 4, 6, 5, 7 ],
  [ 1, 2, 3, 4, 6, 7, 5 ], [ 1, 2, 3, 4, 7, 5, 6 ], [ 1, 2, 3, 4, 7, 6, 5 ] ]
gap> Size (auto);
6
gap> g := Group (List (auto, PermList));
Group([ (), (6,7), (5,6), (5,6,7), (5,7,6), (5,7) ])
gap> id := IdGroup (g);
[ 6, 1 ]
```

The full outputs are also being provided as an annex and a summary table from these runs is provided next.

Table 4.2: Determination of automorphism group for semigroups up to size 7

Size	2	3	4	5	6	7
# Semigroups	4	18	126	1 160	15 973	836 021
[1, 1] trivial	3	12	78	746	10 965	746 277
[2, 1] C_2	1	5	39	342	4 121	76 704
[3, 1] C_3				2	26	412
[4, 1] C_4				1	7	82
[4, 2] $C_2 \times C_2$			3	26	441	7 314
[5, 1] C_5						6
[6, 1] S_3		1	5	33	300	3 638
[6, 2] C_6						37
[8, 2] $C_4 \times C_2$						4
[8, 3] D_8				1	17	169
[8, 5] $C_2 \times C_2 \times C_2$					6	172
[10, 1] D_{10}						2
[12, 4] D_{12}				4	49	790
[16, 11] $C_2 \times D_8$						10
[24, 12] S_4			1	4	30	277
[24, 14] $C_2 \times C_2 \times S_3$						14
[36, 10] $S_3 \times S_3$					2	24
[48, 48] $C_2 \times S_4$					4	45
[72, 40] $(S_3 \times S_3) \wr C_2$						1
[120, 34] S_5				1	4	30
[144, 183] $S_3 \times S_4$						4
[240, 189] $C_2 \times S_5$						4
[720, 763] S_6					1	4
[5040, -] S_7						1

4.3 Small Groups

Using the Small Group **GAP** library [7] the groups from order 2 to 96 were extracted and the **CREAM** functions were used with them:

- Order 2-31 - **CreamAllCongruences**, **CreamAllEndomorphisms** and **CreamIs-Monolithic**
- Order 32-96 - **CreamAllCongruences** and **CreamIsMonolithic**

For these test runs the groups were considered as universal algebras of type (2^1).

An example of this process is shown below:

```
gap> ###################################################
gap> # Small Group size 8 id 1
gap> ###################################################
gap> algebra := [MultiplicationTable(SmallGroup(8,1))];
[ [ [ 1, 2, 3, 4, 5, 6, 7, 8 ], [ 2, 3, 5, 6, 4, 7, 8, 1 ],
      [ 3, 5, 4, 7, 6, 8, 1, 2 ], [ 4, 6, 7, 1, 8, 2, 3, 5 ],
      [ 5, 4, 6, 8, 7, 1, 2, 3 ], [ 6, 7, 8, 2, 1, 3, 5, 4 ],
      [ 7, 8, 1, 3, 2, 5, 4, 6 ], [ 8, 1, 2, 5, 3, 4, 6, 7 ] ] ]
gap> CreamSizeAlgebra(algebra);
8
gap> cong := CreamAllCongruences(algebra);
[ [ -1, -1, -1, -1, -1, -1, -1, -1 ], [ -2, -2, -2, 1, -2, 2, 3, 5 ],
  [ -4, -4, 1, 1, 2, 2, 1, 2 ], [ -8, 1, 1, 1, 1, 1, 1, 1 ] ]
gap> Size (cong);
4
gap> endo := CreamEndomorphisms(algebra);
[ [ 1, 1, 1, 1, 1, 1, 1, 1 ], [ 1, 2, 3, 4, 5, 6, 7, 8 ],
  [ 1, 3, 4, 1, 7, 3, 4, 7 ], [ 1, 4, 1, 1, 4, 4, 1, 4 ],
  [ 1, 5, 7, 4, 2, 8, 3, 6 ], [ 1, 6, 3, 4, 8, 2, 7, 5 ],
  [ 1, 7, 4, 1, 3, 7, 4, 3 ], [ 1, 8, 7, 4, 6, 5, 3, 2 ] ]
gap> Size (endo);
8
gap> CreamIsAlgebraMonolithic(algebra);
true
gap>
```

The full outputs are also being provided as an annex and a the summary table from these runs is provided next.

Table 4.3: Summary for Small Group Runs

Size	**Number of Groups**	**Number of Monolithic**
2-7	8	6
8-15	19	9
16	14	6
17-23	17	7
24	15	2
25-31	19	7
32	50	16
33-47	54	10
48	52	4
49-63	69	16
64	52	22
65-71	17	3
72	50	3
73-79	18	5
80	52	1
81-95	70	13
96	231	13

4.4 Groups, Semigroups and Monoids

For a selection of larger groups, semigroups and monoids, all its congruences and endomorphisms were calculated. It was also calculated whether these algebras were monolithic.

For these test runs the algebras were considered as universal algebras of type (2^1) even in cases where these algebras can have definitions with different types.

An example of this process is shown below:

```
gap> ####################################################
gap> # Brauer Monoid degree 3
gap> ####################################################
gap> algebra := [MultiplicationTable(BrauerMonoid(3))];
[ [ [ 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 1, 2, 3, 2, 3 ],
      [ 1, 2, 3, 1, 2, 3, 1, 2, 3, 2, 3, 1, 1, 3, 2 ],
      [ 1, 2, 3, 1, 2, 3, 1, 2, 3, 3, 2, 3, 2, 1, 1 ],
      [ 4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 4, 5, 6, 5, 6 ],
      [ 4, 5, 6, 4, 5, 6, 4, 5, 6, 5, 6, 4, 4, 6, 5 ],
      [ 4, 5, 6, 4, 5, 6, 4, 5, 6, 6, 5, 6, 5, 4, 4 ],
      [ 7, 8, 9, 7, 8, 9, 7, 8, 9, 7, 7, 8, 9, 8, 9 ],
      [ 7, 8, 9, 7, 8, 9, 7, 8, 9, 8, 9, 7, 7, 9, 8 ],
      [ 7, 8, 9, 7, 8, 9, 7, 8, 9, 9, 8, 9, 8, 7, 7 ],
      [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 ],
      [ 4, 5, 6, 1, 2, 3, 7, 8, 9, 11, 10, 14, 15, 12, 13 ],
      [ 1, 2, 3, 7, 8, 9, 4, 5, 6, 12, 13, 10, 11, 15, 14 ],
      [ 7, 8, 9, 1, 2, 3, 4, 5, 6, 13, 12, 15, 14, 10, 11 ],
      [ 4, 5, 6, 7, 8, 9, 1, 2, 3, 14, 15, 11, 10, 13, 12 ],
      [ 7, 8, 9, 4, 5, 6, 1, 2, 3, 15, 14, 13, 12, 11, 10 ] ] ]
gap> CreamSizeAlgebra(algebra);
15
gap> cong := CreamAllCongruences(algebra);
[ [ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ],
  [ -3, 1, 1, -3, 4, 4, -3, 7, 7, -1, -1, -1, -1, -1, -1 ],
  [ -3, -3, -3, 1, 2, 3, 1, 2, 3, -1, -1, -1, -1, -1, -1 ],
  [ -9, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1 ],
  [ -9, 1, 1, 1, 1, 1, 1, 1, 1, -3, -3, 11, 10, 10, 11 ],
  [ -9, 1, 1, 1, 1, 1, 1, 1, 1, -6, 10, 10, 10, 10, 10 ],
  [ -15, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ]
gap> Size (cong);
7
```

```
gap> endo := CreamEndomorphisms(algebra);
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 10, 10, 10, 10, 10, 10 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 ],
  [ 2, 1, 3, 8, 7, 9, 5, 4, 6, 10, 15, 12, 14, 13, 11 ],
  [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ],
  [ 2, 2, 2, 2, 2, 2, 2, 2, 2, 10, 10, 10, 10, 10, 10 ],
  [ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ],
  [ 3, 3, 3, 3, 3, 3, 3, 3, 3, 10, 10, 10, 10, 10, 10 ],
  [ 3, 3, 3, 3, 3, 3, 3, 3, 3, 10, 12, 12, 10, 10, 12 ],
  [ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ],
  [ 4, 4, 4, 4, 4, 4, 4, 4, 4, 10, 10, 10, 10, 10, 10 ],
  [ 4, 6, 5, 1, 3, 2, 7, 9, 8, 10, 11, 15, 14, 13, 12 ],
  [ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ],
  [ 5, 5, 5, 5, 5, 5, 5, 5, 5, 10, 10, 10, 10, 10, 10 ],
  [ 5, 5, 5, 5, 5, 5, 5, 5, 5, 10, 15, 15, 10, 10, 15 ],
  [ 6, 4, 5, 9, 7, 8, 3, 1, 2, 10, 12, 15, 13, 14, 11 ],
  [ 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6 ],
  [ 6, 6, 6, 6, 6, 6, 6, 6, 6, 10, 10, 10, 10, 10, 10 ],
  [ 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7 ],
  [ 7, 7, 7, 7, 7, 7, 7, 7, 7, 10, 10, 10, 10, 10, 10 ],
  [ 7, 7, 7, 7, 7, 7, 7, 7, 7, 10, 11, 11, 10, 10, 11 ],
  [ 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8 ],
  [ 8, 8, 8, 8, 8, 8, 8, 8, 8, 10, 10, 10, 10, 10, 10 ],
  [ 8, 9, 7, 2, 3, 1, 5, 6, 4, 10, 15, 11, 13, 14, 12 ],
  [ 9, 8, 7, 6, 5, 4, 3, 2, 1, 10, 12, 11, 14, 13, 15 ],
  [ 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9 ],
  [ 9, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 10 ],
  [ 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10 ] ]
gap> Size (endo);
28
gap> CreamIsAlgebraMonolithic(algebra);
false
gap>
```

The full outputs are also being provided as an annex and a summary table from these runs is provided next.

Table 4.4: # of Congruences and Endomorphisms for a selection of larger algebras

Algebra	**Size**	**Number of Congruences**	**Number of Endomorphisms**	**Is Monolithic**
GossipMonoid(3)	11	84	66	No
PlanarPartitionMonoid(2)	14	9	72	No
JonesMonoid(4)	14	9	72	No
BrauerMonoid(3)	15	7	28	No
PartitionMonoid(2)	15	13	89	No
FullPBRMonoid(1)	16	167	1426	No
SymmetricGroup(4)	24	4	58	Yes
FullTransformationSemigroup(3)	27	7	40	Yes
FullTransformationMonoid(3)	27	7	40	Yes
SymmetricInverseMonoid(3)	34	7	54	Yes
JonesMonoid(5)	42	6	113	No
MotzkinMonoid(3)	51	10	98	No
PartialTransformationMonoid(3)	64	7	138	Yes
PartialBrauerMonoid(3)	76	16	165	No
BrauerMonoid(4)	105	19	274	No
SymmetricGroup(5)	120	3	146	Yes
PlanarPartitionMonoid(3)	132	10	393	No
JonesMonoid(6)	132	10	393	No
PartitionMonoid(3)	203	16	687	No
SymmetricInverseMonoid(4)	209	11	282	Yes
FullTransformationSemigroup(4)	256	11	345	Yes
FullTransformationMonoid(4)	256	11	345	Yes

4.5 Algebra Classes

The **CREAM** package has a function that calls **MACE4** to generate lists of algebras of specific type and size based on their axiomatic definition. The generated algebras can then be used with the **CREAM** functions that calculate congruences, automorphisms and endomorphisms.

CREAM was used with the following algebra classes:

- Almost distributive lattices
- BCI-algebras
- Bilattice
- Boolean algebra
- Boolean ring
- Commutative lattice-ordered monoids
- Commutative regular rings
- Complemented distributive lattices
- Complemented modular lattices
- Distributive lattice ordered semigroups
- Ockham algebras
- Ortholattices
- Orthomodular lattices
- Idempotent semirings
- Extra loops
- Digroup
- Commutative dimonoids
- Loops
- Fields
- MV-Algebras

4. APPLICATIONS

An example of this process is shown below:

```
gap> ####################################################
gap> # Almost distributive lattice
gap> ####################################################
gap> axioms;
"(x v y) v z = x v (y v z). x v y = y v x. (x ^ y) ^ z = x ^ (y ^ z). x ^ y = \
y ^ x. (x v y) ^ x = x. (x ^ y) v x = x. x ^ (y v z) = x ^ (y v (x ^ (z v (x ^\
 y)))). x v (y ^ z) = x v (y ^ (x v (z ^ (x v y)))). w ^ (u v (x ^ (y v (x ^ z\
)))) < u v ((x ^ (y v (x ^ z))) ^ ((w v (x ^ y)) v (x ^ z))). w v (u ^ (x v (y\
 ^ (x v z)))) > u ^ ((x v (y ^ (x v z))) v ((w ^ (x v y)) ^ (x v z)))."
gap> algebras := CreamAlgebrasFromAxioms(axioms,4);
[ [ [ [ 1, 1, 1, 1 ], [ 1, 2, 1, 2 ], [ 1, 1, 3, 3 ], [ 1, 2, 3, 4 ] ],
      [ [ 1, 2, 3, 4 ], [ 2, 2, 4, 4 ], [ 3, 4, 3, 4 ], [ 4, 4, 4, 4 ] ] ],
  [ [ [ 1, 1, 1, 1 ], [ 1, 2, 2, 2 ], [ 1, 2, 3, 3 ], [ 1, 2, 3, 4 ] ],
      [ [ 1, 2, 3, 4 ], [ 2, 2, 3, 4 ], [ 3, 3, 3, 4 ], [ 4, 4, 4, 4 ] ] ] ]
gap> Size (algebras);
2
gap> CreamSizeAlgebra(algebras[1]);
4
gap> cong := CreamAllCongruences(algebras[1]);
[ [ -1, -1, -1, -1 ], [ -2, 1, -2, 3 ], [ -2, -2, 1, 2 ], [ -4, 1, 1, 1 ] ]
gap> Size (cong);
4
gap> endo := CreamEndomorphisms(algebras[1]);
[ [ 1, 1, 1, 1 ], [ 1, 1, 2, 2 ], [ 1, 1, 3, 3 ], [ 1, 1, 4, 4 ],
  [ 1, 2, 1, 2 ], [ 1, 2, 3, 4 ], [ 1, 3, 1, 3 ], [ 1, 3, 2, 4 ],
  [ 1, 4, 1, 4 ], [ 2, 2, 2, 2 ], [ 2, 2, 4, 4 ], [ 2, 4, 2, 4 ],
  [ 3, 3, 3, 3 ], [ 3, 3, 4, 4 ], [ 3, 4, 3, 4 ], [ 4, 4, 4, 4 ] ]
gap> Size (endo);
16
gap> CreamIsAlgebraMonolithic(algebras[1]);
false
gap>
```

The type of the algebra will depend on the number and type of operations present in the algebra axioms that will generate models with that specific number of binary and unary operations.

The full outputs are also being provided as an annex and a summary table from these runs is provided next.

Table 4.5: Summary for Algebra Class Runs

Algebra Class	**Size**	**Algebra Type**	**Number of Algebras**	**Number of Monolithic**
Almost distributive lattices	4	(2^2)	2	0
BCI-algebras	5	(2^1)	118	80
Bilattice	6	$(2^4, 1^1)$	32	32
Boolean algebra	8	$(2^2, 1^1)$	1	0
Boolean algebra	16	$(2^2, 1^1)$	1	0
Boolean ring	16	(2^2)	1	0
Commutative lattice-ordered monoids	5	(2^3)	199	97
Commutative regular rings	6	$(2^2, 1^2)$	72	66
Complemented distributive lattices	16	$(2^2, 1^1)$	1	0
Complemented distributive lattices	32	$(2^2, 1^1)$	1	0
Complemented modular lattices	8	$(2^2, 1^1)$	41	40
Distributive lattice ordered semigroups	4	(2^3)	479	170
Ockham algebras	6	$(2^2, 1^1)$	197	20
Ortholattices	7	$(2^2, 1^1)$	46	12
Orthomodular lattices	14	$(2^2, 1^1)$	33	31
Idempotent semiring	5	(2^2)	149	42
Extra loop	10	(2^3)	2	1
Digroup	8	$(2^2, 1^1)$	10	3
Commutative dimonoid	4	(2^2)	101	37
Loops	6	(2^3)	109	108
Fields	16	$(2^1, 1^2)$	6	6
MV-Algebras	12	$(2^1, 1^1)$	4	1

4.6 Future Improvements

Despite all the achievements that were possible in the scope of this work there is room for improvement that will be considered for future work.

Although the achieved performance is already very good it seems to be possible to further improve the performance of **CREAM**. The congruence and automorphism algorithms are fully run on C but other algorithms are still mostly in GAP. An obvious and sure way to achieve further performance improvements would be to rewrite the most time consuming parts of those algorithms in C.

In what concerns the algorithms, two other areas with potential for improvement performance-wise is to do optimizations in the generation of all congruences from principal congruences and the use of pre-calculated graphs in the algorithm when calculating principal congruences.

One additional area of performance improvement related with the architecture of the package would be to evaluate the possibility of using parallelization to achieve further improvements.

One improvement that was identified during later stages of this work would be to add support to nullary operations (constants) in algebras. This will not make a difference in congruence calculation but will present different results in the calculation of automorphisms and endomorphisms. Without considering nullary operations some automorphisms and endomorphisms will be returned that aren't automorphisms and endomorphisms of the algebra with the nullary operations.

Other avenue for improvement would be to have tighter integration with **MACE4** by loading it as a library instead of being run as separate process.

There are a few functions providing intermediate or complementary results and data that are not currently exposed by **CREAM** and that would be good additions to the **CREAM** interface. Examples of such functions would be:

- **CreamCongruenceEndomorphisms** - Returning the endomorphisms derived from a specific congruence of the algebra;
- **CreamCongruenceSubUniverses** - Returning the subuniverses of the subalgebras derived from a specific congruence of the algebra;
- **CreamEndomorphismSubUniverses** - Returning the subuniverses of the subalgebras derived from a specific endomorphism of the algebra;

- **CreamEquivalenceRelationPartition** - Returns the **GAP** equivalence relation defined by the partition;
- **CreamPartitionEquivalenceRelation** - Returns the partition defined by the **GAP** equivalence relation;
- **CreamAutomorphismGroup** - Returns the algebra's automorphisms as a Group;
- **CreamEndomorphismSemigroup** - Returns the algebra's endomorphisms as a Semigroup;
- **CreamJoinPartition** - Returns the join of 2 partitions,
- **CreamMeetPartition** - Returns the meet of 2 partitions;
- **CreamViewHasseDiagram** - Allows the visualisation the Hasse diagram of the congruence lattice.

Chapter 5

Conclusion

The **CREAM** package makes available efficient algorithms for the calculation of congruences, automorphisms, endomorphisms, monomorphisms, isomorphisms, subalgebras, divisors, monoliths, direct decompositions, etc., being the first tool doing that. These algorithms work with algebras of type $(2^m, 1^n)$ covering a very wide set of algebras while being consistently fast.

Supporting algebras of type $(2^m, 1^n)$ covers most extensively studied algebras in conventional and modern algebra since these algebras normally don't have fundamental operations of arity greater than two.

There are situations where other algorithms and implementations were faster but these cases were only more efficient for very specific types of algebras. Only **CREAM** was able to have a consistent performance across different types of algebras.

In particular, the Freese congruence algorithm relies on the performance of the manipulation of the partition data structure that allows a very fast calculation of each of the principal congruences. This is done without limiting the scope of the algebra supported.

While **GAP** is adequate for fast prototyping and implementation of algorithms the resulting code is not very fast because it is an interpreted language. Rewriting the code in C allowed for staggering improvements that reached up to a 670-fold improvement from the **GAP** code to the C code.

Also the integration with **MACE4** allowed to combine the efficiency of algorithms such as [15], with wide set of possibilities provided by a first order axiom-based model searcher giving a very wide flexibility to **CREAM**.

Regarding automorphisms and endomorphisms the comparison to other libraries/application is very difficult since the support is mostly limited to groups and other closely related algebraic structures. For these algebras the performance of the **GAP** packages **Loops** and **Sonata** are comparable and sometimes better than **CREAM**. These libraries rely sometimes on the use of theorems specific to these algebras. But only **CREAM** supports most of the algebras studied in conventional and modern algebra.

The **CREAM** package is on average 20 times faster in calculating congruences of the very limited set semigroups that are supported by the function **CongruencesOfSemigroups** from the package **Semigroups** which to our knowledge is most general function in **GAP** for the calculation of congruences. The only application that has a similar range in terms of supported algebras is **UACalc** command line interface but does that in jython not benefiting of the existing **GAP** ecosystem. When compared with **UACalc** the **CREAM** package is consistently more than 3 times faster. The **CREAM** package has a better overall performance to existing tools (for small models).

Given the importance of congruences, automorphisms and endomorphisms for the study of algebraic structures there is the expectation the package **CREAM** with its performance and versatility can be a useful tool for the **GAP** community and a wide group mathematicians.

CREAM already allowed obtaining hundreds of new results in mathematics and there is no comparable tool in **GAP** or other platforms.

Appendices

Appendix A

The CREAM Package Manual

CREAM stands for "Algebra CongRuences, Endomorphisms and AutoMorphisms". **CREAM** is a GAP Package developed in GAP and C. GAP stands for "Groups, Algorithms, Programming" and is a system for computational discrete algebra, with particular emphasis on Computational Group Theory.

A.1 Installation

A.1.1 Package Dependencies

The CREAM package is written in GAP and C. It runs in GAP version 4.10 or above, with C run-time support.

If you do not see the subfolder pkg/cream in the main directory of GAP then download the CREAM package from the distribution website `https://gap-packages.github.io/cream/` and unpack the downloaded file into the pkg subfolder.

A.1.2 Compiling the kernel module

C is used extensively in the CREAM package for best performance. Therefore the GAP kernel module written in C must be compiled. It is not possible to use the CREAM package without the kernel module.

To compile the kernel component inside the `pkg/cream-2.0.0` directory, type:

```
./configure
make
```

A.1.3 Compiling MACE4

Calls to MACE4 are made by CREAM. MACE4 sources are provided with the CREAM package on the `pkg/cream-2.0.0/src/prover9/` directory. MACE4 needs to be compiled on installation of the package. If the compiled binary cannot be found in the `pkg/cream/src/prover9/build` directory, the system path will be searched for MACE4. This way if the supplied MACE4 cannot be compiled, a separate installation of MACE4 can be used instead as long it is accessible from the system path.

To compile **MACE4** inside the `pkg/cream/src/prover9/build/` directory, type:

```
cmake -DCMAKE_BUILD_TYPE=Release ..
make mace4
```

A.1.4 Building the documentation

The documentation of CREAM can be built with the following command, executed inside the `pkg/cream` directory:

```
make doc
```

A.1.5 Testing your installation

Test files conforming to GAP standards are provided for CREAM and are located in the folder named `tst`. The following command runs all tests for the CREAM package:

```
ReadPackage("Cream","tst/testall.g");
```

A.2 Package Functions

A.2.1 Algebras

CreamSizeAlgebra

▷ `CreamSizeAlgebra(A)` (function)

This function returns the order of the algebra `A`, provided the algebra is valid. An algebra will be valid if all vectors and matrices defining its operations have the same size. CreamSizeAlgebra will return fail if the algebra is not valid.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> CreamSizeAlgebra(algebra);
6
gap> s2 := MultiplicationTable(SymmetricGroup (2));
[ [ 1, 2 ], [ 2, 1 ] ]
gap> c3 := MultiplicationTable(CyclicGroup (3));
[ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ]
gap> notAlgebra := [ s2, c3 ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamSizeAlgebra(notAlgebra);
fail
gap>
```

CreamIsAlgebra

▷ `CreamIsAlgebra(A)` (function)

This function returns true the if the algebra `A` is valid and false otherwise. An algebra will be valid if all the vectors and matrices defining its operations have the same size.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> CreamIsAlgebra(algebra);
true
gap> s2 := MultiplicationTable(SymmetricGroup (2));
[ [ 1, 2 ], [ 2, 1 ] ]
gap> c3 := MultiplicationTable(CyclicGroup (3));
[ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ]
gap> notAlgebra := [ s2, c3 ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamIsAlgebra(notAlgebra);
false
gap>
```

A.2.2 Congruences

CreamNumberOfBlocks

▷ `CreamNumberOfBlocks(P)` (function)

This function returns the number of blocks in a partition `P`. It returns fail if `P` does not correspond to a partition. Similarly, all the following functions will return fail for invalid inputs.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> partition:= CreamPrincipalCongruence(algebra,[1,4]);
[ -3, -3, 2, 1, 1, 2 ]
gap> CreamNumberOfBlocks(partition);
2
gap> CreamNumberOfBlocks([ -3, -3, 2, 1, 1, 7 ]);
fail
gap>
```

CreamContainedPartition

▷ `CreamContainedPartition(P1,P2)` (function)

Returns true if partition `P1` is contained in `P2`, i.e. if every block of `P1` is contained in a block in `P2`.

```
gap> CreamContainedPartition ([ -3, -3, 2, 1, 1, 2 ], [ -6, 1, 1, 1, 1, 1 ]);
true
gap> CreamContainedPartition ([ -6, 1, 1, 1, 1, 1 ], [ -3, -3, 2, 1, 1, 2 ]);
false
gap> CreamContainedPartition ([ -3, -3, 2, 1, 1, 7 ], [ -6, 1, 1, 1, 1, 1 ]);
fail
gap>
```

CreamPrincipalCongruence

▷ `CreamPrincipalCongruence(A,p)` (function)

This function returns the principal congruence of an algebra `A` generated by a pair `p` of elements based on the Ralph Freese's algorithm described in ”Computing congruences efficiently” [15]. The congruence is returned in the partition format described in 3.1.2.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> CreamPrincipalCongruence(algebra,[1,4]);
[ -3, -3, 2, 1, 1, 2 ]
gap> CreamPrincipalCongruence(algebra, [1,7]);
fail
gap> s2 := MultiplicationTable(SymmetricGroup (2));
[ [ 1, 2 ], [ 2, 1 ] ]
gap> c3 := MultiplicationTable(CyclicGroup (3));
[ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ]
gap> notAlgebra := [ s2, c3 ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamPrincipalCongruence(notAlgebra,[1,4]);
fail
gap>
```

CreamAllPrincipalCongruences

▷ `CreamAllPrincipalCongruences(A)` (function)

This function returns all principal congruences of an algebra `A`. This is done by going through all possible pairs in an algebra and calculating the principal congruence for each one of these pairs.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> CreamAllPrincipalCongruences(algebra);
[ [ -3, -3, 2, 1, 1, 2 ], [ -6, 1, 1, 1, 1, 1 ] ]
gap> s2 := MultiplicationTable(SymmetricGroup (2));
[ [ 1, 2 ], [ 2, 1 ] ]
gap> c3 := MultiplicationTable(CyclicGroup (3));
[ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ]
gap> notAlgebra := [ s2, c3 ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamAllPrincipalCongruences(notAlgebra);
fail
gap>
```

CreamIsAlgebraMonolithic

▷ `CreamIsAlgebraMonolithic(A)` (function)

This function returns true if the algebra `A` is monolithic, by determining if there is a principal congruence that is contained in every other principal congruence.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> CreamIsAlgebraMonolithic(algebra);
true
gap> algebra := [ [ [ 1, 1, 3 ], [ 1, 1, 3 ], [ 3, 3, 1 ] ] ];
[ [ [ 1, 1, 3 ], [ 1, 1, 3 ], [ 3, 3, 1 ] ] ]
gap> CreamIsAlgebraMonolithic (algebra);
false
gap> s2 := MultiplicationTable(SymmetricGroup (2));
[ [ 1, 2 ], [ 2, 1 ] ]
gap> c3 := MultiplicationTable(CyclicGroup (3));
[ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ]
gap> notAlgebra := [ s2, c3 ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamIsAlgebraMonolithic (notAlgebra);
fail
gap>
```

CreamAllCongruences

▷ `CreamAllCongruences(A)` (function)

This function returns all congruences of an algebra `A`. All principal congruences are calculated and the trivial congruence is added to the list. The principal congruences are then combined to generate all possible congruences.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> CreamAllCongruences(algebra);
[ [ -1, -1, -1, -1, -1, -1 ], [ -3, -3, 2, 1, 1, 2 ], [ -6, 1, 1, 1, 1, 1 ] ]
gap> s2 := MultiplicationTable(SymmetricGroup (2));
[ [ 1, 2 ], [ 2, 1 ] ]
gap> c3 := MultiplicationTable(CyclicGroup (3));
[ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ]
gap> notAlgebra := [ s2, c3 ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamAllCongruences(notAlgebra);
fail
gap>
```

A.2.3 Isomorphisms, Automorphisms and Endomorphisms

CreamAreIsomorphicAlgebras

▷ `CreamAreIsomorphicAlgebras(A,B)` (function)

This function returns true if the algebras `A` and `B` are isomorphic.

```
gap> algebra1:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> CreamAreIsomorphicAlgebras(algebra1,algebra1);
true
gap> algebra2:= [MultiplicationTable(CyclicGroup (6))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 3, 4, 5, 6, 1 ], [ 3, 4, 5, 6, 1, 2 ],
      [ 4, 5, 6, 1, 2, 3 ], [ 5, 6, 1, 2, 3, 4 ], [ 6, 1, 2, 3, 4, 5 ] ] ]
gap> CreamAreIsomorphicAlgebras(algebra1,algebra2);
false
gap> s2 := MultiplicationTable(SymmetricGroup (2));
[ [ 1, 2 ], [ 2, 1 ] ]
gap> c3 := MultiplicationTable(CyclicGroup (3));
[ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ]
gap> notAlgebra := [ s2, c3 ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamAreIsomorphicAlgebras(notAlgebra,notAlgebra);
fail
gap>
```

CreamAllSubUniverses

▷ `CreamAllSubUniverses(A)` (function)

This function returns the subuniverses of the subalgebras of `A`.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> CreamAllSubUniverses(algebra);
[ [ 1 ], [ 1, 2 ], [ 1, 2, 3, 4, 5, 6 ], [ 1, 3 ], [ 1, 4, 5 ], [ 1, 6 ] ]
gap> c3 := [MultiplicationTable(CyclicGroup (3))];
[ [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> notAlgebra := [ algebra[1], c3[1] ];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ],
  [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamAllSubUniverses(notAlgebra);
fail
gap>
```

CreamSubUniverse2Algebra

▷ `CreamSubUniverse2Algebra(A,S)` (function)

Returns the algebra corresponding a subalgebra represented by its subuniverse

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> salgebras := CreamSubUniverses(algebra);
[ [ 1 ], [ 1, 2 ], [ 1, 2, 3, 4, 5, 6 ], [ 1, 3 ], [ 1, 4, 5 ], [ 1, 6 ] ]
gap> CreamSubUniverse2Algebra(algebra, salgebras[2]);
[ [ [ 1, 2 ], [ 2, 1 ] ] ]
gap> CreamSubUniverse2Algebra(algebra, [ 1, 7 ]);
fail
gap>
```

CreamAlgebrasFromAxioms

▷ `CreamAlgebrasFromAxioms(Ax,d)` (function)

This function returns all isomorphic algebras of dimension `d` that respect the axioms `Ax` using MACE4.

```
gap> axioms;
"x * x = x. (x * y) * x = x * y.y * (x * y) = x * y. x * ((x * y) * z) =
(x *  y) * z."
gap> algebras := CreamAlgebrasFromAxioms (axioms,3);
[ [ [ [ 1, 1, 1 ], [ 1, 2, 1 ], [ 1, 1, 3 ] ] ],
  [ [ [ 1, 1, 1 ], [ 1, 2, 2 ], [ 1, 2, 3 ] ] ] ]
gap>
```

CreamEndomorphisms

▷ `CreamEndomorphisms(A)` (function)

This function returns all endomorphisms of an algebra `A`. All congruences of an algebra `A` are calculated and for each congruence ϕ MACE4 is used to find all subalgebras of `A` that are isomorphic to the quotient `A`$/\phi$. The automorphisms are then calculated using the invariants algoritm implemented in C.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> CreamEndomorphisms(algebra);
[ [ 1, 1, 1, 1, 1, 1 ], [ 1, 2, 2, 1, 1, 2 ], [ 1 .. 6 ],
  [ 1, 2, 6, 5, 4, 3 ], [ 1, 3, 2, 5, 4, 6 ], [ 1, 3, 3, 1, 1, 3 ],
  [ 1, 3, 6, 4, 5, 2 ], [ 1, 6, 2, 4, 5, 3 ], [ 1, 6, 3, 5, 4, 2 ],
  [ 1, 6, 6, 1, 1, 6 ] ]
gap> s2 := MultiplicationTable(SymmetricGroup (2));
[ [ 1, 2 ], [ 2, 1 ] ]
gap> c3 := MultiplicationTable(CyclicGroup (3));
[ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ]
gap> notAlgebra := [ s2, c3 ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamEndomorphisms(notAlgebra);
fail
gap>
```

CreamAutomorphisms

▷ `CreamAutomorphisms(A)` (function)

This function returns all automorphisms of an algebra `A`. The algorithm used in this function calculates characteristics of the algebra's elements that are invariant under automorphisms. This narrows the set of potential images to which each element can map, giving an efficient way to determine the automorphisms.

```
gap> algebra:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> CreamAutomorphisms(algebra);
[ [ 1 .. 6 ], [ 1, 2, 6, 5, 4, 3 ], [ 1, 3, 2, 5, 4, 6 ], [ 1, 3, 6, 4, 5, 2 ],
  [ 1, 6, 2, 4, 5, 3 ], [ 1, 6, 3, 5, 4, 2 ] ]
gap> s2 := MultiplicationTable(SymmetricGroup (2));
[ [ 1, 2 ], [ 2, 1 ] ]
gap> c3 := MultiplicationTable(CyclicGroup (3));
[ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ]
gap> notAlgebra := [ s2, c3 ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamAutomorphisms(notAlgebra);
fail
gap>
```

CreamExistsEpimorphism

▷ `CreamExistsEpimorphism(A,B)` (function)

This function returns whether there is an epimorhisms between two algebras `A` and `B`.

```
gap> algebra1:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> algebra2:= [MultiplicationTable(SymmetricGroup (2))];
[ [ [ 1, 2 ], [ 2, 1 ] ] ]
gap> CreamExistsEpimorphism(algebra1,algebra2);
true
gap> c3 := [MultiplicationTable(CyclicGroup (3))];
[ [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamExistsEpimorphism(algebra1,c3);
false
gap> notAlgebra := [ algebra2[1], c3[1] ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamExistsEpimorphism(notAlgebra, notAlgebra);
fail
gap>
```

CreamOneEpimorphism

▷ `CreamOneEpimorphism(A,B)` (function)

This function returns one epimorhism between two algebras `A` and `B`.

```
gap> algebra1:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> algebra2:= [MultiplicationTable(SymmetricGroup (2))];
[ [ [ 1, 2 ], [ 2, 1 ] ] ]
gap> CreamOneEpimorphism(algebra1,algebra2);
[ 1, 2, 2, 1, 1, 2 ]
gap> c3 := [MultiplicationTable(CyclicGroup (3))];
[ [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamOneEpimorphism(algebra1,c3);
fail
gap> notAlgebra := [ algebra2[1], c3[1] ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamOneEpimorphism(notAlgebra, notAlgebra);
fail
gap>
```

CreamAllEpimorphisms

▷ CreamAllEpimorphisms(A,B) (function)

This function returns the epimorhisms between two algebras A and B.

```
gap> algebra1:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> algebra2:= [MultiplicationTable(SymmetricGroup (2))];
[ [ [ 1, 2 ], [ 2, 1 ] ] ]
gap> CreamAllEpimorphisms(algebra1,algebra2);
[ [ 1, 2, 2, 1, 1, 2 ] ]
gap> c3 := [MultiplicationTable(CyclicGroup (3))];
[ [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamAllEpimorphisms(algebra1,c3);
[  ]
gap> notAlgebra := [ algebra2[1], c3[1] ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamAllEpimorphisms(notAlgebra, notAlgebra);
fail
gap>
```

CreamExistsMonomorphism

▷ CreamExistsMonomorphism(A,B) (function)

This function returns whether there is a monomorphisms between two algebras A and B.

```
gap> algebra1:= [MultiplicationTable(SymmetricGroup (2))];
[ [ [ 1, 2 ], [ 2, 1 ] ] ]
gap> algebra2:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> CreamExistsMonomorphism(algebra1,algebra2);
true
gap> c3 := [MultiplicationTable(CyclicGroup (3))];
[ [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamExistsMonomorphism(algebra1,c3);
false
gap> notAlgebra := [ algebra1[1], c3[1] ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamExistsMonomorphism(notAlgebra, notAlgebra);
fail
gap>
```

CreamOneMonomorphism

▷ `CreamOneMonomorphism(A,B)` (function)

This function returns a monomorphisms between two algebras `A` and `B`.

```
gap> algebra1:= [MultiplicationTable(SymmetricGroup (2))];
[ [ [ 1, 2 ], [ 2, 1 ] ] ]
gap> algebra2:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> CreamOneMonomorphism(algebra1,algebra2);
[ 1, 2 ]
gap> c3 := [MultiplicationTable(CyclicGroup (3))];
[ [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamOneMonomorphism(algebra1,c3);
fail
gap> notAlgebra := [ algebra1[1], c3[1] ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamOneMonomorphism(notAlgebra, notAlgebra);
fail
gap>
```

CreamAllMonomorphisms

▷ `CreamAllMonomorphisms(A,B)` (function)

This function returns the monomorphisms between two algebras `A` and `B`.

```
gap> algebra1:= [MultiplicationTable(SymmetricGroup (2))];
[ [ [ 1, 2 ], [ 2, 1 ] ] ]
gap> algebra2:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> CreamAllMonomorphisms(algebra1,algebra2);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 6 ] ]
gap> c3 := [MultiplicationTable(CyclicGroup (3))];
[ [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamAllMonomorphisms(algebra1,c3);
[  ]
gap> notAlgebra := [ algebra1[1], c3[1] ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamAllMonomorphisms(notAlgebra, notAlgebra);
fail
gap>
```

CreamExistsDivisor

▷ `CreamExistsDivisor(A,B)` (function)

This function returns whether the algebra `B` is a divisor of the algebra `A`.

```
gap> algebra1:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> algebra2:= [MultiplicationTable(SymmetricGroup (2))];
[ [ [ 1, 2 ], [ 2, 1 ] ] ]
gap> CreamExistsDivisor(algebra1,algebra2);
true
gap> c3 := [MultiplicationTable(CyclicGroup (3))];
[ [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamExistsDivisor(c3,algebra2);
false
gap> notAlgebra := [ algebra1[1], c3[1] ];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ],
  [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamExistsDivisor(notAlgebra, notAlgebra);
fail
gap>
```

CreamOneDivisorUniverse

▷ `CreamOneDivisorUniverse(A,B)` (function)

This function returns one divisor universe and respective congruence between two algebras `A` and `B`.

```
gap> algebra1:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> algebra2:= [MultiplicationTable(SymmetricGroup (2))];
[ [ [ 1, 2 ], [ 2, 1 ] ] ]
gap> CreamOneDivisorUniverse(algebra1,algebra2);
[ [ 1, 2 ], [ -1, -1 ] ]
gap> c3 := [MultiplicationTable(CyclicGroup (3))];
[ [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamOneDivisorUniverse(c3,algebra2);
fail
gap> notAlgebra := [ algebra1[1], c3[1] ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamOneDivisorUniverse(notAlgebra, notAlgebra);
fail
gap>
```

CreamAllDivisorUniverses

▷ `CreamAllDivisorUniverses(A,B)` (function)

This function returns the divisors universes and repective congruences between two algebras `A` and `B`.

```
gap> algebra1:= [MultiplicationTable(SymmetricGroup (3))];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ] ]
gap> algebra2:= [MultiplicationTable(SymmetricGroup (2))];
[ [ [ 1, 2 ], [ 2, 1 ] ] ]
gap> CreamAllDivisorUniverses(algebra1,algebra2);
[ [ [ 1, 2 ], [ -1, -1 ] ], [ [ 1, 2, 3, 4, 5, 6 ], [ -3, -3, 2, 1, 1, 2 ] ],
  [ [ 1, 3 ], [ -1, -1 ] ], [ [ 1, 6 ], [ -1, -1 ] ] ]
gap> c3 := [MultiplicationTable(CyclicGroup (3))];
[ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ]
gap> CreamAllDivisorUniverses(c3,algebra2);
[  ]
gap> notAlgebra := [ algebra1[1], c3[1] ];
[ [ [ 1, 2 ], [ 2, 1 ] ], [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamAllDivisorUniverses(notAlgebra, notAlgebra);
fail
gap>
```

CreamExistsDirectlyReducible

▷ `CreamExistsDirectlyReducible(A)` (function)

For algebra `A` there are commuting congruences R, T where $A/R \times A/T$ is isomorphic to `A`.

```
gap> s2:= [MultiplicationTable(SymmetricGroup (2))];
[ [ [ 1, 2 ], [ 2, 1 ] ] ]
gap> algebra:=[MultiplicationTable(DirectProduct(s2,s2))];
[ [ [ 1, 2, 3, 4 ], [ 2, 1, 4, 3 ], [ 3, 4, 1, 2 ], [ 4, 3, 2, 1 ] ] ]
gap> CreamExistsDirectlyReducible(algebra);
true
gap> c3 := [MultiplicationTable(CyclicGroup (3))];
[ [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamExistsDirectlyReducible(c3);
false
gap> notAlgebra := [ algebra[1], c3[1] ];
[ [ [ 1, 2, 3, 4 ], [ 2, 1, 4, 3 ], [ 3, 4, 1, 2 ], [ 4, 3, 2, 1 ] ],
  [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamExistsDirectlyReducible(notAlgebra);
fail
gap>
```

CreamOneDirectlyReducible

▷ `CreamOneDirectlyReducible(A)` (function)

For algebra `A` there is a commuting congruences R, T where $A/R \times A/T$ is isomorphic to `A`.

```
gap> s2:= [MultiplicationTable(SymmetricGroup (2))];
[ [ [ 1, 2 ], [ 2, 1 ] ] ]
gap> algebra:=[MultiplicationTable(DirectProduct(s2,s2))];
[ [ [ 1, 2, 3, 4 ], [ 2, 1, 4, 3 ], [ 3, 4, 1, 2 ], [ 4, 3, 2, 1 ] ] ]
gap> CreamOneDirectlyReducible(algebra);
[ [ -2, 1, -2, 3 ], [ -2, -2, 2, 1 ] ]
gap> c3 := [MultiplicationTable(CyclicGroup (3))];
[ [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamOneDirectlyReducible(c3);
fail
gap> notAlgebra := [ algebra[1], c3[1] ];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ],
  [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamOneDirectlyReducible(notAlgebra);
fail
gap>
```

CreamAllDirectlyReducible

▷ `CreamAllDirectlyReducible(A)` (function)

For algebra `A` find commuting congruences R, T where $A/R \times A/T$ is isomorphic to `A`. For the sake of simplicity, symmetric pairs are only listed once and if `[ R, T ]` then `[ T , R ]` won't be in the output despite being valid for direct product construction.

```
gap> s2:= [MultiplicationTable(SymmetricGroup (2))];
[ [ [ 1, 2 ], [ 2, 1 ] ] ]
gap> algebra:=[MultiplicationTable(DirectProduct(s2,s2))];
[ [ [ 1, 2, 3, 4 ], [ 2, 1, 4, 3 ], [ 3, 4, 1, 2 ], [ 4, 3, 2, 1 ] ] ]
gap> CreamOneDirectlyReducible(algebra);
[ [ -2, 1, -2, 3 ], [ -2, -2, 2, 1 ] ]
gap> c3 := [MultiplicationTable(CyclicGroup (3))];
[ [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamOneDirectlyReducible(c3);
fail
gap> notAlgebra := [ algebra[1], c3[1] ];
[ [ [ 1, 2, 3, 4, 5, 6 ], [ 2, 1, 4, 3, 6, 5 ], [ 3, 5, 1, 6, 2, 4 ],
      [ 4, 6, 2, 5, 1, 3 ], [ 5, 3, 6, 1, 4, 2 ], [ 6, 4, 5, 2, 3, 1 ] ],
  [ [ 1, 2, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ] ] ]
gap> CreamOneDirectlyReducible(notAlgebra);
fail
gap>
```

Appendix B

CREAM: a Package to Compute [Auto, Endo, Iso, Mono, Epi]-morphisms, Congruences, Divisors and More for Algebras of Type $(2^m, 1^n)$

B. CREAM: A PACKAGE TO COMPUTE [AUTO, ENDO, ISO, MONO, EPI]-MORPHISMS, CONGRUENCES, DIVISORS AND MORE FOR ALGEBRAS OF TYPE $(2^M, 1^N)$

CREAM: A PACKAGE TO COMPUTE [AUTO, ENDO, ISO, MONO, EPI]-MORPHISMS, CONGRUENCES, DIVISORS AND MORE FOR ALGEBRAS OF TYPE $(2^n, 1^n)$

JOÃO ARAÚJO, RUI BARRADAS PEREIRA, WOLFRAM BENTZ, CHOIWAH CHOW, JOÃO RAMIRES, LUIS SEQUEIRA, AND CARLOS SOUSA

ABSTRACT. The CREAM GAP package computes automorphisms, congruences, endomorphisms and subalgebras of algebras with an arbitrary number of binary and unary operations; it also decides if between two such algebras there exists a monomorphism, an epimorphism, an isomorphism or if one is a divisor of the other. Thus it finds those objects for almost all algebras used in practice (groups, quasigroups in their various signatures, semigroups possibly with many unary operations, fields, semi-rings, quandles, logic algebras, etc).

As a one-size-fits-all package, it only relies on universal algebra theorems, without taking advantage of specific theorems about, eg, groups or semigroups to reduce the search space. Canon and Holt produced very fast code to compute automorphisms of groups that outperform CREAM on orders larger than 128. Similarly, Mitchell et al. take advantage of deep theorems to compute automorphisms and congruences of completely 0-simple semigroups in a very efficient manner. However these domains (groups of order above 128 and completely 0-simple semigroups) are among the very few examples of GAP code faster than our general purpose package CREAM. For the overwhelming majority of other classes of algebras, either ours is the first code computing the above mentioned objects, or the existing algorithms are outperformed by CREAM, in some cases by several orders of magnitude.

To get this performance, CREAM uses a mixture of universal algebra algorithms together with GAP coupled with artificial intelligence theorem proving tools (AITP) and very delicate C implementations. As an example of the latter, we re-implement Freese's very clever algorithm for computing congruences in universal algebras, in a way that outperforms all other known implementations.

1. INTRODUCTION

Investigation of automorphism groups of mathematical structures is one of the classical algebraic problems. A cornerstone was the work of Evariste Galois, but its impact goes far beyond. In the words of P. J. Cameron [22]:

In the famous Erlanger Programme, Klein proposed that geometry is the study of symmetry; more precisely, the geometric properties of an object are those which are invariant under all automorphisms of the objects. (...) According to Artin, "the investigation of symmetries of a given mathematical structure has always yielded the most powerful results."

These ideas, expressed by Klein in 1872 [36] and by Artin in 1957 [18], give some insight on why the computation of automorphisms of various mathematical structures has been such an attractive topic for so many

MATHEMATICS DEPARTMENT, FACULTY OF SCIENCE AND TECHNOLOGY, UNIVERSIDADE NOVA DE LISBOA, PORTUGAL
DEPARTMENT OF SCIENCE AND TECHNOLOGY, UNIVERSIDADE ABERTA, LISBON, PORTUGAL
DEPARTMENT OF SCIENCE AND TECHNOLOGY, UNIVERSIDADE ABERTA, LISBON, PORTUGAL
DEPARTMENT OF SCIENCE AND TECHNOLOGY, UNIVERSIDADE ABERTA, LISBON, PORTUGAL
DEPARTMENT OF SCIENCE AND TECHNOLOGY, UNIVERSIDADE ABERTA, LISBON, PORTUGAL
MATHEMATICS DEPARTMENT, UNIVERSIDADE DE LISBOA, LISBON, PORTUGAL
DEPARTMENT OF SCIENCE AND TECHNOLOGY, UNIVERSIDADE ABERTA, LISBON, PORTUGAL
E-mail addresses: jj.araujo@fct.unl.pt, rmbper@gmail.com, wolfram.bentz@uab.pt, choiwah.chow@gmail.com, joao.j.ramires@gmail.com, lfsequeira@fc.ul.pt, cfmsousa@sapo.pt.

decades. It should be observed here that this effort is closely linked to an omnipresent problem in mathematics – stated explicitly by Ulam [60] – of describing the mathematical objects whose endomorphisms encode *all the relevant information* about them:

$$\mathrm{End}(A) \cong \mathrm{End}(B) \Rightarrow A \cong B.$$

The connection between endomorphisms and congruences is well known, but the importance of congruences goes far beyond a tool to get endomorphisms; in fact their importance in modern algebra can hardly be overestimated.

Congruences have close ties to the structure of algebras - for example, the subdirectly irreducible algebras are precisely the ones that possess a unique minimal nontrivial congruence; direct irreducibility is also characterized by properties of congruences. In Universal Algebra, a variety is a class of algebras of the same type, that satisfy a certain set of identities, or equivalently, by a celebrated theorem of Garrett Birkhoff (see [21, Theorem II.11.9]), a class closed under taking subalgebras, homomorphic images and direct products. Each variety is also generated by its subdirectly irreducible algebras. Properties of congruence lattices of algebras over a variety are closely related to the identities satisfied in that variety. This gave rise to what is usually referred to the theory of Maltsev conditions - see, for example, [31]. The commutator of normal subgroups was generalized for algebras, first those in a congruence-permutable variety [55], then for any algebra in a congruence modular variety [33]; see [29]. Thus notions like Abelianness or nilpotency can be applied meaningfully in a much wider context, and many fruitful consequences have been derived, for example in the context of natural dualities [24]. Tame Congruence Theory [34] emerged as a powerful tool to study the structure of finite algebras, and continues to provide a big leverage in the study of locally finite varieties. These examples point to the critical importance of the study of congruences in algebra.

As observed above, congruences are closely related to endomorphisms and the latter form a monoid (semigroup with identity) that very often encodes very relevant information about the original object and can be investigated with the increasingly powerful techniques developed by experts in semigroup theory (especially since they started to massively apply the classification of finite simple groups [1,3–8,10–12,19, 49,51,54,57]).

Everything said above should make the case in favour of a general centralized and very effective GAP tool that computes congruences, automorphisms and endomorphisms of general algebras of type $(2^m, 1^n)$. We chose these since the overwhelming majority of algebras used in practice have this type: it is a very rare occurrence that an algebraic structure actually uses ternary or higher arity operations (see [21]).

The problem is that computing [endo]automorphisms/congruences of general algebras is a difficult task and hence the majority of existing tools target specific classes (groups, semigroups, quasigroups, etc.) in order to take advantage of the domain's theorems. Probably the best example is the GAP code to find automorphisms of groups, devised by J. J. Cannon and D. Holt [23], that takes advantage of many deep theorems and builds upon many past computational optimizations to quickly get auxiliary objects. In contrast, since we want our algorithms to apply for general algebras of type $(2^m, 1^n)$, we cannot rely on theorems that only hold in very specific cases. The challenge of this project is thus to provide an effective tool to compute the objects for finite algebras of type $(2^m, 1^n)$ that works for all algebras of this type and does not fall far behind the more dedicated tools that take advantage of very specific domain theorems.

The final result is that CREAM compares favourably even with the specialized Cannon and Holt tool referred to above: for groups up to order 128, our general purpose tool finds the automorphisms of a group faster than their very optimized code. For most other classes of algebras, our tool outperforms specialized code, in some cases by large margins. This was achieved by a combination of results and ideas from different parts of computational algebra, artificial intelligence theorem proving, and optimized C implementations. Regarding congruences, we take advantage of a new implementation of Freese's algorithm [28], an algorithm that stands out by its quality and generality; for automorphisms, after many different tests and approaches, the most effective way turned out to be a generalization of the ideas developed in [48]. Once possessing a very good tool to compute automorphisms, finding [mono,epi,iso]-morphisms is straightforward.

Finally, our tool also handles divisors. Given two algebras, A and B, we say that B divides A if B is a homomorphic image of a subalgebra of A. In different words, B divides A if there is a subalgebra C of A and a congruence ρ of C such that C/ρ is isomorphic to B. Like congruences, divisors are very important tools. For example, the celebrated Krohn–Rhodes theorem states that every finite semigroup S divides a wreath product of finite simple groups, each dividing S, together with copies of the 3-element monoid $\{1, a, b\}$, where $ba = aa = a$, $ab = bb = b$ (for details see [53]). The key difficulty when finding divisors is to find the subalgebras of a given algebra; CREAM has an algorithm to compute them.

There are numerous papers concerning the automorphism groups of particular classes of algebras, for example, Schreier [52] and Mal'cev [42] described all automorphisms of the semigroup of all mappings from a set to itself. Similar results have been obtained for various other structures such as orders, equivalence relations, graphs, and hypergraphs; see the survey papers [46] and [47]. More examples are provided, among others, by Gluskĭn [32], Araújo and Konieczny [14], [15], and [16], Fitzpatrick and Symons [26], Levi [38] and [39], Liber [40], Magill [41], Schein [50], Sullivan [58], and Šutov [59].

Fast algorithms exist to find the congruences and automorphisms of groups and semigroups. For example, with GAP [30], the package SEMIGROUPS includes the function CongruencesOfSemigroup that returns the congruences of a given semigroup. This function only works for a limited set of semigroups, belonging to the classes Simple, Brandt, Group, Zero Simple or Rectangular Band, and only for Rees Matrix semigroups and Rees Zero Matrix semigroups is it highly efficient. On a different platform, UACalc [27] supports a wider range of algebras. UACalc also implements the Freeze algorithm that is the basis for the calculation of congruences described here.

Regarding automorphisms in GAP, many special properties of groups are used to implement the function *AutomorphismGroup* to efficiently find the automorphism group of a given group. However, this specialized method, while extremely efficient, works only on groups and not on any other more general algebraic structures, such as quasigroups, semigroups and magmas. Likewise, the Loops package [48] in GAP provides another version of the function *AutomorphismGroup* to compute all the automorphisms of a given quasigroup.

To date, there are no known implementations of general functions for finding divisors, congruences and [endo, auto, epi, mono, iso]-morphisms of magmas or algebras of type $(2^m, 1^n)$ in GAP. To fill the void, the CREAM package implements such functions in GAP with part of the code written in C for performance.

This article is composed of 6 sections: this introduction, a short description of the mathematical concepts and background, the description of the algorithms used in the CREAM package, a comparative discussion of the performance of the package, a list of applications of the package and finally a short conclusion.

2. Mathematical Background

2.1. **Algebra of Type (2^m,1^n).** An Algebra in the sense of Universal Algebra is an algebraic structure consisting of a set A together with a collection of operations on A. (If we want to be more precise, an algebra also contains an indexed scheme that references the operations, but we are not going to enter those technical details; for a complete definition see [21].)

An n-ary operation on A is a function that takes an n-tuple of elements from A and returns a single element of A, that is, a function from A^n to A. The number n is called the *arity* of the operation. For the scope of this package we are only considering operations with arity 1 or 2, i.e., unary and binary operations. An algebra of type (2^m,1^n) is a universal algebra with m binary and n unary operations.

The package represents a finite Universal Algebra as a list of operations. An operation of arity 1 is represented by a vector, while an operation of arity 2 is represented by a square matrix. The underlying set A is implicitly specified by the vector or matrix sizes, which need to agree for a valid representation. If this size is d, the algebra is defined on the set $A = \{1, \ldots, d\}$. We can safely ignore the ambiguity this introduces in the (uninteresting) case of an algebra without operations. Each vector or matrix describes the corresponding operation by listing all images in the obvious way. The following is an example of a representation for an algebra with a unary and a binary operation.

```
[ [3, 1, 2], [ [1, 2, 3], [2, 3, 1], [3, 1, 2] ] ]
```

For those CREAM functions that involve more than one algebra, the operations of the algebras need to be aligned by a common index scheme. In our representation, this index is indirectly provided by the order in which the operations are listed. The involved algebras need to be compatible, that is, they have the same number of operations of each arity, and the operations are listed in the same position in the algebra representation.

2.2. **Partition.** A partition of a set A is a collection of non-empty subsets of A, such that every element of A is included in exactly one subset [21]. Each subset in a partition is called a block or part. For example, $\{\{1,3\},\{2,4\}\}$ is a 2-blocks partition of the set $\{1,2,3,4\}$.

A block of a partition is efficiently represented as a tree. A partition is represented by a forest, that is, a disjoint union of trees. The forest is physically presented in the computer memory by an array, whose size will be the size of the set A.

In a tree, each node has a parent node, except for the top node or root. In the array representation, each position of the array represents a node and the value of the node points to its parent node. Top nodes should have a value indicating that the node is the root of the tree. A negative number is used to signal the root, and the absolute value of this number is the number of elements of the block.

A shallow tree is a tree in which all non-root nodes are connected directly to the root. To obtain a unique representation, we adopt the convention that the trees used in presenting a partition are shallow, and that the smallest value in a block will be the root node. A partition representation that respects the above conventions will be called a normalized (representation of a) partition.

Thus, for $l, m \geq 1$, the array

$$\begin{array}{cccccc} [\ \ldots & ,-l, & \ldots & ,m, & \ldots\ ] \\ & \uparrow & & \uparrow & \\ & \text{position } i & & \text{position } k & \end{array}$$

means that i is the root of a block (as its entry is negative) of size l; in addition the element k belongs to the block whose root is m.

Using these conventions, the encoding of the partition `[[1], [2], [3], [4], [5], [6]]` induced by the identity relation is

```
[-1, -1, -1, -1, -1, -1]
```

while the partition `[[1, 2, 3, 4, 5, 6]]` with just one block is encoded as

```
[-6, 1, 1, 1, 1, 1]
```

Other examples are given below.

partition	encoded as
`[[1, 6], [2], [3, 5], [4]]`	`[-2, -1, -2, -1, 3, 1]`
`[[1, 3, 5], [2, 6], [4]]`	`[-3, -2, 1, -1, 1, 2]`
`[[1, 2, 5, 6], [3, 4]]`	`[-4, 1, -2, 3, 1, 1]`

This non-intuitive way of representing partitions will prove its usefulness later on.

2.3. **Congruences.** A congruence of an algebra is an equivalence relation on its underlying set that is compatible with all algebraic operations [21].

Technically, a congruence relation is an equivalence relation $\equiv$ on an algebra that satisfies $\mu(a_1, a_2, ..., a_n) \equiv \mu(a'_1, a'_2, ..., a'_n)$ for every n-ary operation μ and all elements $a_1, ..., a_n, a'_1, ..., a'_n$ such that $a_i \equiv a'_i$ for each $i = 1, ..., n$.

Congruences will be represented in the same way as their corresponding partitions, as detailed in the previous section.

2.4. **Homomorphism, Endomorphism, Isomorphism and Automorphism.** If A, B are two algebras of the same type, then a function $f : A \to B$ is a *homomorphism* from A to B if $\mu(f(a_1), f(a_2), ..., f(a_n)) = f(\mu(a_1, a_2, ..., a_n))$ for every n-ary operation μ and $a_1, \ldots a_n \in A$ (more precisely, μ here stands for the two operations of A and B that are indexed equally by the (common) index scheme).

An *endomorphism* is a *homomorphism* from an algebra to itself, a *monomorphism* is an injective *homomorphism*, while an *isomorphism* is a bijective *homomorphism*. An *automorphism* is a *homomorphism* that is both an *isomorphism* and an *endomorphism*.

3. The Algorithms

3.1. **Congruences algorithm.** The starting point for the computation of congruences are the algorithms described in Freese [28] to calculate the smallest congruence containing a given partition Θ of a finite algebra A; in particular this is used to compute the smallest congruence containing a pair of elements $(a, b) \in A \times A$, called the *principal congruence* generated by $\{a, b\}$. From this base algorithm all congruences of the algebra A are generated in an efficient way. Optimizations to the original algorithm were introduced taking into account that we only allow operations of arity at most 2. In addition, we take advantage of C to get a faster implementation than the original implementations made by Freese and his collaborators. Finally, our implementation is integrated with GAP and hence fully compatible with its other resources.

3.1.1. *Partition Functions.* There are several partition functions that play an important role in the algorithm to compute principal congruences.

The function **CreamRootBlock** is an operation that returns the root node for a node i. The root node of a node is itself if the value of the node representation is negative. If the value of the node representation points to a different node then that node is the parent node. This algorithm could be run recursively until reaching the root node but is implemented iteratively for better performance. This operation will work both for normalized and non-normalized partitions.

Before returning, the node parent of i is set to the found root node in order to make the tree representing the partition as shallow as possible, thus avoiding that in future calls the algorithm needs to transverse several nodes to reach the root node.

Algorithm 1 CreamRootBlock (i, partition)

```
$j \leftarrow i$
while $partition[j] \geq 0$ do
  $j \leftarrow partition[j]$
end while
if $i \neq j$ then
  $partition[i] \leftarrow j$
end if
return $j$
```

The function **CreamJoinBlocks** is an operation that joins the blocks containing given elements x and y. This operation will work both for normalized and non-normalized partitions. The resulting partition may not in general be normalized even if the original partition is.

In order to keep the tree representing the partition as shallow as possible, the root node of the merged block will be the root node of the larger original block.

Algorithm 2 CreamJoinBlocks (x, y, partition)

```
$r \leftarrow CreamRootBlock(x, partition)$
$s \leftarrow CreamRootBlock(y, partition)$
**if** $r \neq s$ **then**
  **if** $partition[r] < partition[s]$ **then**
    $partition[r] \leftarrow partition[r] + partition[s]$
    $partition[s] \leftarrow r$
  **else**
    $partition[s] \leftarrow partition[r] + partition[s]$
    $partition[r] \leftarrow s$
  **end if**
**end if**
```

The **CreamNumberOfBlocks** function returns the number of blocks of a partition. Given the encoding of partitions we simply count the number of positions of the array that have negative values.

Algorithm 3 CreamNumberOfBlocks (partition)

```
$nblocks \leftarrow 0$
$dimension \leftarrow ArraySize(partition)$
**for** $i = 1$ to $dimension$ **do**
  **if** $partition[i] < 0$ **then**
    $nblocks \leftarrow nblocks + 1$
  **end if**
**end for**
**return** $nblocks$
```

The **CreamNormalizePartition** function normalizes the partition by making it shallow and having the smallest element of each block the root node.

Algorithm 4 CreamNormalizePartition (partition)

```
$dimension \leftarrow ArraySize(partition)$
**for** $i = 1$ to $dimension$ **do**
  $r \leftarrow CreamRootBlock(i, partition)$
  **if** $r \geq i$ **then**
    $partition[i] \leftarrow -1$
    **if** $r > i$ **then**
      $partition[r] \leftarrow i$
    **end if**
  **else**
    $partition[r] \leftarrow partition[r] - 1$
  **end if**
**end for**
```

The **CreamJoinPartition** function computes the join of two partitions, i.e. the smallest partition containing both input partitions.

Algorithm 5 CreamJoinPartition (partition1, partition2)

```
$dimension \leftarrow ArraySize(partition1)$
**for** $i = 1$ to $dimension$ **do**
    CreamJoinBlocks($i$,CreamRootBlock ($i$,$partition1$),$partition2$)
**end for**
```

The **CreamComparePartitions** function compares normalized partitions. Returns 0 if the partitions are equal, -1 if partition1 $>$ partition2, and 1 if partition1 $<$ partition2. Partitions are ordered by the order of the underlying list. The purpose of this function is to allow binary search in sets of partitions.

Algorithm 6 CreamComparePartitions (partition1, partition2)

```
$dimension \leftarrow Length(partition1)$
**for** $i = 1$ to $dimension - 1$ **do**
    **if** $partition1[i] > partition2[i]$ **then**
        **return** $-1$
    **end if**
    **if** $partition1[i] < partition2[i]$ **then**
        **return** $1$
    **end if**
**end for**
**return** $0$
```

3.1.2. *Computing Principal Congruences.* The base algorithm in [28] computes the smallest congruence containing Θ, a partition of a finite algebra A. The simplest case is when Θ only contains one nontrivial block and this block has only two elements.

The algorithm only works with unary operations. Since our algebras have binary operations, we have to convert them to a family of unary operations. In this case a binary operation $f(x, y)$ can be converted into $2n$ unary operations (where n is the size of the algebra) by assigning to each element $x \in A$ the unary operations c_x and r_x, which are induced by the corresponding column and row in the Cayley table of f:

$$c_x(y) = f(y, x) \text{ and } r_x(y) = f(x, y).$$

For example, the algebra:

$$[[[2, 1, 1], [1, 2, 2], [1, 3, 2]]]$$

induces the following unary operations (after removing duplicates):

$$[[2, 1, 1], [1, 2, 2], [1, 3, 2], [1, 2, 3]].$$

The principal congruence algorithm takes as input the algebra and the generating pair of elements.

The algorithm joins the pair of elements in the same block and applies the algebra's unary functions to each element of the pair. In this way, it obtains one additional pair per function that will be joined in the same block, repeating this process until the list of pairs is exhausted.

Algorithm 7 CreamPrincipalCongruence (algebra,InitialPair)

```
$PairList \leftarrow [InitialPair]$
$partition \leftarrow SingletonPartition()$
$partition \leftarrow CreamJoinBlocks(partition, InitialPair[1], InitialPair[2])$
$NFuncs \leftarrow ArraySize(algebra)$
**while** $PairList \neq empty$ **do**
    $Pair \leftarrow Pop(PairList)$
    **for** $i = 1$ to $NFuncs$ **do**
        $f \leftarrow algebra(i)$
        $r \leftarrow CreamRootBlock(partition, f(Pair[1]))$
        $s \leftarrow CreamRootBlock(partition, f(Pair[2]))$
        **if** $r \neq s$ **then**
            $partition \leftarrow CreamJoinBlocks(partition, r, s)$
            $PairList \leftarrow Push(PairList, [r, s])$
        **end if**
    **end for**
**end while**
**return** $partition$
```

As described in [28] this algorithm is very efficient showing a moderate growth of execution time with n. Apart from the algorithm itself, several implementation choices were used to increase efficiency as well.

One of these aspects is the use of arrays to encode partitions which allow for a constant and very fast random access to the partition element.

This is combined with the balancing and collapsing of the trees representing the blocks in the partition that are parts of the **CreamJoinBlocks** and **CreamRootBlock** algorithms. This approach aims for trees that are as shallow as possible during the execution of the **CreamPrincipalCongruence** algorithm.

In **CreamJoinBlocks** the tree is balanced by keeping as root of the joined block the root of the bigger original block.

In **CreamRootBlock** the tree is collapsed by setting the parent node of the element on which the function is called equal to its return value, avoiding having to traverse several nodes of the tree in future calls.

Both of these implementation details allow for shallower trees representing the blocks, which will make **CreamRootBlock** faster, given that fewer nodes will have to be transversed to determine the root node of a node.

This is especially important since **CreamRootBlock** is the most called function when calculating a principal congruence of an algebra.

To compute all the principal congruences this function is called for all pairs of elements of A.

Algorithm 8 CreamAllPrincipalCongruences (algebra)

```
$dimension \leftarrow SizeAlgebra(algebra)$
$allPrincipalCongruences \leftarrow []$
**for** $i = 1$ to $dimension - 1$ **do**
    **for** $j = i + 1$ to $dimension$ **do**
        $congruence \leftarrow CreamPrincipalCongruence(algebra, [i, j])$
        $AddCongruence(allPrincipalCongruences, congruence)$
    **end for**
**end for**
**return** $allPrincipalCongruences$
```

3.1.3. *Calculating All Congruences.* It is known that the congruences of an algebra form a lattice with the usual set inclusion partial order. The meet is the usual intersection; the join of two congruences is the smallest congruence containing both.

The minimal elements of this lattice are precisely the principal congruences, and we can obtain all congruences by computing joins, starting with the principal congruences.

The key part is to find an efficient way to combine minimal congruences to get all congruences of the algebra. The principal congruences are stored in an ordered set and the congruences are combined from start to end by the join operation. Each congruence resulting from these joins will be added to the ordered set (eliminating duplicates).

Algorithm 9 CreamAllCongruences (algebra)

```
$allPrincipalCongruences \leftarrow CreamAllPrincipalCongruences(algebra)$
$allCongruences \leftarrow allPrincipalCongruences$
$i \leftarrow 1$
while $i < Size(allCongruences)$ do
  for $j = 1$ to $Size(allPrincipalCongruences)$ do
    if $\neg isContained(allPrincipalCongruences[j], allCongruences[i])$ then
      $congruence \leftarrow JoinPartition(allCongruences[i], allPrincipalCongruences[j])$
      $AddCongruence(allCongruences, congruence)$
    end if
  end for
  $i \leftarrow i + 1$
end while
return $allCongruences$
```

The join operation is only called if the principal congruence is not contained in the congruence that is being joined with. This is optimized by preserving the information about which principal partitions were joined.

3.1.4. *Calculating Monolithic Algebras.* Monolithic Algebras are algebras that have a single minimal congruence that is contained in every other congruence except for the identity congruence.

This is simple to calculate having the list of minimal congruences and a function that compares two partitions and calculates whether one is contained in the other.

The **CreamContainedPartition** function returns whether a partition is contained in another. This function is used internally to determine whether an Algebra is monolithic or not. This algorithm assumes that the partitions are normalized.

Algorithm 10 CreamContainedPartition (partition1, partition2)

```
$dimension = ArraySize(partition1)$
**for** $i = 1$ to $dimension$ **do**
  **if** $partition1[i] < 0$ **then**
    **if** $partition2[i] < 0$ **then**
      $block2 \leftarrow i$
    **else**
      $block2 \leftarrow partition2[i]$
    **end if**
    **for** $j = i + 1$ to $dimension$ **do**
      **if** $partition1[j] = i$ and not $block2 = partition2[j]$ **then**
        **return** $false$
      **end if**
    **end for**
  **end if**
**end for**
**return** $true$
```

We calculate the principal congruences and compare them eliminating those that contain other congruences. If a single congruence is returned then this congruence is contained in every other congruence (and the algebra is monolithic). This is done with the **CreamIsAlgebraMonolithic** function.

Algorithm 11 CreamIsAlgebraMonolithic (algebra)

```
$partitions \leftarrow CreamAllPrincipalCongruences(algebra)$
$npartitions \leftarrow Size(partitions)$
**if** $npartitions > 1$ **then**
  $i \leftarrow 2$
  **repeat**
    **if** $CreamContainedPartition(partitions[1], partitions[i])$ **then**
      $Remove(partitions, i)$
    **else if** $CreamContainedPartition(partitions[i], partitions[1])$ **then**
      $partitions[1] \leftarrow partitions[i]$
      $Remove(partitions, i)$
      $i \leftarrow 2$
    **else**
      $i \leftarrow i + 1$
    **end if**
  **until** Size (partitions) < i
**end if**
$npartitions \leftarrow Size(partitions)$
**if** $npartitions > 1$ **then**
  **return** $false$
**else**
  **return** $true$
**end if**
```

3.2. **Automorphism algorithm.** The automorphisms of an algebra A of type (2^m,1^n) can be derived from the calculation of the automorphisms of a set of algebras each containing only one binary operation of

A. The automorphism group of the algebra will be the intersection of all the automorphism groups of these magmas that also commute, function composition-wise, with all the unary operations of the original algebra.

In general, it is difficult to obtain automorphisms efficiently. The authors of the Loops GAP package [48] used an idea that we could apply to general magmas. With some adaptations this allows for very effective computing of the automorphisms of magmas.

3.2.1. *Invariant Vector.* The general idea is to pick a list of properties invariant under homomorphisms and then partition the algebras using these properties. For example, if e is idempotent, then any endomorphism must map e onto an idempotent. This is the basis of the *Discriminator* in the Loops package, which implemented nine such invariants for the domain elements of quasigroups. However, most of these invariants cannot be carried over directly to magmas. We devised a total of seventeen invariants that hold for any magma. In general, unless otherwise said, given an element x in a Magma M and $k \in \mathbb{Z}^+$, we define x^k to be $x^{k-1} * x$ for $k \geq 2$ and $x^1 = x$ (ie. we associate on the left). For each element p in a magma M, CREAM computes the following:

(1) Smallest k such that $p^k = p^n$, with $n > k > 1$.
(2) Number of domain elements for which p is a left identity.
(3) Number of domain elements for which p is a right identity.
(4) Number of elements y such that $p = (py)p$.
(5) Number of distinct elements in row p of the multiplication table.
(6) Number of distinct elements in column p of the multiplication table.
(7) 1 if p is an idempotent, 0 otherwise.
(8) Number of idempotents in column p of the multiplication table.
(9) Number of idempotents in row p of the multiplication table.
(10) 1 if the equality $p(pp) = (pp)p$ holds, 0 otherwise.
(11) Number of domain elements that commute with p.
(12) Number of domain elements s for which $(ss)p = p(ss)$.
(13) Number of domain elements s such that $s^2 = p$.
(14) Number of domain elements s satisfying $p(ps) = (pp)s$
(15) Number of multisets $\{x, y\}$ with $xy = yx = p$.
(16) Number of elements $t \in M$ such that for two idempotents $e, f \in M$, we have $p = et = tf$.
(17) Number of elements $t \in M$ for which there exist two elements $x, y \in M$ such that $p = xy$ and $t = yx$.

Many of these invariants have an algebraic meaning. For example, (17) is the size of the conjugacy class of p when the magma is a group; (16) is the size of the filter generated by p when the magma is a regular semigroup; (13) is the number of square roots of p; (11) is the size of the centralizer of p; (8) and (9) are the number of idempotents in the left or right ideals generated by p; (5) and (6) are the number of elements in the left or right ideals generated by p; (1) deals with the periodicity of p; etc.

The intersection of the blocks induced by these invariants partitions the elements of the algebra, and by a straightforward argument on first order logic any endomorphism must preserve the blocks of this partition.

3.2.2. *Generating Set.* We can cut down the size of the search tree by focusing only on the images of the elements in a generating set. Some generating sets may be more suitable for finding automorphisms than others:

- A smaller generating set is preferred because it would require fewer trials in constructing an isomorphism.
- Generators from partition blocks with fewer elements are preferred since any homomorphism must send the elements of a block into itself.

Our algorithm satisfies both the constraints above and efficiently produces a generating set. It is depicted in Algorithm 12.

Algorithm 12 Constructing Efficient Generating Set

```
$submagma \leftarrow [\,]$
$generator \leftarrow [\,]$
$candidates \leftarrow$ magma elements sorted by ascending block size
**while** $submagma \neq magma$ **do**
    $generator \leftarrow generator \cup$ the element that increase the size of $submagma$ most, and if 2 elements increase the size by the same amount, take the one from the smaller block.
    $submagma \leftarrow$ submagma generated by $generator$
    $candidates \leftarrow candidates$ with elements in submagma removed
**end while**
```

3.2.3. *The Automorphism Algorithm.* To find the automorphisms we start by partitioning the Magma according to the invariant vectors of each element and then obtain an efficient generating set. We then need to find the images of the generating set taking into account that each element in the generating set can only map to elements having the same invariant vector. Then we extend this partial map to a full map and check that the homomorphism condition is satisfied.

We tested the algorithm on groups, loops, and quasigroups, obtaining the same results as yielded by the existing tools. For magmas of orders 2 and 3, and semigroups of orders 6 and 7 (as there are no general tools to compute automorphisms) we double check our results against the output of Mace4 [44].

3.2.4. *Algebras of Type (2^m,1^n).* Given an algebra of type (2^m,1^n) its automorphism group can be computed by taking the intersection of all the automorphism groups of the binary operations that also commute, function composition-wise, with all the unary operations.

A useful observation is that the intersection of the trivial group and any group is the trivial group. Therefore, if any of the automorphism group generated in the process is a trivial group, then the search can be terminated with the trivial group declared the automorphism group for the algebra.

This algorithm is implemented in the **CreamAutomorphisms** function in the CREAM package, as shown in Algorithm 13.

Algorithm 13 CreamAutomorphisms(algebra)

```
$binaryOperations \leftarrow AllBinaryOperations(algebra)$
$unaryOperations \leftarrow AllUnaryOperations(algebra)$
$D \leftarrow \emptyset$
**repeat**
    $B \leftarrow$ Pop a binary operation from $binaryOperations$
    $C \leftarrow$ all automorphisms of $B$
    $C \leftarrow$ all automorphisms in $C$ that commute with every unary operation in $unaryOperations$
    **if** $C = \emptyset$ **then**
        **return** $\emptyset$
    **end if**
    $D \leftarrow D \cup C$
**until** $binaryOperations = \emptyset$
**return** Intersection($D$)
```

3.3. **Endomorphisms algorithm.** The classic approach to calculate endomorphisms would be to use **Mace4** to search for endomorphisms and while this is very effective and fast for low order algebra further optimizations are needed for high order algebras in which the congruences of the algebra can be used to limit the **Mace4** search space for endomorphisms.

To compute all endomorphisms of high order algebras, CREAM does the following:

(1) Compute the congruences of the algebra A;
(2) For each congruence R, compute A/R (except for the trivial congruence);
(3) Compute the subalgebras of A that are isomorphic to A/R (using the finite model builder Mace4);
(4) For each compatible pair subalgebra/congruence derive a corresponding endomorphism;
(5) Add the automorphisms of A.

The congruences of A are calculated with the algorithms described in 3.1. For each congruence R (except for the trivial congruence that will be dealt with later), a representation of the A/R operation can be obtained by replacing the values in A's operation tables by their roots, and eliminating rows and columns not corresponding to roots. If we have the algebra:

```
[ [ [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ],[ 1, 1, 1, 2, 3, 1 ] ] ]
```

and the congruence

```
[ [ 1, 2, 3], [4], [5, 6] ] - [ -3, 1, 1, -1, -2, 5 ],
```

then the A/R operation is:

	1	4	5
1	1	1	1
4	1	1	1
5	1	1	1

and passed to Mace4 as follows:

```
f(a0,a0)=a0.
a0!=a3.
f(a0,a3)=a0.
a0!=a4.
f(a0,a4)=a0.
f(a3,a0)=a0.
f(a3,a3)=a0.
a3!=a4.
f(a3,a4)=a0.
f(a4,a0)=a0.
f(a4,a3)=a0.
f(a4,a4)=a0.
```

where f is the binary operation of the algebra. It is worth observing that Mace4 is zero-based, unlike GAP, which is one-based.

Running Mace4 with the definition of the algebra operation and the above encoding of A/R as assumptions we get 16 possible models each one of them corresponding to a different endomorphism of the algebra A. One of these models would be in Mace4's output:

```
interpretation( 6, [number = 1,seconds = 0], [
    function(a0, [0]),
    function(a3, [1]),
    function(a4, [2]),
    function(fa1(_,_), [
        0,0,0,0,0,0,
        0,0,0,0,0,0,
        0,0,0,0,0,0,
        0,0,0,0,0,0,
        0,0,0,0,0,0,
        0,0,0,1,2,0])]).
```

From this model we get the partial mapping $f : \{1, 4, 5\} \subseteq A \to \{1, 2, 3\}$, defined by $f(1) = 1$, $f(4) = 2$ and $f(5) = 3$, that can be represented by $[1, , , 2, 3,]$. The gaps can be easily filled since elements of the same block must map to the same image. In this example, 2 and 3 have the same image as 1, while 6 and 5 have the same image yielding the endomorphism: $[1, 1, 1, 2, 3, 3]$.

Repeating this process for all congruences and for all models obtained with Mace4 from each congruence, all the algebra's endomorphisms (except for the automorphisms) can be obtained.

Finally, for the trivial congruence the associated endomorphisms would be also automorphisms and in this case it is more efficient to use the the algorithm described in 3.2 than using Mace4.

3.4. **Monomorphisms algorithms.** Invariants can be used to speed up the process of finding an monomorphism from one magma to another, or from one algebra to another. If f is an injective homomorphism from a magma A to a magma B, then it is a isomorphism from A to B restricted to the range of f. Thus, the same ideas of applying invariants in constructing automorphisms can be used for constructing monomorphisms. Specifically, an monomorphism f can map an element $a \in A$ to $b \in B$ only if a and b have the same invariant vector. This greatly reduces the search space for monomorphisms between A and B.

Algorithm 14 CreamAllMonomorphismMagmas(magma1, magma2)

```
$genL \leftarrow$ generating set from $magma1$
**for** $i$ = 1 to $Size(magma2)$ **do**
    $rangeM[i] \leftarrow$ list of elements of $magma2$ that have same invariant vector as $genL[i]$
**end for**
$monoMaps \leftarrow \emptyset$
**for all** mappings $m$ from $genL$ to elements of $rangeM$ s.t. $genL[i]$ maps only to elements in $rangeM[i]$
**do**
    **if** $m$ is an injective homomorphism **then**
        $monoMaps \leftarrow monoMaps \cup m$
    **end if**
**end for**
**return** $monoMaps$
```

Algorithm 15 CreamAllMonomorphism(algebra1, algebra2)

```
b1 $\leftarrow$ first binary operation in $algebra1$
b2 $\leftarrow$ first binary operation in $algebra2$
$monoList \leftarrow$ CreamAllMonomorphismMagmas($b1, b2$)
$algebraMonoList \leftarrow []$
**for** each $monoMap$ in $monoList$ **do**
    $mono \leftarrow true$
    **for** each corresponding pair of operations $b1 \in algebra1, b2 \in algebra2$ **do**
        **if** $monoMap$ is not an isomorphism from $b1$ to $b2$ **then**
            $mono \leftarrow false$
            break out of for loop
        **end if**
    **end for**
    **if** $mono$ **then**
        Add $monoMap$ to $algebraMonoList$ if not already in the list
    **end if**
**end for**
**return** $algebraMonoList$
```

In this section only the function CreamAllMonomorphisms is addressed but the Cream package also includes the functions CreamExistsMonomorphism (returning true if a monomorphism exists) and CreamOneMonomorphism (returning one monomorphism).

3.5. **Epimorphisms algorithm.** CreamAllEpimorphisms(A_1, A_2) returns all epimorphisms from A_1 to A_2, provided the algebras are compatible. The algorithm first finds all congruences φ of A_1 such that A_1/φ and A_2 are isomorphic. For each such φ, the corresponding epimorphisms are obtained by composing the quotient map with an isomorphism to A_2 and all automorphisms of A_2.

For efficiency, a size check is implemented before searching for isomorphisms and the automorphisms of A_2 are only calculated once.

Algorithm 16 CreamAllEpimorphisms(algebra1, algebra2)

```
$allCongruences \leftarrow CreamAllCongruences(algebra1)$
$dimension1 \leftarrow SizeAlgebra(algebra1)$
$dimension2 \leftarrow SizeAlgebra(algebra2)$
$autoList \leftarrow []$
$epiList \leftarrow []$
for all $cong$ in $allConguences$ do
  if $dimension2 = CreamNumberOfBlocks(cong)$ then
    $[qalgebra, mapToQalgebra] \leftarrow QuotientAlgebraFromCongruence(algebra1, cong)$
    $iso \leftarrow IsomorphismAlgebras(qalgebra, algebra2)$
    if not $iso = fail$ then
      if autoList = [] then
        $autoList \leftarrow CreamAutomorphisms(algebra2)$
      end if
      for all $auto$ in $autoList$ do
        $epi \leftarrow []$
        for $j = 1$ to $dimension1$ do
          $epi[j] \leftarrow auto[iso[mapToQalgebra[i]]]$
        end for
        Add $epi$ to $epiList$ if not already in the list
      end for
    end if
  end if
end for
return $epiList$
```

In this section only the function CreamAllEpimorphisms is addressed but the Cream package also includes the functions CreamExistsEpimorphism (returning true if a epimorphism exists) and CreamOneEpimorphism (returning one epimorphism).

3.6. **SubUniverses algorithm.** The SubUniverses algorithm returns a list of all underlying sets of all subalgebras of the input algebra. It first generates all 1-generated subalgebras, then iteratively expands each i-generated algebra by adding another generator.

For performance enhancement, these expansions are limited to one element from each orbit of the algebra's automorphism group. At the end of each cycle, the remaining $(i + 1)$-generated subuniverses are obtained by applying the automorphism group. The algorithm stops when an iteration produces only one subuniverse with the same size of the algebra.

The algorithm relies on the SubUniverseFromElement routine, which calculates the subalgebra generated by a subalgebra and an additional element. This routine assumes that the input subuniverse is closed under the algebra's operations. The algorithm adds the element to the updated subuniverse, calculates all directly generated additional elements, and places them in a temporary list. These elements are then filtered for those that are already in the subuniverse, and the remaining elements are put into a second list. Elements from the second list are then added to the subuniverse iteratively.

Algorithm 17 CreamAllSubUniverses(algebra)

```
$dimension \leftarrow CreamSizeAlgebra(algebra)$
$autoList \leftarrow CreamAutomorphisms(algebra)$
**for** $i = 1$ to $dimension$ **do**
    $sigma[i] \leftarrow []$
    **if** $i = 1$ **then**
        $sigmaMinus \leftarrow [[]]$
    **else**
        $sigmaMinus \leftarrow sigmaExpanded[i-1]$
    **end if**
    **for all** $cSigma$ in $sigmaMinus$ **do**
        $elemList \leftarrow [1..dimension]$
        Remove all elements in $cSigma$ from $elemList$
        **while** $Size(elemList) <> 0$ **do**
            $j \leftarrow elemList[1]$
            $sUniverse \leftarrow SubUniverseFromElement(algebra, cSigma, j)$
            Add $sUniverse$ to $sigma[i]$ if not already in the list
            $autoEs \leftarrow$ the orbit of $j$ under $autoList$
            Remove all elements in $autoEs$ from $elemList$
        **end while**
    **end for**
    $sigmaExpanded[i] \leftarrow []$
    **for all** $sUniverse$ in $sigma[i]$ **do**
        $autoUniverses \leftarrow$ the orbit of $sUniverse$ under $autoList$
        Add all elements of $autoUniverses$ to $sigmaExpanded[i]$ if not already in the list
    **end for**
    **if** $Size(sigma[i]) = 1$ and $Size(sigma[i][1]) = dimension$ **then**
        $break$
    **end if**
**end for**
$sUniverses \leftarrow \cup_{j=1}^{i} sigmaExpended[j]$
**return** $sUniverses$
```

Algorithm 18 SubUniverseFromElement(algebra, initSubUniverse, element)

```
$dimension \leftarrow CreamSizeAlgebra(algebra)$
$elemList \leftarrow [element]$
$newSubUniverse \leftarrow initSubUniverse$
$nElem \leftarrow Size(elemList)$
**while** $nElem > 0$ **do**
  $currentElem \leftarrow elemList[nElem]$
  Remove element in position $nElem$ from $elemList$
  $newElemList \leftarrow []$
  **for all** operations $op$ of the $algebra$ **do**
    **if** $op$ is binary **then**
      Add $op[currentElem][currentElem]$ to $newElemList$ if not already in the list
      **for all** $subUniverseElem$ in $newSubUniverse$ **do**
        Add $op[currentElem][subUniverseElem]$ to $newElemList$ if not already in the list
        Add $op[subUniverseElem][currentElem]$ to $newElemList$ if not already in the list
      **end for**
    **else**
      Add $op[currentElem]$ to $newElemList$ if not already in the list
    **end if**
  **end for**
  Add $currentElem$ to $newSubUniverse$ if not already in the list
  $elemList \leftarrow elemList \cup (newElemList \setminus newSubUniverse)$
  $nElem \leftarrow Size(elements)$
  **if** $Size(newSubUniverse) + nElem = dimension$ **then**
    $newSubUniverse \leftarrow [1..dimension]$
    $nElem \leftarrow 0$
  **end if**
**end while**
**return** $newSubUniverse$
```

3.7. **DivisorUniverses algorithm.** CreamAllDivisorUniverses(A_1, A_2) checks if A_1 has a divisor that is isomorphic to A_2, where A_1 and A_2 are compatible algebras. Its return is a list of all pairs (B, φ), such that B is a subuniverse of A_1, φ is a congruence of the subalgebra B, and B/φ is isomorphic to A_2.

The algorithm first calculates the subalgebras of A_1, then their quotients and then checks those for isomorphisms to A_2, in each case using the corresponding algorithms of the Cream package. A size condition is used to prune unnecessary calculations.

Algorithm 19 CreamAllDivisorUniverses(algebra1, algebra2)

```
  $dimension \leftarrow CreamSizeAlgebra(algebra2)$
  $subUniverseList \leftarrow CreamSubUniverses(algebra1)$
  $divisorList \leftarrow []$
  for all $subUniverse$ in $subUniverseList$ do
    if $dimension <= Size(subUniverse)$ then
      $subAlgebra \leftarrow CreamSubUniverse2Algebra(algebra1, subUniverse)$
      $congList \leftarrow CreamAllCongruences(subAlgebra)$
      $rcongList \leftarrow []$
      for all $cong$ in $congList$ do
        if $dimension = CreamNumberOfBlocks(cong)$ then
          Add $cong$ to $rcongList$ if not already in the list
        end if
      end for
      for all $cong$ in $rcongList$ do
        $[qAlgebra, mapToQalgebra] \leftarrow QuotientAlgebraFromCongruence(subAlgebra, cong)$
        if $CreamAreAlgebrasIsomorphic(qAlgebra, algebra2)$ then
          Add $[subUniverse, cong]$ to $divisorList$ if not already in the list
        end if
      end for
    end if
  end for
  return $divisorList$
```

In this section only the function CreamAllDivisorUniverses is addressed but the Cream package also includes the functions CreamExistsDivisor (returning true if a divisor between the algebras exists) and CreamOneDivisorUniverse (returning one divisor between the algebras).

3.8. **DirectlyReducible algorithm.** CreamAllDirectlyReducible(A) looks for two (non-trivial) commuting congruences ϕ, ψ of A, whose meet is trivial and whose join is $A \times A$. If those exist, A is isomorphic to the direct product $A/\phi \times A/\psi$.

Because all algebras are finite, it suffices to check that the congruences have a trivial meet, and that the corresponding product algebra has the correct size. The algorithm uses numerical constraints to avoid unnecessary calculations.

The return of the algorithm is a list of pairs of congruences. These pairs should be consider as sets, i.e. if (α, β) is listed the (equally valid) pair (β, α) is not listed separately.

Algorithm 20 CreamAllDirectlyReducible(algebra)

```
$dimension \leftarrow CreamSizeAlgebra(algebra)$
$congList \leftarrow CreamAllCongruences(algebra)$
$pCongList \leftarrow []$
**for all** $cong$ in $congList$ **do**
  $cNBlocks \leftarrow CreamNumberOfBlocks(cong)$
  **if** $cNBlocks <> 1$ and $cNBlocks <> dimension$ and $IsInt(dimension/cNBlocks)$ **then**
    Add $cong$ to $pCongList$
  **end if**
**end for**
$pairs \leftarrow []$
**for all** 2-element subsets $\{i, j\}$ of $pCongsList$ **do**
  $iNBlocks \leftarrow CreamNumberOfBlocks(i)$
  $jNBlocks \leftarrow CreamNumberOfBlocks(j)$
  **if** $iNBlocks * jNBlocks = dimension$ **then**
    $cMeet \leftarrow MeetPartition(i, j)$
    $cMeetNBlocks \leftarrow CreamNumberOfBlocks(cMeet)$
    **if** $cMeetNBlocks = dimension$ **then**
      Add $[i, j]$ to $pairs$
    **end if**
  **end if**
**end for**
**return** $pairs$
```

In this section only the function CreamAllDirectlyReducible is addressed but the Cream package also includes the functions CreamExistsDirectlyReducible (returning true if there are commuting congruences R, T where A/R x A/T is isomorphic to A) and CreamOneDirectlyReducible (returning one pair of commuting congruences R, T where A/R x A/T is isomorphic to A).

4. Performance

4.1. **Congruences algorithm performance.** The **semigroup** GAP package [45] includes the function **CongruencesOfSemigroup** that returns the congruences of a semigroup, for some classes of semigroups. For Rees matrix semigroups the function **CongruencesOfSemigroup** takes advantage of theoretical results and hence achieves better results than **CreamAllCongruences**. But to achieve this kind of performance the semigroup needs to be created using the GAP functions **ReesMatrixSemigroup** or **ReesZeroMatrixSemigroup**. If instead an isomorphic copy of a Rees Matrix (Zero) semigroup is created using a multiplication table the performance will be much worse.

For Rees Matrix Semigroups or Rees Zero Matrix Semigroups (generated with **ReesMatrixSemigroup** or **ReesZeroMatrixSemigroup**) the function **CongruencesOfSemigroup** takes advantage of their particular structure by applying the efficient linked triple algorithm [35]. However, this efficiency can only be realized in this restricted setting.

UACalc also implements Freese's algorithm, and was done under his supervision in Java and Jython. It includes a GUI version in Java and a command line interface in Jython. The performance of the GUI version is poor and it will not be used for comparison. On the other hand the command line interface has a performance that is comparable to **CreamAllCongruences**. The same random Rees Matrix Semigroups that were used with **CongruencesOfSemigroup** and **CreamAllCongruences** were written into files readable by **UACalc** and the congruences were calculated in **UACalc**.

Table 1. Performance Comparison on calculating all congruences of Rees Matrix Semigroups.

Size	CongruencesOfSemigroup (ReesMatrixSemigroup) (ms)	CongruencesOfSemigroup (MultiplicationTable) (ms)	CreamAllCongruences (ms)	UACalc (ms)
12	1	15	< 1	1
18	2	28	< 1	3
24	5	55	2	7
30	1	92	4	32
36	4	139	10	34
42	11	203	16	55
48	8	296	24	77
54	5	416	46	138
60	5	553	60	220
66	1	671	82	291
72	9	873	136	409
78	2	1 098	180	555
84	5	1 341	233	743
90	4	1 622	307	961

CreamAllCongruences is consistently more than 3 times faster than **UACalc**. For these algebras with one binary operation the runtime rises roughly proportionally to n^4 (where n is the size of the algebra) or t^2 (being t the number cells of the multiplication matrix of the algebra operation).

4.2. **Automorphisms algorithm performance.**

The CREAM package automorphism function was run on algebras of Type $(2^m,1^n)$ to compare the timings against the Loops package. All experiments were run 7 times, with the lowest and highest times discarded. The averages of the remaining 5 are reported in Table 2. We see that the speeds from both packages are quite close.

Table 2. Performance Comparison between Loops and CREAM on Automorphism Group Generation.

Algebraic Structure	Loops Package(sec)	CREAM Package(sec)
Quasigroups, order 5	0.588	0.562
Quasigroups, order 6	520	541
Loops, order 6	0.101	0.086
Loops, order 7	14.623	13.945
Groups, order 32	0.854	0.935
Groups, order 64	39.98	41.05
Groups, order 128	12,031	12,160

4.3. **Endomorphisms algorithm performance.**

To evaluate the performance of the algorithms to calculate endomorphisms, we tested both the classic algorithm and the congruence algorithm running with **Mace4**, as described in 3.3, on several types of Semigroups, Monoids and Groups. The only GAP function to calculate endomorphisms that came to our attention is the function Endomorphisms from the package SONATA. This function can calculate endomorphisms for a very narrow set of algebras, namely groups and near-rings. The groups in the list algebras were also run with SONATA.

Table 3. Performance Comparison between endomorphisms calculation algorithms

Algebra	Size	Number of Endomorphisms	Classic (ms)	Congruences (ms)	SONATA (ms)
GossipMonoid(3)	11	66	30	1 635	NA
PlanarPartitionMonoid(2)	14	72	56	233	NA
JonesMonoid(4)	14	72	45	232	NA
BrauerMonoid(3)	15	28	38	160	NA
PartitionMonoid(2)	15	89	50	313	NA
FullPBRMonoid(1)	16	1 426	585	5 134	NA
SymmetricGroup(4)	24	58	101	142	166
FullTransformationSemigroup(3)	27	40	116	364	NA
FullTransformationMonoid(3)	27	40	112	360	NA
SymmetricInverseMonoid(3)	34	54	274	504	NA
JonesMonoid(5)	42	113	746	777	NA
MotzkinMonoid(3)	51	98	1 972	2 400	NA
PartialTransformationMonoid(3)	64	138	2 586	1 963	NA
PartialBrauerMonoid(3)	76	165	7 897	7 796	NA
BrauerMonoid(4)	105	274	50 164	19 875	NA
SymmetricGroup(5)	120	146	32 861	2 979	201
PlanarPartitionMonoid(3)	132	393	95 460	25 803	NA
JonesMonoid(6)	132	393	131 472	22 889	NA
PartitionMonoid(3)	203	687	741 441	105 421	NA
SymmetricInverseMonoid(4)	209	282	470 901	75 940	NA

In this test the classic algorithm not using congruences is faster than using congruences for algebras up to size 60 but for larger algebras its runtime raises very fast and the algorithm using congruences becomes faster.

In view of these results, the **CreamEndomorphisms** function uses the classic algorithm for algebras with size up to 60 and the congruences algorithm for larger algebras.

The SONATA package achieves very good results for groups but it only works on a very narrow set of algebras.

5. Conclusion

The CREAM package provides efficient algorithms for the calculation of congruences, automorphisms and endomorphisms. These algorithms work with algebras of type $(2^m, 1^n)$, covering a very wide set of algebras and being consistently fast.

The integration with Mace4 allowed us to combine efficient algorithms such as [28] with the wide set of possibilities provided by a first order axiom-based model searcher.

The appendices of this article include a detailed performance discussion of the main algorithms and applications of the CREAM package to different algebra classes.

The CREAM package can be found and downloaded from `https://gitlab.com/rmbper/cream` and the outputs from the test runs described in section B can be found in `https://gitlab.com/rmbper/cream_dat`

Appendices

A. Detailed Performance

A.1. **Congruences algorithm performance.** The **semigroup** GAP package [45] includes the function **CongruencesOfSemigroup** that returns the congruences of a semigroup. This function doesn't work for all semigroups, but for those in which it works we have run the **CongruencesOfSemigroup** function against our **CreamAllCongruences** 5 times and calculated the average runtime after removing the best and worst runtimes getting the following results:

Table 4. Performance Comparison between **CongruencesOfSemigroup** and CREAM on calculating all congruences.

Size	Number of Algebras	CongruencesOfSemigroup(ms)	CreamAllCongruences(ms)
1	1	7	< 1
2	3	20	< 1
3	4	29	< 1
4	7	41	< 1
5	9	53	< 1
6	8	71	< 1
7	12	153	4
8	14	484	35
1-8	58	858	40

The number of semigroups for which **CongruencesOfSemigroup** works is very limited, yet even for those algebras **CreamAllCongruences** is more than 20 times faster.

The tests show that there is a very specific type of semigroup in which the function **CongruencesOf-Semigroup** takes advantage of theoretical results and hence achieves better results than **CreamAllCongru-ences**. These are the Rees Matrix Semigroups or the Rees Zero Matrix Semigroups. But to achieve this kind of performance the semigroup needs to be created using the GAP functions **ReesMatrixSemigroup** or **ReesZeroMatrixSemigroup**. If instead an isomorphic copy of a Rees Matrix (Zero) semigroup is created using a multiplication table the performance will be similar to what was found above. See in the following table the average runtime for 10 runs with random Rees Matrix Semigroups: For Rees Matrix Semigroups or Rees Zero Matrix Semigroups (generated with **ReesMatrixSemigroup** or **ReesZeroMatrixSemigroup**) the function **CongruencesOfSemigroup** takes advantage of their particular structure by applying the efficient linked triple algorithm [35]. However, this efficiency can only be realized in this restricted setting.

UACalc also implements Freese's algorithm, and was done under his supervision in Java and Jython. It includes a GUI version in Java and a command line interface in Jython. The performance of the GUI version is poor and it will not be used for comparison. On the other hand the command line interface has a performance that is comparable to **CreamAllCongruences**. The same random Rees Matrix Semigroups that were used with **CongruencesOfSemigroup** and **CreamAllCongruences** were written into files readable by **UACalc** and the congruences were calculated in **UACalc**. **CreamAllCongruences** is consistently more than 3 times faster than **UACalc**. For these algebras with one binary operation the runtime rises roughly proportionally to n^4 (where n is the size of the algebra) or t^2 (being t the number cells of the multiplication matrix of the algebra operation).

To further compare the performance of **CreamAllCongruences** with **UACalc**, tests with several specific types of Semigroups, Monoids and Groups were run.

Table 5. Performance Comparison on calculating all congruences of Rees Matrix Semigroup.

Size	CongruencesOfSemigroup (ReesMatrixSemigroup) (ms)	CongruencesOfSemigroup (MultiplicationTable) (ms)	CreamAllCongruences (ms)	UACalc (ms)
12	1	15	< 1	1
18	2	28	< 1	3
24	5	55	2	7
30	1	92	4	32
36	4	139	10	34
42	11	203	16	55
48	8	296	24	77
54	5	416	46	138
60	5	553	60	220
66	1	671	82	291
72	9	873	136	409
78	2	1 098	180	555
84	5	1 341	233	743
90	4	1 622	307	961

Table 6. Performance Comparison between CREAM and UACalc

Algebra	Size	Number of Congruences	CreamAllCongruences (ms)	UACalc (ms)
GossipMonoid(3)	11	84	1	2
PlanarPartitionMonoid(2)	14	9	1	1
JonesMonoid(4)	14	9	< 1	1
BrauerMonoid(3)	15	7	< 1	2
PartitionMonoid(2)	15	13	< 1	1
FullPBRMonoid(1)	16	167	3	4
SymmetricGroup(4)	24	4	3	10
FullTransformationSemigroup(3)	27	7	3	14
FullTransformationMonoid(3)	27	7	3	7
SymmetricInverseMonoid(3)	34	7	6	25
JonesMonoid(5)	42	6	12	36
MotzkinMonoid(3)	51	10	23	73
PartialTransformationMonoid(3)	64	7	63	223
PartialBrauerMonoid(3)	76	16	117	351
BrauerMonoid(4)	105	19	430	1 420
SymmetricGroup(5)	120	3	812	2 700
PlanarPartitionMonoid(3)	132	10	1 094	3 490
JonesMonoid(6)	132	10	950	3 444
PartitionMonoid(3)	203	16	6 500	19 979
SymmetricInverseMonoid(4)	209	11	6 970	22 318
FullTransformationSemigroup(4)	256	11	22 529	53 557
FullTransformationMonoid(4)	256	11	22 385	52 361
MotzkinMonoid(4)	323	11	47 593	134 945
JonesMonoid(7)	429	7	161 987	485 004

The performance for these algebras is mostly similar to what was found for the Rees Matrix Semigroups, with a bigger variance in the results. On average **CreamAllCongruences** is between 3 and 3,5 times faster than **UACalc** but, depending on the algebras, we get improvements spanning from 2 times to 4,5 times. The runtime also rises roughly proportionally to n^4 (although with more variance).

A.2. **Automorphisms algorithm performance.** The CREAM package automorphism function was run on algebras of Type (2^m,1^n) to compare the timings against the Loops package. All experiments were run 7 times, with the lowest and highest times discarded. The averages of the remaining 5 are reported in Table 7. We see that the speeds from both packages are quite close.

Table 7. Performance Comparison between Loops and CREAM on Automorphism Group Generation.

Algebraic Structure	Loops Package(sec)	CREAM Package(sec)
Quasigroups, order 5	0.588	0.562
Quasigroups, order 6	520	541
Loops, order 6	0.101	0.086
Loops, order 7	14.623	13.945
Groups, order 32	0.854	0.935
Groups, order 64	39.98	41.05
Groups, order 128	12,031	12,160

We have also run experiments on generating automorphism groups for magmas and semigroups. Results are displayed on Table 8. For these experiments, we do not have results from other GAP packages to compare with. Therefore the following results can be taken as the benchmark to beat in future improvements.

Table 8. Performance of CREAM on Automorphism Group Generation for Magmas and Semigroups.

Algebraic Structure	Time(sec)
Magmas, order 2	0.016
Magmas, order 3	1.253
Semigroups, order 6	2.316
Semigroups, order 7	45.947

The CREAMAutomorphisms runs substantially faster than the simple intersection of automorphism groups, especially when the automorphism group is the trivial group. Sample results are shown in Table 9.

Table 9. Performance Comparison between CREAMAutomorphisms and Intersection of Automorphism Groups.

Algebra of type (2, 2, 2, ...)	Algebra AutomorphismGroup (sec)	Intersection of Automorphism Groups (sec)	Is Trival Group the Automorphism Group?
All 2,328 non-isomorphic groups of order 128	153.8	201.5	No
All 67 non-isomorphic groups of order 243	10.3	56.3	No
All 15 non-isomorphic groups of order 625	11.3	122.9	No
All 15 non-isomorphic groups of order 999	2.1	168.9	Yes

A.3. **Endomorphisms algorithm performance.** To evaluate the performance of the algorithms to calculate endomorphisms, we tested both the classic algorithm and the congruence algorithm running with **Mace4**, as described in 3.3, on several types of Semigroups, Monoids and Groups. The only GAP function to calculate endomorphisms that came to our attention is the function Endomorphisms from the package SONATA. This function can calculate endomorphisms for a very narrow set of algebras, namely groups and near-rings. The groups in the list algebras were also run with SONATA.

Table 10. Performance Comparison between endomorphisms calculation algorithms

Algebra	Size	Number of Endomorphisms	Classic (ms)	Congruences (ms)	SONATA (ms)
GossipMonoid(3)	11	66	30	1 635	NA
PlanarPartitionMonoid(2)	14	72	56	233	NA
JonesMonoid(4)	14	72	45	232	NA
BrauerMonoid(3)	15	28	38	160	NA
PartitionMonoid(2)	15	89	50	313	NA
FullPBRMonoid(1)	16	1 426	585	5 134	NA
SymmetricGroup(4)	24	58	101	142	166
FullTransformationSemigroup(3)	27	40	116	364	NA
FullTransformationMonoid(3)	27	40	112	360	NA
SymmetricInverseMonoid(3)	34	54	274	504	NA
JonesMonoid(5)	42	113	746	777	NA
MotzkinMonoid(3)	51	98	1 972	2 400	NA
PartialTransformationMonoid(3)	64	138	2 586	1 963	NA
PartialBrauerMonoid(3)	76	165	7 897	7 796	NA
BrauerMonoid(4)	105	274	50 164	19 875	NA
SymmetricGroup(5)	120	146	32 861	2 979	201
PlanarPartitionMonoid(3)	132	393	95 460	25 803	NA
JonesMonoid(6)	132	393	131 472	22 889	NA
PartitionMonoid(3)	203	687	741 441	105 421	NA
SymmetricInverseMonoid(4)	209	282	470 901	75 940	NA

In this test the classic algorithm not using congruences is faster than using congruences for algebras up to size 60 but for larger algebras its runtime raises very fast and the algorithm using congruences becomes faster.

In view of these results, the **CreamEndomorphisms** function uses the classic algorithm for algebras with size up to 60 and the congruences algorithm for larger algebras.

B. Applications

B.1. **Monolithic Algebras.** Some examples of the use of the package to calculate whether an algebra is monolithic can be seen next:

```
gap> algebra := [RecoverMultiplicationTable(6,1)];
[ [ [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ],
      [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ] ] ]
gap> CreamAllPrincipalCongruences(algebra);
[ [ -1, -1, -1, -1, -2, 5 ], [ -1, -1, -1, -2, 4, -1 ],
  [ -1, -1, -1, -2, -1, 4 ], [ -1, -1, -2, 3, -1, -1 ],
  [ -1, -1, -2, -1, 3, -1 ], [ -1, -1, -2, -1, -1, 3 ],
  [ -1, -2, 2, -1, -1, -1 ], [ -1, -2, -1, 2, -1, -1 ],
  [ -1, -2, -1, -1, 2, -1 ], [ -1, -2, -1, -1, -1, 2 ],
  [ -2, 1, -1, -1, -1, -1 ], [ -2, -1, 1, -1, -1, -1 ],
  [ -2, -1, -1, 1, -1, -1 ], [ -2, -1, -1, -1, 1, -1 ],
  [ -2, -1, -1, -1, -1, 1 ] ]
gap> CreamIsAlgebraMonolithic(algebra);
false
gap> algebra := [RecoverMultiplicationTable(6,19)];
[ [ [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ],
      [ 1, 1, 1, 1, 1, 1 ], [ 1, 1, 1, 2, 1, 1 ], [ 1, 1, 2, 1, 1, 1 ] ] ]
gap> CreamAllPrincipalCongruences(algebra);
[ [ -2, 1, -1, -1, -1, -1 ], [ -2, 1, -1, -1, -2, 5 ],
  [ -2, 1, -1, -2, 4, -1 ], [ -2, 1, -1, -2, -1, 4 ],
  [ -2, 1, -2, 3, -1, -1 ], [ -2, 1, -2, -1, 3, -1 ],
  [ -2, 1, -2, -1, -1, 3 ], [ -3, 1, 1, -1, -1, -1 ],
  [ -3, 1, -1, 1, -1, -1 ], [ -3, 1, -1, -1, 1, -1 ],
  [ -3, 1, -1, -1, -1, 1 ] ]
gap> CreamIsAlgebraMonolithic(algebra);
true
```

B.2. **Small Semigroups.** For all small semigroups up to size 6, all its congruences and endomorphisms were calculated. It was moreover determined whether the semigroups were monolithic (this was calculated up to size 8).

Table 11. Determination of all monolithic semigroups up to size 8

Size	Number of semigroups	Number of monoliths	Performance (ms)
1	1	0	0
2	4	4	0
3	18	7	0
4	126	16	8
5	1 160	103	108
6	15 973	1 823	1 712
7	836 021	149 020	127 356
8	1 843 120 128	48 438 046	462 897 348

All automorphisms for semigroups up to size 7 were also calculated separately and the automorphism group and its ID was calculated for each of these semigroups. The group ID is the identification of a group in the Small Group GAP library [20].

The CREAM library doesn't calculate the automorphisms of an algebra as an automorphism group but as a list of bijective mappings, but there is an easy way to calculate the automorphism group from the list of mappings. To do this, all the mappings are converted into permutations using the **PermList** function and each of these permutations is used as generator for the group. Having this we can get the Group Id using the **IdGroup** function.

The following table gives the automorphism of semigroups up to size 7 (and reproduces the results in [9]):

Table 12. Computation of the automorphism groups of semigroups up to size 7

Size	2	3	4	5	6	7
# Semigroups	4	18	126	1 160	15 973	836 021
[1, 1] trivial	3	12	78	746	10 965	746 277
[2, 1] C_2	1	5	39	342	4 121	76 704
[3, 1] C_3				2	26	412
[4, 1] C_4				1	7	82
[4, 2] $C_2 \times C_2$			3	26	441	7 314
[5, 1] C_5						6
[6, 1] S_3		1	5	33	300	3 638
[6, 2] C_6						37
[8, 2] $C_4 \times C_2$						4
[8, 3] D_8				1	17	169
[8, 5] $C_2 \times C_2 \times C_2$					6	172
[10, 1] D_{10}						2
[12, 4] D_{12}				4	49	790
[16, 11] $C_2 \times D_8$						10
[24, 12] S_4			1	4	30	277
[24, 14] $C_2 \times C_2 \times S_3$						14
[36, 10] $S_3 \times S_3$					2	24
[48, 48] $C_2 \times S_4$					4	45
[72, 40] $(S_3 \times S_3) \wr C_2$						1
[120, 34] S_5				1	4	30
[144, 183] $S_3 \times S_4$						4
[240, 189] $C_2 \times S_5$						4
[720, 763] S_6					1	4
[5040, -] S_7						1

B.3. **Small Groups.** Using the Small Group GAP library [20] the groups from order 2 to 96 were extracted and the Cream functions were used with them:

- Order 2-31 - CreamAllCongruences, CreamAllEndomorphisms and CreamIsMonolithic
- Order 32-96 - CreamAllCongruences and CreamIsMonolithic

A summary from these runs is provided in Table 13.

Table 13. Summary for Small Group Runs

Size	**Number of Groups**	**Number of Monolithic**
2-7	8	6
8-15	19	9
16	14	6
17-23	17	7
24	15	2
25-31	19	7
32	50	16
33-47	54	10
48	52	4
49-63	69	16
64	52	22
65-71	17	3
72	50	3
73-79	18	5
80	52	1
81-95	70	13
96	231	13

B.4. **Groups, Semigroups and Monoids.** For a selection of larger groups, semigroups and monoids, all congruences and endomorphisms were calculated. It was also calculated whether these algebras were monolithic. A summary from these runs is provided in Table 14.

Table 14. # of Congruences and Endomorphisms for a selection of larger algebras

Algebra	**Size**	**Number of Congruences**	**Number of Endomorphisms**	**Is Monolithic**
GossipMonoid(3)	11	84	66	No
PlanarPartitionMonoid(2)	14	9	72	No
JonesMonoid(4)	14	9	72	No
BrauerMonoid(3)	15	7	28	No
PartitionMonoid(2)	15	13	89	No
FullPBRMonoid(1)	16	167	1426	No
SymmetricGroup(4)	24	4	58	Yes
FullTransformationSemigroup(3)	27	7	40	Yes
FullTransformationMonoid(3)	27	7	40	Yes
SymmetricInverseMonoid(3)	34	7	54	Yes
JonesMonoid(5)	42	6	113	No
MotzkinMonoid(3)	51	10	98	No
PartialTransformationMonoid(3)	64	7	138	Yes
PartialBrauerMonoid(3)	76	16	165	No
BrauerMonoid(4)	105	19	274	No
SymmetricGroup(5)	120	3	146	Yes
PlanarPartitionMonoid(3)	132	10	393	No
JonesMonoid(6)	132	10	393	No
PartitionMonoid(3)	203	16	687	No
SymmetricInverseMonoid(4)	209	11	282	Yes
FullTransformationSemigroup(4)	256	11	345	Yes
FullTransformationMonoid(4)	256	11	345	Yes

B.5. **Automorphisms of Specific Algebra Classes.** The following table contains the groups that appear as automorphims groups of associative rings up to order 15.

Table 15. Computation of the automorphism groups of rings up to size 15

Size	2	3	4	5	6	7	8	9	10	11	12	13	14	15
# Rings	2	2	11	2	4	2	52	11	4	2	22	2	4	4
[1, 1] trivial	2	1	3		2		4				3			
[2, 1] C_2		1	6	1		1	12	1	2		4		2	1
[4, 1] C_4										1				
[4, 2] $C_2 \times C_2$					1		6	3				1		
[6, 1] S_3			2				1							
[6, 2] C_6							1							
[8, 3] D_8							3							
[8, 5] $C_2 \times C_2 \times C_2$							4	1			2			
[12, 4] D_{12}								2						
[16, 11] $C_2 \times D_8$							3							
[16, 14] $C_2 \times C_2 \times C_2 \times C_2$											2			
[24, 12] S_4					1									
[24, 13] $C_2 \times A_4 \times S_3$							2							
[32, 46] $C_2 \times C_2 \times D_8$											4			
[36, 10] $S_3 \times S_3$							4							
[48, 48] $C_2 \times S_4$							7							
[64, 261] $C_2 \times C_2 \times C_2 \times D_8$														1
[72, 40] $(S_3 \times S_3) \wr C_2$								2						
[120, 34] S_5					1									
[144, 183] $S_3 \times S_4$							2							
[216, 162] $S_3 \times S_3 \times S_3$											2			
[576, 8653] $S_4 \times S_4$									1					
[720, 763] S_6						1								
[5040, -] S_7							3							
[13824, -] $S_4 \times S_4 \times S_4$														1
[14400, -] $S_5 \times S_5$											1			
[17280, -] $S_6 \times S_4$											2			
[40320, -] S_8								2						
[362880, -] S_{10}									1	1				
[518400, -] $S_6 \times S_6$													1	
[39916800, -] S_{11}											2			
[479001600, -] S_{12}												1		
[6227020800, -] S_{13}													1	
[87178291200, -] S_{14}														1

The variety of *Quasi-MV-algebras* [37] is defined by the following identities (in ProverX syntax [13,17]):

```
(x + z) + y = x + (y + z).
x'' = x.
x + 1 = 1.
(x' + y)' + y = (y' + x)' + x.
(x + 0)' = x' + 0.
(x + y) + 0 = x + y.
0' = 1.
```

Table 16. Computation of the automorphism groups of Quasi-MV-algebras of size up to 12

Size	2	3	4	5	6	7	8	9	10	11	12
# Quasi-MV-algebras	1	1	4	4	11	11	27	27	60	62	131
[1, 1] trivial	1	1	3	2	7	4	16	8	35	17	76
[2, 1] C_2			1	2	3	4	5	9	10	18	19
[4, 2] $C_2 \times C_2$						2	1	4	2	10	5
[6, 1] S_3					1		3		5		9
[8, 5] $C_2 \times C_2 \times C_2$								2	1	4	2
[12, 4] D_{12}							1		2		5
[16, 14] $C_2 \times C_2 \times C_2 \times C_2$										2	1
[24, 12] S_4						1		2		4	
[24, 14] $C_2 \times C_2 \times C_3$									1		2
[48, 48] $C_2 \times S_4$								1		2	
[48, 51] $C_2 \times C_2 \times C_2 \times S_3$											1
[96, 226] $C_2 \times C_2 \times S_4$										1	
[120, 34] S_5							1	1	2		4
[240, 189] $C_2 \times S_5$									1		2
[480, 1186] $C_2 \times C_2 \times S_5$											1
[720, 763] S_6										2	
[1440, 5842] $C_2 \times S_6$										1	
[5040, -] S_7									1		2
[10080, -] $C_2 \times S_7$											1
[40320, -] S_8										1	
[362880, -] S_9											1

The quasivariety of *BCI algebras* [2] is defined by the following identities (in ProverX syntax [13, 17]):

```
((x * y) * (x * z)) * (z * y) = 0.
(x * (x * y)) * y = 0.
x * x = 0.
((x * y = 0) & (y * x = 0)) -> (x = y).
(x * 0 = 0) -> (x = 0).
```

Table 17. Computation of the automorphism groups of BCI algebras of size up to 7

Size	2	3	4	5	6	7
# BCI algebras	2	5	22	118	974	10,834
[1, 1] trivial	2	3	13	78	679	7,970
[2, 1] C_2		2	7	31	241	2,384
[4, 1] C_4				1	1	2
[4, 2] $C_2 \times C_2$				2	20	207
[6, 1] S_3			2	5	25	195
[6, 2] C_6						1
[8, 2] $C_4 \times C_2$						1
[8, 3] D_8						4
[8, 5] $C_2 \times C_2 \times C_2$						3
[12, 4] D_{12}					4	35
[24, 12] S_4				1	4	22
[36, 10] $S_3 \times S_3$						2
[48, 48] $C_2 \times S_4$						3
[120, 34] S_5					1	4
[720, 763] S_6						1

The variety of *quandles* [43] is defined by the following identities (in ProverX syntax [13, 17]):

```
x v (y v z) = (x v y) v (x v z).
(x ^ y) ^ z = (x ^ z) ^ (y ^ z).
(x v y) ^ x = y.
x v (y ^ x) = y.
x v x = x.
```

Table 18. Computation of the automorphism groups of quandles of size up to 6

Size	2	3	4	5	6
# quandles	1	3	7	22	73
[2, 1] C_2	1	1	1		
[3, 1] C_3			1	1	
[4, 1] C_4				1	2
[4, 2] $C_2 \times C_2$			1	5	8
[5, 1] C_5					1
[6, 1] S_3		2	1	2	3
[6, 2] C_6				2	15
[8, 2] $C_4 \times C_2$					2
[8, 3] D_8			1	3	8
[8, 5] $C_2 \times C_2 \times C_2$					7
[12, 3] A_4			1	1	
[12, 4] D_{12}				3	6
[16, 11] $C_2 \times D_8$					5
[18, 3] $C_3 \times S_3$					3
[20, 3] $C_5 \times C_4$				3	3
[24, 12] S_4			1		2
[24, 13] $C_2 \times A_4 \times S_3$					3
[36, 10] $S_3 \times S_3$					1
[48, 48] $C_2 \times S_4$					2
[72, 40] $(S_3 \times S_3) \wr C_2$					1
[120, 34] S_5				1	
[720, 763] S_6					1

B.6. **Algebra Classes.** The CREAM package allows for the generation of lists of algebras of specific types based on its axiomatic definition. This is done using the function **CreamAlgebrasFromAxioms** that uses Mace4 to generate algebras according with this axiomatic definition. This function takes as input a string with the axioms in Mace4 format and a positive integer with the size of the generated algebras. The generated algebras can then be used with the CREAM functions that calculate congruences, endomorphisms and whether the algebra is monolithic. CREAM was applied in this way to the following classes of algebras:

- Almost distributive lattices
- BCI-algebras
- Bilattices
- Boolean algebras
- Boolean rings
- Commutative lattice-ordered monoids
- Commutative regular rings
- Complemented distributive lattices
- Complemented modular lattices
- Distributive lattice ordered semigroups
- Ockham algebras
- Ortholattices
- Orthomodular lattices
- Idempotent semirings
- Extra loops
- Digroups
- Commutative dimonoids

A summary from these runs is provided in Table 19.

Table 19. Summary for Algebra Class Runs

Algebra Class	Size	Algebra Type	Number of Algebras	Number of Monolithic
Almost distributive lattices	4	(2^2)	2	0
BCI-algebras	5	(2^1)	118	80
Bilattice	6	$(2^4, 1^1)$	32	32
Boolean algebra	8	$(2^2, 1^1)$	1	0
Boolean algebra	16	$(2^2, 1^1)$	1	0
Boolean ring	16	(2^2)	1	0
Commutative lattice-ordered monoids	5	(2^3)	199	97
Commutative regular rings	6	$(2^2, 1^2)$	72	66
Complemented distributive lattices	16	$(2^2, 1^1)$	1	0
Complemented distributive lattices	32	$(2^2, 1^1)$	1	0
Complemented modular lattices	8	$(2^2, 1^1)$	41	40
Distributive lattice ordered semigroups	4	(2^3)	479	170
Ockham algebras	6	$(2^2, 1^1)$	197	20
Ortholattices	7	$(2^2, 1^1)$	46	12
Orthomodular lattices	14	$(2^2, 1^1)$	33	31
Idempotent semiring	5	(2^2)	149	42
Extra loop	10	(2^3)	2	1
Digroup	8	$(2^2, 1^1)$	10	3
Commutative dimonoid	4	(2^2)	101	37

REFERENCES

[1] J. André, J. Araújo, P. J. Cameron, The classification of partition homogeneous groups with applications to semigroup theory. *J. Algebra* **452** (2016), 288–310.

[2] Y. Arai, K. Iséki and S. Tanaka, Characterizations of BCI, BCK-algebras, *Proc. Japan Acad.* (1966), **42** (2), 105–107.

[3] J. Araújo, J.P. Araújo, W. Bentz, P. J. Cameron, P. Spiga, A transversal property for permutation groups motivated by partial transformations. *J. Algebra* **573** (2021), 741–759.

[4] J. Araújo, W. Bentz, P. J. Cameron, The existential transversal property: a generalization of homogeneity and its impact on semigroups. *Trans. Amer. Math. Soc.* **374** (2021), no. 2, 1155–1195.

[5] J. Araújo, W. Bentz, P. J. Cameron, Primitive permutation groups and strongly factorizable transformation semigroups. *J. Algebra* **565** (2021), 513–530.

[6] J. Araújo, W. Bentz, P. J. Cameron, Orbits of primitive k-homogenous groups on (n?k)-partitions with applications to semigroups. *Trans. Amer. Math. Soc. 371* (2019), no. 1, 105–136.

[7] J. Araújo, W. Bentz, P. J. Cameron, G. Royle, A. Schaefer, Primitive groups, graph endomorphisms and synchronization. *Proc. Lond. Math. Soc.* (3) **113** (2016), no. 6, 829–867.

[8] J. Araújo, W. Bentz, E. Dobson, J. Konieczny, J. Morris, Automorphism groups of circulant digraphs with applications to semigroup theory. *Combinatorica* **38** (2018), no. 1, 1–28.

[9] J. Araújo, P.V. Bünau, J.D. Mitchell, M. Neunhöffer, Computing automorphisms of semigroups, J. Symbolic Comput., 45 (3) (2010), pp. 373-392

[10] J. Araújo, P. J. Cameron, B. Steinberg, Between primitive and 2-transitive: synchronization and its friends. *EMS Surv. Math. Sci.* **4** (2017), no. 2, 101–184.

[11] J. Araújo, P. J. Cameron, Two generalizations of homogeneity in groups with applications to regular semigroups. *Trans. Amer. Math. Soc.* **368** (2016), no. 2, 1159–1188.

[12] J. Araújo, P. J. Cameron, Primitive groups synchronize non-uniform maps of extreme ranks. *J. Combin. Theory Ser. B* **106** (2014), 98–114.

[13] J. Araújo, M. Kinyon and Yves Robert, Varieties of regular semigroups with uniquely defined inversion. Port. Math. 76 (2019), 205–228.

[14] J. Araújo and J. Konieczny, Automorphism groups of centralizers of idempotents, *J. Algebra* **269** (2003), 227–239.

[15] J. Araújo, and J. Konieczny, A Method of Finding Automorphism Groups of Endomorphism Monoids of Relational Systems, *Discrete Math.* **307** (2007) 1609–1620.

[16] J. Araújo and J. Konieczny, Automorphisms of Endomorphism Monoids of Relatively Free Bands, *Proc. Edinburgh Math. Soc.* **50** (2007) 1–21

[17] J. Araújo and Yves Robert, The System ProverX, www.proverx.com.

[18] E. Artin, "Geometric Algebra," Interscience, New York, 1957.

[19] M. V. Berlinkov, R. Ferens, M. Szykua, Preimage problems for deterministic finite automata. *J. Comput. System Sci.* **115** (2021), 214–234.

[20] Hans Ulrich Besche, Bettina Eick & Eamonn O'Brien, *GAP package SmallGrp - The GAP Small Groups Library, Version 1.4.2*; 2020, `https://www.gap-system.org/Packages/smallgrp.html`.

[21] Stanley N. Burris & H.P. Sankappanavar, *A Course in Universal Algebra*; 1981, `http://www.math.uwaterloo.ca/~snburris/htdocs/UALG/univ-algebra2012.pdf`.

[22] P.J. Cameron, Automorphism groups of graphs, "Selected Topics in Graph Theory. 2," 89–127, Academic Press, London, 1983.

[23] J. J. Cannon and D. Holt, Automorphism group computation and isomorphism testing in finite groups, *J. Symbolic Comput.* **35** (2003), 241–267.

[24] David Clark and Brian Davey, *Natural dualities for the working algebraist*, Cambridge Studies in Advanced Mathematics 57, 1998

[25] Andreas Distler & James Mitchel, *GAP package Smallsemi - A library of small semigroups, Version 0.6.12*; 2019, `https://www.gap-system.org/Packages/smallsemi.html`.

[26] S. P. Fitzpatrick and J. S. Symons, Automorphisms of transformation semigroups, *Proc. Edinburgh Math. Soc.* **19** (1974/75), 327-329.

[27] Ralph Freese, *UACalc, A Universal Algebra Calculator*; 2015, `http://www.uacalc.org/`.

[28] Ralph Freese, Computing congruences efficiently, *Algebra Universalis* **59(3)** (2008), 337–343. DOI:

[29] Ralph Freese, and Ralph McKenzie, *Commutator Theory for congruence modular varieties*, London Mathematical Society Lecture Notes, Vol. 125, 1987

[30] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.11.0*; 2020, `https://www.gap-system.org`.

[31] O. Garcia and Walter Taylor, *The lattice of interpretability types of varieties*, Memoirs of the American Mathematical Society, Vol. 50, no. 305, 1984

[32] L. M. Gluskĭn, Semigroups and rings of endomorphisms of linear spaces I, *Amer. Math. Soc. Transl.* **45** (1965), 105–137.

[33] J. Hagemmann and C. Herrmann, A concrete ideal multiplication for algebraic systems and its relation to congruence distributivity, *Arch. Math.* **32**, 234-245, 1979

[34] David Hobby and Ralph McKenzie, *The structure of finite algebras*, Contemporary Mathematics, Vol. 76, 1988

[35] J. M. Howie, *Fundamentals of Semigroup Theory*, Oxford University Press, New York, 1995.

[36] F. Klein, "Vergleichende Betrachtungen über neuere geometrische Forschungen," Andreas Diechert, Erlanger, 1872.

[37] A. Ledda, M. Konig, F. Paoli, and R. Giuntini. MV-algebras and quantum computation. *Studia Logica* (2006) **82** (2):245–270.

[38] I. Levi, Automorphisms of normal transformation semigroups, *Proc. Edinburgh Math. Soc. (2)* **28** (1985), 185–205.

[39] I. Levi, Automorphisms of normal partial transformation semigroups, *Glasgow Math. J.* **29** (1987), 149–157.

[40] A. E. Liber, On symmetric generalized groups, *Mat. Sbornik N. S.* **33** (1953), 531–544. (Russian)

[41] K. D. Magill, Semigroup structures for families of functions, I. Some homomorphism theorems, *J. Austral. Math. Soc.* **7** (1967), 81–94.

[42] A. I. Mal'cev, Symmetric groupoids, *Mat. Sbornik N.S.* **31** (1952), 136–151. (Russian)

[43] S. Matveev, Distributive groupoids in knot theory. *Math. USSR Sbornik* (1984) **47**, 73–83.

[44] W. McCune, *Mace4, Version 2009-02A*; 2009 `https://www.cs.unm.edu/~mccune/prover9/manual/2009-02A/mace4.html`.

[45] J. D. Mitchell and others, Semigroups - GAP package, Version 2.6, (2015). `https://gap-packages.github.io/Semigroups/`

[46] V. A. Molchanov, Semigroups of mappings on graphs, *Semigroup Forum* **27** (1983), 155–199.

[47] A. V. Molchanov, On definability of hypergraphs by their semigroups of homomorphisms, *Semigroup Forum* **62** (2001), 53–65.

[48] Gábor Nagy & Petr Vojtěchovský, *The LOOPS Package - a GAP package, Version 3.4.1*; 2018, `https://gap-packages.github.io/loops/`.

[49] P. M. Neumann, Primitive permutation groups and their section-regular partitions, *Michigan Math. J.*, **58** (2009), 309–322.

[50] B. M. Schein, Ordered sets, semilattices, distributive lattices and Boolean algebras with homomorphic endomorphism semigroups, *Fund. Math.* **68** (1970), 31–50.

[51] C. Schneider and A. C. Silva, Cliques and colorings in generalized Paley graphs and an approach to synchronization, *J. Algebra Appl.*, **14** (2015), no. 6, 13 pp.

[52] J. Schreier, Über Abbildungen einer abstrakten Menge Auf ihre Teilmengen, *Fund. Math.* **28** (1936), 261–264.

[53] J. Rhodes and B. Steinberg, The q-theory of finite semigroups. Springer Verlag. (2008).

[54] Shahzamanian, B. Steinberg, Simplicity of augmentation submodules for transformation monoids. *Algebr. Represent. Theory* **24** (2021), no. 4, 1029–1051.

[55] Smith, JDH, *Mal'cev varieties*, Lecture Notes in Mathematics, Vol. 554, 1976

[56] L. Soicher, *GRAPE: Graph algorithms using permutation groups*; version 4.2, `http://www.maths.qmul.ac.uk/~leonard/grape/`.

[57] M. R. Sorouhesh, On ideals of quasi-commutative semigroups. *Bull. Iranian Math. Soc.* **45** (2019), no. 2, 447–453.

[58] R. P. Sullivan, Automorphisms of transformation semigroups, *J. Austral. Math. Soc.* **20** (1975), 77–84.

[59] È. G. Šutov, Homomorphisms of the semigroup of all partial transformations, *Izv. Vysš. Učebn. Zaved. Matematika* **3** (1961), 177–184. (Russian)

[60] S.M. Ulam, "A Collection of Mathematical Problems," Interscience Publishers, New York, 1960.

Bibliography

[1] Aichinger, E., Binder, F., Ecker, J., Mayr, P. and Nöbauer, *SONATA, System of near-rings and their applications, Version 2.9.1*; 2018, `https://gap-packages.github.io/sonata/`.

[2] J. Araújo and J. Konieczny, *Automorphisms of Endomorphism Monoids of Relatively Free Bands*, Proc. Edinburgh Math. Soc. 50 (2007) 1–21

[3] J. Araújo and J. Konieczny, *Automorphism groups of centralizers of idempotents*, J. Algebra 269 (2003), 227–239.

[4] J. Araújo, and J. Konieczny, *A Method of Finding Automorphism Groups of Endomorphism Monoids of Relational Systems*, Discrete Math. 307 (2007) 1609–1620.

[5] João Araújo, Rui Barradas Pereira, Wolfram Bentz, Choiwah Chow, João Ramires, Luís Sequeira, Carlos Sousa, *CREAM: a Package to Compute [Auto, Endo, Iso, Mono, Epi]-morphisms, Congruences, Divisors and More for Algebras of Type* $(2^n, 1^n)$, 2022.

[6] E. Artin, *Geometric Algebra*, Interscience, New York, 1957.

[7] Hans Ulrich Besche, Bettina Eick & Eamonn O'Brien, *GAP package SmallGrp - The GAP Small Groups Library, Version 1.4.2*; 2020, `https://www.gap-system.org/Packages/smallgrp.html`.

[8] Stanley N. Burris & H.P. Sankappanavar, *A Course in Universal Algebra of Graduate Texts in Mathematics*; Springer-Verlag, Berlin, 1981.

[9] P.J. Cameron, *Automorphism groups of graphs*, Selected Topics in Graph Theory. 2, 89–127, Academic Press, London, 1983.

[10] J. J. Cannon and D. Holt, *Automorphism group computation and isomorphism testing in finite groups*, J. Symbolic Comput. 35 (2003), 241–267.

[11] J. Demel, M. Demlová, V. Koubek *Fast Algorithms Constructing Minimal Subalgebras, Congruences and Ideals in a Finite Algebra,*; 1983, Theoretical Computer Science 36 (1985), 203–216.

[12] William DeMeo & Ralph Freese, *A command line and scripting version of UACalc using Jython*; 2014, `http://www.uacalc.org/Jython/`.

[13] Andreas Distler & James Mitchel, *GAP package Smallsemi - A library of small semigroups, Version 0.6.12*; 2019, `https://www.gap-system.org/Packages/smallsemi.html`.

[14] S. P. Fitzpatrick and J. S. Symons, *Automorphisms of transformation semigroups*, Proc. Edinburgh Math. Soc. 19 (1974/75), 327-329.

[15] Ralph Freese, *Computing congruences efficiently*; Algebra universalis 59, no. 3-4 (November 2008), 337–43.

[16] Ralph Freese, *UACalc, A Universal Algebra Calculator*; 2015, `http://www.uacalc.org/`.

[17] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.11.0*; 2020, `https://www.gap-system.org`.

[18] L. M. Gluskĭn, *Semigroups and rings of endomorphisms of linear spaces I*, Amer. Math. Soc. Transl. 45 (1965), 105–137

[19] David Hobby and Ralph McKenzie, *The structure of finite algebras*, Contemporary Mathematics, Vol. 76, 1988

[20] I. Levi, *Automorphisms of normal partial transformation semigroups*, Glasgow Math. J. 29 (1987), 149–157.

[21] I. Levi, *Automorphisms of normal transformation semigroups*, Proc. Edinburgh Math. Soc. (2) 28 (1985), 185–205.

[22] A. E. Liber, *On symmetric generalized groups*, Mat. Sbornik N. S. 33 (1953), 531–544. (Russian)

[23] F. Klein, *Vergleichende Betrachtungen über neuere geometrische Forschungen,* Andreas Diechert, Erlanger, 1872.

[24] Gábor Nagy & Petr Vojtěchovský, *GAP package loops - Computing with quasigroups and loops in GAP, Version 3.4.1*; 2018, `https://www.gap-system.org/Packages/loops.html`.

[25] K. D. Magill, *Semigroup structures for families of functions, I. Some homomorphism theorems*, J. Austral. Math. Soc. 7 (1967), 81–94.

[26] A. I. Mal'cev, *Symmetric groupoids*, Mat. Sbornik N.S. 31 (1952), 136–151. (Russian)

[27] W. McCune, *Mace4, Version 2009-11A*; 2009 `https://www.cs.unm.edu/~mccune/prover9/manual/2009-11A/mace4.html`.

[28] W. McCune, *Prover9, Version 2009-11A*; 2009 `https://www.cs.unm.edu/~mccune/prover9/manual/2009-11A/`.

[29] James Mitchel, *GAP package Semigroups - A package for semigroups and monoids, Version 3.4.0*; 2020, `https://www.gap-system.org/Packages/semigroups.html`.

[30] A. V. Molchanov, *On definability of hypergraphs by their semigroups of homomorphisms*, Semigroup Forum 62 (2001), 53–65.

[31] V. A. Molchanov, *Semigroups of mappings on graphs*, Semigroup Forum 27 (1983), 155–199.

[32] J. Rhodes and B. Steinberg, *The q-theory of finite semigroups*, Springer Verlag (2008).

[33] B. M. Schein, *Ordered sets, semilattices, distributive lattices and Boolean algebras with homomorphic endomorphism semigroups*, Fund. Math. 68 (1970), 31–50.

[34] J. Schreier, *Über Abbildungen einer abstrakten Menge Auf ihre Teilmengen*, Fund. Math. 28 (1936), 261–264.

[35] R. P. Sullivan, *Automorphisms of transformation semigroups*, J. Austral. Math. Soc. 20 (1975), 77–84.

[36] È. G. Šutov, *Homomorphisms of the semigroup of all partial transformations*, Izv. Vysš. Učebn. Zaved. Matematika 3 (1961), 177–184. (Russian)

[37] R. E. Ralph Tarjan, *Efficiency of a good but not linear set union algorithm,*; 1975, , J. Assoc. Comput. Mach. 22 (1975), 215–225.