ALGEBRAIC STRUCTURES IN PROOF ASSISTANT SYSTEMS

ALGEBRAIC STRUCTURES IN PROOF ASSISTANT SYSTEMS

BY

AKSHOBHYA KATTE MADHUSUDANA, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTERS OF SCIENCE

Masters of Science (2023)	McMaster University
(Department of Computing and Software)	Hamilton, Ontario, Canada

TITLE:	Algebraic Structures in Proof Assistant Systems
AUTHOR:	Akshobhya Katte Madhusudana B.Eng. (Computer Science and Engineering), Bangalore University, Bangalore, India
SUPERVISOR:	Dr. Jacques Carette
NUMBER OF PAGES:	xi, 100

Abstract

Abstract here (no more than 300 words)

Your Dedication

Optional second line

Acknowledgements

Acknowledgements go here.

Contents

Abstract **iii**

Acknowledgements **v**

Declaration of Academic Achievement **xi**

1 Introduction **1**

1.1 Research Outline 3

1.2 Thesis Outline 4

2 Universal Algebra: An Overview **5**

2.1 Relation and function 6

2.2 Universe, type and signature 7

2.3 Congruence and Morphism 9

3 Agda **11**

3.1 Types and functions 12

3.2 Structure definition 14

4 Algebraic Structures in Proof Assistant Systems - Survey **18**

4.1 Experimental setup 21

4.2 Algebraci Structures 21

4.3 Morphism 29

4.4 Properties 31

5 Theory Of Quasigroup and Loop in Agda **33**

5.1 Definition 34

5.2 Morphism 37

5.3 Morphism composition 40

5.4 DirectProduct 41

5.5 Properties 43

6 Theory of Semigroup and Ring in Agda **51**

6.1 Definition 52

6.2 Morphism 55

6.3 Morphism composition 57

6.4 Direct Product 57

6.5 Properties 61

7 Theory of Kleene Algebra in Agda **68**

7.1 Definition 68

7.2 Direct Product 72

7.3 Properties 73

8 Problem in Programming Algebra **79**

8.1 Ambiguity in naming 80

8.2 Equivalent but structurally different 82

8.3 Redundant field in structural inheritance 84

8.4 Identical structures 87

8.5 Equivalent structures 88

8.6 Mitigation using product family algebra 89

9 Conclusion **93**

A Your Appendix **94**

List of Figures

1.1 Algebraic structure hierarchy Wikipedia contributors (2022g) 2
8.1 Feature diagram of semigroup 90
8.2 Feature diagram of nearring 90
8.3 Feature diagram of quasigroup 91
8.4 Feature diagram of Nearring 92

List of Tables

4.1 Algebraic structures in proof assistant systems 24

Declaration of Academic Achievement

The student will declare his/her research contribution and, as appropriate, those of colleagues or other contributors to the contents of the thesis.

Chapter 1

Introduction

Abstract algebra, the study of algebraic structures is comparatively a new branch in mathematics that is being explored by many mathematicians. In recent years, applications of algebraic structures are explored in many fields. As an example, similar to differential equations on a function space, a time dependent partial differential equation can be studied using semigroup Liaqat and Younas (2021). Kleene algebra, semigroup structures are used in finite automata to better model and understand the finite state machines. Groups are one of the oldest structures that are used in number theory, in atomic and molecular theory, cryptography Wikipedia contributors (2023a). Quasigroups and loops are used in encryption techniques for image data Didurik and Shcherbacov (2018). The simplest algebraic structure is magma. A magma has a set with a binary operation that is closed by definition. A magma with associative property is called a semigroup and with division operation is called a quasigroup. Figure 1.1 shows the algebra hierarchy from magma to group.

With growing help of technology, mathematicians are more indulged in automated

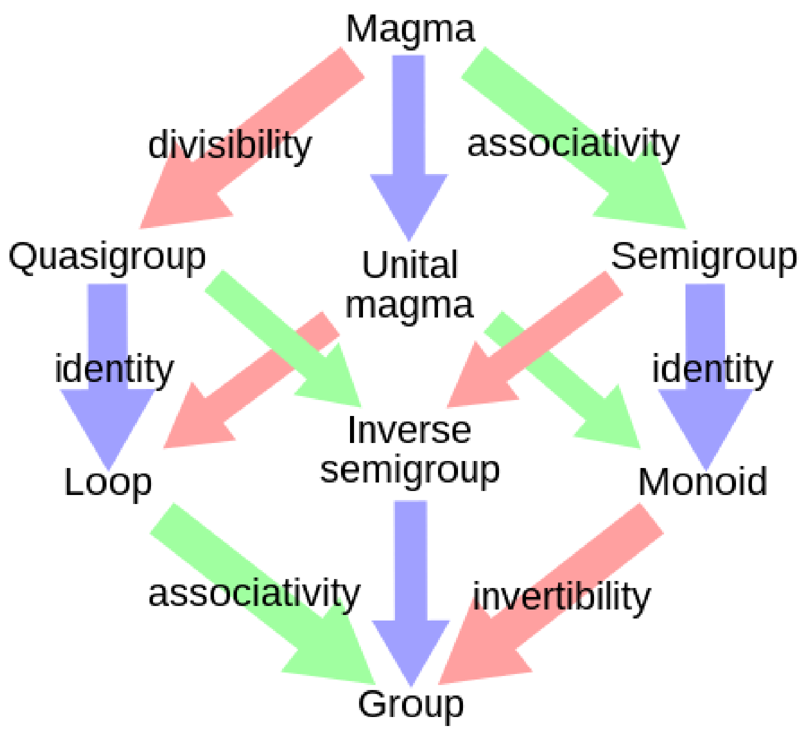

Figure 1.1: Algebraic structure hierarchy Wikipedia contributors (2022g)

reasoning. Increasing powers of computers, software tools that help towards automated reasoning becomes useful in their research. Although the proof systems that support first order logic are successful, developing a tool that supports higher order logic is complex Phillips and Stanovskỳ (2010). Proof assistant systems are the bridge between computer intelligence and human effort. Agda, Coq, Isabelle, Lean and Idris are some of the commonly used proof assistant systems. These systems help in automated reasoning in deriving mathematical proofs. For the scope of the thesis we only discuss about algebraic structures in proof systems.

1.1 Research Outline

For any software system to be robust, the libraries of these systems should be strong. In the sense that the software tool should support the user with all necessary functionalities. There are some experiments on how to automate the process of generating libraries with minimum human effort Jacques Carette and Sharoda (2019). These methods work in theory but becomes difficult in practice. For the complications in generated libraries, the standard libraries of proof systems do not rely on these generated code. This led to the question of what is the current scope of algebraic structures in the proof assistant systems. A survey was conducted to better understand the coverage of algebra in four proof systems agda, idris, lean and coq. Agda was one such system where there was better scope to contribute to the standard library. I was exposed to agda in course work as part of the program and added weight to chose agda over other systems.

As part of this thesis, more than twenty three structures was defined in the standard library for agda. Inspired by the application of structure semigroup, quasigroup, loop, ring, and kleene algebra, we define morphisms, direct product construct and prove some of the properties of the structures. One advantage of contributing to standard library is that others can use it to build on it.

When defining these structures, we analyze five problems that arise in programming algebra. Ambiguity in naming that is a structure can be misunderstood for other due to different usage of names in literature. Two structures that are equivalent but can be structurally different. When following the algebra hierarchy, it is possible to introduce redundant fields. A structure can be identical in the sense that it can have many names.

Same structure can be expressed and defined in more than one way but they result in equivalent structures. To overcome these problems we briefly introduce the use of product family algebra.

1.2 Thesis Outline

We start with background knowledge on universal algebra in chapter 2. Chapter 3 gives a brief overview of agda in relevance with algebraic structures. Chapter 4 justifies the scope of the thesis contribution by a survey on algebraic coverage in proof systems. The next three chapters are dedicated to discuss the structures in details. Chapter 5 discuss the properties of semigroup and rings with variations of ring structure. Chapter 6 is dedicated to quasigroup and loop that uses division operation. Chapter 7 discuss about kleene algebra, definition, construct and properties. Chapter 8 describes the problems that should be handled when programming algebra in proof systems with a brief overview of product family algebra. Conclusion and future works are discussed in chapter 9.

Chapter 2

Universal Algebra: An Overview

By the eary 18th century, mathematicians had discovered how to solve polynomial equation of up-to degree 4. There was no general rule to solve a polynomial equation of any degree. It wasn't until, mathematician Evariste Galois discovered a tool called *group* to solve polynomial equation of any degree. This tool was later discovered to be useful in other fields of mathematics such as modulus theory and geometryWikipedia contributors (2022g). Knowing the usefulness of this tool, mathematicians abstracted out the axioms of the group into a general tool. Thus evolved the structure *group* that we know today. As group theory, the study of *group* structures evolved, other abstract structures were invented to solve problems. This gave rise to a new field in mathematics called *abstract algebra*. Abstract algebra is the study of algebraic structure and its models or examples Wikipedia contributors (2022g).An algebraic structure is a tuple containing a 'carrier' set, A, a set of operations that act on A, and a set of axioms involving the operations and A. Some mathematicians were only interested in studying the structures themselves that is the arbitrary interpretation of the language and not the models that includes a theory that holds in the structure. Universal algebra is the study of algebraic structures and its properties. In the recent years, *universal algebra* has seen an exponential growth in its study of theories and applications Sankappanavar and Burris (1981).

2.1 Relation and function

In order to understand algebraic structures, it is essential to know some basics of relations and functions. In this section, we define relations and functions. We can start with defining a set.

A set is a well defined collection of objects. The *Cartesian product* between two sets X and Y, X × Y is defined as a pair {(x,y) : x ∈ X and y ∈ Y }.

A *binary relation* is a subset of the Cartesian product of two sets that is a mapping between one set called domain to the other set called the co-domain. A binary relation R on the set X to Y is denoted as an ordered pair (x,y) or xRy and element x in X and y in Y.

A *reflexive relation* R on set X is a subset on $X \times X$ is defined as $R : \{(x,x) : x \in X\}$ and can be denoted as xRx

A *symmetric relation* R on set X is a subset of $X \times X$ is defined as $R : \forall xy \in X : xRy \Longleftrightarrow yRx$

A relation R is said to be *transitive* on set X, is a subset of $X \times X$ such that $\forall xyz \in X$ if $(x,y) \in R$ and $(y,z) \in R$ then $(x,z) \in R$

A relation R is *equivalence* if it is reflexive, symmetric and transitive.

If in a relation, if every element in domain is mapped to only one element in the co-domain, then we call it a *function*.

Image of the function is the set of all elements in co-domain that is reachable from the function f in its domain.

A function f is *injective* if f maps distinct elements of domain to distinct elements of co-domain. That is function f on domain X in injective if $f(a) = f(b)$ then a = b.

A function is called *surjective* if the image of the function is same as its co-domain.

A function is called *bijective* if it is both injective and surjective.

An *operation* is defined as a function that can take zero or more inputs and maps it to a well defined output value. The number of operands is the arity of the operation.

2.2 Universe, type and signature

The naive set theory defines a set as well-defined collection of objects. If a set is defined using unrestricted comprehensive principleWikipedia contributors (2022c), then it leads to contradiction. This was first discovered by mathematician Bertrand Russell, and the paradox is called Russell's paradox. The paradox defines the set of all sets that are not the member of themselves Brilliant Math (2023). This develops to two kinds of contradiction:

1. If the set contains itself, then it should not be a member of itself by definition

2. If the set does not contain itself the it is not a member of itself.

A Formal definition of algebra is given in Sankappanavar and Burris (1981) as "For A a nonempty set and n a nonnegative integer we define $A^0 = \{\emptyset\}$, and, for n > 0, A^n is the set of n-tuples of elements from A. An n-ary operation (or function) on A is any function f from A^n to A; n is the arity (or rank) of f. A finitary operation is an n-ary operation, for some n. The image of $<a_1, . . ., a_n>$ under an n-ary operation f is denoted by $f(a_1, . . ., a_n)$. An operation f on A is called a nullary operation (or constant) if its arity is zero; it is completely determined by the image $f(\emptyset)$ in A of the only element $\emptyset$ in A^0, and as such it is convenient to identify it with the element $f(\emptyset)$. Thus a nullary operation is thought of as an element of A. An operation f on A is unary, binary, or ternary if its arity is 1,2, or 3, respectively"

In previous section we saw that group structure is one of the early structures studied in universal algebra. A group G is an algebra with one nullary (1), one unary ($^{-1}$) and one binary ($\cdot$) operation represented as $(G, \cdot, ^{-1}, 1)$ which satisfy the following axioms.

1. Associativity - $\forall xyz \in G, x \cdot (y \cdot z) \approx (x \cdot y) \cdot z$

2. Identity - $\forall x \in G, x \cdot 1 \approx 1 \cdot x \approx x$

3. Inverse - $\forall x \in G, x \cdot x^{-1} \approx x^{-1} \cdot x \approx 1$

Where $\approx$ is the equivalence relation.

The type (or language) of the algebra is a set of function symbols. Each member of this set is assigned a positive number that is the arity of the member. For example an algebra of type (2,0) denotes an algebra with one binary operation and one nullary operation. The group structure defined in previous section is of type (2,1,0). That is $\cdot$ is a binary operation, $^{-1}$ is a unary operation and 1 is the nullary operation.

The signature of an algebraic structure can be defined as a collection of relation and operations with their arity on the set of an algebraic structure. A structure with Ω signature is called as Ω algebra.

2.3 Congruence and Morphism

The congruence relation for an algebraic structure can be defined as an equivalence relation that is compatible with the structure such that the operations are well defined on the equivalence class. A more formal definition is for an algebra A of type F, congruence relation θ on A is defined using compatibility property that states that for each n-ary function symbol $f \in F$ and x_i , $y_i \in A$, If $x_i \,\theta\, y_i$ holds for $1 \leq i \leq n$ then $f^A(x_1,...,x_n)\theta f^A(y_1,....,y_n)$ holdsSankappanavar and Burris (1981).

A Morphism is a structure preserving map between two algebraic structures. It is an abstraction that generalizes the map between two structures or mathematical objects in general.
If A and B are two algebras of same type F, then a homomorphism is defined as a mapping α from algebra A to B such that

$$\alpha f^{\mathrm{A}}(a_1, a_2, ..., a_{\mathrm{n}}) = f^{\mathrm{B}}(\alpha a_1, \alpha a_2, ..., \alpha a_{\mathrm{n}})$$

for each n-ary f in F and each sequence $a_1, a_2, ..., a_n$ from A. In Sankappanavar and Burris (1981), the author proves that if α: A → B and β: A → B are homomorphisms on algebra A to B such that $\alpha(a) = \beta(a)$ then $\alpha = \beta$

For two algebras A and B, if $\alpha : A \to B$ is a homomorphism from A to B, if α satisfies one-to-one mapping (i.e., α is injective) then the morphism α is called monomorphism.

For two algebras A and B, if $\alpha : A \to B$ is a Monomorphism from A to B, if α is a bijection from A to B, then α is called isomorphism.

In Sankappanavar and Burris (1981), the author proves that the composite of two homomorphism (monomorphism/isomorphism) is also a homomorphism (monomorphism/isomorphism).

Chapter 3

Agda

Agda is a dependently typed programming. Agda is pure in the sense that the functions have no effect and does not use state. Agda uses lazy evaluation. That is the expressions are not evaluated until the are needed. Therefore in Agda, the order of evaluation is hard to predict. Strict evaluation is to evaluate all arguments to the function before evaluating the function. Since Agda is lazy, both strict and lazy evaluation will give the same output Kidney (2020) Agda is total. A function in agda gives output in a finite amount of time for a valid input.

The process of validating and imposing constraints on values is called type checking. Agda can be compiled to haskell or javascript but it is only type checked so it can be used as a proof assistant. Agda is based on unified theory of dependent types Wikipedia contributors (2022b) hence the program written in Agda is in line with the Martin-Löf Type Theory Kidney (2020). This chapter provides a brief overview of programming in Agda in the context of algebraic structures.

3.1 Types and functions

Agda provides logical framework in the sense that the core of Agda provides a framework that gives the type Set and dependent functions $(x : A) \to B$. Agda supports inductive types.Bove et al. (2009). Agda provides simple types and parameter types. These data types are declared using the keyword data.

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat
```

In the above example, Nat is the data type that has two constructors zero and suc. Constructor are used ti assign values tot eh variables in the type. This type is called inductive type. In this example, the smallest set is a set containing an element zero and is closed under the function suc. Since the properties of this function can be proved inductively, the type us called inductive type. Wikipedia contributors (2022b)

The datatype can be have parameters and indexed. Another way of defining a type is using the keyword record. A record type can be defined by referencing other types and creating a synonym. An example of record type is discussed later in the chapter when we define the algebraic structure.

Since types are values in Agda, there is no real way to distinguish between them. If bool is a simple type or type_0, then what is the type of type_0? Note that this is similar to the Rusell's paradox that is discussed in previous chapter. Agda uses universe polymorphism to resolve this issue. That is the type of 'true' is Bool and its type is type_0. The type of type_0 is type_1 and so on. Kidney (2020).

Similarly, in Agda not every type is a set and the set type can be defined using keyword

Set_1. "A type whose elements are types are called universe" uni (2023). This primitive type is useful to define and prove theorems about functions that operate on large set.

The functions in Agda is very similar to function in Haskell. A function in Agda is defined by declaring the type followed by the clauses Wikipedia contributors (2022a).

```
f : (x₁ : A₁) → ... → (xₙ : Aₙ) → B
f p₁ ... pₙ = d
...
f q₁ ... qₙ = e
```

Where f is the function identified, p_i and q_i are the patterns of type A_i. d and e are expressions. The agda documentation discuss other techniques to define a function such as using dot patterns, absurd patterns, as patterns and case trees Wikipedia contributors (2022a).

An operator in agda is also defined as a function. Underscore is used to indicate where an argument is expected. and operator can be defined as

```
_and_ : Bool → Bool → Bool
true and x = x
false and _ = false
```

In context of algebra, agda defines unary (Op_1) and binary (Op_2) operations.

```
Op₁ : ∀ {ℓ} → Set ℓ → Set ℓ
Op₁ A = A → A
```

```
Op₂ : ∀ {ℓ} → Set ℓ → Set ℓ
Op₂ A = A → A → A
```

3.2 Structure definition

Algebraic structures are defined as record types in Agda. Records types are used to group values together and they provide named fields to generalise dependent product types. The structures are obtained by wrapping the predicates that are expressed as "is-a" relation. (Hu and Carette, 2021)

```
record IsMagma (· : Op₂ A) : Set (a ⊔ ℓ) where
  field
    isEquivalence : IsEquivalence _≈_
    ·-cong        : Congruent₂ ·

  open IsEquivalence isEquivalence public
```

In the above example structure IsMagma is defined as a record type with fields isEquivalence and ·-cong. · is a binary operation on set A. a $\sqcup$ ℓ is the least upper bound for the set. _≈ _is the binary operation argument for IsEquivalence. If a relation P on set A is equivalent to relation Q on set B, then we say f preserves p for some map f from set A to B. Congruent_2 · represents that the binary operation · preserves equivalence relation. IsEquivalence and Congruent_2 are predicates defined in standard library.

We open the module isEquivalence to be able to use it in defining other structures in the algebra hierarchy. The open statement is made public using the keyword public to be able to re-export the names from another module.

Morphisms of the structure are defined as record type in Agda standard library.

Agda standart library defines the bundled version of the structures that contains the operations of the structures, sets and axioms.

```
record Magma c ℓ : Set (suc (c ⊔ ℓ)) where
  infixl 7 _·_
  infix  4 _≈_
  field
    Carrier : Set c
    _≈_      : Rel Carrier ℓ
    _·_     : Op₂ Carrier
    isMagma : IsMagma _≈_ _·_

  open IsMagma isMagma public

  rawMagma : RawMagma _ _
  rawMagma = record { _≈_ = _≈_; _·_ = _·_ }

  open RawMagma rawMagma public
    using (_≉_)
```

Above is the bundled version of IsMagma structure. RawMagma is the raw version of the magma with only the operators and set. infix<l,r> denotes the fixity and precedence of the operator. using keyword is used to export only the fields that are mentioned in its arguments.

When exporting the modules we may need to rename the fields to avoid having duplicate names.

```
open IsMagma *-isMagma public
  using ()
  renaming
  ( ·-congˡ  to *-congˡ
  ; ·-congʳ  to *-congʳ
  )
```

Keyword renaming is used to rename the fields. In the above sample code, we rename $\cdot$-cong^l to $*$-cong^l and $\cdot$-cong^r to $*$-cong^r.

3.2.1 Equational Proofs in Agda

In the Russell's paradox discussed in the previous sections, if a theorem is to be proved, we need to use the axiom that is the theorem itself. To overcome this dutch mathematician L. E. J. Brouwer discovered intuitionism that led to constructive mathematics Wikipedia contributors (2022e). In constructive mathematics, knowledge comes with implicit arguments. In constructive proofs, the existence of a mathematical object is given by giving a way to create the method. Wikipedia contributors (2022d). In agda, a function to and from a each type is provided if there is a bijection between two types. Dependently typed language agda thus allow to compile and run proofs Kidney (2020). In Agda, 'begin' is used to indicate the start of the proof. begin is a function that relates two objects.

```
begin_ : ∀ {x y} → x IsRelatedTo y → x ∼ y
begin relTo x∼y = x∼y
```

IsRelatedTo is a type defined to infer arguments even if the underlying equality evaluates. standard step to relation is defined as step-∼

```
step-∼ : ∀ x {y z} → y IsRelatedTo z → x ∼ y → x IsRelatedTo z
step-∼ _ (relTo y∼z) x∼y = relTo (trans x∼y y∼z)
```

step using equality is given as

```
step-≈ = Base.step-∼
syntax step-≈ x y≈z x≈y = x ≈⟨ x≈y ⟩ y≈z
```

The termination of the proof is given using _■

```
_■ : ∀ x → x IsRelatedTo x
x ■ = relTo refl
```

Agda supports quantifiers. Universal quantifier is denoted as ∀ and existential quantifier is denoted as ∃

Below is the example to the proposition x · (y · z) = (x · y) · z for a semigroup i.e., a Magma with associative property (x · (y · z) = (x · y) · z)

```
x·yz≈xy·z : ∀ x y z → x · (y · z) ≈ (x · y) · z
x·yz≈xy·z x y z = begin
  x · (y · z) ≈⟨ sym (assoc x y z) ⟩
  (x · y) · z ■
```

In the proposition x y z are in set S that is a semigroup. "sym (assoc x y z)" is the reasoning for the proof.

Chapter 4

Algebraic Structures in Proof Assistant Systems - Survey

The proof assistant systems are computer software that helps to derive formal proofs with a joint effort of computers and humans. Computer proof assistants are used to formalize theories, and extend them by logical reasoning and defining properties. Saqib Nawaz et al. (2019) These systems are used to perform mathematics on computers. The proof assistant systems are widely used in defining and proving rather than doing numerical computations. The automated theorem proving is different from proof assistants in that they have less expressivity and make it almost impossible to define a generic mathematical theory. In Michael and Florian (2021). the author discusses several concerns on considering the proofs that are exclusively verified using a computer to be considered as a valid mathematical proof. However, the proofs written using the proof assistant systems are widely accepted among mathematicians and computer scientists.

The strength of any system is directly dependent on the availability of standard libraries for those systems. The standard libraries are expected to provide resources to ease the use of such systems. In Jacques Carette and Sharoda (2019), the author discusses the difficulties in building such large libraries. One such problem is to structurally derive algebraic structures from one another in the hierarchy without explicitly defining axioms that become redundant. The author also proposes a solution to make use of the interrelationship in mathematics and thus reduce the efforts in building the library.

We consider the four more commonly used proof assistant systems that are all dependently typed, higher order programming languages that supports (atleast partially) proof by reflection. Proof by reflection is a technique where the system allows to derive proofs by systematic reasoning methods.

Agda 2 is a proof assistant system where proofs are expressed in a functional programming style. The Agda standard library aims to provide tools to ease the effort of writing proofs and also programs. The current version of the Agda standard library (1.7) is fully supported for the changes and developments in Agda. It provides clear documentation for installation, contribution, and style guide for the standard library.

Idris is developed as a functional programming language but is also used as a proof assistant system. The proofs are alike with coq and the type systems in Idris is uniform with agda. Idris 2 is a self-hosted programming language that combines linear-type-system. In this article, Idris 2 and Idris in used interchangeably and refer to Idris 2. Currently, there are no official package managers for Idris 2. However, several versions are under development.

Coq Paulin-Mohring (2012) is a theorem proving system that is written in the Ocaml programming language. It was first released in 1989 and is one of the most widely used proof assistant systems to define mathematical definitions, theory and to write proofs. The mathematical components library (1.12.0) includes various topics from data structures to algebra. In this article, we consider the mathematical component repository (mathcomp) that contains formalized mathematical theories. Mahboubi and Tassi (2021) The latest available release of mathcomp library is 1.12.0. The mathcomp library was started with the Four Colour Theorem to support formal proof of the odd order theorem.

Lean mathlib Community (2020) is an open-source project by Microsoft Research. Lean is a proof assistant system written in C++. The last official version of Lean was 3.4.2 and is now supported by the lean community. Lean 4 is the latest version of Lean and is a complete rewrite of previous versions of Lean. The mathlib mathlib Community (2020) library for lean 3 has the most coverage of algebra compared to the other 3 proof assistant systems discussed in the paper. The mathlib library of Lean is also maintained by the lean community for community versions of lean. It was developed on a small library that was in lean. It contained definitions of natural numbers, integers, and lists and had some coverage over algebra hierarchy. The latest version of mathlib has over 2794 definitions of algebra [3].

The main aim of this paper is to provide documentation for the algebraic coverage in various proof assistant systems. The most commonly used proof assistant systems are Agda, Idris2, Coq, and Lean. In this article, the latest available versions are considered i.e., Agda standard library 1.7, Idris 2.0,The Mathematical Components Library 1.13.0, and The Lean mathematical library.

4.1 Experimental setup

It is not time efficient to manually look for the definitions in a large library. The source code of the standard libraries of Agda, Idris, Coq and lean are publicly available. We created a web crawler that extracts the code from the source code webpage and built a regular expression that is unique to each system to extract definitions. Thus a part of the process of building the table 1. was automated. Since the standard libraries are open source projects, it is difficult to maintain uniformity in the code. For example the definition might start with comment in the same line or structure parameters might be written in a new line. All this makes it difficult to correctly build the regular expression and will necessitate the task of verifying the results manually to some extent.

The rest of the article is structured as follows. Section 2 discuss about the algebraic structure definitions and its coverage in the proof assistant systems, while the section 3 covers the morphism definitions in those systems. The properties and solvers coverage in presented in section 4. The last section of the paper is the conclusion and discussion.

4.2 Algebraci Structures

The Agda standard library provides definitions with bundled versions of several algebraic structures. Algebra hierarchy is followed in defining structures Michael and Florian (2021). As an example, a semigroup is derived from magma and a monoid from semigroup.

```
record IsMagma (· : Op₂ A) : Set (a ⊔ ℓ) where
  field
    isEquivalence : IsEquivalence _≈_
    ·-cong        : Congruent₂ ·

  open IsEquivalence isEquivalence public

record IsSemigroup (· : Op₂ A) : Set (a ⊔ ℓ) where
  field
    isMagma : IsMagma ·
    assoc   : Associative ·

  open IsMagma isMagma public
```

The same follows for the bundle definitions of respective structures. Since the current version of the library has a limited number of structures, there might arise a problem of extending the hierarchy as described in [2]. One exemption for this hierarchical definition is the definition of a lattice. A lattice is defined independently in the standard library to overcome the redundant idempotent fields. A lattice structure that is defined in terms of join and meet semilattice is being added as a biased structure. The 1.7 version of the agda standard library has the definitions of structures with the respective bundle versions of Magma, Commutative Magma, to Ring, CommutativeRing, and Boolean Algebra. Another definition of most of the above algebraic structures is provided as a direct product of two other instances of algebraic structures. However, from semigroups, the structures are defined in terms of relevant categories. The structures also include respective bundle definitions. A module is an abelian group with the ring of scalars. The ring of scalars has an identity element. The agda standard library defines left, right, and bi semimodules and modules. A similar hierarchical approach as other algebraic structures is followed in defining modules. As an example, a module is defined using bimodules and bimodules using bi-semimodules. An alternative definition of modules is given in “Algebra.Module.Structure.Biased”. The structures have respective bundled versions.

In Idris 2, there is a considerable overlap of abstract algebra and category theory. The library defines various algebraic structures that include semigroup, monoid, group, abelian-group, semiring, and ring. It follows a hierarchical approach in defining structures similar to that in agda. For example, a semigroup is defined as a set with a binary operation that is associative and a monoid is defined in terms of semigroup with an identity element. Idris addresses identity as a neutral element.

```
interface Semigroup t where
  (<+>) : t -> t -> t
  semigroupOpIsAssociative : (l, c, r : t) -> l <+> (c <+> r) = (l <+> c) <+> r

interface Semigroup t => Monoid t where
  neutral : t
  monoidNeutralIsNeutralL : (l : t) -> l <+> neutral = l
  monoidNeutralIsNeutralR : (r : t) -> neutral <+> r = r
```

The algebra structures design hierarchy of the mathcomp library is inspired by the Packing mathematical structures. The ssralg file defines most of the basic algebraic structures with their type, packers, and canonical properties. The hierarchy extends from Zmodule, rings to ring morphisms. The countalg file extends ssralg file to define countable types.

The mathlib extends the algebra hierarchy from semigroup to ordered fields. The library defines instances of free magma, free semigroup, free Abelian group, etc. An example of semigroup structure definition in the library is given below:

```
structure semigroup (G : Type u) :
  Type u
  mul : G → G → G
  mul_assoc : forall (a b c : G), (a * b) * c = a * b * c
```

Other instances of semigroups are derived from the definition of semigroup structure such as commutative semigroup, left and right cancellative semgroup. Similar definitions are extended from monoid and other structures.

Table 4.1: Algebraic structures in proof assistant systems

Algebraic Structure	**Agda**	**Coq**	**Idris**	**Lean**
Magma	✓	-	-	-
Commutative Magma	✓	-	-	-
Selective Magma	✓	-	-	-

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
IdempotentMagma	✓	-	-	-
AlternativeMagma	✓	-	-	-
FlexibleMagma	✓	-	-	-
MedialMagma	✓	-	-	-
SemiMedialMagma	✓	-	-	-
Semigroup	✓	✓	✓	✓
Band	✓	-	-	-
Commutative Semigroup	✓	-	-	✓
Semilattice	✓	-	-	✓
Unital magma	✓	-	-	-
Monoid	✓	✓	✓	✓
Commutative monoid	✓	✓	-	✓
Idempotent commutative monoid	✓	-	-	-
Bounded Semilattice	✓	-	-	-
Bounded Meetsemilattice	✓	-	-	-
Bounded Joinsemilattice	✓	-	-	-
Invertible Magma	✓	-	-	-
IsInvertible UnitalMagma	✓	-	-	-
Quasigroup	✓	-	-	-
Loop	✓	-	-	-

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
Moufang Loop	✓	-	-	-
Left Bol Loop	✓	-	-	-
Middle Bol Loop	✓	-	-	-
Right Bol Loop	✓	-	-	-
NilpotentGroup	-	-	-	✓
CyclicGroup	-	-	-	✓
SubGroup	-	-	-	✓
Group	✓	✓	✓	✓
Abelian group	✓	-	✓	✓
Lattice	✓	-	-	✓
Distributive lattice	✓	-	-	-
Near semiring	✓	-	-	-
Semiringwithout one	✓	-	-	-
Idempotent Semiring	✓	-	-	-
Commutative semiring without one	✓	-	-	-
Semiring without annihilating zero	✓	-	-	-
Semiring	✓	✓	-	✓
Commutative semiring	✓	-	-	✓
Non associative ring	✓	-	-	-
Nearring	✓	-	-	-

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
Quasiring	✓	-	-	-
Local ring	-	-	-	✓
Noetherian ring	-	-	-	✓
Ordered ring	-	-	-	✓
Cancellative commutative semiring	✓	-	-	-
Sub ring	-	-	-	✓
Ring	✓	✓	✓	✓
Unit Ring	✓	✓	✓	-
Commutative Unit ring	-	✓	-	-
Commutative ring	✓	✓	-	✓
Integral Domain	-	✓	-	-
LieAlgebra	-	-	-	✓
LieRing module	-	-	-	✓
Lie module	-	-	-	✓
Boolean algebra	✓	-	-	-
Preleft semimodule	✓	-	-	-
Left semimodule	✓	-	-	-
Preright semimodule	✓	-	-	-
right semimodule	✓	-	-	-
Bi semimodule	✓	-	-	-

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
Semimodule	✓	-	-	-
Left module	✓	✓	-	-
Right module	✓	-	-	-
Bi module	✓	-	-	-
Module	✓	✓	-	✓
Field	-	✓	✓	✓
Decidable Field	-	✓	-	-
Closed field	-	✓	-	-
Algebra	-	✓	-	-
Unit algebra	-	✓	-	✓
Lalgebra	-	✓	-	-
Commutative unit algebra	-	✓	-	-
Commutative algebra	-	✓	-	-
NumDomain	-	✓	-	-
Normed Zmodule	-	✓	-	-
Num field	-	✓	-	-
Real domain	-	✓	-	-
Real field	-	✓	-	-
Real closed field	-	✓	-	-
Vector space	-	✓	-	-

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
Zmodule Quotients type	-	✓	-	-
Ring Quotient type	-	✓	-	-
Unit rint quotient type	-	✓	-	-
Additive group	-	✓	-	-
characteristic zero	-	-	-	✓
Domain	-	-	-	✓
Chain Complex	-	-	-	✓
Kleene Algebra	✓	-	-	-
IsHeytingCommutativeRing	✓	-	-	-
IsHeytingField	✓	-	-	-

4.3 Morphism

One of the benefits of the Agda standard library is that it provides morphisms for the structures defined in the library. A raw bundle instance is defined in Algebra. Bundles and the morphisms for those raw structures are provided. For example, raw magma is used in magma morphisms. The library defines homomorphism, monomorphism, and isomorphism for those structures. The library also provides the composition of morphisms between algebraic structures. The morphism definitions for Magma, Monoid, Group, NearSemiring, Semiring, Ring, Lattice are available in the standard library. An example of magma morphisms as defined in the standard library is as follows.

```
record IsMagmaHomomorphism (⟦_⟧ : A → B) : Set (a ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    isRelHomomorphism : IsRelHomomorphism _≈₁_ _≈₂_ ⟦_⟧
    homo              : Homomorphic₂ ⟦_⟧  _·_ _∘_

  open IsRelHomomorphism isRelHomomorphism public
    renaming (cong to ⟦⟧ -cong)
```

Similar definitions for monomorphism and isomorphism are included in agda standard library.

The morphism definitions in the Idris library define morphisms in category theory. A group homomorphism is a structure-preserving function between two groups and is defined as follows :

```
interface (Group a, Group b) => GroupHomomorphism a b where
  to : a -> b

  toGroup : (x, y : a) ->  to (x <+> y) = (to x) <+> (to y)
```

The group theory directory defines groups, group morphisms, subgroups, cyclic, nilpotent groups, and isomorphism theorems. There is no group homomorphism instead, it is defined with proofs for map-one and map-mul for monoid homomorphism. The definition of monoid homomorphism is give below:

```
structure monoid_hom (M : Type*) (N : Type*) [mul_one_class M] [mul_one_class N]
  extends one_hom M N, mul_hom M N
```

The mathlib library extends monoid and groups to define rings and ring morphisms. Bundled structure is used to define ring morphisms.

4.4 Properties

The Agda standard library provides constructs of modules such as a bi-product construct and tensor unit using two R-modules. The library also includes the relation between function properties with basal setoid and sets for propositional equalities. The library includes ring, monoid solvers for equations of the same. However, these solvers are under construction and not optimized for performance.

The coq library has rings and field tactics to achieve algebraic manipulations in some of the algebraic structures. The library also includes specialized tactics such as interval and gappa to work with real numbers and floating point nubmers. Paulin-Mohring (2012)

The Idris library defines properties or laws of algebraic structures. The unique-Inverse defines that the inverses of monoids are unique. Other laws on groups include self-squaring i.e., identity element of a group is self-squaring, inverse elements of a group satisfy the commutative property, laws of double negation. It also defines squareId-Commutative i.e., a group is abelian if every square in a group is neutral, inverseNeutralIsNeutral, and other properties of an algebraic group. The Latin-square-property is defined as for any two elements a and b, ax =b and ya = b exists. Other algebraic properties for groups are y = z if x + y = x + z, y = z if y + x = z + x, ab = 0 -> a = b-1, and ab = 0 -> a-1 = b. An example of a definition is shown below.

```
public export
neutralProductInverseL : Group ty => (a, b : ty) ->
  a <+> b = neutral {ty} -> inverse a = b
neutralProductInverseL a b prf =
  cancelLeft a (inverse a) b $
    trans (groupInverseIsInverseL a) $ sym prf
```

The library also includes laws on homomorphism that homomorphism over group preserves identity and inverses. Some laws on ring structures are also included in the library such as x0 = 0, (-x)y = -(xy), x(-y) = -(xy), (-x)(-y) = xy, (-1)x = -x, and x(-1) = -x. The algebraic coverage of Idris 2 is limited and is under development. There are no official definitions for solvers or higher structures such as modules, fields, or vector space. The Idris 2 is comparatively new and is under continuous development to strengthen the language and also as a mechanical reasoning system.

The mathlib library of Lean 3 includes algebra over rings such as associative algebra over a commutative ring, Lie algebra, Clifford algebra, etc. Lie algebra is defined as a module satisfying Jacobi identity. Without scalar multiplication, a lie algebra is a lie ring. The library extends rings to define fields and division ring covering many aspects of fields such as the existence of closure for a field, Galois correspondence, rupture field, and others.

Chapter 5

Theory Of Quasigroup and Loop in Agda

Applications of non associative algebras are explored in various fields of study. For example, Einstein's formula of addition of velocities gives a loop structure Ungar (2007). Quasigroups of various orders are used in field of cryptography Phillips and Stanovskỳ (2010). Lie algebra is used in differential geometryWikipedia contributors (2022h). Proof assistant systems such as Agda are helpful in verifying some of the properties of these structures. They are interactive software that help to derive complex mathematical proofs. In this chapter we formalize two important non associative algebras - quasigroup, loop structure. A Quasigroup (Q, ·, /, \) is a type (2,2,2) algebra for which the binary operations \and / are defined such that division is always possible. A loop is a quasigroup with identity. We explore morphisms and direct product for these structures and derive proofs for some of the properties of these structures.

5.1 Definition

A set that has a binary operation is called Magma. In this case a Magma is total and should not be confused with groupoid that need not be total. Left and division is defined with identities.

$$y = x \cdot (x \backslash y) \tag{5.1.1}$$

$$y = x \backslash (x \cdot y) \tag{5.1.2}$$

$$y = (y / x) \cdot x \tag{5.1.3}$$

$$y = (y \cdot x) / x \tag{5.1.4}$$

The Agda definition is given below

```
LeftDividesˡ  : Op₂ A → Op₂ A → Set _
LeftDividesˡ  _·_  _\\_ = ∀ x y → (x · (x \\ y)) ≈ y


LeftDividesʳ : Op₂ A → Op₂ A → Set _
LeftDividesʳ _·_ _\\_ = ∀ x y → (x \\ (x · y)) ≈ y


RightDividesˡ  : Op₂ A → Op₂ A → Set _
RightDividesˡ  _·_ _//_ = ∀ x y → ((y // x) · x) ≈ y


RightDividesʳ : Op₂ A → Op₂ A → Set _
RightDividesʳ _·_ _//_ = ∀ x y → ((y · x) // x) ≈ y
```

We can combine the left and right division as follows

```
LeftDivides : Op₂ A → Op₂ A → Set _
LeftDivides · \\ = (LeftDividesˡ · \\) × (LeftDividesʳ · \\)


RightDivides : Op₂ A → Op₂ A → Set _
RightDivides · // = (RightDividesˡ · //) × (RightDividesʳ · //)
```

Note that we use // and \\instead of / and \respectively to overcome the conflict with overloaded or escape characters.

The Quasigroup structure can be structurally derived from Magma in Agda as

```
record IsQuasigroup (· \\ // : Op₂ A) : Set (a ⊔ ℓ) where
  field
    isMagma       : IsMagma ·
    \\-cong       : Congruent₂ \\
    //-cong       : Congruent₂ //
    leftDivides   : LeftDivides · \\
    rightDivides  : RightDivides · //

  open IsMagma isMagma public
```

A loop is a quasigroup that has identity element.

$$x{\cdot}e = e{\cdot}x = x \tag{5.1.5}$$

```
LeftIdentity : A → Op₂ A → Set _
LeftIdentity e _·_ = ∀ x → (e · x) ≈ x


RightIdentity : A → Op₂ A → Set _
RightIdentity e _·_ = ∀ x → (x · e) ≈ x


Identity : A → Op₂ A → Set _
Identity e · = (LeftIdentity e ·) × (RightIdentity e ·)
```

Loop structure can be structurally derived from quasigroup.

```
record IsLoop (· \\ // : Op₂ A) (ϵ : A) : Set (a ⊔ ℓ) where
  field
    isQuasigroup : IsQuasigroup · \\ //
    identity         : Identity ϵ ·

  open IsQuasigroup isQuasigroup public
```

A loop is called a right bol loop if it satisfies the identity (Equation 5.1.6)

$$((z \cdot x) \cdot y) \cdot x = z \cdot ((x \cdot y) \cdot x) \tag{5.1.6}$$

A loop is called a left bol loop if it satisfies the identity (Equation 5.1.7)

$$x \cdot (y \cdot (x \cdot z)) = (x \cdot (y \cdot x)) \cdot z \tag{5.1.7}$$

A loop is called middle bol loop if it satisfies the identity (Equation 5.1.8)

$$(z \cdot x) \cdot (y \cdot z) = z \cdot ((x \cdot y) \cdot z) \tag{5.1.8}$$

A left-right bol loop is called a moufang loop if it satisfies identity (Equation 5.1.9)

$$(z \cdot x) \cdot (y \cdot z) = z \cdot ((x \cdot y) \cdot z) \tag{5.1.9}$$

```
LeftBol : Op₂ A → Set _
LeftBol _·_ = ∀ x y z → (x · (y · (x · z))) ≈ ((x · (y · x)) · z )


RightBol : Op₂ A → Set _
RightBol _·_ = ∀ x y z → (((z · x) · y) · x) ≈ (z · ((x · y) · x))


MiddleBol : Op₂ A → Op₂ A  → Op₂ A  → Set _
MiddleBol _·_ _\\_ _//_ = ∀ x y z → (x · ((y · z) \\ x)) ≈ ((x // z) · (y \\ x))

Identical : Op₂  A → Set _
Identical _·_ = ∀ x y z → ((z · x) · (y · z)) ≈ (z · ((x · y) · z))
```

5.2 Morphism

A structure preserving map f between two structures of same type is called morphism or homomorphism. That is f : A → B and · is an operation on the structure then homomorphism is defined as f(x · y) = f(x) · f(y). A homomorphism that is injective is called monomorphism. If the structures are identical that is they are more than just similar in structure then we can compare the structures with isomorphism. A homomorphism that is bijective is called isomorphism. The quasigroup homomorphism preserves both left and right division operations.

```
record IsQuasigroupHomomorphism (⟦_⟧ : A → B) : Set (a ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    isRelHomomorphism : IsRelHomomorphism _≈₁_ _≈₂_ ⟦_⟧
    ∙-homo            : Homomorphic₂ ⟦_⟧ _∙₁_ _∙₂_
    \\-homo           : Homomorphic₂ ⟦_⟧ _\\₁_ _\\₂_
    //-homo           : Homomorphic₂ ⟦_⟧ _//₁_ _//₂_

  open IsRelHomomorphism isRelHomomorphism public
    renaming (cong to ⟦⟧-cong)

record IsQuasigroupMonomorphism (⟦_⟧ : A → B) : Set (a ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    isQuasigroupHomomorphism : IsQuasigroupHomomorphism ⟦_⟧
    injective                : Injective ⟦_⟧

  open IsQuasigroupHomomorphism isQuasigroupHomomorphism public

record IsQuasigroupIsomorphism (⟦_⟧ : A → B) : Set (a ⊔ b ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    isQuasigroupMonomorphism : IsQuasigroupMonomorphism ⟦_⟧
    surjective               : Surjective ⟦_⟧

  open IsQuasigroupMonomorphism isQuasigroupMonomorphism public
```

The loop morphism preserves left and right divisions along with the identity element

```
record IsLoopHomomorphism (⟦_⟧ : A → B) : Set (a ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    isQuasigroupHomomorphism : IsQuasigroupHomomorphism ⟦_⟧
    ε-homo                   : Homomorphic₀ ⟦_⟧ ε₁ ε₂

  open IsQuasigroupHomomorphism isQuasigroupHomomorphism public

record IsLoopMonomorphism (⟦_⟧ : A → B) : Set (a ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    isLoopHomomorphism   : IsLoopHomomorphism ⟦_⟧
    injective            : Injective ⟦_⟧

  open IsLoopHomomorphism isLoopHomomorphism public

record IsLoopIsomorphism (⟦_⟧ : A → B) : Set (a ⊔ b ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    isLoopMonomorphism   : IsLoopMonomorphism ⟦_⟧
    surjective           : Surjective ⟦_⟧

    open IsLoopMonomorphism isLoopMonomorphism public
```

5.3 Morphism composition

If f is a morphism such that f : a → b and g is a morphism on same structure such that g : b → c then composition of morphism can be defined as g ∘ f : a → c.

```
isQuasigroupHomomorphism : IsQuasigroupHomomorphism Q₁ Q₂ f
  → IsQuasigroupHomomorphism Q₂ Q₃ g
  → IsQuasigroupHomomorphism Q₁ Q₃ (g ∘ f)
isQuasigroupHomomorphism f-homo g-homo = record
  { isRelHomomorphism = isRelHomomorphism
              F.isRelHomomorphism
              G.isRelHomomorphism
  ; ∙-homo    = λ x y → ≈₃-trans
              (G.⟦⟧-cong ( F.∙-homo x y ))
              ( G.∙-homo (f x) (f y) )
  ; \\-homo     = λ x y → ≈₃-trans
              (G.⟦⟧-cong ( F.\\-homo x y ))
              ( G.\\-homo (f x) (f y) )
  ; //-homo     = λ x y → ≈₃-trans
              (G.⟦⟧-cong ( F.//-homo x y ))
              ( G.//-homo (f x) (f y) )
  } where module F = IsQuasigroupHomomorphism f-homo;
          module G = IsQuasigroupHomomorphism g-homo
```

```
isLoopHomomorphism : IsLoopHomomorphism L₁ L₂ f
  → IsLoopHomomorphism L₂ L₃ g → IsLoopHomomorphism L₁ L₃ (g ∘ f)
isLoopHomomorphism f-homo g-homo = record
  { isQuasigroupHomomorphism = isQuasigroupHomomorphism ≈₃-trans
                  F.isQuasigroupHomomorphism G.isQuasigroupHomomorphism
  ; ϵ-homo              = ≈₃-trans (G.⟦⟧-cong F.ϵ-homo) G.ϵ-homo
  } where module F = IsLoopHomomorphism f-homo;
            module G = IsLoopHomomorphism g-homo
```

Monomorphism and isomorphism compositions constructs for quasigroup and loop are defined similar to homomorphism and can be found in agda standard library.

5.4 DirectProduct

The direct product M × N of two quasigroups M and N is defined as a pair (m,n) where m ∈ M and n ∈ N. The direct product construct of left (right/middle) bol loop and moufang loop can be found in agda standard library and can be derived from loop structure.

```
quasigroup : Quasigroup a ℓ₁ → Quasigroup b ℓ₂
                 → Quasigroup (a ⊔ b) (ℓ₁ ⊔ ℓ₂)
quasigroup M N = record
  { _\\_  = zip M._\\_ N._\\_
  ; _//_    = zip M._//_ N._//_
  ; isQuasigroup = record
    { isMagma = Magma.isMagma (magma M.magma N.magma)
    ; \\-cong = zip M.\\-cong N.\\-cong
    ; //-cong = zip M.//-cong N.//-cong
    ; leftDivides = (λ x y → M.leftDividesˡ ,
                 N.leftDividesˡ <*> x <*> y),
                  (λ x y → M.leftDividesʳ ,
                N.leftDividesʳ <*> x <*> y)
    ; rightDivides = (λ x y → M.rightDividesˡ ,
                 N.rightDividesˡ <*> x <*> y),
                    (λ x y → M.rightDividesʳ ,
                 N.rightDividesʳ <*> x <*> y)
    }
  } where module M = Quasigroup M; module N = Quasigroup N
```

```
loop : Loop a ℓ₁ → Loop b ℓ₂ → Loop (a ⊔ b) (ℓ₁ ⊔ ℓ₂)
loop M N = record
  { ϵ = M.ϵ , N.ϵ
  ; isLoop = record
    { isQuasigroup = Quasigroup.isQuasigroup
                  (quasigroup M.quasigroup N.quasigroup)
    ; identity = (M.identityˡ , N.identityˡ <*>_),
                      (M.identityʳ , N.identityʳ <*>_)
    }
  } where module M = Loop M; module N = Loop N
```

5.5 Properties

In this section we prove some of the properties of quasigroups and loops in Agda.

5.5.1 Properties of Quasigroup

Let (Q, ·, /, \) be a quasigroup then

1. Q is cancellative. A quasigroup is left cancellative if x · y = x · z then y = z and a quasigroup is right cancellative if y · x = z · x then y = z. A quasigroup is cancellative if it is both left and right cancellative.
2. $\forall x, y, z \in Q$: x · y = z then y = x \z
3. $\forall x, y, z \in Q$: x · y = z then x = z / y

Proof:

1.
```
cancelˡ : LeftCancellative _·_
cancelˡ x {y} {z} eq = begin
  y              ≈⟨ sym( leftDividesʳ  x y) ⟩
  x \\ (x · y)   ≈⟨ \\-congˡ eq ⟩
  x \\ (x · z)   ≈⟨ leftDividesʳ  x z ⟩
  z              ∎

cancelʳ : RightCancellative _·_
cancelʳ {x} y z eq = begin
  y              ≈⟨ sym( rightDividesʳ  x y) ⟩
  (y · x) // x   ≈⟨ //-congʳ  eq ⟩
  (z · x) // x   ≈⟨ rightDividesʳ  x z ⟩
  z              ∎

cancel : Cancellative _·_
cancel = cancelˡ , cancelʳ
```

2.
```
y≈x\z : ∀ x y z → x · y ≈ z → y ≈ x \\  z
y≈x\z x y z eq = begin
  y              ≈⟨ sym (leftDividesʳ x y) ⟩
  x \\  (x · y) ≈⟨ \\ -congˡ eq ⟩
  x \\  z        ∎
```

3. ```
   x≈z/y : ∀ x y z → x · y ≈ z → x ≈ z // y
   x≈z/y x y z eq = begin
     x             ≈⟨ sym (rightDividesʳ y x) ⟩
     (x · y) // y ≈⟨ //-congʳ eq ⟩
     z // y       ∎
   ```

5.5.2 Properties of Loop

Properties of division operation holds for a loop.

Let (L, ·, /, \) be a Loop with identity x · e = x then the following properties holds

1. $\forall x \in L$: x / x = e
2. $\forall x \in L$: x \x = e
3. $\forall x \in L$: e \x = x
4. $\forall x \in L$: x / e = x

Proof:

1. ```
   x//x≈ϵ : ∀ x → x // x ≈ ϵ
   x//x≈ϵ x = begin
     x // x        ≈⟨ //-congʳ (sym (identityˡ x)) ⟩
     (ϵ · x) // x  ≈⟨ rightDividesʳ x ϵ ⟩
     ϵ             ∎
   ```

2. ```
x\\x≈ε : ∀ x → x \\ x ≈ ε
   x\\x≈ε x = begin
     x \\ x   ≈⟨ \\-congˡ (sym (identityʳ x)) ⟩
     x \\ (x · ε) ≈⟨ leftDividesʳ x ε ⟩
     ε   ∎
```

3. ```
ε\\x≈x : ∀ x → ε \\ x ≈ x
   ε\\x≈x x = begin
     ε \\ x         ≈⟨ sym (identityˡ (ε \\ x)) ⟩
     ε · (ε \\ x)   ≈⟨ leftDividesˡ ε x ⟩
     x              ∎
```

4. ```
x//ε≈x : ∀ x → x // ε ≈ x
   x//ε≈x x = begin
    x // ε   ≈⟨ sym (identityʳ (x // ε)) ⟩
    (x // ε) · ε   ≈⟨ rightDividesˡ ε x ⟩
    x   ∎
```

### 5.5.3 Properties of Middle bol loop

Let (M, ·, /, \) be a middle bol loop then the following identities holds.

1. $\forall xyz \in M$: x · ((y · x) \x) = y \x
2. $\forall xyz \in M$: x · ((x · z) \x) = x // z
3. $\forall xyz \in M$: x · (z \x) ≈ (x / z) · x
4. $\forall xyz \in M$: (x / (y · z)) · x ≈ (x / z) · (y \x)
```

5. $\forall xyz \in M$: (x / (y · x)) · x ≈ y \x

6. $\forall xyz \in M$: (x / (x · z)) · x ≈ x / z

Proof:

1. `xyx\x≈y\\x : ∀ x y → x · ((y · x) \\ x) ≈ y \\ x`

```
xyx\x≈y\\x x y = begin
  x · ((y · x) \\ x)   ≈⟨ middleBol x y x ⟩
  (x // x) · (y \\ x) ≈⟨ ·-congʳ (x//x≈ϵ x) ⟩
  ϵ · (y \\ x)        ≈⟨ identityˡ ((y \\ x)) ⟩
  y \\ x              ■
```

2. `xxz\\x≈x//z : ∀ x z → x · ((x · z) \\ x) ≈ x // z`

```
xxz\\x≈x//z x z = begin
  x · ((x · z) \\ x)  ≈⟨ middleBol x x z ⟩
  (x // z) · (x \\ x) ≈⟨ ·-congˡ (x\\x≈ϵ x) ⟩
  (x // z) · ϵ        ≈⟨ identityʳ ((x // z)) ⟩
  x // z              ■
```

3. `xz\\x≈x//zx : ∀ x z → x · (z \\ x) ≈ (x // z) · x`

```
xz\\x≈x//zx x z = begin
  x · (z \\ x)       ≈⟨ ·-congˡ (\\-congʳ( sym (identityˡ z))) ⟩
  x · ((ϵ · z) \\ x) ≈⟨ middleBol x ϵ z ⟩
  x // z · (ϵ \\ x) ≈⟨ ·-congˡ (ϵ\\x≈x x) ⟩
  x // z · x        ■
```

4. ```
x//yzx≈x//zy\\x : ∀ x y z → (x // (y · z)) · x ≈ (x // z) · (y \\ x)
x//yzx≈x//zy\\x x y z = begin
 (x // (y · z)) · x ≈⟨ sym (xz\\x≈x//zx x ((y · z))) ⟩
 x · ((y · z) \\ x) ≈⟨ middleBol x y z ⟩
 (x // z) · (y \\ x) ■
```

5. ```
x//yxx≈y\\x : ∀ x y → (x // (y · x)) · x ≈ y \\ x
x//yxx≈y\\x x y = begin
  (x // (y · x)) · x  ≈⟨ x//yzx≈x//zy\\x  x y x ⟩
  (x // x) · (y \\ x) ≈⟨ ·-congʳ(x//x≈ϵ x) ⟩
  ϵ · (y \\ x)        ≈⟨ identityˡ ((y \\ x)) ⟩
  y \\ x              ■
```

6. ```
x//xzx≈x//z : ∀ x z → (x // (x · z)) · x ≈ x // z
x//xzx≈x//z x z = begin
  (x // (x · z)) · x ≈⟨ x//yzx≈x//zy\\x x x z ⟩
  (x // z) · (x \\ x) ≈⟨ ·-congˡ (x\\x≈ϵ x) ⟩
  (x // z) · ϵ   ≈⟨ identityʳ (x // z) ⟩
  x // z   ■
```

### 5.5.4 Properties of Moufang Loop

Let $(M, \cdot, /, \backslash)$ be a moufang loop then the following identities holds.

1. Moufang loop is alternative. A moufang loop is left alternative if it satisfies $(x \cdot x) \cdot y = x \cdot (x \cdot y)$, a moufang loop is right alternative if it satisfies $x \cdot (y \cdot y) = (x \cdot y) \cdot y$ and if a moufang loop alternative if it is both left and right alternative.
```

2. Moufang loop is flexible. A Moufant loop is flexible if it satisfies flexible identity (x · y) · x = x · (y · x)

3. $\forall xyz \in M$: z · (x · (z · y)) = ((z · x) · z) · y

4. $\forall xyz \in M$: x · (z · (y · z)) = ((x · z) · y) · z

5. $\forall xyz \in M$: z · ((x · y) · z) = (z · (x · y)) · z

Proof:

1.
```
alternativeˡ : LeftAlternative _·_
alternativeˡ x y = begin
  (x · x) · y        ≈⟨ ·-congʳ (·-congˡ (sym (identityˡ x))) ⟩
  (x · (ε · x)) · y ≈⟨ sym (leftBol x ε y) ⟩
  x · (ε · (x · y)) ≈⟨ ·-congˡ (identityˡ ((x · y))) ⟩
  x · (x · y)       ∎

alternativeʳ : RightAlternative _·_
alternativeʳ x y = begin
  x · (y · y)          ≈⟨ ·-congˡ (·-congʳ(sym (identityʳ y))) ⟩
  x · ((y · ε) · y)   ≈⟨ sym (rightBol y ε x) ⟩
  ((x · y) · ε ) · y  ≈⟨ ·-congʳ (identityʳ ((x · y))) ⟩
  (x · y) · y          ∎

alternative : Alternative _·_
alternative = alternativeˡ , alternativeʳ
```

2. flex : Flexible _·_

```
flex x y = begin
  (x · y) · x       ≈⟨ ·-congˡ (sym (identityˡ x)) ⟩
  (x · y) · (ϵ · x) ≈⟨ identical y ϵ x ⟩
  x · ((y · ϵ) · x) ≈⟨ ·-congˡ (·-congʳ (identityʳ y)) ⟩
  x · (y · x)       ■
```

3. z·xzy≈zxz·y : ∀ x y z → (z · (x · (z · y))) ≈ (((z · x) · z) · y)

```
z·xzy≈zxz·y x y z = sym (begin
  ((z · x) · z) · y ≈⟨ (·-congʳ  (flex z x )) ⟩
  (z · (x · z)) · y ≈⟨ sym (leftBol z x y) ⟩
  z · (x · (z · y)) ■)
```

4. x·zyz≈xzy·z : ∀ x y z → (x · (z · (y · z))) ≈ (((x · z) · y) · z)

```
x·zyz≈xzy·z x y z = begin
  x · (z · (y · z))  ≈⟨ (·-congˡ  (sym (flex z y ))) ⟩
  x · ((z · y) ·  z) ≈⟨ sym (rightBol z y x) ⟩
  ((x · z) · y) · z  ■
```

5. z·xyz≈zxy·z : ∀ x y z → (z · ((x · y) · z)) ≈ ((z · (x · y)) · z)

```
z·xyz≈zxy·z x y z = sym (flex z (x · y))
```

Chapter 6

Theory of Semigroup and Ring in Agda

In early 20th century, mathematician Hilbert proposed the H_{10} problem that argues if there exists a useful approach to verify whether a general Diophantine equation is solvable. Larchey-Wendling and Forster (2020). Although this problem was solved by 1970, In 1987 Siekmann and Szabo concluded that the unification problem of D_A-rewriting system cannot be predicted. In Deng et al. (2016) the author proposes a type (2,2,0) algebra that is a semigroup that is a general construct of D_A-rewriting system. Other applications of semigroup in finite automata systems, probability theory and partial differential equations are explored in Liaqat and Younas (2021).

Ring is an algebraic structure has its applications in various branch if studies. Ring structures are studied in number theory unknown (2022a), quantum computing Netto et al. (2008), in cryptography unknown (2022b) and many other fields. More variations of rings such as nearring, quasiring, ideal ring are being explored to make ring theory more dynamic, concrete and useable. The aim of this chapter is to define these structures and prove some of the most commonly used properties in the standard library that can help build other systems that uses these structures.

6.1 Definition

A set that has a binary operation is called Magma. Magma with associative property is called a semigroup. For binary operation · on a set S, the associative property is defined as

$$\forall x y z \in S : x \cdot (y \cdot z) = (x \cdot y) \cdot z. \tag{6.1.1}$$

A semigroup that satisfies commutative property is called commutative semigroup. For binary operation · on a set S, commutative property is defined as

$$\forall x y \in S : x \cdot y = y \cdot x \tag{6.1.2}$$

The definition of associative and commutative property in Agda.

```
Associative : Op₂ A → Set _
Associative _·_ = ∀ x y z → ((x · y) · z) ≈ (x · (y · z))


Commutative : Op₂ A → Set _
Commutative _·_ = ∀ x y → (x · y) ≈ (y · x)
```

The Semigroup structure can be structurally derived from Magma in Agda as

```
record IsSemigroup (· : Op₂ A) : Set (a ⊔ ℓ) where
  field
    isMagma : IsMagma ·
    assoc   : Associative ·

  open IsMagma isMagma public
```

Similarly commutative semigroup can be derived from semigroup as

```
record IsCommutativeSemigroup (· : Op₂ A) : Set (a ⊔ ℓ) where
  field
    isSemigroup : IsSemigroup ·
    comm        : Commutative ·

  open IsSemigroup isSemigroup public
```

Semigroup and commutative semigroup structure definitions with direct product and morphism constructs were previously defined in agda standard library and hence will not be discussed in details in this chapter.

Non associative ring is an algebraic structure with two binary operations addition and multiplication. Addition is an abelian group and multiplication is unital magma and multiplication distributes over addition. A unital magma is a magma with identity element.

```
record IsNonAssociativeRing (+ * : Op₂ A) (-_ : Op₁ A) (0# 1# : A) : Set (a ⊔ ℓ) w
  field
    +-isAbelianGroup : IsAbelianGroup + 0# -_
```

```
    *-cong          : Congruent₂ *
    identity        : Identity 1# *
    distrib         : * DistributesOver +
    zero            : Zero 0# *

  open IsAbelianGroup +-isAbelianGroup public
```

A quasiring is a type (2,2) algebraic structure for which both addition and multiplication is a monoid and multiplication distributes over addition.

```
record IsQuasiring (+ * : Op₂ A) (0# 1# : A) : Set (a ⊔ ℓ) where
  field
    +-isMonoid    : IsMonoid + 0#
    *-cong        : Congruent₂ *
    *-assoc       : Associative *
    *-identity    : Identity 1# *
    distrib       : * DistributesOver +
    zero          : Zero 0# *

  open IsMonoid +-isMonoid public
```

A quasiring with additive inverse is called a nearring. This implies that for nearring addition is a group and multiplication is a monoid and multiplication distributes over addition.

```
record IsNearring (+ * : Op₂ A) (0# 1# : A) (_⁻¹ : Op₁ A) : Set (a ⊔ ℓ) where
  field
```

```
    isQuasiring : IsQuasiring + * 0# 1#
    +-inverse   : Inverse 0# _⁻¹ +
    ⁻¹-cong     : Congruent₁ _⁻¹

  open IsQuasiring isQuasiring public
```

Ring without one or rig or ring without unit is an algebraic structure with two binary operations such that addition is an abelian group and multiplication is a semigroup and multiplication distributes over addition Ring is rig with identity.

```
record IsRingWithoutOne (+ * : Op₂ A) (-_ : Op₁ A) (0# : A) : Set (a ⊔ ℓ) where
  field
    +-isAbelianGroup : IsAbelianGroup + 0# -_
    *-cong           : Congruent₂ *
    *-assoc          : Associative *
    distrib          : * DistributesOver +
    zero             : Zero 0# *

  open IsAbelianGroup +-isAbelianGroup public
```

6.2 Morphism

A structure preserving map between two structures is called morphism. In this section morphism of ring without one is given. Morphisms of other structures that are structurally different from ring without one discussed in this section can be found in agda standard library

```
record IsRingWithoutOneHomomorphism (⟦_⟧ : A → B) : Set (a ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    +-isGroupHomomorphism : +.IsGroupHomomorphism ⟦_⟧
    *-homo : Homomorphic₂ ⟦_⟧ _*₁_ _*₂_

  open +.IsGroupHomomorphism +-isGroupHomomorphism public
    renaming (homo to +-homo; ε-homo to 0#-homo;
              isMagmaHomomorphism to +-isMagmaHomomorphism)
```

A Homomorphism that is injective is called monomorphism

```
record IsRingWithoutOneMonomorphism (⟦_⟧ : A → B) : Set (a ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    isRingWithoutOneHomomorphism : IsRingWithoutOneHomomorphism ⟦_⟧
    injective                    : Injective ⟦_⟧

  open IsRingWithoutOneHomomorphism isRingWithoutOneHomomorphism public
```

A monomorphism that is bijective is called an isomorphism

```
record IsRingWithoutOneMonomorphism (⟦_⟧ : A → B) : Set (a ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    isRingWithoutOneHomomorphism : IsRingWithoutOneHomomorphism ⟦_⟧
    injective                    : Injective ⟦_⟧

  open IsRingWithoutOneHomomorphism isRingWithoutOneHomomorphism public
```

6.3 Morphism composition

If f is a morphism such that f : a → b and g is a morphism on same structure such that g : b → c then composition of morphism can be defined as g ∘ f : a → c.

```
isRingWithoutOneHomomorphism
  : IsRingWithoutOneHomomorphism R₁ R₂ f
  → IsRingWithoutOneHomomorphism R₂ R3 g
  → IsRingWithoutOneHomomorphism R₁ R3 (g ∘ f)
isRingWithoutOneHomomorphism f-homo g-homo = record
  { +-isGroupHomomorphism = isGroupHomomorphism ≈3-trans
             F.+-isGroupHomomorphism G.+-isGroupHomomorphism
  ; *-homo               = λ x y → ≈3-trans
            (G.⟦⟧-cong (F.*-homo x y)) (G.*-homo (f x) (f y))
  } where module F = IsRingWithoutOneHomomorphism f-homo;
             module G = IsRingWithoutOneHomomorphism g-homo
```

6.4 Direct Product

The direct product M × N of two ring without one structures M and N is defined as a pair (m,n) where m ∈ M and n ∈ N.

```
ringWithoutOne : RingWithoutOne a ℓ₁ →
               RingWithoutOne b ℓ₂ → RingWithoutOne (a ⊔ b) (ℓ₁ ⊔ ℓ₂)
ringWithoutOne R S = record
  { isRingWithoutOne = record
     { +-isAbelianGroup = AbelianGroup.isAbelianGroup
```

```
                 ((abelianGroup R.+-abelianGroup S.+-abelianGroup))
      ; *-cong           = Semigroup.·-cong
                 (semigroup R.*-semigroup S.*-semigroup)
      ; *-assoc   = Semigroup.assoc (semigroup R.*-semigroup S.*-semigroup)
      ; distrib     = (λ x y z →
                (R.distribˡ , S.distribˡ) <*> x <*> y <*> z)
                           , (λ x y z →
                (R.distribʳ , S.distribʳ) <*> x <*> y <*> z)
      ; zero     = uncurry (λ x y → R.zeroˡ x , S.zeroˡ y)
                           , uncurry (λ x y → R.zeroʳ x , S.zeroʳ y)
      }


  } where module R = RingWithoutOne R; module S = RingWithoutOne S
```

The direct product M × N of two non associative ring a structures M and N is defined as a pair (m,n) where m ∈ M and n ∈ N.

```
nonAssociativeRing : NonAssociativeRing a ℓ₁ →
                 NonAssociativeRing b ℓ₂ → NonAssociativeRing (a ⊔ b) (ℓ₁ ⊔ ℓ₂)
nonAssociativeRing R S = record
  { isNonAssociativeRing = record
      { +-isAbelianGroup = AbelianGroup.isAbelianGroup
                ((abelianGroup R.+-abelianGroup S.+-abelianGroup))
      ; *-cong           = UnitalMagma.·-cong
                (unitalMagma R.*-unitalMagma S.*-unitalMagma)
      ; *-identity       = UnitalMagma.identity
```

```
                (unitalMagma R.*-unitalMagma S.*-unitalMagma)
      ; distrib          = (λ x y z →
                (R.distribˡ , S.distribˡ) <*> x <*> y <*> z)
                           , (λ x y z →
                (R.distribʳ , S.distribʳ) <*> x <*> y <*> z)
      ; zero             = uncurry (λ x y → R.zeroˡ x , S.zeroˡ y)
                           , uncurry (λ x y → R.zeroʳ x , S.zeroʳ y)
      }

  } where module R = NonAssociativeRing R; module S = NonAssociativeRing S
```

The direct product M × N of two quasiring a structures M and N is defined as a pair (m,n) where m ∈ M and n ∈ N.

```
quasiring : Quasiring a ℓ₁ →
                 Quasiring b ℓ₂ → Quasiring (a ⊔ b) (ℓ₁ ⊔ ℓ₂)
quasiring R S = record
  { isQuasiring = record
      { +-isMonoid = Monoid.isMonoid
                 ((monoid R.+-monoid S.+-monoid))
      ; *-cong           = Monoid.·-cong
                 (monoid R.*-monoid S.*-monoid)
      ; *-assoc          = Monoid.assoc
                 (monoid R.*-monoid S.*-monoid)
      ; *-identity       = Monoid.identity
                 ((monoid R.*-monoid S.*-monoid))
```

```
      ; distrib          = (λ x y z →
                 (R.distribˡ , S.distribˡ) <*> x <*> y <*> z)
                           , (λ x y z →
                 (R.distribʳ , S.distribʳ) <*> x <*> y <*> z)
      ; zero             = uncurry (λ x y → R.zeroˡ x , S.zeroˡ y)
                           , uncurry (λ x y → R.zeroʳ x , S.zeroʳ y)
      }

  } where module R = Quasiring R; module S = Quasiring S
```

The direct product M × N of two nearring a structures M and N is defined as a pair (m,n) where m ∈ M and n ∈ N.

```
nearring : Nearring a ℓ₁ →
                Nearring b ℓ₂ → Nearring (a ⊔ b) (ℓ₁ ⊔ ℓ₂)
nearring R S =  record
  { isNearring = record
      { isQuasiring = Quasiring.isQuasiring
                 (quasiring R.quasiring S.quasiring)
      ; +-inverse   = (λ x → (R.+-inverseˡ , S.+-inverseˡ) <*> x)
                    , (λ x → (R.+-inverseʳ , S.+-inverseʳ) <*> x)
      ; ⁻¹-cong     = map R.⁻¹-cong S.⁻¹-cong
      }
  } where module R = Nearring R; module S = Nearring S
```

6.5 Properties

This section only provides proof for properties that was contributed by the author and other properties can be found in agda standard library.

6.5.1 Properties of Semigroup

Let (S, ·) be a semigroup then

1. S is alternative. The Semigroup S left alternative if $\forall x, y \in S : (x \cdot x) \cdot y = x \cdot (x \cdot y)$ and right alternative is $\forall x, y \in S : x \cdot (y \cdot y) = (x \cdot y) \cdot y$. Semigroup is said to be alternative if it is both left and right alternative.

2. S is flexible. The Semigroup S is flexible if $\forall x y \in S : x \cdot (y \cdot x) = (x \cdot y) \cdot x$

3. S has Jordan identity. Jordan identity for binary operation · can be defined on set S as $\forall x y z \in S : (x \cdot y) \cdot (x \cdot x) = x \cdot (y \cdot (x \cdot x))$

Proof:

1.
```
alternativeˡ : LeftAlternative _·_
alternativeˡ x y = assoc x x y

alternativeʳ : RightAlternative _·_
alternativeʳ x y = sym (assoc x y y)

alternative : Alternative _·_
alternative = alternativeˡ , alternativeʳ
```

2.
```
flexible : Flexible _·_
flexible x y = assoc x y x
```

3.
```
xy·xx≈x·yxx : ∀ x y → (x · y) · (x · x) ≈ x · (y · (x · x))
xy·xx≈x·yxx x y = assoc x y ((x · x))
```

6.5.2 Properties of Commutative Semigroup

Let (S, ·) be a commutative semigroup then

1. S is semimedial. The commutative semigroup S is left semimedial if $\forall x y z \in S$: $(x \cdot x) \cdot (y \cdot z) = (x \cdot y) \cdot (x \cdot z)$ and right semimedial if $\forall x y z \in S : (y \cdot z) \cdot (x \cdot x) = (y \cdot x) \cdot (z \cdot x)$. A structure is semimedial if it is both left and right semimedial.

2. S is middle semimedia. The commutative semigroup S is middle semimedial if $\forall x y z \in S : (x \cdot y) \cdot (z \cdot x) = (x \cdot z) \cdot (y \cdot x)$

Proof:

1.
```
semimedialˡ : LeftSemimedial _·_
semimedialˡ x y z = begin
  (x · x) · (y · z) ≈⟨ assoc x x (y · z) ⟩
  x · (x · (y · z)) ≈⟨ ·-congˡ (sym (assoc x y z)) ⟩
  x · ((x · y) · z) ≈⟨ ·-congˡ (·-congʳ (comm x y)) ⟩
  x · ((y · x) · z) ≈⟨ ·-congˡ (assoc y x z) ⟩
  x · (y · (x · z)) ≈⟨ sym (assoc x y ((x · z))) ⟩
  (x · y) · (x · z) ∎

semimedialʳ : RightSemimedial _·_
semimedialʳ x y z = begin
  (y · z) · (x · x) ≈⟨ assoc y z (x · x) ⟩
  y · (z · (x · x)) ≈⟨ ·-congˡ (sym (assoc z x x)) ⟩
  y · ((z · x) · x) ≈⟨ ·-congˡ (·-congʳ (comm z x)) ⟩
  y · ((x · z) · x) ≈⟨ ·-congˡ (assoc x z x) ⟩
  y · (x · (z · x)) ≈⟨ sym (assoc y x ((z · x))) ⟩
  (y · x) · (z · x) ∎

semimedial : Semimedial _·_
semimedial = semimedialˡ , semimedialʳ
```

2.
```
middleSemimedial : ∀ x y z → (x · y) · (z · x) ≈ (x · z) · (y · x)
middleSemimedial x y z = begin
  (x · y) · (z · x) ≈⟨ assoc x y ((z · x)) ⟩
  x · (y · (z · x)) ≈⟨ ·-congˡ (sym (assoc y z x)) ⟩
  x · ((y · z) · x) ≈⟨ ·-congˡ (·-congʳ (comm y z)) ⟩
  x · ((z · y) · x) ≈⟨ ·-congˡ ( assoc z y x) ⟩
  x · (z · (y · x)) ≈⟨ sym (assoc x z ((y · x))) ⟩
  (x · z) · (y · x) ∎
```

6.5.3 Properties of Ring without one

Let (R, +, *, -, 0) be ring without one structure then:

1. $\forall x, y \in R$: - (x * y) = - x * y

2. $\forall x, y \in R$: - (x * y) = x * - y

proof:

1.
```
-‿distribˡ-* : ∀ x y → - (x * y) ≈ - x * y
-‿distribˡ-* x y = sym $ begin
  - x * y
        ≈⟨ sym $ +-identityʳ (- x * y) ⟩
  - x * y + 0#
        ≈⟨ +-congˡ $ sym ( -‿inverseʳ (x * y) ) ⟩
  - x * y + (x * y + - (x * y))
```

```
        ≈⟨ sym $ +-assoc (- x * y) (x * y) (- (x * y)) ⟩
  - x * y + x * y + - (x * y)
        ≈⟨ +-congʳ $ sym ( distribʳ y (- x) x ) ⟩
  (- x + x) * y + - (x * y)
        ≈⟨ +-congʳ $ *-congʳ $ -‿inverseˡ x ⟩
  0# * y + - (x * y)
        ≈⟨ +-congʳ $ zeroˡ y ⟩
  0# + - (x * y)
        ≈⟨ +-identityˡ (- (x * y)) ⟩
  - (x * y)
        ∎
```

2.
```
-‿distribʳ-* : ∀ x y → - (x * y) ≈ x * - y
-‿distribʳ-* x y = sym $ begin
  x * - y
        ≈⟨ sym $ +-identityˡ (x * (- y)) ⟩
  0# + x * - y
        ≈⟨ +-congʳ $ sym ( -‿inverseˡ (x * y) ) ⟩
  - (x * y) + x * y + x * - y
        ≈⟨ +-assoc (- (x * y)) (x * y) (x * (- y)) ⟩
  - (x * y) + (x * y + x * - y)
        ≈⟨ +-congˡ $ sym ( distribˡ x y ( - y) )  ⟩
  - (x * y) + x * (y + - y)
        ≈⟨ +-congˡ $ *-congˡ $ -‿inverseʳ y ⟩
  - (x * y) + x * 0#
```

```
        ≈⟨ +-congˡ $ zeroʳ x ⟩
  - (x * y) + 0#
        ≈⟨ +-identityʳ (- (x * y)) ⟩
  - (x * y)
        ∎
```

6.5.4 Properties of Ring

Let (R, +, *, -, 0, 1) be a ring structure then

1. $\forall x \in R$: - 1 * x = -x
2. $\forall x \in R$: if x + x = 0 then x = 0
3. $\forall x, y, z \in R$: x * (y - z) = x * y - x * z
4. $\forall x, y, z \in R$: (y - z) * x = (y * x) - (z * x)

Proof:

1. ```
   -1*x≈-x : ∀ x → - 1# * x ≈ - x
   -1*x≈-x x = begin
     - 1# * x     ≈⟨ sym (-‿distribˡ-* 1# x ) ⟩
     - (1# * x)   ≈⟨ -‿cong ( *-identityˡ x ) ⟩
     - x          ∎
   ```

2. ```
   x+x≈x⇒x≈0 : ∀ x → x + x ≈ x → x ≈ 0#
   x+x≈x⇒x≈0 x eq = begin
     x ≈⟨ sym(+-identityʳ x) ⟩
   ```

```
    x + 0# ≈⟨ +-congˡ (sym (-‿inverseʳ x)) ⟩
    x + (x - x) ≈⟨ sym (+-assoc x x (- x)) ⟩
    x + x - x ≈⟨ +-congʳ(eq) ⟩
    x - x ≈⟨ -‿inverseʳ x ⟩
    0# ∎
```

3. x[y-z]≈xy-xz : ∀ x y z → x * (y - z) ≈ x * y - x * z

```
  x[y-z]≈xy-xz x y z = begin
    x * (y - z)      ≈⟨ distribˡ x y (- z) ⟩
    x * y + x * - z  ≈⟨ +-congˡ (sym (-‿distribʳ-* x z)) ⟩
    x * y - x * z    ∎
```

4. [y-z]x≈yx-zx : ∀ x y z → (y - z) * x ≈ (y * x) - (z * x)

```
  [y-z]x≈yx-zx x y z = begin
    (y - z) * x      ≈⟨ distribʳ x y (- z) ⟩
    y * x + - z * x  ≈⟨ +-congˡ (sym (-‿distribˡ-* z x)) ⟩
    y * x - z * x    ∎
```

Chapter 7

Theory of Kleene Algebra in Agda

Kleene algebra is an algebraic structure named after Stephen Cole Kleene, for his invention of finite automata and regular expressions. Kleene algebras are used in various contexts such as relational algebra, automata and formal theory, design and analysis of algorithms and program analysis and compiler optimization Kozen (1997). Kleene algebra generalizes operations from regular expressions. The axiomization of the algebra if regular events was recently proposed in 1966 but it was in 1984, a completeness theorem for relational algebra with a proper subclass of Kleene algebra was given. Kozen (1994). Although there are some differences in axioms of kleene algebra, in this chapter we consider the axioms defined in Kozen (1994)

7.1 Definition

A set S with two binary operations + and * generally called addition and multiplication such that (S,+) is a commutative monoid, (S,*) is a monoid and + distributes over * with annhiliating zero is called a semiring. A semiring satisfying idempotent property is called idempotent semiring. A Kleene Algebra over set S is idempotent semiring with $\star$ operator that satisfies the following axioms.

$$1 + (x \cdot (x*)) \leq x* \tag{7.1.1}$$

$$1 + (x*) \cdot x \leq x* \tag{7.1.2}$$

$$\forall a, b, x \in S: \text{ If } b + a \cdot x \leq x \text{ then, } (a*) \cdot b \leq x \tag{7.1.3}$$

$$\forall a, b, x \in S: \text{ If } b + x \cdot a \leq x \text{ then, } b \cdot (a*) \leq x \tag{7.1.4}$$

In Agda, strong axioms of $\star$ are given. That is equivalance is directly given and kleene algebra with partial and pre order structures are defined in Algebra.Ordered hierarchy.

```
StarRightExpansive : A → Op₂ A → Op₂ A
        → Op₁ A → Set _
StarRightExpansive e _+_ _·_ _* = ∀ x → (e + (x · (x *))) ≈ (x *)


StarLeftExpansive : A → Op₂ A → Op₂ A
         → Op₁ A → Set _
StarLeftExpansive e _+_ _·_ _* = ∀ x →  (e + ((x *) · x)) ≈ (x *)


StarLeftDestructive : Op₂ A → Op₂ A
         → Op₁ A → Set _
StarLeftDestructive _+_ _·_ _* = ∀ a b x → (b + (a · x)) ≈ x
        → ((a *) · b) ≈ x


StarRightDestructive : Op₂ A → Op₂ A
         → Op₁ A → Set _
StarRightDestructive _+_ _·_ _* = ∀ a b x → (b + (x · a)) ≈ x
        → (b · (a *)) ≈ x
```

The Kleene algebra can be structurally derived from idempotent semiring.

```
record IsKleeneAlgebra (+ * : Op₂ A) (⋆ : Op₁ A)
                 (0# 1# : A) : Set (a ⊔ ℓ) where
  field
    isIdempotentSemiring  : IsIdempotentSemiring + * 0# 1#
    starExpansive         : StarExpansive 1# + * ⋆
    starDestructive       : StarDestructive + * ⋆

  open IsIdempotentSemiring isIdempotentSemiring public
```

The bundle version of kleene algebra is defined as:

```
record KleeneAlgebra c ℓ : Set (suc (c ⊔ ℓ)) where
  infix  8 _⋆
  infixl 7 _*_
  infixl 6 _+_
  infix  4 _≈_
  field
    Carrier               : Set c
    _≈_                   : Rel Carrier ℓ
    _+_                   : Op₂ Carrier
    _*_                   : Op₂ Carrier
    _⋆                    : Op₁ Carrier
    0#                    : Carrier
    1#                    : Carrier
    isKleeneAlgebra       : IsKleeneAlgebra _≈_ _+_ _*_ _⋆ 0# 1#
```

```
open IsKleeneAlgebra isKleeneAlgebra public

idempotentSemiring : IdempotentSemiring _ _
idempotentSemiring = record { isIdempotentSemiring = isIdempotentSemiring }

open IdempotentSemiring idempotentSemiring public
  using
  ( _≉_; +-rawMagma; +-magma; +-unitalMagma; +-commutativeMagma
  ; +-semigroup; +-commutativeSemigroup
  ; *-rawMagma; *-magma; *-semigroup
  ; +-rawMonoid; +-monoid; +-commutativeMonoid
  ; *-rawMonoid; *-monoid
  ; nearSemiring; semiringWithoutOne
  ; semiringWithoutAnnihilatingZero
  ; rawSemiring; semiring
  )
```

7.2 Direct Product

The direct product K × L of two kleene algebra structures K and NL is defined as a pair (k,l) where k∈ K and l ∈ L.

```
kleeneAlgebra : KleeneAlgebra a ℓ\textsubscript{1}
        → KleeneAlgebra b ℓ\textsubscript{2} →
         KleeneAlgebra (a ⊔ b) (ℓ\textsubscript{1} ⊔ ℓ\textsubscript{2})
```

```
kleeneAlgebra K L = record
  { isKleeneAlgebra = record
      { isIdempotentSemiring = IdempotentSemiring.isIdempotentSemiring
                (idempotentSemiring K.idempotentSemiring L.idempotentSemiring)
      ; starExpansive = (λ x → (K.starExpansiveˡ  , L.starExpansiveˡ) <*> x)
                      , (λ x → (K.starExpansiveʳ  , L.starExpansiveʳ) <*> x)
      ; starDestructive = (λ a b x x₁ →
                 (K.starDestructiveˡ  , L.starDestructiveˡ)
                 <*> a <*> b <*> x <*> x₁)
                        , (λ a b x x₁ →
                (K.starDestructiveʳ  , L.starDestructiveʳ)
                 <*> a <*> b <*> x <*> x₁)
      }
  } where module K = KleeneAlgebra K;  module L = KleeneAlgebra L
```

7.3 Properties

In this section we prove some of the properties of Kleene algebra

Let (K, +, *, ⋆, 0, 1) be a Kleene algebra then:

1. $0\star = 1$

2. $1\star = 1$

3. $\forall x \in K$: $1 + x\star = x\star$

4. $\forall x \in K$: $x + x * x\star = x\star$

5. $\forall x \in K$: x + x★ * x = x★

6. $\forall x \in K$: x + x★ = x★

7. $\forall x \in K$: 1 + x + x★ = x★

8. $\forall x \in K$: 0 + x + x★ = x★

9. $\forall x \in K$: x★ * x★ = x★

10. $\forall x \in K$: x★★ = x★

11. $\forall x, y \in K$: If x = y then, x★ = y★

12. $\forall a, b, x \in K$: If a * x = x * b then, a★ * x = x * b★

13. $\forall x, y \in K$: (x * y) ★ * x ≈ x * (y * x) ★

Proof:

1. `0★≈1 : 0# ★ ≈ 1#`

```
0★≈1 = begin
  0# ★            ≈⟨ sym (starExpansiveˡ 0#) ⟩
  1# + 0# ★ * 0#  ≈⟨ +-congˡ ( zeroʳ (0# ★)) ⟩
  1# + 0#         ≈⟨ +-identityʳ 1# ⟩
  1#              ∎
```

2. `1+11≈1 : 1# + 1# * 1# ≈ 1#`

```
1+11≈1 = begin
  1# + 1# * 1#  ≈⟨ +-congˡ ( *-identityʳ 1#) ⟩
  1# + 1#       ≈⟨ +-idem 1# ⟩
```

```
  1#             ∎
```

```
1⋆≈1 : 1# ⋆ ≈ 1#
1⋆≈1 = begin
  1# ⋆        ≈⟨ sym (*-identityʳ (1# ⋆)) ⟩
  1# ⋆ * 1#   ≈⟨ starDestructiveˡ 1# 1# 1# 1+11≈1 ⟩
  1#          ∎
```

3. `1+x⋆≈x⋆ : ∀ x → 1# + x ⋆ ≈ x ⋆`

```
1+x⋆≈x⋆ x = sym (begin
  x ⋆                    ≈⟨ sym (starExpansiveʳ x) ⟩
  1# + x * x ⋆           ≈⟨ +-congʳ (sym (+-idem 1#)) ⟩
  1# + 1# + x * x ⋆      ≈⟨ +-assoc 1# 1# ((x * x ⋆ )) ⟩
  1# + (1# + x * x ⋆)    ≈⟨ +-congˡ (starExpansiveʳ x) ⟩
  1# + x ⋆               ∎)
```

4. `x⋆+xx⋆≈x⋆ : ∀ x → x ⋆ + x * x ⋆ ≈ x ⋆`

```
x⋆+xx⋆≈x⋆ x = begin
  x ⋆ + x * x ⋆          ≈⟨ +-congʳ (sym (1+x⋆≈x⋆ x)) ⟩
  1# + x ⋆ + x * x ⋆     ≈⟨ +-congʳ (+-comm 1# ((x ⋆))) ⟩
  x ⋆ + 1# + x * x ⋆     ≈⟨ +-assoc ((x ⋆)) 1# ((x * x ⋆ )) ⟩
  x ⋆ + (1# + x * x ⋆)   ≈⟨ +-congˡ (starExpansiveʳ x) ⟩
  x ⋆ + x ⋆              ≈⟨ +-idem (x ⋆) ⟩
  x ⋆                    ∎
```

5. `x⋆+x⋆x≈x⋆ : ∀ x → x ⋆ + x ⋆ * x ≈ x ⋆`

```
x★+x★x≈x★ x = begin
  x ★ + x ★ * x          ≈⟨ +-congʳ (sym (1+x★≈x★ x)) ⟩
  1# + x ★ + x ★ * x     ≈⟨ +-congʳ (+-comm 1# (x ★)) ⟩
  x ★ + 1# + x ★ * x     ≈⟨ +-assoc (x ★) 1# (x ★ * x) ⟩
  x ★ + (1# + x ★ * x)   ≈⟨ +-congˡ (starExpansiveˡ x) ⟩
  x ★ + x ★              ≈⟨ +-idem (x ★) ⟩
  x ★                    ∎
```

6. `x+x★≈x★ : ∀ x → x + x ★ ≈ x ★`

```
x+x★≈x★ x = begin
  x + x ★                  ≈⟨ +-congˡ (sym (starExpansiveʳ x)) ⟩
  x + (1# + x * x ★)       ≈⟨ +-congʳ (sym (*-identityʳ x)) ⟩
  x * 1# + (1# + x * x ★)  ≈⟨ sym (+-assoc (x * 1#) 1# (x * x ★)) ⟩
  x * 1# + 1# + x * x ★    ≈⟨ +-congʳ (+-comm (x * 1#) 1#) ⟩
  1# + x * 1# + x * x ★    ≈⟨ +-assoc 1# (x * 1#) (x * x ★) ⟩
  1# + (x * 1# + x * x ★)  ≈⟨ +-congˡ (sym (distribˡ x 1# ((x ★)))) ⟩
  1# + x * (1# + x ★)      ≈⟨ +-congˡ (*-congˡ (1+x★≈x★ x)) ⟩
  1# + x * x ★             ≈⟨ (starExpansiveʳ x) ⟩
  x ★                      ∎
```

7. `1+x+x★≈x★ : ∀ x → 1# + x + x ★ ≈ x ★`

```
1+x+x★≈x★ x = begin
  1# + x + x ★     ≈⟨ +-assoc 1# x (x ★) ⟩
  1# + (x + x ★)   ≈⟨ +-congˡ (x+x★≈x★ x) ⟩
  1# + x ★         ≈⟨ 1+x★≈x★ x ⟩
  x ★              ∎
```

8.
```
0+x+x★≈x★ : ∀ x → 0# + x + x ★ ≈ x ★
0+x+x★≈x★ x = begin
  0# + x + x ★      ≈⟨ +-assoc 0# x (x ★) ⟩
  0# + (x + x ★)    ≈⟨ +-identityˡ ((x + x ★)) ⟩
  (x + x ★)         ≈⟨ x+x★≈x★ x ⟩
  x ★               ∎
```

9.
```
x★x★≈x★ : ∀ x → x ★ * x ★ ≈ x ★
x★x★≈x★ x = starDestructiveˡ x (x ★) (x ★) (x★+xx★≈x★ x)
```

10.
```
1+x★x★≈x★ : ∀ x → 1# + x ★ * x ★ ≈ x ★
1+x★x★≈x★ x = begin
  1# + x ★ * x ★   ≈⟨ +-congˡ (x★x★≈x★ x) ⟩
  1# + x ★         ≈⟨ 1+x★≈x★ x ⟩
  x ★              ∎
```

```
x★★≈x★ : ∀ x → (x ★) ★ ≈ x ★
x★★≈x★ x = begin
  (x ★) ★        ≈⟨ sym (*-identityʳ ((x ★) ★)) ⟩
  (x ★) ★ * 1#   ≈⟨ starDestructiveˡ (x ★) 1# (x ★) (1+x★x★≈x★ x) ⟩
  x ★            ∎
```

11.
```
x≈y⇒1+xy★≈y★ : ∀ x y → x ≈  y → 1# + x * y ★ ≈ y ★
x≈y⇒1+xy★≈y★ x y eq = begin
  1# + x * y ★  ≈⟨ +-congˡ (*-congʳ (eq)) ⟩
  1# + y * y ★  ≈⟨ starExpansiveʳ y ⟩
```

```
  y ★           ∎


x≈y⇒x★≈y★ : ∀ x y → x ≈ y → x ★ ≈ y ★
x≈y⇒x★≈y★ x y eq = begin
  x ★       ≈⟨ sym (*-identityʳ (x ★)) ⟩
  x ★ * 1#  ≈⟨ (starDestructiveˡ x 1# (y ★) (x≈y⇒1+xy★≈y★ x y eq)) ⟩
  y ★       ∎
```

12.
```
ax≈xb⇒x+axb★≈xb★ : ∀ x a b →
        a * x ≈ x * b → x + a * (x * b ★) ≈ x * b ★
ax≈xb⇒x+axb★≈xb★ x a b eq = begin
  x + a * (x * b ★)       ≈⟨ +-congˡ (sym(*-assoc a x (b ★))) ⟩
  x + a * x * b ★         ≈⟨ +-congʳ (sym (*-identityʳ x)) ⟩
  x * 1# + a * x * b ★    ≈⟨ +-congˡ (*-congʳ (eq)) ⟩
  x * 1# + x * b * b ★    ≈⟨ +-congˡ (*-assoc x b (b ★) ) ⟩
  x * 1# + x * (b * b ★)  ≈⟨ sym (distribˡ x 1# (b * b ★)) ⟩
  x * (1# + b * b ★)      ≈⟨ *-congˡ (starExpansiveʳ b) ⟩
  x * b ★                 ∎


ax≈xb⇒a★x≈xb★ : ∀ x a b → a * x ≈ x * b → a ★ * x ≈ x * b ★
ax≈xb⇒a★x≈xb★ x a b eq =
        starDestructiveˡ a x ((x * b ★)) (ax≈xb⇒x+axb★≈xb★ x a b eq)
```

13.
```
[xy]★x≈x[yx]★ : ∀ x y → (x * y) ★ * x ≈ x * (y * x) ★
[xy]★x≈x[yx]★ x y = ax≈xb⇒a★x≈xb★ x (x * y) (y * x) (*-assoc x y x)
```

Chapter 8

Problem in Programming Algebra

Algebraic structures show variations in syntax and semantics depending on the system or language in which they are defined. Each systems discussed in chapter 1 have their own style of defining structures in the standard libraries. For example, in Coq Ring is defined without multiplicative identity. However, in Agda, Ring has multiplicative identity and Rng is defined as RingWithoutOne that has no multiplicative identity. This ambiguity in naming is also seen in literature. Another example is same structure having multiple definitions like Quasigroups. Quasigroups can be defined as type(2) algebra with latin square property or as type(2,2,2) with left and right division operators. Both the definitions are equivalent but they are structurally different. This chapter identifies and classifies five important problems that arises when defining algebraic structures in proof assistant systems.

8.1 Ambiguity in naming

Ambiguity arises when something can be interpreted in more than one way. The example of quasigroup having more than one definition can give rise to a scenario of making an incorrect interpretation of the algebraic structure when it is not clearly stated. In abstract algebra and algebraic structure these scenarios can be more common. This can be attributed to lack of naming convention that is followed in naming algebraic structures and it's properties. For example Ring and Rng. Some mathematicians define Ring as an algebraic structure that is an abelian group under addition and a monoid under multiplication. This definition is also be named explicitly as ring with unit or ring with identity. Rng is defined as an algebraic structure that is an abelian group under addition and a semigroup under multiplication. The same structure is also defined as ring without identity. However, this definitions are often interchanged i.e., some mathematicians define ring as ring without identity that is multiplication has no identity or is a semigroup. This ambiguity is some time attributed to the language of origin of the algebraic structures. In this case rng is used in French where as ring in english. These confusions can be seen in literature and in online blogs where it is difficult to imply the definition of intent when they are not explicitly defined.

In Agda, ring is defined as an algebraic structure with two binary operations + and * where + is an abelian group and * is a monoid. The binary operation * distributes over + that is multiplication distributes over addition and it has a zero.

```
record IsRing (+ * : Op₂ A) (-_ : Op₁ A) (0# 1# : A) : Set (a ⊔ ℓ) where
  field
    +-isAbelianGroup : IsAbelianGroup + 0# -_
    *-cong           : Congruent₂ *
    *-assoc          : Associative *
    *-identity       : Identity 1# *
    distrib          : * DistributesOver +
    zero             : Zero 0# *

  open IsAbelianGroup +-isAbelianGroup public
```

Rng is defined as ring wihthout one where one is assumend to be multiplication identity.

```
record IsRingWithoutOne (+ * : Op₂ A) (-_ : Op₁ A) (0# : A) : Set (a ⊔ ℓ) where
  field
    +-isAbelianGroup : IsAbelianGroup + 0# -_
    *-cong           : Congruent₂ *
    *-assoc          : Associative *
    distrib          : * DistributesOver +
    zero             : Zero 0# *
```

Another example of ambiguity is Nearring. In some papers, Nearring is defined as a structure where addition is a group and multiplication is a monoid. But some mathematicians use the definition where multiplication is a semigroup. The same confusion also arises in defining semiring and rig structures. Wikipedia states that the term rig originated as a joke that it is similar to rng that is missing alphabet n and i to represent the identity does not exist for these structures. In Agda rig is defined as semiring without one where one is represents the multiplicative identity.

For axioms of structures, the names are usually invented when defining the structure. As an example when defining Kleene Algebra in Agda, starExpansive and starDestructive names were invented (inspired from what is used in literature). Due to lack of common practice many names can be coined for the same axiom.

```
record IsKleeneAlgebra (+ * : Op₂ A) ( ⁻* : Op₁ A)
                                  (0# 1# : A) : Set (a ⊔ ℓ) where
  field
    isIdempotentSemiring    : IsIdempotentSemiring + * 0# 1#
    starExpansion               : StarLeftExpansion 1# + * ⁻*
    starDestructive             : StarRightExpansion+ * ⁻*
  open IsIdempotentSemiring isIdempotentSemiring public
```

8.2 Equivalent but structurally different

Quasigroup structure is an example that can be defined in two ways. A type (2) Quasigroup can be defined as a set Q and binary operation · can be defined as that is a magma and satisfies latin square property. Quasigroup of type (2,2,2) is a structure with three binary operations, a magma for which division is always possible. Latin square property states that for each a , b in set Q there exists unique elements x , y in Q such that the following property is satisfied (Wikipedia contributors, 2022h)

$$a \cdot x = b$$

$$y \cdot a = b$$

Another definition of quasigroup is given as type (2,2,2) algebra in which for a set Q and binary operations ·, \ , / quasigroup should satisfy the below identities that implies left division and right division.

$$y = x \cdot (x \setminus y)$$

$$y = x \setminus (x \cdot y)$$

$$y = (y \,/\, x) \cdot x$$

$$y = (y \cdot x) \,/\, x$$

In Agda standard library the quasigroup is defined as type (2,2,2) algebra given below.

```
record IsQuasigroup (· \ \ // : Op₂ A) : Set (a ⊔ ℓ) where
  field
    isMagma       : IsMagma ·
    \ \-cong       : Congruent₂ \ \
    //-cong       : Congruent₂ //
    leftDivides   : LeftDivides · \\
    rightDivides  : RightDivides · //


  open IsMagma isMagma public
```

A quasigroup with signature (2) and a quasigroup with signature (2,2,2) are equivalent but are structurally different. In the algebra hierarchy, a Loop is an algebraic structure that is a quasigroup with identity. It can be observed the same problem persists through the hierarchy. If a loop is defined with a quasigroup that is type (2,2,2) algebra then it a loop structure of type (2) will be forced to be defined with sub-optimal name. One plausible solution to this problem is to define the structures in different modules and import restrict them when using. This problem of not being able to overload names for structures also affects when defining types of quasigroup or loops such as bol loop and moufang loop.

Since quasigroup is defined in terms of division operation, loop is also defined as a type (2,2,2) algebra in Agda. The definition of loop structure in Agda is given below.

```
record IsLoop (· \ \ // : Op₂ A) (ε : A) : Set (a ⊔ ℓ) where
  field
    isQuasigroup : IsQuasigroup · \ \ //
    identity     : Identity ε ·

  open IsQuasigroup isQuasigroup public
```

8.3 Redundant field in structural inheritance

Redundancy arises when there is duplication of the same field. In programming redundant of code is considered a bad practice and is usually avoided by modularising and creating functions that perform similar tasks. In algebraic structures, redundant fields can be introduced in structures that are defined in terms of two or more structures. For example semiring can be as commutative monoid under addition and a monoid under multiplication and multiplication distributes over addition. In Agda, both monoid and commutative monoid have an instance of equivalence relation. If semiring is defined in terms of commutative monoid and monoid then this definition of the semiring will have a redundant equivalence field. This redundancy can also be seen in other structures like ring, lattice, module, etc., To remove this redundant field in Agda the structure except the first is opened and expressed in terms of independent axioms that they satisfy. For example, semiring without identity or rig structure in Agda is defined as

```
record IsSemiringWithoutOne (+ * : Op₂ A) (0# : A) : Set (a ⊔ ℓ) where
  field
    +-isCommutativeMonoid : IsCommutativeMonoid + 0#
    *-cong                : Congruent₂ *
    *-assoc               : Associative *
    distrib               : * DistributesOver +
    zero                  : Zero 0# *

  open IsCommutativeMonoid +-isCommutativeMonoid public
```

From the above definition it is evident that an instance of semigroup should be constructed and is not directly available when using semiring without one structure. To overcome this problem an instance is created in the definition as follows along with near semiring structure.

```
  *-isMagma : IsMagma *
  *-isMagma = record
    { isEquivalence = isEquivalence
    ; ∙-cong        = *-cong
```

```
  }

*-isSemigroup : IsSemigroup *
*-isSemigroup = record
  { isMagma = *-isMagma
  ; assoc   = *-assoc
  }

isNearSemiring : IsNearSemiring + * 0#
isNearSemiring = record
  { +-isMonoid    = +-isMonoid
  ; *-cong        = *-cong
  ; *-assoc       = *-assoc
  ; distribʳ      = proj₂ distrib
  ; zeroˡ         = zeroˡ
  }
```

The above technique will effectively remove the redundant equivalence relation but it also fails to express the structure in terms of two or more structures that is commonly used in literature and in other systems. Agda 2.0 removed redundancy by unfolding the structure. This solution should make sure that the structure clearly exports the unfolded structure whose properties can be imported when required.

8.4 Identical structures

In abstract algebra when formalising algebraic structures from the hierarchy, same algebraic structure can be derived from two or more structures. One such example is Nearring. Nearring is an algebraic structure with two binary operations addition and multiplication. Near ring is a group under addition and is a monoid under multiplication and multiplication right distributes over addition. In this case near-ring is defined using two algebraic structures group and monoid. Other definition of near-ring can be derived using the structure quasiring. Quasiring is an algebraic structure in which addition is a monoid, multiplication is a monoid and multiplication distributes over addition. Using this definition of quasiring, near-ring can be defined as a quasiring which has additive inverse.

In Agda nearring is defined in terms of quasiring with additive inverse

```
record IsNearring (+ * : Op₂ A) (0# 1# : A) (_⁻¹ : Op₁ A) : Set (a ⊔ ℓ) where
  field
    isQuasiring : IsQuasiring + * 0# 1#
    +-inverse   : Inverse 0# _⁻¹ +
    ⁻¹-cong     : Congruent₁ _⁻¹

  open IsQuasiring isQuasiring public
```

Note that in some literature, near-ring is defined in which multiplication is a semigroup that is without identity. This can be attributed to the problem with ambiguity. It can be analysed that having two different definitions for same structure is not a good practice. If near-ring is defined using quasiring then it should also give an instance of additive group without having it to construct it when using the above formalisation. This solution might solve the problem at first but in practice this becomes tedious and can go to a point at which this can be impractical especially when formalising structures at higher level in the algebra hierarchy.

8.5 Equivalent structures

Consider the example of idempotent-commutative-monoid and bounded semilattice. It can be observed that both are essentially same structure. In this case it could be redundant to define two different structures from different hierarchy. Instead in Agda, aliasing is used. Idempotent-commutative-monoid is defined and an aliasing for bounded semilattice is given.

```
record IsIdempotentCommutativeMonoid (· : Op₂ A) (ε : A) : Set (a ⊔ ℓ) where
  field
    isCommutativeMonoid : IsCommutativeMonoid · ε
    idem                : Idempotent ·

  open IsCommutativeMonoid isCommutativeMonoid public

IsBoundedSemilattice = IsIdempotentCommutativeMonoid
module IsBoundedSemilattice · ε (L : IsBoundedSemilattice · ε) where

  open IsIdempotentCommutativeMonoid L public
```

Note that some mathematicians argue that bounded semilattice and idempotent commutative monoid are not structurally the same structures but are isomorphic to each other. We do not consider this argument in the scope of this thesis.

8.6 Mitigation using product family algebra

A product family algebra is an idempotent commutative semiring $(S, +, \cdot, 0, 1)$ where S is the set of product families.

$\forall$ a, b $\in$ S: a + b represents the choice between a and b.

$\forall$ a, b $\in$ S: a $\cdot$ b represents combinations between a and b.

0 is the additive identity that is the empty product family

1 is the multiplicative identity that is the product family containing empty product with no features.

This section provides a brief insight over feature modelling and product family algebra but developing the model that satisfies the idea is out of scope of this thesis.

The feature model is an and/or diagram as in Feature Oriented Domain Analysis (FODA) Kang et al. (1990). Single arc between two arrows represent and decomposition and double arcs between two arrows represent or decomposition of features. The root node is the algebraic structure and the leaf nodes represent the axioms or the identities that the algebraic structure satisfy. The nodes that are not leaf or root nodes represent an algebraic structure that is below the hierarchy of the root node.

The semigroup structure can be represented using algebraic structure magma with associative property. In the figure 8.1 the arrow has single arc between them. That means all the properties are essential in defining a semigroup.

Using product family algebra, a semigroup is represented as

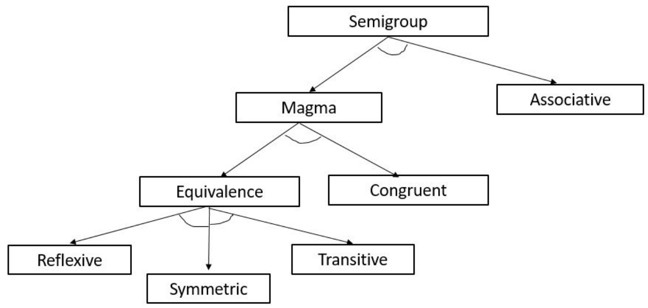

Figure 8.1: Feature diagram of semigroup

Equivalence = Reflexive · Symmetric · Transitive

Magma = Equivalence · Congruent

Semigroup = Magma · Associativity

A nearring has the problem of identical structure definition that can be defined using quasiring or group and semigroup. This figure 8.2 shows the feature diagram of nearring.

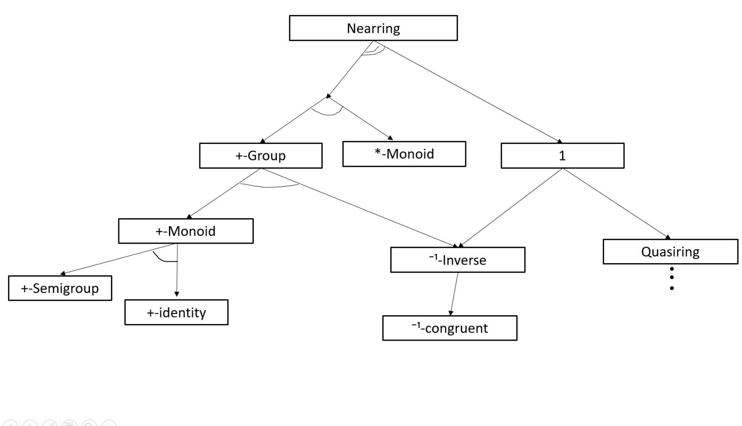

Figure 8.2: Feature diagram of nearring

Using product family algebra, nearring can be represented using above semigroup definition as

+-Monoid = +-Semigroup · +-identity

+-Group = +-Monoid · $-^1$-Inverse

Nearring = (+-Group · *-Monoid) + (Quasiring · $-^1$-Inverse)

In the above representation of the issue with identical structures, operation + represents a union. That means Nearring can have both (+-Group · *-Monoid) and (Quasiring · $-^1$-Inverse) but it will have many redundant fields. To over come this problem it is best to use XOR decomposition (filled diamond) that prevents having two full definition of the same structure. Figure 8.3 shows an example of xor decomposition when defining quasigroup The product family algebra of quasigroup can be defined as

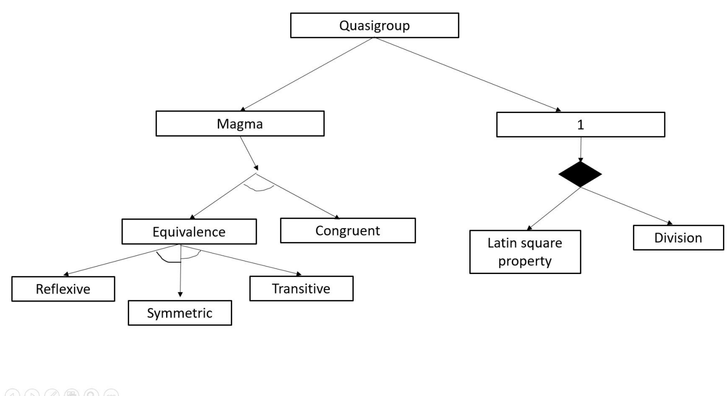

Figure 8.3: Feature diagram of quasigroup

Equivalence = Reflexive · Symmetric · Transitive

Magma = Equivalence · Congruent

Quasigroup = Magma · (Latin Square Property ⊕ Division)

In figure 8.2 there are multiple definition of same structures for different operations. In the feature diagram this can be eliminated by mentioning the operation as weight to the arrow. Figure 8.4 shows the feature diagram where the binary operation is represented at the highest possible arrow. The product family algebra of does not change as

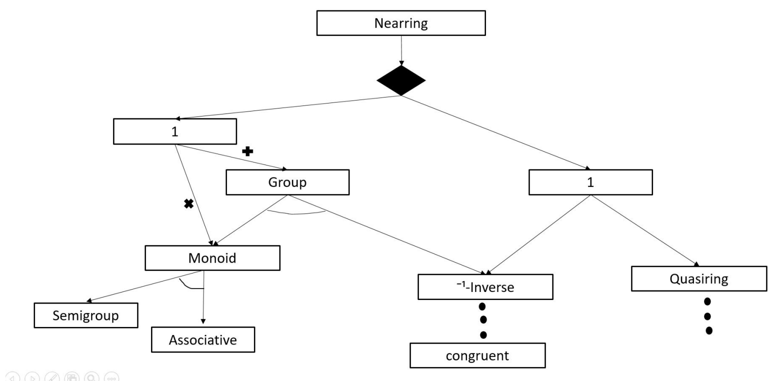

Figure 8.4: Feature diagram of Nearring

it should explicitly mention the operations of the structures.

Chapter 9

Conclusion

Every thesis also needs a concluding chapter

Appendix A

Your Appendix

Your appendix goes here.

Bibliography

2023. Universe levels. `https://agda.readthedocs.io/en/v2.6.1.3/language/universe-levels.html`

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda – A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–78.

Brilliant Math. 2023. Russell'sParadox. `https://brilliant.org/wiki/russells-paradox/` [Online; accessed 16-January-2023].

Sabine Broda, Sílvia Cavadas, and Nelma Moreira. 2014. *Kleene algebra completeness.* Technical Report. Technical report, Universidade de Porto.

Richard H Bruck. 1944. Some results in the theory of quasigroups. *Trans. Amer. Math. Soc.* 55 (1944), 19–52.

Fang-an Deng, Lu Chen, ShouHeng Tuo, and ShengZhang Ren. 2016. Characterizations of Algebras. *Algebra* 2016 (2016).

Natalia N Didurik and Victor A Shcherbacov. 2018. Some properties of Neumann quasigroups. *arXiv preprint arXiv:1809.07095* (2018).

Zoltán Ésik and Laszlo Bernátsky. 1995. Equational properties of Kleene algebras of relations with conversion. *Theoretical Computer Science* 137, 2 (1995), 237–251.

Trevor Evans. 1974. Identities and relations in commutative Moufang loops. *Journal of Algebra* 31, 3 (1974), 508–513.

Jean-Gabriel Ganascia. 1993. Algebraic structure of some learning systems. In *International Workshop on Algorithmic Learning Theory*. Springer, 398–409.

J.H. Geuvers. 2009. Proof assistants : history, ideas and future. *Sadhana : Academy Proceedings in Engineering Sciences (Indian Academy of Sciences)* 34, 1 (2009), 3–25. `https://doi.org/10.1007/s12046-009-0001-5`

Jason ZS Hu and Jacques Carette. 2021. Formalizing category theory in agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 327–342.

Russell O'Connor Jacques Carette and Yasmine Sharoda. 2019. Building on the Diamonds between Theories: Theory Presentation Combinators. *arXiv preprint arXiv:1812.08079* (2019).

Temitope Gbolahan Jaiyeola, Sunday Peter David, and Oyeyemi O Oyebola. 2021. New algebraic properties of middle Bol loops II. *Proyecciones (Antofagasta)* 40, 1 (2021), 85–106.

Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

Donnacha Oisín Kidney. 2020. Finiteness in cubical type theory. (2020).

Dexter Kozen. 1994. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and computation* 110, 2 (1994), 366–390.

Dexter Kozen. 1997. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 3 (1997), 427–443.

Kenneth Kunen. 1996. Moufang quasigroups. *Journal of Algebra* 183, 1 (1996), 231–234.

Dominique Larchey-Wendling and Yannick Forster. 2020. Hilbert's Tenth Problem in Coq (Extended Version). *arXiv preprint arXiv:2003.04604* (2020).

Iqra Liaqat and Wajeeha Younas. 2021. Some important applications of semigroups. *Journal of Mathematical Sciences & Computational Mathematics* 2, 2 (2021), 317–321.

Assia Mahboubi and Enrico Tassi. 2021. *Mathematical Components*. Zenodo. `https://doi.org/10.5281/zenodo.4457887`

The mathlib Community. 2020. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 367–381. `https://doi.org/10.1145/3372885.3373824`

Carette Jacques Farmer William M. Kohlhase Michael and Rabe Florian. 2021. Big Math and the One-Brain Barrier: The Tetrapod Model of Mathematical Knowledge. *Math Intelligencer* 43 (2021), 78–87. `https://doi.org/doi.org/10.1007/s00283-020-10006-0`

AL Silva Netto, C Chesman, and Claúdio Furtado. 2008. Influence of topology in a quantum ring. *Physics Letters A* 372, 21 (2008), 3894–3897.

Christine Paulin-Mohring. 2012. *Introduction to the Coq Proof-Assistant for Practical Software Verification*. Springer Berlin Heidelberg, Berlin, Heidelberg, 45–95. https://doi.org/10.1007/978-3-642-35746-6_3

JD Phillips and David Stanovskỳ. 2010. Automated theorem proving in quasigroup and loop theory. *Ai Communications* 23, 2-3 (2010), 267–283.

Valentin N Redko. 1964. On defining relations for the algebra of regular events. *Ukrainskii Matematicheskii Zhurnal* 16 (1964), 120–126.

Bertrand Russell. 2020. *The principles of mathematics*. Routledge.

Hanamantagouda P Sankappanavar and Stanley Burris. 1981. A course in universal algebra. *Graduate Texts Math* 78 (1981), 56.

M. Saqib Nawaz, Moin Malik, Yi Li, Meng Sun, and M. Ikram Ullah Lali. 2019. A Survey on Theorem Provers in Formal Methods. *arXiv e-prints*, Article arXiv:1912.03028 (Dec. 2019), arXiv:1912.03028 pages. arXiv:1912.03028 [cs.SE]

Mikael Stener. 2016. Moufang Loops: General theory and visualization of non-associative Moufang loops of order 16.

Abraham A Ungar. 2007. Einstein's velocity addition law and its hyperbolic geometry. *Computers & Mathematics with Applications* 53, 8 (2007), 1228–1250.

unknown. 2022a. Applications of Ring Theory. https://www.britannica.com/science/ring-mathematics. [Online; accessed 16-January-2023].

unknown. 2022b. Applications of Ring Theory. https://discover.hubpages.com/education/Applications-of-Ring-Theory [Online; accessed 16-January-2023].

Wikipedia contributors. 2022a. Agda functions. https://agda.readthedocs.io/en/v2.5.2/language/function-definitions.html [Online; accessed 21-February-2023].

Wikipedia contributors. 2022b. Agda (programming language) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Agda_(programming_language)&oldid=1127496533 [Online; accessed 21-February-2023].

Wikipedia contributors. 2022c. Axiom schema of specification — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Axiom_schema_of_specification&oldid=1125383109 [Online; accessed 28-February-2023].

Wikipedia contributors. 2022d. Constructive proof — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Constructive_proof&oldid=1090644431 [Online; accessed 22-February-2023].

Wikipedia contributors. 2022e. Intuitionism — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Intuitionism&oldid=1122615242 [Online; accessed 22-February-2023].

Wikipedia contributors. 2022f. Magma (algebra) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Magma_(algebra)&oldid=1125597787 [Online; accessed 16-January-2023].

Wikipedia contributors. 2022g. Outline of algebraic structures — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Outline_of_algebraic_structures&oldid=1107380309 [Online; accessed 16-January-2023].

Wikipedia contributors. 2022h. Quasigroup — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Quasigroup&oldid=1123964335 [Online; accessed 7-January-2023].

Wikipedia contributors. 2023a. Group (mathematics) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Group_(mathematics)&oldid=1133598242 [Online; accessed 16-January-2023].

Wikipedia contributors. 2023b. Ring (mathematics) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Ring_(mathematics)&oldid=1133737666 [Online; accessed 16-January-2023].