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:	x, 67

Abstract

Algebra is the abstract encapsulation of mathematical intuition. Proof systems can be described as inference system that has provable statements or theorems being their final products. It is important to study the intersection of these powerful concepts in mathematics and computer science by carefully defining mathematical concepts in computer language.

In this work, we study algebraic structures in proof systems especially Agda, Coq, Idris, and Lean 3 to determine the coverage of algebra in these systems and to set the scope of our research. We contribute to the Agda standard library, a proof assistant system so it can be extended to other relevant fields of algebra. We focus on commonly studied structures such as quasigroups, loops, semigroups, rings, and Kleene algebra. These structures are well-studied in universal algebra and have its applications in various fields including computer science, quantum physics, and mathematics. In the effort of defining several structures with their constructs like morphisms, and direct products and proving it's properties, we analyze five problems that arise and may not be as relevant in classical mathematics. We define more than 20 algebraic structures and add more than 40 proofs to Agda standard library.

To all my teachers
You are my greatest blessing

Acknowledgements

I would like to thank Dr. Jacques Carette for the guidance, encouragement, contributions and support he has provided during my studies as his student. Your expertise and feedback were invaluable in shaping my research. I have been very lucky to have you as my supervisor and I have learned a lot from you. I would also like to thank Dr. Ridha Khedri and Dr. Wolfram Kahl for being in my supervisory committe.

I would like to express my sincere gratitude to all the maintainers and contributors of Agda standard library. Your code review was very helpful for critical thinking and pushed me to understand the subject better.

Thanks to my parents Shanthala and Madhusudana, my siblings Utpala and Anagha for their endless love and support of me while I continue my education. Finally, Thanks to my family and friends for all the motivation and support.

Contents

List of Figures

List of Tables

Declaration of Academic Achievement

I, Akshobhya Katte Madhusudana, declare that this thesis is my own work unless otherwise stated through citations or otherwise. My supervisor Dr. Jacques Carette provided guidance and support at all stages of this work.

Major part of this thesis contributes to Agda standard library. The library aims to contain all the tools needed to write both programs and proofs easily in Agda and has been contributed to by many developers and researchers before me.

Chapter 1

Introduction

Abstract algebra is the study of algebraic structure that came into existence in early nineteenth century as complex problems and solutions evolved in other branch of mathematics such as geometry, number theory and polynomial equations. Being relatively new subject in mathematics, algebraic structures are used in various fields. For example, [2] use semigroup to study time dependent partial differential equations in technique similar to differential equations on a function space. 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 [3]. [4] uses *Quasigroups* and *loops* structures for encryption of image data. 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. Magma 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 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 [5] and requires carefully defining mathematical objects and concept. Proof assistant systems act as a bridge between computer intelligence and human effort in developing mathematical proofs. Agda, Coq, Isabelle, Lean and Idris are some commonly used proof assistant systems. Mathematicians use these proof assistants to check their proof for validity, build proofs and sometimes even generate them via proof search tools For the scope of the thesis we only discuss algebraic structures in proof systems.

1.1 Research Outline

For any software system to be robust, all its dependencies must similarly be robust. The standard libraries of these systems should support the user with all necessary functionalities to be able to use the system easily without having to define all functionalities. To generate robust libraries of knowledge, the author of [6] explore technique to generate libraries with minimum

[0] A kind of algebraic structure

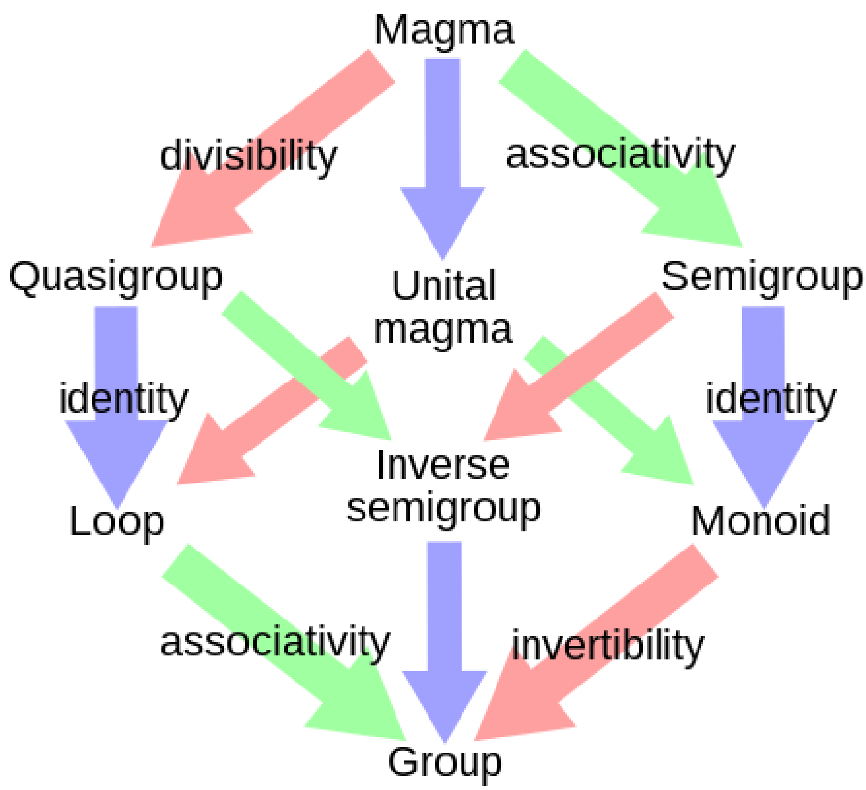

Figure 1.1: Algebraic structure hierarchy [1]

human efforts. However, while their methods do work in theory, they are difficult (and expensive) in practice. Although, generated libraries can define the algebraic concepts required, they are not fully reliable and hence not considered as "standard library" for any proof system. For now, building standard libraries for proof systems rely on human efforts. 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. [1]

As part of this thesis, more than twenty-three structures was defined in the standard library for Agda. Inspired by the ways algebraic structures are used in research, in this work we explore capturing a select subset of them in Agda standard library. Following the algebra hierarchy in Figure 1.1, we study magma with division operations that is quasigroup and loop structures. We also explore various types of loop such as bol-loop and moufang-loop and their properties. Quasigroup only have weak associative property and do not have inverse. In order to have well-defined inverse, the structure should have associative property. Semigroup is the simplest structure with associative property and are used in various fields such as probability theory and formal systems. One of the most commonly studied algebraic structure is Ring. In this thesis we study types of rings such as near-ring, quasi-ring and non-associative ring. Along with ring structure, the most used structure is Kleene algebra. The applications of Kleene algebra is seen

[1] I was exposed to Agda during course work for my Master's degree, further adding bias to choosing Agda over other systems

in finite state machines, regular expressions and other branch of computer science. As part of this thesis, we study Kleene algebra by providing proofs for its properties that may be used in developing other systems or in applications. By contributing to Agda standard library, we hope that this work will be used by others.

Notably, as we explore capturing these structures in Agda, we analyze five problems that arise:

1. Ambiguity in naming structures.
2. Equivalent structures that are structurally different.
3. Redundant field during structural inheritance.
4. Identical structures that can be derived in many ways in algebra hierarchy
5. Equivalent structures that are structurally same.

To overcome these problems we explore the use of *product family algebra*.

1.2 Thesis Outline

Chapters 2 and 3 focus on background information necessary for reading this work, focusing on reviewing universal algebra and algebraic structures in Agda, respectively. Chapter 4 justifies the scope of the thesis contribution by a survey on algebraic coverage in proof systems. The next three chapters 5,6 and 7 are dedicated to discuss the structures in details. Chapter 5 explores quasigroup and loop structures that uses division operation. Chapter 6 discusses the properties of semigroup and rings with variations of ring structure. Chapter 7 explores Kleene algebra, definition, construct and properties in Agda. Chapter 8 describes the various problems we faced during this work, as well as advice on handling common issues in programming algebras in proof systems. Finally, Chapter 9 concludes this work with notes on related future works and some closing thoughts.

Chapter 2

Universal Algebra: An Overview

By the early 18th century, mathematicians had discovered how to solve polynomial equation of up-to degree 4. While trying to find a general solution for polynomial equation, Lagrange and Abel established permutation groups [7]. Mathematician Gauss developed modular arithmetic from number theory and proved that equation $x^m - 1 = 0$ can always be solved. These developments including in the field theory of algebraic structures and geometry became the main source for *group theory* [8]. It was mathematician Evariste Galois who coined the term *group* and established group theory. He used group to determine the solvability of polynomial equations. Group theory was later discovered to be useful in various fields of mathematics such as modulus theory and geometry [1]. As group theory, the study of group structures evolved, other abstract structures were invented to solve different 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 [1]. 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. Algebraic structures, like monoids, loops, groups and rings have similar properties. Universal algebra studies these structures by abstracting out the specific definitions and properties of algebraic structures. Universal algebra will deal with these algebraic structures as axiomatic theories in equational first-order logic [9]. In the recent years, *universal algebra* has seen an exponential growth in its study of theories and applications [10]. The definitions in this chapter are adapted from [10] and [11].

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 elements or members of the set can be mathematical object of any kind such as numbers, symbols, geometrical shapes, or even other sets. If x is an object in set S then we say x is an element of S and is denoted as $x \in S$.

- The *Cartesian product* between two sets X and Y, $X \times Y$ is defined as a pair $\{(x, y) : x \in$

$X,\ y \in\ Y\}$.

- 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 *codomain*. A relation assigns elements of domain to some elements in the codomain. 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.

- When talking about a set, we discuss how different elements of the set can be related in some way. For example, in set of integers Z, we may say that some $s, y \in Z$ are related if $x - y$ is divisible by 2. In other way, x and y are related only if they are both odd or both even. This idea of expressing same relation in different way can be formalized as *equivalence relation*. A relation R is equivalence if it satisfies:

  1. Reflexive: 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
  2. Symmetric: A *symmetric relation* R on set X is a subset of $X \times X$ is defined as $R : \forall xy \in X : xRy \Longleftrightarrow yRx$
  3. Transitive: 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 $(x, z) \in R$

  In other words, 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 codomain, then we call it a *function*. In other words, function is a map f from set X (domain) to Y (codomain) is a rule that assigns each element of X to a unique element in Y. This can be expressed using the notation:

$$f :\ X \rightarrow\ Y$$

$$x\ \rightarrow\ f(x)$$

  For example, on set of natural numbers N to N we can define a function as:

$$f :\ N \rightarrow N$$

$$x\ \rightarrow\ x^2$$

- Let X and Y be two sets, and $f : X \rightarrow Y$ be the function then:

  1. The function f is *identity* if $X = Y$ and $f(x)\ =\ x,\ \forall x\ \in\ Y$. The identity function can be denoted as $f = Id_s$.
  2. A function f is *injective* if f maps distinct elements of domain to distinct elements of codomain. f is injective if $f(x) = f(y) \Rightarrow x = y \forall x, y \in X$.
  3. A function is called *surjective* if given $y \in Y$, there exists $x \in X$ such that $f(x) = y$.
  4. 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 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.

A Formal definition of algebra is given in [10] as: For A a nonempty set and n a non-negative 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 $< a_1, a_2, ..., a_n >$ under an n-ary operation f is denoted by $f(a_1, a_2, ..., a_n)$. 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.

For example, a group G is an algebra with one nullary (1), one unary ($^{-1}$) and one binary ($\cdot$) operation represented as $(G, \cdot, ^{-1}, 1)$. Here, $^{-1}$ is an inverse operation. For a binary operation $\cdot$, on set S, if x · y = e where e is an identity element, and $x, y \in S$ then x is called the left inverse of y, and y is called the right inverse of x. An element is said to have an inverse if it has both left and right inverse. The group structure should 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 ≈ 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.

2.3 Constructions

Universal algebra provides definitions of constructions related to algebraic structures. In this section, we will describe some of these constructions.

- 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 is given as, 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 $1 \le i \le n$ then $f^A(x_1, ..., x_n)\ \theta\ f^A(y_1, ..., y_n)$ holds [10].

For example, consider group structure $(G, \cdot, ^{-1}, 1)$. A congruence relation on G with binary operation $\cdot$ is an equivalence relation $\equiv$ on G such that,

$$g_1 \equiv g_2 \text{ and } h_1 \equiv h_2 \Rightarrow g_1 \cdot h_1 \equiv g_2 \cdot h_2$$

- 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:

  $$\alpha f^A(a_1, a_2, ..., a_n) \ = \ f^B(\alpha a_1, \alpha a_2, ..., \alpha a_n)$$

  For each n-ary f in F and each sequence $a_1, a_2, ..., a_n$ from A.

  As an example, consider $G_1 = \{1, -1, i, -i\}$, which is a group under multiplication, and G_2 = group of all integers under addition. A mapping f from G_1 to G_2 such that $f(x) = i^n \ \forall n \in G_2$ is a homomorphism.

  In [10], the author proves that if $\alpha : A \to B$ and $\beta : A \to B$ are homomorphism on algebra A to B such that $\alpha(a) = \beta(a)$ then $\alpha = \beta$

  Some variants of homomorphism are:

  1. Monomorphism: For two algebras A and B, if $\alpha : A \to B$ is a homomorphism from A to B, and if α satisfies one-to-one mapping (i.e., α is injective) then the morphism α is called a *monomorphism*.
  2. Isomorphism: For two algebras A and B, if $\alpha : A \to B$ is a monomorphism from A to B, and if α is a bijection from A to B, then α is called an *isomorphism*.
  3. Endomorphism: A homomorphism from an algebra A to itself is called *endomorphism*. In other words, if f is a homomorphism on A such that $f : A \to A$ then, f is endomorphism.
  4. Automorphism: An isomorphism from an algebra A to itself is called *automorphism*.
  5. Epimorphism: For two algebras A and B, if $\alpha : A \to B$ is a homomorphism from A to B, and if α is surjective then the morphism α is called a *epimorphism*.

- For algebras A, B, and C the *composition of morphisms* $f : A \to B$ and $g : B \to C$ is denoted by the function $g \circ f : A \to C$ and is defined as $(g \circ f)\ a = g\ (f\ a)$, $\forall\ a \in A$. In [10], the author proves that the composite of two homomorphism (monomorphism/isomorphism) is also a homomorphism (monomorphism/isomorphism).

- The *direct product* between two algebras A and B is defined as $A \times B = \{(a, b) : a \in A\ b \in B\}$.

Chapter 3

Agda

Agda is a dependently typed programming language based on unified theory of dependent types [12] and is an extension of Martin-Löf type theory. Dependent programming allows programmers to define types that depend on values, to write functions that utilize these types, and to prove the correctness of the program in the same language. In other words, dependent type theory allow types to depend on values and expressions. Agda has been used in various applications such as formal verification, program synthesis, theorem proving, and automated reasoning. It is also used by researchers and academician to teach and explore the concepts of functional programming, type theory, and formal methods.

Agda is also a poroof assistant system. Agda is designed to help programmers to write and verify correct and efficient programs by allowing them to express their intentions in a precise and formal way. One of the key features of Agda is its support for interactive, and constructive programming. Interactive programming allows the programmer to incrementally develop and refine their code, by testing and verifying each intermediate step. Constructive programming ensures that every expression and function in the language has a well-defined meaning and computation rules, which makes it easier to reason about their behavior and correctness. This chapter provides a brief overview of programming in Agda in the context of algebraic structures .

3.1 Types and functions

Agda is based on a core language that provides a minimal set of primitives and types, and is extended with libraries and modules that define more complex data structures, algorithms, and abstractions. Agda's type system allows for the definition of new types and operations that are tailored to the specific needs of a particular application or domain. Agda supports inductive types, simple types, and parameterized types [13]. A data type in Agda can be declared using the keyword `data`. Let us consider an example of inductive datatype to define natural numbers `Nat`.

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

An inductive datatype is a datatype that is defined in terms of itself. In the code snippet 3.1, `Nat` is an inductive type defined with base constant `zero` and an inductive data constructor `suc`. `zero` and `suc` are constructors, where `suc` has a parameter (`Nat`) and `zero` has no parameters. In this example, the smallest element is `zero`. It is important to note that Agda is a total language, i.e., each program in Agda will terminate, and all possible patterns will be matched[14]. Another way of defining a type is using the keyword `record`. Record type helps to put values together, and the values are tuples of values of specified type. 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 algebraic structure.

Those familiar with Haskell will find Agda to be somewhat familiar. For example, functions have a very similar syntax to those in Haskell. A function in Agda is defined by declaring the type followed by the clauses [15].

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

Where `f` is the function identified, `p` and `q` are the patterns of type `A`. `d` and `e` are expressions. There are other ways to define a function such as using dot patterns, absurd patterns, as patterns and case trees [15]. In Agda, a function to and from each type is provided if there is a bijection between two types.

For example, we can define addition on natural numbers as a recursive function:

```
_+_ : Nat -> Nat -> Nat
zero + m = m
suc n + m = suc (n + m)
```

In the above example, function `_+_` takes two arguments of type `Nat` and returns a value that is sum of the two arguments of type `Nat`. To guarantee that the program always terminate, a recursive call in must be made on a structurally smaller argument. For the function `_+_` above, the first argument `n` is smaller in the recursive call `suc n`. This ensures that the function `_+_` always terminates.

3.2 Structure definition

Let us first understand how unary and binary operations are defined in Agda standard library. Below code shows how unary operation $\mathtt{Op}_1$ and binary operation $\mathtt{Op}_2$ are defined:

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

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

In Agda, not every type belongs to Set. Every type belongs somewhere in the hierarchy $\texttt{Set}_0$, $\texttt{Set}_1$, $\texttt{Set}_2$, and so on. Set abbreviates $\texttt{Set}_0$, and $\texttt{Set}_0 : \texttt{Set}_1$, $\texttt{Set}_1 : \texttt{Set}_2$, and so [16]. This definition works if we are comparing two values of some type in Set. But, we cannot compare two values that belong to Set ℓ for some arbitrary ℓ. To solve this problem, Agda provides type Level. This type helps us to define equality generalized to an arbitrary level.

An algebraic structure can be defined in Agda using the record keyword, which is used to define a new data type along with its properties. The structures are obtained by wrapping the predicates that are expressed as "is-a" relation [17]. The following example shows how to define a magma structure in Agda:

```
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 a parameter $\texttt{Op}_2$ A. The properties of the structure IsMagma are declared as the fields of the record, which include equivalence isEquivalence and congruence ·-cong. · is a binary operation on the set A. a ⊔ 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. $\texttt{Congruent}_2$ · represents that the binary operation · preserves equivalence relation. IsEquivalence and $\texttt{Congruent}_2$ are predicates defined in standard library. We open the module isEquivalence to bring its definition into scope. The open statement is made public using the keyword public to be able to re-export the names from another module.

The bundled version of the structures 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. The operator with higher fixity binds more strongly than an operator with a lower numeric value. using keyword is used to limit the imported components. When exporting the modules, we may need to rename the fields to avoid having ambiguity. Keyword renaming is used to rename the fields.

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

In the sample code 3.2, we rename ·-congˡ to *-congˡ and ·-congʳ to *-congʳ thus avoiding conflict with same elements exported by other module.

3.3 Equational Proofs in Agda

In constructive mathematics, knowledge comes with implicit arguments. Constructive proofs use the existence of a mathematical object is given by giving a way to create the method. [18]. An equational proof is a sequence of steps that transform one expression into another using a set of rules. Writing proofs in Agda follows a syntax called dependent types, which allows us to declare properties of functions and data types that need to be verified by the compiler. [14].

In the previous section, we have seen how to define natural number and addition function on it. Now, we will write an inductive proof using pattern matching that states that the addition of two natural numbers is commutative.

```
comm : ∀ (m n : Nat) → m + n ≡ n + m
comm zero zero = refl
comm zero (suc n) = cong suc (comm zero n)
comm (suc m) n = cong suc (comm m n)
```

In the above example, the proof comm zero zero represents commutative property where both m and n are zero. The refl function is used to prove that two expressions are equal using the reflexivity of equality. comm zero suc n and suc m + n are reduced recursively until the base case is reached. The cong function is used to apply the inductive hypothesis to the successive suc constructors. This is just a simple example of proof, but Agda allows us to express and verify more complex properties, such as type soundness, termination, and correctness of algorithms.

In algebraic structure, consider the example to the proposition of the associative property x · (y · z) = (x · y) · z for a semigroup i.e., a Magma with associative property (x · (y · z) = (x · y) · z). The proof can be written in Agda as:

```
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 ∎
```

To make proofs more readable, people have tried to emulate textual proofs, for example, by creating "begin" and "end" syntax. `begin` indicates 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)
```

similarly, step using equality is given as

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

The termination (i.e., QED) of the proof is given using _∎ that relates object to itself.

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

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

Chapter 4

Algebraic Structures in Proof Assistant Systems - Survey

Proof assistant systems are computer software that helps to derive formal proofs with a joint effort of computers and humans. Proof assistants are used to formalize theories, and extend them by logical reasoning and defining properties[19]. 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 [6], the author discusses the difficulties in building the libraries that support these systems by providing tools to write proolfs easily. 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 proof assistant systems that are all dependently typed, higher order programming languages and supports, (at least partially) proof by reflection. Proof by reflection is a technique where the system allows to deriving 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, v1.7.1 is fully supported for the changes and developments in Agda version 2.6.2.

Coq [20] 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. [21] 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.

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

Lean [22] 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 [22] 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 [19].

The aim of this chapter is to provide documentation for the algebraic coverage in proof assistant systems Agda, Idris, Coq, and Lean. In this chapter, the latest available versions are considered i.e., Agda standard library v1.7.1, Idris 2.0, The Mathematical Components Library v1.13.0, 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 4.1 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 a 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 chapter is structured as follows. Section 2 discusses the algebraic structure definitions and their coverage in the proof assistant systems. Section 3 covers the morphism definitions. The properties and solvers are discussed in section 4.

4.2 Algebraic Structures

The Agda standard library provides definitions with bundled versions of several algebraic structures. For 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 bundled 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 [23]. 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 added as a biased structure. In Idris, some algebraic structures is provided as an extension of two other algebraic structures. However, from semigroups, in the algebra hierarchy, 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".

In Idris 2, there is a considerable overlap between abstract algebra and category theory. The library defines various algebraic structures that include semigroup, monoid, group, abeliangroup, 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 some of the simple 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 the 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
```

Note: In table 4.1, every checkmark links to the implementation in the source code of the library.

Table 4.1: Algebraic structures in proof assistant systems

Algebraic Structure	Agda	Coq	Idris	Lean
Magma	✓	-	-	-
Commutative Magma	✓	-	-	-
Selective Magma	✓	-	-	-
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	✓	-	-	-
Moufang Loop	✓	-	-	-
Left Bol Loop	✓	-	-	-
Middle Bol Loop	✓	-	-	-
Right Bol Loop	✓	-	-	-
NilpotentGroup	-	-	-	✓
CyclicGroup	-	-	-	✓
SubGroup	-	-	-	✓
Group	✓	✓	✓	✓
Abelian group	✓	-	✓	✓

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
Lattice	✓	-	-	✓
Distributive lattice	✓	-	-	-
Near semiring	✓	-	-	-
Semiringwithout one	✓	-	-	-
Idempotent Semiring	✓	-	-	-
Commutative semiring without one	✓	-	-	-
Semiring without annihilating zero	✓	-	-	-
Semiring	✓	✓	-	✓
Commutative semiring	✓	-	-	✓
Non associative ring	✓	-	-	-
Nearring	✓	-	-	-
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	✓	-	-	-
Semimodule	✓	-	-	-
Left module	✓	✓	-	-
Right module	✓	-	-	-
Bi module	✓	-	-	-
Module	✓	✓	-	✓
Field	-	✓	✓	✓
Decidable Field	-	✓	-	-
Closed field	-	✓	-	-
Algebra	-	✓	-	-

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
Unit algebra	-	✓	-	✓
Lalgebra	-	✓	-	-
Commutative unit algebra	-	✓	-	-
Commutative algebra	-	✓	-	-
NumDomain	-	✓	-	-
Normed Zmodule	-	✓	-	-
Num field	-	✓	-	-
Real domain	-	✓	-	-
Real field	-	✓	-	-
Real closed field	-	✓	-	-
Vector space	-	✓	-	-
Zmodule Quotients type	-	✓	-	-
Ring Quotient type	-	✓	-	-
Unit rint quotient type	-	✓	-	-
Additive group	-	✓	-	-
characteristic zero	-	-	-	✓
Domain	-	-	-	✓
Chain Complex	-	-	-	✓
Kleene Algebra	✓	-	-	-
HeytingCommutativeRing	✓	-	-	-
HeytingField	✓	-	-	-

4.3 Morphism

One of the benefits of the Agda standard library is that it provides morphisms for the structures defined in the library.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, and 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:

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

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 sets for propositional equalities. The library includes ring, and 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 numbers[20]. 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., the identity element of a group is self-squaring, inverse elements of a group satisfy the commutative property, and laws of double negation. It also defines 'squareIdCommutative' i.e., a group is abelian if every square in a group is neutral, inverseNeutralIsNeutral, and other properties of an algebraic group. Other algebraic properties for groups such as 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 are given in the library. 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 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 ring structure to define field 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 [24]. Quasigroups of various orders are used in field of cryptography [5]. Lie algebra is used in differential geometry[25]. With proof assistants, such as Agda, we can verify the relevant mathematical proofs of these algebraic structures. They are interactive software that help to derive complex mathematical proofs. In this chapter, we formalize two important non-associative algebras - quasigroup, and loop structure. A *quasigroup* $(Q, \cdot, /, \backslash)$ is a type (2,2,2) algebra for which the binary operations $\backslash$(left division) and / (right division) are defined such that division is always possible. A *loop* is a quasigroup with identity. In this chapter, we explore morphisms and direct product for these structures and derive proofs for some of the properties of these structures.

5.1 Definitions

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. Formally, A magma is a set S with a binary operation $\cdot$ such that, $\forall x, y \in S \Rightarrow (x \cdot y) \in S$. In Agda, magma structure is defined as `IsMagma` with binary operation $\cdot$. A quasigroup can be defined as a magma with left and right division identities. In other words, a quasigroup is a set with a binary operation that satisfies the property that for every element in the set, there is a unique element in the set that provides a solution to the equation. The operation \ (left division) and / (right division) for elements x, y in a quasigroup is defined as:

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

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

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

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

Conversely, in Agda, we may write this as:

```
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
```

Afterwards, we can form left and right divisions as:

```
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
```

In the above definition of `IsQuasigroup` is a record type with three binary operations ·, \\ // on some set A. (a ⊔ ℓ) returns the largest of two `Level` [1] (a,ℓ). The structure has five fields. `isMagma` field is used to say that the structure `IsQuasigroup` has a structure `IsMagma` with other following predicates. `\\-cong` and `//-cong` field are used to say that the division operations are congruent. `Congruent₂ <//,\\>` is used to say that the binary division operation is congruent. The division predicates are given using `leftDivides` and `rightDivides` from the definition `LeftDivides` and `RightDivides` above. We then oepn `IsMagma` as public to bring its definitions into scope.

A loop is a quasigroup that has identity element. The identity axiom is given as:

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

Conversely, in Agda, (left-right) identity is defined as:

[1] Level are used in universe polymorphism discussed in Chapter 3

```
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 ·)
```

Similar to quasigroup, 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 in general. That is $f : A \to B$ and $\cdot$ is an operation on the structure then homomorphism is defined as

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

A homomorphism that is injective is called *monomorphism*. If the structures are identical i.e., 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. Morphisms are important in understanding the relationships between different quasigroups and loops and can be used to prove important theorems about these structures.

For quasigroups $(Q_1, \cdot, \backslash\backslash, //)$ and $(Q_2, \circ, \backslash, /)$, homomorphism is defined as a structure preserving map $f : (Q_1, \cdot, \backslash\backslash, //) \to (Q_2, \circ, \backslash, /)$ such that:

- f preserves the binary operation: $f(x \cdot y) = f(x) \circ f(y)$
- f preserves the left division operation : $f(x \backslash\backslash y) = f(x) \backslash f(y)$
- f preserves the right division operation: $f(x // y) = f(x) / f(y)$

In Agda, quasigroup homomorphism can be defined as:

```
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)
```

In the above definition of quasigroup homomorphism, `Homomorphic₂` is the structure preserving map on some set A and B with binary operations $\cdot$ and $\circ$ respectively.

```
Homomorphic₂ : (A → B) → Op₂ A → Op₂ B → Set _
Homomorphic₂ ⟦_⟧ _∙_ _∘_ = ∀ x y → ⟦ x ∙ y ⟧ ≈ (⟦ x ⟧ ∘ ⟦ y ⟧)
```

In the code above, ⟦_⟧ is the map $(A \to B)$ that takes one argument. Similar to quasigroup homomorphism, quasigroup monomorphism and isomorphism can be defined as:

```
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 homomorphism preserves left and right divisions along with the identity element. The homomorphism f preserves all the binary operations as quasigroup along with the identity element. That is if $f : (L_1, \cdot, \backslash\backslash, //, e_1) \to (L_2, \circ, \backslash, /, e_2)$ is a loop homomorphism if it is a quasigroup homomorphism such that:

$$f(e_1) = e_2$$

where e_1 is the identity element of loop L_1 and e_2 is the identity element of loop L_2 In Agda, loop homomorphism can be defined using quasigroup homomorphism as:

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

  open IsQuasigroupHomomorphism isQuasigroupHomomorphism public
```

In the loop homomorphism defined above, `Homomorphic₀` is a structure preserving map for a nullary element and is defined as:

```
Homomorphic₀ : (A → B) → A → B → Set _
Homomorphic₀ ⟦_⟧ · ∘ = ⟦ · ⟧ ≈ ∘
```

Similarly, loop monomorphism and loop isomorphism are defined in Agda as:

```
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 $f : a \to b$ and g is a morphism such that $g : b \to c$, then composition of morphism can be defined as $g \circ f : a \to c$.

```
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
```

In the above quasigroup homomorphism composition, `f` is a homomorphism from quasigroup Q_1 to Q_2, g is a homomorphism from quasigroup Q_2 to Q_3. `isRelHomomorphism` field givese the composition of homomorphism for a homogeneous binary relation ($\approx$). We can prove that the composition for binary operations homomorphism ($\cdot$) for quasigroup is homomorphic using transitive relation `≈₃-trans` such that

$$g(f((Q_1 \cdot x)y)) \approx (g((Q_2 \cdot fx)(fy)) \text{ and } g((Q_2 \cdot fx)(fy))) \approx ((Q_3 \cdot g(fx))(g(fy)))$$

$$\Rightarrow g(f((Q_1 \cdot x)y)) \approx ((Q_3 \cdot g(fx))(g(fy)))$$

Similarly, composition of loop homomorphism is defined as:

```
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 Direct Product

The *direct product* $M \times N$ of two quasigroups M and N is defined as a pair (m, n) where $m \in M$ and $n \in N$. The direct product construct of left (right/middle) bol loop and moufang loop can be found in Agda standard library.

```
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
```

In the above code, zip gives a Σ-type of dependent pairs. <*> is used to convert the curried functions to a function on pair. Currying a function is to break down a function that takes multiple arguments into a series of function that takes exactly one argument. The direct product of loop structure can be defined similar to quasigroup as:

```
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 properties of quasigroup, loop, middle bol loop, and moufang loop using Agda.

5.5.1 Properties of Quasigroup

Let $(Q, \cdot, /, \backslash)$ be a quasigroup then:

1. Q is cancellative. A quasigroup is left cancellative if $x \cdot y = x \cdot z$ then $y = z$ and a quasigroup is right cancellative if $y \cdot x = z \cdot x$ then $y = z$. A quasigroup is cancellative if it is both left and right cancellative.

2. $\forall\, x,\ y,\ z \in Q$, If $x \cdot y = z$ then $y = x \setminus z$

3. $\forall\, x,\ y,\ z \in Q$, If $x \cdot y = z$ 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 $(L, \cdot, /, \setminus)$ be a Loop with identity $x \cdot e = x$ then the following properties holds

1. $\forall\, x \in L,\ x \,/\, x = e$

2. $\forall\, x \in L,\ x \setminus x = e$

3. $\forall\ x \in L,\ e \setminus x = x$

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 $(M, \cdot, /, \setminus)$ be a middle bol loop then the following identities holds.

1. $\forall\ x\ y\ z \in M,\ x \cdot ((y \cdot x) \setminus x) = y \setminus x$

2. $\forall\ x\ y\ z \in M,\ x \cdot ((x \cdot z) \setminus x) = x\ /\ z$

3. $\forall\ x\ y\ z \in M,\ x \cdot (z \setminus x) = (x\ /\ z) \cdot x$

4. $\forall\ x\ y\ z \in M,\ (x\ /\ (y \cdot z)) \cdot x = (x\ /\ z) \cdot (y \setminus x)$

5. $\forall\ x\ y\ z \in M,\ (x\ /\ (y \cdot x)) \cdot x = y \setminus x$

6. $\forall\ x\ y\ z \in M,\ (x\ /\ (x \cdot z)) \cdot x = x\ /\ z$

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 Moufang loop is flexible if it satisfies flexible identity $(x \cdot y) \cdot x = x \cdot (y \cdot x)$

3. $\forall\ x\ y\ z \in M,\ z \cdot (x \cdot (z \cdot y)) = ((z \cdot x) \cdot z) \cdot y$

4. $\forall\ x\ y\ z \in M,\ x \cdot (z \cdot (y \cdot z)) = ((x \cdot z) \cdot y) \cdot z$

5. $\forall\ x\ y\ z \in M,\ z \cdot ((x \cdot y) \cdot z) = (z \cdot (x \cdot y)) \cdot z$

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: does there exist a general approach to verify whether a general Diophantine equation is solvable[26]. Although this problem was solved by 1970, In 1987 Siekmann and Szabo concluded that the unification problem of D_A-rewriting system[27] cannot be predicted. In [28] the authors proposes a type (2,2,0) algebra that is a *semigroup* that can be used to give a general construct of D_A-rewriting system. Semigroup structures are also used in finite automata systems, probability theory and partial differential equations are explored in [2].

Similarly, *ring* is an algebraic structure that also have notable applications such as in number theory [29], quantum computing [30], in cryptography [31] and many other fields. Variations of ring structure such as a near-ring, quasi-ring, and Non-associative ring are being explored to make ring theory (study of ring structures), more dynamic, concrete and useable. Now, the question arises: how can we encode these structures in Agda, we will explore this question. The aim of this chapter is to define these structures and prove some properties in the Agda standard library that can help build other systems that uses these structures.

6.1 Definition

A magma is an algebraic structure with a set S and a binary operation $\cdot$ such that, $\forall x, y \in S \Rightarrow (x \cdot y) \in S$. Following Figure 1.1, we may observe that we can derive semigroups from magma by adding associative property. For binary operation $\cdot$ on a set S, the associative property is defined as

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

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

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

Conversely, in Agda, we can describe associativity and commutativity as follows:

```
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)
```

With this declaration of associativity and commutativity, we may further restrict the operations used to build a magma to one that is also associative to make it a semigroup. This we obtain the code below, semigroup that is structurally derived from magma.[1]

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

open IsMagma isMagma public
```

In the above definition `IsSemigroup` is a record type with two fields `isMagma` and `assoc`. · is a parameter of type Op_2 on set A denotes the binary operation for the semiring. `a ⊔ ℓ` is the least upper bound for the set. 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
```

Continuing on, we may encode various ring structures as follows: Non-associative ring is an algebraic structure with two binary operations addition (+) and multiplication (*). Addition is an Abelian group that is a group with commutative property and multiplication is unital magma that is a magma with identity. A group is a monoid with inverse property and a monoid is a semigroup with an identity element. A magma is called unital if it has an identity element. In non-associative ring, multiplication distributes over addition and it has an annihilating zero. Formally, nonAssociativeRing $(R, +, *, ^{-1}, 0, 1)$ should satisfy the following axioms:

- + is an abelianGroup:
  - Associativity: $\forall x, y, z \in R : x + (y + z) = (x + y) + z$
  - Identity: $\forall x \in R : (x + 0) = x = (0 + x)$
  - Inverse: $\forall x \in R : (x + x^{-1}) = 0 = (x^{-1} + x)$
- * is a unital magma
  - Identity: $\forall x, y \in R : (x * 1) = x = (1 * x)$

[1]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.

- Multiplication distributes over addition: $\forall x, y, z \in R : (x * (y + z)) = (x * y) + (x * z)$ and $(x + y) * z = (x * z) + (y * z)$
- Annihilating zero: $\forall x \in R : (x * 0) = 0 = (0 * x)$

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

 open IsAbelianGroup +-isAbelianGroup public
```

A quasiring is a type (2,2,0,0) algebraic structure for which both addition and multiplication is a monoid and multiplication distributes over addition, and has an annihilating zero. A quasiring $(Q, +, *, 0, 1)$ should satisfy the following axioms:

- + is a monoid:
  - Associativity: $\forall x, y, z \in Q : x + (y + z) = (x + y) + z$
  - Identity: $\forall x \in Q : (x + 0) = x = (0 + x)$
- * is a monoid:
  - Associativity: $\forall x, y, z \in Q : x * (y * z) = (x * y) * z$
  - Identity: $\forall x \in Q : (x * 1) = x = (1 * x)$
- Multiplication distributes over addition: $\forall x, y, z \in Q : (x * (y + z)) = (x * y) + (x * z)$ and $(x + y) * z = (x * z) + (y * z)$
- Annihilating zero: $\forall x \in Q : (x * 0) = 0 = (0 * x)$

```
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 the structure nearring, addition is a group, multiplication is a monoid, multiplication distributes over addition, and has an annihilating zero.

```
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 with a unary and a nullary operations. The binary operation addition (+) is an Abelian group and the binary operation multiplication (*) is a semigroup. For RingWithoutOne, multiplication distributes over addition and has an annihilating zero. A ringWithoutOne $(R, +, *, ^{-1}, 0)$ should satisfy the following axiom:

- + is an abelianGroup:
  - Associativity: $\forall x, y, z \in R : x + (y + z) = (x + y) + z$
  - Identity: $\forall x \in R : (x + 0) = x = (0 + x)$
  - Inverse: $\forall x \in R : (x + x^{-1}) = 0 = (x^{-1} + x)$
- * is a semigroup
  - Associativity: $\forall x, y, z \in R : x * (y * z) = (x * y) * z$
- Multiplication distributes over addition: $\forall x, y, z \in R : (x * (y + z)) = (x * y) + (x * z)$ and $(x + y) * z = (x * z) + (y * z)$
- Annihilating zero: $\forall x \in R : (x * 0) = 0 = (0 * x)$

```
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 RingWithoutOne structure is discussed. Morphisms of quasiring, nearring can be found in Agda standard library. The homomorphism for ringWithoutOne structure can be defined using group homomorphism. For two group structures $(G_1, +_1,)$ and $(G_2, +_2)$, homomorphism $f : G_1 \to G_2$ is a structure preserving map such that:

- f preserves the binary operation: $f(x +_1 y) = f(x) +_2 f(y)$
- f preserves the inverse operation: $f(x^{-1}) = f(x)^{-1}$
- f preserves the identity: $f(e_1) = e_2$ where e_1 is the identity in G_1 and e_2 is the identity in G_2

Homomorphism for ringWithoutOne is extended from group homomorphism such that got two ringWithoutOne structures $(R_1, +_1, *_1)$ and $(R_2, +_2, *_2)$, the homomorphism $f : R_1 \to R_2$ is a group homomorphism and preserves the multiplication operation. That is f is a group homomorphism and $f(x *_1 y) = f(x) *_2 f(y)$.

```
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)
```

In the above definition of ringWithoutOne homomorphism `IsRingWithoutOneHomomorphism` is defined as a record type with two fields `+-isGroupHomomorphism` and `*-homo`. `Homomorphic₂` is used to define the homomorphism for $*_1$ and $*_2$. A Homomorphism that is injective is called monomorphism and can ve defined as:

```
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. Isomorphism of ringWithoutOne structure can be defined as:

```
record IsRingWithoutOneIsoMorphism (⟦_⟧ : A → B) : Set (a ⊔ b ⊔ ℓ₁ ⊔ ℓ₂)
 ↪ where
  field
    isRingWithoutOneMonomorphism : IsRingWithoutOneMonomorphism ⟦_⟧
    surjective                   : Surjective ⟦_⟧

  open IsRingWithoutOneMonomorphism isRingWithoutOneMonomorphism public
```

6.3 Morphism composition

If f is a morphism such that $f : a \to b$ and g is a morphism such that $g : b \to c$, then composition of morphism can be defined as $g \circ f : a \to c$.

```
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
```

In the above ringWithoutOne homomorphism composition, `f` is a homomorphism from ringWithoutOne structures R_1 to R_2, `g` is a homomorphism from ringWithoutOne structures R_2 to R_3. `isGroupHomomorphism` field givese the composition of group homomorphism. We can define the composition for binary operations homomorphism (*) using transitive relation `≈₃-trans` from R_1 to R_3 such that

$$g(f((R_1 * x)y)) \approx (g((R_2 * fx)(fy)) \text{ and } g((R_2 * fx)(fy))) \approx ((R_3 * g(fx))(g(fy)))$$

$$\Rightarrow g(f((R_1 * x)y)) \approx ((R_3 * g(fx))(g(fy)))$$

6.4 Direct Product

The *direct product* $M \times N$ of two ringWithoutOne structures M and N is defined as a pair (m, n) where $m \in M$ and $n \in N$.

```
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* $M \times N$ of two non-associative ring structures M and N is defined as a pair (m, n) where $m \in M$ and $n \in N$.

```
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* $M \times N$ of two quasiring structures M and N is defined as a pair (m, n) where $m \in M$ and $n \in N$.

```
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* $M \times N$ of two nearring structures M and N is defined as a pair (m, n) where $m \in M$ and $n \in N$.

```
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

With these definitions, we can prove some frequently used properties and theories about the structures.[2]

[2]This section 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 $(S, \cdot)$ be a semigroup then

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

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

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

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 $(S, \cdot)$ be a commutative semigroup then

1. S is semimedial. The commutative semigroup S is left semimedial if $\forall\ x\ y\ z \in S\colon (x \cdot x) \cdot (y \cdot z) = (x \cdot y) \cdot (x \cdot z)$ and right semimedial if $\forall\ x\ y\ z \in S\colon (y \cdot z) \cdot (x \cdot x) = (y \cdot x) \cdot (z \cdot x)$. 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 $\forall\ x\ y\ z \in S\colon (x \cdot y) \cdot (z \cdot x) = (x \cdot z) \cdot (y \cdot x)$

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 contribution in the field 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 [32]. 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. [33]. Although there are some differences in axioms of Kleene algebra, in this chapter we consider the axioms defined in [33]

7.1 Definition

A set S with two binary operations + and $\cdot$ generally called addition and multiplication such that $(S,+)$ is a commutative monoid, $(S,\cdot)$ is a monoid and + distributes over $\cdot$ with annihilating zero is called a semiring. A semiring satisfying idempotent property is called idempotent semiring. An idempotentSemiring $(S,+,*,0,1)$ should satisfy the following axioms:

- + is a commutative monoid:
  - Associativity: $\forall x,y,z \in S : x+(y+z) = (x+y)+z$
  - Identity: $\forall x \in S : (x+0) = x = (0+x)$
  - Commutativity: $\forall x,y \in S : (x+y) = (y+x)$
- * is a monoid:
  - Associativity: $\forall x,y,z \in S : x*(y*z) = (x*y)*z$
  - Identity: $\forall x \in S : (x*1) = x = (1*x)$
- Idempotent: $\forall x \in S : (x+x) = x$
- Multiplication distributes over addition: $\forall x,y,z \in S : (x*(y+z)) = (x*y)+(x*z)$ and $(x+y)*z = (x*z)+(y*z)$
- Annihilating zero: $\forall x \in S : (x*0) = 0 = (0*x)$

A Kleene Algebra over set S that is an idempotent semiring with * operator that satisfies the following axioms.

$$\forall\ x \in S: 1 + (x \cdot (x^*)) \leq x^* \tag{7.1.1}$$

$$\forall\ x \in S: 1 + (x^*) \cdot x \leq x^* \tag{7.1.2}$$

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

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

In Agda, strong axioms of operator * is given. That is instead of partial order, equivalance is given in the above equations.[1]

```
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 *)

StarExpansive : A → Op₂ A → Op₂ A → Op₁ A → Set _
StarExpansive e _+_ _·_ _* = (StarLeftExpansive e _+_ _·_ _*) ×
  ↪ (StarRightExpansive e _+_ _·_ _*)

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

StarDestructive : Op₂ A → Op₂ A → Op₁ A → Set _
StarDestructive _+_ _·_ _* = (StarLeftDestructive _+_ _·_ _*) ×
  ↪ (StarRightDestructive _+_ _·_ _*)
```

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
```

[1]Kleene algebra with partial and pre order structures are defined in "Algebra.Ordered.Structures" in Agda standard library.

In the above definition, IsKleeneAlgebra structure is defined as a record type with three fields. Since $*$ is used to denote binary multiplication operation, we use $\star$ for the unary star operator. The field isIdempotentSemiring makes an idempotentSemiring with operator $+, *, 0$, and 1. Fields starExpansive and starDestructive are used to give the axioms for the star operator. We open isIdempotentSemiring to bring its definitions into scope.

7.2 Morphism

A morphism of Kleene algebra is a function between two Kleene algebras that preserves the algebraic structure of the underlying semiring and the Kleene star operation. Morphisms of Kleene algebra are important in the study of regular languages and automata, as they allow us to relate the behavior of different automata and regular expressions to each other. Morphism of Kleene algebra help to generalize the theory of regular languages and finite automata to more general algebraic structures.

For Kleene algebra $(K_1, +_1, *_1, ^{*_1}, 0_1, 1_1)$ and $(K_2, +_2, *_2, ^{*_2}, 0_2, 1_2)$, the homomorphism $f : K_1 \to K_2$ can be defined by using the homomorphism of structure idempotent semiring and preserving the * operator. Formally, $f : K_1 \to K_2$ is a structure preserving map such that:

- f preserves binary operation $+$: $f(x +_1 y) = f(x) +_2 f(y)$
- f preserves binary operation $*$: $f(x *_1 y) = f(x) *_2 f(y)$
- f preserves additive identity: $f(0_1) = 0_2$
- f preserves multiplicative identity: $f(1_1) = 1_2$
- f preserves star operation: $f(x^{*_1}) = f(x)^{*_2}$

```
record IsKleeneAlgebraHomomorphism (⟦_⟧ : A → B) : Set (a ⊔ ℓ₁ ⊔ ℓ₂)
↪ where
field
  isSemiringHomomorphism : IsSemiringHomomorphism ⟦_⟧
  ⋆-homo :  Homomorphic₁ ⟦_⟧ _⋆₁ _⋆₂

open IsSemiringHomomorphism isSemiringHomomorphism public
```

In Agda, Homomorphic₁ is used to give the homomorphism for unary operation, and is defined as:

```
Homomorphic₁ : (A → B) → Op₁ A → Op₁ B → Set _
Homomorphic₁ ⟦_⟧ ∙_ ∘_ = ∀ x → ⟦ ∙ x ⟧ ≈ (∘ ⟦ x ⟧)
```

A Kleene algebra homomorphism which is injective gives a monomorphism.

```
record IsKleeneAlgebraMonomorphism (⟦_⟧ : A → B) : Set (a ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    isKleeneAlgebraHomomorphism   : IsKleeneAlgebraHomomorphism ⟦_⟧
    injective                     : Injective ⟦_⟧

  open IsKleeneAlgebraHomomorphism isKleeneAlgebraHomomorphism public
```

A surjective monomorphism of a Kleene algebra gives isomorphism.

```
record IsKleeneAlgebraIsomorphism (⟦_⟧ : A → B) : Set (a ⊔ b ⊔ ℓ₁ ⊔ ℓ₂)
↪ where
  field
    isKleeneAlgebraMonomorphism   : IsKleeneAlgebraMonomorphism ⟦_⟧
    surjective                   : Surjective ⟦_⟧

  open IsKleeneAlgebraMonomorphism isKleeneAlgebraMonomorphism public
```

7.3 Morphism composition

If f is a morphism such that $f : a \to b$ and g is a morphism such that $g : b \to c$, then composition of morphism can be defined as $g \circ f : a \to c$.

```
  isKleeneAlgebraHomomorphism
  : IsKleeneAlgebraHomomorphism K₁ K₂ f
  → IsKleeneAlgebraHomomorphism K₂ K₃ g
  → IsKleeneAlgebraHomomorphism K₁ K₃ (g ∘ f)
isKleeneAlgebraHomomorphism f-homo g-homo = record
  { isSemiringHomomorphism = isSemiringHomomorphism ≈₃-trans
↪   F.isSemiringHomomorphism G.isSemiringHomomorphism
  ; ⋆-homo                = λ x → ≈₃-trans (G.⟦⟧-cong (F.⋆-homo x))
↪   (G.⋆-homo (f x))
  } where module F = IsKleeneAlgebraHomomorphism f-homo; module G =
↪   IsKleeneAlgebraHomomorphism g-homo
```

In the above quasigroup homomorphism composition, `f` is a homomorphism from Kleene algebra K_1 to K_2, `g` is a homomorphism from Kleene algebra K_2 to K_3. The proof for homomorphism composition is homomorphic is given using the proof for semiring homomorphism composition. `⋆-homo` gives composition for star operator using transitive relation such that:

$$g(f(x^{*_1})) = (g(fx)^{*_2}) \text{ and } (g(fx)^{*_2}) = (g(fx))^{*_3}$$

$$\Rightarrow g(f(x^{*_1})) = (g(fx))^{*_3}$$

The composition of monomorphism and isomorphism can be defined similar to homomorphism and can be found in Agda standard library.

7.4 Direct Product

The *direct product* $K \times L$ of two Kleene algebra structures K and L is defined as a pair (k, l) where $k \in K$ and $l \in L$.

```
KleeneAlgebra : KleeneAlgebra a ℓ₁ → KleeneAlgebra b ℓ₂ → KleeneAlgebra (a ⊔ b) (ℓ₁ ⊔ ℓ₂)
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
```

7.5 Properties

In this section we prove some properties of Kleene algebra

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

1. $0^* = 1$
2. $1^* = 1$
3. $\forall\, x \in K: 1 + x^* = x^*$
4. $\forall\, x \in K: x + x * x^* = x^*$
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 system discussed in chapter 3 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 a type(2) algebra with Latin square property or as a 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 and this can be attributed to lack of naming convention that is followed in naming algebraic structures and it's properties. For example, consider algebraic structures 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 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, these 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 may be attributed to the language of origin of the algebraic structures. In this case rng is used in French whereas 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 annihilating 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 ringWihthoutOne where one is the 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# *

open IsAbelianGroup +-isAbelianGroup public
```

Another example of ambiguity arises when defining structure nearring. 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. [34] states that the term rig originated as a joke that it is similar to rng that is missing the alphabet n and i to represent the identity does not exist for these structures. In Agda, the algebraic structure 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 standardized names, 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#
  starExpansive         : StarExpansive 1# + * ⋆
  starDestructive       : StarDestructive + * ⋆

open IsIdempotentSemiring isIdempotentSemiring public
```

8.2 Equivalent but structurally different

Quasigroup structure is an example that can be defined in two ways that are equivalent but structurally different. A type (2) Quasigroup can be defined as a set Q and binary operation · 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: [25]

$$a \cdot x = b$$

$$y \cdot a = b$$

Another definition of quasigroup is given as type a (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.

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

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

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

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

In Agda standard library, the quasigroup is defined as a type (2,2,2) algebra (shown 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 that is a type (2) algebra and a quasigroup that is a type (2,2,2) algebra are equivalent but are structurally different [35]. 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 a type (2,2,2) algebra then, a loop structure of type (2) will be forced to be defined with suboptimal name. One possible 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 as follows:

```
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 modularizing 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 defined as commutative monoid under addition and a monoid under multiplication. In Agda, both monoid and commutative monoid have an instance of equivalence relation. Hence, 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, and other algebraic structures. 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, we can observe that the operation $*$ is a semigroup is expressed with axioms congruent and associative. But, there is no field to say that $*$ is a semigroup. 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. However, it 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 ensure that the structure clearly exports the unfolded structure whose properties can be imported when required.

8.4 Identical structures

In abstract algebra when formalizing 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. Nearring is a group under addition and is a monoid under multiplication and multiplication right distributes over addition. In this case nearring is defined using two algebraic structures group and monoid. Other definition of nearring 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, nearring can be defined as a quasiring which has an 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, nearring is defined in which multiplication is a semigroup that is without identity. This can be attributed to the problem with ambiguity. It can be analyzed that having two different definitions for same structure is not a good practice. If nearring is defined using quasiring then it should also give an instance of additive group without having it to construct it when using the above formalization. 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 formalizing 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 the same structure. It is redundant to define two different structures from different hierarchy. Instead, in Agda, aliasing may be used to say that the bounded semilattice is same as idempotent commutative monoid. 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 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, +, ·, 0, 1) where S is the set of product families then:

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

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

3. 0 is the additive identity that is the empty product family

4. 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) [36]. 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 figure 8.1 the arrow has a single arc between them. That means all the properties are essential in defining a semigroup. Using product family algebra, a semigroup is

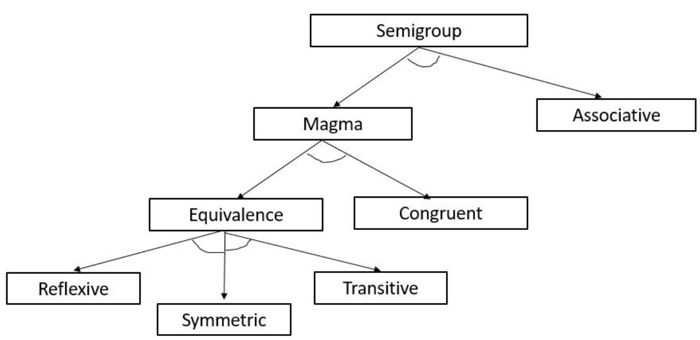

Figure 8.1: Feature diagram of semigroup

represented as

1. $Equivalence = Reflexive \cdot Symmetric \cdot Transitive$

2. $Magma = Equivalence \cdot Congruent$

3. $Semigroup = Magma \cdot Associativity$

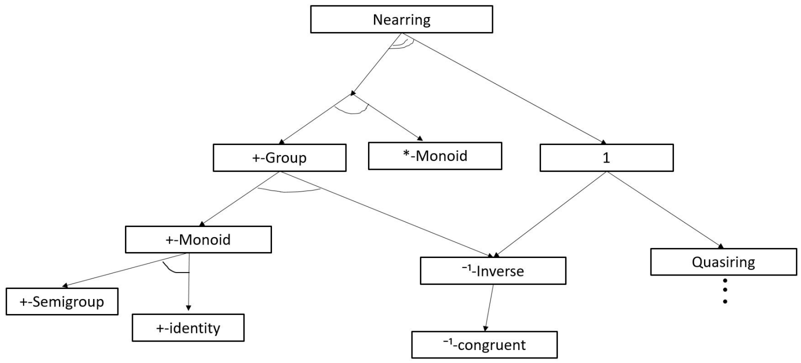

Figure 8.2: Feature diagram of nearring

In the above section, we saw that nearring has the problem of identical structure definition that can be defined using quasiring or group and semigroup. The figure 8.2 shows the feature diagram of nearring.

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

1. $+-Monoid = +-Semigroup \cdot +-identity$
2. $+-Group = +-Monoid \cdot {}^{-1}-Inverse$
3. $Nearring = (+-Group \cdot *-Monoid) + (Quasiring \cdot {}^{-1}-Inverse)$

In the above representation of the issue with identical structures, operation + represents a union. That means Nearring can have both $(+-Group \cdot *-Monoid)$ and $(Quasiring \cdot {}^{-1}-Inverse)$ but it will have many redundant fields. To over come this problem it is best to use XOR decomposition (denoted as 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:

1. $Equivalence = Reflexive \cdot Symmetric \cdot Transitive$
2. $Magma = Equivalence \cdot Congruent$

3. $Quasigroup = Magma \cdot (LatinSquareProperty \oplus$ Division) In figure 8.2, there are multiple definitions of the same structures for different operations. In the feature diagram this can be eliminated by mentioning the binary operation as weight to the arrow. Figure 8.4 shows the feature diagram where the binary operation is represented at the highest possible arrow. In this case, the equation using product family algebra does not change as it should explicitly mention the operations of the structures.

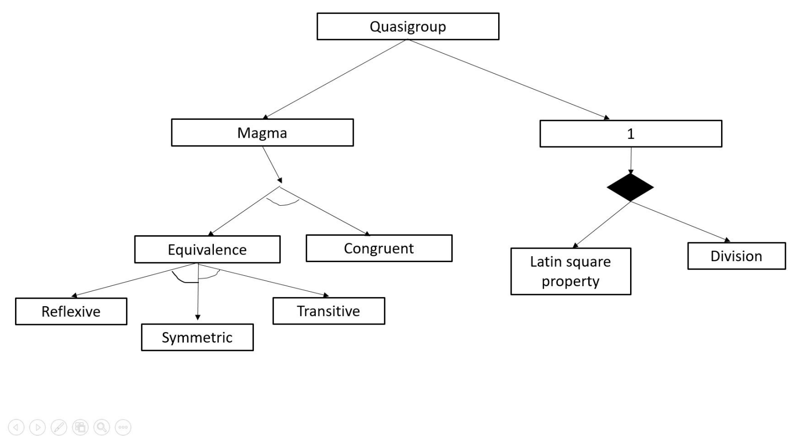

Figure 8.3: Feature diagram of quasigroup

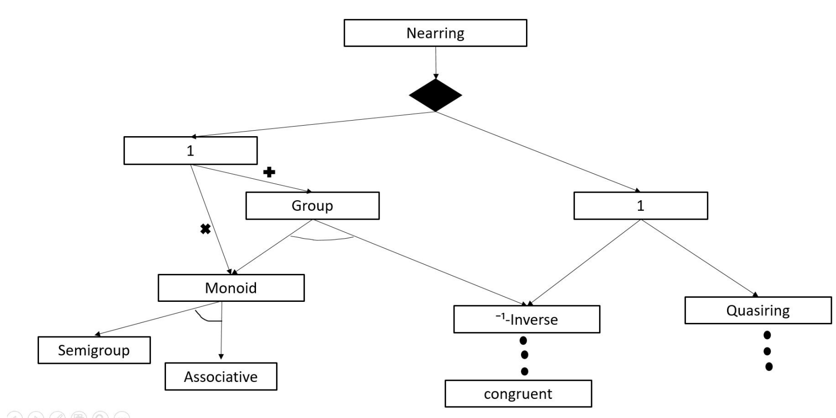

Figure 8.4: Feature diagram of Nearring

Chapter 9

Conclusion and Future Work

The primary of this work was to study algebraic structures in proof assistant systems. To define the scope the work we do a survey on coverage of algebraic on four proof assistant systems that are Agda, Idris, Coq and Lean. The thesis shows how to define a structure with some of its constructs and properties in Agda. We divide this into three main chapters based on closeness of structures that is quasigroup and loop, semigroup and ring, and Kleene algebra. We then analyze five problems that arises when defining algebraic structures in proof systems and give a brief overview of how it can be tackled with product family algebra.

In section 9.1, we summarize the contributions of this work and how it refers to the research outline described in Chapter 1. Section 9.2 discuss some extensions or future work of this work.

9.1 Summary of contributions

Universal algebra is a well studied and evolving branch of mathematics. Proof systems are useful in automated reasoning and becoming popular in research and applications more than ever. With introduction to universal algebra in Chapter 1 and Agda in Chapter 2, Chapter 3 provides an overview of quantitative use of algebraic structures in proof assistant systems. We create a clickable table that takes to the definition of structures in the standard libraries of the systems studied (Agda, Idris, Lean and Coq).

This leads to define the scope of contribution to Agda standard library. Chapter 5 is dedicated to study the structures quasigroup, loop and their variations. Chapter 6 provides an overview of semigroup and ring structures with their properties and morphisms. Chapter 7 is dedicated to study of Kleene algebra and it's properties in Agda. Along with these structures, we define structures unital magma, invertible magma, invertible unital magma, idempotent magma, alternate magma, flexible magma, semimedial magma, medial magma, with their direct products and morphisms.

Our approach of defining these structures led us to analyze the problems such as ambiguity in naming, equivalent and identical structures. Chapter 8 discuss how these problems becomes more evident in proof systems that might be ignored in classical 'pen-and-paper' technique. We give an overview of how product family algebra can be used to represent and tackle these problems.

9.2 Future work

Our work can be extended in different ways. The direct products defined in this thesis do not clearly differentiate between direct products, products and co-products of algebraic structures. There is currently discussion on Agda standard library to overcome this issue, but the changes are yet to come. Product family algebra is a powerful tool to solve many problems in software requirement, cryptography and other fields. Only a brief overview of how this tool can be used is discussed in Chapter 8. A more detailed study with implementation is required to concretely say to what extent the discussed problems can be solved. This work will rely on human efforts in building strong libraries in field of abstract algebra. A more robust and reliable generative library will be helpful to reduce human efforts.

Bibliography

[1] Wikipedia contributors, "Outline of algebraic structures — Wikipedia, the free encyclopedia," 2022, [Online; accessed 16-January-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Outline_of_algebraic_structures&oldid=1107380309

[2] I. Liaqat and W. Younas, "Some important applications of semigroups," *Journal of Mathematical Sciences & Computational Mathematics*, vol. 2, no. 2, pp. 317–321, 2021.

[3] Wikipedia contributors, "Group (mathematics) — Wikipedia, the free encyclopedia," 2023, [Online; accessed 16-January-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Group_(mathematics)&oldid=1133598242

[4] R. H. Bruck, "Some results in the theory of quasigroups," *Transactions of the American Mathematical Society*, vol. 55, pp. 19–52, 1944.

[5] J. Phillips and D. Stanovskỳ, "Automated theorem proving in quasigroup and loop theory," *Ai Communications*, vol. 23, no. 2-3, pp. 267–283, 2010.

[6] Jacques Carette, Russell O'Connor, and Yasmine Sharoda, "Building on the diamonds between theories: Theory presentation combinators," *arXiv preprint arXiv:1812.08079*, 2019.

[7] J. H. Barnett, "The roots of early group theory in the works of lagrange," 2017. [Online]. Available: https://digitalcommons.ursinus.edu/cgi/viewcontent.cgi?article=1002&context=triumphs_abstract

[8] Wikipedia contributors, "History of group theory — Wikipedia, the free encyclopedia," 2023, [Online; accessed 22-March-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=History_of_group_theory&oldid=1145792803

[9] Y. Sharoda, "Leveraging information contained in theory presentations," 2021. [Online]. Available: http://hdl.handle.net/11375/26272

[10] H. P. Sankappanavar and S. Burris, "A course in universal algebra," *Graduate Texts Math*, vol. 78, p. 56, 1981.

[11] Alexander Paulin, "Introduction to Abstract Algebra (Math 113)," 2021. [Online]. Available: https://math.berkeley.edu/~apaulin/AbstractAlgebra.pdf

[12] Wikipedia contributors, "Agda (programming language) — Wikipedia, the free encyclopedia," 2022, [Online; accessed 21-February-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Agda_(programming_language)&oldid=1127496533

[13] A. Bove, P. Dybjer, and U. Norell, “A brief overview of agda – a functional language with dependent types,” in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–78.

[14] Kidney, Donnacha Oisín, “Finiteness in cubical type theory,” 2020. [Online]. Available: https://doisinkidney.com/pdfs/masters-thesis.pdf

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

[16] P. Wadler, W. Kokke, and J. G. Siek, *Programming Language Foundations in Agda*, Aug. 2022. [Online]. Available: https://plfa.inf.ed.ac.uk/22.08/

[17] J. Z. Hu and J. Carette, “Formalizing category theory in agda,” in *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2021, pp. 327–342.

[18] Wikipedia contributors, “Constructive proof — Wikipedia, the free encyclopedia,” 2022, [Online; accessed 22-February-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Constructive_proof&oldid=1090644431

[19] M. Saqib Nawaz, M. Malik, Y. Li, M. Sun, and M. Ikram Ullah Lali, “A Survey on Theorem Provers in Formal Methods,” *arXiv e-prints*, p. arXiv:1912.03028, Dec. 2019.

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

[21] A. Mahboubi and E. Tassi, *Mathematical Components*. Zenodo, Jan. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4457887

[22] T. mathlib Community, “The lean mathematical library,” in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 367–381. [Online]. Available: https://doi.org/10.1145/3372885.3373824

[23] Carette Jacques, Farmer William, M. Kohlhase Michael, and Rabe Florian, “Big math and the one-brain barrier: The tetrapod model of mathematical knowledge.” *Math Intelligencer*, vol. 43, pp. 78–87, 2021.

[24] A. A. Ungar, “Einstein’s velocity addition law and its hyperbolic geometry,” *Computers & Mathematics with Applications*, vol. 53, no. 8, pp. 1228–1250, 2007.

[25] Wikipedia contributors, “Quasigroup — Wikipedia, the free encyclopedia,” 2022, [Online; accessed 7-January-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Quasigroup&oldid=1123964335

[26] D. Larchey-Wendling and Y. Forster, "Hilbert's tenth problem in coq (extended version)," *arXiv preprint arXiv:2003.04604*, 2020.

[27] J. Siekmann and P. Szabo, "The undecidability of the da-unification problem," *The Journal of Symbolic Logic*, vol. 54, no. 2, pp. 402–414, 1989. [Online]. Available: http://www.jstor.org/stable/2274856

[28] F.-a. Deng, L. Chen, S. Tuo, and S. Ren, "Characterizations of algebras," *Algebra*, vol. 2016, 2016.

[29] unknown, "Applications of ring theory," 2022, [Online; accessed 16-January-2023]. [Online]. Available: https://www.britannica.com/science/ring-mathematics.

[30] A. S. Netto, C. Chesman, and C. Furtado, "Influence of topology in a quantum ring," *Physics Letters A*, vol. 372, no. 21, pp. 3894–3897, 2008.

[31] unknown, "Applications of ring theory," 2022, [Online; accessed 16-January-2023]. [Online]. Available: https://discover.hubpages.com/education/Applications-of-Ring-Theory

[32] D. Kozen, "Kleene algebra with tests," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 3, pp. 427–443, 1997.

[33] ——, "A completeness theorem for kleene algebras and the algebra of regular events," *Information and computation*, vol. 110, no. 2, pp. 366–390, 1994.

[34] Wikipedia contributors, "Ring (mathematics) — Wikipedia, the free encyclopedia," 2023, [Online; accessed 16-January-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Ring_(mathematics)&oldid=1133737666

[35] E. Flinn, "Algebraic structures and variations: From latin squares to lie quasigroups," 2021. [Online]. Available: https://commons.nmu.edu/cgi/viewcontent.cgi?article=1704&context=theses

[36] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 1990.

[37] J. Geuvers, "Proof assistants : history, ideas and future," *Sadhana : Academy Proceedings in Engineering Sciences (Indian Academy of Sciences)*, vol. 34, no. 1, pp. 3–25, 2009.

[38] T. Evans, "Identities and relations in commutative moufang loops," *Journal of Algebra*, vol. 31, no. 3, pp. 508–513, 1974.

[39] J.-G. Ganascia, "Algebraic structure of some learning systems," in *International Workshop on Algorithmic Learning Theory*. Springer, 1993, pp. 398–409.

[40] Z. Ésik and L. Bernátsky, "Equational properties of kleene algebras of relations with conversion," *Theoretical Computer Science*, vol. 137, no. 2, pp. 237–251, 1995.

[41] S. Broda, S. Cavadas, and N. Moreira, "Kleene algebra completeness," Technical report, Universidade de Porto, Tech. Rep., 2014.

[42] T. G. Jaiyeola, S. P. David, and O. O. Oyebola, "New algebraic properties of middle bol loops ii," *Proyecciones (Antofagasta)*, vol. 40, no. 1, pp. 85–106, 2021.

[43] K. Kunen, "Moufang quasigroups," *Journal of Algebra*, vol. 183, no. 1, pp. 231–234, 1996.

[44] N. N. Didurik and V. A. Shcherbacov, "Some properties of neumann quasigroups," *arXiv preprint arXiv:1809.07095*, 2018.

[45] M. Stener, "Moufang loops: General theory and visualization of non-associative moufang loops of order 16," 2016.

[46] Wikipedia contributors, "Magma (algebra) — Wikipedia, the free encyclopedia," 2022, [Online; accessed 16-January-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Magma_(algebra)&oldid=1125597787

[47] B. Russell, *The Principles of Mathematics*. Routledge, 2020.

[48] Brilliant Math, "Russell's paradox," 2023, [Online; accessed 16-January-2023]. [Online]. Available: https://brilliant.org/wiki/russells-paradox/

[49] "Universe levels," 2023. [Online]. Available: https://agda.readthedocs.io/en/v2.6.1.3/language/universe-levels.html

[50] V. N. Redko, "On defining relations for the algebra of regular events," *Ukrainskii Matematicheskii Zhurnal*, vol. 16, pp. 120–126, 1964.

[51] Wikipedia contributors, "Intuitionism — Wikipedia, the free encyclopedia," 2022, [Online; accessed 22-February-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Intuitionism&oldid=1122615242

[52] ——, "Axiom schema of specification — Wikipedia, the free encyclopedia," 2022, [Online; accessed 28-February-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Axiom_schema_of_specification&oldid=1125383109

[53] ——, "Partially ordered set — Wikipedia, the free encyclopedia," 2023, [Online; accessed 6-March-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Partially_ordered_set&oldid=1142884446