TYPES OF ALGEBRAIC STRUCTURES IN PROOF ASSISTANT SYSTEMS

TYPES OF 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:	Types of Algebraic Structures in Proof Assistant Systems
AUTHOR:	Akshobhya Katte Madhusudana B.Eng. (Computer Science and Engineering), Bangalore University, Bangalore, India
SUPERVISOR:	Dr. Jacques Carette
NUMBER OF PAGES:	xi, 107

Abstract

Building a standard library of mathematical knowledge for a proof system is a complex task that relies on human effort. By doing a survey on the standard library of four proof systems (Agda, Idris, Lean, and Coq), we define the scope for our research to study types of algebraic structures in proof systems. From the result of the survey, we establish our focus to contribute to the Agda standard library.

Universal algebra studies structures by abstracting out the specific definitions and properties of algebraic structures. By providing an extensive and well-defined library of algebraic structures and theorems in Agda, it will enable researchers to explore new domains and build upon existing definitions (and theorems). We explore capturing a select subset of algebraic structures such as quasigroups, loops, semigroups, rings, and Kleene algebra with some of their constructs. Constructs like morphisms and direct product are given to us by universal algebra provide a way to relate different structures in a systematic and rigorous way. Morphisms allow us to understand how different structures are related.

During our exploration of capturing these structures in Agda, we encountered several issues. We categorized these issues into five classes and analyzed each problem to provide plausible solutions (except for naming). As part of this research, we define more than 20 algebraic structures and add more than 40 proofs to the 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 the early nineteenth century as complex problems and solutions evolved in other branches of mathematics such as geometry, number theory, and polynomial equations. With the growing help of technology, mathematicians are more indulged in automated reasoning. Increasing powers of computers and software tools that help automated reasoning become 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 and requires carefully defining mathematical objects and concepts [Phillips and Stanovskỳ(2010)]. 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 types of algebraic structures in proof systems.

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 necessary functionalities to be able to use the system easily without having to define all functionalities. The paper [Jacques Carette, Russell O'Connor, and Yasmine Sharoda(2019)] explores techniques to generate libraries with minimum human effort. Although generated libraries can define algebraic concepts, they are not considered as "standard library" for any proof system. For now, building standard libraries for proof systems relies on human efforts. This led to the question of what is the current scope of algebraic structures in the standard libraries of proof assistant systems. A survey of the coverage of algebraic structures in the standard libraries of proof assistant systems can help us understand which algebraic structures are already supported by various proof assistants, and which structures are still missing. This information can help researchers identify gaps in existing proof assistants and guide future development. 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.

Agda is used by mathematicians and computer scientists for research purposes. Contributing certain algebraic structures and theorems to Agda would help researchers to explore new domains by building upon the existing definitions and theorems easily. The Agda standard library follows an algebra hierarchy that starts with Magma as the initial structure from which other structures are defined. A magma is a set S with a binary operation $\cdot$ such that, $\forall x, y \in S, (x \cdot y) \in S$. A magma with associativity is called a semigroup. A Magma with division operation is called a quasigroup.

The definitions of constructs like homomorphism and direct product is given to us by universal algebra. Universal algebra provides a common framework by abstracting out the specific definitions and properties of algebraic structures. It helps us to study the commonalities of algebraic structures and define their constructs. An algebra in universal algebra is defined as an ordered pair (S,F) where S is a set and $F = (F_i : i \in I)$ is a finitary operations on A for some indexing set I [Sannella, Donald and Tarlecki, Andrzej(2012)]. Certain constructs like morphisms and direct products help us to relate different mathematical objects and structures in a systematic and rigorous way. Morphisms allow us to understand how different algebraic structures are related to one another. Direct product, on the other hand, is a useful tool for combining structures, such as monoids, groups, or rings, to create new and more complex structures that retain many of the desirable properties of the original structures. This allows us to study and understand larger, more complex systems and their properties.

1.1 Research Outline

To define the scope of our research to study algebraic structures in proof assistant systems, we capture the current coverage of algebraic structures in the standard libraries of these systems. As part of the survey, we consider four libraries: The Agda standard library (v1.7.1), the mathematical component library (1.12.0) for Coq, Idris 2, and mathematical library for Lean 3. In the effort of finding the coverage of algebraic structures in these libraries, we develop a clickable table that directs to the definition of the structure in the source code of these systems. Through the survey, we establish our focus for contributing to the Agda standard library[1].

Inspired by the ways algebraic structures are used in research, in this work we explore capturing a select subset of them in the Agda standard library. We study magma with

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

division operation that is quasigroup and loop structures. By defining them with their morphisms and direct product constructs, we can study their properties and relationships in a more systematic way. We also explore various types of loops such as bol-loop and moufang-loop and their properties. Semigroups 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. I was exposed to Kleene Algebra in discrete mathematics course. Inspired by the applications of Kleene algebra in finite state machines, regular expressions, and other branches of computer science, we study Kleene algebra by providing proof for its properties that may be used in developing other systems or applications. By contributing to Agda standard library, we hope that this work will be used by others.

As we explore capturing these structures in Agda, we encountered several problems. In this work, we abstract out these problems into five classes:

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 the same.

We analyze each problem and provide plausible solutions except for "Ambiguity in naming structures".

1.2 Thesis Outline

Chapters 2 and 3 focus on the background information necessary for reading this work, focusing on reviewing universal algebra and algebraic structures in Agda, respectively. Chapter 4 is a survey on algebraic coverage in proof systems. The next three chapters 5, 6 and 7 are dedicated to discussing the structures in detail. Chapter 5 explores quasigroup and loop structures with its variations. Chapter 6 discusses the properties of semigroup and ring. 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

Universal algebra is a branch of mathematics that studies algebraic structures in a general and abstract way. It provides a framework that allows mathematicians to study algebraic structures such as groups, rings, fields, lattices, and Boolean algebras, rather than studying them individually. Universal algebra provides constructs like homomorphisms, subalgebras, direct products, and more. These constructs help understand the algebraic structures and relationships between them. 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 [Sharoda(2021)].

In this chapter we study the concepts from universal algebra that help understand the characterization of algebraic structures with its constructs in Agda in the later chapters. We assume the reader to have basic knowledge on set theory (set, functions, and relations), knowledge of notation and concepts of first order logic. Section 2.1 defines terms like signature, theory, and algebra. We introduce constructs such as morphisms and direct product in Section 2.2. The definitions in this chapter are adapted from [Sankappanavar and Burris(1981)], [Wechler(2012)] and [Sannella, Donald and Tarlecki, Andrzej(2012)].

2.1 Universe, type, and signature

Before we dive into defining algebra, we introduce to some concepts that are used later in the chapter.

- A *term* in logic represents an object in the domain of discourse.
- A *function* $f : X \to Y$ is a mapping that associates each element of domain ($x \in X$) with a unique element in co-domain ($y \in Y$).
- A *function symbol* (or operation name) represents an operation that maps elements of domain to unique element in the co-domain.
- The number of operands in a function (or operation) is the *arity* of the operation.
- A *formula* is finite sequence of symbols from the set of alphabets of a language. A well-formed formula is a formula that is valid according to the rules of the specific language being used.
- *Term expressions* is a composition of terms with function symbols.
- For some formulas in propositional logic (a, b), we say a is a substitution instance of b if and only if a may be obtained from b by substituting formulas for symbols in b.

Logic allows us to describe properties of entities as formulas and provide reasoning about them. Equational logic limits these formulas (such as axioms or theorems) to be universally quantified equations of the form $t_1 = t_2$. Here t_1 and t_2 are terms expressible in the language of theory. A proposition is true if it is derivable from other true propositions using inference rules. The three inference rules in equational logic described in [Gries and Schneider(2013)] are:

- Leibniz equality: Two expressions are equal if one expression can be substituted with other without changing the truth statement.

$$\frac{t_1 = t_2}{t[x \mapsto t_1] = t[x \mapsto t_2]}$$

- Transitivity: If $t_1 = t_2$ and $t_2 = t_3$ then $t_1 = t_3$.

$$\frac{t_1 = t_2 \;\; t_2 = t_3}{t_1 = t_3}$$

- Substitution: For predicate p, if $p\ t$ is true, it remains true on all conditions.

$$\frac{p t}{p(t[xs \mapsto ts])}$$

where $t, t_1, t_2, \text{and} t_3$ are term expressions, x is some symbol in the language, xs and ts denotes list of symbols and list of expressions respectively.

- A *theory* in universal algebra is defined as a tuple (S, F, E) such that S is the carrier set, F is a finite set of function symbols with their arities, and E is a set of equations that are satisfied in S.

- A *sub theory* of a theory (S, F, E) is defined as $(S_\Delta, F_\Delta, E_\Delta)$ such that $S_\Delta \subseteq S$. The operations in sub theory ($op_\Delta \in |F_\Delta|$) is defined as

$$op_\Delta\ x_1...x_n = op\ x_1...x_n \in S_\Delta, \forall op \in |F|, \text{and } x_1...x_n \in S_\Delta$$

- A *signature* is a pair $\Sigma = (S, F)$ such that S is the carrier set and F is set of operation names.
- A Σ-*algebra* A is defined as pair $A = (A, F_A)$, a mathematical structure consisting of a carrier set (A) and a family of functions (F_A) defined for each function symbols in the signature. Algebra provides an interpretation for the carrier set A and function symbols F_A of a theory.
- The type (or language) of the algebra is a set of function symbols. Each member of this set is assigned a positive number which is the arity of the member.

2.2 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. For an algebra (A, F), θ is a congruence on A if θ satisfies the compatibility property. The compatibility property states that for each n-ary function symbol $f \in F$ and $x_i,\ y_i \in A$, If $x_i\ \theta\ y_i$ holds for $1 \leq i \leq n$ then $f^A(x_1,...,x_n)\ \theta\ f^A(y_1,....,y_n)$ holds [Sankappanavar and Burris(1981)].

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

$$g_1 \equiv g_2 \text{ and } h_1 \equiv h_2 \Rightarrow g_1 \cdot h_1 \equiv g_2 \cdot h_2 \; \forall g_1, g_2, h_1, h_2 \in G$$

- 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 function $\alpha : A \to B$ such that:

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

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

  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 algebra A and B, a homomorphism $f : A \to B$ is an isomorphism if it has an inverse, i.e. there is a homomorphism $f^{-1} : B \to A$ such that $f f^{-1} = id_{|A|}$ and $f^{-1} f = id_{|B|}$
  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 an 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 [Sankappanavar and Burris(1981)], the author proves that the composite of two homomorphisms (monomorphisms/isomorphisms) is also a homomorphism (monomorphism/isomorphism).

- A quotient algebra for some theory (S, F, E) with respect to congruence relation $(\cong)$ is defined on the theory (S_Q, F_Q, E_Q) where S_Q is the factor set of S such that:

$$S_Q = \{[x] | x \in S\}$$

The operations op_Q in the quotient algebra is defined as:

$$op_Q\ [x_1]...[x_n] = [op\ x_1...x_n]$$

where $op_Q \in F_Q$ and $op \in F$, and $[x]$ denotes the equivalence class such that $[x] = \{y \in S | x \cong y\}$.

- *Direct product*: For set of algebra $\{A_i | i \in I\}$ of same type indexed by some arbitrary set I, the cartesian product of the underlying sets is defined as $A = \prod_{i \in I} A_i$. Let ω_{A_i}

be the corresponding n-ary operator on A_i. We can define $\omega_A : A^n \rightarrow A$ by

$$\omega_A(a_1, ...a_n)(i) = \omega_{A_i}(a_1(i), ..., a_n(i)) \forall i \in I$$

where element $a \in A$ is a function from indexing set I to $\bigcup A_i$ such that $i \in I, a(i) \in A$. The algebra A equipped with all ω_A on A is the direct product of A_i. Each A_i is called the direct factor of A.

Chapter 3

Agda

Agda is a dependently typed programming language based on unified theory of dependent types and is an extension of Martin-Löf type theory [Agd(2023)]. Agda 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[Stump(2016)]. Agda is also a proof 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. Agda has been used in various applications such as formal verification, program synthesis, theorem proving, and automated reasoning [Saqib Nawaz et al.(2019)]. It is also used by researchers and academician to teach and explore the concepts of functional programming, type theory, and formal methods. This chapter provides a brief overview of programming in Agda in the context of algebraic structures.

3.1 Types and functions in Agda

3.1.1 Types in Agda

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 [Bove et al.(2009)]. A data type in Agda can be declared using the keyword `data`.

```
data Bool : Set where
  false : Bool
  true : Bool
```

In the example code 3.1.1, there are four things to notice.

1. `data` is the keyword used to define a new data type.

2. `Bool` is the name of the data type.

3. `Bool` is a type of kind `Set`. (More about `Set` is explained later in the chapter)

4. There are two constructor values of type `Bool`. They are `false` and `true`.

Let us consider another example of inductive datatype[1] to define natural numbers `Nat`.

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

[1]An inductive datatype is a datatype that is defined in terms of itself.

We can see that for defining natural number, it is impractical to list all the constructors like how we did for `Bool`. Instead, we give two ways to construct a natural number: `zero` is a natural number and `suc` is the successor of a natural number. In the above definition, `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 of type `Nat` and `zero` has no parameters. Another way of defining a type is using the keyword **`record`**. A record type can be defined by referencing other types and creating a synonym. An example of record type is discussed later in the chapter when we define algebraic structure.

3.1.2 Functions in Agda

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.

```
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 [Bove et al.(2009)].

With the above definition of type `Bool`, let us define `not` function using pattern matching as:

```
not : Bool → Bool
not false = true
not true = false
```

`not` function takes an argument of type `Bool`. The equal (`=`) sign is used to say that when a clause on left hand side of the equal sign is seen, the right hand side is what's computed.

Similar to Haskell, Agda doesn't have the concept of multi-argument functions. For example, to define addition (`Add`) function on natural numbers (`Nat`), we take an argument `Nat` and return a function that takes `Nat` and returns `Nat`.

```
add : Nat → Nat → Nat
add zero m = m
add (suc n) m = suc (add n m)
```

Operators in Agda are typically defined using symbolic notation or special operator symbols. Addition as an operation can be defined in Agda as:

```
_+_ : 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`. The underscore symbol in the name specifies where the argument goes. A recursive call must be made on a structurally smaller argument. For the function `_+_` above, the first argument `n` is smaller in the recursive call `suc n`. Operators can have different associativity and precedence rules. You can specify the fixity of operators to control how they are parsed. For example, `infixl 5 _+_`

3.2 Type levels in Agda

In the above section we say that `Bool` is a type of kind `Set`. If `Set` is a type of types, is it possible that `Set` is it's own type? If we make `Set` a type of itself, then the program becomes nonterminating [Stump(2016)]. But Agda is total that means all programs must terminate. This kind of paradox was introduced by Bertrand Russell and is called Russel's paradox.

According to Russel's paradox [Russell(2020)] the collection of all set is not a set. The naive set theory defines a set as well-defined collection of objects. The paradox defines the set of all sets that are not the member of themselves. This develops to two kinds of contradiction [Brilliant Math(2023)].

- If the set contains itself, then it should not be a member of itself by definition
- If the set does not contain itself the it is not a member of itself.

Agda introduces a series of universes to create the type hierarchy, and each universe represents a level of types. A universe is a type whose elements are type [uni(2023)]. This primitive type is useful to define and prove theorems about functions that operate on large set. In Agda, not every type belongs to `Set`. Since we cannot have a type `Set : Set`, Agda provides a hierarchy of universes `Set`, $\texttt{Set}_1$, $\texttt{Set}_2$ and so on. `Set` is the basic universe level that contains non dependent types like `Nat`, `Bool`. $\texttt{Set}_1$ contains all types from `Set` and allows dependent types likes `List A` where `A : Set`. Agda doesn't allow types at a given level to depend on types from higher universes.

Now we have seen that 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. 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`. The type `Set` ℓ represents the type of all types at level ℓ. For example, `Set 0` represents $\texttt{Set}_0$, `Set 1` represents $\texttt{Set}_1$, and so on. This type helps us to define equality generalized to an arbitrary level.

3.3 Equality

In Chapter 2, when defining theory, we say that equation is of the form $t_1 = t_2$ where t_1 and t_2 are term expressions and = represents equality relation. In dependent type theory, equality is a complex concepts. Equality says that two things are "equal". But asking "when two things are equal" is non trivial. In [Al Hassy(2021)], a hierarchy of "sameness" is given.

3.3.1 Syntactic equality

For some symbol $t_1 \text{and} t_2$, $t_1 = t_2$ if $t_1 and t_2$ are literally the same symbols. This is called synctatic equality.

3.3.2 Definitional equality

Definitional equality says that $t_1 = t_2$ when solving one symbol by applying some definitions leads to synctatic equality. Two programs are equal if they compute to the same value. For example, $(\lambda x \to x + y)5$ and $5 + y$ are the same. $5 + y$ is obtained when we compute the value of the expression $(\lambda x \to x + y)5$.

When we write a function in Agda, we add defining equations to Agda's definitional equality. For example, let us write a logical AND funciton (_ˆ_) in Agda:

```
_ˆ_ : Bool → Bool → Bool
true ˆ true = true
x ˆ y = false
```

In Agda, not every equations we write holds literally. In the above or function, only the equation `true ˆ true = true` holds. The equation `x ˆ y = false` overlaps with the first equation when both x and y are `true`. This equation does not definitionally hold. Agda will split this clause to three equations which holds definitionally:

```
false ^ true = false

true ^ false = false

false ^ false = false
```

3.3.3 Propositional equality

When we write a proof to say that two programs are equal, this proof cannot be a definitional equality. Instead this proof itself is a program that expresses that two things are equal. In a universe polymorphic type system like Agda, types are classified into various levels denoted as Set_0, Set_1, Set_2, and so on. The definition of propositional equalityin Agda standard library is universe polymorphic.

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

In Agda, propositional equality ($\_ \equiv \_$) is defined for a type A and an element $x \in A$ with a constructor `refl` that provides evidence that $x \equiv x$. Therefore every value is equal to itself and there is no alternative way to show values are equal. From this definition of equality, we can prove that it is an equivalence relation[2].

```
sym : ∀ {A : Set} {x y : A} → x ≡ y → y ≡ x
sym refl = refl


trans : ∀ {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl  =  refl
```

[2]An equivalence relation is a relation that is reflexive, symmetric, and transitive

We can also show that equality also holds substitution introduced in Chapter 2 with inference rules.

```
subst : ∀ {A : Set} {x y : A} (P : A → Set) → x ≡ y → P x → P y
subst P refl px = px
```

3.4 Structure definition

Let us now try to define Monoid, an algebraic structure in Agda. Monoid is an algebraic structure with a binary operation that satisfies associativity and has an identity element. In Agda we can define a structure as a record type using the keyword `record`. The record type allows to have parameters immediately after the record's name declaration or may be declared with `field` keyword.

```
record IsMonoid (A : Set) : Set₁ where
  field
    e : A
    op : A → A → A

    assoc : ∀ {x y z} → op x (op y z) ≡ op (op x y) z
    leftId : ∀ {x} → op e x ≡ x
    rightId : ∀ {x} → op x e ≡ x
```

In the above example, we see that `IsMonoid` structure has a parameter `A : Set` with fields `e` - the identity element and `op` - the binary operation. We also give the laws of monoid as its field. Another way to define a monoid structure is to parameterize the binary operation and the identity element.

```
record IsMonoid {A : Set} (_·_ : A → A → A) (ε : A) : Set where
  field
    assoc : ∀ {x y z} → op x (op y z) ≡ op (op x y) z
    leftId : ∀ {x} → op e x ≡ x
    rightId : ∀ {x} → op x e ≡ x
```

In the above definition we see that the carrier set `A` becomes implicit and we parameterize the operations of the structure. In theory, both the definitions are the same. Using fields inside the record may provide a more encapsulated and self-contained representation of the algebraic structure, while having them after the record name allows more flexibility in choosing the carrier set and operation when creating instances of the record.

From the above definition of `IsMonoid`, when we try to define `IsGroup`[3], we see that both monoid and group have things in common. They both have a carrier set (`A`), a binary operation (`op`), and an identity element (`e`). Given two structures that share some components, expressing that sharing component becomes difficult [Al Hassy(2021)]. To overcome these difficulties, we may parameterize the sharing components like the operations and the carrier set.

We may observe that all the algebraic structures has a carrier set. When defining algebraic structures in a module, we can make the carrier set as the argument of the module so it is accessible by all the structures defined under that module. The module declaration is treated as a top-level function that take the parameters of module as arguments. The parameters can be values and types but not other modules.

In section 3.3, we introduce different ways to say when two things are equal. When defining `IsMonoid`, we use Agda's propositional equality (`_≡_`) to compare the terms. In practice, this definition of propositional equality is too strong and one prefers to use a finer equivalence relation [Al Hassy(2021)]. Equivalence is useful when we want to capture "sameness" in a more flexible way, such as when dealing with quotient types. Agda standard library give a binary relation as an argument to the module and equivalence relation (`isEquivalence`) as a field to the `IsMagma` (defined later in the chapter) structure from which other structures are extended.

[3] Group is an algebraic structure that is a monoid with inverse operation.

```
module Algebra.Structures
  {a ℓ} {A : Set a}
  (_≈_ : Rel A ℓ)
  where
```

In the above code, we see that Agda standard library allows to define things in some arbitrary level. `A` is a `Set` in some level a and `_≈_` is a homogeneous binary relation `Rel` on universe `A ℓ`.

Let us understand how algebraic structure is defined in Agda standard library. An algebraic structure is defined in Agda standard library as a record type using the `record` keyword. The structures are obtained by wrapping the predicates that are expressed as "is-a" relation [Hu and Carette(2021)]. The types of algebraic structures are defined in module `Algebra.Structures` that have an underlying set `A` and the homogeneous binary relation `_≈_`. The following example shows how to characterize magma structures 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` gives the largest of two levels. `_≈_` is the binary operation argument for `IsEquivalence`. `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.

In the above definition, we see `(· : Op₂ A)`, the binary operation. Instead of writing `A → A → A`, Agda standard library defines a type level function `Op₂`. Type-level functions refer to functions that operate on types rather than on values. They are functions that take types as input and return types as output.

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

The subscript 2 represents that it is a binary operation. Similarly, the standard library defines `Op₁`:

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

Although parameterised structures are same as the unparameterised (unbundled) versions, in practice there may be certain presentations that are useful. Paper [Al-hassy et al.(2019)] discuss ways to unbundle structure at will. When building a library, it is not practical to provide all ways of parameterised structures. Agda standard library provides a bundeled version of the structures. The bundled version of the structures contains the operations of the structures, sets and axioms. The structures are imported from "Algebra.Structures" so we can parameterize the definitions with equality that is used to compare the terms of the structure.

```
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. _≈_ defines equality used to compare terms of `Magma`. `using` keyword is used to limit the imported components.

Before we finish discussing structure definition, there is one important concept to discuss that is *renaming*. Although the choice of name is theoretically irrelevant, renaming is often used to provide more generic and consistent naming conventions, making the library easier to use and more accessible to users. Agda standard library uses certain conventions for renaming. Keyword `renaming` is used to rename the fields. Consider the below example:

```
record IsNearSemiring (+ * : Op₂ A) (0# : A) : Set (a ⊔ ℓ) where
field
  +-isMonoid    : IsMonoid + 0#
  *-cong        : Congruent₂ *
  *-assoc       : Associative *
  distribʳ      : * DistributesOverʳ +
  zeroˡ         : LeftZero 0# *

open IsMonoid +-isMonoid public
  renaming
  ( assoc         to +-assoc
  ; ∙-cong        to +-cong
  ; ∙-congˡ       to +-congˡ
  ; ∙-congʳ       to +-congʳ
  ; identity      to +-identity
  ; identityˡ     to +-identityˡ
  ; identityʳ     to +-identityʳ
  ; isMagma       to +-isMagma
  ; isUnitalMagma to +-isUnitalMagma
  ; isSemigroup   to +-isSemigroup
  )

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

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

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

We use using, hiding, and renaming to control which names are brought into scope. From the above example, we see that for addition operation (+), the fields of the form $\mathscr{X}$ is renamed to $+ - \mathscr{X}$. [Al Hassy(2021)] proposes packaging the renaming to helper modules. However, as the new algebraic structures are added to the library, it becomes more difficult to maintain the conventions and requires carefully defining the structures.

3.5 Morphism in Agda

A homomorphism is a structure preserving map between two structures. A homomorphism for two magma structures is defined as a record type:

```
module MagmaMorphisms (M₁ : RawMagma a ℓ₁) (M₂ : RawMagma b ℓ₂) where

  open RawMagma M₁ renaming (Carrier to A; _≈_ to _≈₁_; _∙_ to _∙_)
  open RawMagma M₂ renaming (Carrier to B; _≈_ to _≈₂_; _∙_ to _∘_)

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

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

The raw structures, in the above example, RawMagma is the definition of signature the structure. IsMagmaHomomorphism is a record type with fields isRelHomomorphism and homo. Since the formalization of the types of algebraic structures in Agda is based on setoid, IsRelHomomorphism is defined for homomorphism between the homogeneous equivalence relations $\_\approx_1\_$ and $\_\approx_2\_$. $\texttt{Homomorphic}_2$ is defined for two binary operations as:

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

From this definition of homomorphism, monomorphism of the structure is given as:

```
  record IsMagmaMonomorphism (⟦_⟧ : A → B) : Set (a ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    isMagmaHomomorphism : IsMagmaHomomorphism ⟦_⟧
    injective           : Injective ⟦_⟧

  open IsMagmaHomomorphism isMagmaHomomorphism public
```

IsMagmaMonomorphism is defined as a record type with field isMagmaHomomorphism and injective. The Injective function is a one to one map defined as:

```
Injective : (A → B) → Set (a ⊔ ℓ₁ ⊔ ℓ₂)
Injective f = ∀ {x y} → f x ≈₂ f y → x ≈₁ y
```

where $\_\approx_1\_$ is the equality over the domain A and $\_\approx_2\_$ is the equality over codomain B.

Isomorphism of a structure can be derived from monomorphism with surjectivity.

```
record IsMagmaIsomorphism (⟦_⟧ : A → B) : Set (a ⊔ b ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    isMagmaMonomorphism : IsMagmaMonomorphism ⟦_⟧
    surjective          : Surjective ⟦_⟧

  open IsMagmaMonomorphism isMagmaMonomorphism public
```

IsMagmaIsomorphism is defined as a record type with field isMagmaMonomorphism and surjective. A surjective relation requires equality ($\_\approx_2\_$) on the codomain B and is defined as:

```
Surjective : (A → B) → Set (a ⊔ b ⊔ ℓ₂)
Surjective f = ∀ y → ∃ λ x → f x ≈₂ y
```

3.6 Direct Product in Agda

For two algebra A and B of the same theory with set S_A and S_B respectively, the product of algebra is defined with carrier set $(S_A \times S_B)$ and for each operation f in the theory is defined as:

$$f\ (x_{1_A}, x_{1_B})...(x_{n_A}, x_{n_B}) = (f_A\ x_{1_A}...x_{n_b}\ ,\ f_B\ x_{1_B}, x_{n_B})$$

where $x_{1_A}, ..., x_{n_B}$ are elements in S_A and $x_{1_B}, ..., x_{n_B}$ are elements in S_B. The direct products of structures are defined in `Algebra.Construct.DirectProducts` in Agda standard library. The direct product of magma structure is defined as:

```
magma : Magma a ℓ₁ → Magma b ℓ₂ → Magma (a ⊔ b) (ℓ₁ ⊔ ℓ₂)
magma M N = record
  { Carrier = M.Carrier × N.Carrier
  ; _≈_     = Pointwise M._≈_ N._≈_
  ; _·_     = zip M._·_ N._·_
  ; isMagma = record
    { isEquivalence = ×-isEquivalence M.isEquivalence N.isEquivalence
    ; ·-cong = zip M.·-cong N.·-cong
    }
  } where module M = Magma M; module N = Magma N
```

where `Magma` is the bundled version of the magma structure. The carrier set for direct product of `M` and `N` is the product $M \times N$. `Pointwise` gives the product of relations (`_≈_`) in `M` and `N`. `zip` gives a Σ-type of dependent pairs. `×-isEquivalence` is the product of equivalence relations in `M` and `N`.

3.7 Equational Proofs in Agda

A proof is a sequence of steps that transform one expression into another using a set of rules. Agda allows us to declare properties of functions and data types that need to be verified by the compiler. [Kidney, Donnacha Oisín(2020)].

In the section 3.1, 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 $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ for a semigroup i.e., a Magma with associative property $(x \cdot (y \cdot z) = (x \cdot y) \cdot z)$. 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

Types of 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[Saqib Nawaz et al.(2019)]. 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 [Jacques Carette, Russell O'Connor, and Yasmine Sharoda(2019)], the authors discuss the difficulties in building the libraries that support these systems by providing tools to write proofs 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 four proof assistant systems that are all dependently typed, higher order programming languages and supports, (at least partially) proof by reflection.

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

Idris is 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 [mathlib Community(2020)] is an open-source project by Microsoft Research. Lean is a proof assistant system written in C++. The last official version of Lean was 3.4.2 and is now supported by the Lean community. Lean 4 is the latest version of Lean and is a complete rewrite of previous versions of Lean. The mathlib [mathlib Community(2020)] library for Lean 3 has the most coverage of algebra compared to the other 3 proof assistant systems discussed in the paper. The mathlib library of Lean is also maintained by the Lean community for community versions of Lean. It was developed on a small library that was in Lean. It contained definitions of natural numbers, integers, and lists and had some coverage over algebra hierarchy. The latest version of mathlib has over 2794 definitions of algebra [Saqib Nawaz et al.(2019)].

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 table 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 [Carette Jacques, Farmer William, M. Kohlhase Michael, and Rabe Florian(2021)]. 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 semi-lattice is added as a biased structure. The Agda standard library defines left, right, and bi semi-modules and modules. A similar hierarchical approach as other algebraic structures is followed in defining modules. For example, a module is defined using bimodules and bi-modules using bi-semimodules. An alternative definition of modules is given in "Algebra.Module.Structure.Biased".

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

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

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

The algebra structures design hierarchy of the mathcomp library is inspired by the Packing mathematical structures. The "ssralg" file defines 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	✓	✓	✓	✓

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	**Agda**	**Coq**	**Idris**	**Lean**
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	✓	-	✓	✓
Lattice	✓	-	-	✓
Distributive lattice	✓	-	-	-

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
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	-	✓	-	-

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
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	-	✓	-	-
Unit algebra	-	✓	-	✓
Lalgebra	-	✓	-	-

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
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 the structures defined. 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 [Paulin Mohring(2012)].

The Idris library defines properties or laws of algebraic structures. The unique-Inverse defines that the inverses of monoids are unique. Other laws on groups include self-squaring i.e., 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 'squareId-Commutative' 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 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 [Ungar(2007)]. Quasigroups of various orders are used in the field of cryptography [Phillips and Stanovskỳ(2010)]. Lie algebra is used in differential geometry[Wikipedia contributors(2022h)]. With proof assistants, such as Agda, we can verify the relevant mathematical proofs of these algebraic structures. They are interactive software that helps 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 satisfying division operations. A *loop* is a quasigroup with identity. In this chapter, we explore morphisms and direct products for these structures and derive proofs for some of the properties of these structures.

5.1 Definitions

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 on setoid A as `IsMagma` with binary operation $\cdot$ and equivalence relation $\_\approx\_$. A quasigroup can be defined as a magma with left and right division identities. The operation \ (left division) and / (right division) for elements x, y in a quasigroup is defined as:

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

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

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

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

Agda supports most unicode characters (UTF-8) that can be used in identifiers. However, we cannot use reserved characters and keywords. Backslash (\) is a reserved character and cannot be used independently but can be used with other character. To overcome this issue, for division operation we use // and \\instead of / and \ respectively.

```
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 · //)
```

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 setoid 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. 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:

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

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

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

```
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 $f : A \rightarrow B$ 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*. 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 · and ∘ 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 in Agda as:

```
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 functions that take 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. If $x \cdot y = z$ then $y = x \backslash z$

3. 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 $(L, \cdot, /, \backslash)$ be a Loop with identity $x \cdot e = x = e \cdot x$ then the following properties holds

1. $x \,/\, x = e$

2. $x \backslash x = e$

3. $e \backslash x = x$

4. $x \,/\, e = x$

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

1. $x \cdot ((y \cdot x) \setminus x) = y \setminus x$
2. $x \cdot ((x \cdot z) \setminus x) = x \,/\, z$
3. $x \cdot (z \setminus x) = (x \,/\, z) \cdot x$
4. $(x \,/\, (y \cdot z)) \cdot x = (x \,/\, z) \cdot (y \setminus x)$
5. $(x \,/\, (y \cdot x)) \cdot x = y \setminus x$
6. $(x \,/\, (x \cdot z)) \cdot x = x \,/\, z$

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. $z \cdot (x \cdot (z \cdot y)) = ((z \cdot x) \cdot z) \cdot y$

4. $x \cdot (z \cdot (y \cdot z)) = ((x \cdot z) \cdot y) \cdot z$

5. $z \cdot ((x \cdot y) \cdot z) = (z \cdot (x \cdot y)) \cdot z$

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[Larchey-Wendling and Forster(2020)]. Although this problem was solved by 1970, In 1987 Siekmann and Szabo concluded that the unification problem of D_A-rewriting system[Siekmann and Szabo(1989)] cannot be predicted. In [Deng et al.(2016)], the author uses a type (2,2,0) algebra that is a *semigroup* 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 [Liaqat and Younas(2021)].

Similarly, *ring* is an algebraic structure that also have notable applications such as in number theory [britannica(2022)], in quantum computing [Netto et al.(2008)], in cryptography [hubpages(2022)], 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 in this chapter. 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

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

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

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

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]

[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.

```
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`. $\cdot$ is a parameter of type `Op₂ 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 on set R is an algebraic structure with two binary operations $(+)$ addition and $(*)$ multiplication. Addition $(R, +, ^{-1}, 0)$ is an Abelian group that is a group with commutative property. Multiplication $(R, *, 1)$ is a 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 identity. 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:

- $(R, +, ^{-1}, 0)$ is an Abelian Group:
  - Associativity: $\forall x, y, z \in R, x + (y + z) = (x + y) + z$
  - commutativity : $\forall x, y \in R, (x + y) = (y + x)$
  - Identity: $\forall x \in R, (x + 0) = x = (0 + x)$

  - Inverse: $\forall x \in R, (x + x^{-1}) = 0 = (x^{-1} + x)$

- $(R, *, 1)$ is a unital magma

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

- 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
```

We don't define `IsNonAssociativeRing` with `*-isUnitalMagma` to remove the redundant equivalence relation. This is discussed in Chapter 8. The same technique is followed when defining other ring like structures.

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:

- $(Q, +, 0)$ is a monoid:

  - Associativity: $\forall x, y, z \in Q, x + (y + z) = (x + y) + z$

  - Identity: $\forall x \in Q, (x + 0) = x = (0 + x)$

- $(Q, *, 1)$ 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. For RingWithoutOne, multiplication distributes over addition and has an annihilating zero. A ringWithoutOne $(R, +, *, ^{-1}, 0)$ should satisfy the following axiom:

- $(R, +, ^{-1}, 0)$ is an Abelian Group:
  - Associativity: $\forall x, y, z \in R, x + (y + z) = (x + y) + z$
  - commutativity: $\forall x, y \in R, (x + y) = (y + x)$
  - Identity: $\forall x \in R, (x + 0) = x = (0 + x)$
  - Inverse: $\forall x \in R, (x + x^{-1}) = 0 = (x^{-1} + x)$
- $(R, *)$ 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. The homomorphism for ringWithoutOne structure can be defined using group homomorphism. For two group structures $(G_1, +_1, ^{-1}, e_1)$ and $(G_2, +_2, ^{-1}, e_2)$, homomorphism $f : (G_1, +_1, ^{-1}, e_1) \to (G_2, +_2, ^{-1}, e_2)$ 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 and R_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`. A Homomorphism that is injective is called monomorphism and can be 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 $\approx_3$`-trans` from R_1 to R_3 such that

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

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

6.4 Direct Product

The *direct product* of ring like structures in Agda.

```
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 definition of direct product is similar to quasigroups discussed in Chapter 5. The direct products of `nonAssociativeRing`, `quasiring`, and `nearring` can be defined similar to `ringWithoutOne`. These definitions can be found in the Agda standard library.

6.5 Properties

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

6.5.1 Properties of Semigroup

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

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

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

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

Proof:

[2]This section provides proof for properties that was contributed by the author and other properties can be found in Agda standard library.

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 semigroup S is left semimedial if $(x \cdot x) \cdot (y \cdot z) = (x \cdot y) \cdot (x \cdot z)$ and right semimedial if $(y \cdot z) \cdot (x \cdot x) = (y \cdot x) \cdot (z \cdot x)$. A structure is semimedial if it is both left and right semimedial.

2. S is middle semimedial. The semigroup S is middle semimedial if $(x \cdot y) \cdot (z \cdot x) = (x \cdot z) \cdot (y \cdot x)$

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. $-(x * y) = -x * y$

2. $-(x * y) = x * -y$

Proof:

1. -⌣distrib^l-* : ∀ 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. -⌣distrib^r-* : ∀ 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. $-1 * x = -x$
2. if $x + x = 0$ then $x = 0$
3. $x * (y - z) = x * y - x * z$
4. $(y - z) * x = (y * x) - (z * x)$

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 fields such as relational algebra, automata and formal theory, design and analysis of algorithms and program analysis and compiler optimization [Kozen(1997)]. Kleene algebra generalizes operations from regular expressions. The axiomization of the algebra if regular events was recently proposed in 1966 but it was in 1984, a completeness theorem for relational algebra with a proper subclass of Kleene algebra was given [Kozen(1994)]. Although there are some differences in axioms of Kleene algebra, in this chapter we consider the axioms defined in [Kozen(1994)]

7.1 Definition

A set S with two binary operations + and $*$ generally called addition and multiplication such that $(S,+,0)$ is a commutative monoid, $(S,*,1)$ is a monoid, and $*$ distributes over + 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:

- $(S, +, 0)$ 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: $\forall x, y \in S, (x + y) = (y + x)$
- $(S, *, 1)$ 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 unary operator $(^*)$ that satisfies the following axioms.

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

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

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

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

where ≤ refers to the natural partial order:

$$a \leq b \leftrightarrow a + b = b$$

In Agda we define the partial order axioms in terms of equality. [1]

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

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

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

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

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

The Kleene algebra can be structurally derived from idempotent semiring. In Agda standard library, $+$ and $*$ operations are used to denote addition and multiplication. To keep the same template, $\star$ symbol is selected to denote the unary star operation.

```
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
```

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

```
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* of two Kleene algebra structures in Agda is defined using the product definition of idempotent semiring structure as:

```
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
```

where `idempotentSemiring` is the product of two idempotent semiring structures.

7.5 Properties

In this section we prove some properties of Kleene algebra

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

1. $1 + x^* = x^*$

2. $x * x^* + x^* = x^*$

3. $x^* + x^* * x = x^*$

4. $0 + x + x^* = x^*$

5. $1 + x + x^* = x^*$

6. $x + x^* = x^*$

7. $x^* * x^* + x^* = x^*$

8. $1 + x^* * x^* + x^* = x^*$

9. If $a * x = x * b$ then, $a^* * x + x * b^* = x * b^*$

10. If $x = y$ then, $1 + x * y^* + y^* = y^*$

11. $(x * y)^* * x + x * (y * x)^* = x * (y * x)^*$

Proof:

1.
```
1+x⋆≈x⋆ : ∀ x → 1# + x ⋆ ≈ x ⋆
1+x⋆≈x⋆ x = begin
  1# + x ⋆                     ≈⟨ +-congˡ (sym(starExpansiveʳ x)) ⟩
  1# + (1# + x ∗ x ⋆ + x ⋆)    ≈⟨ +-congˡ (+-assoc 1# (x ∗ x ⋆) (x
↪  ⋆)) ⟩
  1# + (1# + (x ∗ x ⋆ + x ⋆)) ≈⟨ sym(+-assoc 1# 1# (x ∗ x ⋆ + x
↪  ⋆)) ⟩
  1# + 1# + (x ∗ x ⋆ + x ⋆)    ≈⟨ +-congʳ (+-idem 1#) ⟩
  1# + (x ∗ x ⋆ + x ⋆)         ≈⟨ sym(+-assoc 1# (x ∗ x ⋆) (x ⋆)
↪  ) ⟩
  1# + x ∗ x ⋆ + x ⋆           ≈⟨ starExpansiveʳ x ⟩
  x ⋆                          ∎
```

2.
```
xx⋆+x⋆≈x⋆ : ∀ x → x * x ⋆ + x ⋆ ≈ x ⋆
xx⋆+x⋆≈x⋆ x = begin
  x * x ⋆ + x ⋆                    ≈⟨ +-comm _ _ ⟩
  x ⋆ + x * x ⋆                    ≈⟨ +-congʳ (sym(starExpansiveʳ
 ↪  x)) ⟩
  1# +  x * x ⋆ + x ⋆ + x * x ⋆    ≈⟨ +-assoc _ _ _ ⟩
  1# + x * x ⋆ + (x ⋆ + x * x ⋆)   ≈⟨ +-congˡ(+-comm (x ⋆) ( x *
 ↪  x ⋆)) ⟩
  1# + x * x ⋆ + (x * x ⋆ + x ⋆)   ≈⟨ +-assoc _ _ _ ⟩
  1# + (x * x ⋆ + (x * x ⋆ + x ⋆)) ≈⟨ +-congˡ (sym (+-assoc _ _
 ↪  _)) ⟩
  1# + (x * x ⋆ + x * x ⋆ + x ⋆)   ≈⟨ +-congˡ (+-congʳ (+-idem
 ↪  _)) ⟩
  1# + (x * x ⋆ + x ⋆)             ≈⟨ sym( +-assoc 1# (x * x ⋆)
 ↪  (x ⋆) ) ⟩
  1# + x * x ⋆ + x ⋆               ≈⟨ starExpansiveʳ x ⟩
  x ⋆                              ∎
```

3.
```
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    ≈⟨ +-assoc _ _ _ ⟩
  1# + (x ⋆ + x ⋆ * x)  ≈⟨ +-congˡ (+-comm (x ⋆) (x ⋆ * x)) ⟩
  1# + (x ⋆ * x + x ⋆)  ≈⟨ sym (+-assoc _ _ _) ⟩
  1# + x ⋆ * x + x ⋆    ≈⟨ starExpansiveˡ x ⟩
  x ⋆                   ∎
```

4.
```
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 ⋆              ∎
```

5. `1+x+x⋆≈x⋆ : ∀ x → 1# + x + x ⋆ ≈ x ⋆`

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

7.
```
x⋆+xx⋆+x⋆≈x⋆ : ∀ x → x ⋆ + x * x ⋆ + x ⋆ ≈ x ⋆
x⋆+xx⋆+x⋆≈x⋆ x = begin
  x ⋆ + x * x ⋆ + x ⋆   ≈⟨ +-assoc _ _ _ ⟩
  x ⋆ + (x * x ⋆ + x ⋆) ≈⟨ +-congˡ (+-comm _ _) ⟩
  x ⋆ + (x ⋆ + x * x ⋆) ≈⟨ sym (+-assoc _ _ _) ⟩
  x ⋆ + x ⋆ + x * x ⋆   ≈⟨ +-congʳ (+-idem _) ⟩
  x ⋆ + x * x ⋆        ≈⟨ +-comm _ _ ⟩
  x * x ⋆ + x ⋆        ≈⟨  xx⋆+x⋆≈x⋆ x  ⟩
  x ⋆                  ∎
x⋆x⋆+x⋆≈x⋆ : ∀ x → x ⋆ * x ⋆ + x ⋆ ≈ x ⋆
x⋆x⋆+x⋆≈x⋆ x =  starDestructiveˡ x (x ⋆) (x ⋆) (x⋆+xx⋆+x⋆≈x⋆
 ↪  x)
```

8.
```
1+x⋆x⋆+x⋆≈x⋆ : ∀ x → 1# + x ⋆ * x ⋆ + x ⋆ ≈ x ⋆
1+x⋆x⋆+x⋆≈x⋆ x = begin
  1# + x ⋆ * x ⋆ + x ⋆   ≈⟨ +-assoc _ _ _ ⟩
  1# + (x ⋆ * x ⋆ + x ⋆) ≈⟨ +-congˡ (x⋆x⋆+x⋆≈x⋆ x) ⟩
  1# + x ⋆              ≈⟨ 1+x⋆≈x⋆ x ⟩
  x ⋆                   ∎
```

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

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

10. x≈y⇒1+xy⋆≈y⋆ : ∀ x y → x ≈ y → 1# + x * y ⋆ + y ⋆ ≈ y ⋆
```
x≈y⇒1+xy⋆≈y⋆ x y eq = begin
  1# + x * y ⋆ + y ⋆    ≈⟨ +-assoc _ _ _ ⟩
  1# + (x * y ⋆ + y ⋆) ≈⟨ +-congˡ (+-congʳ (*-congʳ (eq))) ⟩
  1# + (y * y ⋆ + y ⋆) ≈⟨ sym(+-assoc _ _ _) ⟩
  1# + y * y ⋆ + y ⋆    ≈⟨ starExpansiveʳ y ⟩
  y ⋆                   ∎
```

11. [xy]⋆x+x[yx]⋆≈x[yx]⋆ : ∀ x y → (x * y) ⋆ * x + x * (y * x) ⋆
```
  ↪ ≈ x * (y * x) ⋆
[xy]⋆x+x[yx]⋆≈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 types 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, a ring structure is defined as an algebraic structure with two binary operations + and $*$, one unary operator $^{-1}$, and two elements 0 and 1 on setoid A. $(A, +, ^{-1}, 0)$ is an abelian group and $(A, *, 1)$ 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 that is a ring structure without multiplicative 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 for which addition is a group and multiplication is a monoid. But some mathematicians use the definition where multiplication is a semigroup. The same confusion also arises in defining semiring and rig structures. [Wikipedia contributors(2023d)] 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 `SemiringWithoutOne` 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 with division operation. 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:

$$a \cdot x = b$$

$$y \cdot a = b$$

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 [Flinn(2021)]. 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 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

+-isGroup : IsGroup + 0# _⁻¹
+-isGroup = record
  { isMonoid = +-isMonoid
  ; inverse = +-inverse
  ; ⁻¹-cong = ⁻¹-cong
  }
```

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
```

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.

Chapter 9

Conclusion and Future Work

The primary of this work was to study types algebraic structures in proof assistant systems. To define the scope of the work, we do a survey on the coverage of types of algebraic structures in four proof assistant systems which 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 the closeness of structures that is quasigroup and loop, semigroup and ring, and Kleene algebra. We then analyze five problems that arise when defining types algebraic structures in proof systems.

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 an introduction to universal algebra in Chapter 1 and Agda in Chapter 2, Chapter 3 provides an overview of the 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 definining the scope of contribution to the Agda standard library. Chapter 5 is dedicated to studying 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 the study of Kleene algebra and its 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 to defining these structures led us to encounter and analyze some problems such as ambiguity in naming, equivalent and identical structures. Chapter 8 discusses how these problems become more evident in proof systems that might be ignored in classical the 'pen-and-paper' technique.

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 a discussion on Agda standard library to overcome this issue, but the changes are yet to come. The current solution adapted in the Agda standard library to remove the redundant field will only remove the equivalence. However, there can be other redundant fields. For example, in commutative monoid, right identity can be obtained from left identity and commutativity. These problems are yet to be addressed.

The current work will rely on human efforts in building strong libraries in the field of abstract algebra. A more robust and reliable generative library will be helpful to reduce human efforts.

Bibliography

[uni(2023)] 2023. Universe levels. `https://agda.readthedocs.io/en/v2.6.1.3/language/universe-levels.html`

[Agd(2023)] 2023. What is Agda. `https://agda.readthedocs.io/en/v2.6.3/getting-started/what-is-agda.html`

[Al Hassy(2021)] Musa Al Hassy. 2021. Do-it-Yourself Module Systems. `http://hdl.handle.net/11375/26373`

[Al-hassy, Musa and Carette, Jacques and Kahl, Wolfram(2019)] Al-hassy, Musa and Carette, Jacques and Kahl, Wolfram. 2019. A language feature to unbundle data at will (short paper). , 14–19 pages.

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

[Barnett(2017)] Janet Heine Barnett. 2017. The roots of early group theory in the works of Lagrange. `https://digitalcommons.ursinus.edu/cgi/viewcontent.cgi?article=1002&context=triumphs_abstract`

[Bove et al.(2009)] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda – A Functional Language with Dependent Types. In *Theorem Proving in Higher Order*

Logics, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–78.

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

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

[Broda et al.(2014)] Sabine Broda, Sílvia Cavadas, and Nelma Moreira. 2014. *Kleene algebra completeness.* Technical Report. Technical report, Universidade de Porto.

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

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

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

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

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

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

[Flinn(2021)] Erik Flinn. 2021. Algebraic Structures and Variations: From Latin Squares to Lie Quasigroups. `https://commons.nmu.edu/cgi/viewcontent.cgi?article=1704&context=theses`

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

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

[Gries and Schneider(2013)] David Gries and Fred B Schneider. 2013. *A logical approach to discrete math.* Springer Science & Business Media.

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

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

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

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

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

[Kidney, Donnacha Oisín(2020)] Kidney, Donnacha Oisín. 2020. Finiteness in cubical type theory. `https://doisinkidney.com/pdfs/masters-thesis.pdf`

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

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

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

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

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

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

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

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

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

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

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

[Russell(2020)] Bertrand Russell. 2020. *The Principles of Mathematics*. Routledge.

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

[Sannella, Donald and Tarlecki, Andrzej(2012)] Sannella, Donald and Tarlecki, Andrzej. 2012. Foundations of algebraic specification and formal software development. https://link.springer.com/book/10.1007/978-3-642-17336-3

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

[Sharoda(2021)] Yasmine Sharoda. 2021. Leveraging Information Contained in Theory Presentations. http://hdl.handle.net/11375/26272

[Siekmann and Szabo(1989)] J. Siekmann and P. Szabo. 1989. The Undecidability of the DA-Unification Problem. *The Journal of Symbolic Logic* 54, 2 (1989), 402–414. http://www.jstor.org/stable/2274856

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

[Stump(2016)] Aaron Stump. 2016. *Verified functional programming in Agda.* Morgan & Claypool.

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

[Wadler et al.(2022)] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2022. *Programming Language Foundations in Agda.* https://plfa.inf.ed.ac.uk/22.08/

[Wechler(2012)] Wolfgang Wechler. 2012. *Universal algebra for computer scientists*. Vol. 25. Springer Science & Business Media.

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

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

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

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

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

[Wikipedia contributors(2022f)] Wikipedia contributors. 2022f. Magma (algebra) — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.`

php?title=Magma_(algebra)&oldid=1125597787 [Online; accessed 16-January-2023].

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

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

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

[Wikipedia contributors(2023b)] Wikipedia contributors. 2023b. History of group theory — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=History_of_group_theory&oldid=1145792803 [Online; accessed 22-March-2023].

[Wikipedia contributors(2023c)] Wikipedia contributors. 2023c. Partially ordered set — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Partially_ordered_set&oldid=1142884446 [Online; accessed 6-March-2023].

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