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, 118

Abstract

Building a standard library of mathematical knowledge for a proof system is a complex task that relies on human effort. By conducting 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. Providing an extensive and well-defined library of algebraic structures and theorems in Agda 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 products are given to us by universal algebra which provides 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. 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 a 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 on my supervisory committee.

I would like to express my sincere gratitude to all the maintainers and contributors of the 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, and 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.

The major part of this thesis contributes to the 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 the necessary functionalities to be able to use the system easily without having to define all functionalities. The paper [Carette et al.(2018)] explores techniques to generate libraries with minimum human effort. Although generated libraries can define algebraic concepts, they are not considered "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 to 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 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.

The definitions of constructs like homomorphism and direct product are 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 and Tarlecki(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 the 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, that is to study algebraic structures in proof assistant systems, we capture the current coverage of algebraic structures in the standard libraries of some commonly used proof assistant systems. As part of the survey, we consider four libraries: The Agda standard library, the mathematical component library for Coq, Idris 2, and mathematical library for Lean 3. In the effort to find 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 on 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 two important

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

non-associative algebra 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 structures is the 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 the 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 to each one of them.

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 their 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 algebra 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 us 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 their constructs in Agda in the later chapters. We assume the reader to have basic knowledge of 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

products in Section 2.2. The definitions in this chapter are adapted from [Sankappanavar and Burris(1981)], [Wechler(2012)] and [Sannella and Tarlecki(2012)].

2.1 Universe, type, and signature

Before we dive into defining algebra, we introduce 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 the elements of domain to a unique element in the co-domain.
- The number of operands in a function (or operation) is the *arity* of the operation.
- A *formula* is a 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 $t, t_1, t_2,$ 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 the 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 symbol 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)].

- 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 $\alpha : A \rightarrow B$ 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 $\alpha : A \rightarrow B$ 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 $f : A \rightarrow B$ is an isomorphism if it has an inverse, i.e. there is a homomorphism $f^{-1} : B \rightarrow A$ 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 $f : A \rightarrow A$ 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 $\alpha : A \rightarrow B$ 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).

- *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 $\omega_A : A^n \to A$ 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 Martin-Löf type theory [Team(2023c)]. 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 write and verify correct 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 academicians 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` or `record`.

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

In the above example code, 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 numbers, 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.

A record type in Agda is defined by using the keyword `record`. For example:

```
record Person : Set where
  field
    name : String
    age : Nat
```

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

- `Person` is the name of the data type.
- In record type, parameters may be defined after the record's name declaration or may be declared with `field` keyword.
- `field` keyword indicates the start of field declaration.
- `name : String` and `age : Nat` denotes that `name` and `age` are fields of type `String` and `Nat` respectively.

You can then create instances of this record type by providing values for the fields:

```
alice : Person
alice = record { name = "Alice" ; age = 25 }
```

We can access the fields of a record using dot notation:

```
nameOfAlice : String
nameOfAlice = alice.name

ageOfAlice : Nat
ageOfAlice = alice.age
```

We can use `constructor` keyword in `record` type declaration to define a constructor function for creating instances of the record type. For example:

```
record Person₁ : Set where
  constructor makePerson
  field
    name : String
    age : Nat
```

We can use the constructor `makePerson` to create instances of the $\texttt{Person}_1$ record:

```
alice : Person₁
alice = makePerson "Alice" 25
```

In Agda, the types of fields within a `record` can depend on the values of other fields within the same record. This way we can express the relationship or constraints between the components of a record. An example of this is the Agda's built-in Σ-type of dependent pairs.

```
record Σ {a b} (A : Set a) (B : A → Set b) : Set (a ⊔ b) where
  constructor _,_
  field
    fst : A
    snd : B fst
```

The Σ type represents a pair of values where the type of the second value depends on the value of the first. The underscore in the constructor denotes where the argument goes. We see more examples of this kind when we talk about functions later in the chapter.

To instantiate this Σ record type, we need to provide an element of type `A` and a value of type `B fst`:

```
alice : Σ String (λ _ → Nat)
alice = "Alice" , 25
```

Σ is a dependent pair type constructor that takes two arguments of type `String` and `(λ _ → Nat)`. The underscore in `λ _ → Nat` serves as a placeholder indicating that the type of the second component depends on the value of the first component. The underscore `(_)` placeholder is often used in Agda to indicate that you don't need to provide a name for a variable when its value isn't explicitly used in the expression. We see more examples of record type when we define algebraic structure later in the chapter.

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 the left-hand side of the equal sign is seen, and 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 infix operation can be defined in Agda as:

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

In the above example, the function `_+_` takes two arguments of type `Nat` and returns a value that is the 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`. What we normally call `Type` in programming, Agda calls it `Set`. If `Set` is a type of type, is it possible that `Set` is its own type? If we make `Set` a type of itself, then the language becomes non-terminating [Stump(2016)].

Bertrand Russell introduced a paradox when defining the collection of all sets and is called Russell's paradox. The naive set theory defines a set as well-defined collection of objects. The paradox [Russell(2020)] defines the set of all sets that are not members of themselves. This develops into two kinds of contradiction.

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

In Martin-Löf's type theory, when we make a `Set` its own type, it causes inconsistency, by Girard's paradox [Coquand(1986)]. To overcome this paradox, Agda introduces a series of universes to create the type hierarchy, and each universe represents a level of types [Team(2023a)]. A universe is a type whose elements are type [Team(2023b)]. This primitive type is useful to define and prove theorems about functions that operate on large sets. 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` stands for $\texttt{Set}_0$ and it is the base universe. From the definition of `Bool` in section 3.1, `false` and `true` is of type `Bool`, the type of `Bool` is `Set`, `Set` is of type $\texttt{Set}_1$, and so on. Agda doesn't allow types at a given level to depend on types from higher universes.

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 concept. Equality says that two things are "equal". But asking "when two things are equal" is nontrivial. In this section, we discuss a hierarchy of "sameness" from [Bocquet(2020)] and [Eremondi et al.(2022)].

3.3.1 Syntactic equality

For some symbol t_1 and t_2, $t_1 = t_2$ if $t_1 and t_2$ are literally the same symbols. This is called syntactic equality.

3.3.2 Definitional equality

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

When we write a function in Agda, we add defining equations to Agda's definitional

equality. For example, let us write a logical AND function (_^_) in Agda:

```
_^_ : Bool → Bool → Bool
true ^ true = true
x ^ y = false
```

In Agda, not every equation we write holds literally. In the above example, only the equation `true ^ true = true` holds. The equation `x ^ y = false` overlaps with the first equation when both x and y are `true`. This equation does not hold definitionally. In Agda, when pattern matching, cases are tried in order from top to bottom. Agda will split the above clause to three equations which holds definitionally [Abel(2012)]:

```
false ^ true = false

true ^ false = false

false ^ false = false
```

Some fundamental rules that Agda follows for definitional equality are:

- *Beta reduction* - We apply a lambda abstraction to an argument by substituting the argument into the body of the function. In Agda, we can replace the formal parameter of a lambda abstraction with an actual argument. This leads to the simplification of the expression.

- *Congruence Rules* - If two expressions are equal, and you perform an operation on both expressions, the results should also be equal. In Agda, if two expressions are definitionally equal, we can replace the sub-expressions with equal expressions that will result in equal expressions.

- *eta-expansion* - For record definition `Person` given in section 3.1, every `x : Person` is definitionally equal to `record {name = Person.name x ; age = Person.age x}`. It is based on the principle that two functions are equal if they produce equal results for all possible arguments.

We limit the scope of definitional equality here. Some references to find more information about definitional equality are [Norell(2007)] and [Martin-Löf and Sambin(1984)].

3.3.3 Propositional equality

When we write proof to say that two programs are equal, this proof may not be a definitional equality. Instead, this proof itself can be 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 $\texttt{Set}_0$, $\texttt{Set}_1$, $\texttt{Set}_2$, and so on. The definition of propositional equality in the Agda standard library is universe polymorphic. That is a generic definition of propositional equality is given using universes that can be used in different levels.

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

In the above definition, `a` is an implicit parameter representing the universe level of the set. In Agda, propositional equality `_≡_` is defined for any type A and any element `x` of type `A`, the identifier `refl` provides evidence that `x ≡ 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.

```
sym : Symmetric {A = A} _≡_
sym refl = refl
```

```
trans : Transitive {A = A} _≡_
trans refl eq = eq
```

We see how `Symmetric` and `Transitive` are defined in subsection discussing equivalence.

3.3.4 Equivalence

In Agda's standard library, equivalence (_≈_) is often preferred over propositional equality (_≡_) when defining algebraic structures [Al Hassy(2021)]. In Agda, equivalence is defined as a record type with three fields to say that the relation is reflexive, symmetric and transitive:

```
record IsEquivalence : Set (a ⊔ ℓ) where
  field
    refl  : Reflexive _≈_
    sym   : Symmetric _≈_
    trans : Transitive _≈_
```

In the above code, `IsEquivalence` is defined over for carrier `A : Set a` and binary relation `_≈_ : REL A ℓ` that are parameters to the module `Relation.Binary.Core`. We see why modules are parameterized with carrier set and equality relation later in the chapter when defining algebraic structures. The field `refl` is of type `Reflexive _≈_` and is defined as:

```
Reflexive : Rel A ℓ → Set _
Reflexive _~_ = ∀ {x} → x ~ x
```

Where _~_ is a relation of type Rel A ℓ that says for all element x, the elements are related to itself x ~ x. 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.

Symmetric relation is defined over a generalized symmetry that flips the order of arguments.

```
Sym : REL A B ℓ₁ → REL B A ℓ₂ → Set _
Sym P Q = P ⇒ flip Q
```

The first line declares `Sym` that takes two arguments: `P` of type `REL A B ℓ₁` and `Q` of type `REL B A ℓ₂`. Where `A` and `B` are carrier sets over arbitrary universe level. The module result type `Set _`, where the underscore represents a universe level that will be inferred. `flip` is a function to flip the order of the arguments.

```
Symmetric : Rel A ℓ → Set _
Symmetric _~_ = Sym _~_ _~_
```

`Symmetric` uses the previously defined `Sym` that states that a relation `_~_` is symmetric if it satisfies the conditions of symmetry as defined in the `Sym`. `Symmetric` will evaluate to type that `∀ x y : A, x ~ y → y ~ x` for relation ~ of type `REL A ℓ`.

Similar to symmetric relation, transitivity is defined using generalized transitive relation and `Transitive` will evaluate to type that `∀ i j k : A, i ~ j → j ~ k → i ~ k` for relation ~ of type `REL A ℓ`.

```
Trans : REL A B ℓ₁ → REL B C ℓ₂ → REL A C ℓ₃ → Set _
Trans P Q R = ∀ {i j k} → P i j → Q j k → R i k

Transitive : Rel A ℓ → Set _
Transitive _~_ = Trans _~_ _~_ _~_
```

3.4 Structure definition

A design decision was made in the Agda standard library to define algebraic structures as record types. The category theory library [Hu and Carette(2021)] also follows the same

design pattern to use record types. There are several advantages to using record type:

- Record types provide a convenient and flexible way to bundle together data and operations that satisfy certain algebraic properties.
- Algebraic structures may have dependent relationships between their components. For example, the type of an identity element depends on the type of elements in the set. Record types support dependent types, allowing you to express these relationships accurately.
- Records behave as modules. This allows us to export symbols in record type and bring them to scope. We may also need to make sure doing so does not create ambiguity.
- Record types have good IDE support(via Emacs)

Let us now try to define `IsMonoid`, an algebraic structure in Agda. A 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) : Set 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 structure as its field. Another way to define a monoid structure is to parameterize the binary operation and the identity element.

```
record IsMonoid₀ {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 `IsMonoid₀`, when we try to define `IsGroup`[2], 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 have 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 takes the parameters of the 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

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

defining `IsMonoid`, we use Agda's propositional equality (`_≡_`) to compare the terms. However, 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. Agda standard library gives 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.

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

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

Now we can define a magma structure in Agda with equivalence as:

```
record IsMagma₀ {A : Set} (_·_ : A → A → A) : Set where
  field
    isEquivalence : IsEquivalence _≈_
```

Although the equivalence allows us to compare the terms, it becomes restrictive to play with equal terms. In this case, we can use congruence which says that if two elements are equivalent, then applying certain operations to them should yield equivalent results. For example, let $\approx$ be an equivalence relation on a set S and operation $f : S \times S \to S$. The operation f is said to be congruent with respect to the equivalence if, for all $a, b, c, d \in S$, $a \approx b$ and $c \approx d$, then $f(a,c) \approx f(b,d)$. Therefore when defining a structure we give congruence with the operation.

Let us understand how algebraic structure is defined in the Agda standard library. An algebraic structure is defined in the 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 `Op₂ 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 `Congruent₂` 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`, the Agda standard library defines a type-level function `Op₂`.

```
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 $\mathtt{Op}_1$:

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

Although parameterized structures are same as the unparameterized (unbundled) versions, in practice there may be certain presentations that are useful. Paper [Al-hassy et al.(2019)] discusses ways to unbundle structure at will. When building a library, it is not practical to provide all ways of parameterized structures. Agda standard library provides a bundled version of the structures. The bundled version of the structures contains the operations of the structures, sets and axioms. Agda standard library defines the raw representation of a theory that is the definition of its signature. `RawMagma` in Agda standard library is defined as:

```
record RawMagma c ℓ : Set (suc (c ⊔ ℓ)) where
  infixl 7 _∙_
  infix  4 _≈_
  field
    Carrier : Set c
    _≈_     : Rel Carrier ℓ
    _∙_     : Op₂ Carrier

  infix 4 _≉_
  _≉_ : Rel Carrier _
  x ≉ y = ¬ (x ≈ y)
```

`_≉_` is the inequality relation that states that two elements are not equal `x ≉ y` if they are not equal under the equivalence relation. Bundled version structures are defined by importing structures 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 the 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. The 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 *
  distrib^r     : * DistributesOver^r +
  zero^l        : LeftZero 0# *

open IsMonoid +-isMonoid public
  renaming
  ( assoc          to +-assoc
  ; ·-cong         to +-cong
  ; ·-cong^l       to +-cong^l
  ; ·-cong^r       to +-cong^r
  ; identity       to +-identity
  ; identity^l     to +-identity^l
  ; identity^r     to +-identity^r
  ; 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
  ( ·-cong^l  to *-cong^l
  ; ·-cong^r  to *-cong^r
  )
```

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 $\mathscr{X}$ are renamed to $+-\mathscr{X}$. [Al Hassy(2021)] proposes packaging the renaming to helper modules. However, as 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. For two magma structures $(A,\cdot)$ and $(B,\circ)$, homomorphism $f : A \to B$ is defined as:

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

In Agda, 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 the signature of 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

In Chapter 2, we define the direct product of algebra. The standard library defines objects that are bi-products of appropriate algebras. In the context of algebraic structures, a bi-product is defined in such a way that it encompasses the properties of both a direct product and a direct sum. In many cases, the bi-product coincides with the direct product when certain conditions are met. However, bi-products and direct products may have distinct properties and behaviors for many algebraic structures. For the scope of the thesis, we do not consider this distinction. There is currently an issue in the standard library to address this problem.

The product of algebraic structures takes a more structured approach. It involves creating a new structure where the operations are carefully defined to combine the operations of the individual structures in a certain way such that they respect the individual structures' properties. 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 difference between direct product and cartesian product is mainly related to the

type of mathematical structures you are dealing with. Cartesian product refers to sets with no additional structure. The Cartesian product of two sets A and B, denoted as $A \times B$, is a new set that contains ordered pairs (a, b) where a is an element from set A, and b is an element from set B. A direct product typically deals with algebraic structures, such as groups, rings, or vector spaces.

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 the 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(2020)]. A constructive equational proof in Agda refers to the process of proving a logical proposition using equational reasoning within Agda's

type system [Murray(2022)].

In section 3.1, we have seen how to define natural numbers 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, we see three cases:

- Case 1: When `comm zero zero`, that is $m = n = 0$. Then `zero + zero = zero` holds by reflexivity. 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.

- Case 2: `comm zero (suc n)`, in this case, `m` is `zero` and `n` is a successor of some natural number. The proof proceeds recursively using induction on `n`. The recursive assumption is that `comm zero n` is already proved. That is `zero + n = n + zero`. Using this assumption, we can conclude that `zero + suc n` is equal to `suc n + zero`, by incrementing both sides of the equation with `suc`.

- Case 3: `comm (suc m) n`, In this case, `m` is a successor of some natural number, and `n` can be any natural number. The proof uses induction on `m`. The inductive step relies on the assumption that `comm m n` is true. The proof applies the successor function suc to both sides of the equation, to show that `suc m + n` is equal to `n + suc m`.

In algebraic structure, consider the example of the proposition that $x \cdot (y \cdot z) = y \cdot (x \cdot z)$ for a commutative semigroup i.e., a Magma with associativity $(x \cdot (y \cdot z) = (x \cdot y) \cdot z)$ and commutativity $(x \cdot y) = (y \cdot x)$. The proof can be written in Agda as:

```
x·yz≈y·xz :  ∀ x y z → x · (y · z) ≈ y · (x · z)
x·yz≈y·xz x y z = begin
  x · (y · z)    ≈⟨ sym (assoc x y z) ⟩
  (x · y) · z    ≈⟨ ·-cong^r (comm x y) ⟩
  (y · x) · z    ≈⟨ assoc y x z ⟩
  y · (x · 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 takes two type arguments `x` and `y`, and an argument of type `x IsRelatedTo y`. It returns a proof that `x` is equivalent `(~)` to `y`. The function simply uses pattern matching to extract the proof `x~y` and returns it.

```
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)
```

The `step-~` function provides a way to extend an equational proof using the relation `IsRelatedTo` while maintaining the equality `(~)`. It takes an initial proof that `x ~ y`, a proof `relTo y~z` of `y IsRelatedTo z`, and produces a proof of `x IsRelatedTo z`. The `trans` is the transitivity used to combine two proofs of relatedness.

The `step-≈` gives convenient syntax for invoking the `step-~`. step using equality is given as:

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

It provides a syntax shortcut for using the ≈⟨ ⟩ notation, which allows you to chain relatedness proofs using equational reasoning.

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

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

Chapter 4

Types of Algebraic Structures in Proof Assistant Systems - Survey

A survey of coverage of algebraic structures in proof systems will help to identify the gaps in the system. A survey can provide insights into how well each proof assistant supports the formalization of algebraic structures, making it easier for researchers and developers to choose the right platform for their mathematical formalization. Researchers and developers may be inspired to work on formalizing missing algebraic structures, which can lead to improvements in the tools and expand their utility.

This chapter provides the coverage of algebraic structures in proof assistant systems. Since I was exposed to Agda in the coursework, it added bias to select systems that are comparable with Agda thus eliminating systems such as Mizar and Isabelle. We consider four proof assistant systems that are all dependently typed higher-order programming languages [Saqib Nawaz et al.(2019)]. A standard library for a system is a collection of mathematical definitions with logical constructs, proof tactics, and utility functions that provide a foundational set of tools and definitions for users of these proof assistants

[Kohlhase and Rabe(2021)]. For the scope of the thesis, we consider the libraries as standard libraries that are commonly used and referenced for the respective proof systems in the context of algebra. The aim of this chapter is to provide documentation for the algebraic coverage in the standard libraries of proof assistant systems Agda, Idris, Coq, and Lean. In this chapter, the latest available versions are considered i.e., Agda standard library v1.7 [Community([n. d.])], Idris 2.0 [Brady(2021)], The Mathematical Components Library v1.13.0 [Mahboubi and Tassi(2021)], and The Lean mathematical library[mathlib Community(2020)].

The rest of the chapter is organized as follows. Section 4.1 gives a brief overview of the proof systems that we discuss in this chapter. The experiment setup is discussed in Section 4.2. In Section 4.3, we discuss the threat to validity. Section 4.4 discuss how algebraic structures are defined in each system by taking Monoid as an example. Section 4.5 provides a table that provides the results of the survey.

4.1 Proof assistant systems

This section provides an overview of proof assistant systems Coq, Idris, and Lean. We intentionally omit discussing Agda in this section since it is discussed in the previous chapter.

4.1.1 Coq

Coq [Paulin Mohring(2012)] is a theorem-proving system that is written in the OCaml programming language. It is designed to assist in the formalization of mathematical theorems by using constructive and higher-order logic. It was first released in 1989 and is

one of the most widely used proof assistant systems to define mathematical definitions, and theory and to write proofs. The Coq specification language is called Gallina. Users can write functions and algorithms in Gallina and then formally verify their correctness within the Coq environment [Bertot and Cast'eran(2013)]. Coq provides an interactive proof development environment where users can interact with Coq through a command line interface (or supported IDEs) to construct proofs.

The mathematical components library (v1.13.0) [Mahboubi and Tassi(2021)] includes various topics from data structures to algebra. The library provides an extensive collection of predefined data structures, algebraic structures (e.g., groups, rings, fields), and mathematical concepts (e.g., natural numbers, integers, rational numbers) [Saqib Nawaz et al.(2019)]. The mathcomp library was started with the Four Colour Theorem to support formal proof of the odd order theorem.

4.1.2 Idris

Idris is a functional programming language and interactive theorem prover created by Dr. Edwin Brady. Idris is built on a foundation of dependent type theory. The proofs are alike with Coq and the type system in Idris is uniform with Agda [Brady(2022)]. Idris includes a totality checker, which helps ensure that functions are defined for all inputs and that programs are guaranteed to terminate [Brady(2013)]. Idris 2 is a self-hosted programming language that combines linear-type systems. Idris like Agda, supports literate programming, this helps code and documentation to be interleaved in a natural and readable way. This is helpful for creating well-documented, readable formalizations. In this chapter, Idris 2 and Idris are used interchangeably and refer to Idris 2. Currently, there are no official package managers for Idris 2. However, several versions are under

development.

4.1.3 Lean

Lean [de Moura et al.(2015)] is an open-source project by Microsoft Research. Lean is a proof assistant system written in C++. Lean is based on a foundation of dependent type theory, similar to Coq and Idris. Lean has a powerful metaprogramming system that allows users to extend the language and develop domain-specific features [Ebner et al.(2017)]. 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. Lean has been used in various research projects and has seen adoption in both academia and industry. Lean has found applications in formal mathematics, formalization of mathematical proofs, software verification, and formal methods research. It has been used to verify complex software systems and to formalize mathematical concepts.

Lean comes with a standard library that covers a wide range of mathematics, including number theory, algebra, analysis, and set theory. The mathematical library (mathlib) [mathlib Community(2020)] 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 the algebra hierarchy. The latest version of mathlib has over 2794 definitions of algebra [Saqib Nawaz et al.(2019)].

4.2 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. To build the web crawler, we first manually select the source pages where the algebraic definitions can be found in each system. The regular expression will make use of the keyword used in each system to define algebraic structure. Thus, a part of the process of building the table 4.1 was automated. We then verify manually to make sure the algebraic structures are correctly extracted by the web crawler. 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. Section 4.3 will discuss the threat to validity of this approach.

4.3 Threat to Validity

In this section, we discuss threat to validity of the survey data.

- The libraries that we considered are under continuous development. The definitions that our web crawler extracted may not reflect the most recent code. We limit to a specific version of the library so that all the definitions in that version are captured.

- The threat in using regular expression is that it may not capture all variations and

formats of definitions. Some definitions may be missed, leading to incomplete data. We manually verify the result to look for missing definitions in the library.

- Each proof assistant has different language features that regular expressions need to account for. Ensuring that regular expressions work seamlessly across multiple proof assistants is challenging. We build unique regular expressions for each system.

- Different formalizations of algebraic structures may vary in their definitions and naming conventions. For example, In Agda a ring without multiplicative identity is called `RingWithoutOne`. The same structure in Idris is called `Ring`. These changes in naming will necessitate manually verifying the result in building the table 4.1.

- Regular expressions might capture irrelevant text that matches the pattern but is not part of the actual definition. For example, the comments in the source file may mislead the web crawler.

- Regular expressions are typically pattern-based and may not consider the broader context in which the definitions appear. This lack of context can result in misinterpretations.

4.4 Algebraic Structures

In this section, we discuss the characterization of monoid in different libraries. A monoid is a mathematical structure consisting of a set, an associative binary operation, and an identity element for that operation. This section discusses the definition of the monoid

structure in each proof system and gives a brief overview of the syntax used in the definition. We intentionally keep some overlap from the previous chapter when discussing the definition of monoid in Agda to have uniform coverage for each system.

4.4.1 Agda standard library

In the Agda standard library, algebraic structures are defined over setoid [Danielsson et al.(2011)].

```
record IsMonoid (· : Op₂ A) (ε : A) : Set (a ⊔ ℓ) where
  field
    isSemigroup : IsSemigroup ·
    identity    : Identity ε ·

  open IsSemigroup isSemigroup public
```

`IsMonoid` is defined as a record type over set `A` and equivalence `_≈_` that takes two parameters: ·, is a binary operation `Op₂ A` on a set A. ϵ, an element. The `Set (a ⊔ ℓ)` specifies the universe level at which this record exists. `field` the keyword indicates the start of the field declaration. Field `isSemigroup`, says that the binary operation · forms a semigroup, which means it is associative. The `IsSemigroup` type is defined as proof of associativity for the binary operation. Field `identity` indicates that ϵ is an identity element for the binary operation ·. The `Identity` type says that ϵ behaves as a left and right identity element for ·. Agda opens the `IsSemigroup` field `isSemigroup` so that when you have an instance of `IsMonoid`, you can directly access its `isSemigroup` field without any additional qualifiers.

The same follows for the bundled definitions of respective structures. A hierarchical approach is adapted to define algebraic structures to make the system scalable with minimal redundancy. 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`.

4.4.2 The mathematical components library

The algebra structures design hierarchy of the mathcomp library is inspired by the Packing mathematical structures. The `ssralg.v` 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. A monoid in mathcomp library is defined as:

```
Module Monoid.

Section Definitions.
Variables (T : Type) (idm : T).

Structure law := Law {
  operator : T -> T -> T;
  _ : associative operator;
  _ : left_id idm operator;
  _ : right_id idm operator
}.
Local Coercion operator : law >-> Funclass
```

The definition starts a new Coq module named `Monoid`. Modules in Coq are used to

group related definitions and theorems together for better organization and encapsulation. A new section named `Definitions` is opened to manage the scope of variables and definitions. Two variables within the scope of the definition section are: `T` is a type variable representing the underlying set and `idm` is a variable of type `T` representing the identity element of the monoid. `Structure law := Law { ... }` line represents the properties of a monoid. `operator` is a binary operation of type `T -> T -> T`. The next three lines denote associativity and left and right identity laws.

4.4.3 Idris

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. 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 a semigroup with an identity element. Idris addresses identity as a neutral element.

```
public export
interface Semigroup t => Monoid t where
  neutral : t

  monoidNeutralIsNeutralL : (l : t) -> l <+> neutral = l
  monoidNeutralIsNeutralR : (r : t) -> neutral <+> r = r
```

The definition is made public and available for export, making them accessible outside the module or scope where they are defined. An interface named `Monoid` extends

the Semigroup interface so that any type `t` that is an instance of `Monoid` must also satisfy the requirements of the Semigroup interface. `neutral` represents the identity element for the binary operation defined on type `t`. `monoidNeutralIsNeutralL` and `monoidNeutralIsNeutralR` specifies that for any value `l` (or `r`) of type `t`, when you combine (using the <+> operator) `l` (or `r`) with the neutral element on the left side (or right side), it should result in `l` (or `r`). `neutral` element acts as an identity element on the binary operation.

4.4.4 The mathematical library

The mathlib library for Lean 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 monoid structure definition in the library is given below [Baanen(2022)]:

```
@[ancestor semigroup mul_one_class, to_additive]
class monoid (M : Type u) extends semigroup M, mul_one_class M :=
(npow : ℕ → M → M := npow_rec)
(npow_zero' : ∀ x, npow 0 x = 1 . try_refl_tac)
(npow_succ' : ∀ (n : ℕ) x, npow n.succ x = x * npow n x .
 ↪ try_refl_tac)
```

In Lean, classes can inherit properties and methods from other classes. The monoid class inherits from the `semigroup` and `mul_one_class` classes. `to_additive` indicates that an additive version of the monoid class should be generated. The parameter `(M : Type u)` specifies that `M` is a type parameter that is the underlying set and `Type u` indicates that `M` is of a certain universe level. `monoid` class introduces an operation `npow`, a power operation that takes a natural number and an element of the monoid and calculates the repeated application of the monoid's binary operation. The class provides

two properties for this npow operation: `npow_zero'` specifies that raising any element to the power of `0` results in the identity element, and `npow_succ'` specifies the recursive behavior of the `npow` operation, where raising an element to a successor natural number is equivalent to multiplying it with the element raised to the previous power. The `. try_refl_tac` suggests that Lean should try to automatically prove this property using reflexivity tactics.

4.5 Result

In table 4.1, every checkmark links to the implementation in the source code of the library. In the Agda column, the checkmark in blue (✓) indicates the algebraic structures defined as part of this thesis.

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	✓	-	-	-

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
Semigroup	✓	✓	✓	✓
Band	✓	-	-	-
Commutative Semigroup	✓	-	-	✓
Semilattice	✓	-	-	✓
Unital magma	✓	-	-	-
Monoid	✓	✓	✓	✓
Commutative monoid	✓	✓	-	✓
Idempotent commutative monoid	✓	-	-	-
Bounded Semilattice	✓	-	-	-
Bounded Meetsemilattice	✓	-	-	-
Bounded Joinsemilattice	✓	-	-	-
Invertible Magma	✓	-	-	-
IsInvertible UnitalMagma	✓	-	-	-
Quasigroup	✓	-	-	-
Loop	✓	-	-	-
Moufang Loop	✓	-	-	-
Left Bol Loop	✓	-	-	-
Middle Bol Loop	✓	-	-	-
Right Bol Loop	✓	-	-	-
NilpotentGroup	-	-	-	✓

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
CyclicGroup	-	-	-	✓
SubGroup	-	-	-	✓
Group	✓	✓	✓	✓
Abelian group	✓	-	✓	✓
Lattice	✓	-	-	✓
Distributive lattice	✓	-	-	-
Near semiring	✓	-	-	-
Semiringwithout one	✓	-	-	-
Idempotent Semiring	✓	-	-	-
Commutative semiring without one	✓	-	-	-
Semiring without annihilating zero	✓	-	-	-
Semiring	✓	✓	-	✓
Commutative semiring	✓	-	-	✓
Non associative ring	✓	-	-	-
Nearring	✓	-	-	-
Quasiring	✓	-	-	-
Local ring	-	-	-	✓
Noetherian ring	-	-	-	✓
Ordered ring	-	-	-	✓
Cancellative commutative semiring	✓	-	-	-

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
Sub ring	-	-	-	✓
Ring	✓	✓	✓	✓
Unit Ring	✓	✓	✓	-
Commutative Unit ring	-	✓	-	-
Commutative ring	✓	✓	-	✓
Integral Domain	-	✓	-	-
LieAlgebra	-	-	-	✓
LieRing module	-	-	-	✓
Lie module	-	-	-	✓
Boolean algebra	✓	-	-	-
Preleft semimodule	✓	-	-	-
Left semimodule	✓	-	-	-
Preright semimodule	✓	-	-	-
right semimodule	✓	-	-	-
Bi semimodule	✓	-	-	-
Semimodule	✓	-	-	-
Left module	✓	✓	-	-
Right module	✓	-	-	-
Bi module	✓	-	-	-
Module	✓	✓	-	✓

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
Field	-	✓	✓	✓
Decidable Field	-	✓	-	-
Closed field	-	✓	-	-
Algebra	-	✓	-	-
Unit algebra	-	✓	-	✓
Lalgebra	-	✓	-	-
Commutative unit algebra	-	✓	-	-
Commutative algebra	-	✓	-	-
NumDomain	-	✓	-	-
Normed Zmodule	-	✓	-	-
Num field	-	✓	-	-
Real domain	-	✓	-	-
Real field	-	✓	-	-
Real closed field	-	✓	-	-
Vector space	-	✓	-	-
Zmodule Quotients type	-	✓	-	-
Ring Quotient type	-	✓	-	-
Unit rint quotient type	-	✓	-	-
Additive group	-	✓	-	-
characteristic zero	-	-	-	✓

Continued on next page

Table 4.1 – continued from previous page

Algebraic Structure	Agda	Coq	Idris	Lean
Domain	-	-	-	✓
Chain Complex	-	-	-	✓
Kleene Algebra	✓	-	-	-
HeytingCommutativeRing	✓	-	-	-
HeytingField	✓	-	-	-

Chapter 5

Theory Of Quasigroup and Loop in Agda

Quasigroups and loops find applications in various fields of mathematics, computer science, and physics. For example, Einstein's formula of addition of velocities gives a loop structure [Ungar(2007)]. In computer science, they are used in error-correcting codes and cryptography, where properties like uniqueness of solutions and error detection play a crucial role [Phillips and Stanovskỳ(2010)]. The properties of loops help the development of efficient computational algorithms, in the areas of optimization, scheduling, and permutation-based problems [Khan et al.(2015)]. In this chapter, we formalize two important non-associative algebras - quasigroup, and loop structure.

5.1 Definitions

A quasigroup is a set equipped with binary operations that satisfies the quasigroup property. Formally, a quasigroup $(Q, \cdot / \backslash)$ is an algebra consisting of a set (Q) and three binary operations $\cdot$ (multiplication), $\backslash$ (left division), and / (right division). These operations satisfy the following identities $\forall\, x\, y \in Q$:

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

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

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

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

In Chapter 3, we may observe that Agda supports many unicode characters (UTF-8) that can be used in identifiers. However, there are some limitations in the use of certain characters due to their reserved status that can cause conflict with Agda's syntax. Backslash (\) is used in Agda's syntax for various purposes, such as defining Unicode characters and escape sequences, which would result in potential conflicts if it were allowed as an identifier. To overcome this issue, we use // and \\instead of / and \ respectively.

We can write the above predicates in Agda as:

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

LeftDivides^r : Op₂ A → Op₂ A → Set _
LeftDivides^r _·_ _\\_ = ∀ x y → (x \\ (x · y)) ≈ y

RightDivides^l : Op₂ A → Op₂ A → Set _
RightDivides^l _·_ _//_ = ∀ x y → ((y // x) · x) ≈ y

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

We can combine two predicates using the product (×) as:

```
LeftDivides : Op₂ A → Op₂ A → Set _
LeftDivides · \\ = (LeftDivides^l · \\) × (LeftDivides^r · \\)

RightDivides : Op₂ A → Op₂ A → Set _
RightDivides · // = (RightDivides^l · //) × (RightDivides^r · //)
```

A quasigroup in Agda is defined as a record type:

```
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 `IsQuasigroup` is a record type with three binary operations $\cdot$, $\backslash\backslash$ $//$ on setoid A. The structure has five fields. We restrict discussing the syntax of definition as it is covered in Chapter 3.

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

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

Like left/right division, the identity predicate in Agda is given 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 ·)
```

A loop structure can be characterized in Agda as:

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

open IsQuasigroup isQuasigroup public
```

Bol loops are nonassociative algebraic systems that satisfy a weakened form of the associative property. A loop is called a *right bol loop* if it satisfies the right bol identity

$$((z \cdot x) \cdot y) \cdot x \;=\; z \cdot ((\, x \cdot y) \cdot x)$$

A loop is called a *left bol loop* if it satisfies the left bol identity

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

A loop is called *middle bol loop* if it satisfies the middle bol identity

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

A left-right bol loop is called a *moufang loop* if it satisfies the moufang identity. Moufang loops have applications in the fields of geometry, and theoretical physics [Kunen(1996)].

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

The definition of these structures is similar to `IsLoop` and can be found in the Agda standard library in module `Algebra.Structures`.

5.2 Morphism

A structure-preserving map f between two structures of the same type is called homomorphism or morphism in general. For quasigroups $(Q_1, \cdot, \backslash\backslash, //)$ and $(Q_2, \circ, \backslash, /)$, homomorphism is defined as a function $f : (Q_1, \cdot, \backslash\backslash, //) \to (Q_2, \circ, \backslash, /)$ 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)
```

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. For two loop structure $(L_1, \cdot, \backslash\backslash, //, e_1)and(L_2, \circ, \backslash, /, e_2)$, the function $f : (L_1, \cdot, \backslash\backslash, //, e_1) \to (L_2, \circ, \backslash, /, e_2)$ is a loop homomorphism if it is a quasigroup homomorphism and:

$$f(e_1) = e_2$$

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
```

The definitions of quasigroup loop monomorphism and isomorphism can be defined similarly to magma morphism discussed in Chapter 3. These definitions can be found in the Agda standard library under module Algebra.Morphism.Structures.

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 the composition of morphism can be defined as $g \circ f : a \to c$. In Agda we can prove that the composition of quasigroup homomorphism is homomorphic as:

```
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 gives the composition of homomorphism for a homogeneous binary relation ($\approx$). We can prove that the composition of binary operations homomorphism ($\cdot$) for quasigroup is homomorphic using transitive relation $\approx_3$`-trans` such that

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

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

Monomorphism and isomorphism composition constructs for quasigroup and loop are defined similar to homomorphism and can be found in the 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.leftDivides¹ , N.leftDivides¹ <*> x <*>
 ↪  y) , (λ x y → M.leftDivides² , N.leftDivides² <*> x <*> y)
    ; rightDivides = (λ x y → M.rightDivides¹ , N.rightDivides¹ <*> x
 ↪  <*> y) , (λ x y → M.rightDivides² , N.rightDivides² <*> 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 a 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 similarly to quasigroup.

5.5 Properties

In this section, we prove some properties of quasigroup, loop, middle bol loop, and moufang loop using Agda. The proofs for quasigroup and loop are adapted from [Stener(2016a)] and [Bruck(1944)].

5.5.1 Properties of Quasigroup

Cancellative quasigroups are used in cryptographic protocols. Properties such as left and right cancellation can be used to ensure the confidentiality of data during encryption and decryption processes [Shcherbacov(2003)]. 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.
```
cancel^l : LeftCancellative _·_
cancel^l x y z eq = begin
  y             ≈⟨ sym( leftDivides^r x y) ⟩
  x \\ (x · y)  ≈⟨ \\-cong^l eq ⟩
  x \\ (x · z)  ≈⟨ leftDivides^r x z ⟩
  z             ■

cancel^r : RightCancellative _·_
cancel^r x y z eq = begin
  y             ≈⟨ sym( rightDivides^r x y) ⟩
  (y · x) // x  ≈⟨ //-cong^r eq ⟩
  (z · x) // x  ≈⟨ rightDivides^r x z ⟩
  z             ■

cancel : Cancellative _·_
cancel = cancel^l , cancel^r
```

2.
```
y≈x\\z : ∀ x y z → x · y ≈ z → y ≈ x \\ z
y≈x\\z x y z eq = begin
  y            ≈⟨ sym (leftDivides^r x y) ⟩
  x \\ (x · y) ≈⟨ \\-cong^l 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 (rightDivides^r y x) ⟩
  (x · y) // y ≈⟨ //-cong^r eq ⟩
  z // y       ■
```

5.5.2 Properties of Loop

Properties of division operation hold for a loop.

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

1. $x / x = e$

2. $x \setminus x = e$

3. $e \setminus x = x$

4. $x / e = x$

Proof:

1. `x//x≈ϵ : ∀ x → x // x ≈ ϵ`
   ```
   x//x≈ϵ x = begin
     x // x       ≈⟨ //-cong^r (sym (identity^l x)) ⟩
     (ϵ · x) // x ≈⟨ rightDivides^r x ϵ ⟩
     ϵ            ■
   ```

2. `x\\x≈ϵ : ∀ x → x \\ x ≈ ϵ`
   ```
   x\\x≈ϵ x = begin
     x \\ x       ≈⟨ \\-cong^l (sym (identity^r x )) ⟩
     x \\ (x · ϵ) ≈⟨ leftDivides^r x ϵ ⟩
     ϵ            ■
   ```

3. `ϵ\\x≈x : ∀ x → ϵ \\ x ≈ x`
   ```
   ϵ\\x≈x x = begin
     ϵ \\ x       ≈⟨ sym (identity^l (ϵ \\ x)) ⟩
     ϵ · (ϵ \\ x) ≈⟨ leftDivides^l ϵ x ⟩
     x            ■
   ```

4. `x//ϵ≈x : ∀ x → x // ϵ ≈ x`
   ```
   x//ϵ≈x x = begin
     x // ϵ       ≈⟨ sym (identity^r (x // ϵ)) ⟩
     (x // ϵ) · ϵ ≈⟨ rightDivides^l ϵ x ⟩
     x            ■
   ```

5.5.3 Properties of Middle bol loop

Middle Bol loops are used in combinatorial design theory to construct specific types of combinatorial designs. The proofs for properties of the middle bol loop are adapted from [Jaiyeola et al.(2021)]. Let $(M, \cdot, /, \backslash, e)$ be a middle bol loop then the following identities hold.

1. $x \cdot ((y \cdot x) \backslash x) = y \backslash x$
2. $x \cdot ((x \cdot z) \backslash x) = x / z$
3. $x \cdot (z \backslash x) = (x / z) \cdot x$
4. $(x / (y \cdot z)) \cdot x = (x / z) \cdot (y \backslash x)$
5. $(x / (y \cdot x)) \cdot x = y \backslash x$
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) ≈⟨ ·-cong^r (x//x≈ϵ x) ⟩
  ϵ · (y \\ x)        ≈⟨ identity^l ((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) ≈⟨ ·-cong^l (x\\x≈ϵ x) ⟩
  (x // z) · ϵ        ≈⟨ identity^r ((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)       ≈⟨ ·-cong^l (\\-cong^r( sym (identity^l z))) ⟩
  x · ((ϵ · z) \\ x) ≈⟨ middleBol x ϵ z ⟩
  x // z · (ϵ \\ x)  ≈⟨ ·-cong^l (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) ≈⟨ ·-cong^r (x//x≈ϵ x) ⟩
  ϵ · (y \\ x)        ≈⟨ identity^l ((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) ≈⟨ ·-cong^l (x\\x≈ϵ x) ⟩
  (x // z) · ϵ        ≈⟨ identity^r (x // z) ⟩
  x // z              ■
```

5.5.4 Properties of Moufang Loop

The properties of Moufang loops have applications in computational mathematics and algorithm design. They provide insights into the efficient implementation of mathematical algorithms and data structures [Kunen(1996)]. The proofs in this sub-section are adapted

from [Stener(2016a)]. Let $(M, \cdot, /, \backslash, e)$ be a moufang loop then the following identities hold.

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.
```
alternative^l : LeftAlternative _·_
alternative^l x y = begin
  (x · x) · y        ≈⟨ ·-cong^r (·-cong^l (sym (identity^l x))) ⟩
  (x · (ϵ · x)) · y  ≈⟨ sym (leftBol x ϵ y) ⟩
  x · (ϵ · (x · y))  ≈⟨ ·-cong^l (identity^l ((x · y))) ⟩
  x · (x · y)        ■

alternative^r : RightAlternative _·_
alternative^r x y = begin
  x · (y · y)          ≈⟨ ·-cong^l(·-cong^r(sym (identity^r y))) ⟩
  x · ((y · ϵ) · y)    ≈⟨ sym (rightBol y ϵ x) ⟩
  ((x · y) · ϵ ) · y   ≈⟨ ·-cong^r (identity^r ((x · y))) ⟩
  (x · y) · y          ■

alternative : Alternative _·_
alternative = alternative^l , alternative^r
```

2. **flex** : Flexible _·_
   flex x y = begin
     (x · y) · x ≈⟨ ·-cong^{l} (sym (identity^{l} x)) ⟩
     (x · y) · (ϵ · x) ≈⟨ identical y ϵ x ⟩
     x · ((y · ϵ) · x) ≈⟨ ·-cong^{l} (·-cong^{r} (identity^{r} 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 ≈⟨ (·-cong^{r} (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)) ≈⟨ (·-cong^{l} (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: "Is there a general approach to verify whether a 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 *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 [Liaqat and Younas(2021)].

Similarly, *ring* is an algebraic structure that also has notable applications in number theory [Pedrouzo-Ulloa et al.(2021)], in quantum computing [Netto et al.(2008)], in cryptography [Khathuria et al.(2021)], and many other fields. Variations of ring structure such as near-ring, quasi-ring, and non-associative rings are 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. This chapter aims to define these structures and prove some properties in the Agda

standard library that can help build other systems that use these structures.

6.1 Definition

A semigroup is an algebraic structure that consists of a set equipped with an associative binary operation. Formally, a semigroup is defined as: Let S be a set, and let $\cdot$ be a binary operation on S, the structure $(S, \cdot)$ is a semigroup if the following property holds:

$$\forall\ x\ y\ z \in S,\ x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

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

$$\forall\ x\ y \in S,\ x \cdot y = y \cdot x$$

In Agda, the above predicates can be written as:

```
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)
```

In Agda, we can define a semigroup as a record type to ensure that the properties of the semigroup are satisfied.

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

open IsMagma isMagma public
```

Similarly, commutative semigroup can be defined using semigroup as:

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

open IsSemigroup isSemigroup public
```

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 an inverse and a monoid is a semigroup with identity. A magma is called unital if it has an identity. In a non-associative ring, multiplication distributes over addition, and it has an annihilating zero. Formally, nonAssociativeRing $(R, +, *, ^{-1}, 0, 1)$ should satisfy the following identities:

- $(R, +, ^{-1}, 0)$ 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 an algebraic structure for which both addition and multiplication forms a monoid, multiplication distributes over addition and has an annihilating zero. A quasiring $(Q, +, *, 0, 1)$ should satisfy the following identities:

- $(Q, +, 0)$ 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. For the structure nearring, addition forms a group, multiplication forms 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
```

A ring without one or rig or ring without unit is an algebraic structure with two binary operations, a unary and a nullary operation. A ringWithoutOne $(R, +, *, ^{-1}, 0)$ should satisfy the following identities:

- $(R, +, ^{-1}, 0)$ 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

In Agda, homomorphism for ringWithoutOne is defined using group homomorphism such that for 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, `IsRingWithoutOneHomomorphism` is defined as a record type with two fields `+-isGroupHomomorphism` and `*-homo`. The definition of isomorphism and monomorphism can be found in the Agda standard library under the module `Algebra.Morphism.Structures`.

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 gives 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

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

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

6.4 Direct Product

The *direct product* of ring-like structures in Agda is defined as:

```
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.distrib^l , S.distrib^l) <*> x <*> y <*> z)
                             , (λ x y z →
                 (R.distrib^r , S.distrib^r) <*> x <*> y <*> z)
      ; zero     = uncurry (λ x y → R.zero^l x , S.zero^l y)
                             , uncurry (λ x y → R.zero^r x , S.zero^r 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.[1]

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

6.5.1 Properties of Semigroup

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

1. S is alternative. The Semigroup S left alternative if $(x \cdot x) \cdot y = x \cdot (x \cdot y)$ 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:

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

An application of semimedial property of commutative semigroup is seen in study of quasigroups and loops [Liaqat and Younas(2021)]. The proofs in this section are adapted

from [Deng et al.(2016)]. 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.
```
semimedial^l : LeftSemimedial _·_
semimedial^l x y z = begin
(x · x) · (y · z) ≈⟨ assoc x x (y · z) ⟩
x · (x · (y · z)) ≈⟨ ·-cong^l (sym (assoc x y z)) ⟩
x · ((x · y) · z) ≈⟨ ·-cong^l (·-cong^r (comm x y)) ⟩
x · ((y · x) · z) ≈⟨ ·-cong^l (assoc y x z) ⟩
x · (y · (x · z)) ≈⟨ sym (assoc x y ((x · z))) ⟩
(x · y) · (x · z) ■

semimedial^r : RightSemimedial _·_
semimedial^r x y z = begin
(y · z) · (x · x) ≈⟨ assoc y z (x · x) ⟩
y · (z · (x · x)) ≈⟨ ·-cong^l (sym (assoc z x x)) ⟩
y · ((z · x) · x) ≈⟨ ·-cong^l (·-cong^r (comm z x)) ⟩
y · ((x · z) · x) ≈⟨ ·-cong^l (assoc x z x) ⟩
y · (x · (z · x)) ≈⟨ sym (assoc y x ((z · x))) ⟩
(y · x) · (z · x) ■

semimedial : Semimedial _·_
semimedial = semimedial^l , semimedial^r
```

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)) ≈⟨ ·-cong^l (sym (assoc y z x)) ⟩
  x · ((y · z) · x) ≈⟨ ·-cong^l (·-cong^r (comm y z)) ⟩
  x · ((z · y) · x) ≈⟨ ·-cong^l ( 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
-‿distrib^l-* x y = sym $ begin
  - x * y
        ≈⟨ sym $ +-identity^r (- x * y) ⟩
  - x * y + 0#
        ≈⟨ +-cong^l $ sym ( -‿inverse^r (x * y) ) ⟩
  - x * y + (x * y + - (x * y))
        ≈⟨ sym $ +-assoc (- x * y) (x * y) (- (x * y)) ⟩
  - x * y + x * y + - (x * y)
        ≈⟨ +-cong^r $ sym ( distrib^r y (- x) x ) ⟩
  (- x + x) * y + - (x * y)
        ≈⟨ +-cong^r $ *-cong^r $ -‿inverse^l x ⟩
  0# * y + - (x * y)
        ≈⟨ +-cong^r $ zero^l y ⟩
  0# + - (x * y)
        ≈⟨ +-identity^l (- (x * y)) ⟩
  - (x * y)
        ■
```

```
2. -‿distrib^r-* : ∀ x y → - (x * y) ≈ x * - y
   -‿distrib^r-* x y = sym $ begin
     x * - y
           ≈⟨ sym $ +-identity^l (x * (- y)) ⟩
     0# + x * - y
           ≈⟨ +-cong^r $ sym ( -‿inverse^l (x * y) ) ⟩
     - (x * y) + x * y + x * - y
           ≈⟨ +-assoc (- (x * y)) (x * y) (x * (- y)) ⟩
     - (x * y) + (x * y + x * - y)
           ≈⟨ +-cong^l $ sym ( distrib^l x y ( - y) )  ⟩
     - (x * y) + x * (y + - y)
           ≈⟨ +-cong^l $ *-cong^l $ -‿inverse^r y ⟩
     - (x * y) + x * 0#
           ≈⟨ +-cong^l $ zero^r x ⟩
     - (x * y) + 0#
           ≈⟨ +-identity^r (- (x * y)) ⟩
     - (x * y)
           ■
```

6.5.4 Properties of Ring

Properties of rings can be found in number theory and algebraic geometry, where they are used to study algebraic curves, surfaces, and other geometric objects. They help in understanding the properties of prime numbers, factorization, and algebraic varieties [Pedrouzo-Ulloa et al.(2021)]. 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 (-‿distrib^l-* 1# x ) ⟩
  - (1# * x)  ≈⟨ -‿cong ( *-identity^l x ) ⟩
  - x         ■
```

2.
```
x+x≈x⇒x≈0 : ∀ x → x + x ≈ x → x ≈ 0#
x+x≈x⇒x≈0 x eq = begin
  x           ≈⟨ sym(+-identity^r x) ⟩
  x + 0#      ≈⟨ +-cong^l (sym (-‿inverse^r x)) ⟩
  x + (x - x) ≈⟨ sym (+-assoc x x (- x)) ⟩
  x + x - x   ≈⟨ +-cong^r(eq) ⟩
  x - x       ≈⟨ -‿inverse^r 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)      ≈⟨ distrib^l x y (- z) ⟩
  x * y + x * - z  ≈⟨ +-cong^l (sym (-‿distrib^r-* 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      ≈⟨ distrib^r x y (- z) ⟩
  y * x + - z * x  ≈⟨ +-cong^l (sym (-‿distrib^l-* 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, program analysis and compiler optimization [Kozen(1997)]. Kleene algebra generalizes operations from regular expressions. The axiomization of the algebra of regular events was proposed in 1966 but it was in 1984, that 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.

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

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

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

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

where $\leq$ 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 defined using idempotent semiring. In the Agda standard library, + and ∗ operations are used to denote addition and multiplication. To keep the same template, ⋆ 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 ⋆ for the unary star operator. The field `isIdempotentSemiring` makes an idempotentSemiring with the 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.

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
```

By characterizing morphisms in Kleene algebra, researchers can gain insights into the structural properties between two regular languages.

7.3 Morphism composition

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

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

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

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

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

The composition of monomorphism and isomorphism can be defined similar to homomorphism and can be found in the 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.starExpansive⁽ˡ⁾  , L.starExpansive⁽ˡ⁾)
 ↪  <*> x)
                    , (λ x → (K.starExpansive⁽ʳ⁾  , L.starExpansive⁽ʳ⁾)
 ↪  <*> x)
    ; starDestructive = (λ a b x x₁ → (K.starDestructive⁽ˡ⁾  ,
 ↪  L.starDestructive⁽ˡ⁾) <*> a <*> b <*> x <*> x₁)
                      , (λ a b x x₁ → (K.starDestructive⁽ʳ⁾  ,
 ↪  L.starDestructive⁽ʳ⁾) <*> 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. The proofs are adapted from [Kozen(1997)]. Let $(K, +, *, ^*, 0, 1)$ be a Kleene algebra then:

1.

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

$$x + x^* = x^*$$

This property allows for the repetition and concatenation of language constructs. An application of this property is found in the development of pattern-matching algorithms in text processing and computational linguistics [Ulus(2018)].

```
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 ⋆                           ■

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 ⋆                               ■
```

2. The absorption property is used in the simplification of regular expressions.

$$x * x^* + x^* = x^*$$

$$x^* + x^* * x = x^*$$

They are used in the analysis of language hierarchies and the development of language recognition algorithms in natural language processing and computational linguistics [Desharnais et al.(2004)].

```
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 ⋆                              ■


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 ⋆                   ■
```

3. Zero and one element By the zero element property simplifies to x^*. That is the zero (and one) element does not alter the result when combined with language elements and their Kleene star. These properties are also used in regular expressions and

pattern-matching algorithms.

$$0 + x + x^* = x^*$$

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

```
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 ⋆)   ≈⟨ +-identity¹ ((x + x ⋆)) ⟩
  (x + x ⋆)        ≈⟨ x+x⋆≈x⋆ x ⟩
  x ⋆              ■
```

```
1+x+x⋆≈x⋆ : ∀ x → 1# + x + x ⋆ ≈ x ⋆
1+x+x⋆≈x⋆ x = begin
  1# + x + x ⋆    ≈⟨ +-assoc _ _ _ ⟩
  1# + (x + x ⋆)  ≈⟨ +-cong¹ (x+x⋆≈x⋆ x) ⟩
  1# + x ⋆        ≈⟨ 1+x⋆≈x⋆ x ⟩
  x ⋆             ■
```

4. To prove the property:

$$x^* * x^* + x^* = x^*$$

We need to prove that $x^* + x * x^* + x^* \approx x^*$. This proof can be used in Agda as shown in the proof below.

```
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 ⋆) ≈⟨ +-cong^l (+-comm _ _) ⟩
  x ⋆ + (x ⋆ + x * x ⋆) ≈⟨ sym (+-assoc _ _ _) ⟩
  x ⋆ + x ⋆ + x * x ⋆   ≈⟨ +-cong^r (+-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 =  starDestructive^l x (x ⋆) (x ⋆) (x⋆+xx⋆+x⋆≈x⋆
 ↪  x)
```

Here are some other notable properties of Kleene algebra along with their proofs in Agda.

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

```
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 ⋆) ≈⟨ +-cong^l (+-cong^r (*-cong^r (eq))) ⟩
  1# + (y * y ⋆ + y ⋆) ≈⟨ sym(+-assoc _ _ _) ⟩
  1# + y * y ⋆ + y ⋆   ≈⟨ starExpansive^r y ⟩
  y ⋆                  ■
```

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

```
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)
```

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

```
[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 Idris, the ring is defined without a multiplicative identity (Ring). However, in Agda, the ring has a multiplicative identity (IsRing) and rng is defined as ringWithoutOne that has no multiplicative identity (IsRingWithoutOne). This ambiguity in naming is also seen in literature. For example, [Ánh and Márki(1987)], [Jacobson(1956)], [Persson(1999)] define ring without the multiplicative identity and [Lehmann(1977)], [Geuvers et al.(2002)] define ring with multiplicative identity. Another example is the 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 definitions are equivalent [Shcherbacov(2003)], but they are structurally different. This chapter identifies and classifies five important problems that arise when defining types of 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 a 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 the lack of naming convention that is followed in naming algebraic structures and their properties. For example, consider algebraic structures ring and rng. Some mathematicians define a ring as an algebraic structure that is an Abelian group under addition, a monoid under multiplication, multiplication distributes over addition and has an annihilating zero. 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 or ring without unit. 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 is 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 $(A, *, 1)$ forms a monoid. The binary operation $*$ distributes over +, 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 IsRingWithoutOne 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
```

Lam in [Lam(1991)] discusses the confusion between "Ring" and "Rng" and how the term "Rng" refers to rings without unity, leading to potential misunderstandings in the literature. The issue is also seen in [Bosma et al.(1997)], [Jacobson(1956)], [Persson(1999)], [Lehmann(1977)], and [Geuvers et al.(2002)].

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, and multiplication distributes over addition. However, some mathematicians use the definition where multiplication is a semigroup. The same confusion also arises in defining

semiring and rig structures. [Rasuli(2022)] states that the term rig originated as a joke that it is similar to rng and is missing the alphabet "n" and "i" to represent that the identity does not exist for these structures. In Agda, the algebraic structure rig is defined as `SemiringWithoutOne` where one 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 Chapter 7, `starExpansive` and `starDestructive` names were invented (inspired from what is used in literature). Due to the lack of standardized names, many names can be coined for the same axiom.

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 $\cdot$ that forms a magma and satisfies the Latin square property. 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:

$$a \cdot x = b \tag{8.2.1}$$

$$y \cdot a = b \tag{8.2.2}$$

Another definition of quasigroup is given as type a (2,2,2) algebra in which a set Q and binary operations $\cdot, \backslash, /$. In Chapter 5, we define quasigroup with three binary operations that satisfy the below identities.

$$x \cdot (x \backslash y) \;=\; y \tag{8.2.3}$$

$$x \setminus (x \cdot y) = y \tag{8.2.4}$$

$$(y \,/\, x) \cdot x = y \tag{8.2.5}$$

$$(y \cdot x) \,/\, x = y \tag{8.2.6}$$

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. We can show that both the definitions of quasigroup are isomorphic to each other. The proof is adapted from [Shcherbacov(2003)].

1. Let $(Q,\cdot)$ be a quasigroup (satisfies Latin square property). Since $a\cdot x = b$, we can associate this with an operation such that $a\cdot x = b \mapsto a\backslash b = x$. We can substitute this in equation $a\cdot x = b$ to get $a\cdot(a\backslash b) = b \forall a, b \in Q$ which is 8.2.3.

   Similarly, $y\cdot a = b \mapsto a/b = y$. Substituting this we get $(a/b)\cdot a = y$ which is 8.2.5.

   Equation 8.2.4 and 8.2.6, follows from the definition of operation \ and /. That is $x\backslash(x\cdot y) = y \mapsto x\cdot y = x\cdot y$ and $(y\cdot x)/x \mapsto y\cdot x = y\cdot x$.

2. Let $(Q,\cdot,\backslash,/)$ be a quasigroup (satisfies division). We need to show the existence of a unique solution of equation $a\cdot x = b$ and $y\cdot a = b$.

   To prove the existence of a solution, let $x = a\backslash b$ and $y = b/a$. Substituting x and y in above equation, we get $a\cdot x = a\cdot(a\backslash b) = b$ from equation 8.2.3 and $y\cdot a = (b/a)\cdot a = b$ from equation 8.2.5.

   To prove the uniqueness of the solution, we can assume there exist two solutions x_1 and x_2 that is $a\cdot x_1 = b$ and $a\cdot x_2 = b$. Then $x_1 = a\backslash b$. Therefore we get $x_1 = a\backslash b = a\,(a\cdot x_2) = x_2$ from equation 8.2.5

   Similarly, if y_1 and y_2 two solutions that is $y_1\cdot a = b$ and $y_2\cdot a = b$, we get $y_1 = b/a = (y_2\cdot a)/a = y_2$ from equation 8.2.6.

In the algebra hierarchy, a Loop is an algebraic structure that is a quasigroup with identity. It can be observed the same problem persists in 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 a suboptimal name. One possible solution to this problem is to define the structures in different modules and import restrict them when used. 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, the loop is also defined as a type (2,2,2) algebra in the Agda standard library.

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 with 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 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
  ; distrʳ        = proj₂ distrib
  ; zeroˡ         = zeroˡ
  }
```

The above technique will remove the redundant equivalence relation. However, it

fails to express the structure in terms of two or more structures that are commonly used in literature and other systems. Agda 2.0 removed redundancy by unfolding the structure. This solution should ensure that the structure 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, the same algebraic structure can be derived from two or more structures. One such example is Nearring. Nearring is a group under addition, a monoid under multiplication, and multiplication right distributes over addition. In this case, nearring is defined using two algebraic structures group and monoid. Another 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 that has an additive inverse. In Agda nearring is defined in terms of quasiring with additive inverse as:

```
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 such that multiplication is a semigroup. This can be attributed to the problem of ambiguity. It can be analyzed that having two different definitions for the same structure is not a good practice. If nearring is defined using quasiring then it should also give an instance of an additive group without having to construct it when using the above formalization. This solution might solve the problem at first but in practice, this becomes tedious and may go to a point at which this can be impractical especially when formalizing structures at higher levels in the algebra hierarchy.

8.5 Equivalent structures

Consider the example of idempotent commutative monoid and bounded semilattice. Both idempotent commutative monoid and bounded semilattice are equivalent structures that are also of same type. It is redundant to define two different structures from different hierarchies. Instead, in Agda, aliasing may be used to say that the bounded

semilattice is same as the 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.

8.6 Summary

The table below provides a summary of the issues classified and recommendations to mitigate the issues.

Table 8.1: Problem in programming algebra

Issue	Recommendation

Ambiguity in naming	• Encouraging standardization by adopting commonly accepted definitions and terminology. • When defining terms in standard libraries or in literature, provide contextual clarification when introducing potentially ambiguous terms.
Equivalent but structurally different	• Provide proofs for equivalence in algebraic structures. • Make the system capable of overloading names. • Provide commonly used definitions in different modules to avoid conflict.
Redundant field in structural inheritance	• Unfold structure to remove redundant fields. • Provide all instances of structure that were available before unfolding.
Identical structures	• Make a note of diamonds in the hierarchy when defining structures. • Provide the instance of structure that was not defined with the field.

Equivalent structures	• Give aliasing instead of defining the structure again. • Provide proof if the structures are isomorphic to each other.

Chapter 9

Conclusion and Future Work

The primary of this work was to study types of 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 of 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, and Chapter 3 provide 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 defining 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 the 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

[Abel(2012)] Andreas Abel. 2012. Agda: Equality. https://www2.tcs.ifi.lmu.de/~abel/Equality.pdf

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

[Al-hassy et al.(2019)] Musa Al-hassy, Jacques Carette, and Wolfram Kahl. 2019. A Language Feature To Unbundle Data At Will (Short Paper). , 14–19 pages.

[Alexander et al.(2023)] Katz Alexander, Williams Christopher, and Khim Jimin. 2023. Russell'S Paradox. https://brilliant.org/wiki/russells-paradox/ [Online; accessed 16-January-2023].

[Ánh and Márki(1987)] PN Ánh and L Márki. 1987. Morita Equivalence For Rings Without Identity. *Tsukuba journal of mathematics* 11, 1 (1987), 1–16.

[Baanen(2022)] Anne Baanen. 2022. Use And Abuse Of Instance Parameters In The Lean Mathematical Library. *arXiv preprint arXiv:2202.01629* (2022).

[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

[Bertot and Cast'eran(2013)] Yves Bertot and Pierre Cast'eran. 2013. Interactive Theorem Proving And Program Development: coq'Art: the Calculus Of Inductive Constructions.

[Bocquet(2020)] Rafaël Bocquet. 2020. Coherence Of Strict Equalities In Dependent Type Theories.

[Bosma et al.(1997)] Wieb Bosma, John Cannon, and Catherine Playoust. 1997. The Magma Algebra System I: the User Language. *Journal of Symbolic Computation* 24, 3-4 (1997), 235–265.

[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.

[Brady(2013)] Edwin Brady. 2013. Idris, A General-Purpose Dependently Typed Programming Language: Design And Implementation. *Journal of functional programming* 23, 5 (2013), 552–593.

[Brady(2021)] Edwin Brady. 2021. Idris 2: quantitative Type Theory In Practice. *ArXiv* abs/2104.00480 (2021). `https://api.semanticscholar.org/CorpusID:232478760`

[Brady(2022)] Edwin Brady. 2022. Idris: A Language For Type-Driven Development.

[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 et al.(2021)] Jacques Carette, William M Farmer, Michael Kohlhase, and Florian Rabe. 2021. Big Math And The One-Brain Barrier: the Tetrapod Model Of Mathematical Knowledge. *The Mathematical Intelligencer* 43 (2021), 78–87.

[Carette et al.(2018)] Jacques Carette, Russell O'Connor, and Yasmine Sharoda. 2018. Building on the diamonds between theories: theory presentation combinators. *arXiv preprint arXiv:1812.08079* (2018).

[Community(2023)] The Agda Community. 2023. Agda Standard Library. `https://github.com/agda/agda-stdlib`

[Coquand(1986)] Thierry Coquand. 1986. *An Analysis Of Girard'S Paradox.* Ph. D. Dissertation. INRIA.

[Danielsson et al.(2011)] Nils Anders Danielsson, Ulf Norell, SC Mu, S Bronson, D Doel, P Jansson, and LT Chen. 2011. The Agda Standard Library. *Url: http://www. cs. nott. ac. uk/nad/repos/lib* (2011).

[de Moura et al.(2015)] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*. Springer, 378–388.

[Deng et al.(2016)] Fang-an Deng, Lu Chen, ShouHeng Tuo, and ShengZhang Ren. 2016. Characterizations Of N(2,2,0) Algebras. *Algebra* 2016 (2016).

[Desharnais et al.(2004)] Jules Desharnais, Bernhard Möller, and Georg Struth. 2004. Modal Kleene Algebra And Applications-A Survey. (2004).

[Didurik and Shcherbacov(2018)] Natalia N Didurik and Victor A Shcherbacov. 2018. Some Properties Of Neumann Quasigroups. *arXiv preprint arXiv:1809.07095* (2018).

[Ebner et al.(2017)] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A Metaprogramming Framework For Formal Verification. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–29.

[Eremondi et al.(2022)] Joseph Eremondi, Ronald Garcia, and Éric Tanter. 2022. Propositional Equality For Gradual Dependently Typed Programming. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 165–193.

[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 et al.(2002)] Herman Geuvers, Randy Pollack, Freek Wiedijk, and Jan Zwanenburg. 2002. A Constructive Algebraic Hierarchy In Coq. *Journal of Symbolic Computation* 34, 4 (2002), 271–286.

[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.

[Grove(2012)] Larry C Grove. 2012. *Algebra*. Courier Corporation.

[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.

[Jacobson(1956)] Nathan Jacobson. 1956. *Structure Of Rings*. Vol. 37. American Mathematical Soc.

[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.

[Judson(2020)] Thomas W Judson. 2020. *Abstract Algebra: theory And Applications*.

[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.

[Khan et al.(2015)] Majid Ali Khan, Nazeeruddin Mohammad, Shahabuddin Muhammad, and Asif Ali. 2015. A Mining Based Approach For Efficient Enumeration Of Algebraic Structures. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 1–6.

[Khathuria et al.(2021)] Karan Khathuria, Giacomo Micheli, and Violetta Weger. 2021. On The Algebraic Structure Of E_pˆ(M) E p (m) And Applications To Cryptography. *Applicable Algebra in Engineering, Communication and Computing* 32 (2021), 495–505.

[Kidney(2020)] Donnacha Oisín Kidney. 2020. Finiteness In Cubical Type Theory. `https://doisinkidney.com/pdfs/masters-thesis.pdf`

[Kohlhase and Rabe(2021)] Michael Kohlhase and Florian Rabe. 2021. Experiences From Exporting Major Proof Assistant Libraries. *Journal of Automated Reasoning* 65, 8 (2021), 1265–1298.

[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.

[Lam(1991)] Tsit-Yuen Lam. 1991. *A First Course In Noncommutative Rings*. Vol. 131. Springer.

[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).

[Lehmann(1977)] Daniel J Lehmann. 1977. Algebraic Structures For Transitive Closure. *Theoretical Computer Science* 4, 1 (1977), 59–76.

[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

[Martin-Löf and Sambin(1984)] Per Martin-Löf and Giovanni Sambin. 1984. *Intuitionistic Type Theory*. Vol. 9. Bibliopolis Naples.

[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

[Murray(2022)] Zachary Murray. 2022. Constructive Analysis In The Agda Proof Assistant.

[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.

[Norell(2007)] Ulf Norell. 2007. *Towards A Practical Programming Language Based On Dependent Type Theory*. Vol. 32. Chalmers University of Technology.

[Paulin(2021)] Alexander Paulin. 2021. Introduction To Abstract Algebra (math 113).

[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

[Pedrouzo-Ulloa et al.(2021)] Alberto Pedrouzo-Ulloa, Juan Ramón Troncoso-Pastoriza, Nicolas Gama, Mariya Georgieva, and Fernando Pérez-González. 2021. Revisiting Multivariate Ring Learning With Errors And Its Applications On Lattice-Based Cryptography. *Mathematics* 9, 8 (2021), 858.

[Persson(1999)] Henrik Persson. 1999. An Application Of The Constructive Spectrum Of A Ring. *Type Theory and the Integrated Logic of Programs. Chalmers University and University of Göteborg* (1999).

[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.

[Rasuli(2022)] Rasul Rasuli. 2022. Anti-Fuzzy Bi-Ideals In Semirings Under S-Norms. *Fuzzy Optimization and Modeling Journal* 3, 4 (2022), 19–31.

[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 and Tarlecki(2012)] Donald Sannella and Andrzej Tarlecki. 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

[Shcherbacov(2003)] Victor Shcherbacov. 2003. Elements Of Quasigroup Theory And Some Its Applications In Code Theory And Cryptology. *Lectures in Prague, Czech Republic* (2003).

[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(2016a)] Mikael Stener. 2016a. Moufang Loops : General Theory And Visualization Of Non-Associative Moufang Loops Of Order 16. https://api.semanticscholar.org/CorpusID:125264680

[Stener(2016b)] Mikael Stener. 2016b. 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.

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

[Team(2023a)] Agda Development Team. 2023a. Sort System. `https://agda.readthedocs.io/en/v2.6.3.20230805/language/sort-system.html`

[Team(2023b)] Agda Development Team. 2023b. Universe Levels. `https://agda.readthedocs.io/en/v2.6.1.3/language/universe-levels.html`

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

[Ulus(2018)] Dogan Ulus. 2018. *Pattern Matching With Time: theory And Applications*. Ph. D. Dissertation. Universite Grenoble Alpes.

[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.