Extending CAS with Algebraic Reductions

Zongzhe Yuan
Christ's College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
UNITED KINGDOM

Email: zy272@cl.cam.ac.uk

April 18, 2018

Declaration

I Zongzhe Yuan of Christ's College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 14,235

Signed:

Date:

Abstract

This is the abstract. Write a summary of the whole thing. Make sure it fits in one page.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction to the Path Problem and Algebraic Solution

The path problem has always fascinated mathematicians and computer scientists. At the very beginning, programmer and scientists designed algorithms to solve each of the path problem. The most famous path problem is the shortest distance problem and there are several well-known algorithm that can solve such a problem: Dijkstra's algorithm, BellmanFord algorithm, A* search algorithm and FloydWarshall algorithm. People use different primitive metrics and various complicated algorithms to solve different path problems.

However, such an approach has its obvious shortcomings. Even at some point, designing a new algorithm can "steal" the ideas of the original algorithm, people must design a completely new independent algorithm in the face of each new problem (new metric), and this makes it difficult to have a generic (or framework) approach to solve this type of problem. Even if the path problem has minor changes to the problem, it is difficult for people to solve the new problem by slightly modifying existing algorithms.

Hence, lots of predecessors have found the algebraic approaches to work around this kind of problem. Using the knowledge of abstract algebra, people find that the routing problem (path problem) can be represent using a data structure called "semiring" $(S, \oplus, \otimes, \bar{0}, \bar{1})$[1, 2, 3, 4, 5]. For example, the popular "shortest path problem" can be represented as $(S, min, +, \infty, 0)$[4] and the "maximal capacity path problem" can be represented as $(S, max, min, 0, \infty)$.

For each path problem that represented as a semiring, we can construct a corresponding matrix semiring that represent the concrete problem (the edges and the distances for the shortest path problem for example). Then using matrix multiplication and stability of the closure (the semiring), we can solve the real problem of each concrete path problem. However, the simple matrix approach can only solve the ”trivial” path problem. Those complicated problem, for example the widest shortest path problem that constructed from the shortest path problem semiring and the widest path problem (maximal capacity path problem) semiring using lexicograhpic product, can’t be solved by this ”traditional” theory approach. Some times even the method can find an optimal solution, there is no guarantee to find all optimal solutions using the classical method.

Therefore, people have found a non-classical theory of algebraic path finding method, so that algebras that violated the distribution law can be accepted. This non-classical theory can handle the problem the simple classical theory can’t handle, such as the problem that can’t be solved by Dijkstra or Bellman-Ford. This kind of method is dedicated to finding the local optimal solution at first, and then the local optimal solution is exactly the same as the global optimal solution by some verification or some addition restriction on the computation. For example, the famous protocol, the routing information protocol which is based on distributed Bellman-Ford algorithms is one of the non-traditional theory.

However, even so, RIP will also have a series of problems. For example, when a node does not have edges connected to it, the RIP matrix calculation (without pre-setting the maximum number of calculation steps) will continue infinitely. Even if the maximum number of calculation steps is set in advance, RIP still has some deficiencies in efficiency.

So while we use another protocol BGP (Border Gateway Protocol), we add an annihilator to the entire complex semiring. At the same time, we found that when we represent the path, we may have a loop path (a node in the path has been passed more than once). So we need our problem set to change from the original path to elementary path (A path p is elementary if no node is repeated). In this process, we performed two operations $S \to S$, and here we call it reduction in general, which comes to the main problem of our project.

1.2 Reduction

Algebraic reduction, introduced by Ahnont Wongseelashote in 1977[2] is an unary operator for a given set of problem, $reduce : S \longrightarrow S$. It has several properties, satisfying $reduce(\emptyset) = \emptyset$, $\forall A \in S, reduce(reduce(A)) = reduce(A)$ (which is called idempotent property) and $\oplus : S \times S \to S, \forall A, B \in S, reduce(reduce(A) \oplus B) = reduce(A \oplus B) = reduce(A \oplus reduce(B))$ (which is left and right invariant property) and this paper will discuss these properties in the later section.

It is hard to specify the reduced problem set in the most programming language. However, in the world of logic and those programming language that can be used to prove properties, programmer can represent the reduced problem set as $\{x | x \in S, reduce(x) = x\}$ which is also a subset of the original problem set. The idea to represent the reduced set explicitly is to form a pair $< x, Pr(x) >$ where $x \in S$ is the element in the set and $Pr(x) : r(x) = x$ is the proof that the element is in the subset (the element will not change after the reduced function, otherwise it will be reduced).

The first example of the reduction is id which maps all the stuff to itself. Another example is the min-set where $min_{\leq}(x) = \{x \in S | \forall y \in S, \neg(y < x)\}$. Regarding to $\mathcal{P}(S)$ it works well with $\cup$ that the min-set contain all the elements and remove the element that is non-trivial set. And this reduction is used in the construction of elementary path.

1.3 Motivation

Wongseelashote defined the reduction operator in his paper[2]. However the definition there is not constructive and it is the traditional reduction. Gurney and L11 claimed reductions could be used for ”non-traditional” reductions such as elementary paths and combining elementary paths with lexicographic product [6]. However, Gurney and L11 never worked out the details, and the reduction there is still not constructive.

It is worth mentioning that, the algebraic approaches that using the matrix as the computation rely on the property of the operators a lot, for example, the left and right distributivity of the semiring. However, most of the cases mentioned above are aimed at some simple problems, or the ideal situations. In reality, we need to face the problem that, in the most time for the complex path problem (especially for lexicographic product). The CAS system can derive most of the properties for the new semirings from the original ”simple” simirings. However, sometimes the problem set is not the original problem set the provided to us, but the subset of it. As I mentioned before, as the problem that consist of the lexicographic product of the shortest path problem semiring and the maximum capacity semiring, there exists some path that have 0 capacity which shouldn't be concerned in the solution of the problem. This kind of reduction can be represent as a unary operation on the original problem set in our paper.

Another example, when we are doing path problems, for the most time the path that has negative value or have infinite value is not quite interesting. However, the operation we defined there is on the whole families of object. When we are defining some data structure, like semiring, we want to know that the properties (like commutative, selective and etc..) of the proper subset (the set of objects after reduction) will hold or not, or for some cases we can't do further calculation.

As the method mentioned previously, the algebraic approach to the path problem is depend on the properties of semirings. If one, or some of the properties don't hold for the semiring, the algebraic approach can't guarantee to find the optimal solution (depends on the semiring structure and the operations). Thus, after we applying those reductions on the original problem set, there is no guarantee that the original properties will still hold for the new subset of problem set, and it comes to our point. The existing CAS system doesn't have the functionality to derive and prove the properties for the reduced problem. Hence, I want to figure out the relationship between the reduction and those properties for those operators on the problem set.

Furthermore, in most programming languages, it is extremely difficult to express the reduction properly, on contrast, we can represent the reduction as a proof or proposition in our proof world. The second goal to the project is to figure out the friendly-extraction to those reduction.

Chapter 2

Background

This chapter will introduce several basic concept, and some related ideas that will help us to understand the main idea of the project.

2.1 Basic Definition

In the world of logic, we need to define several basic concepts before we are really getting started. We can simply understand that a Type is a collection of terms which have the same ”properties”. Under a Type S, we can define a binary relationship (in most cases we will define the binary relationship of equality $=_S$ at first). In mathematics, A binary relation R in an arbitrary Type S (here I restrict the relationship to be inside a single Type, or say the element from the same type) is a collection of ordered pairs of elements of set (type) S, which is a subset of the Cartesian product $S \times S$.

In order to link the mathematical concept with the proposition in logic, I provides each relation a representation (hold or not) as a boolean value, which is regarded as a property in Coq.

$\Lambda S.brel : S \rightarrow S \rightarrow bool$.

Next we should define the operators that exist on a given Type:

binary operation : $\Lambda S.binary\_op : S \rightarrow S \rightarrow S$

unary operation : $\Lambda S.unary\_op : S \rightarrow S$.

2.1.1 Properties for the Relationship

For our definition of binary relationship, we need to focus on a few properties related to it.
Here we have four properties for a binary relationship for type S. As it is mentioned in the previous section, in order to define the proposition in the logic proof, the binary relationship will return a boolean value indicating that the relationship is holding or not, here we provide a arbitrary type variable S.
Then for a given relationship $=_S$: $S \to S \to bool$, we should define:
reflexivity: $\forall a \in S, a =_S a \equiv true$,
symmetric : $\forall a, b \in S, a =_S b \equiv b =_S a$,
transitivity: $\forall a, b, c \in S, a =_S b \equiv true \wedge b =_S c \equiv true \to a =_S c \equiv true$,
congruence : $\forall a, b, c, d \in S, a =_S b \equiv true \to c =_S d \equiv true \to a =_S c \equiv b =_S d$

2.1.2 Properties for the Operator

Next we need to discuss some of the properties of the unary operator and the binary operator that we have defined.

Properties for Unary Operator

For the unary operator we have defined, we want it to have the following properties, or we will analyse the unary operator we have defined from the following perspectives.
In fact, our unary operator is used to represent reduction, so our discussion of the properties of the unary operator is actually a discussion of some of the properties of reduction.
There are six properties that we need to be discussed. Three of them are of the same properties as the reduction mentioned in the previous section, here we give a arbitrary type variable S, a binary relationship in S : $=_S$ and our unary operation u_S.
congruence : $\forall a, b \in S, a =_S b \to u_S(a) =_S u_S(b)$,
idempotent : $\forall a \in S, u_S(u_S(a)) =_S u_S(a)$,
left_invariant: $\Lambda \oplus_S : S \to S \to S, \forall a, b \in S, u_S(a) \oplus_S b =_S a \oplus_S b$,
right_invariant: $\Lambda \oplus_S : S \to S \to S, \forall a, b \in S, a \oplus_S u_S(b) =_S a \oplus_S b$,
preserve_id : $\forall i \in S$, if i is the identity for type S, equality $=_S$ and a binary operator $\oplus_S$, $(\forall a \in S, a \oplus_S i =_S a =_S i \oplus_S a)$ then $u_S(i) =_S i$,

preserve_annihilator : $\forall a \in S$, if i is the annihilator for type S, equality $=_S$ and a binary operator $\oplus_S$, ($\forall b \in S, a \oplus_S b =_S a =_S b \oplus_S a$) then $u_S(b) =_S b$.

Properties for Binary Operator

Also for the binary operator we have defined, we hope that we will focus on the following seven properties of it. For a given type variable S, the binary relationship $=_S$ and our pre-defined binary operator $\oplus_S : S \times S \to S$, we have:
congruence: $\forall s_1, s_2, t_1, t_2 \in S, s_1 \equiv t_1 \wedge s_2 \equiv t_2 \to s_1 \oplus_S s_2 \equiv t_1 \oplus_S t_2$,
idempotent : $\forall a \in S, a \oplus_S a =_S a$,
associativity: $\forall a, b, c \in S, a \oplus_S (b \oplus_S c) \equiv (a \oplus_S b) \oplus_S c$,
commutativity: $\forall a, b \in S, a \oplus_S b = b \oplus_S a$,
selectivity: $\forall a, b \in S, a \oplus_S b \in \{a, b\}$,
hasId: $\exists \bar{0} \in S, \forall a \in S, a \oplus_S \bar{0} = a = \bar{0} \oplus_S a$,which means $\bar{0}$ is the identity for type S, equality $=_S$ and the binary operator $\oplus_S$,
hasAnn: $\exists \bar{1} \in S, \forall a \in S, a \oplus_S \bar{1} = \bar{1} = \bar{1} \oplus_S a$, which means $\bar{1}$ is the identity for type S, equality $=_S$ and the binary operator $\oplus_S$,
In this paper, we will mostly concentrate on those properties that are holding or not in the reduced set of problem.

2.2 Direct Product and its Properties

Here we need to mention Product Type and some properties about Product Type.
CAS will use pair to combine two semirings (or just semigroups) to form a new data structure. Therefore we must discuss the relationship between the properties of the single semirings and the properties they have been combined with.
First, as with a given Type and its binary relationship, we need to define a binary relationship to describe the relationship between product type terms.
$\Lambda S, T, =_s, =_t .brel\_product : S * T \to S * T \to bool$,
$:= \lambda(s_1, t_1)(s_2, t_2).s_1 =_s s_2 \wedge t_1 =_t t_2$, (we can write this binary relationship as $=_S \times =_T$).
At the same time, we should give an expansion on existing unary operators and binary operators for product.
For two given type S and T, unary operators u_S and u_T, binary operators $\oplus_S$ and $\oplus_T$:

uop_product: $S * T \rightarrow S * T := \lambda(s,t), (u_S(s), u_T(t))$,
bop_product: $S * T \rightarrow S * T \rightarrow S * T := \lambda(s_1, t_1)(s_2, t_2), (s_1 \oplus_S s_2, t_1 \oplus_T t_2)$, (we can write this new binary operator as $(\oplus_S, \oplus_T)$).

2.2.1 Properties for the Relationship

In fact, we also focus on whether or not some of the properties of the binary relationship previously defined apply to the product type. Although the proof of this part is not the key to our project. Fortunately, the CAS system has systematically proved the product type properties, including binary relationship and binary operator.
For two given type S and T and their corresponding binary relationship $=_S$ and $=_T$,
If both $=_S$ and $=_T$ are reflexivity, then $=_S \times =_T$ is reflexivity,
If both $=_S$ and $=_T$ are symmetric, then $=_S \times =_T$ is symmetric,
If both $=_S$ and $=_T$ are transitivity, then $=_S \times =_T$ is transitivity,
If both $=_S$ and $=_T$ are congruence, then $=_S \times =_T$ is congruence.

2.2.2 Properties for the Operator

As mentioned above, fortunately the CAS system has given us a systematic proof of the binary operator of the product type.
For two given type S and T, binary operators $\oplus_S$ and $\oplus_T$:
If both $\oplus_S$ and $\oplus_T$ are congruence, then $(\oplus_S, \oplus_T)$ is congruence,
If both $\oplus_S$ and $\oplus_T$ are commutativity, then $(\oplus_S, \oplus_T)$ is commutativity,
If both $\oplus_S$ and $\oplus_T$ are associativity, then $(\oplus_S, \oplus_T)$ is associativity,
If both $\oplus_S$ and $\oplus_T$ have Id $\bar{0}_S$ and $\bar{0}_T$, then $(\oplus_S, \oplus_T)$ has Id $(\bar{0}_S, \bar{0}_T)$,
If both $\oplus_S$ and $\oplus_T$ have Ann $\bar{1}_S$ and $\bar{1}_T$, then $(\oplus_S, \oplus_T)$ has Ann $(\bar{1}_S, \bar{1}_T)$.
It is worth mentioning that although both $\oplus_S$ and $\oplus_T$ may be selective, this does not lead to the conclusion that $(\oplus_S, \oplus_T)$ is a selective. The explanation of selective will be mentioned later in the lexicographic product section.

2.3 Semiring and its Properties

In abstract algebra, a semiring is a data structure $(S, \oplus, \otimes, \bar{0}, \bar{1})$ where S is a set (Type) and $\oplus, \otimes$ are two operators : $S \times S \to S$.

$(S, \oplus)$ is a commutative semigroup (has associative property) and $(S, \otimes)$ is a semigroup:
$\forall a, b, c \in S, a \oplus (b \oplus c) = (a \oplus b) \oplus c, a \oplus b = b \oplus a,$
$\forall a, b, c \in S, a \otimes (b \otimes c) = (a \otimes b) \otimes c.$

$\oplus, \otimes$ are also left and right distributive on S :
$\forall a, b, c \in S : a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c),$
$\forall a, b, c \in S : (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c).$

$\bar{0}$ is the identity of $\oplus$ and $\bar{1}$ is the identity of $\otimes$:
$\forall a \in S, a \oplus \bar{0} = a = \bar{0} \oplus a,$
$\forall a \in S, a \otimes \bar{1} = a = \bar{1} \otimes a.$

Finally, $\bar{0}$ is the annihilator of $\otimes$:
$\forall a \in S, a \otimes \bar{0} = \bar{0} = \bar{0} \otimes a.$
Some definition will include that $\bar{1}$ is the annihilator of $\oplus$:
$\forall a \in S, a \oplus \bar{1} = \bar{1} = \bar{1} \oplus a.$

2.4 Matrix Semiring and Stability

For each semiring $(S, \oplus, \otimes, \bar{0}, \bar{1})$ (that represent the rule of a path problem), we can define a matrix semiring $(M_n(S), \oplus, \otimes, \bar{J}, \bar{I})$ to represent the concrete path problem.
$M_n(S)$ is a $n \times n$ matrices over S,
$(A \oplus B)(i, j) = A(i, j) \oplus B(i, j),$
$(A \otimes B)(i, j) = \bigoplus_{1 \leq q \leq n} A(i, q) \otimes B(q, j),$
$\bar{J}(i, j) = \bar{0},$
$$\bar{I}(i, j) = \begin{cases} \bar{1} & i = j \\ \bar{0} & otherwise \end{cases}$$
So here we can easily use this matrix to encode any specific path problem and use matrix multiplication to calculate our answer.

For a graph $G = (V, E)$ a directed graph and a weight function $w \in E \rightarrow S$, we can define the weight of a path $p = i_1, i_2, \dots i_k$ is $w(p) = w(i_1, i_2) \otimes w(i_2, i_3) \otimes \cdots \otimes w(i_{k-1}, i_k)$, while the empty path is given the weight of $\bar{1}$. Then we can define the adjacency matrix A as
$A(i,j) = \begin{cases} w(i,j) & (i,j) \in E \\ \bar{0} & otherwise \end{cases}$
And our problem will be represent as $A^*(i,j) = \bigoplus_{p \in \pi(i,j)} w(p)$.

However, our calculations depend on the properties of the semiring.
$a \in S$,we define the powers a^k as, $a^0 = \bar{1}$ and $a^{k+1} = a \otimes a^k$.
$a \in S$,we define the closure a^* as, $a^{(k)} = a^0 \oplus a^1 \dots a^k$ and $a^* = a^0 \oplus a^1 \dots a^k \oplus \dots$.
Here we say, if there exists a q such that $a^q = a^{(q+1)}$, then a is q-stable, which means $a^q = a^*$.
And if we know that S is 0-stable, then $M_n(S)$ is $n-1$-stable (because we can ignore paths with loops, and that is our reduction!). This allows us to actually calculate at most $n-1$ steps when calculating path problems.
Therefore we can define the power and closure on the matrix semiring,
$A \in M_n(S)$,we define the powers A^k as, $A^0 = \bar{I}$ and $A^{k+1} = A \otimes A^k$.
$A \in M_n(S)$,we define the closure A^* as, $A^{(k)} = A^0 \oplus A^1 \dots A^k$ and $A^* = A^0 \oplus A^1 \dots A^k \oplus \dots$.

Hence we have $\pi(i,j)$ which is the set of paths from i to j, then $\pi^k(i,j)$ will be the set of paths from i to j with exactly k arcs, then $\pi^{(k)}(i,j)$ will be the set of paths from i to j with at most k arcs.
Then we have
$A^k(i,j) = \bigoplus_{p \in \pi^k(i,j)} w(p)$,
$A^{(k)}(i,j) = \bigoplus_{p \in \pi^{(k)}(i,j)} w(p)$ and
$A^*(i,j) = \bigoplus_{p \in \pi^*(i,j)} w(p)$.
It is worth mentioning that $A^*(i,j)$ may not be well defined, but if $M_n(S)$ is k stable, then for $A \in M_n(S)$, $A^*(i,j) = A^{(k)}(i,j)$.

2.5 Distributivity and Lexicographic Product

It is worth mentioning that the properties of distributivity (left and right) play an important role in the computation of routing problem. We've defined $M_n(S)$ the matrix and its related semiring on the given semiring $(S, \oplus, \otimes, \bar{0}, \bar{1})$. However we still need to check the properties of $M_n(S)$, to make sure that it is exactly a semiring. Consider about the distributivity properties, $M_n(S)$ is

left/right distributive if S has the distributivity properties.
But some times when we are defining some complex routing problem, such as the widest-shortest path problem, which is constructed by shortest path problem semiring and widest path problem semiring by lexicographic product, it is not guarantee that the new data structure still have the properties of distributivity.

Suppose $(S, \oplus_S)$ is a commutative and selective semigroup and $(T, \oplus_t)$ is a semigroup, then the lexicographic product of two semigroups $(S, \oplus_S)\bar{\times}(T, \oplus_T) = (S \times T, \oplus_{\bar{\times}})$ where

$$(s_1, t_1) \oplus_{\bar{\times}} (s_2, t_2) = \begin{cases} (s_1 \oplus_S s_2, t_1 \oplus_T t_2) & s_1 = s_1 \oplus_S s_2 = s_2 \\ (s_1 \oplus_S s_2, t_1) & s_1 = s_1 \oplus_S s_2 \neq s_2 \\ (s_1 \oplus_S s_2, t_2) & s_1 \neq s_1 \oplus_S s_2 = s_2 \end{cases}$$

As mentioned above, we used the lexicographic product when constructing the widest shortest path problem, and used the elementary path reduction. We also added a distinct annihilator in order to prevent excessive useless calculations.
These constructs make it difficult to speculate about the properties of our semiring (or new data structure), which is what our project is devoted to research that the relationship between reduction and semiring properties.

2.6 Global optimality VS Left/Right local optimality

On the other hand, we also need to discuss how to make our approach embrace the algebras that violate distributivity.
The global optimality: $A^*(i, j) = \bigoplus_{p \in \pi(i,j)} w(p)$,
The left local optimality which is the distributed Bellman-Ford algorithm: $L = (A \otimes L) \oplus \bar{I}$ which is $L(i, j) = \bigoplus_{q \in V} A(i, q) \otimes L(q, j)$,
The right local optimality which is the Dijkstra's algorithm: $R = (R \otimes A) \oplus \bar{I}$ which is $L(i, j) = \bigoplus_{q \in V} R(i, q) \otimes A(q, j)$,
note that with distributivity, $M_n(S)$ is a semiring and the three optimality problems are essentially the same, the local optimal solutions are global solutions :$A^* = L = R$,
However with out distributivity, those three solutions may all exists but all distinct, and that comes to the part of our project – discussing about the relationship between reductions and semirings.

2.7 Combinator for Algebraic System

Combinator for Algebraic system (CAS)[7] is introduced in $L11$. It is a language to design algebraic systems, in which many algebraic properties are automatically received and people can combine different operators to obtain a new semiring[7]. We can also generalize a more complex path problem (in other words, we can abstract a more complex path problem with this new semiring, such as the lexicographic products [5]). CAS can easily return the properties of those combined-operation semirings and it is already defined in Coq[8] (mentioned in $L11$ by Dr Timothy Griffin).

Chapter 3

Design and Implementation

3.1 Direct (Traditional) Reduction and Reduction Theory

3.1.1 Basic Definition

Initially, First of all, I first made a detailed analysis (proof) of the reduction (we call it traditional reduction) mentioned by Wongseelashote in his article [2].

For a given type S, a binary relationship $=_S$ on S, a unary operator and a binary operator u_S and $\oplus_S$, we define the traditional reduction as following:

$Pr(x : S) := u_S(x) =_S x$ which is the proof that, for a given element x of type S, x is in the reduced set.

$red\_Type := \{x : S \& Pr(x)\}$, which is a set that all the elements in the set are pairs of the element in S itself together with the proof that the element is in the reduced set.

$red\_eq : red\_Type \rightarrow red\_Type \rightarrow bool := \lambda p_1 p_2, projT_1(p_1) =_S projT_1(p_2)$.

Here $projT_1/projT_2$ are two auxiliary function that map the element/the proof part from the reduced set.

It is worth mentioning that since the definition follows Wongseelashote in his article [2], our unary operator(reduction) needs congruence and idempotent properties. And when we discuss the relation and properties of the unary operator (reduciton) and the binary operator, we need to have two properties left_invariant and right_invariant.

With these definitions we can discuss the nature of the binary relationship of traditional reduction.

3.1.2 Properties for the Relationship

For our given type S and binary relationship $=_S$, we assume that $=_S$ on S has such properties:

- reflexive
- symmetric
- transitive
- congruence

It is easy to prove that red_eq on red_Type has those four properties:

- Lemma red_ref : reflexive red_Type red_eq.
- Lemma red_sym : symmetric red_Type red_eq.
- Lemma red_cong : congruence red_Type red_eq red_eq.
- Lemma red_trans : transitive red_Type red_eq.

3.1.3 Properties for the Binary Operator

Before starting our proof, we need to prove an auxiliary Lemma.
Lemma Pr_br : $\forall(p1p2 : red\_Type), Pr(u_S(projT1(p1) \oplus_S projT1(p2)))$.
The significance of this Lemma is that for any two elements that exist in a reduced set, we take out its element separately and execute a binary operation, and finally do a reduction process, and the result is still in the reduced set.

Because our binary operator $\oplus_S$ is working on type S, we need a new reduced binary operator for our red_Type.
$red\_bop : binary\_op\ red\_Type := p1\ p2, existTPr(u_S(projT1(p1) \oplus_S projT1(p2)))(Pr\_br\ p1\ p2)$.
Here existT is also an auxiliary function that take the element of type S and the proof of the element to form the an element in the reduced set. (which is inverse to projT_1 and projT_2)
Here we assume that our binary operator $\oplus_S$ has such properties separately on S:

- congruence
- associativity

- commutativity
- selective
- idempontent
- has id $\bar{0}$
- has annihilator $\bar{1}$

It is easy to proof that red_bop has such properties on red_Type:

- Lemma red_bop_cong : $congruence \rightarrow bop\_congruence\ red\_Type\ red\_eq\ red\_bop.$
- Lemma red_bop_ass : $associativity \rightarrow bop\_associative\ red\_Type\ red\_eq\ red\_bop.$
- Lemma red_bop_comm : $commutativity \rightarrow bop\_commutative\ red\_Type\ red\_eq\ red\_bop.$
- Lemma red_bop_sel : $selective \rightarrow bop\_selective\ red\_Type\ red\_eq\ red\_bop.$
- Lemma red_bop_idem : $idempontent \rightarrow bop\_idempotent\ red\_Type\ red\_eq\ red\_bop.$
- Lemma red_bop_id :
  $uop\_preserves\_id\ S =_S \oplus_S u_S \rightarrow$
  $bop\_exists\_id\ S =_S \oplus_S \rightarrow$
  $bop\_exists\_id\ red\_Type\ red\_eq\ red\_bop.$
- Lemma red_bop_ann :
  $uop\_preserves\_ann\ S =_S \oplus_S u_S \rightarrow$
  $bop\_exists\_ann\ S =_S \oplus_S \rightarrow$
  $bop\_exists\_ann\ red\_Type\ red\_eq\ red\_bop.$

It is worth mentioning that although the proof in this direction looks very obvious, those proof are not all sufficient. For example, sometimes red_bop can be commutative on red_Type even if $\oplus_S$ is not commutative on S.
Fortunately, in our project we only care about whether our reduced set has the same properties when the original problem set has certain properties. So after our construction, we will specify the properties of the original problem set.

However, as we mentioned in the introduction, there are quite a few problems with this traditional reduction.
First of all, the proof part can not be well expressed in the outside world. We must know that

many programming languages do not have the ability to represent and prove proposition. Second, when we apply a lot of reduction at the same time, this traditional representation will make our reduced set winding and confusing. For example, suppose we have two reductions r_1 and r_2, then the reduced problem set that apply reduction 2 after reduction 1 on the problem set must be represent as $\{x \in \{y \in S\&r_1(y) = y\}\&r_2(x) = x\}$. At the same time we need to pay attention that the type of r_2 must be $\{y \in S\&r_1(y) = y\} \rightarrow \{y \in S\&r_1(y) = y\}$, but it should be $S \rightarrow S$ by our definition.

Because of the above problems, we have to consider other ways to represent our reduction (this is also the purpose and significance of our project).

3.2 Semigroup With Direct Reduction

The previous section mentioned that the purpose of our project is to clarify the relationship between reduction and semiring properties. In addition to the semiring's distribution relationship between the two operators, we can first think of it as two semigroups (then consider identity and annihilator issues). Hence, we need to give a primitive definition on semigroup (Some more detailed definitions will be added later as needed).
Initially, according to the properties of the binary relationship, we define a record that contains those properties:

```
Record eqv_proofs (S : Type) (eq : brel S) :=
{
  eqv_reflexive      : brel_reflexive S eq
; eqv_transitive     : brel_transitive S eq
; eqv_symmetric      : brel_symmetric S eq
; eqv_congruence     : brel_congruence S eq eq
; eqv_witness        : S
}.
```

The definition includes the four properties: reflexive, transitive, symmetric and congruence for a binary relationship on a given type that I mentioned above. At the same time, in order to ensure that our definition of semigroup is not a trivial semigroup (which means there is at least one element in the semigroup), we have added a witness property.

Next we need to define the properties of the binary operator in the semigroup:

```
Record semigroup_proofs (S: Type) (eq : brel S) (b : binary_op S) :=
{
  sg_associative    : bop_associative S eq b
; sg_congruence     : bop_congruence S eq b
; sg_commutative_d  : bop_commutative_decidable S eq b
}.
```

This includes associativity, congruence and commutativity. It is worth noting that since commutativity is not a property that semigroups must possess, we hereby conclude that a semigroup can be commutative or not commutative, but this must be decidable.
For a given type S, binary relationship $=_S$ and binary operator $\oplus_S$,
not_commutativity: $\exists a, b \in S, a \oplus_S b \neq_S b \oplus_S a$,
commutativity_decidable: $commutative\ S =_S \oplus_S + not\_commutative\ S =_S \oplus_S$.

Finally we can define our semigroup record as:

```
Record semigroup (S : Type) :=
{
   eq   : brel S
;  bop  : binary_op S
;  eqv  : eqv_proofs S eq
;  sgp  : semigroup_proofs S eq bop
}.
```

According to our previous definition of traditional reduction and its properties, and the semigroup record we defined, we can try to define our reduction semigroup. But first, because our reduction is a traditional reduction, we need to define the properties of reduction together as a record.

```
Record reduction_eqv_proofs (S : Type) (eq : brel S)
(r : unary_op S) :=
{
  rep_cong   : uop_congruence S eq r
; rep_idem   : uop_idempotent S eq r
}.
```

```
Record reduction_bop_proofs (S : Type) (eq : brel S)
(r : unary_op S) (b : binary_op S) :=
{
  rb_left   : bop_left_uop_invariant S eq b r
; rb_right  : bop_right_uop_invariant S eq b r
}.
```

The definitions of the two blocks contain two properties $\forall A \in S, reduce(reduce(A)) = reduce(A)$ and $\oplus : S \times S \to S, \forall A, B \in S, reduce(reduce(A) \oplus B) = reduce(A \oplus B) = reduce(A \oplus reduce(B))$ respectively, and it is also the definition of the reduction mentioned earlier by Ahnont Wongseelashote in his article [2].

For a given type S, binary relationship eqS, unary operator r, binary operator b, if we have a semigroup sg : semigroup S on eqS and b, then we can extract the eqv_proof from sg as eqv, and we should initially construct the eqv_proof on the red_Type (reduction type).

```
Definition reduced_eqv_proofs :
(S : Type) (eq : brel S) (r : unary_op S) (b : binary_op S)
        (eqv : eqv_proofs S eq),
        reduction_eqv_proofs S eq r ->
        eqv_proofs (red_Type S r eq) (red_eq S r eq)
```

It is worth mentioning that if the unary operator r is a reduction, then we assume that r has the properties of reduction.

Next, we extract the semigroup proof csg from sg and we construct the semigroup proof for red_Type:

```
Definition reduced_semigroup_proofs :
    (S : Type)
    (eq : brel S)
    (r : unary_op S)
    (b : binary_op S)
    (eqv : eqv_proofs S eq)
    (csg : semigroup_proofs S eq b)
    (req  : reduction_eqv_proofs S eq r)
    (rb   : reduction_bop_proofs S eq r (bop_reduce r b))
    (dec : bop_commutative_decidable (red_Type S r eq)
        (red_eq S r eq) (red_bop S b r eq (rep_idem _ _ _ req))),
     semigroup_proofs (red_Type S r eq)
        (red_eq S r eq) (red_bop S b r eq (rep_idem _ _ _ req))
```

It is worth noting here that since we need to give the new semigroup we defined about the properties of commutative, we have to know whether operator *red_bop* after reduction is commutative on *red_Type*, but for the most time this is not an easy task.

Finally, we can define a semigroup on *red_Type* following the traditional reduction.

```
Definition semigroup_reduced:
     (S : Type)
     (csg : semigroup S)
     (r : unary_op S)
     (req : reduction_eqv_proofs S (eq S csg) r)
     (rb  : reduction_bop_proofs S (eq S csg) r
        (bop_reduce r (bop S csg)))
     (dec : bop_commutative_decidable (red_Type S r (eq S csg))
        (red_eq S r (eq S csg)) (red_bop S (bop S csg) r
        (eq S csg) (rep_idem _ _ _ req))),
           semigroup (red_Type S r (eq S csg))
```

There are two obvious problems with defining the reduction semigroup in this way.
1, As mentioned above, even if we know that the original semigroup is commutative or not, it is difficult to infer whether *red_bop* is commutative under red_Type. According to our previous proof we know that if b is commutative on S, then *red_bop* is also commutative on red_Type. However, even if b is not commutative on S, *red_bop* may still be commutative on red_Type. This proof will become very complicated, and it is very dependent on the functionality of reduction and b. It is difficult to be generalized.
As mentioned in the previous section, for the most time we actually relate to our semigroup "is" commutative and not concerned with it "is whether or not" commutative. So we can add the

properties of the commutative to our defined record:

```
Record commutative_semigroup_proofs (S: Type)
(eq : brel S) (b : binary_op S) :=
{
  csg_associative   : bop_associative S eq b
; csg_congruence    : bop_congruence S eq b
; csg_commutative   : bop_commutative S eq b
}.

Record commutative_semigroup (S : Type) :=
{
   ceq    : brel S
;  cbop   : binary_op S
;  ceqv   : eqv_proofs S ceq
;  csgp   : commutative_semigroup_proofs S ceq cbop
}.
```

Unlike before, here we assume that our semigroup sg is commutative, so it is easy to construct a commutative semigroup on red_Type.

```
Definition reduced_commutative_semigroup_proofs :
      (S : Type)
    (eq : brel S)
    (r : unary_op S)
    (b : binary_op S)
    (eqv : eqv_proofs S eq)
    (csg : commutative_semigroup_proofs S eq b)
    (req : reduction_eqv_proofs S eq r)
    (rb  : reduction_bop_proofs S eq r (bop_reduce r b)),
         commutative_semigroup_proofs (red_Type S r eq)
         (red_eq S r eq) (red_bop S b r eq (rep_idem _ _ _ req))

Definition commutative_semigroup_direct_reduction :
      (S : Type)
     (csg : commutative_semigroup S)
```

```
(r : unary_op S)
(req : reduction_eqv_proofs S (ceq S csg) r)
(rb  : reduction_bop_proofs S (ceq S csg) r
   (bop_reduce r (cbop S csg))),
    commutative_semigroup (red_Type S r (ceq S csg))
```

2. After our construction, although we successfully defined semigroup on $\mathit{red\_Type}$, we lose information about reduction r (the function of r is not defined in the semigroup's record).

Therefore, the use of traditional reduction and direct definition of reduced semigroups is limited by the limitations of reduction itself and the limitations of semigroup definitions, which is not a good choice, and we need to find a better representation.

3.3 Binary Reduced Relationship and Binary Full Reduced Operator

Let have a review on the definition of the traditional reduction with its binary relationship and binary operator.
$Pr(x : S) := u_S(x) =_S x$.
$\mathit{red\_Type} := \{x : S\&Pr(x)\}$.
$\mathit{red\_eq} : \mathit{red\_Type} \to \mathit{red\_Type} \to \mathit{bool} := \lambda p_1 p_2, projT_1(p_1) =_S projT_1(p_2)$.
$\mathit{red\_bop}$: $\mathit{binary\_op}\ \mathit{red\_Type} := p1\ p2, existTPr(u_S(projT1(p1) \oplus_S projT1(p2)))(Pr\_br$ $p1\ p2)$.
We realised that actually when we are using the reduction, we wind it on our binary relationship and binary operator. So we try to discard the proof part and define new eq_r and b_r according to eq, r and b.
Definition $\mathit{brel\_reduce} : \mathit{brel}\ S := \lambda x\ y, eq\ (r\ x)\ (r\ y)$.
Definition $\mathit{bop\_reduce} : \mathit{binary\_op}\ S := \lambda x\ y, r\ (b\ x\ y)$.
$y)$.
Definition $\mathit{bop\_full\_reduce} : \mathit{binary\_op}\ S := \lambda x\ y, r(b\ (r\ x)\ (r\ y))$.
We enwrap reduction on equality to define a new binary relationship. We use the same method and define two new binary operators based on the difference in the reduction placement. However, in fact, according to the properties of the left_invariant and right_invariant of reduction

mentioned earlier, the two binary operators are the same.

Next we need to prove that by using of this representation of reduction, whether the properties of the binary operator bS we mentioned earlier is still valid on S. Here we assume that our binary operator eqS has such properties separately on S:

- congruence
- associativity
- commutativity
- selective
- idempontent
- has id $\bar{0}$
- has annihilator $\bar{1}$

And the reduction r (unary operator) has the properties of congruence, idempontent, left/right invariant.
It is easy to proof that $bop\_full\_reduce\ r\ bS$ has such properties on $brel\_reduce\ r\ eqS$:

- Lemma bop_full_reduce_congruence :
  $congruence \rightarrow congruence\ S\ (brel\_reduce\ r\ eqS)\ (bop\_full\_reduce\ r\ bS)$.
- Lemma bop_full_reduce_associative :
  $associative \rightarrow associative\ S\ (brel\_reduce\ r\ eqS)\ (bop\_full\_reduce\ r\ bS)$.
- Lemma bop_full_reduce_commutative :
  $commutative \rightarrow commutative\ S\ (brel\_reduce\ r\ eqS)\ (bop\_full\_reduce\ r\ bS)$.
- Lemma bop_full_reduce_idempotent :
  $idempotent \rightarrow idempotent\ S\ (brel\_reduce\ r\ eqS)\ (bop\_full\_reduce\ r\ bS)$.
- Lemma bop_full_reduce_selective :
  $selective \rightarrow selective\ S\ (brel\_reduce\ r\ eqS)\ (bop\_full\_reduce\ r\ bS)$.
- Lemma bop_full_reduce_exists_id :
  $uop\_preserves\_id\ S\ eqS\ bS\ u_S \rightarrow$
  $bop\_exists\_id\ S\ eqS\ bS \rightarrow$
  $bop\_exists\_id\ S\ (brel\_reduce\ r\ eqS)\ (bop\_full\_reduce\ r\ bS)$.

- Lemma bop_full_reduce_exists_ann :
  $uop\_preserves\_ann\ S\ eqS\ bS\ u_S \rightarrow$
  $bop\_exists\_ann\ S\ eqS\ bS \rightarrow$
  $bop\_exists\_ann\ S\ (brel\_reduce\ r\ eqS)\ (bop\_full\_reduce\ r\ bS).$

We find that we only use the left/right invariant properties of reduction to prove the property of the associative. Thus, maybe there is more than one way to axiomatise a reduction. In the following chapters we will introduce a new associative: the pseudo associative, and use it to prove the properties of reductions.

3.3.1 Isomorphic to Traditional Reduction

However, in fact it is not enough to justify the correctness of this new representation by establishment of these properties. We need to establish some isomorphisms to prove that this new reduction (brel_reduce and bop_full_reduce) is equivalent to the $red_Type\ red_eq$ and red_bop.

- Lemma red_ref_iso : brel_reflexive red_Type red_eq ↔ brel_reflexive S (brel_reduce r eqS).
- Lemma red_sym_iso : brel_symmetric red_Type red_eq ↔ brel_symmetric S (brel_reduce r eqS).
- Lemma red_tran_iso : brel_transitive red_Type red_eq ↔ brel_transitive S (brel_reduce r eqS).
- Lemma red_cong_iso : bop_congruence red_Type red_eq red_bop ↔ bop_congruence S (brel_reduce r eqS) (bop_full_reduce r b).
- Lemma red_bop_ass_iso : bop_associative red_Type red_eq red_bop ↔ bop_associative S eqS r b.
- Lemma red_comm_iso : bop_commutative red_Type red_eq red_bop ↔ bop_commutative S (brel_reduce r eqS) (bop_full_reduce r b).
- Lemma red_exists_id_iso : bop_exists_id red_Type red_eq red_bop ↔ bop_exists_id S (brel_reduce r eqS) (bop_full_reduce r b).

Based on these proofs, we can conclude that binary reduced relationship and binary full reduced operator have the same properties as traditional reduction. They are both a form of reduction. This form of reduction can be well expressed in the outside world in terms of the unary operator.

At the same time, we can also avoid (ease) the winding problem between the reduction and original problem set.

3.4 Reduced Semigroup

In the section of direct reduction, we defined the semigroup, and discovered that the direct reduced semigroup would lose the reduction information. Now we try to write reduction as a definition in the record of the semigroup.

```
Record rsemigroup (S : Type) :=
{
   r_eq    : brel S
;  r_rep   : unary_op S
;  r_bop   : binary_op S
;  r_eqv   : eqv_proofs S r_eq
;  r_rpv   : reduction_eqv_proofs S r_eq r_rep
;  r_sgp   : semigroup_proofs S r_eq r_bop
}.

Record commutative_rsemigroup (S : Type) :=
{
   cr_eq    : brel S
;  cr_rep   : unary_op S
;  cr_bop   : binary_op S
;  cr_eqv   : eqv_proofs S cr_eq
;  cr_rpv   : reduction_eqv_proofs S cr_eq cr_rep
;  cr_sgp   : commutative_semigroup_proofs S cr_eq cr_bop
}.
```

These two are rsemigroups that do not contain commutative properties and rsemigroups that contain commutative properties. Based on our previous experience, most of the time we will choose to directly use rsemigroups that contain commutative properties.

We also define the conversion from rsemigroup to semigroup (actually omission the reduction part and its associated proof).

```
Definition rsemigroup_to_semigroup (S : Type):
rsemigroup S -> semigroup S
:=    rsg,
{|
   eq   := r_eq S rsg
;  bop  := r_bop S rsg
;  eqv  := r_eqv S rsg
;  sgp  := r_sgp S rsg
|}.

Definition commutative_rsemigroup_to_commutative_semigroup
(S : Type):
commutative_rsemigroup S -> commutative_semigroup S
:=    crsg,
{|
   ceq   := cr_eq S crsg
;  cbop  := cr_bop S crsg
;  ceqv  := cr_eqv S crsg
;  csgp  := cr_sgp S crsg
|}.
```

3.4.1 Product RSemigroup

Chapter 4

Evaluation

Chapter 5

Summary and Conclusions

Bibliography

[1] B. A. CARR. An Algebra for Network Routing Problems. *IMA Journal of Applied Mathematics*, 7(3):273–294, June 1971.

[2] Ahnont Wongseelashote. Semirings and path spaces. *Discrete Mathematics*, 26(1):55 – 78, 1979.

[3] Seweryn Dynerowicz and Timothy G. Griffin. On the forwarding paths produced by internet routing algorithms. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pages 1–10. IEEE, 2013.

[4] Mehryar Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.

[5] Alexander JT Gurney and Timothy G. Griffin. Lexicographic products in metarouting. In *Network Protocols, 2007. ICNP 2007. IEEE International Conference on*, pages 113–122. IEEE, 2007.

[6] Alexander James Telford Gurney. Construction and verification of routing algebras.

[7] Timothy G. Griffin and Joo Lus Sobrinho. Metarouting. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 1–12. ACM, 2005.

[8] The Coq proof assistant. https://coq.inria.fr.