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
Department of Computer Science and Technology
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: zy272@cl.cam.ac.uk

June 6, 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 topic is originated from the expansion of a routing problem that was not specifically reasoned in the L11 class. We classify this class of problems as a specific type of problem, reduction. Although we discussed some of the properties of reduction in L11, however, the construction of the reduction for what we need in the final construction of path problem will conflict with the definition of reduction itself. Therefore, it shows that our definition of the reduction does not resolve our actual problem.

When we traced the source back, we found the definition of reduction from a paper by Wongseelashote in 1979. Although Wongseelashote very skilfully proposed the concept of reduction when discussing path problems, it is regrettable that this paper did not provide detailed structural proof of reduction for its properties.

Therefore, our project is comprised of the following three parts. First of all, the classical reduction is expressed, and we are trying to reason the properties of reduction itself. Next, we will try to represent/define reduction in another way, not only to facilitate the implementation, but also to decrease the limitation of the reduction definition to practical problems. That generalised reduction in good representation could help us defining the reduction that we need in the problem. After that, we will define a kind of reduction according to our requirements, and use this kind of reduction to construct numerous reduction for realistic examples. Finally, we will use these practical examples as a combination to resolve the path problem that was mentioned in L11 lecture.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Our dissertation problem originates from a concept that was mentioned in the L11 Algebraic Path Problem lecture, which has been introduced but not in detail [?]. In the L11 lecture, the concept of reduction was proposed, with examples and was utilised to solve problems that we were unable to solve before. However, in addition to the definition of reduction provided in the lecture, the property of reduction has not been thoroughly discussed. Although we have found the definition of reduction and practical examples in other papers and theses before, unfortunately, none of them have discussed the property of reduction itself or provided a detailed, sound reasoning process. This makes the detailed discussion of reduction a valuable research topic and thus becomes the topic of our project.

In our research, we will not only conduct detailed reasoning on the properties of reduction, but we will also accomplish the following three goals:

- The definition in L11 and the definition of reduction in the previous paper are all defined mathematically, which leads to the representation of reduction being difficult to implement. Therefore, we looked for an implementation-friendly representation equivalent to a traditional-reduction representation.
- The functionality of classical reduction is limited, and it cannot represent some reductions (in particular, one of the reductions we used in the construction of our path problem). We will use a more generalised method to define reduction, generalise reduction, and evaluate its reasoning.
- Based on the application of reduction to real problems, we define predicate reduction on

the basis of generalised reduction. Although predicate reduction is more concrete than generalised reduction (there are some reductions that cannot be represented by predicate reduction), it is a fairly generalised reduction and can be used to define many concrete instances of reduction.

Finally, we will use predicate reduction combined with the properties of reduction, and some other mathematical structures, to define the path problem we encountered in the L11 lecture.

Chapter 2

Background

2.1 Combinator for an Algebraic System

Initially, let us introduce the concept of CAS, in which we are going to extend the functionality with reduction. The Combinator for Algebraic systems (CAS)[**?**] is entered into $L11$. It is a language for the design of algebraic systems, in which many algebraic properties are automatically received, enabling people to combine different operators to obtain a new semiring[**?**]. We can also generalise a more complex path problem (in other words, we can abstract a more complex path problem with this new semiring, such as lexicographic products) [**?**]. CAS can easily return the properties of those combined-operation semirings: this is already defined in Coq[**?**] (mentioned in $L11$ by Dr Timothy Griffin).

2.2 Basic Definition

Before all the introductory information was mentioned, we first introduced a series of mathematical concepts that will be the focus for our project. Let us begin with our problem set S, which is a collection of elements. Here in our project, we force our problem set S to be non-trivial, which means there is at least one element inside the problem set.

2.2.1 Equality

In problem set S, the first thing we need to consider is the equality $=_S$. In the mathematical definition, we can think of equality as a particular binary relationship, which is a set of ordered pairs $=_S \in S \times S$. For equality, we have some properties that need to be discussed.

Congruence:

$$\forall a, b, c, d \in S, a =_{S_1} b \wedge c =_{S_1} d \rightarrow a =_{S_2} c = b =_{S_2} d \tag{2.1}$$

Reflexive:

$$\forall a \in S, a =_S a \tag{2.2}$$

Symmetric:

$$\forall a, b \in S, a =_S b \rightarrow b =_S a \tag{2.3}$$

Transitive:

$$\forall a, b, c \in S, a =_S b \wedge b =_S c \rightarrow a =_S c \tag{2.4}$$

2.2.2 Unary Operator

The unary operator on S will be used to express our reduction. A unary operator r on S is a function on S, $r : S \rightarrow S$. For unary operators, we are mainly interested in the following properties; these properties are also discussed later when reasoning the properties of reduction.

Congruence:

$$\forall a, b \in S, a =_S b \rightarrow r(a) =_S r(b) \tag{2.5}$$

Idempotent:

$$\forall a \in S, r(a) =_S r(r(a)) \tag{2.6}$$

Preserve Identity: Given a binary operator $\oplus : S \times S \rightarrow S$

$$\exists i \in S, \forall a \in S, a \oplus i =_S a =_S i \oplus a \rightarrow r(i) =_S i \tag{2.7}$$

Preserve Annihilator: Given a binary operator $\oplus : S \times S \rightarrow S$

$$\exists a \in S, \forall x \in S, x \oplus a =_S a =_S a \oplus x \rightarrow r(a) =_S a \tag{2.8}$$

Left Invariant: Given a binary operator $\oplus : S \times S \to S$

$$\forall a, b \in S, r(a) \oplus b =_S a \oplus b \tag{2.9}$$

Right Invariant: Given a binary operator $\oplus : S \times S \to S$

$$\forall a, b \in S, a \oplus r(b) =_S a \oplus b \tag{2.10}$$

2.2.3 Binary Operator

The major focus of this project is the binary operator on S. A binary operator $\oplus$ on S is a function $\oplus : S \times S \to S$. The following properties of the binary operator will be discussed.

Congruence:

$$\forall a, b, c, d \in S, a =_S b \wedge c =_S d \to a \oplus c =_S b \oplus d \tag{2.11}$$

Associative:

$$\forall a, b, c \in S, a \oplus (b \oplus c) =_S (a \oplus b) \oplus c \tag{2.12}$$

Commutative:

$$\forall a, b \in S, a \oplus b =_S b \oplus a \tag{2.13}$$

Not Commutative:

$$\exists a, b \in S, a \oplus b \neq_S b \oplus a \tag{2.14}$$

Selective:

$$\forall a, b \in S, a \oplus b =_S a \bigvee a \oplus b =_S b \tag{2.15}$$

Left Distributive: Given another binary operator $\times : S \times S \to S$

$$\forall a, b, c \in S, a \otimes (b \oplus c) =_S (a \otimes b) \oplus (a \otimes c) \tag{2.16}$$

Not Left Distributive: Given another binary operator $\times : S \times S \to S$

$$\exists a, b, c \in S, a \otimes (b \oplus c) \neq_S (a \otimes b) \oplus (a \otimes c) \tag{2.17}$$

Right Distributive: Given another binary operator $\times : S \times S \to S$

$$\forall a, b, c \in S, (a \oplus b) \otimes c =_S (a \otimes c) \oplus (b \otimes c) \tag{2.18}$$

Not Right Distributive: Given another binary operator $\times : S \times S \to S$

$$\exists a, b, c \in S, (a \oplus b) \otimes c \neq_S (a \otimes c) \oplus (b \otimes c) \tag{2.19}$$

2.3 Semiring and Path Problem

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

However, such an approach has its obvious shortcomings. At some point, designing a new algorithm can 'steal' the ideas of other algorithms, so people must redesign a completely new independent algorithm for each new problem (new metric). This makes it difficult to use a generic (or framework) approach to solve this type of problem. Even if the path problem has minor changes, it is difficult for people to solve new problems by slightly modifying existing algorithms.

Hence, lots of predecessors have found algebraic approaches to resolve this kind of problem. Using the knowledge of abstract algebra, people find that the routing problems (path problems) can be represented using a mathematical structure called a 'semiring' $(S, \oplus, \otimes, \bar{0}, \bar{1})$[**?**, **?**, **?**, **?**, **?**]. For example, the popular 'shortest path problem' can be represented as $(S, min, +, \infty, 0)$[**?**] and the 'maximal capacity path problem' can be represented as $(S, max, min, 0, \infty)$.

Before explaining our real problem, let us introduce a basic definition of a 'semiring'. We will then discuss the reason that semiring can be applied to solve the path problem.

2.3.1 Semiring

In abstract algebra, a semiring is a mathematical structure $(S, \oplus, \otimes, \bar{0}, \bar{1})$ where S is a set (Type) and $\oplus, \otimes$ are two binary 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 $\oplus$:

$$\forall a \in S, a \oplus \bar{1} = \bar{1} = \bar{1} \oplus a$$

Here we will include this property in our definition.

2.3.2 Matrix Semiring and Stability

For each semiring $(S, \oplus, \otimes, \bar{0}, \bar{1})$ (that represent the calculus 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 the problem.

For a directed graph $G = (V, E)$ and a weight function $w \in E \to S$, we can define the weight of a path $p = i_1, i_2, \ldots i_k$ as $w(p) = w(i_1, i_2) \otimes w(i_2, i_3) \otimes \cdots \otimes w(i_{k-1}, i_k)$, while the empty path is given the weight of $\bar{1}$ and $\bar{0}$ stands for the infinite path (not path between two nodes). 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 represented 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, $a^{(k)} = a^0 \oplus a^1 \ldots a^k$ and $a^* = a^0 \oplus a^1 \ldots a^k \oplus \ldots$.

Here we say, if there exists a q such that $a^q = a^{(q+1)}$, then a is q-stable, which means $a^q = a^*$.

If we know that S is 0-stable, then $M_n(S)$ is $n-1$-stable [?] (here we can ignore paths with loops, which is a reduction that will be introduced in a later section). 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, $A^{(k)} = A^0 \oplus A^1 \ldots A^k$ and $A^* = A^0 \oplus A^1 \ldots A^k \oplus \ldots$.

Hence, we have $\pi(i,j)$, which is the set of paths from i to j, and $\pi^k(i,j)$ will be the set of paths from i to j with exactly k arcs, and $\pi^{(k)}(i,j)$ will be the set of paths from i to j with at most k arcs. Then we have:

$$A^k(i,j) = \bigoplus_{p \in \pi^k(i,j)} w(p)$$

$$A^{(k)}(i,j) = \bigoplus_{p \in \pi^{(k)}(i,j)} w(p)$$

$$A^*(i,j) = \bigoplus_{p \in \pi^*(i,j)} w(p)$$

It is worth mentioning that $A^*(i,j)$ may not be well-defined, but if $M_n(S)$ is k stable, then for $A \in M_n(S)$,

$$A^*(i,j) = A^{(k)}(i,j)$$

2.3.3 Distributivity and Lexicographic Product

It is worth mentioning that the properties of distributivity (left and right) play an important role in the computation of the routing problem. We have 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. Considering the distributivity properties, $M_n(S)$ is left/right distributive if S has distributivity properties.

Sometimes when we are defining complex routing problems, such as the widest-shortest path problem, constructed by a shortest path problem semiring and a widest path problem semiring using the lexicographic product, it is not guaranteed that the new data structure will 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 is $(S, \oplus_S) \bar{\times} (T, \oplus_T) = (S \times T, \oplus_{\bar{\times}})$ 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 our final path problem and utilised the elementary path reduction. We also added a distinct annihilator to avoid excessive useless calculations.

These constructs make it difficult to speculate about the properties of our semiring (or new data structure); this is why our project is devoted to research regarding the relationship between reduction and semiring properties.

2.3.4 Global optimality vs. left/right local optimality

On the other hand, we are required to discuss how to ensure our approach embraces the algebras that violate distributivity.

The global optimality: $A^*(i,j) = \bigoplus_{p \in \pi(i,j)} w(p)$,

The left local optimality: 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: Dijkstra's algorithm: $R = (R \otimes A) \oplus \bar{I}$, which is $L(i,j) = \bigoplus_{q \in V} R(i,q) \otimes A(q,j)$,

Noting 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, without distributivity, those three solutions may all exist distinctively, and that brings us to the part of our project – discussing the relationship between reductions and semirings on distributive property.

2.3.5 Semiring Representation of Path Problems

For each path problem that is represented as a semiring, we can construct a corresponding matrix semiring that represents the concrete problem (for example, the edges and the distances for the shortest path problem). 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 'trivial' path problems. Complicated problems, for example, the widest shortest path problem that was constructed from the shortest path problem semiring and the widest path problem (maximal capacity path problem) semiring using the lexicographic product, cannot be solved by this 'traditional' theory approach. Sometimes, even the method can find an optimal solution; there are no guarantees that all optimal solutions will be found using the classic method.

Therefore, a non-classical algebraic path-finding method is used, so algebras that violate the distribution law can be accepted. This non-classical theory can handle problems that the simple classical theory cannot handle, such as the problems that cannot be solved by Dijkstra or Bellman-Ford. This kind of method is dedicated, initially, to finding the local optimal solution. The local optimal solution is exactly the same as the global optimal solution by some verification

or some additional restriction on the computation. For instance, the famous routing information protocol, which is based on distributed Bellman-Ford algorithms, is a non-traditional theory. Here, we provide two examples; if we follow the basis of distributed Bellman-Ford algorithm Protocol [**?**] that calculates the path from i to destination j, using the knowledge from its immediate neighbourhood and applying its own path to the neighbour available in the process, we will obtain the result using matrix multiplication.

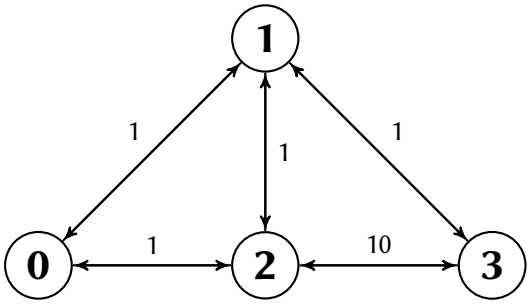

Figure 2.1: Example of RIP solution 1

We will obtain the following initial path problem adjacency matrix by using the RIP protocol.

$$\begin{bmatrix} \infty & 1 & 1 & \infty \\ 1 & \infty & 1 & 1 \\ 1 & 1 & \infty & 10 \\ \infty & 1 & 10 & \infty \end{bmatrix}$$

After using the RIP protocol for matrix calculations (multiplication), we will obtain such an adjacency matrix as a result:

$$\begin{bmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix}$$

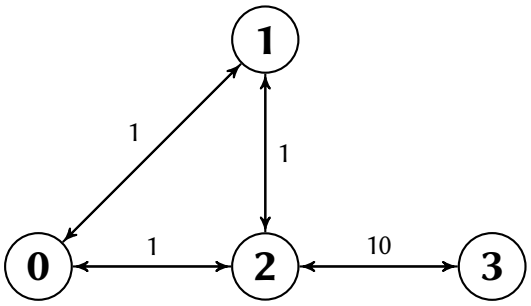

Figure 2.2: Example of RIP solution 2

Then for example 2, we will get the following initial path problem adjacency matrix.

$$\begin{bmatrix} \infty & 1 & 1 & \infty \\ 1 & \infty & 1 & \infty \\ 1 & 1 & \infty & 10 \\ \infty & \infty & 10 & \infty \end{bmatrix}$$

After using the RIP protocol for matrix calculations (multiplication), we obtained such an adjacency matrix as the result:

$$\begin{bmatrix} 0 & 1 & 1 & 11 \\ 1 & 0 & 1 & 11 \\ 1 & 1 & 0 & 10 \\ 11 & 11 & 10 & 0 \end{bmatrix}$$

2.4 Previous Problem in L11

However, RIP will also have a series of problems. For example, when a node has no edges connected to all other nodes, the RIP matrix calculation (without pre-limiting 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. This leads to the problem that not all real-world problems can be solved directly using the simple matrix semiring. For instance, we may meet the following situation where node 3 is not connected to all other nodes.

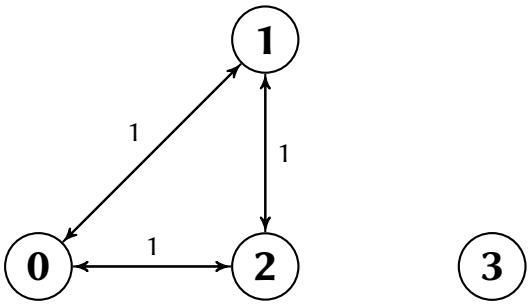

Figure 2.3: Example of RIP solution 3

We will obtain the following initial path problem matrix.

$$\begin{bmatrix} 0 & 1 & 1 & \infty \\ 1 & 0 & 1 & \infty \\ 1 & 1 & 0 & \infty \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

After using the RIP protocol for n-step matrix calculations (multiplication), a matrix such as this was obtained:

$$\begin{bmatrix} 0 & 1 & 1 & n+1 \\ 1 & 0 & 1 & n+1 \\ 1 & 1 & 0 & n+1 \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

The result shows that if we do not limit the number of steps in the calculation, even though we may get the results of the path we need in the early steps (e.g. the path from a to b), but a better solution for the entire matrix may be found after many steps (or the calculation will be infinite, which is shown in this example).

2.4.1 Possible Solution

Therefore, in the course of L11, instead of using RIP and a simple matrix semiring approach, we used another protocol called BGP [**?**]. We started from the simple $(\mathbb{N}, min, +)$ semiring that calculated the shortest distance and used the lexicographic product to construct a new semiring that contained the shortest-path metric and the set of its path.

Here we need to define some new operators/new rules for our semigroup. Assume $(S, \bullet)$ is a

semigroup. Let

$$lift(S, \bullet) \equiv (fin(2^S), \hat{\bullet}) \tag{2.20}$$

where $X\hat{\bullet}Y = \{x \bullet y | x \in X, y \in Y\}$.

Then we can use our $lift$ to construct a bi-semigroup. Assume $(S, \bullet)$ is a semigroup. Let

$$union\_lift(S, \bullet) \equiv (\mathcal{P}(S), \cup, \hat{\bullet}) \tag{2.21}$$

where $X\hat{\bullet}Y = \{x \bullet y | x \in X, y \in Y\}$, and $X, Y \in \mathcal{P}(S)$, which is the set of finite subsets of S.

Then for a given graph $G = (V, E)$, we define

$$path(E) \equiv union\_lift(E^*, .) \tag{2.22}$$

where . is the concatenation function of the sequence.

Finally, we get our 'shortest paths with paths' semiring from a given graph $G = (V, E)$:

$$spwp \equiv AddZero(\bar{0}, (\mathbb{N}, min, +) \overrightarrow{\times} path(E)) \tag{2.23}$$

2.4.2 Introduce Reduction into Our Problem

We come to a problem when we are using $spwp$ for doing calculations; because there may be loops in our $path(E)$, we need a lot of extra computation to prove that the paths that have loops are not the shortest paths. To eliminate these paths with loops, we introduced a new concept called reduction.

If $(S, \oplus, \otimes)$ is a semiring and r is a function from S to S, then r is a reduction if $\forall a, b \in S$,

$$r(a) = r(r(a))$$

$$r(a \oplus b) = r(r(a) \oplus b) = r(a \oplus r(b))$$

and

$$r(a \otimes b) = r(r(a) \otimes b) = r(a \otimes r(b))$$

And, if $(S, \oplus, \otimes)$ is a semiring and r is a reduction, then $red_r(S) = (S_r, \oplus_r, \otimes_r)$, where

$$S_r = \{s \in S | r(s) = s\}$$

$$x \oplus_r y = r(x \oplus y)$$

$$x \otimes_r y = r(x \otimes y)$$

After that we went back to our path problem where, for a given path p, we say p is elementary if there is no node inside p that is repeated. Then we can define our elementary path using the reduction

$$r(X) = \{p \in X | p \text{ is elementary } \} \tag{2.24}$$

and

$$epaths(E) = red_r(paths(E)) \tag{2.25}$$

Therefore, our path problem semiring became

$$AddZero(\bar{0}, (\mathbb{N}, min, +) \overrightarrow{\times} epath(E)) \tag{2.26}$$

However, we still encounter the problem that there exist elements in our problem set that have a path distance value but do not have edges in their path . So, we need to define the second reduction, which turns all elements that do not satisfy the condition into a single element (the $zero$ we added into our semiring).

$$\begin{aligned} r_2(inr(\infty)) &= inr(\infty) \\ r_2(inl(s, \{\})) &= inr(\infty) \\ r_2(inl(s, W)) &= inl(s, W) \end{aligned} \tag{2.27}$$

Therefore, our path problem semiring becomes

$$red_{r_2}(AddZero(\bar{0}, (\mathbb{N}, min, +) \overrightarrow{\times} epath(E))) \tag{2.28}$$

It is useful to mention that we did not discuss the properties of reduction in detail during the L11 course. We also did not know whether the constructed semiring still satisfies the semiring property after the reduction. At the same time, whether some properties of the original operators (such as commutative rules) remain, or not, after reduction, is also a mystery to us.

These are the topics that require our focus during the discussion. It is worth mentioning that the first reduction (reduce all paths that are not elementary) does not follow the property of the reduction on the min operation (it does not have left/right invariant property). Imagine we have path $A = \{a, b, c, d\}$ that is longer but does not have loop and $B = \{a, b, b\}$ that is shorter path but does have loop. Then $r(A \oplus B) = r(B) = \infty$ because B is shorter but has a loop, but $r(A \oplus r(B)) = r(A \oplus \infty) = r(A) = A$ because B has loops but A does not.

Therefore, we are not only discussing the properties of reduction, but also trying to locate another way to represent and define the reduction to solve our path problem.

It is also worth noting that the two binary operators we used to construct the elementary path semiring in the later section are not $union$ and $concat$ but min and $concat$. Such a definition will select the path that has the least number of nodes in the path, instead of finding all qualified paths as in the mathematical definition above. But both definition is meaningful and our later definition will help us resolve the path problem in advance.

Chapter 3

Mathematical Reasoning and Coq Implementation

3.1 Classical Reduction, Example and Reasoning

3.1.1 Origin of the Concept of Reduction

To better understand reduction and further analyse it in detail, we trace the source to find out if there are any other previous definitions and studies relating to reduction. The earliest definition we could find about reduction came from Ahnont Wongseelashote in 1977[**?**]. Wongseelashote also encountered the problem similar to ours when studying the path problem in the paper. Our definition of reduction in L11 follows the step of Wongseelashoth in the paper. However, Wongseelashoth only present the concept of reduction in the paper without detailed reasoning or structure proof.

Next, we found that Alexander James Telford Gurney also mentioned the concept of reduction in his Ph.D thesis[**?**]. Gurney's discussion of reduction is founded on the definition by Wongseelashote and did not focus on reduction for detailed structure proof.

3.1.2 Classical Reduction Definition

Therefore, here we refer to the definition of reduction in L11 (also the definition of Wongseelashote in the paper) as the classical reduction, and we refer to the way it was represented by Wongseelashote as the traditional representation of reduction.

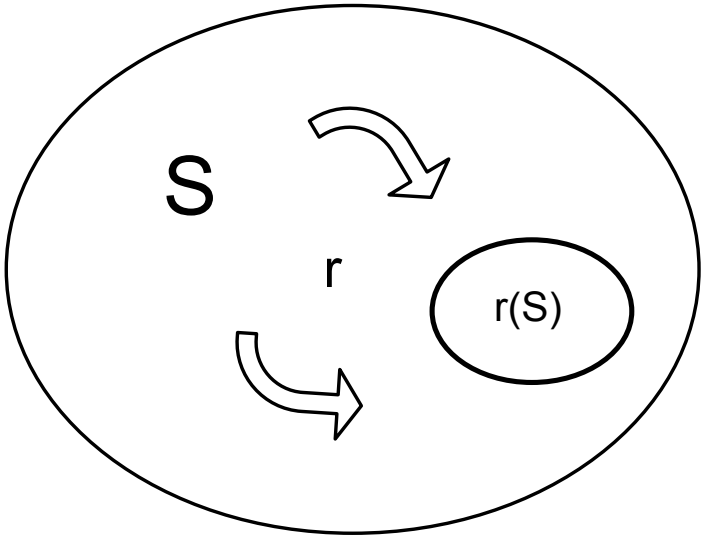

Figure 3.1: Illustration of Reduction

For a problem set S, the reduced problem set under reduction $r : S \to S$ is represented as

$$r(S) \equiv \{s \in S | r(s) =_S s\} \tag{3.1}$$

Then for the problem set S that represented as a semigroup $(S, =_S, \oplus)$, the reduced problem set will be represented as $(r(S) \equiv \{s \in S | r(s) =_S s\}, =_S, \oplus_r)$ where

$$\forall a, b \in r(S) \equiv \{s \in S | r(s) =_S s\}, a \oplus_r b \equiv r(a \oplus b) \tag{3.2}$$

It is worth mentioning that although we use reduction to reduce the problem set from S to $\{r(S) \equiv \{s \in S | r(s) =_S s\}$, our equality $=_S$ is still the original equality $=_S$. This can be seen from the figure (3.1.2). Although we only focus on a subset of the original problem set S, the

equality established in the problem set is still established in this subset. Here are the properties that a reduction r needs to have:

Congruence:

$$\forall a, b \in S, a =_S b \rightarrow r(a) =_S r(b) \tag{3.3}$$

Idempotent:

$$r(a) = r(r(a)) \tag{3.4}$$

Left Invariant:

$$r(a \oplus b) = r(r(a) \oplus b) = r(a \oplus r(b)) \tag{3.5}$$

Right Invariant:

$$r(a \otimes b) = r(r(a) \otimes b) = r(a \otimes r(b)) \tag{3.6}$$

3.1.3 Example from the L11 Lecture

To help us understand reduction and study its properties, we found another example of reduction in the L11 course.

The reduction is call $min_{\leq}$ (Martelli's semiring)[?] (also called min set [?]), which means removing all the superset. For a given graph $G = (V, E)$, A cut set $C \in E$ for nodes i and j is a set of edges such there is no path from i to j in the graph $(V, E - C)$. C is minimal if no proper subset of C is a cut set. Martelli's semiring is such that $A^{(*)}(i, j)$ is the set of all minimal cut sets for i and j. So, Martelli's semiring is represented as

$$(S, \oplus, \otimes, \bar{0}, \bar{1})$$

where

$$S = min_{\leq}(2^{2^E})$$

$$X \oplus Y = min_{\leq}(\{U \cup V | U \in X, V \in Y\})$$

$$X \otimes Y = min_{\leq}(X \cup Y)$$

$$\bar{0} = \{\{\}\}$$

$$\bar{1} = \{\}$$

Here we can easily prove that $min_{\leq}$ satisfies our previous definition of reduction. So $min_{\leq}$ is

a classical reduction.

3.1.4 Reasoning on Classical Reduction

Next, we need to make a detailed reasoning on the properties of the reduction. Because we are talking about classical reduction, we assume that our reduction r has properties of congruence (3.3), idempotent (3.4), left invariant (3.5) and right invariant (3.6).

First, we discuss the properties of equality to the reduced problem set. We assume the equality $=_S$ has the properties of reflexive (2.2), symmetric (2.3), transitive (2.4), and congruence (2.1) on the original problem set S. Then, because we have mentioned 'although we use reduction to reduce the problem set from S to $\{r(S) \equiv \{s \in S | r(s) =_S s\}$, we equality $=_S$ is still the original equality $=_S$', the equality $=_S$ still has the property of reflexive, symmetric, transitive, and congruence on the reduced problem set $\{r(S) \equiv \{s \in S | r(s) =_S s\}$.

Next, we need to discuss the properties of the binary operator under the reduced problem set. Recalling our definition, for a binary operator $\oplus : S \times S \to S$, the reduced binary operator under reduction r is defined as

$$\forall a, b \in r(S) \equiv \{s \in S | r(s) =_S s\}, a \oplus_r b \equiv r(a \oplus b)$$

So, we first assume that the original binary operator $\oplus$ has certain properties on S, and then reasoning that these properties remain after reduction ($\oplus_r$ still has the same property under $\{r(S) \equiv \{s \in S | r(s) =_S s\}$).

Commutative: we initially assume that $\oplus$ is commutative under S, which is $\forall a, b \in S, a \oplus b =_S b \oplus a$. Then we want to prove that $\forall a, b \in \{s \in S | r(s) =_S s\}, a \oplus_r b =_S b \oplus_r a$.

$$\begin{aligned} a \oplus_r b &=_S r(a \oplus b) && \text{by the definition of } \oplus_r \\ &=_S r(b \oplus a) && \text{by commutative of } \oplus \text{ and congruence of } r \\ &=_S b \oplus_r a && \text{by the definition of } \oplus_r \end{aligned} \tag{3.7}$$

So, we can conclude that commutative of $\oplus$ plus congruence of r implies commutative of $\oplus_r$.

Congruence: we initially assume that $\oplus$ is congruence under S, which is $\forall a, b, c, d \in S, a =_S b \wedge c =_S d \to a \oplus c =_S b \oplus d$. Then we want to prove that $\forall a, b, c, d \in \{s \in S | r(s) =_S s\}, a =_S$

$b \wedge c =_S d \rightarrow a \oplus_r c =_S b \oplus_r d$.

$$\begin{array}{rcll} a \oplus_r c & =_S & r(a \oplus c) & \text{by the definition of } \oplus_r \\ & =_S & r(b \oplus c) & \text{by congruence of } \oplus \text{ and congruence of } r \\ & =_S & r(b \oplus d) & \text{by congruence of } \oplus \text{ and congruence of } r \\ & =_S & b \oplus_r d & \text{by the definition of } \oplus_r \end{array} \tag{3.8}$$

So, we can conclude that congruence of $\oplus$ plus congruence of r implies congruence of $\oplus_r$.

Selective: we initially assume that $\oplus$ is selective under S, which is $\forall a, b \in S, a \oplus b =_S a \bigvee a \oplus b =_S b$. Then we want to prove that $\forall a, b \in \{s \in S | r(s) =_S s\}, a \oplus_r b =_S a \bigvee a \oplus_r b =_S b$. By using the selective property, we can split the cases of $a \oplus b$.

case 1: $a \oplus b =_S a$

$$\begin{array}{rcll} a \oplus_r b & =_S & r(a \oplus b) & \text{by the definition of } \oplus_r \\ & =_S & r(a) & \text{congruence of } r \\ & =_S & a & \text{because } r(a) = a \end{array}$$

case 2: $a \oplus b =_S b$

$$\begin{array}{rcll} a \oplus_r b & =_S & r(a \oplus b) & \text{by the definition of } \oplus_r \\ & =_S & r(b) & \text{congruence of } r \\ & =_S & b & \text{because } r(b) = b \end{array}$$

So, we can conclude that selective of $\oplus$ plus congruence of r implies selective of $\oplus_r$.

Associative: we initially assume that $\oplus$ is associative under S, which is $\forall a, b, c \in S, a \oplus (b \oplus c) =_S (a \oplus b) \oplus c$. Then we want to prove that $\forall a, b, c \in \{s \in S | r(s) =_S s\}, a \oplus_r (b \oplus_r c) =_S (a \oplus_r b) \oplus_r c$.

$$\begin{array}{rcll} a \oplus_r (b \oplus_r c) & =_S & r(a \oplus r(b \oplus c)) & \text{by the definition of } \oplus_r \\ & =_S & r(a \oplus (b \oplus c)) & \text{by right invariant property of } r \text{ on } \oplus \\ & =_S & r((a \oplus b) \oplus c) & \text{by associative of } \oplus \text{ and congruence of } r \\ & =_S & r(r(a \oplus b) \oplus c) & \text{by left invariant property of } r \text{ on } \oplus \\ & =_S & (a \oplus_r b) \oplus_r c & \text{by the definition of } \oplus_r \end{array} \tag{3.9}$$

So, we can conclude that associative of $\oplus$ plus congruence, left and right invariant of r on $\oplus$ implies associative of $\oplus_r$. Here we find that associative is not the same as the other three

properties under binary operators. The proof that a binary operator satisfies the associative, under reduction, uses the left/right invariant properties of reduction.

Left Distributive: by adding another binary operator $\otimes : S \times S \rightarrow S$, the reduced binary operator $\otimes_r$ as $\forall a, b \in \{s \in S | r(s) =_S s\}, a \otimes_r b \equiv r(a \otimes b)$, we initially assume that $\oplus$ and $\otimes$ are left distributive under S, which is $\forall a, b, c \in S, a \otimes (b \oplus c) =_S (a \otimes b) \oplus (a \otimes c)$. Then we want to prove that $\forall a, b, c \in \{s \in S | r(s) =_S s\}, a \otimes_r (b \oplus_r c) =_S (a \otimes_r b) \oplus_r (a \otimes_r c)$.

$$
\begin{array}{rll}
a \otimes_r (b \oplus_r c) & =_S \; r(a \otimes r(b \oplus c)) & \text{by the definition of } \oplus_r \text{ and } \otimes_r \\
& =_S \; r(a \otimes (b \oplus c)) & \text{by right invariant property of } r \text{ on } \otimes \\
& =_S \; r((a \otimes b) \oplus (a \otimes c)) & \text{by left distributive of } \oplus \text{ and } \otimes \\
& =_S \; r(r(a \otimes b) \oplus r(a \otimes c)) & \text{by left and right invariant of } r \text{ on } \oplus \\
& =_S \; r((a \otimes_r b) \oplus (a \otimes_r c)) & \text{by the definition of } \otimes_r \\
& =_S \; (a \otimes_r b) \oplus_r (a \otimes_r c) & \text{by the definition of } \oplus_r
\end{array}
\tag{3.10}
$$

So, we can conclude that left distributive of $\oplus$ and $\otimes$ plus congruence, left and right invariant of r on $\oplus$, right invariant of r on $\otimes$ implies left distributive of $\oplus_r$ and $\otimes_r$.

Similarly, we can conclude that right distributive of $\oplus$ and $\otimes$ plus congruence, left and right invariant of r on $\oplus$, left invariant of r on $\otimes$ implies right distributive of $\oplus_r$ and $\otimes_r$.

Thus, we conclude that a distributive of $\oplus$ and $\otimes$ plus congruence, left and right invariant of r on $\oplus$ and $\otimes$ implies a distributive of $\oplus_r$ and $\otimes_r$. It is also worth mentioning that the property of the distributive is one of the major features that will be discussed in the later section.

3.2 New Reduction Representation

It is obvious that our definition of classical reduction traditional representation is not implementation friendly. This is because the traditional reduction representation depicts the actual problem set as $r(S) = \{s \in S | r(s) =_S s\}$ which is a type with record . It is difficult to accurately represent records in most situations.

Imagine we have two different reductions r_1 and r_2 for S, then the problem set after applying reduction r_1 and r_2 will be $r_2(r_1(S)) = \{y \in \{x \in S | r_1(x) =_S x\} | r_2(y) =_S y\}$.

The nested record of reduction in our problem domain creates problems with the reasoning and calculation steps. At the same time, since our real problem set has changed from the original S

to the current $r(S) = \{s \in S | r(s) =_S s\}$ (suppose we only have one reduction r), the domain of our binary operator has changed. If we originally have a binary operator $\oplus : S \times S \to S$, then the binary operator after reduction will become the binary operator

$$\oplus_r : \{s \in S | r(s) =_S s\} \times \{s \in S | r(s) =_S s\} \to \{s \in S | r(s) = s\}$$

If we apply two/more reductions simultaneously, the entangling of reductions with the problem set and operator makes it almost impossible for us to define the combinator that exists on reduction and solve the problem at all.

Our goal is to redefine the equality and binary operators without changing the problem set S. Then the reduced problem set based on $(S, =_S, \oplus)$ by reduction r will become $(S, =_S^r, \oplus^r)$, which is still a problem set on S, but not $(\{x \in S | r(x) =_S x\}, =, \oplus_r)$. Then we will prove the isomorphism between these two representations.

3.2.1 Generalised Representation of Equality and Binary Operator

Let us look back at the figure (3.1.2). In the figure, our reduction will reduce the problem set S to a subset of S. In our traditional representation, because reduction has already brought the problem set from S to $\{x \in S | r(x) =_S x\}$, which means when we are comparing two elements from the problem set, those two elements are already inside $\{x \in S | r(x) =_S x\}$, we can also directly use the original equality $=_S$.

However, if we want to keep our problem set unchanged as S, when we are doing an equality comparison, we must guarantee that the elements we compared are already inside the reduced problem set. Thus, we define the new equality as

$$\forall a, b \in S, a =_S^r b \equiv r(a) =_S r(b) \tag{3.11}$$

The same reason applies for our binary operator $\oplus$, because the arguments we need have been reduced to $\{x \in S | r(x) =_S x\}$ for our previous defined $\oplus_r$, we only need to consider that the result of operation must also be in $\{x \in S | r(x) =_S x\}$, so we give the definition above (3.2). But now, if we want to keep our problem set unchanged as S, when we are doing binary operations, we not only need to consider that the result of the operation is in the reduced problem set, but also to ensure that the two arguments of the binary operator are in the reduced problem set.

Therefore, we define our binary operator under reduction:

$$\forall a, b \in S, a \oplus^r b \equiv r(r(a) \oplus r(b)) \tag{3.12}$$

3.2.2 Isomorphism between Transitional and Generalised Representation

Next, we need to prove that our new generalised representation is isomorphic to the previous traditional representation.

Isomorphic on Equality Properties

First, we discuss the properties on equality between the traditional representation and the generalised representation.

Congruence: we need to prove that $=_S$ is congruence on $\{x \in S | r(x) =_S x\}$ is isomorphic to $=_S^r$ is congruence on S, which means

$$\forall a, b, c, d \in \{x \in S | r(x) =_S x\}, a =_S b \wedge c =_S d \rightarrow a =_S c = b =_S d$$

$$\longleftrightarrow$$

$$\forall a, b, c, d \in S, a =_S^r b \wedge c =_S^r d \rightarrow a =_S^r c = b =_S^r d$$

Reflexive: we need to prove that $=_S$ is reflexive on $\{x \in S | r(x) =_S x\}$ is isomorphic to $=_S^r$ is reflexive on S, which means

$$\forall a \in \{x \in S | r(x) =_S x\}, a =_S a$$

$$\longleftrightarrow$$

$$\forall a \in S, a =_S^r a$$

Symmetric: we need to prove that $=_S$ is symmetric on $\{x \in S | r(x) =_S x\}$ is isomorphic to $=_S^r$ is symmetric on S, which means

$$\forall a, b \in \{x \in S | r(x) =_S x\}, a =_S b \rightarrow b =_S a$$

$$\longleftrightarrow$$

$$\forall a, b \in S, a =_S^r b \to b =_S^r a$$

Transitive: we need to prove that $=_S$ is transitive on $\{x \in S | r(x) =_S x\}$ is isomorphic to $=_S^r$ is transitive on S, which means

$$\forall a, b, c \in \{x \in S | r(x) =_S x\}, a =_S b \wedge b =_S c \to a =_S c$$

$$\longleftrightarrow$$

$$\forall a, b, c \in S, a =_S^r b \wedge b =_S^r c \to a =_S^r c$$

This proof is really trivial and only needs the property of idempotent of r, which is $\forall a \in S, r(a) = r(r(a))$.

Isomorphic on Binary Operator Properties

Next, we are going to discuss the properties on binary operator $\oplus$ (and $\otimes$ when we are talking about distributive) between the traditional representation and the generalised representation.

Commutative: we need to prove that $\oplus_r$ is commutative on $\{x \in S | r(x) =_S x\}$ is isomorphic to $\oplus^r$ is commutative on S, which means

$$\forall a, b \in \{x \in S | r(x) =_S x\}, a \oplus_r b =_S b \oplus_r a$$

$$\longleftrightarrow$$

$$\forall a, b \in S, a \oplus^r b =_S^r b \oplus^r a$$

Selective: we need to prove that $\oplus_r$ is selective on $\{x \in S | r(x) =_S x\}$ is isomorphic to $\oplus^r$ is selective on S, which means

$$\forall a, b \in \{x \in S | r(x) =_S x\}, a \oplus_r b =_S a \bigvee a \oplus_r b =_S b$$

$$\longleftrightarrow$$

$$\forall a, b \in S, a \oplus^r b =_S^r a \bigvee a \oplus^r b =_S^r b$$

Congruence: we need to prove that $\oplus_r$ is congruence on $\{x \in S | r(x) =_S x\}$ is isomorphic to $\oplus^r$ is congruence on S, which means

$$\forall a, b, c, d \in \{x \in S | r(x) =_S x\}, a =_S b \wedge c =_S d \to a \oplus_r c =_S b \oplus_r d$$

$$\longleftrightarrow$$

$$\forall a, b, c, d \in S, a =_S^r b \wedge c =_S^r d \to a \oplus^r c =_S^r b \oplus^r d$$

Associative: we need to prove that $\oplus_r$ is associative on $\{x \in S | r(x) =_S x\}$ is isomorphic to $\oplus^r$ is associative on S, which means

$$\forall a, b, c \in \{x \in S | r(x) =_S x\}, a \oplus_r (b \oplus_r c) =_S (a \oplus_r b) \oplus_r c$$

$$\longleftrightarrow$$

$$\forall a, b, c \in S, a \oplus^r (b \oplus^r c) =_S^r (a \oplus^r b) \oplus^r c$$

Left Distributive: we need to prove that $\oplus_r$ and $\otimes_r$ are left distributive on $\{x \in S | r(x) =_S x\}$ is isomorphic to $\oplus^r$ and $\oplus^r$ are left distributive on S, which means

$$\forall a, b, c \in \{x \in S | r(x) =_S x\}, a \otimes_r (b \oplus_r c) =_S (a \otimes_r b) \oplus_r (a \otimes_r c)$$

$$\longleftrightarrow$$

$$\forall a, b, c \in S, a \otimes^r (b \oplus^r c) =_S^r (a \otimes^r b) \oplus^r (a \otimes^r c)$$

Right Distributive: we need to prove that $\oplus_r$ and $\otimes_r$ are right distributive on $\{x \in S | r(x) =_S x\}$ is isomorphic to $\oplus^r$ and $\oplus^r$ are right distributive on S, which means

$$\forall a, b, c \in \{x \in S | r(x) =_S x\}, (a \oplus_r b) \otimes_r c =_S (a \otimes_r c) \oplus_r (b \otimes_r c)$$

$$\longleftrightarrow$$

$$\forall a, b, c \in S, (a \oplus^r b) \otimes^r c =_S^r (a \otimes^r c) \oplus^r (b \otimes^r c)$$

This proof will be shown in the Coq file and we only need the property of congruence and idempotent of r, which is $\forall a \in S, r(a) = r(r(a))$ and $\forall a, b \in S, a =_S b \to r(a) =_S r(b)$.

This proves that our generalised representation of reduction is isomorphic to the traditional representation of reduction, which means all the properties we have proved in the classical reduction section using traditional representation could be used directly in our generalised representation.

3.3 Generalised Reduction

To our previously defined classical reduction (either in traditional representation or generalised representation), reduction must satisfy four properties: congruence, idempotent, left invariant, and right invariant. However, the reduction we encounter in reality does not have the above four properties, especially the left/right invariants. In particular, the elementary reduction in the elementary path problem that we mentioned earlier, which will also be used in the latter section to construct our final path problem, does not satisfy the properties of the left/right invariant on the min operator.

Assuming $P_1, P_2 \in Path, P_1 \leq P_2, loop(P_1) \wedge \not{l}oop(P_2)$. Then $r(min(P_1, P_2)) = r(P_1) = \infty$. However, $r(min(r(P_1), P_2)) = r(min(\infty, P_2))r(P_2) = P_2$.

Therefore, in order to define our elementary path reduction and also to better represent other reductions, we generalise the definition reduction as: r is a reduction on S if r has the properties of congruence and idempotent and gets rid of the constraint of left/right invariant properties; we refer to this kind of reduction as generalised reduction.

Do not confuse generalised representations of reduction with generalised reduction here. Generalised representation of reduction is a new representation of reduction, compared to (and isomorphic to) the traditional representation of reduction, which is implementation friendly. A reduction that is represented by generalised representation could be classical reduction (that has left/right invariant properties) or generalised reduction (left/right invariant properties that can be removed). The generalised reduction we define here is a type of reduction that only asks for idempotent and congruence properties without forcing left/right invariants. Generalised reduction could be represented using traditional representation or generalised representation.

Because the generalised representation is implementation friendly, and we have proved the isomorphism between the generalised representation and traditional representation, we will use generalised representation to represent a reduction in the following paragraph.

3.3.1 Properties of Generalised Reduction, Pseudo Associative, and Pseudo Distributive

We have reasoned about the relationship between reduction and equality/binary operators in the previous section regarding classical reduction. Since in the last section we have already reached a grossing representation that is isomorphic to the traditional representation, and generalised reduction is just classical reduction without left/right invariant properties, by assuming that the reduction r is congruence and idempotent, we can come to the following conclusion:

If $=_S$ is congruence/reflexive/symmetric/transitive on the problem set S, then $=_S^r$ is congruence/reflexive/symmetric/transitive on the problem set S.

If $\oplus$ has the properties of commutative/selective/congruence on the problem set S, then $\oplus^r$ has the properties of commutative/selective/congruence on the problem set S.

There is another property that we may be interested in; if i/a is the identity/annihilator for $\oplus$ on S, and the reduction r has the properties of preserve id/ann (2.7,2.8), then i/a is the identity/annihilator for $\oplus^r$ on S.

The properties of associative and distributive are more troublesome. Since the proof used the property of left/right invariants, and we do not possess those properties inside generalised reduction by default, we cannot directly obtain these two properties. So, we conclude that generalised reduction with left/right properties will allow this operator(s) remaining associative and distributive properties.

However, the elementary path reduction we want to define afterwards is only a generalised reduction and does not obtain left/right invariant properties on the min operator. We still want to discuss the properties and relationship between the reduction and the operator and discuss whether min has associative properties under reduction and whether the entire semiring has distributive properties.

Therefore, we found two sufficient conditions, separately for associative properties and distributive properties, respectively, with which we can prove the properties if we can prove the sufficient condition is holding.

Pseudo Associative:

$$\forall a, b, c \in S, r(r(r(a) \oplus r(b)) \oplus r(c)) = r(r(a) \oplus r(r(b) \oplus r(c))) \tag{3.13}$$

Pseudo Left Distributive:

$$\forall a, b, c \in S, r(r(a) \otimes r(r(b) \oplus r(c))) = r(r(r(a) \otimes r(b)) \oplus r(r(a) \otimes r(c))) \tag{3.14}$$

Pseudo Right Distributive:

$$\forall a, b, c \in S, r(r(r(a) \oplus r(b)) \otimes r(c)) = r(r(r(a) \otimes r(c)) \oplus r(r(b) \otimes r(c))) \tag{3.15}$$

It is easy to prove $\oplus^r$ is associative on S, and $\oplus^r$ and $\otimes^r$ are distributive on S by using pseudo associative and pseudo distributive properties.

Associative: by definition of $=_S^r$, $\oplus^r$

$$\forall a, b, c \in S, a \oplus^r (b \oplus^r c) =_S^r (a \oplus^r b) \oplus^r c$$

$$\longleftrightarrow$$

$$r(r(r(a) \oplus r(r(r(b) \oplus r(c))))) =_S r(r(r(r(r(a) \oplus r(b))) \oplus r(c)))$$

$r(r(r(a) \oplus r(r(r(b) \oplus r(c)))))$

$$\begin{aligned}
=_S \quad & r(r(a) \oplus r(r(r(b) \oplus r(c)))) && \text{by idempotent property of } r \\
=_S \quad & r(r(a) \oplus r(r(b) \oplus r(c))) && \text{by idempotent property of } r \text{ and congruence of } \oplus \\
=_S \quad & r(r(r(a) \oplus r(b)) \oplus r(c)) && \text{by pseudo associative} \\
=_S \quad & r(r(r(r(a) \oplus r(b))) \oplus r(c)) && \text{by idempotent property of } r \text{ and congruence of } \oplus \\
=_S \quad & r(r(r(r(r(a) \oplus r(b))) \oplus r(c))) && \text{by idempotent property of } r
\end{aligned} \tag{3.16}$$

Left Distributive: by definition of $=_S^r$, $\oplus^r$ and $\otimes^r$

$$\forall a, b, c \in S, a \otimes^r (b \oplus^r c) =_S^r (a \otimes^r b) \oplus^r (a \otimes^r c)$$

$$\longleftrightarrow$$

$$r(r(r(a)) \otimes r(r(r(b) \oplus r(c))))) =_S r(r(r(r(r(a) \otimes r(b))) \oplus r(r(r(a) \otimes r(c)))))$$

$r(r(r(a)) \otimes r(r(r(b) \oplus r(c)))))$

$$
\begin{array}{lll}
=_S & r(r(a) \otimes r(r(b) \oplus r(c)))) & \text{by idempotent property of } r \text{ and congruence of } \otimes \\
=_S & r(r(r(a) \otimes r(b)) \oplus r(r(a) \otimes r(c))) & \text{by pseudo left distributive} \\
=_S & r(r(r(r(a) \otimes r(b))) \oplus r(r(r(a) \otimes r(c)))) & \text{by idempotent property of } r \text{ and congruence of } \oplus \\
=_S & r(r(r(r(r(a) \otimes r(b))) \oplus r(r(r(a) \otimes r(c)))))) & \text{by idempotent property of } r
\end{array}
\tag{3.17}
$$

Similar proof for right distributivity.

So, we can conclude that for the generalised reduction, although it may not have the property of left/right invariants, if we can directly prove the properties of pseudo associative/pseudo distributives, then we can also prove that the binary operator(s) remains associative/distributive under reduction.

3.4 Predicate Reduction

To better implement the construction of the path problem we mentioned earlier, in this section we will define a type of reduction and perform detailed reasoning on its properties. If we deeply consider our reduction, whether it is to eliminate the path with a loop, or to turn an unqualified element tuple into a specific element (additive identity/multiplicative annihilator), we could easily realise that our reduction depends on a predicate and reduce the elements that satisfy/not satisfy the condition. Therefore, we call this kind of reduction 'predicate reduction'.

We define predicate reduction based on generalised reduction. Predicate reduction is a type of generalised reduction, but is less generalised than the generalised reduction. Generalised reduction could define almost all kinds of reductions, but predicate reduction could not, and it is for this reason that it is not as generalised as the previous one. For example, when we are talking about the min set $min_{\leq} = \{x \in X | \forall y \in X, y \not\leq x\}$, which is introduced as Martelli's semiring in our previous section, we cannot define $min_{\leq}$ using predicate reduction. This is because when we are defining a predicate, the predicate only knows the information on its current element, but not the whole problem set. Predicate reduction is still a kind of generalised reduction, but not classical reduction, because it is not forcing the reduction to have left/right invariant properties. Having, or not having, those properties depends on the predicate inside reduction. For example, we will use predicate reduction to define three different reductions in a later section. Two of them (min plus with ceiling and reduction annihilator) have the properties

of left/right invariants on both operators, but one of them (elementary path) does not have them on its min operator.

Here is our definition. Initially for our problem set S, we define a predicate P, which is a function from S to $bool$ and we need to provide a specific element c (the predicate reduction will reduce all the element that satisfied the predicate to $c \in S$). Then we can define the predicate reduction as

$$\forall a \in S, r_p(a) = \begin{cases} c & P(a) \\ a & otherwise \end{cases} \tag{3.18}$$

As previously mentioned, predicate reduction is defined based on generalised reduction, which means it has all the properties that generalised reduction has. Hence, we are not interested in the properties of equality, congruence, commutative and selective, has identity and has annihilator properties of binary operator any more. We mainly talk about associative and distributive in the reasoning section (although we may use the properties of 'has identity' and 'has annihilator' during the proof process).

3.4.1 Properties of Predicate

In predicate reduction, we hope to be able to explore the properties of predicate reduction that are different from generalized reduction, but as the predicate reduction is a kind of generalised reduction, it must have the properties of the idempotent and congruence properties, which are $\forall a \in S, r(a) = r(r(a))$ and $\forall a, b \in S, a = b \rightarrow r(a) = r(b)$. So, we need to discuss the reasoning on these two properties. For the property of the idempotent, we are required to discuss the input element to split the case of whether the element satisfies the predicate or not. If the element does not satisfy the predicate, then $r(a) = a = r(r(a))$. If the element does satisfy the predicate, then $r(a) = c$ and $r(r(a)) = r(c)$, hence we need to prove/provide the property that c must satisfy the predicate

$$P(c) \tag{3.19}$$

For the property of the congruence, we can conclude from

$$\forall a, b \in S, r_p(a) = \begin{cases} c & P(a) \\ a & otherwise \end{cases} = r_p(b) = \begin{cases} c & P(b) \\ b & otherwise \end{cases}$$

that we need such a property

$$\forall a, b \in S, a = b \to P(a) = P(b) \tag{3.20}$$

which is the congruence of the predicate P.

Next, we need to reduce the binary operator according to the generalised representation we defined earlier using the predicate reduction. For our problem set S and a given binary operator $\oplus$, we defined the reduced binary operator as

$$\forall a, b \in S, a \oplus_r b = r_p(r_p(a) \oplus r_p(b)) \tag{3.21}$$

Therefore, here, we mainly discuss the properties of predicates and their relationship with binary operators, and then try to prove associative/distributive by utilising pseudo associative/distributives without forcing left/right invariants on the reduction. During this process, we discovered three very interesting properties about predicates.

Before describing these three properties, we first introduce an implicit definition. If our operator is selective, then we actually implicitly define a partial order on the problem set.

$$\forall a, b \in S, a \oplus b =_S a \leftrightarrow a \leq b$$

Next, we will formally introduce our three properties for the predicate.

Decomposition:

$$\forall a, b \in S, P(a \oplus b) \to P(a) \bigvee P(b) \tag{3.22}$$

Composition:

$$\forall a, b \in S, P(a) \bigvee P(b) \to P(a \oplus b) \tag{3.23}$$

Preserve Order:

$$\forall a, b \in S, a \leq b \wedge P(a) \to P(b) \tag{3.24}$$

From the property of decomposition, we can infer that

$$\forall a, b \in S, \not{P}(a) \bigwedge \not{P}(b) \to \not{P}(a \oplus b) \tag{3.25}$$

which means the binary operator will not create elements that satisfy the predicate condition

from the elements that do not satisfy the condition.

From the property of decomposition, we can infer that

$$\forall a, b \in S, \not{P}(a \oplus b) \rightarrow \not{P}(a) \bigwedge \not{P}(b) \tag{3.26}$$

which means the result of the binary operation will decide the predicate condition of its arguments.

Generally speaking, because our additive component of the semiring is mostly selective, most of our additive components have decomposition properties; we are mainly concerned about whether our additive component preserve the order. At the same time, our multiplicative component always plays the role of an accumulator, which makes the composition property important for multiplicative components.

Let us discuss the above two properties at the level of reduction. If a predicate is decomposed/-composed on an operator, it means that the reduction formed by this predicate has the following properties on this operator.

$$\forall a, b \in S, r_p(a \oplus b) = c \rightarrow r_p(a) = c \bigvee r_p(b) = c$$

which is

$$\forall a, b \in S, r_p(a) = a \bigwedge r_p(b) = b \rightarrow r_p(a \oplus b) = a \oplus b$$

or

$$\forall a, b \in S, r_p(a) = c \bigvee r_p(b) = c \rightarrow r_p(a \oplus b) = c$$

which is

$$\forall a, b \in S, r_p(a \oplus b) = a \oplus b \rightarrow r_p(a) = a \bigwedge r_p(b) = b$$

Another thing to note is that in the construction of predicate reduction we will manually add a constant value c and reduce all elements that satisfy the predicate to c. Although, generally speaking, the constant c could be an arbitrary element in the problem set S. However, in most cases it is the identity/annihilator to the additive/multiplicative component.

Then we can start our reasoning on the predicate reduction. We firstly assume that the predicate P has property of P_true (3.19) and $P\_congruence$ (3.20).

We were surprised to find the following two properties:

1, If the special constant c is the identity to the operator, then the property of the operator, preserving order on the predicate, is isomorphic to the left/right invariant properties.

If

$$\forall a \in S, c \oplus a =_S a =_S a \oplus c$$

Then

$$\forall a, b \in S, a \leq b \wedge P(a) \to P(b)$$

$$\longleftrightarrow$$

$$\forall a, b \in S, r_p(r_p(a) \oplus b) =_S r_p(a \oplus b) \bigwedge r_p(a \oplus r_p(b)) =_S r_p(a \oplus b)$$

2, If the operator is compositional, then we can get left/right invariant properties from the operator under the predicate reduction.

$$\forall a, b \in S, P(a) \bigvee P(b) \to P(a \oplus b)$$

$$\longrightarrow$$

$$\forall a, b \in S, r_p(r_p(a) \oplus b) =_S r_p(a \oplus b) \bigwedge r_p(a \oplus r_p(b)) =_S r_p(a \oplus b)$$

Detailed proof will be presented in the later Coq file, and no more details will be given here.

In conjunction with the above proof and the universal properties of the previously mentioned additive/multiplicative, we can draw the following conclusions.

1) For a semiring, if the additive component has the properties of a preserving order, and the constant c is the identity for the additive component, then the predicate reduction under additive component is classical.

2) For a semiring, if the multiplicative component has the properties of composition, then the predicate reduction under multiplicative component is classical.

Predicate and Associative

According to our previous proof, we can draw two methods that make operators keep the associative property under the predicate reduction.

1) The reduced constant c is the identity to the operator, and the operator has the property of

the preserving order.

2) The operator holds the property of composition.

The reason is that both can infer that reduction is classical on the operator, and we have shown that classical reduction has the property of retaining associative.

However, it is surprising that we have found a third sufficient condition that can prove associative property.

3) the reduced constant c is the identity/annihilator to the operator, and the operator has the property of decomposition.

Because the proof needs too many case analyses, we only provide the Coq proof version instead of formal mathematical proof.

Therefore, we summarise the sufficient conditions for a semiring that can demonstrate that the operator is associative under reduction, which further proves that left/right invariant properties are only sufficient conditions for associative rather than sufficient and necessary conditions.

- c is identity for $\oplus$ and $\oplus$ has the property of preserving order (3.24).
- $\otimes$ has the property of composition (3.23).
- c is identity/annihilator for $\oplus$ and $\oplus$ has the property of decomposition (3.22).

Predicate and Distributive

For a semiring, like associative proof, if the additive component has the properties of preserving order, the constant c is the identity for the additive component, and the multiplicative component has the properties of composition, then the predicate reduction is classical under the two operators; it has been proven that the sufficient condition to the proposition is that the semiring is left/right distributive under the reduction.

Also surprising is that we found that if the two operators in a semiring are both decomposition, then the semiring is also distributive under predicate reduction.

We take the left distributive property as an example, $\forall a, b, c \in S$, we get the left-hand side of the equality $r_p(r_p(a) \otimes r_p(r_p(r_p(b) \oplus r_p(c))))$

If any of a, b, c satisfies predicate P, the result is definite, and the distributive property remains after reduction.

The most critical problem is that a, b, and c alone do not satisfy predicate P, but $a \otimes b$ or $a \otimes c$ does. However, because we have assumed that both operators are decompositions, this completely eliminates the occurrence of such things. This means that if $\not P(a)$, $\not P(b)$, $\not P(c)$, then $\not P(a \otimes b)$ and $\not P(a \otimes c)$, which leads to the property of the left distributive.

Similar proof is constructed for the right distributive. This means that apart from the classical reduction, we also find another way to prove distributive, such as associative.

Summary for Predicate Reduction

We now make a small summary of our definition of predicate reduction. In the process of our later constructing/reasoning predicate reduction instance, we need to consider the following things:

- reasoning the P_true property (3.19)
- reasoning the $P\_congruence$ property (3.20)
- checking whether c is the identity/annihilator for $\oplus^r/\otimes^r$ or not
- reasoning the properties of decomposition (3.22), composition (3.23) and preserving order (3.24) for both operator
- reasoning the associative property if the reduction is not classical
- reasoning the distributive property if the reduction is not classical

3.5 Path Problem Construction

After the general reasoning of predicate reduction, we used predicate reduction to define three concrete reductions. These three reductions will be used in the construction of our final path problem semiring in this section, and we will carry out detailed reasoning on these three reductions.

3.5.1 Min Plus with Ceiling Reduction

Here we define a reduction based on predicate reduction, which we will use in constructing the final semiring; it is working on $(\mathbb{N}, min, +)$, the first part of the path problem. As previously mentioned, our path problem is represented by a tuple (d, P), which represents the path problem with a path. The distance in our path is calculated by $(\mathbb{N}, min, +)$ semiring. However, sometimes the distance is too long, which may lead to some crucial problems during the calculation (for example, an infinite loop in the calculation). In addition, we will also consider limiting the length of the path in real-world problems and we will not consider long paths (paths with a long distance). So, we can define a reduction that will reduce all long path distances in one element, which we call 'ceiling', and the reduction semiring problem is called min plus with a ceiling problem.

$$\forall n \in \mathbb{N}, r_c(n) = \begin{cases} ceiling & n \geq ceiling \\ n & otherwise \end{cases} \tag{3.27}$$

and the predicate here is that

$$P : \forall n \in \mathbb{N}, n \geq ceiling$$

Next, based on the steps we mentioned in the predicate reduction, we perform one-by-one reasoning on the ceiling predicate that we have defined.

P_true: it is obvious that $P(ceiling)$ because $ceiling \leq ceiling$.

P_congruence: it is also obvious because we are living in a world of natural number, which means

$$\forall a, b \in \mathbb{N}, a = b \wedge a \leq ceiling \rightarrow b \leq ceiling$$

Ceiling is the identity for min^r operator, because in our reduced problem set, the scope of natural number element will be limited into $[0, ceiling]$ (all the number that is greater than $ceiling$ will be reduced to $ceiling$ due to our reduction definition). This means that $ceiling$ is the biggest element in our problem set, which means

$$\forall a \in r(\mathbb{N}), min^r(a, ceiling) = min^r(ceiling, a) = a$$

Ceiling is the annihilator for $+^r$ operator, because we have a lemma in natural number that

$$\forall a \in \mathbb{N}, \forall b \in \mathbb{N}, a \leq a + b$$

Then it is obvious that $\forall n \in \mathbb{N}, ceiling \leq a + ceiling$. Hence, we get

$$\forall n \in \mathbb{N}, ceiling +^r n = ceiling = n +^r ceiling$$

due to the definition of our reduction.

Decomposition: the additive component, min operator, has the property of selective, which means it is decomposition. We can also prove the property by

$$\forall a, b \in \mathbb{N}, ceiling \leq min(a, b) \rightarrow ceiling \leq a \bigwedge ceilling \leq b \rightarrow ceiling \leq a \bigvee ceilling \leq b$$

Composition: the multiplicative component, $+$ operator, has the property of composition and we can prove it directly.

$$\forall a, b \in \mathbb{N}, ceiling \leq a \bigvee ceiling \leq b \rightarrow ceiling \leq a + b$$

Preserving order: the additive component, min operator, has the property of preserving order and we can prove it directly by the transitive property of $\leq$

$$\forall a, b \in \mathbb{N}, a \leq b \wedge ceiling \leq a \rightarrow ceiling \leq b$$

So, we can conclude that our definition of reduction is classical on both operators. At the same time, we consider the following properties:

- $\forall a \in \mathbb{N}, min(0, n) = 0 = min(n, 0)$ (0 is the annihilator for min)
- $\forall a \in \mathbb{N}, 0 + n = n = n + 0$ (0 is the identity for $+$)
- $0 \neq ceiling \rightarrow ceiling \not\leq 0$ (reduction preserve annihilator/identity for $min/+$)
- min has the properties of selective, congruence, associative, commutative
- $+$ has the properties of congruence, associative, commutative
- min and $+$ are distributive on $\mathbb{N}$

We can conclude that $(\mathbb{N}, min^r, +^r, ceiling, 0)$ is the reduced semiring that

- additive component min^r has the properties of selective, congruence, associative, commutative, has identity of $ceiling$, has annihilator of 0

- multiplicative component $+^r$ has the properties of congruence, associative, commutative, has identity of 0, and has annihilator of $ceiling$
- min^r and $+^r$ are distributive on $\mathbb{N}$

3.5.2 Elementary Path

The second predicate reduction we need to define is the reduction that reduces all the paths that have loops to a single elementary called the elementary path.

First of all, we need to consider the problem that constructing the representation of our path creates. For a graph $G = (V, E)$, our problem set should be $\mathbb{P}(E^*)$. In fact, we can also use V to represent the path, and this representation is more convenient for us to calculate. For a path p that passes through several points $n_1, n_2...n_k \in V$ in turn, we use an ordered list to represent the path $p = [n_1, n_2...n_k]$. Since the node here is just a reference for a name, to facilitate our proof and calculation, we define our path $Path \equiv List(\mathbb{N})$, using the natural number to indicate nodes.

Next, we need to define our two binary operators, min and $concat$. Before defining the operator min, we need to define a helper binary order, the dictionary order: for all $a, b \in Path$,

$$\begin{aligned} d(nil, b) &= nil \\ d(a, nil) &= nil \\ d(a :: as, b :: bs) &= \begin{cases} a & a \leq b \\ b & otherwise \end{cases} \end{aligned}$$

Then we can use the dictionary order d to define our min operator as the additive component:

$$\forall a, b \in Path, min(a, b) = \begin{cases} a & length(a) \leq length(b) \\ d(a, b) & length(a) = length(b) \\ b & otherwise \end{cases} \tag{3.28}$$

Then the $concat$ operator, as the multiplicative component, is just the simple concatenation function of a list.

Finally, we should define the predicate: the path has loops in its representation. It is really easy, because, if a path has loop, it must pass at least one node more than once. Therefore, as long as

there is a duplicate in the list represented for a path, it means that there is a loop in this path.

$$\forall p \in Path, P(p) \equiv dup(p)$$

However, the next problem we have is that there is no identity/annihilator for our additive/-multiplicative component in our problem set, since there is no path with an infinite length. Analogous to the previous min plus problem, there is no infinite element in the problem set either. In the previous question, we manually added a ceiling to the reduction and made it an identity/annihilator for our additive/multiplicative components. However, in this case, we cannot do the same thing. Hence, we need to manually add an identity/annihilator for our additive/multiplicative component using disjoint union, and our problem set becomes

$$c + List(\mathbb{N})$$

So, we can define our predicate and reduction as $\forall p \in (c + Path)$,

$$P(inl(c))$$

$$P(inr(p)) \equiv dup(p)$$

$$r(inl(c)) = inl(c)$$

$$r(inr(a)) = \begin{cases} inl(c) & P(a) \\ inr(a) & otherwise \end{cases}$$

Next, based on the steps we mentioned in the predicate reduction, we perform one-by-one reasoning on the loop predicate that we have defined.

P_true: it is obvious from our definition that $P(inl(c))$.

P_congruence: here we need to define a helper lemma that $\forall p_1, p_2 \in Path, p_1 = p_2 \rightarrow dup(p_1) = dup(p_2)$ and it is trivially true, due to our definition of a list and duplicate. Thus, it is obvious that $\forall a, b \in (c + Path), a = b \rightarrow P(a) = P(b)$.

Considering our problem set has changed from $Path$ to $(c + Path)$, we need to redefine our

min and $concat$ operator. $\forall a, b \in Path$,

$$\begin{aligned} min_d(inl(c), a) &= a \\ min_d(a, inl(c)) &= a \\ min_d(inr(p_1), inr(p_2)) &= min(p_1, p_2) \end{aligned} \tag{3.29}$$

$$\begin{aligned} concat_d(inl(c), \_) &= inl(c) \\ concat_d(\_, inl(c)) &= inl(c) \\ concat_d(inr(p_1), inr(p_2)) &= concat(p_1, p_2) \end{aligned} \tag{3.30}$$

According to our definition, it is obvious that $inl(c)$ is the identity/annihilator for min_d/$concat_d$.

Then because we defined our additive operator ourselves, we need to reason the property of this operator. We have proved that min_d has the properties of selective, congruence, associative and commutative. Detailed proof can be found in the Coq file.

Decomposition: because min_d is selective, then it is obvious that min_d is decomposition under the predicate.

Composition: we can prove that $concat_d$ is composition. Considering $p_1, p_2 \in Path$, if either of them has loop inside the path, then the result of the concatenation of them must have a loop. Detailed proof can be found in the Coq file.

Preserving Order: although we defined a selective additive component, which implicitly defines a partial order on $Path$, the order has no relationship with either path, whether it has a loop or not. Therefore, the additive component min_d does not have the property of a preserving order and we can easily find an example. Imagine $a = inr([1, 2, 2])$ and $b = inr([1, 2, 3, 4])$, it is obvious that $a \leq_p b$ because $min_d(a, b) = a$. However, by our definition we have $P(a)$ but $/P(b)$. Similarly, we can prove that the reduction on the min_d is not classical because $r(min_d(r(a), b)) = b$ but $r(min_d(a, b)) = inl(c)$.

Another point worthy of our attention is that $concat_d$ does not have commutative property because $concat$ does not. For example, $a = inr([1])$ and $b = inr([2])$, then $concat_d(a, b) = inr([1, 2])$ but $concat_d(b, a) = inr([2, 1])$.

At the same time, $inr([])$ is the annihilator/identity for our additive/multiplicative component according to our definition.

Thus, we can conclude that our definition of reduction is classical regarding multiplicative com-

ponents but is not classical regarding additive component. At the same time, we consider the following properties.

- $\forall a \in (c+Path), min_d(inr([]), n) = inr([]) = min_d(n, inr([]))$ ($inr([])$ is the annihilator for min_d)
- $\forall a \in (c + Path), concat_d(inr([]), n) = n = concat_d(n, inr([]))$ ($inr([])$ is the identity for $concat_d$)
- $\not{dup}([]) \to \not{P}(inr[])$ (reduction preserve annihilator/identity for $min/+$)
- min_d has the properties of selective, congruence, associative, commutative
- $concat_d$ has the properties of congruence and associative
- min_d and $concat_d$ are distributive on $(c + Path)$

However, min_d^r and $concat_d^r$ are not distributive on $(c+Path)$ and we can provide a counterexample. Let $a, b, c \in (c + Path), a = inr([1]), b = inr([1, 2]), c = inr([2, 3, 4])$, then

$$concat_d^r(a, min_d^r(b, c)) = concat_d^r(a, b) = r(inr([1, 1, 2])) = inl(c)$$

$$min_d^r(concat_d^r(a, b), concat_d^r(a, c)) = min_d^r(inl(c), inr([1, 2, 3, 4])) = inr([1, 2, 3, 4])$$

Hence, the left distributive property does not hold, and we can find similar proof for the right distributive.

Thus, we can conclude that $((c + Path), min_d^r, concat_d^r, inl(c), inr([]))$ is the reduced semiring that

- additive component min_d^r has the properties of selective, congruence, associative, commutative, has identity of $inl(c)$, has annihilator of $inr([])$.
- multiplicative component $concat_d^r$ has the properties of congruence and associative, but not commutative, has identity of $inr([])$, has annihilator of $inl(c)$.
- min_d^r and $concat_d^r$ are not distributive on $(c + Path)$.

3.5.3 Lexicographic Product and Direct Product

We have already defined the two components needed for our path problem, $(\mathbb{N}, min^r, +^r, ceiling, 0)$ and $((c + Path), min_d^r, concat_d^r, inl(c), inr([]))$. Next, we need to combine these two semirings together to form the semiring of our final path problem. The method we used to combine two semirings was derived from the lexicographic product mentioned in L11. Path Problem =

$$(\mathbb{N}, min^r, +^r, ceiling, 0) \bar{\times} ((c + Path), min_d^r, concat_d^r, inl(c), inr([]))$$

=

$$(\mathbb{N} \times (c + Path), min^r \bar{\times} min_d^r, +^r \times concat_d^r, (ceiling, inl(c)), (0, inr([])))$$

Following the definition of the lexicographic product of two semirings from L11, according to the definition in L11, for two semirings $(S_1, \oplus_1, \otimes_1)$,$(S_2, \oplus_2, \otimes_2)$, the lexicographic product of them is $(S_1, \oplus_1, \otimes_1) \bar{\times} (S_2, \oplus_2, \otimes_2) = (S_1 \times S_2, \oplus_{\bar{\times}}, \otimes_\times)$.

Here is the definition of the lexicographic product and direct product for two operators.

Suppose $(S_1, \oplus_1)$ is a commutative and selective semigroup and $(S_2, \oplus_2)$ is a semigroup, then the lexicographic product of two semigroups $(S_1, \oplus_1) \bar{\times} (S_2, \oplus_2) = (S_1 \times S_2, \oplus_{\bar{\times}})$ where

$$(s_1, t_1) \oplus_{\bar{\times}} (s_2, t_2) = \begin{cases} (s_1 \oplus_1 s_2, t_1 \oplus_2 t_2) & s_1 = s_1 \oplus_1 s_2 = s_2 \\ (s_1 \oplus_1 s_2, t_1) & s_1 = s_1 \oplus_1 s_2 \neq s_2 \\ (s_1 \oplus_1 s_2, t_2) & s_1 \neq s_1 \oplus_1 s_2 = s_2 \end{cases}$$

And suppose $(S_1, \otimes_1)$ and $(S_2, \otimes_2)$ are semigroups, then the direct product of two semigroups $(S_1, \otimes_1) \times (S_2, \otimes_2) = (S_1 \times S_2, \otimes_\times)$ where

$$(s_1, t_1) \otimes_\times (s_2, t_2) = (s_1 \otimes_1 s_2, t_1 \otimes_2 t_2)$$

Next, we need to discuss the properties of the direct product and lexicographic product. Because these properties are not the focus of our project, we will not provide detailed proof of those properties here. Some theorems are directly referenced from existing properties from the L11 lecture.

First, we discuss the properties of the direct product. Obviously, the direct product has the ability to retain binary operator properties. For example, if the original two binary operators

both have associative property, then the direct product of those binary operators also has the property of associative. Similarly, if $i_1, i_2/a_1, a_2$ are the identity/annihilator to the binary operator $\oplus_1$, $\oplus_2$, then $(i_1, i_2)/(a_1, a_2)$ is the identity/annihilator to the direct product of those two operator $\oplus_1 \times \oplus_2$.

So, we can conclude that our multiplicative component $+^r \times concat_d^r$ of the path problem has the properties of congruence and associative. However, because the $concat_d^r$ operator in the elementary path semiring is not commutative, the multiplicative component of the path problem is not commutative either. We can conclude that $(0, inr([]))$ is the identity of $+^r \times concat_d^r$ and $(ceiling, inl(c))$ is the annihilator of $+^r \times concat_d^r$.

Then we discuss the properties of the lexicographic product. Because the reasoning part of these properties is not the main content discussed by our project, we directly borrowed the properties stated in the L11 class.

We assume $(S_1, \oplus_1)$ is associative, commutative and selective, so we get:

$$\oplus_{\bar{\times}} \text{ is associative} \longleftrightarrow \oplus_2 \text{ is associative.}$$

$$\oplus_{\bar{\times}} \text{ has identity} \longleftrightarrow \oplus_1 \text{ and } \oplus_2 \text{ both has identity.}$$

$$\oplus_{\bar{\times}} \text{ has annihilator} \longleftrightarrow \oplus_1 \text{ and } \oplus_2 \text{ both has annihilator.}$$

$$\oplus_{\bar{\times}} \text{ is commutative} \longleftrightarrow \oplus_2 \text{ is commutative.}$$

$$\oplus_{\bar{\times}} \text{ is selective} \longleftrightarrow \oplus_2 \text{ is selective.}$$

$$\oplus_{\bar{\times}} \text{ is congruence} \longleftrightarrow \oplus_1 \text{ and } \oplus_2 \text{ are both congruence.}$$

$$\otimes_1 \text{ and } \otimes_2 \text{ are not distributive on } S_2 \longrightarrow \oplus_{\bar{\times}} \text{ and } \otimes_{\times} \text{ are not distributive on } S_1 \times S_2.$$

So, we can conclude that our additive component $min^r \bar{\times} min_d^r$ of the path problem has the properties of congruence, associative, commutative, and selective. We can conclude that $(ceiling, inl(c))$ is the identity of $min^r \bar{\times} min_d^r$ and $(0, inr([]))$ is the annihilator of $+^r \times concat_d^r$. $min^r \bar{\times} min_d^r$ and $+^r \times concat_d^r$ are not distributive on $\mathbb{N} \times (c + Path)$.

3.5.4 Reduce Annihilator

Finally, we define our third predicate reduction. The reduction will reduce all the pairs of elements $(x, y) \in X \times Y$ to (a, b), where a is the annihilator for the multiplicative component of X and b is the annihilator for the multiplicative component of Y, if $x = a \vee y = b$.

According to our previous reasoning, $(ceiling, inl(c))$ is the identity/annihilator for the additive/multiplicative component of the path problem semiring. Therefore, following the definition, the predicate reduction will reduce all pairs of $(ceiling, \_)$ and $(\_, inl(c))$ to $(ceiling, inl(c))$.

Hence, we define the predicate as follows

$$\forall (n, p) \in \mathbb{N} \times (c + Path), P((n, p)) \equiv n = ceiling \vee p = inl(c) \tag{3.31}$$

And the predicate reduction of reduce annihilator is defined as

$$r_a(a) = \begin{cases} (ceiling, inl(c)) & P(a) \\ a & otherwise \end{cases} \tag{3.32}$$

Then we are going to reason the properties of the reduce annihilator reduction.

P_true: It is obvious that $P((ceiling, inl(c)))$.

P_congruence: because we have both congruence property for natural number and list, then it is obvious that P has the property of congruence.

What we defined in the reduction, $(ceiling, inl(c))$ is the identity/annihilator for the additive/-multiplicative component of the path problem semiring.

Then we need to discuss the property between the predicate and our binary operators.

Decomposition: since we have proved (using the properties of lexicographic product), $min^r \bar{\times} min^r_d$ is selective, then it is obvious that $min^r \bar{\times} min^r_d$ has the property of decomposition.

Composition: it is easy to prove that $+^r \times concat^r_d$ is composition under the predicate. Suppose we have $(n_1, p_1), (n_2, p_2) \in \mathbb{N} \times (c + Path)$, if $P(((n_1, p_1)))$, which means $n_1 = ceiling \vee p_1 = inl(c)$, in both cases, the results of the binary operation will yield a tuple that contains at least one of $ceiling$ or $inl(c)$, leading to the result, satisfying the predicate and proof for $P(((n_2, p_2)))$.

Preserving Order: we can prove that $min^r \bar{\times} min^r_d$ does not have the property of preserving order and provide the counter example. Suppose we have $a = (0, inl(c)), b = (1, inr([])) \in \mathbb{N} \times (c + Path)$, it is obvious that $amin^r \bar{\times} min^r_d b = a$ by the definition of lexicographic product, and $P(a)$. However, it is also obvious that $\not P(b)$ and that the additive component does not have left/right invariant properties by the example we provided here.

Thus, we can conclude that our definition of reduction is classical on a multiplicative component but is not classical on an additive component.

We also know that, by the assumption that $ceiling$ is not 0, $\not P((0, inr([])))$. Hence, the reduction preserves the annihilator/identity for additive/multiplicative components.

Because our additive component is decomposition, and $(ceiling, inl(c))$ is the identity for $min^r \bar{\times} min^r_d$, and $min^r \bar{\times} min^r_d$ has the property of associative, $min^r \bar{\times} min^r_d$ remains the property of associative after reduction.

By the assumption that $ceiling$ is not 0, because our multiplicative component is not commutative, then $+^r \times concat^r_d$ is still not commutative after reduction. We can easily find the counterexample that, let $a = (0, inr([1])), b = (0, inr([2])) \in \mathbb{N} \times (c + Path)$,

$$(+^r \times concat^r_d)_a(a, b) = (0, inr([1, 2])) \neq (0, inr([2, 1])) = (+^r \times concat^r_d)_a(b, a)$$

with the assumption that $ceiling \neq 0$.

By the assumption that $ceiling$ is not 0, and $min^r \bar{\times} min^r_d$ and $+^r \times concat^r_d$ are not distributive on $\mathbb{N} \times (c + Path)$, we obtain the result that $min^r \bar{\times} min^r_d$ and $+^r \times concat^r_d$ are not distributive on $\mathbb{N} \times (c + Path)$ after reduction. Similarly to commutative properties, we can easily find a counterexample, by assigning $a = (0, inr([1])), b = (0, inr([1])), c = b = (0, inr([2, 3])) \in \mathbb{N} \times (c + Path)$, the left hand side of the equation will become $(ceiling, inl(c))$ after calculation, but the right hand side of equation will become $(0, inr([1, 2, 3]))$ after calculation.

Thus, we can conclude that

$$(\mathbb{N} \times (c + Path), (min^r \bar{\times} min^r_d)_a, (+^r \times concat^r_d)_a, (ceiling, inl(c)), (0, inr([])))$$

is the reduced semiring, and the semiring for our final path problem that

- additive component $(min^r \bar{\times} min^r_d)_a$ has the properties of selective, congruence, associative, commutative, has identity of $(ceiling, inl(c))$, has annihilator of $(0, inr([]))$.
- multiplicative component $(+^r \times concat^r_d)_a$ has the properties of congruence and associative, but not commutative, has identity of $(0, inr([]))$, has annihilator of $(ceiling, inl(c))$.
- $(min^r \bar{\times} min^r_d)_a$ and $(+^r \times concat^r_d)_a$ are not distributive on $\mathbb{N} \times (c + Path)$.

Chapter 4

Evaluation, Summary, and Conclusions

4.1 Evaluation to the Result

First of all, we first assess the results of our definition of reduction.

Before we defined the generalised representation, the original representation of reduction was not very implementation friendly, as we have seen. The generalised representation of our definition represents the reduced problem set using reduction, a unary operator, the equality and binary operators, and is implementation friendly. Furthermore, our proof in Coq can be extracted to other languages for implementation reasons, such as Occam and Haskell, which are functional programming languages. Because of the difficulty, or even impossibility, of representing records in these languages, our generalised representation will be very important. The proof of isomorphism between the two representation also helps us to better express and use reduction. It is obvious that the properties of reduction under generalised representation should be more easily proved.

At the same time, although the original classical reduction gives us two more specific properties, it also limits the definition of reduction, so many types of reduction cannot be classified into it. After our generalised reduction, most of this operation type can be defined and classified into it. We can also define more concrete reductions based on generalised reduction, such as our defined predicate reduction. The reasoning of predicate reduction is also very detailed; latecomers can use its properties directly.

We have defined generalised representation and generalised reduction, with the advantage that

it is very easy to implement a reduction combination. For a reduction, by our definition, it does not matter whether the problem set has been reduced before. All that matters is the nature of the existing equality/binary operator. So, we can easily combine a lot of reduction together, just as we finally defined the path problem.

4.2 Summary to the Project

Before this study, no one had detailed reasoning on reduction. Although many people mentioned the concept of reduction and discussed its properties, constructive reasoning is still necessary. This is also the purpose of our project.

Of course, during this process, we have made many detours and there have been many useless or wrong directions regarding proof. For example, we wanted to add the property of reduction in the definition of the semigroup, but we could not solve the problem of reduction combination. We also wanted to define the reduce annihilator as a separate type of reduction, but later found that the reduce annihilator was actually an example of predicate reduction. There were many other issues like this, and as a topic for research, there is no certain direction. We need to explore all the steps ourselves. This is the most important thing we learned while carrying out the project. We keep all the files we have defined in the Coq folder. Those who are interested can go to the Coq file to view them.

4.3 Conclusion to the Thesis

Our discussion of reduction is almost complete. During the process of reasoning the predicate reduction we confirmed, at our earliest guess, was that classical reduction is only a sufficient condition for associative and distributive, although we did not find a very meaningful example. We can provide an example that, similar to the elementary path problem, we defined another operator max, which was opposite to min. It is obvious that max is decomposition, and not a preserving order. However, we can still prove the associative, since it is decomposition and the reduced element is its identity. We can also prove the distributive under those two operators. However, this example does not have practical significance, and we have not yet found an example that is meaningful and satisfies the conditions.

Our proof covers almost all the properties that will be used. Late adopters only need to provide

basic proof of properties to use our proof and definitions to construct semirings for compounding requirements.

Appendix A

Some Code of Proof

A.1 Coq Implementation

All the aforementioned proofs can be found in my Coq file. In the Coq file, we directly admitted some lemmas in places where it has nothing to do with our problem (for example, the properties of lexicographic product) or the proof of the lemma is trivial true (for example, the proof of $a :: as \neq as$).

In this section, we will list some key definitions and key proofs to help the reading understand the content mentioned above.

A.1.1 Basic Definition

First we need to define some basic mathematical concepts in Coq. We define the equality as a binary relationship, the unary operator and the binary operator.

```
Definition brel (S : Type)              := S → S → bool.
Definition unary_op (S : Type)          := S → S.
Definition binary_op (S : Type)         := S → S → S.
```

Listing 1: Basic Definition

Then we need to define the product type: direct product and lexicographic product.

```
Definition brel_product {S T : Type} (eqS : brel S) (eqT : brel T) : brel (S * T)
:= λ x y, match x, y with
          | (s1, t1), (s2, t2) => andb (eqS s1 s2) (eqT t1 t2)
          end.
```

Listing 2: Binary Relationship of Product Type

```
Definition bop_product {S T : Type} (bS : binary_op S) (bT : binary_op T): binary_op (S * T)
:= λ x y, match x, y with
          | (s1, t1), (s2, t2) => (bS s1 s2, bT t1 t2)
          end.
```

Listing 3: Direct Product of Binary Operator

```
Definition brel_complement : ∀ {S : Type}, brel S -> brel S
:= λ {S} r x y,  if (r x y) then false else true.
Definition brel_conjunction : ∀ {S : Type}, brel S -> brel S -> brel S
:= λ {S} r1 r2 x y,  (r1 x y) && (r2 x y).
Definition brel_llte : ∀ {S : Type}, brel S → binary_op S → brel S
:= λ {S} eq b x y, eq x (b x y).
Definition brel_llt : ∀ {S : Type}, brel S → binary_op S → brel S
:= λ {S} eq b, brel_conjunction (brel_llte eq b) (brel_complement eq).
Definition bop_llex : ∀ {S T : Type}, brel S → binary_op S → binary_op T → binary_op (S * T)
:= λ {S T} eq b1 b2 x y,
   match x, y with
    | (a, b), (c, d) =>
       (b1 a c,
        if eq a c
        then (b2 b d)
        else if brel_llt eq b1 a c
             then b
             else d)
   end.
```

Listing 4: Lexicographic Product

Finally we need to define the method to add a constant by disjoint union.

```
Definition cas_constant : Type          := string.
Definition brel_add_constant : ∀ {S : Type}, brel S → cas_constant → brel (cas_constant + S)
:= λ  {S} rS c x y,
   match x, y with
   | (inl _), (inl _) => true (* all constants equal! *)
   | (inl _), (inr _) => false
   | (inr _), (inl _) => false
   | (inr a), (inr b) => rS a b
   end.
```

Listing 5: Definition of Adding a Constant

```
Definition bop_add_ann : ∀ {S : Type}, binary_op S → cas_constant → binary_op (cas_constant + S)
:= λ {S} bS c x y,
   match x, y with
   | (inl _), _       => inl c
   |       _, (inl _) => inl c
   | (inr a), (inr b) => inr _ (bS a b)
   end.
```

Listing 6: Construct Binary Operator by Adding Annihilator

```
Definition bop_add_id : ∀ {S : Type}, binary_op S → cas_constant → binary_op (cas_constant + S)
:= λ  {S} bS c x y,
   match x, y with
   | (inl _), (inl _) => inl c
   | (inl _), (inr _) => y
   | (inr _), (inl _) => x
   | (inr a), (inr b) => inr _ (bS a b)
   end.
```

Listing 7: Construct Binary Operator by Adding Identity

A.1.2 Basic Properties

Next, we define the properties that we mentioned in our previous section in Coq.

Properties for Binary Relationship

We define the properties of reflexive, symmetric, transitive and congruence for binary relationship (equality).

```
Definition brel_reflexive (S : Type) (r : brel S) :=
  ∀ s : S, r s s = true.
Definition brel_symmetric (S : Type) (r : brel S) :=
    ∀ s t : S, (r s t = true) → (r t s = true).
Definition brel_transitive (S : Type) (r : brel S) :=
  ∀ s t u: S, (r s t = true) → (r t u = true) → (r s u = true).
Definition brel_congruence (S : Type) (eq : brel S) (r : brel S) :=
   ∀ s t u v : S, eq s u = true → eq t v = true → r s t = r u v.
```

Listing 8: Binary Relationship Property

Properties for Unary Operator

We define the properties of preserving identity/annihilator, congruence, idempotent, left/right invariant for the unary operator (reduction).

```
Definition uop_preserves_id (S : Type) (eq : brel S) (b : binary_op S) (r : unary_op S) :=
  ∀ (s : S), bop_is_id S eq b s -> eq (r s) s = true.
Definition uop_preserves_ann (S : Type) (eq : brel S) (b : binary_op S) (r : unary_op S) :=
  ∀ (s : S), bop_is_ann S eq b s -> eq (r s) s = true.
Definition uop_congruence (S : Type) (eq : brel S) (r : unary_op S) :=
  ∀ (s1 s2 : S), eq s1 s2 = true -> eq (r s1) (r s2) = true.
Definition uop_idempotent (S : Type) (eq : brel S) (r : unary_op S) :=
  ∀ s : S, eq (r (r s)) (r s) = true.
Definition bop_left_uop_invariant (S : Type) (eq : brel S) (b : binary_op S) (r : unary_op S) :=
  ∀ s1 s2 : S, eq (b (r s1) s2) (b s1 s2)  = true.
Definition bop_right_uop_invariant (S : Type) (eq : brel S) (b : binary_op S) (r : unary_op S) :=
  ∀ s1 s2 : S, eq (b s1 (r s2)) (b s1 s2)  = true.
```

Listing 9: Unary Operator Property

Properties for Binary Operator

We define the properties of commutative/not-commutative, congruence, selective, associative, has identity and has annihilator for our binary operator.

```
Definition bop_commutative (S : Type) (r : brel S) (b : binary_op S)
    := ∀ s t : S, r (b s t) (b t s) = true.
Definition bop_not_commutative(S : Type) (r : brel S) (b : binary_op S)
  := { z : S * S & match z with (s, t) => r (b s t) (b t s) = false end }.
Definition bop_selective (S : Type) (eq : brel S) (b : binary_op S)
    := ∀ s t : S, (eq (b s t) s = true) + (eq (b s t) t = true).
Definition bop_is_id (S : Type) (r : brel S) (b : binary_op S) (i : S)
    := ∀ s : S, (r (b i s) s = true) * (r (b s i) s = true).
Definition bop_is_ann (S : Type) (r : brel S) (b : binary_op S) (a : S)
    :=  ∀ s : S, (r (b a s) a = true) * (r (b s a) a = true).
Definition bop_congruence (S : Type) (r : brel S) (b : binary_op S) :=
   ∀ (s1 s2 t1 t2 : S), r s1 t1 = true -> r s2 t2 = true -> r (b s1 s2) (b t1 t2) = true.
Definition bop_associative (S : Type) (r : brel S) (b : binary_op S)
  := ∀ s t u : S, r (b (b s t) u) (b s (b t u)) = true.
```

Listing 10: Basic Binary Operator Property

We also define the distributive properties for two binary operators.

```
Definition bop_left_distributive (S : Type) (r : brel S)
(add : binary_op S) (mul : binary_op S)
   := ∀ s t u : S, r (mul s (add t u)) (add (mul s t) (mul s u)) = true.
Definition bop_right_distributive (S : Type) (r : brel S)
(add : binary_op S) (mul : binary_op S)
   := ∀ s t u : S, r (mul (add t u) s) (add (mul t s) (mul u s)) = true.
Definition bop_not_left_distributive (S : Type) (r : brel S)
(add : binary_op S) (mul : binary_op S)
   := {a : S * S * S & match a with (s,t,u)
   => r (mul s (add t u)) (add (mul s t) (mul s u)) = false end}.
Definition bop_not_right_distributive (S : Type) (r : brel S)
(add : binary_op S) (mul : binary_op S)
   := {a : S * S * S & match a with (s,t,u)
   => r (mul (add t u) s) (add (mul t s) (mul u s)) = false end}.
```

Listing 11: Binary Operator Distributive

Finally we define those pseudo properties (we mentioned in the previous section) to help us do reasoning on our reduction.

```
Definition bop_pseudo_associative (S : Type) (eq : brel S) (r : unary_op S) (b : binary_op S)
  := ∀ s t u : S, eq (r (b (r (b (r s) (r t))) (r u))) (r (b (r s) (r (b (r t) (r u))))) = true.
Definition bop_pseudo_left_distributive (S : Type) (eq : brel S) (r : unary_op S)
(add mul : binary_op S)
  := ∀ a b c : S,
  eq (r (mul (r a) (r (add (r b) (r c)))))
     (r (add (r (mul (r a) (r b))) (r (mul (r a) (r c))))) = true.
Definition bop_pseudo_right_distributive (S : Type) (eq : brel S) (r : unary_op S)
(add mul : binary_op S)
  := ∀ a b c : S,
  eq (r (mul (r (add (r b) (r c))) (r a)))
     (r (add (r (mul (r b) (r a))) (r (mul (r c) (r a))))) = true.
```

Listing 12: Binary Operator Pseudo Properties

A.1.3 Construction of Semiring

Then we need to provide a definition of semring together with its proofs. Since finally we need to construct two different versions of semiring: distributive semiring with commutative multiplicative component and non-distributive semiring with non-commutative multiplicative component semiring, which resulting of three different versions of semigroup: commutative selective semigroup, commutative semigroup and non-commutative semigroup, we need to provide separate definition for them.

First, let us define the record of proofs of the those properties for binary relationship

```
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
}.
```

Listing 13: Proof of Properties for Binary Relationship

Inside the record, we provide a witness which guarantee that the semiring we defined is not a trivial semiring (at least has one element, which is usually the identity or annihilator for an operator in the simiring).

Then we can define our two separate versions of semigroup. We will provide the proof of properties for the semigroup at first.

```
Record commutative_selective_semigroup_proofs (S: Type) (eq : brel S) (b : binary_op S) :=
{
  cssg_associative   : bop_associative S eq b
; cssg_congruence    : bop_congruence S eq b
; cssg_commutative   : bop_commutative S eq b
; cssg_selective     : bop_selective S eq b
}.
```

Listing 14: Proof of Properties for Commutative Selective Semigroup

```
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
}.
```

Listing 15: Proof of Properties for Commutative 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
}.
```

Listing 16: Proof of Properties for (None Commutative) Semigroup

Then we can construct our semigroup based on these different properties records.

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

Listing 17: Commutative Selective Semigroup

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

Listing 18: Commutative Semigroup

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

Listing 19: Semigroup

Next, we will define the proof of properties for a simiring. We also provides two different versions of record, based on the distributive properties.

```
Record dioid_proofs (S: Type) (eq : brel S) (add mul : binary_op S) (zero : S) (one : S) :=
{
  dioid_left_distributive  : bop_left_distributive S eq add mul
; dioid_right_distributive : bop_right_distributive S eq add mul
; dioid_zero_is_add_id     : bop_is_id S eq add zero
; dioid_one_is_mul_id      : bop_is_id S eq mul one
; dioid_zero_is_mul_ann    : bop_is_ann S eq mul zero
; dioid_one_is_add_ann     : bop_is_ann S eq add one
}.
```

Listing 20: Proof of Properties for Semiring

```
Record bioid_proof (S: Type) (eq : brel S) (add mul : binary_op S) (zero : S) (one : S) :=
{
  bioid_left_distributive_decidable  : bop_left_distributive_decidable S eq add mul
; bioid_right_distributive_decidable : bop_right_distributive_decidable S eq add mul
; bioid_zero_is_add_id     : bop_is_id S eq add zero
; bioid_one_is_mul_id      : bop_is_id S eq mul one
; bioid_zero_is_mul_ann    : bop_is_ann S eq mul zero
; bioid_one_is_add_ann     : bop_is_ann S eq add one
}.
```

Listing 21: Proof of Properties for Bioid (None Distributive Semiring)

Finally, we can construct our semiring and non-distributive semiring using our previous definition.

```
Record commutative_selective_dioid (S : Type) := {
    csdioid_eq          : brel S
  ; csdioid_add         : binary_op S
  ; csdioid_mul         : binary_op S
  ; csdioid_zero        : S
  ; csdioid_one         : S
  ; csdiode_eqv         : eqv_proofs S csdioid_eq
  ; csdiode_add_pfs     : commutative_selective_semigroup_proofs S csdioid_eq csdioid_add
  ; csdiode_mul_pfs     : commutative_semigroup_proofs S csdioid_eq csdioid_mul
  ; csdioid_pfs         : dioid_proofs S csdioid_eq csdioid_add csdioid_mul csdioid_zero csdioid_one
}.
```

Listing 22: Commutative Selective Semiring

```
Record selective_bioid (S : Type) := {
  sbioid_eq          : brel S
  ; sbioid_add         : binary_op S
  ; sbioid_mul         : binary_op S
  ; sbioid_zero        : S
  ; sbioid_one         : S
  ; sbioid_eqv         : eqv_proofs S sbioid_eq
  ; sbioid_add_pfs     : commutative_selective_semigroup_proofs S sbioid_eq sbioid_add
  ; sbioid_mul_pfs     : semigroup_proofs S sbioid_eq sbioid_mul
  ; sbioid_pfs         : bioid_proof S sbioid_eq sbioid_add sbioid_mul sbioid_zero sbioid_one
}.
```

Listing 23: Selective None Distributive Semiring

A.1.4 Traditional Representation of Reduction and Classical Reduction

After that, we initially define the traditional representation of reduction.

```
Variable S : Type.
Variable b : binary_op S.
Variable r : unary_op S.
Variable eqS    : brel S.

Definition Pr (x : S) := eqS (r x) x = true.
Definition red_Type   := { x : S & Pr x}.
Definition red_eq : brel red_Type := λ p1 p2, eqS ((projT1 p1)) ((projT1 p2)).
Definition red_bop : binary_op red_Type :=
  λ p1 p2,  existT Pr (bop_reduce r b (projT1 p1) (projT1 p2)) (Pr_br p1 p2).
```

Listing 24: Traditional Representation of Reduction

Then, we can do some reasoning on that representation, to prove that the equality remains all the properties that holding on the original problem set.

```
Variable refS   : brel_reflexive S eqS.
Variable symS   : brel_symmetric S eqS.
Variable transS : brel_transitive S eqS.
Variable eqS_cong : brel_congruence S eqS eqS.

Lemma red_ref : brel_reflexive red_Type red_eq.
Lemma red_sym : brel_symmetric red_Type red_eq.
Lemma red_cong : brel_congruence red_Type red_eq red_eq.
Lemma red_trans : brel_transitive red_Type red_eq.
```

Listing 25: Proof of Properties on Equality

Finally we prove the properties of binary operators under traditional representation of reduction.

```
Variable b_cong : bop_congruence S eqS b.
Variable b_ass  : bop_associative S eqS b.
Variable b_sel  : bop_selective S eqS b.
Variable b_com  : bop_commutative S eqS b.

Variable r_cong  : uop_congruence S eqS r.
Variable r_idem  : uop_idempotent S eqS r.
Variable r_left  : bop_left_uop_invariant S eqS (bop_reduce r b) r.
Variable r_right : bop_right_uop_invariant S eqS (bop_reduce r b) r.

Lemma red_bop_cong : bop_congruence red_Type red_eq red_bop.
Lemma red_bop_ass : bop_associative red_Type red_eq red_bop.
Lemma red_bop_comm : bop_commutative red_Type red_eq red_bop.
Lemma red_bop_sel : bop_selective red_Type red_eq red_bop.
Lemma red_bop_id : uop_preserves_id S eqS b r ->
                   bop_exists_id S eqS b ->
                   bop_exists_id red_Type red_eq red_bop.
Lemma red_bop_ann : uop_preserves_ann S eqS b r ->
                    bop_exists_ann S eqS b ->
                    bop_exists_ann red_Type red_eq red_bop.
```

Listing 26: Proof of Properties on Binary Operator

And we can prove the properties of distributive of two binary operators.

```
Definition T : Type := red_Type S r eq.
Definition eqT : brel T := red_eq S r eq.
Variable add mul : binary_op S.
Definition addT : binary_op T := red_bop S add r eq r_idem.
Definition mulT : binary_op T := red_bop S mul r eq r_idem.

Lemma addT_mulT_left_distributive :
  bop_left_uop_invariant S eq (bop_reduce r add) r ->
  bop_right_uop_invariant S eq (bop_reduce r add) r ->
  bop_right_uop_invariant S eq (bop_reduce r mul) r ->
  bop_left_distributive S eq add mul ->
  bop_left_distributive T eqT addT mulT.
Lemma addT_mulT_right_distributive :
  bop_left_uop_invariant S eq (bop_reduce r add) r ->
  bop_right_uop_invariant S eq (bop_reduce r add) r ->
  bop_left_uop_invariant S eq (bop_reduce r mul) r ->
  bop_right_distributive S eq add mul ->
  bop_right_distributive T eqT addT mulT.
```

Listing 27: Distributive on Binary Operators

Here we only list the lemma as a skeleton for what we need to prove, detailed proof could be found in Coq file.

A.1.5 Generalized Representation of Reduction and Generalized Reduction

Then we can define our generalized representation of reduction.

```
Definition brel_reduce {S : Type} (r : unary_op S) (eq : brel S) : brel S
:= λ x y,  eq (r x) (r y).
Definition bop_reduce {S : Type} (r : unary_op S) (b : binary_op S) : binary_op S
:= λ x y,  r (b x y).
Definition bop_full_reduce {S : Type} (r : unary_op S) (b : binary_op S) : binary_op S
:= λ x y,  r(b (r x) (r y)).
```

Listing 28: Generalized Representation of Reduction

In the case that the reduction is classical, bop_reduce will equal to bop_full_reduce, otherwise we give a way that the reduction can only have the properties of congruence and idempotent with out left/right invariant.

Then we have a bunch of isomorphism to prove that those two representation are exactly representing the same problem set.

```
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_brel_cong_iso : brel_congruence red_Type red_eq red_eq <->
                          brel_congruence S (brel_reduce r eqS) (brel_reduce r eqS).
```

Listing 29: Equality Isomorphism Between Two Representation

```
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_pseudo_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_not_comm_iso :  bop_not_commutative red_Type red_eq red_bop <->
                          bop_not_commutative S (brel_reduce r eqS) (bop_full_reduce r b).
Lemma red_sel_iso :  bop_selective red_Type red_eq red_bop <->
                     bop_selective S (brel_reduce r eqS) (bop_full_reduce r b).
```

Listing 30: Binary Operation Isomorphism Between Two Representation

Then we define the isomorphism between the distributive properties.

```
Lemma red_bop_left_dist_iso : bop_left_distributive T eqT addT mulT <->
                              bop_left_distributive S (brel_reduce r eq) (bop_full_reduce r add)
                                                                        (bop_full_reduce r mul).
Lemma red_bop_right_dist_iso : bop_right_distributive T eqT addT mulT <->
                               bop_right_distributive S (brel_reduce r eq) (bop_full_reduce r add)
                                                                          (bop_full_reduce r mul).
```

Listing 31: Distributive Isomorphism Between Two Representation

Finally, we prove the isomorphism between the pseudo properties we defined, and the properties we need to prove on the binary operator.

```
  Variable refS   : brel_reflexive S eqS.
  Variable symS   : brel_symmetric S eqS.
  Variable transS : brel_transitive S eqS.
  Variable r_cong  : uop_congruence S eqS r.
  Variable r_idem  : uop_idempotent S eqS r.
  Variable b_cong : bop_congruence S eqS b.

  Lemma bop_full_reduce_pseudo_associative_implies_associative :
         bop_pseudo_associative S eqS r bS ->
         bop_associative S (brel_reduce r eqS) (bop_full_reduce r bS).
  Lemma bop_full_reduce_associative_implies_pseudo_associative :
         bop_associative S (brel_reduce r eqS) (bop_full_reduce r bS) ->
         bop_pseudo_associative S eqS r bS.

  Lemma bop_reduce_pseudo_left_distributivity_iso :
         bop_left_distributive S (brel_reduce r eq) (bop_full_reduce r add) (bop_full_reduce r mul).
         <->
         bop_pseudo_left_distributive S eq r add mul.
  Lemma bop_reduce_pseudo_right_distributivity_iso :
         bop_right_distributive S (brel_reduce r eq) (bop_full_reduce r add) (bop_full_reduce r mul).
         <->
         bop_pseudo_right_distributive S eq r add mul.
```

Listing 32: Isomorphism Between Pseudo and Real Properties

A.1.6 Predicate Reduction

Initially we provide a definition of the predicate, and define several properties of our predicate we've mentioned in the previous section.

```
Definition pred (S : Type)              := S → bool.
Definition pred_true (S : Type) (P : pred S) (s : S)
  := P s = true.
Definition pred_congruence (S : Type) (eq : brel S) (P : pred S)
  := ∀ (a b : S), eq a b = true -> P a = P b.
Definition pred_bop_decompose (S : Type) (P : pred S) (bS : binary_op S)
  := ∀ (a b : S), P (bS a b) = true -> (P a = true) + (P b = true).
Definition pred_bop_compose (S : Type) (P : pred S) (bS : binary_op S)
  := ∀ (a b : S), (P a = true) + (P b = true) -> P (bS a b) = true.
Definition pred_preserve_order (S : Type) (P : pred S) (eqS : brel S) (bS : binary_op S)
  := ∀ (a b : S), eqS (bS a b) a = true -> P a = true -> P b = true.
```

Listing 33: Predicate Definition and Properties

Then we need to define the reduction over the predicate

```
Definition uop_predicate_reduce : ∀ {S : Type}, S -> (S -> bool) -> unary_op S
  := λ  {S} s1 P s2,  if P s2 then s1 else s2.
Definition bop_fpr {S : Type} (s : S ) (P : S -> bool) (bS : binary_op S) :=
  bop_full_reduce (uop_predicate_reduce s P) bS.
```

Listing 34: Predicate Reduction

As we mentioned in the early section, predicate reduction have all the properties that belongs to generalized reduction. Therefore we only need to concern about the properties that is specific to the predicate reduction.

```
Lemma bop_pseudo_associative_fpr_decompositional_id :
  ∀ (c : S) (bS : binary_op S),
    pred_true S P c ->
    pred_congruence S eq P ->
    bop_congruence S eq bS ->
    bop_associative S eq bS ->
    pred_bop_decompose S P bS ->
    bop_is_id S eq bS c ->
    bop_pseudo_associative S eq (uop_predicate_reduce c P) bS.
Lemma bop_pseudo_associative_fpr_decompositional_ann :
  ∀ (s : S) (bS : binary_op S),
    pred_true S P s ->
    pred_congruence S eq P ->
    bop_associative S eq bS ->
    pred_bop_decompose S P bS ->
    bop_is_ann S eq bS s ->
    bop_pseudo_associative S eq (uop_predicate_reduce s P) bS.
```

Listing 35: Associative For Predicate

```
Lemma bop_fpr_left_distributive :
  ∀ (s : S) (add mul : binary_op S),
     pred_true S P s ->
     pred_congruence S eq P ->
     pred_bop_decompose S P add ->
     pred_bop_decompose S P mul ->
     bop_congruence S eq add ->
     bop_congruence S eq mul ->
     bop_is_id S eq add s ->
     bop_is_ann S eq mul s ->
     bop_left_distributive S eq add mul ->
     bop_left_distributive S (brel_reduce (uop_predicate_reduce s P) eq)
                             (bop_fpr s P add) (bop_fpr s P mul).
Lemma bop_fpr_right_distributive :
  ∀ (s : S) (add mul : binary_op S),
     pred_true S P s ->
     pred_congruence S eq P ->
     pred_bop_decompose S P add ->
     pred_bop_decompose S P mul ->
     bop_congruence S eq add ->
     bop_congruence S eq mul ->
     bop_is_id S eq add s ->
     bop_is_ann S eq mul s ->
     bop_right_distributive S eq add mul ->
     bop_right_distributive S (brel_reduce (uop_predicate_reduce s P) eq)
                              (bop_fpr s P add) (bop_fpr s P mul).
```

Listing 36: Distributive For Predicate

For properties of preserving order and composition, we have proved that reduction is a classical if it has one of such properties, and we can directly prove associative/distributive properties from the classical reduction.

```
Lemma bop_left_uop_invariant_predicate_reduce :
  ∀ (s : S) (bS : binary_op S),
    pred_true S P s ->
    pred_bop_compose S P bS ->
    bop_left_uop_invariant S eq (bop_reduce (uop_predicate_reduce s P) bS)
                                (uop_predicate_reduce s P).
Lemma bop_right_uop_invariant_predicate_reduce :
  ∀ (s : S) (bS : binary_op S),
    pred_true S P s ->
    pred_bop_compose S P bS ->
    bop_right_uop_invariant S eq (bop_reduce (uop_predicate_reduce s P) bS)
                                 (uop_predicate_reduce s P).
```

Listing 37: Composition implies Classical

```
Lemma bop_left_uop_invariant_predicate_reduce_v2 :
  ∀ (s : S) (bS : binary_op S),
    pred_true S P s ->
    pred_congruence S eq P ->
    bop_selective S eq bS ->
    bop_is_id S eq bS s ->
    pred_preserve_order S P eq bS ->
    bop_left_uop_invariant S eq (bop_reduce (uop_predicate_reduce s P) bS)
                                (uop_predicate_reduce s P).
Lemma bop_right_uop_invariant_predicate_reduce_v2 :
  ∀ (s : S) (bS : binary_op S),
    pred_true S P s ->
    pred_congruence S eq P ->
    bop_selective S eq bS ->
    bop_commutative S eq bS ->
    bop_is_id S eq bS s ->
    pred_preserve_order S P eq bS ->
    bop_right_uop_invariant S eq (bop_reduce (uop_predicate_reduce s P) bS)
                                 (uop_predicate_reduce s P).
```

Listing 38: Preserving Order implies Classical

Next we need to use our predicate reduction to construct three different reductions, and to define our path problem semiring.

A.1.7 Min Plus With Ceiling

Initially we provide the definition of the equality, two binary operators and the predicate for our problem.

```
Definition brel_eq_nat  : brel nat  := Arith.EqNat.beq_nat.
Definition min := Nat.min.
Definition plus := Nat.add.
Definition P (ceiling : nat): nat -> bool := λ n, ceiling <=? n.
```

Listing 39: Min Plus With Ceiling Definition

By using the lemmas in Coq library for min and $+$, we can easily get the properties of those two operators, such as commutative, associative, etc.

Then we need to discuss the properties of the predicate, and the properties between the predicate and the two binary operators.

```
Lemma P_congruence (ceiling : nat): pred_congruence nat brel_eq_nat (P ceiling).
Lemma P_true (ceiling : nat): pred_true nat (P ceiling) ceiling.
Lemma P_min_decompose (ceiling : nat): pred_bop_decompose nat (P ceiling) min.
Lemma P_min_preserve_order (ceiling : nat): pred_preserve_order nat (P ceiling) brel_eq_nat min.
Lemma P_plus_compose (ceiling : nat): pred_bop_compose nat (P ceiling) plus.
```

Listing 40: Properties for the Predicate

Then we can define our reduction based on the predicate.

```
Definition uop_nat (ceiling : nat) : unary_op nat := uop_predicate_reduce ceiling (P ceiling).
Definition bop_nat_min (ceiling : nat) : binary_op nat := bop_fpr ceiling (P ceiling) min.
Definition bop_nat_plus (ceiling : nat) : binary_op nat := bop_fpr ceiling (P ceiling) plus.
```

Listing 41: Min Plus With Ceiling Reduction

The rest of the properties can be proved using the properties defined in the predicate reduction and generalized reduction section.

Finally we can construct our semiring.

```
Definition eqv_proofs_eq_nat (ceiling : nat) : eqv_proofs nat
(brel_reduce (uop_nat ceiling) brel_eq_nat)
:= {|
     eqv_reflexive   := brel_reduce_nat_reflexive ceiling
   ; eqv_transitive  := brel_reduce_nat_transitive ceiling
   ; eqv_symmetric   := brel_reduce_nat_symmetric ceiling
   ; eqv_congruence  := brel_reduce_nat_congruence ceiling
   ; eqv_witness     := 0
|}.
```

Listing 42: Equality Proof

```
Definition min_proofs (ceiling : nat) :
commutative_selective_semigroup_proofs nat
(brel_reduce (uop_nat ceiling) brel_eq_nat) (bop_nat_min ceiling)
:= {|
  cssg_associative   := bop_nat_min_associative ceiling
; cssg_congruence    := bop_nat_min_congruence ceiling
; cssg_commutative   := bop_nat_min_commutative ceiling
; cssg_selective     := bop_nat_min_selective ceiling
|}.
```

Listing 43: Proof for Min Operator

```
Definition plus_proofs (ceiling : nat) :
commutative_semigroup_proofs nat
(brel_reduce (uop_nat ceiling) brel_eq_nat) (bop_nat_plus ceiling)
:= {|
  csg_associative   := bop_nat_plus_associative ceiling
; csg_congruence    := bop_nat_plus_congruence ceiling
; csg_commutative    := bop_nat_plus_commutative ceiling
|}.
```

Listing 44: Proof for Plus Operator

```
Definition min_plus_dioid_proofs (ceiling : nat) :
dioid_proofs nat (brel_reduce (uop_nat ceiling) brel_eq_nat)
(bop_nat_min ceiling) (bop_nat_plus ceiling) ceiling  0
:= {|
  dioid_left_distributive  := bop_left_distributive_ceiling_min_plus ceiling
; dioid_right_distributive := bop_right_distributive_ceiling_min_plus ceiling
; dioid_zero_is_add_id     := bop_is_id_ceiling_min_ceiling ceiling
; dioid_one_is_mul_id      := bop_is_id_ceiling_plus_zero ceiling
; dioid_one_is_add_ann     := bop_is_ann_ceiling_min_zero ceiling
; dioid_zero_is_mul_ann    := bop_is_ann_ceiling_plus_ceiling ceiling
|}.
```

Listing 45: Proof for Semiring Property

```
Definition min_plus_dioid (ceiling : nat) : commutative_selective_dioid nat
:= {|
    csdioid_eq         := brel_reduce (uop_nat ceiling) brel_eq_nat
  ; csdioid_add        := bop_nat_min ceiling
  ; csdioid_mul        := bop_nat_plus ceiling
  ; csdioid_zero       := ceiling
  ; csdioid_one        := 0
  ; csdiode_eqv        := eqv_proofs_eq_nat ceiling
  ; csdiode_add_pfs    := min_proofs ceiling
  ; csdiode_mul_pfs    := plus_proofs ceiling
  ; csdioid_pfs        := min_plus_dioid_proofs ceiling
|}.
```

Listing 46: Min Plus With Ceiling Semiring

A.1.8 Elementary Path

Initially we provide the definition of the equality, two operators and the predicate.

```
Definition brel_list : ∀ {S : Type}, brel S → brel(list S)
:= fix f {S} U x y :=
      match x, y with
         | nil, nil => true
         | nil, _ => false
         | _, nil => false
         | a::tla, b::tlb => andb (U a b) (f U tla tlb)
      end.
Definition S := nat.
Definition eqS := Arith.EqNat.beq_nat.
Definition brel_list_S  : brel (list S) := brel_list S eqS.
Variable c : cas_constant.
Definition brel_list_const : brel (cas_constant + list S )
:= brel_add_constant brel_list_S c.
Definition T := cas_constant + list S.
```

Listing 47: Elementary Path Equality

```
Fixpoint dic_order (l1 l2 : list S) : bool :=
match l1,l2 with
| nil,_ => true
| _,nil => false
| x::xl, y :: yl => if eqS x y
                    then dic_order xl yl
                    else x <? y
end.
Definition minS : binary_op (list S) :=
  λ l1 l2, if length l1 =? length l2
           then dic_minS l1 l2
           else left_shortest l1 l2.
Definition minT := bop_add_id minS c.
```

Listing 48: Elementary Path Additive Component

```
Definition app := List.app.
Definition appS := app S.
Definition appT := bop_add_ann appS c.
```

Listing 49: Elementary Path Multiplicative Component

```
Fixpoint elem_in_list  (S : Type)(eqS : brel S)(x : S)(l : list S) : bool :=
match l with
| nil => false
| y :: yl => orb (eqS x y) (elem_in_list S eqS x yl)
end.
Fixpoint dup_in_list (S : Type)(eqS : brel S)(l : list S): bool :=
match l with
| nil => false
| y :: yl =>  orb (elem_in_list S eqS y yl) (dup_in_list S eqS yl)
end.
Definition P : T -> bool :=  λ x,
match x with
| inr xl => dup_in_list S eqS xl
| inl _ => true
end.
```

Listing 50: Elementary Path Predicate

We also reasoning the properties for the additive component that it has the properties of associative, congruence, selective and commutative.

```
Lemma bop_list_minT_commutative : bop_commutative T brel_list_const minT.
Lemma bop_list_minT_selective : bop_selective T brel_list_const minT.
Lemma bop_list_minT_associative : bop_associative T brel_list_const minT.
Lemma bop_list_minT_congruence : bop_congruence T brel_list_const minT.
```

Listing 51: Properties for Min

By using the lemmas in Coq library for *app*, we can easily get the properties of the multiplicative component.

Then we need to discuss the properties of the predicate, and the properties between the predicate and the two binary operators.

```
Lemma P_true : pred_true T P (inl c).
Lemma P_congruence : pred_congruence T brel_list_const P.
Lemma P_min_decompose : pred_bop_decompose T P minT.
Lemma P_app_compose : pred_bop_compose T P appT.
```

Listing 52: Properties for the Predicate

Then based on the predicate we can define our reduction.

```
Definition uop_list : unary_op T := uop_predicate_reduce (inl c) P.
Definition bop_list_app : binary_op T := bop_fpr (inl c) P appT.
Definition bop_list_min : binary_op T := bop_fpr (inl c) P minT.
```

Listing 53: Elementary Path Reduction

The rest of the properties can be proved using the properties defined in the predicate reduction and generalized reduction section (not commutative for multiplicative component, and not distributive on both operator).

Finally we can construct our semiring.

```
Definition eqv_proofs_eq_T : eqv_proofs T (brel_reduce uop_list brel_list_const)
:= {|
     eqv_reflexive   := brel_reduce_list_const_reflexive
   ; eqv_transitive  := brel_reduce_list_const_transitive
   ; eqv_symmetric   := brel_reduce_list_const_symmetric
   ; eqv_congruence  := brel_reduce_list_const_congruence
   ; eqv_witness     := (inl c)
|}.
```

Listing 54: Equality Proof

```
Definition min_proofs  :
commutative_selective_semigroup_proofs T
(brel_reduce uop_list brel_list_const) bop_list_min
:= {|
  cssg_associative   := bop_list_min_associative
; cssg_congruence    := bop_list_min_congruence
; cssg_commutative   := bop_list_min_commutative
; cssg_selective     := bop_list_min_selective
|}.
```

Listing 55: Proof for Min Operator

```
Definition app_proofs:
semigroup_proofs T (brel_reduce uop_list brel_list_const) bop_list_app
:= {|
  sg_associative   := bop_list_app_associative
; sg_congruence    := bop_list_app_congruence
; sg_commutative_d    := bop_list_app_commutative_decidable
|}.
```

Listing 56: Proof for Plus Operator

```
Definition min_app_non_distributive_dioid_proofs :
bioid_proof T (brel_reduce uop_list brel_list_const)
bop_list_min bop_list_app (inl c) (inr nil)
:= {|
  bioid_left_distributive_decidable := bop_left_distributive_min_app_decidable
; bioid_right_distributive_decidable := bop_right_distributive_min_app_decidable
; bioid_zero_is_add_id     := bop_is_id_min
; bioid_one_is_mul_id      := bop_is_id_app
; bioid_zero_is_mul_ann     := bop_is_ann_app
; bioid_one_is_add_ann    := bop_is_ann_min
|}.
```

Listing 57: Proof for None Distributive Semiring Property

```
Definition min_app_non_distributive_dioid : selective_bioid T
:= {|
    sbioid_eq         := brel_reduce uop_list brel_list_const
  ; sbioid_add        := bop_list_min
  ; sbioid_mul        := bop_list_app
  ; sbioid_zero       := inl c
  ; sbioid_one        := inr nil
  ; sbioid_eqv        := eqv_proofs_eq_T
  ; sbioid_add_pfs    := min_proofs
  ; sbioid_mul_pfs    := app_proofs
  ; sbioid_pfs        := min_app_non_distributive_dioid_proofs
|}.
```

Listing 58: Elementary Path Semiring

A.1.9 Final Path Problem

Finally we construct our final Path problem semiring by using the lexicographic product we defined previously.

```
Variable ceiling : nat.
Variable c : cas_constant.
Definition T := cas_constant + list nat.
Definition min_plus_ceiling_dioid := min_plus_dioid ceiling.
Definition elementary_path_bioid := min_app_non_distributive_dioid c.
Definition add1 := csdioid_add nat min_plus_ceiling_dioid.
Definition mul1 := csdioid_mul nat min_plus_ceiling_dioid.
Definition add2 := sbioid_add T elementary_path_bioid.
Definition mul2 := sbioid_mul T elementary_path_bioid.
Definition eqN := csdioid_eq nat min_plus_ceiling_dioid.
Definition eqT := sbioid_eq T elementary_path_bioid.
Definition M := nat * T.
Definition eqM : brel M := brel_product eqN eqT.
Definition add :binary_op M := bop_llex eqN add1 add2.
Definition mul :binary_op M:= bop_product mul1 mul2.
Definition zero1 : nat := ceiling.
Definition one1  : nat := 0.
Definition zero2 : T := inl c.
Definition one2  : T := inr nil.
Definition zero  : M := (zero1,zero2).
Definition one   : M := (one1,one2).
```

Listing 59: Path Problem Basic Definition

```
Definition P := reduce_annihilators.P nat T eqN eqT zero1 zero2.
Lemma P_true : pred_true M P zero.
Lemma P_cong : pred_congruence M eqM P.
Lemma P_decompose_add : pred_bop_decompose M P add.
Lemma P_compose_mul : pred_bop_compose M P mul.
```

Listing 60: Path Problem Predicate

And we can define our reduction.

```
Definition uop_rap : unary_op M := reduce_annihilators.uop_rap nat T eqN eqT zero1 zero2.
Definition brel_eq_M :brel M := brel_reduce uop_rap eqM.
Definition bop_rap_add : binary_op M := bop_fpr zero P add.
Definition bop_rap_mul : binary_op M := bop_fpr zero P mul.
```

Listing 61: Path Problem Reduction

The rest of the properties can be proved using the properties defined in the predicate reduction and generalized reduction section (not commutative for multiplicative component, and not distributive on both operator).

Finally we can construct our path problem semiring.

```
Definition eqv_proofs_eq_T : eqv_proofs M brel_eq_M
:= {|
     eqv_reflexive   := brel_eq_M_reflexive
   ; eqv_transitive  := brel_eq_M_transitive
   ; eqv_symmetric   := brel_eq_M_symmetric
   ; eqv_congruence  := brel_eq_M_congruence
   ; eqv_witness     := zero
|}.
```

Listing 62: Equality Proof

```
Definition min_proofs  :
commutative_selective_semigroup_proofs M brel_eq_M bop_rap_add
:= {|
  cssg_associative   := bop_rap_add_associative
; cssg_congruence    := bop_rap_add_congruence
; cssg_commutative   := bop_rap_add_commutative
; cssg_selective     := bop_rap_add_selective
|}.
```

Listing 63: Proof for Min Operator

```
Definition app_proofs:
semigroup_proofs M brel_eq_M bop_rap_mul
:= {|
  sg_associative   := bop_rap_mul_associative
; sg_congruence    := bop_rap_mul_congruence
; sg_commutative_d    := bop_rap_mul_commutative_decidable
|}.
```

Listing 64: Proof for Plus Operator

```
Definition min_app_non_distributive_dioid_proofs :
bioid_proof M brel_eq_M bop_rap_add bop_rap_mul zero one
:= {|
bioid_left_distributive_decidable := bop_left_distributive_add_mul_decidable
; bioid_right_distributive_decidable := bop_right_distributive_add_mul_decidable
; bioid_zero_is_add_id     := bop_is_id_add_zero
; bioid_one_is_mul_id      := bop_is_id_mul_one
; bioid_zero_is_mul_ann    := bop_is_ann_mul_zero
; bioid_one_is_add_ann     := bop_is_ann_add_one
|}.
```

Listing 65: Proof for None Distributive Semiring Property

```
Definition min_app_non_distributive_dioid : selective_bioid M
:= {|
    sbioid_eq          := brel_eq_M
  ; sbioid_add         := bop_rap_add
  ; sbioid_mul         := bop_rap_mul
  ; sbioid_zero        := zero
  ; sbioid_one         := one
  ; sbioid_eqv         := eqv_proofs_eq_T
  ; sbioid_add_pfs     := min_proofs
  ; sbioid_mul_pfs     := app_proofs
  ; sbioid_pfs         := min_app_non_distributive_dioid_proofs
|}.
```

Listing 66: Elementary Path Semiring

Finally we defined and constructed all the stuff we want and all the properties we need to proof. Detailed proof could be found in my Coq file.