CAPP: Combinators for Algebraic Path Problems

Paper 40

Abstract—

Many complex path problems for directed graphs can be solved using a well-developed algebraic theory based on semirings. This theory represents an open-ended generalization of familiar shortest-path problems. All conventional path-finding algorithms (Dijkstra's, Bellman-Ford, etc) have been generalized to the semiring framework. However, defining complex semirings and proving that they have the algebraic properties required for the correctness of a given algorithm can be a tedious and error-prone task.

This paper describes our open-source implementation of Combinators for Algebraic Path Problems (CAPP). Using our CAPP library researchers and designers can explore a large class of path problems while automatically certifying algorithmic correctness. CAPP is a collection of combinators that build new algebraic structures. Proofs of algebraic properties are automatically constructed in a recursive, bottom-up, manner.

The CAPP library is implemented in the functional programming language OCaml. The correctness of this library was verified using the Coq theorem prover.

I. Introduction

Among the most widely known algorithms are those for computing shortest paths. It has long been known that these algorithms can be generalized to work with a large class of algebraic structures called semirings [3], [7], [4]. Generalized path problems are important in many areas such as internet routing research, operations research, and finance. However, defining specialized semirings and proving that they exhibit the algebraic properties required for the correct operation of a given algorithm can be a very tedious and error-prone task.

In this paper we describe an open-source library implementing Combinators for Algebraic Path Problems (CAPP). The library is implemented in OCaml [12]. CAPP is based on a collection of simple algebras and combinators that combine algebras to create new ones. The novel feature of these combinators is that proofs of algebraic properties are automatically synthesized in a bottom-up manner. For each generalized path algorithm we have implemented an algorithmic combinator that can be instantiated with a defined algebra only when it conforms to the algorithm's correctness requirements.

Users of the CAPP library can explore a large class of path problems while automating the tedious and error-prone task of proving correctness by following these steps.

(1) Define a new algebra using algebraic combinators.
(2) Instantiate a path algorithm with the defined algebra. This step may fail if the properties required for the correctness of the algebra are not met. In such a case, counter-examples to the properties not met are provided.
(3) Configure an adjacency matrix over the defined path algebra.
(4) Use the algorithm instantiated at Step 2 to solve the path problem defined by the adjacency matrix defined at Step 3.

CAPP is implemented as an OCaml library, so this type of step-by-step development can be done using OCaml's read-eval-print-loop. This OCaml library facilitates the open-ended integration of CAPP with other tools such as graph visualization, document generation, or the compilation of algebraic specifications from higher-level sources.

Section II provides a brief overview of semirings, path algebras, and path problems. Section II-A describes how our algebraic structures carry proofs of their algebraic properties. Combinators construct proofs from the proofs carried in their arguments. Section III presents CAPP by way of examples. Section IV describes how we use the Coq theorem prover [11] to verify the correctness of CAPP's OCaml combinators. We imagine that typical users will only need access to our OCaml implementation. The Coq development would be required only by those users interested in extending our language of verified combinators. Section V discusses related work while Section VI presents directions for future research.

This paper is not intended be a user's manual for CAPP library. The open-source CAPP repository includes a detailed user's guide. Our intention here is to give the reader a good understanding of the combinator concept, the expressive power of the CAPP library, and our solutions to several implementation challenges. We hope that the paper can be understood by those readers not familiar with the OCaml programming language or with the Coq theorem prover.

II. Path algebras

Computing shortest paths relies on two binary operations; $\min$ is used to select a best weights while $+$ is used to compute path weights from individual link weights. Researchers in the 1960's discovered that we can abstract away from $(\min,\ +)$ to algebraic structures of the form $(S,\ \oplus,\ \otimes,\ \overline{0},\ \overline{1})$, where $\oplus$ and $\otimes$ are binary operators over the set S, $\overline{0}$ is the identity for $\oplus$, and $\overline{1}$ is the identity for $\otimes$. The standard shortest-path algorithms can then be "made to work" as long as $\oplus$ and $\otimes$ obey certain algebraic properties. These properties are generally associated with algebraic structures called *semirings*. This section provides a very brief summary of the use of semirings in solving network path problems. The interested reader can find proofs of all claims made here in [5], [7], [4].

Shortest-paths enthusiasts may find the semiring notation perverse since in the shortest-paths case we have $\oplus = \min$, $\otimes = +$, $\overline{0} = \infty$, and $\overline{1} = 0$. However, this standard notation highlights the fact that semirings are themselves a generalization of the familiar ring $(\mathbb{R},\ +,\ \times,\ 0,\ 1)$.

Suppose we are given a directed graph $G = (V,\ E)$ and a weight function $w \in E \to S$ that assigns S-weights to every directed edge in the graph. The weight of a path $p = i_1, i_2, i_3, \cdots, i_k$ is defined as the $\otimes$-product

$$w(p) = w(i_1,\ i_2) \otimes w(i_2,\ i_3) \otimes \cdots \otimes w(i_{k-1},\ i_k),$$

and the empty path is given the weight $\overline{1}$ (which is 0 in the case of shortest paths!). We encode w in an $n \times n$ adjacency matrix, where n is the size of V, as

$$\mathbf{A}(i,\ j) = \begin{cases} \overline{1} & \text{if } i = j \\ w(i,\ j) & \text{if } (i,\ j) \in E \\ \overline{0} & \text{otherwise} \end{cases}$$

An adjacency matrix $\mathbf{A}$ then represents an instance of a *path problem*. A *solution* to a path problem, if one exists, is a matrix $\mathbf{A}^*$ such that

$$\mathbf{A}^*(i,\ j) = \bigoplus_{p \in \pi(i,\ j)} w(p) \tag{1}$$

where $\pi(i,\ j)$ denotes the set of all paths, possibly including cycles, from i to j. The "big-$\oplus$" notation implicitly assumes that $\oplus$ is associative and commutative.

In general, $\mathbf{A}^*$ may not exists. To ensure that $\mathbf{A}^*$ does exist we must constrain the topology of the graph (for example, to a DAG) require specific algebraic properties be obeyed. Figure 1 lists the algebraic properties needed to define a semiring. The structure $(S,\ \oplus,\ \otimes,\ \overline{0},\ \overline{1})$ is a semiring when it satisfies these properties:

$\mathrm{AS}(S,\ \oplus)$	:	$\oplus$ is associative
$\mathrm{AS}(S,\ \otimes)$	:	$\otimes$ is associative
$\mathrm{CM}(S,\ \oplus)$	:	$\oplus$ is commutative
$\mathrm{LD}(S,\ \oplus,\ \otimes)$	:	left distributivity holds
$\mathrm{RD}(S,\ \oplus,\ \otimes)$	:	right distributivity holds
$\mathrm{ID}(S,\ \oplus,\ \overline{0})$	:	$\overline{0}$ is the identity for $\oplus$
$\mathrm{ID}(S,\ \otimes,\ \overline{1})$	:	$\overline{1}$ is the identity for $\otimes$
$\mathrm{IA}(S,\ \otimes,\ \overline{0})$	:	$\overline{0}$ is the annihilator for $\otimes$

The semiring properties alone are not sufficient to guarantee that $\mathbf{A}^*$ exists. In this paper we consider algebraic properties that are strong enough so that $\mathbf{A}^*$ will exist in any directed graph. To this end we define a subclass of semirings called *path algebras* in which these two additional properties hold:

$\mathrm{IP}(S,\ \oplus)$	:	$\oplus$ is idempotent
$\mathrm{IA}(S,\ \oplus,\ \overline{1})$	:	$\overline{1}$ is the annihilator for $\oplus$

These properties ensure, among other things, that there is no point of going around a loop when computing best Equation 1. When $\oplus$ be a commutative and idempotent the (left and right) "natural orders" defined as

$$a \leq^L_\oplus b \equiv a = a \oplus b \qquad a \leq^R_\oplus b \equiv b = a \oplus b$$

are both are partial orders. We can then think of Equation 1 as representing a minimization over $\leq^L_\oplus$ or maximization over the dual order $\leq^R_\oplus$.

The simplest method of computing $\mathbf{A}^*$ over a path algebra is by using matrix multiplication. Suppose $\mathbf{A}$ is a $m \times n$ matrix and $\mathbf{B}$ is a $n \times p$ matrix. Define their product, a $m \times p$ matrix, as

$$(\mathbf{A} \otimes \mathbf{B})(i,\ j) \equiv \bigoplus_{1 \leq q \leq n} \mathbf{A}(i,\ q) \otimes \mathbf{B}(q,\ j).$$

From this we can define matrix exponentiation for any square $(n \times n)$ matrix as

$$\mathbf{A}^0 = \mathbf{I} \qquad \mathbf{A}^{k+1} = \mathbf{A} \otimes \mathbf{A}^k$$

where $\mathbf{I}$ is the identity with respect to matrix multiplication,

$$\mathbf{I}(i,\ j) = \begin{cases} \overline{1} & \text{(if } i = j\text{)} \\ \overline{0} & \text{(otherwise)} \end{cases}$$

For path algebras we can then compute $\mathbf{A}^* = \mathbf{A}^{n-1}$ where $\mathbf{A}$ is an $n \times n$ matrix.

Figure 2 presents Dijkstra's algorithm generalized to work with path algebras. First, we need to discuss the partial order associated with the additive component $\oplus$ of a path algebra. However, Dijkstra's algorithm requires a *total order*, so we can use it only when the additive component of a path algebra

associative:
$\mathrm{AS}(S,\ \bullet) \equiv \forall a, b, c \in S,\ (a \bullet b) \bullet c = a \bullet (b \bullet c)$
commutative:
$\mathrm{CM}(S,\ \bullet) \equiv \forall a, b \in S,\ a \bullet b = b \bullet a$
idempotent:
$\mathrm{IP}(S,\ \bullet) \equiv \forall a \in S,\ a \bullet a = a$
is identity:
$\mathrm{ID}(S,\ \bullet,\ \alpha) \equiv \forall a \in S,\ a = \alpha \bullet a \wedge a \bullet \alpha = a$
is annihilator:
$\mathrm{IA}(S,\ \bullet,\ \omega) \equiv \forall a \in S,\ \omega = \omega \bullet a \wedge a \bullet \omega = a$
left distributive:
$\mathrm{LD}(S,\ \oplus,\ \otimes) \equiv$
$\forall a, b, c \in S,\ a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$
right distributive:
$\mathrm{RD}(S,\ \oplus,\ \otimes) \equiv$
$\forall a, b, c \in S,\ (b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a)$

Fig. 1. The algebraic properties needed to define semirings and path algebras.

Input : a source vertex $i \in V$ and
an $n \times n$ adjacency matrix $\mathbf{A}$.
Output : a $1 \times n$ row vector $\mathbf{D}$.

$S \leftarrow \{i\}$
$\mathbf{D}(i) \leftarrow \overline{1}$
for each $q \in V - \{i\} : \mathbf{D}(q) \leftarrow \mathbf{A}(i,\ q)$
while $S \neq V$
find $q \in V - S$ such that $\mathbf{D}(q)$ is $\leq^L_\oplus$ -minimal
$S \leftarrow S \cup \{q\}$
for each $j \in V - S$
$\mathbf{D}(j) \leftarrow \mathbf{D}(j) \oplus (\mathbf{R}(q) \otimes \mathbf{A}(q,\ j))$

Fig. 2. Dijkstra's algorithm for selective path algebras.

yields a total order. This can be achieved by insisting that $\oplus$ is *selective*:

$$\mathrm{SL}(S,\ \oplus) \quad \equiv \quad \forall a, b \in S,\ a \oplus b = a \vee a \oplus b = b$$

Note that selectivity implies idempotence. It can be shown that for each j we have $\mathbf{D}(j) = \mathbf{A}^*(i,\ j)$.

A. Proof-carrying combinators

The CAPP library defines OCaml types `'a pa` and `'a selective_pa` for implementing path algebras and selective path algebras. These are polymorphic types where `'a` is a variable ranging over types. Users can construct instances `t pa` and `t selective_pa` for many concrete types `t`.

The two path algorithms discussed in Section II are implemented with these functions:

```
pa_mtx_solver :
 'a pa -> 'a sq_mtx -> 'a sq_mtx

pa_dijkstra_solver :
  'a selective_pa -> nat
    -> 'a sq_mtx -> 'a row_vector
```

Since combinators construct new path algebras it may seem reasonable to insist that every combinator take path algebras as arguments and return a path algebra as a result. We start by defining one combinator, the direct product, that conforms to this scheme. Then we argue that this approach seriously reduces the expressive power of a combinator language or it leads to an exponential explosion of types and combinators.

A semiring $(S,\ \oplus,\ \otimes)$ is comprised of two *semigroups* $(S,\ \oplus)$ and $(S,\ \otimes)$. Our combinators over semirings are usually defined using two semigroup combinators.

Suppose we have semigroups $(S,\ \bullet)$ and $(T,\ \circ)$. We define their *direct product* as

$$(S,\ \bullet) \times (T,\ \circ) \equiv (S \times T,\ {}_{\bullet}\!\times_{\circ})$$

where

$$(s,\ t)\ {}_{\bullet}\!\times_{\circ}\ (s',\ t') \equiv (s \bullet s',\ t \circ t').$$

Now, define a direct product on semirings as

$$\begin{aligned}(S,\ \oplus,\ \otimes,\ \overline{0}_S,\ \overline{1}_S) \times (T,\ \boxplus,\ \boxtimes,\ \overline{0}_T,\ \overline{1}_T)\\ \equiv (S \times T,\ {}_{\oplus}\!\times_{\boxplus},\ {}_{\otimes}\!\times_{\boxtimes},\ (\overline{0}_S,\ \overline{0}_T),\ (\overline{1}_S,\ \overline{1}_T))\end{aligned}$$

It is fairly easy to check that if the both arguments are path algebras, then their direct product is a path algebra. We could then implement this as a function of type

```
pa_Product : 'a pa -> 'b pa -> ('a * 'b) pa
```

The type `('a * 'b)` represents the type of pairs.

However, when we look closely at how the metrics of many routing protocols are defined it becomes very difficult to see how they might be defined using combinators that conform to this simple scheme. Lexicographic comparison provides a good example of the issues that can arise. Consider *widest shortest paths (WSP)* (see [18] or [4] page 30). Path weights are of the form $(d,\ c)$ where d represents distance and c represents capacity. If path p has weight $(d,\ c)$ and we extend it with an arc of weight $(d',\ c')$ the resulting path has weight $(d'+d,\ c' \min c)$. In other words the multiplicative component is the direct product ${}_{+}\!\times_{\min}$. We want the additive component to compare path weights $(d,\ c)$ and $(d',\ c')$ lexicographically. That is, if $d < d'$ then $(d,\ c)$ is selected as best. If $d' < d$ then $(d',\ c')$ is selected. Otherwise, if $d = d'$, then the result will be $(d,\ c \max c')$.

left cancellative:
$\mathrm{LC}(X,\ \bullet) \quad \equiv \quad \forall a, b, c \in X,\ c \bullet a = c \bullet b \Rightarrow a = b$
right cancellative:
$\mathrm{RC}(X,\ \bullet) \quad \equiv \quad \forall a, b, c \in X,\ a \bullet c = b \bullet c \Rightarrow a = b$
left constant:
$\mathrm{LK}(X,\ \bullet) \quad \equiv \quad \forall a, b, c \in X,\ c \bullet a = c \bullet b$
right constant:
$\mathrm{RK}(X,\ \bullet) \quad \equiv \quad \forall a, b, c \in X,\ a \bullet c = b \bullet c$

Fig. 3. The four auxiliary semigroup properties needed to establish distributivity for the lexicographic-product combinator.

Keeping in mind the relationship between order and semigroups described in Section II, we capture lexicographic comparison with a combinator over semigroups:

$$(S,\ \oplus)\ \vec{\times}\ (T, \boxplus) \equiv (S \times T,\ {}_{\oplus}\vec{\times}_{\boxplus})$$

where

$$\begin{aligned}(s,\ t)\ {}_{\oplus}\vec{\times}_{\boxplus}\ (s',\ t') \quad &\equiv \quad (s \oplus s',\ t \boxplus t')\\ &\quad\ (\text{if } s = s \oplus s' = s')\\ (s,\ t)\ {}_{\oplus}\vec{\times}_{\boxplus}\ (s',\ t') \quad &\equiv \quad (s \oplus s',\ t)\\ &\quad\ (\text{if } s = s \oplus s' \neq s')\\ (s,\ t)\ {}_{\oplus}\vec{\times}_{\boxplus}\ (s',\ t') \quad &\equiv \quad (s \oplus s',\ t')\\ &\quad\ (\text{if } s \neq s \oplus s' = s')\end{aligned}$$

Putting this together with the direct product we have a combinator that could be used to define algebras such as the one used in WSP:

$$(S,\ \oplus,\ \otimes)\ \vec{\times}\ (T, \boxplus,\ \boxtimes) \quad \equiv \quad (S \times T,\ {}_{\oplus}\vec{\times}_{\boxplus},\ {}_{\otimes}\!\times_{\boxtimes})$$

When does this construction result in a path algebra? It is clear from the definition of ${}_{\oplus}\vec{\times}_{\boxplus}$ that it is only well-defined when $\oplus$ is selective. As pointed out in [10], the troublesome properties are left and right distributivity. In order to establish distributivity we need both $(S,\ \oplus,\ \otimes)$ and $(T, \boxplus,\ \boxtimes)$ to be distributive. But that is not enough. To determine distributivity we must introduce the four *auxiliary properties* on semigroups listed in Figure 3. Using these properties, Figure 4 contains a formula that determines when the lexicographic-product is left distributive. A similar formula is holds with right-versions of these properties.

We note that the result of Figure 4, as well as all other similar results in this paper, are asserted under the assumption that all carrier sets are non-trivial. That is, each carrier set contains at least two elements. This seems to us a reasonable constraint since we are interested in constructing useful path algebras and without this simplifying assumption formulas

$$\begin{array}{c}\mathrm{LD}((S,\ \oplus,\ \otimes) \vec{\times} (T, \boxplus,\ \boxtimes)) \leftrightarrow \\ \mathrm{LD}(S,\ \oplus,\ \otimes) \wedge \mathrm{LD}(T,\ \boxplus,\ \boxtimes) \wedge \\ (\mathrm{LC}(X,\ \otimes) \vee \mathrm{LK}(X,\ \boxtimes).\end{array}$$

Fig. 4. A formula from [10] characterizing when the left lexicographic product is left distributive. This formula holds under the assumptions that $\oplus$ is both commutative and selective and that the carrier sets S and T are not trivial.

such as the one in Figure 4 are greatly complicated in the presence of empty or singleton carriers.

It is important to notice that no semiring, and so no path algebra, can satisfy cancellativity. $\overline{0}$ is a multiplicative annihilator and so it is always the case that

$$\overline{0} \otimes a = \overline{0} \otimes b = \overline{0},$$

but this does not imply that $a = b$!

How do we implement such a combinator? We describe two approaches. The first approach implements *strict combinators* whose types tell us exactly what properties hold for arguments and results. Suppose that both $(S,\ \oplus,\ \otimes)$ and $(T, \boxplus,\ \boxtimes)$ are left and right distributive. From Figure 4 (and a similar version for right distributivity) we see that their lexicographic product will be left and right distributive when

$$(\mathrm{LC}(X,\ \otimes) \vee \mathrm{LK}(X,\ \boxtimes) \wedge (\mathrm{RC}(X,\ \otimes) \vee \mathrm{RK}(X,\ \boxtimes).$$

This give us four case:

1) $\mathrm{LC}(X,\ \otimes) \wedge \mathrm{RC}(X,\ \otimes)$
2) $\mathrm{LC}(X,\ \otimes) \wedge \mathrm{RK}(X,\ \boxtimes)$
3) $\mathrm{RC}(X,\ \otimes) \wedge \mathrm{LK}(X,\ \boxtimes)$
4) $\mathrm{LK}(X,\ \boxtimes \wedge \mathrm{LK}(X,\ \boxtimes)$

Under our assumption of non-trivial carrier sets the last case is impossible and can be eliminated.

Using strict combinators we would implement three distinct combinators corresponding to these cases. The types used would tell us exactly which properties hold:

```
'a selective_cancellative_pa_wo_zero
  -> 'b pa -> ('a * 'b) pa_wo_zero

'a selective_left_cancellative_pa_wo_zero
  -> 'b right_constant_pa_wo_one
    -> ('a * 'b) pa_wo_zero_one

'a selective_right_cancellative_pa_wo_zero
  -> 'b left_constant_pa_wo_one
    -> ('a * 'b) pa_wo_zero_one
```

But why stop there? We could then implement additional versions:

```
'a selective_cancellative_pa_wo_zero
 -> 'b selective_pa
 -> ('a * 'b) selective_pa_wo_zero

'a selective_cancellative_pa_wo_zero
-> 'b selective_cancellative_pa ->
('a * 'b) selective_cancellative_pa_wo_zero
```

. . .
. . .

To compound the problem other combinators will require additional auxiliary properties and the strict combinators approach quickly confronts an exponential explosion of types and combinators.

In order to avoid this problem the CAPP library implements *relaxed combinators*. In this approach the algebraic structures carry *certificates* associated with every property and combinators use their argument's certificates to construct the certificates for their result.

Define a *bi-semigroup* to be of the form $(S,\ \oplus,\ \otimes)$ where we insist only that both $\oplus$ and $\otimes$ are associative. The OCaml type `'a ebs` implements bi-semigroups that carry certificates asserting which properties hold or don't hold. That is, for every property of Figure 1, selectivity, and the auxiliary properties required by our combinators, the `'ebs` structure will carry a certificate asserting that the property does or does not hold. As explained in Section IV, our OCaml combinators represent the "computational content" of combinators implemented in the Coq theorem prover that carry full proofs of their properties. That section explains why we can trust the certificates computed by combinators of the CAPP library.

Since `ebs` structures carry certificates we can implement a the lexicographic product as a single combinator:

```
ebs_LeftLexProduct :
  'a ebs -> 'b ebs -> ('a * 'b) ebs
```

This function checks that the additive component of its first argument is commutative and selective. If this is not the case then it will raise an exception (thus the prefix `e` of `ebs`). Otherwise, this combinator will construct all certificates for its result.

What do these certificates look like? When the result of `ebs_LeftLexProduct` is left distributive it will return a result containing the certificate

```
Certify_Left_Distributive
```

Otherwise, it will return a certificate

```
Certify_Not_Left_Distributive
    ((a, b), ((c, d), (e, f)))
```

where the expressions `a` - `f` represent a *counter-example* for left distributivity. That is, they are values such that

$$\begin{array}{c}(a,b) \otimes\times\boxtimes ((c,d) \oplus \vec{\times}_{\boxplus} (e,f)) \\ \neq \\ ((a,b) \otimes\times\boxtimes (c,d)) \oplus \vec{\times}_{\boxplus} ((a,b) \otimes\times\boxtimes (e,f)).\end{array}$$

How is this accomplished? Let's first switch our focus from certificates to proofs. We recast the $\leftrightarrow$ of Figure 4 as two

implications.

$$\begin{array}{l}\mathrm{LD}(S,\ \oplus,\ \otimes) \wedge \mathrm{LD}(T,\ \boxplus,\ \boxtimes) \wedge \\ (\mathrm{LC}(X,\ \otimes) \vee \mathrm{LK}(X,\ \boxtimes) \\ \rightarrow \mathrm{LD}((S,\ \oplus,\ \otimes) \vec{\times} (T, \boxplus,\ \boxtimes))\end{array}$$

$$\begin{array}{l}\neg\mathrm{LD}(S,\ \oplus,\ \otimes) \vee \neg\mathrm{LD}(T,\ \boxplus,\ \boxtimes) \vee \\ (\neg\mathrm{LC}(X,\ \otimes) \wedge \neg\mathrm{LK}(X,\ \boxtimes) \\ \rightarrow \neg\mathrm{LD}((S,\ \oplus,\ \otimes) \vec{\times} (T, \boxplus,\ \boxtimes))\end{array}$$

Note that this reformulation gives us a way of synthesizing a proof of left distributivity, or its negation, in a bottom-up way from proofs associated with the arguments of the combinator. We construct similar rules for every property and every combinator in our language. This gives a way of synthesizing a (constructive!) proof of P $\vee \neg$P.

However, there is a big problem if we use negation defined in the traditional way ($\neg P \equiv P \implies$ FALSE). A proof of $\neg$LD is not very informative. It does tell us *why* left distributivity holds.

We have found that computing counter examples is much more informative and provides CAPP users with valuable feedback. To achieve this we compute a proof of P$\vee \sim$P for every property P, where $\sim P$ is a *strong negation* [17], [9]. We are treating strong negation as a syntactic transformation on formulas:

$$\begin{array}{rcl}\sim P & \equiv & \neg P \ \ (P \text{ atomic}) \\ \sim\neg P & \equiv & P \\ \sim(P \wedge Q) & \equiv & \sim P \vee \sim Q \\ \sim(P \vee Q) & \equiv & \sim P \wedge \sim Q \\ \sim(P \implies Q) & \equiv & P \wedge \sim Q \\ \sim\forall x \in S, P(x) & \equiv & \exists x \in S, \sim P(x) \\ \sim\exists x \in S, P(x) & \equiv & \forall x \in S, \sim P(x)\end{array}$$

For example,

$$\sim\mathrm{LD}(S,\ \oplus, \otimes) = \exists a, b, c \in S,\ a \otimes (b \oplus c) \neq (a \otimes b) \oplus (a \otimes b).$$

We give the resulting formula a new name, $\sim$NOTLD and do the same with all other properties. Thus, the "bottom-up" rule for showing that the lexicographic product is not left distributive becomes

$$\begin{array}{l}\mathrm{NOTLD}(S,\ \oplus,\ \otimes) \vee \mathrm{NOTLD}(T,\ \boxplus,\ \boxtimes) \vee \\ (\mathrm{NOTLC}(X,\ \otimes) \wedge \mathrm{NOTLK}(X,\ \boxtimes) \\ \rightarrow \mathrm{NOTLD}((S,\ \oplus,\ \otimes) \vec{\times} (T, \boxplus,\ \boxtimes))\end{array}$$

It is important to understand that now proofs of the assumptions of this rule will contain counter-examples from which the counter example to left-distributivity can be constructed.

How are such proofs represented with certificates in OCaml? First, we need a type to represent (constructive) disjunction:

```
type ('a, 'b) sum = Inl of 'a | Inr of 'b
```

We call this a `sum` because it will perform double duty to represent disjoint unions in many of our combinators. Certificates for left distributivity are then defined as simple data types in OCaml.

```
type bs_left_distributive =
  Certify_Left_Distributive

type 'a bs_not_left_distributive =
  Certify_Not_Left_distributive
    of ('a * ('a * 'a))

type 'a bs_left_distributive_decidable =
  (bop_commutative,
   'a bop_not_commutative) sum
```

```
let bops_llex_product_left_distributive_decide
     rS rT argT addS mulS addT mulT
     wS wT ldS_d ldT_d lcS_d lkT_d =
match ldS_d with
| Inl Certify_Left_Distributive ->
  (match ldT_d with
  | Inl Certify_Left_Distributive ->
    (match lcS_d with
    | Inl Certify_Left_Cancellative ->
        Inl Certify_Left_Distributive
    | Inr (Certify_Not_Left_Cancellative (s1, (s2, s3))) ->
      (match lkT_d with
      | Inl Certify_Left_Constant ->
          Inl Certify_Left_Distributive
      | Inr (Certify_Not_Left_Constant (t1, (t2, t3))) ->
          Certify_Not_Left_Distributive
           (if (eqS (addS s2 s3) s2)
            then if eqS (addS s2 s3) s3
                 then ((s1, t1), ((s2, t2), (s3, t3)))
                 else  if eqS (mulS s1 s2) (addS (mulS s1 s2) (mulS s1 s3))
                       then if eqT (mulT t1 t2) (addT (mulT t1 t2) (mulT t1 t3))
                            then ((s1, t1), ((s2, t3), (s3, t2)))
                            else ((s1, t1), ((s2, t2), (s3, t3)))
                       else ((s1, t1), ((s2, t2), (s3, t3)))
            else if eqS (addS s2 s3) s3
                 then if eqT (mulT t1 t3) (addT (mulT t1 t2) (mulT t1 t3))
                      then ((s1, t1), ((s2, t3), (s3, t2)))
                      else ((s1, t1), ((s2, t2), (s3, t3)))
                 else ((s1, t1), ((s2, t2), (s3, t3)))
           )
  | Inr (Certify_Not_Left_Distributive (t1, (t2, t3))) ->
        Inr (Certify_Not_Left_Distributive  ((wS, t1), ((wS, t2), (wS, t3)))))
| Inr (Certify_Not_Left_Distributive (s1, (s2, s3))) ->
      Inr (Certify_Not_Left_Distributive  ((s1, wT), ((s2, wT), (s3, wT))))
```

Fig. 5. An OCaml decision procedure to determine if the (left) lexicographic-product is left distributive or not.

Figure 5 shows the OCaml decision procedure used in the implementation of `ebs_LeftLexProduct` to computes the certificate for left distributivity. Similar procedures are used to compute certificates for all other properties. The reader is not expected to understand this code, only to be reassured that it has been verified correct using the Coq theorem prover. For example, the nested conditional expressions in this code mirror the structure of a proof by case analysis in Coq.

III. CAPP BY EXAMPLES

In order to construct path algebras with relaxed combinators and use one of the path algorithms we need two coercion: functions:

```
cast_ebs_to_pa  : 'a ebs -> 'a pa

cast_ebs_to_selective_pa  : 'a ebs
  -> 'a selective_pa
```

Applications of these functions will raise an exception when the argument structure does not have the properties required. In that case the exception describes the reasons for failure. At any point while constructing an algebra we can query our results with this function:

```
ebs_describe : 'a ebs -> unit
```

The result type `unit` here simply indicates that the function has a side-effect. In this case it prints out a (rather verbose) description of an algebra including all of its properties.

A. Variations on shortest paths

We start with path algebras for shortest paths SP and highest capacity paths CP.

name	S	$\oplus$	$\otimes$	$\overline{0}$	$\overline{1}$
SP	$\{\infty\} \uplus \mathbb{N}$	$\min'$	$+'$	∞	0
CP	$\{\infty\} \uplus \mathbb{N}$	$\max'$	$\min'$	$\mathrm{Inr}\ 0$	$\mathrm{Inl}\ \infty$

Here $\uplus$ represents a disjoint (tagged) union,

$$X \uplus Y \equiv \{\mathrm{Inl}\ x \mid x \in X\} \cup \{\mathrm{Inr}\ y \mid y \in Y\}.$$

The functions $\min'$, $\max'$, and $+'$ represent the functions $\min$, $\max$, and $+$ extended to the disjoint union.

We will construct SP and CP from these structures:

name	S	$\oplus$	$\otimes$	$\overline{0}$	$\overline{1}$
MinPlus	$\mathbb{N}$	min	$+$		0
MaxMin	$\mathbb{N}$	max	min	0	

We need to add a $\overline{0}$ to MinPlus and a $\overline{1}$ to MaxMin. For this we will use general combinators.

This combinator adds an identity:

$$\textbf{AddId}\ c\ (S,\ \bullet) \equiv (\{c\} \uplus S,\ \bullet^c_{id})$$

where

$$\begin{array}{rcl} \mathrm{Inr}(s) \bullet^c_{id} \mathrm{Inr}(t) & \equiv & \mathrm{Inr}(s \bullet t) \\ \mathrm{Inr}(s) \bullet^c_{id} \mathrm{Inl}(c) & \equiv & \mathrm{Inr}(s) \\ \mathrm{Inl}(c) \bullet^c_{id} \mathrm{Inr}(t) & \equiv & \mathrm{Inr}(t) \\ \mathrm{Inl}(c) \bullet^c_{id} \mathrm{Inl}(c) & \equiv & \mathrm{Inl}(c) \end{array}$$

Similarly, this combinator adds an annihilator:

$$\textbf{AddAn}\ c\ (S,\ \bullet) \equiv (\{c\} \uplus S,\ \bullet^c_{an})$$

where

$$\begin{array}{rcl} \mathrm{Inr}(s) \bullet^c_{an} \mathrm{Inr}(t) & \equiv & \mathrm{Inr}(s \bullet t) \\ \mathrm{Inr}(s) \bullet^c_{an} \mathrm{Inl}(c) & \equiv & \mathrm{Inl}(c) \\ \mathrm{Inl}(c) \bullet^c_{an} \mathrm{Inr}(t) & \equiv & \mathrm{Inl}(c) \\ \mathrm{Inl}(c) \bullet^c_{an} \mathrm{Inl}(c) & \equiv & \mathrm{Inl}(c) \end{array}$$

These are combined to form bi-semigroup combinators:

$$\begin{array}{rcl} \textbf{AddZero}\ c\ (S,\ \oplus,\ \otimes) & \equiv & (S \uplus \{c\},\ \oplus^c_{id},\ \otimes^c_{an}) \\ \textbf{AddOne}\ c\ (S,\ \oplus,\ \otimes) & \equiv & (S \uplus \{c\},\ \oplus^c_{an},\ \otimes^c_{id}) \end{array}$$

These mathematical constructions are implemented in CAPP with structures and combinators:

```
ebs_MinPlus : nat ebs
ebs_MaxMin  : nat ebs
ebs_AddZero : constant -> 'a ebs ->
                ((constant, 'a) sum) ebs
ebs_AddOne : constant -> 'a ebs ->
               ((constant, 'a) sum) ebs
```

We implement the disjoint union using the `sum` type described in Section II-A. The type `constant` represents CAPP constants. We can construct constants like this,

```
let infinity = mk_constnt "INF" "\\infty";;
```

Here the first string argument is for ASCII display, while the second is for LaTeX-ed display.

We implement SP and CP as follows.

```
let ebs_SP =
  ebs_AddZero infinity ebs_MinPlus;;
let pa_SP =
  cast_ebs_to_selective_pa ebs_SP ;;
let ebs_CP =
  ebs_AddOne infinity ebs_MaxMin;;
let pa_CP =
  cast_ebs_to_selective_pa ebs_CP ;;
```

Both `pa_SP` and `pa_CP` have type

```
((constant, nat) sum) selective_pa.
```

Suppose we configure a matrix[1] `mtx_a` corresponding to Figure 6 (a). Note that each weight d in the figure is implemented with a value `Inr d`. The weight of the shortest path from node 0 to node 2 is computed this way:

```
row_vector_get 2
  (pa_dijkstra_solver pa_SP 0 mtx_a);;
```

This will return the value `Inr 2`.

In a similar way we can configure a matrix `mtx_b` corresponding to Figure 6 (b) and compute the highest capacity path from node 0 to node 2:

```
row_vector_get 2
  (pa_dijkstra_solver pa_CP 0 mtx_b);;
```

This will return the value `Inr 10`.

Section II-A described combinators for the direct product and the lexicographic product. These are available in CAPP:

```
ebs_Product :
  'a ebs -> 'b ebs -> ('a * 'b) ebs
ebs_LeftLexProduct :
  'a ebs -> 'b ebs -> ('a * 'b) ebs
```

First, let's try the direct product.

```
let ebs_SP_x_CP = ebs_Product ebs_SP ebs_CP;;
let pa_SP_x_CP = cast_ebs_to_pa ebs_sp_x_cap;;
```

Inspecting `ebs_SP_x_CP` using the `ebs_describe` will show is that this is a path algebra, but not a selective. The computed counter-example to selectivity is the pair

```
((Inr 1, Inr 1), (Inr 0, Inr 0))
```

since their sum is `(Inr 0, Inr 1)`.

Suppose we have configured a matrix `mtx_c` corresponding to Figure 6 (c). In this case a weight $(d,\ c)$ will be implemented as `(Inr d, Inr c)`. The weight on any arc that does not exist (such as $(0,\ 4)$) is the $\overline{0}$, `(Inl infinity, Inr 0)`, and every entry along the

[1]Our user's guide provides details on matrix configuration and display.

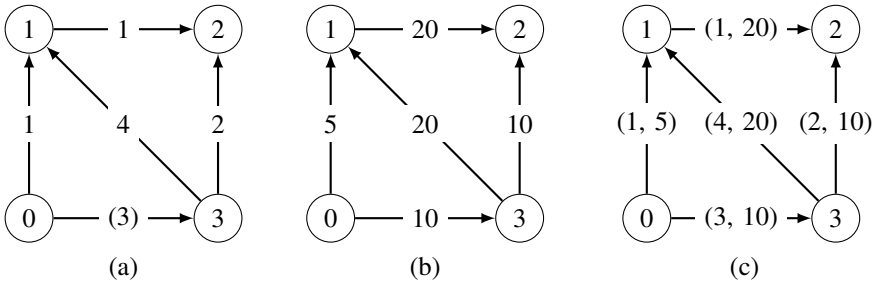

Fig. 6. Three labellings of the same underlying graph. (a) Shortest paths. (b) Highest capacity (widest) paths. (c) Combines (a) and (b) for two examples: the direct product and widest-shortest paths (WSP) using a lexicographic product.

diagonal of will contain the $\overline{1}$, `(Inr 0, Inl infinity)`. We can then compute the solution

```
mtx_get 0 2
  (pa_matrix_solver pa_SP_x_CP mtx_c)
```

which returns the value `(Inr 5, Inr 10)`.

Let's replace the direct product with the lexicographic product in order to implement widest-shortest paths:

```
let ebs_WSP =
  ebs_LeftLexProduct ebs_SP ebs_CP;;
let pa_WSP =
  cast_ebs_to_selective_pa ebs_WSP;;
```

This time the cast fails and an error is reported. In particular, this `ebs_WSP` is not distributive! For example, the decision procedure of Figure 5 computes this counter-example to left distributivity:

```
((Inl infinity, Inl infinity),
   ((Inr 0, Inr 1), (Inr 1, Inr 0)))
```

The problem is that we have added a zero too soon. The algebra is defind correctly as

```
let ebs_WSP =
  ebs_AddZero infinity
   (ebs_LefLexProduct
      ebs_MinPlus ebs_CP);;
let pa_WSP =
  cast_ebs_to_selective_pa ebs_WSP;;
```

This time the cast succeeds.

Suppose we have configured a matrix `mtx_d` corresponding to Figure 6 (c), but this time with weights consistent with `pa_WSP`. That is, a weight $(d,\ c)$ is implemented with `Inr(d, Inr c)`. The weight on any arc that does not exist is the $\overline{0}$, `Inl infinity`, and every entry along the diagonal of will contain the $\overline{1}$, `Inr(0, Inl infinity)`. We can then find the weight of the shortest path from node 0 to node 2:

```
row_vector_get 2
  (pa_dijkstra_solver pa_WSP 0 mtx_d);;
```

This will return the value `Inr(2, Inr 5)`.

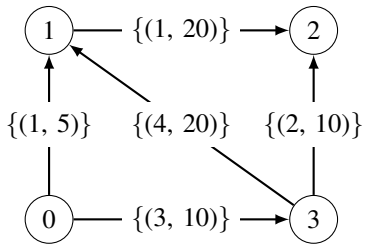

Fig. 7. This example is taken from [19].

B. Multi-Objective Optimization

We now construct a path algebra for the running example from the SIGCOMM 2020 paper [19]. Consider again the three configurations in Figure 6. We will focus on the three paths from node 0 to node 2:

$$p_1 = [0,\ 1,\ 2] \quad p_2 = [0,\ 3,\ 2] \quad p_3 = [0,\ 3,\ 1,\ 2].$$

Using shortest paths (Figure 6 (a)), these paths will have weights

$$w_a(p_1) = \text{Inr } 2 \quad w_a(p_2) = \text{Inr } 5 \quad w_a(p_3) = \text{Inr } 8.$$

Using widest paths (Figure 6 (b)), these paths will have weights

$$w_b(p_1) = \text{Inr } 5 \quad w_b(p_2) = \text{Inr } 10 \quad w_b(p_3) = \text{Inr } 10.$$

Using the direct product (Figure 6 (c)), these paths will have weights

$$w_c(p_1) = (\text{Inr } 2,\ \text{Inr } 5) \quad w_c(p_2) = (\text{Inr } 5,\ \text{Inr } 10)$$
$$w_c(p_3) = (\text{Inr } 8,\ \text{Inr } 10).$$

As shown in the previous section, the weight of the solution is $(\text{Inr } 2,\ \text{Inr } 10)$, which does not correspond to the weight of any single path!

How can we tie the two weights to a single path? Looking at the three path weights we can see that there is a *trade-off* between the distance metric and the capacity metric. Path p_1 has the shortest path of weight 2, while it has the worst capacity at 5. On the other hand, path p_2 has a higher capacity (10), but a longer path (5). A standard way of dealing with such trade-offs is to compute a *Pareto Frontier* comprised of those elements of a set that are not dominated by any other

element. In this case, the Pareto Frontier of the three path weights

$$\{(\text{Inr } 2,\ \text{Inr } 5),\ (\text{Inr } 5,\ \text{Inr } 10),\ (\text{Inr } 8,\ \text{Inr } 10)\}$$

is the set $\{(\text{Inr } 2,\ \text{Inr } 5),\ (\text{Inr } 5,\ \text{Inr } 10)\}$ since in the product order we have $(\text{Inr } 5, \text{Inr } 10) < (\text{Inr } 8, \text{Inr } 10)$. The Pareto Frontier contains both $(\text{Inr } 2,\ \text{Inr } 5)$ and $(\text{Inr } 5, \text{Inr } 10)$ because these two values as incomparable (neither dominates the other).

This is an instance of a *Multi-Objective Optimization (MOO)*. The survey [6] includes many examples of MOO in the context of network routing problems. However, that survey does not frame the problem in terms of semirings. How can we compute Pareto Frontier within the semiring framework?

First, suppose that $\leq$ is a partial order on a set S. For any subset $X \subseteq S$, we can compute its Pareto Frontier with respect to $\leq$ as

$$\min_{\leq} X \equiv \{x \in X \mid \forall y \in X, \neg(y < x)\}$$

where $y < x$ denotes $(y \leq x) \wedge (y \neq x)$.

For the additive component of a path algebra we can construct this commutative and idempotent semigroup:

$$(\{X \subseteq S \mid \min_{\leq} X = X\},\ \cup^{\min}_{\leq})$$

where

$$X \cup^{\leq}_{\min} Y \equiv \min_{\leq}(X \cup Y).$$

For the multiplicative component, we start with a semigroup $(S,\ \otimes)$ and then construct this semigroup that *lifts* $\otimes$ to sets and then computes a Pareto Frontier,

$$(\{X \subseteq S \mid \min_{\leq} X = X\},\ \otimes^{\min}_{\leq})$$

where $X \otimes^{\leq}_{\min} Y \equiv \min_{\leq}\{x \otimes y \mid x \in X,\ y \in Y\}$.

Given a semiring $(S,\ \oplus,\ \otimes, \overline{0},\ \overline{1})$ we can now define

$$\textbf{MinUnionLiftLeft}(S,\ \oplus,\ \otimes, \overline{0},\ \overline{1})$$

to be

$$(\{X \subseteq S \mid \min_{\leq} X = X\},\ \cup^{\min}_{\leq},\ \otimes^{\min}_{\leq},\ \{\},\ \{\overline{1}\})$$

where $\leq$ is $\leq^{L}_{\oplus}$, the left partial order derived from $\oplus$ (see Section II). This construction is an instance of a more general one called a *reduction* (see [23] and [4] page 14).

This is implemented in CAPP as

```
ebs_MinUnionLiftLeft :
  'a ebs -> ('a finite_set) ebs
```

There is a technical point to note. It is very difficult represent the set $\{X \subseteq S \mid \min_{\leq} X = X\}$ as in OCaml. However, our combinators construct equivalence relations (the equality) for every algebraic structure, and in this case we modify the equivalence relation as follows. If $=$ is an equivalence relation over finite subsets of S, then we can define a new equivalence relation $=^{\min}_{\leq}$ over finite subsets of S as

$$X =^{\min}_{\leq} Y \equiv \min_{\leq}(X) = \min_{\leq}(Y).$$

This explains the OCaml type of `ebs_MinLeftUnionLift`. The constructed algebra is over the same carrier type, but internally the equality has been modified.

Now we can define

```
let ebs_sigcomm_moo =
   ebs_MinLeftUnionLift
       (ebs_Product ebs_SP ebs_CP);;

let pa_sigcomm_moo =
    cast_ebs_to_pa ebs_sigcomm_moo;;
```

Next, we configure the adjacency matrix and use the OCaml function `pa_mtx_solver` to compute the associated Pareto Frontiers.

To fully capture the results of [19] we need to attach paths to each element of the Pareto Frontier. That is, would like a solution of the form $\{((5,\ 2),\ p_1),\ ((10,5),\ p_2)\}$ This is exactly what is accomplished by Manger [13] using a novel type of lexicographic product on sets, which we will call the Manger product. (We are currently in the process of implementing this in CAPP and have described this in [2].) However, Manger proves that in order to construct a path algebra using this construct the first component ($\text{sp} \times \text{cap}$ in this example) must be cancellative (see Figure 3). In this example, ${}_{\min}\times_{+}$ is not cancellative since $\min$ is not. This may help explain why [19] requires a new proof of convergence. However, it is interesting to note that if we started with a cancellative operation, such as ${}_{+}\times_{+}$, then we would end up with a path-algebra. In this case we could use a Link-State approach to routing rather than the (slower) path-vectoring method used in [19]. (A link-state approach does not require Dijkstra's algorithm be used as a local solver.)

C. Other path algebra combinators

Space does not permit presenting examples using all of the CAPP combinators designed for defining path algebras. Here we present just a brief overview.

We have discussed the CAPP representation of the basic mathematical structures **MinPlus** and **MaxMin**. CAPP also implements the following mathematical structures.

AndOr	$\equiv$	$(\{T,\ F\},\ \wedge,\ \vee)$
OrAnd	$\equiv$	$(\{T,\ F\},\ \vee,\ \wedge)$
MinMax	$\equiv$	$(\mathbb{N},\ \min,\ \max)$
MaxPlus	$\equiv$	$(\mathbb{N},\ \max,\ +)$
MinTimes	$\equiv$	$(\mathbb{N},\ \min,\ \times)$
UnionIntersect	$\equiv$	$(\mathcal{P}_{fin}(S),\ \cup,\ \cap)$
IntersectUnion	$\equiv$	$(\mathcal{P}_{fin}(S),\ \cap,\ \cup)$

Here $\mathcal{P}_{fin}(S)$ represents *finite* subsets of S.

UnionIntersect and **IntersectUnion** can be very useful. For example, suppose we give and arc $(i,\ j)$ a weight X_{ij}, where X_{ij} is a subset of possible arc attributes. Then, using **UnionIntersect**, we have $x \in \mathbf{A}^*(i,\ j)$ iff there is at least one path from i to j with x in every arc weight along the path. On the other hand, using **IntersectUnion**, we have $x \in \mathbf{A}^*(i,\ j)$

iff every path from i to j has at least one arc with weight containing x.

Of course, any of the basic structures can be combined using our CAPP implementations of the combinators already discussed: direct product, lexicographic product, adding a zero, adding a one, and the minimal set construction. Additional combinators have been implemented for these mathematical combinators (where names indicate the additive and multiplicative components constructed):

UnionLift$(S,\ \otimes)$
IntersectLift$(S,\ \otimes)$
MinUnionLiftRight$(S,\ \oplus,\ \otimes)$
MinLiftUnionLeft$(S,\ \oplus,\ \otimes)$
MinLiftUnionRight$(S,\ \oplus,\ \otimes)$

Let's consider a simple example using our implementation of **UnionLift**:

```
ebs_UnionLift :
  'a esg -> ('a finite_set) ebs
```

We can construct semigroups (type `esg`) with combinators corresponding to every semigroup construction we have discussed.

```
esg_Product :
  'a esg -> 'b esg -> ('a * 'b) esg
esg_LeftLexProduct :
  'a esg -> 'b esg -> ('a * 'b) esg
 ...        ...
```

We also have combinators for constructing structures of the form $(S,\ =)$, represented by the `eeqv` type.

```
eeqv_nat : nat eeqv
eeqv_Product :
  'a eeqv -> 'b eeqv -> ('a * 'b) eeqv
eeqv_Sum :
  'a esg -> 'b esg -> (('a, 'b) sum) eeqv
 ...        ...
```

These are useful for constructing semigroups such as

```
esg_Concat : 'a eeqv -> ('a list) esg
```

implementing concatenation of lists. We can then define a semiring of paths as

```
let ebs_Paths =
  ebs_UnionLift (esg_Concat (
    eeqv_Product
      eeqv_nat eeqv_nat));;
```

This can be used for computing sets of paths from one node to another. It is a semiring, but not a path algebra.

```
let ebs_SP_with_paths =
  ebs_AddZero infinity
   (ebs_LefLexProduct
      ebs_MinPlus ebs_Paths);;
```

This is not a path algebra either. The problem is that we could assign every arc the weight 0. Then the matrix method will not terminate because we can go around and around loops forever. The next section will break out of the semiring framework to solve this problem.

The combinator **MinLiftUnionLeft** is define like **MinUnionLiftLeft** except the additive and multiplicative components are exchanged. In [1] we show that this combinator can be used to define Martelli's semiring [15] for computing minimal cut sets. We generalize this construction to compute a network's shared risk groups.

IV. Combinator correctness

Coq is widely used theorem prover that implements a version of dependent type theory. A subset of the Coq's language is, except for minor syntactic differences, a subset of OCaml. We have written our combinators in this subset of Coq and proved them correct in a manner described below. We then use Coq's extraction mechanism to translate our Coq-combinators into OCaml-combinators.

Our implementation is comprised of two components; the Coq development and the OCaml library. Both components will be publicly available on GitHub (currently private for anonymity). The Coq component contains over 120K lines of proofs and definitions. The certified kernel of the OCaml component, about 30K lines, was extracted directly from the Coq development. We have written additional "wrappers" in OCaml to make it easier for users to access the kernel's functionality, to query the properties of constructed algebras, and to solve path problems.

We expect that a typical CAPP user will only download the OCaml component. The Coq development would be needed only when a user wishes to extend the combinator language.

For every algebraic class that we have implemented in Coq there are two versions — one containing dependent types and annotated with full Coq proofs, the other containing certificates. For example, for semigroups we have the type `A_sg S` containing the implementation of a semigroup together with the proofs associated with all properties while the corresponding type `sg S` contains the same implementation of the semigroup together with certificates for all properties.

Proofs in Coq are terms in its dependently typed language while certificates are simple data types representable as data types in OCaml. For example, a Coq proof that a semirgroup is not selective may be a very large expression in Coq, but it will be translated to a certificate of the form

```
Certity_Not_Selective (e1, e2)
```

in the OCaml language where expressions `e1` and `e2` are counter-examples to selectivity.

We then implement two versions of every combinator. For example, we have two versions of the lift combinator over semigroups:

```
A_sg_lift (S : Type) :
   A_sg S -> A_sg (finite_set A)
```

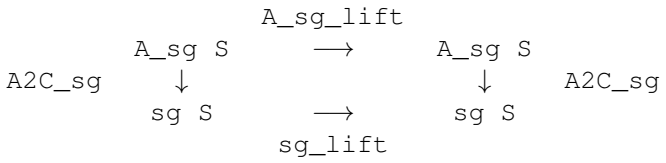

Fig. 8. A commutative diagram to establish the correctness of the semigroup lift combinator.

```
sg_lift (S : Type) :
   sg S -> sg (finite_set A)
```

The first version produces a dependently typed record containing full proofs of all of our semigroup properties, while the second version produces a record containing simple certificates.

In Coq we have implemented a translation function

```
A2C_sg (S : Type) : A_sg S -> sg A
```

that translates the component proof contained in the dependently type record into certificates. In order to prove the correctness of our OCaml implementation we prove that the diagram in Figure 8 commutes. That is, we prove the following theorem in Coq.

```
Theorem correct_sg_product (S : Type) :
forall s : A_sg S,
sg_lift (A2C_sg s) = A2C_sg (A_sg_lift s).
```

We have proved similar "commutative diagram theorems" for every combinator.

Finally, the Coq function `sg_product` is extracted to produce the OCaml function

```
sg_lift : 'a sg -> ('a finite_set) sg
```

where `'a sg` represents the type of records extracted from our implementation of `sg S` in Coq. Note that this step does assume the correctness of the Coq extraction mechanism and our use of it.

V. Related work

Our initial inspiration came from the metarouting paper [8]. Our CAPP library can be considered the first publicly available implementation of metarouting. Besides a robust implementation we have enhanced the conceptual framework of [8]. The combinators of [8] only enforced sufficient conditions, while our combinators enforce necessary and sufficient conditions. Thus, deriving rules for synthesizing P$\vee$ $\sim$P proofs requires considerably more effort. As we argue in Section III-A, this provides important feedback to users developing a new algebra.

The only other project we know of that attempts provide "correct-by-construction" path algebras is that directed by Robert Manger [14], [13]. Using the terminology we introduced in Section II-A, we would say that Manger's approach uses strict combinators. As we argued, this greatly restricts the number of such combinators that can be defined over the type of path algebras. Indeed, Manger's language essentially consists three combinators — the direct product, a minimal set construction similar to our combinator described in Section III-B, and Manger's lexicographic product on sets. This may be sufficient for Mangers goals of his project is focussed on exploring path algebras for multi-criteria optimization. However, as we noted in Section III-B, his lexicographic product on sets requires one argument with cancellativity, but the current implementation [14] does not check this holds. So, full correctness is left up to the user.

VI. Future work

Thus far we have focussed on building bi-semigroups of the form $(S,\ \oplus,\ \otimes)$. We are currently extending the CAPP implementation to go beyond semirings using structures that we call *semigroup transforms*.

A *left semigroup transform* is a structure of the form $(S,\ L,\ \oplus,\ \triangleright)$ where $(S,\ \oplus)$ is a commutative semigroup and $(S,\ L,\ \triangleright)$ is a *left transform*. A left transform is a structure of the form $(S,\ L,\ \triangleright)$ where $\triangleright$ is a function $\triangleright \in L \to S \to S$. The weight of an arc $(i,\ j)$ is now assigned an element of L, $w(i,\ j) \in L$. The left weight of a path is defined as

$$w_L((i,\ j)\ p) = w(i,\ j) \triangleright w_L(p)$$

where $\triangleright$ is written as an infix operator. (The left weight of the empty path is $\overline{1}$.) The elements of L can be thought of as an abstraction of *route maps* used in routing protocol configurations. A $\triangleright$-function applies route maps to routes.

Transforms greatly enhance the expressive power of routing algebras. For example, consider the following transform combinator. Suppose $(S,\ L_S,\ \triangleright_S)$ and $(T,\ L_T,\ \triangleright_T)$ are two transforms. Construct the transform $(S \times T,\ L_S \uplus L_T,\ \triangleright)$ where

$$\begin{aligned}(\text{Inl}\ \lambda_S) \triangleright (s,\ t) &= (\lambda_S \triangleright_S s,\ t)\\ (\text{Inr}\ \lambda_T) \triangleright (s,\ t) &= (s,\ \lambda_T \triangleright_T t)\end{aligned}$$

When used with the lexicographic product as the additive component, we can construct a semigroup transform that captures network partitions. This can be used to model the kinds of partitions seen in eBGP/iBGP or on OSPF areas.

We intend to connect the CAPP library with real routing code such as generalized code from routing stacks such as Quagga or BIRD [21], [20]. For this leap we take inspiration from the Verified Software Toolchain (VST) project [22] that allows low-level C code to be verified from a high-level Coq specification. An example of using the VST approach is [16] that develops verified C implementations of Dijkstra's, Kruskal's, and Prim's Algorithms from functional specification in Coq.

References

[1] Anonymous. Generalizing martelli's semiring to compute shared risk groups, 2023. To appear on the arXiv.

[2] Anonymous. On the composition of semigroup and semiring reductions, 2023. To appear on the arXiv.

[3] R.C. Backhouse and B.A. Carré. Regular algebra applied to path-finding problems. *J. Inst. Math. Appl.*, 15:161–18, 1975.

[4] John S Baras and George Theodorakopoulos. Path problems in networks. *Synthesis Lectures on Communication Networks*, 3(1):1–77, 2010.

[5] Bernard Carré. *Graphs and Networks*. Oxford University Press, 1979.

[6] Rosario G. Garroppo, Stefano Giordano, and Luca Tavanti. A survey on multi-constrained optimal path computation: Exact and approximate algorithms. *Computer Networks*, 54(17):3081–3107, 2010.

[7] Michel Gondran and Michel Minoux. *Graphs, Dioids and Semirings : New Models and Algorithms*. Springer, 2008.

[8] Timothy G Griffin and Joäo Luís Sobrinho. Metarouting. *Proc. ACM SIGCOMM*, 2005.

[9] Yuri Gurevich. Intuitionistic logic with strong negation. *Studia Logica*, 36(1/2), 1977.

[10] Alexander Gurney and Timothy G. Griffin. Lexicographic products in metarouting. In *Proc. Inter. Conf. on Network Protocols (ICNP)*, 2007.

[11] INRIA. The coq theorem prover, 2023.

[12] INRIA. The ocaml programming language, 2023.

[13] Robert Manger. An algebraic framework for mult-objective and robust variants of path problems. *Glasnik Matematicki*, 55(75):143 – 176, 2020.

[14] Robert Manger. Efficient algorithms for robust discrete optimization (rodiopt), 2020.

[15] Alberto Martelli. A gaussian elimination algorithm for the enumeration of cut sets in a graph. *Journal of the ACM*, 23(1):58–73, 1976.

[16] Anshuman Mohan, Wei Xiang Leow, and Aquinas Hobor. Functional correctness of c implementations of dijkstra's, kruskal's, and prim's algorithms. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 801–826. Springer International Publishing, 2021.

[17] David Nelson. Constructible falsity. *Journal of Symbolic Logic*, 14(1), 1949.

[18] Joao Luis Sobrinho. Algebra and algorithms for QoS path computation and hop-by-hop. *IEEE/ACM Transactions on Networking*, 10(4):541–550, August 2002.

[19] Joäo Luís Sobrinho and Miguel Alves Ferreira. Routing on multiple optimality criteria. *Proc. IEEE INFOCOM*, 2020.

[20] Bird Team. The bird internet routing daemon, 2023.

[21] Quagga Team. Quagga routing suite, 2023.

[22] VST Team. Verified software toolchain (vst), 2023.

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