Modeling Routing Algebras and the Stable Routing Problem in Cubical Type Theory

Yanjun Yang, 2020

Advisors: David Walker, Matthew Weaver

May 18, 2020

Abstract

Homotopy type theory (HoTT) is a new area of research that offers new techniques for reasoning formally about mathematics. Previous efforts in HoTT have led to new results in pure mathematics, such as the generalized Blakers-Massey theorem. In this work, we attempt to use cubical type theory, a variant of HoTT, to model computer routing networks as routing algebras in order to develop new proof strategies for reasoning about these routing algebras and the stable routing problem (SRP). We present a fully-formalized model of the routing algebra and the SRP in cubical type theory, and we provide proofs of several useful theorems about abstractions of SRPs that are formally verified in cubical type theory. Many of these proofs use techniques that are not available in traditional Martin-Löf type theory.

Contents

1 Introduction **5**

2 Problem Background and Related Work **5**

3 Definitions **6**

3.1 Cubical Type Theory 7

3.1.1 Basic Type Theory 7

3.1.2 Dimension and Path Types 8

3.1.3 Higher Inductive Types (HITs) 10

3.1.4 Type Equivalences and Univalence 11

3.2 Stable Routing Problem 13

3.2.1 Generalized Directed Graphs 13

3.2.2 Routing Algebra 14

3.2.3 Routing Algebra Semantics 15

3.2.4 Stable Routing Problem 17

3.2.5 Example: Computing Shortest Path with SRPs 18

3.3 Algebraic Relations between SRPs 20

3.3.1 Relations between Network States 20

3.3.2 Relations between SRP Solutions 22

3.3.3 Algebraic Properties of the Relations 24

4 Results **25**

4.1 Basic Results 25

4.2 Topology-Preserving Abstractions 30

4.3 Network Compression 33

5 Discussion **40**

5.1 Applications of the Cubical SRP Model 40

5.1.1 Anonymization Schemes 40
5.1.2 Step-wise Abstraction Algorithm 42
5.2 Using Cubical Type Theory 44
5.2.1 Advantages of Modeling in Cubical Type Theory 44
5.2.2 Disadvantages of Modeling in Cubical Type Theory 45

6 Future Work **46**

7 Acknowledgments **47**

1 Introduction

Homotopy type theory (HoTT) is a relatively new area of research that provides new techniques for reasoning formally about mathematics. As HoTT is closely related to homotopy theory and higher category theory, previous research into HoTT has focused more closely on these topics. Consequently, new advances in these fields, such as a generalized version of the Blakers-Massey Theorem [1], have been made using insights derived from HoTT. However, many implications of HoTT outside of pure mathematics, particularly in computer science, have not yet been explored extensively.

The goal of this project is to use cubical type theory, a variant of HoTT, to study a common topic in computer science: network routing. As we are becoming increasingly more reliant on the internet in our daily lives, it is becoming more and more important to ensure that our networks are configured to route our packets correctly. However, in many cases, verifying the correctness of configurations can be difficult, which motivates the formalization of routing networks in mathematics, where we can use formal methods, such as proofs, for verification. To study and analyze the routing of internet packets formally, we use a mathematical model called the routing algebra [2, 3], to model interactions among network nodes, and we will study the behavior of routing algebras in this work, as others have done before, through the stable routing problem [4, 5]. We will seek to model the routing algebra and the stable routing problem in cubical Agda [6, 7], a proof assistant that reasons using cubical techniques. Then, we use techniques from cubical type theory to develop new methods for proving properties about their behavior.

2 Problem Background and Related Work

Homotopy type theory evolved from the earlier Martin-Löf type theory, extending the definition of equality in the theory to allow for extensional equality of functions and univalence of types. The idea of extending the definition of equality began with Hofmann and Stre-

icher's groupoid interpretation of type theory [8], which sought to characterize an equality type with non-trivial witnesses. This was later generalized to the strict ∞-groupoid interpretation by Warren [9], and to homotopy-theoretic models by Awodey and Warren [10]. Ultimately, this led to the development of the full-fledged homotopy type theory by the Univalent Foundations Project [1].

Following the formalization of HoTT, there was much work done in finding a model for the theory that is fully constructive, which lead to the development of cubical type theories. Some of these models include the earliest model by Bezem, Coquand, and Huber (BCH) [11], a later model Cohen, Coquand, Huber, and Mörtberg (CCHM) [12], upon which cubical Agda, the proof assistant used in this study, was based [6], and more recently, a version known as Cartesian cubical type theory (ABCFHL) [13].

Meanwhile, in the study of routing networks, much effort has been made in building a mathematical model for their behaviors. In particular, Griffin and Sobrinho created an algebraic model, known as the routing algebra [2, 3], which models the network nodes and topology as a graph, and models the communication protocol among the nodes as an initialization vector, a merge function, and a family of transfer functions (one along each edge of the graph). This algebraic model is used in later studies, such as in the design of the Bonsai algorithm for network compression [4] and in the abstract interpretations model of routing networks [5], both of which focused on characterizing the long-term behavior of routing networks using the stable routing problem.

3 Definitions

In this section, we introduce and define the formalization used for our study of routing algebras and the stable routing problem in cubical type theory.

3.1 Cubical Type Theory

Cubical type theory is an implementation of homotopy type theory that is entirely constructive. Many versions of cubical type theory have been proposed [11–16]. As mentioned before, in this project, we use cubical Agda [6,7], which is based off of the earlier CCHM model [12]. Here, we introduce the ingredients from cubical type theory, as well as notation for these ingredients, that we use later to formalize routing algebras.

3.1.1 Basic Type Theory

In type theory, types represent collections of terms. We say that a term a has type A if and only if there exists a proof based on the axioms of the type theory that allow us to deduce that the type of a is indeed A. In turn we define the type A to be the smallest collection that contains all terms that can be proven to have type A. We denote that a has type A as $\mathsf{a} : \mathsf{A}$. Similarly, we denote that A is a type as $\mathsf{A} : \mathsf{Type}$ (i.e. the "type" of A is the type of all types[1]).

There are three basic methods for defining types in type theory:

1. Inductive types: An inductive type is a type defined together with a set of "constructors," which axiomatically define terms of the types. By the canonicity property, only terms that can (ultimately) be built from these constructors belong to the type. Examples of this include Bool, $\mathbb{N}$, and Fin, which encode Boolean values, natural numbers, and finite sets, respectively. We give sample definitions below:

$$
\begin{array}{lll}
\mathsf{Bool} : \mathsf{Type\ where} & \mathbb{N} : \mathsf{Type\ where} & \mathsf{Fin} : (\mathsf{n} : \mathbb{N}) \to \mathsf{Type\ where} \\
\quad \mathsf{true} : \mathsf{Bool} & \quad \mathsf{zero} : \mathbb{N} & \quad \mathsf{fzero} : \mathsf{Fin\ n} \\
\quad \mathsf{false} : \mathsf{Bool} & \quad \mathsf{suc} : \mathbb{N} \to \mathbb{N} & \quad \mathsf{fsuc} : \mathsf{Fin\ n} \to \mathsf{Fin\ (suc\ n)}
\end{array}
$$

[1] We refer to the collection of all types as a universe, and to avoid Russell's paradox, we index universe levels such that universes do not contain themselves, but instead are contained by a higher-level universe. Throughout this work, we suppress the universe level when mentioning Type for simplicity (this is often known as typical ambiguity).

For simplicity, we will denote terms of type $\mathbb{N}$ and Fin with numerals and interchange (i.e. cast) between these two types whenever it is safe to do so.

2. Function types: The function type is the type of functions mapping from one type to another. For any types A, B, we denote the type of (total) functions mapping from A to B as $\mathsf{A} \to \mathsf{B}$. Functions can either be defined directly via λ-abstractions, which bind the argument as a variable, or they can be defined inductively in the case when the argument type A is an inductive type (i.e. we can define a behavior for each constructor of A separately). A variant of this is the dependent function type, where the target type can depend on the argument. For argument type $\mathsf{A} : \mathsf{Type}$ and target type family $\mathsf{B} : \mathsf{A} \to \mathsf{Type}$ (i.e. B is a function from A to Type), we can derive the dependent function type $(\mathsf{a} : \mathsf{A}) \to \mathsf{B}\ \mathsf{a}$.

3. Product types: The product type is the type of ordered tuples. For any types A, B, we can define the product type $\mathsf{A} \times \mathsf{B}$. A variant of this is the dependent product type, where the type of the second component may depend on the first component. For type A and type family $\mathsf{B} : \mathsf{A} \to \mathsf{Type}$, we can derive the dependent product type $(\mathsf{a} : \mathsf{A}) \times \mathsf{B}\ \mathsf{a}$. Additionally, we have the operators fst and snd, which definitionally extract the first and second components of a (dependent) product type, respectively.

3.1.2 Dimension and Path Types

One relationship among terms and among types that we often reason about is equality. Proofs of equality in cubical type theory are captured by the path type, given as $\mathsf{Path} : (\mathsf{A} : \mathsf{Type}) \to \mathsf{A} \to \mathsf{A} \to \mathsf{Type}$. Throughout this work, we use $\mathsf{a} \equiv \mathsf{a'}$ as a shorthand for $\mathsf{Path}\ \_\ \mathsf{a}\ \mathsf{a'}$, thereby suppressing the type information. Terms of the path type are special λ-like expressions that bind a variable of the dimension type $\mathbb{I}$. The dimension type in CCHM consists of two special values, $\mathsf{i0}$ and $\mathsf{i1}$, and three constructors $\wedge : \mathbb{I} \to \mathbb{I} \to \mathbb{I}$, $\vee : \mathbb{I} \to \mathbb{I} \to \mathbb{I}$, and $\sim : \mathbb{I} \to \mathbb{I}$, representing maximum, minimum, and inverse, respectively, such that they form

a De Morgan algebra (i.e. a Boolean algebra without the law of excluded middle) [12]. We interpret the dimension type $\mathbb{I}$ as an interval, the special values i0 and i1 as endpoints of the interval, and paths as functions mapping out of the interval and into the type that the path resides in. In particular, if we have a type A, terms a a' : A, and a function e : $\mathbb{I} \to$ A such that e maps i0 to a and i1 to a', then we have that ⟨i⟩ (e i) : a ≡ a', where the notation ⟨i⟩ denotes the binding of the dimension variable i in the λ-like expression.

It is important to note that the dimension type is not defined inductively, so it is not possible to define paths by induction on the dimension type. Instead, paths are constructed in one of three ways (or any combination thereof):

1. Constant/axiomatic paths: The constant path, given by the construction above where we take e to be a constant function on $\mathbb{I}$, is always a valid path. Alternatively, we can define paths to exist axiomatically in certain types (see below for higher inductive types).

2. Action on paths: Given a base path p : a ≡ a' in type A, we can construct a new path that uses the base path as part of its function body. In particular, given a function f that maps from A to some type B, we can construct ⟨i⟩ (f (p i)) : f a ≡ f a'.

3. Composition of paths: We can compose paths when the right endpoint of one path lines up with the left endpoint of a second path. This can be done with the hcomp operator[2].

As defined above, we can derive the usual properties of equality: reflexivity, symmetry, transitivity, and congruence, for path types. We denote their witnesses by refl, sym, trans, and cong, respectively.

[2] hcomp is actually more general than what we have just described, but the details are not very relevant for this work.

3.1.3 Higher Inductive Types (HITs)

In cubical type theory, one can extend the idea of inductive types to higher inductive types (HITs) by allowing one to define path constructors on top of term constructors. Similar to how term constructors axiomatically define terms that exist in the given type, path constructors axiomatically define paths that exist in the type. Since equality in cubical type theory is witnessed by paths, adding path constructors allows us to define artificial equivalences between terms of a type. For example, we can give the following definition for the natural numbers modulo 2:

$$
\begin{aligned}
&\mathbb{N}/2\mathbb{N} : \mathsf{Type\ where} \\
&\quad \mathsf{nat} : \mathbb{N} \to \mathbb{N}/2\mathbb{N} \\
&\quad \mathsf{eq} : (\mathsf{n} : \mathbb{N}) \to \mathsf{nat\ n} \equiv \mathsf{nat\ (suc\ (suc\ n))}
\end{aligned}
$$

The term constructor nat stipulates that every term of $\mathbb{N}$ is a term of $\mathbb{N}/2\mathbb{N}$. The path constructor eq then adds a path between every natural number n and suc (suc n) (i.e. 2 + n). Therefore, in the type $\mathbb{N}/2\mathbb{N}$, we have the paths eq 0 : nat 0 ≡ nat 2, eq 1 : nat 1 ≡ nat 3, eq 2 : nat 2 ≡ nat 4, etc. Indeed, if we quotient over all paths in $\mathbb{N}/2\mathbb{N}$, we get the natural numbers modulo 2 in the mathematical sense.

Using higher induction, we can also define a type family AddPath that allows us to add a path between any two distinguished terms of any preexisting type. This will be useful later. We define AddPath as follows:

$$
\begin{aligned}
&\mathsf{AddPath} : (\mathsf{A} : \mathsf{Type})\ (\mathsf{x\ y} : \mathsf{A}) \to \mathsf{Type\ where} \\
&\quad \mathsf{in} : \mathsf{A} \to \mathsf{AddPath\ A\ x\ y} \\
&\quad \mathsf{same} : \mathsf{in\ x} \equiv \mathsf{in\ y}
\end{aligned}
$$

Analogous to regular induction, we can define a function that maps out of a HIT by

higher induction, where we supply a definition for each constructor of the HIT, including the path constructors. However, the definition supplied for the case of the path constructors cannot be arbitrary; for a function f mapping from A to B, if the path constructor defines a path of type $\mathsf{a} \equiv \mathsf{a'}$, where $\mathsf{a}\ \mathsf{a'} : \mathsf{A}$, then in the definition of the higher inductive case, we must give a path of type $\mathsf{f}\ \mathsf{a} \equiv \mathsf{f}\ \mathsf{a'}$. In other words, f must be defined to respect actions on path defined by path constructors.

3.1.4 Type Equivalences and Univalence

For all types A, B, an equivalence between A and B is a pair of functions $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ and $\mathsf{g} : \mathsf{B} \to \mathsf{A}$, and proofs $\mathsf{gf} : (\mathsf{x} : \mathsf{A}) \to \mathsf{g}\ (\mathsf{f}\ \mathsf{x}) \equiv \mathsf{x}$ and $\mathsf{fg} : (\mathsf{y} : \mathsf{B}) \to \mathsf{f}\ (\mathsf{g}\ \mathsf{y}) \equiv \mathsf{y}$, i.e., f and g are inverses of one another up to a path. We encode the equivalence type as the following type function:

$$\mathsf{Equiv} : \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type}$$

$$\mathsf{Equiv}\ \mathsf{A}\ \mathsf{B} = (\mathsf{f} : \mathsf{A} \to \mathsf{B}) \times (\mathsf{g} : \mathsf{B} \to \mathsf{A}) \times ((\mathsf{x} : \mathsf{A}) \to \mathsf{g}\ (\mathsf{f}\ \mathsf{x}) \equiv \mathsf{x}) \times ((\mathsf{y} : \mathsf{B}) \to \mathsf{f}\ (\mathsf{g}\ \mathsf{y}) \equiv \mathsf{y})$$

In HoTT, we have the concept of univalence, which is that two types which are equivalent should also be connected by a path. As cubical type theory is a constructive implementation of HoTT, it provides a constructive definition for univalence. Firstly, we have the ua constant, which has the following type:

$$\mathsf{ua} : (\mathsf{A}\ \mathsf{B} : \mathsf{Type}) \to \mathsf{Equiv}\ \mathsf{A}\ \mathsf{B} \to \mathsf{A} \equiv \mathsf{B}$$

The ua constant allows us to convert an equivalence between types A and B into a path connecting A and B in Type. Next, using the axioms of cubical type theory, it can be shown that for every equivalence between types A and B, there exists an associated path between A and B, and *vice versa*. In other words, equivalences between types *are* paths in Type. This

theorem has the following type encoding in cubical type theory:

$$\mathsf{ua\text{-}equiv} : (\mathsf{A}\ \mathsf{B} : \mathsf{Type}) \to \mathsf{Equiv}\ (\mathsf{Equiv}\ \mathsf{A}\ \mathsf{B})\ (\mathsf{A} \equiv \mathsf{B})$$

This also implies that we can turn any path between types into an equivalence, including paths that are built using action on paths and composition of paths. For an action on a path p, the associated equivalence uses the associated equivalence of p in the place where p is used in the definition, and uses the identity equivalence for everything else. For example, if we have $\mathsf{p} : \mathsf{A} \equiv \mathsf{B}$ for some $\mathsf{A}, \mathsf{B} : \mathsf{Type}$, then the new path $\langle \mathsf{i} \rangle\ (\mathsf{C} \to (\mathsf{p}\ \mathsf{i}))$ for some $\mathsf{C} : \mathsf{Type}$ has type $(\mathsf{C} \to \mathsf{A}) \equiv (\mathsf{C} \to \mathsf{B})$, and the path $\langle \mathsf{i} \rangle\ (\mathsf{C} \to (\mathsf{p}\ \mathsf{i}))$ is associated with an equivalence between the two function types, where the argument is left untouched, and the output varies according to the associated equivalence of the path p. For a composition of paths, the associated equivalence is simply the composition of the underlying equivalences.

Building off of the idea that paths in Type are equivalences, we can define the primitive operation $\mathsf{transport}$, which allows us to compute along a path in Type by applying the function f of the associated equivalence. In particular, we have:

$$\mathsf{transport} : (\mathsf{A} : \mathsf{Type})\ (\mathsf{B} : \mathsf{Type}) \to \mathsf{A} \equiv \mathsf{B} \to \mathsf{A} \to \mathsf{B}$$

together with the reduction rule:

$$\begin{aligned}&\mathsf{ua\text{-}beta} : (\mathsf{A} : \mathsf{Type})\ (\mathsf{B} : \mathsf{Type})\ ((\mathsf{f}, \mathsf{g}, \mathsf{gf}, \mathsf{fg}) : \mathsf{Equiv}\ \mathsf{A}\ \mathsf{B})\ (\mathsf{a} : \mathsf{A}) \to \\ &\quad \mathsf{transport}\ \mathsf{A}\ \mathsf{B}\ (\mathsf{ua}\ (\mathsf{f}, \mathsf{g}, \mathsf{gf}, \mathsf{fg}))\ \mathsf{a} \equiv \mathsf{f}\ \mathsf{a}\end{aligned}$$

Note that the reduction rule $\mathsf{ua\text{-}beta}$ uses path equality, which implies that computation using $\mathsf{transport}$ is only equal to applying the underlying function up to a path. While this distinction is important for the theory of cubical types, it is not very relevant for the applications in this work.

Since transport can compute along any path, it can, in particular, compute along a path between function types. The transported function will simply use the original function as a black box, and transports the arguments and output appropriately in order to conform to the type definition.

3.2 Stable Routing Problem

The stable routing problem is the main object of study in this work. We will develop the definition of the SRP in several steps, starting with some basic ingredients.

3.2.1 Generalized Directed Graphs

We model the topology of a network using graphs. Each router is modeled by a single vertex in the graph, and a directed edge between two vertices represents a directed message flow between two routers. Bidirectional flow, which is the norm for routers, is represented as a pair of directed edges. As we are working in a type theory, we must encode graphs as types, which we do as follows:

Definition 3.2.1 (Generalized Directed Graph)**.** *A generalized directed graph is a function from two terms of an arbitrary type (called the generalized vertex type) to the type of Booleans denoting the presence or absence of an edge. We encode it as follows:*

$$\mathit{Graph} : (V : \mathit{Type}) \to \mathit{Type}$$

$$\mathit{Graph}\ V = V \to V \to \mathit{Bool}$$

As a convention, we will take the first argument to represent the source, the second argument to represent the target, and the value of true : Bool to indicate the existence of a directed edge from the source to the target. In particular, to encode a finite graph, we can simply parameterize the Graph type with the finite set type Fin.

3.2.2 Routing Algebra

Next, we will define the routing algebra, which is the algebraic structure used to model computer networks protocols. The approach used in this project is based on previous works on the topic [2–5].

Definition 3.2.2 (Routing Algebra)**.** *The routing algebra is a tuple* $(V, A, G, I, \oplus, f)$*, where the components are defined as follows:*

V – *A set of vertices*

A – *A set of attributes*

G – *A directed graph with vertex set* V

$I : V \to A$ – *An initialization vector that assigns each vertex an initial attribute*

$\oplus : A \to A \to A$ – *A merge function that merges two attributes. It is assumed to be commutative and associative.*

$f : E \to A \to A$ – *A transfer function that transforms an attribute along an edge of the graph* G*, where* E *denotes the edge set of* G*.*

The topology of the network is modeled by the graph G, as mentioned before. The attribute set A models the set of all possible messages that may be sent in the network. The initialization vector I models the information that each router begins with. The merge function $\oplus$ models the way in which network nodes combine attribute information sent by its neighbors. Finally, the transfer function f models the way in which a network node modifies its current attribute before sending it to an adjacent node.

In the type theory, we represent the routing algebra as follows:

Definition 3.2.3 (Routing Algebra Type)**.** *We encode the routing algebra as the following*

product type:

$$RA : (V : Type)\ (A : Type) \to Type$$

$$RA\ V\ A = (Graph\ V) \times (V \to A) \times (A \to A \to A) \times (V \to V \to A \to A)$$

There are two peculiarities in the definition above. Firstly, the merge function is defined to be an arbitrary function of type $\mathsf{A} \to \mathsf{A} \to \mathsf{A}$, as opposed to one that is commutative and associative. This is done because explicitly enforcing commutativity and associativity directly is messy and inelegant. However, while this technically broadens the definition of the routing algebra, we do not lose proving power because whenever such properties are needed to complete a proof, we can simply assume them as premises. Secondly, the transfer function is defined above for all pairs of vertices, instead of only for edges. Similarly, this is done to simplify the definition of the transfer function, as it is messy to enforce the existence of an edge[3].

3.2.3 Routing Algebra Semantics

Now that we have defined the routing algebra, the next step is to define the way we will model the behavior of the routing network. We begin with the network state.

Definition 3.2.4 (Network State)**.** *A network state of a routing algebra is a function that assigns an attribute to each vertex of the routing algebra. The type definition is as follows:*

$$RA\text{-}State : (V : Type)\ (A : Type) \to RA\ V\ A \to Type$$

$$RA\text{-}State\ V\ A\ ra = V \to A$$

[3]This could be done by adding names to the first two arguments, say u and v, and adding in an additional argument of the type $\mathsf{G\ u\ v} \equiv \mathsf{true}$, where G is the name given to the first component of the product–that is, we require G to evaluate to $\mathsf{true} : \mathsf{Bool}$ for arguments u and v. However, since equalities are non-trivial and proof-relevant in cubical type theory, this leads to additional complications later on. For example, it would not be trivially true (but still technically provable) that the transfer function would behave the same under two different proofs of equality of $\mathsf{G\ u\ v} \equiv \mathsf{true}$.

A term of the RA-State type is used to represent the state of a routing network at a given moment in time. We include the routing algebra as part of the type definition of the RA-State type so that two routing algebras with the same vertex and attribute types will still have different state types. Next, we define semantics, which are what describe the evolution process of network states.

Definition 3.2.5 (Network Semantics)**.** *A network semantics is a function that computes the next state of a routing algebra given its current state. The type definition is as follows:*

$$\mathit{RA\text{-}Semantics} : (V : \mathit{Type})\ (A : \mathit{Type}) \to \mathit{RA}\ V\ A \to \mathit{Type}$$

$$\mathit{RA\text{-}Semantics}\ V\ A\ \mathit{ra} = \mathit{RA\text{-}State}\ V\ A\ \mathit{ra} \to \mathit{RA\text{-}State}\ V\ A\ \mathit{ra}$$

The above definition places very few restrictions on how the semantics of a network could be defined. The reason for defining the semantics in this way is that it is difficult to define a consistent semantics for all routing, particularly those with non-trivial vertex types. However, it is possible to define a consistent semantics if we fix the vertex type to be from the Fin type family. This leads us to the following definition, which is consistent with the typical notion of semantics for routing algebras in previous works [2–5]:

Definition 3.2.6 (Standard Semantics)**.** *Assume that we have* $n : \mathbb{N}, A : \mathit{Type}, (G, I, \oplus, f) : \mathit{RA}\ (\mathit{Fin}\ (\mathit{suc}\ n))\ A$*. Let* $s : \mathit{RA\text{-}State}\ (\mathit{Fin}\ (\mathit{suc}\ n))\ A\ (G, I, \oplus, f)$ *be the current state of the network, and let* s' *denote a vector of attributes indexed by the vertices. Then, the standard semantics is the function that maps* $s\ v$ *to* $s'[v]$*, where* $s'[v]$ *is computed as follows:*

Algorithm 1 Calculating the next state under standard semantics.

```
for v : Fin (suc n) do
    s'[v] ← I v
    for u : Fin (suc n) do
        if G u v = true then
            s'[v] ← s'[v] ⊕ f u v (s u).
        else
            s'[v] remains unchanged.
        end if
    end for
end for
```

In the type theory, we define it with the following signature:

$$\mathit{std\text{-}sem} : (n : \mathbb{N})\ (A : \mathit{Type})\ (\mathit{ra} : \mathit{RA}\ (\mathit{Fin}\ (\mathit{suc}\ n))\ A) \to \mathit{RA\text{-}Semantics}\ (\mathit{Fin}\ (\mathit{suc}\ n))\ A\ \mathit{ra}$$

The inner loop is always computed starting with vertex 0 and counting up to vertex n. While we impose a fixed order in which the computation of the next state is carried out, the order should not matter if we assume that the merge function is commutative and associative.

3.2.4 Stable Routing Problem

We are now able define the stable routing problem. Given a routing algebra and an associated semantics, the stable routing problem asks to find the network states of the routing algebra that are stable under the given semantics (i.e. the fixed points of the semantics). We define and encode these concepts as follows:

Definition 3.2.7 (SRP Type)**.** *The stable routing problem has the following dependent product type:*

$$\mathit{SRP} : \mathit{Type} \to \mathit{Type} \to \mathit{Type}$$

$$\mathit{SRP}\ V\ A = (\mathit{ra} : \mathit{RA}\ V\ A) \times (\mathit{RA\text{-}Semantics}\ V\ A\ \mathit{ra})$$

Additionally, we refer to SRPs defined with the standard semantics as the second component as ***standard SRPs****.*

Definition 3.2.8 (Stable State)**.** *Let* V, A *be types and* (ra, next-state) : SRP V A *be an instance of a stable routing problem. A state* s : RA-State V A ra *is stable if* s ≡ next-state s*. The type definition is as follows:*

```
Stable : (V : Type) (A : Type) (srp : SRP V A) → RA-State V A (fst srp) → Type
Stable V A (ra, next-state) s = s ≡ next-state s
```

Definition 3.2.9 (SRP Solution)**.** *Let* V, A *be types and* (ra, next-state) : SRP V A *be an instance of a stable routing problem. A solution to* (ra, next-state) *is a state* s : RA-State V A ra*, together with a proof that it is stable. The type definition is as follows:*

```
SRP-Solution : (V : Type) (A : Type) (srp : SRP V A) → Type
SRP-Solution V A (ra, next-state) = (s : RA-State V A ra) × (Stable V A (ra, next-state) s)
```

3.2.5 Example: Computing Shortest Path with SRPs

To see how the stable routing problem works, consider the example in Figure 1. First, let us define the routing algebra. Let $\mathsf{V} = \mathsf{Fin}\ 6$ and $\mathsf{A} = \mathbb{N}_\infty$[4]. Let G be the graph in Figure 1, let I be the initial state on the left hand side of Figure 1, let $\oplus = \min$, and let f be as indicated on the edges of the graph in Figure 1. In the first round, the top vertex sends its attribute $\mathsf{0}$ to its two adjacent vertices. On the left edge, $\mathsf{2}$ is added to $\mathsf{0}$, while on the middle edge, $\mathsf{8}$ is added to $\mathsf{0}$, and on the right edge, $\mathsf{3}$ is added to 0. All other edges send ∞, which equals ∞ when any number is added. The nodes receiving $\mathsf{2}$, $\mathsf{8}$, and $\mathsf{3}$ merge the attribute with their initial value ∞, and yield $\mathsf{2}$, $\mathsf{8}$, and $\mathsf{3}$, respectively. All other nodes merge ∞ with ∞ to yield ∞. In the second round, every other node remains the same, but the fifth node receives

[4]This is natural numbers augmented with infinity.

$2 + 6 = 8$ from its left neighbor, $8 + 9 = 17$ from its middle neighbor, and $3 + 1 = 4$ from its right neighbor. It merges these three along with its initial value ∞ to get 4 (as 4 is the smallest among all four). In the third round, all nodes remain the same except the sixth node, which receives 7 from the fifth node and yields 7 after merging. At this point, one can confirm that the next round yields the same network state, implying that network state is stable and that we have arrived at the (unique) solution of this SRP (on the right hand side of Figure 1). In essence, we have executed the distributed Bellman-Ford algorithm using the SRP to compute the shortest path distances to the initial node.

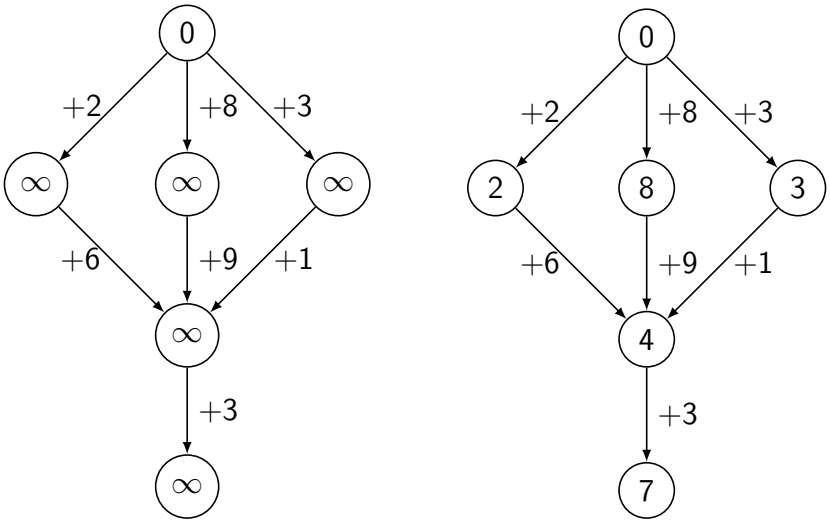


Figure 1: SRP Initial and Final State

In general, finding solutions to the stable routing problem can be difficult. One way in which standard SRPs are typically solved is to first reduce the SRP instance to an instance of SAT[5] (boolean satisfiability), and then use a preexisting SAT-solver.

[5]Deciding whether standard SRPs have solutions is in **NP**, as verifying whether a given state is a solution is possible in polynomial time by simply computing the next state using the standard semantics and checking that it equals the given state.

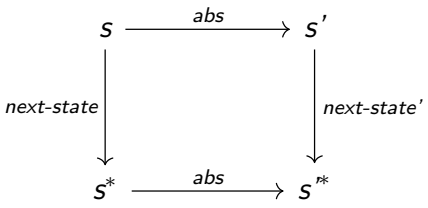
3.3 Algebraic Relations between SRPs

For this project, we will focus on studying algebraic properties of and relations between different SRPs. There are two kinds of relations that we study in this project: relations between network states and relations between SRP solutions. We define these notions below:

3.3.1 Relations between Network States

The types of relations that we focus on in this project are abstraction relations. Intuitively, an abstraction relation is one that seeks to simplify the SRP by eliding parts that are "unimportant" while preserving relevant structures and behaviors. In the case of SRPs, the "relevant behavior" that we seek to preserve is the semantics. Below, we offer three different notions of abstraction between SRPs in ascending order of strength:

Definition 3.3.1 (Quasi-abstraction)**.** *Let* U, V, A, B *be types and assume we have* (ra, next-state) : SRP U A *and* (ra', next-state') : SRP V B*. We say that* (ra', next-state') *is a quasi-abstraction of* (ra, next-state) *if there exists a function* abs : RA-State U A ra → RA-State V B ra' *such that for all states* s : RA-State U A ra*,* abs (next-state s) ≡ next-state' (abs s)*. Equivalently, we say that the following diagram commutes (up to a path):*



The type definition is as follows:

$$
\begin{aligned}
&\mathit{Quasi\text{-}Abstraction} : (U\ V\ A\ B : \mathit{Type}) \to \mathit{SRP}\ U\ A \to \mathit{SRP}\ V\ B \to \mathit{Type} \\
&\mathit{Quasi\text{-}Abstraction}\ U\ V\ A\ B\ (\mathit{ra}, \mathit{next\text{-}state})\ (\mathit{ra'}, \mathit{next\text{-}state'}) = \\
&\quad (\mathit{abs} : \mathit{RA\text{-}State}\ U\ A\ \mathit{ra} \to \mathit{RA\text{-}State}\ V\ B\ \mathit{ra'}) \\
&\quad \times ((s : \mathit{RA\text{-}State}\ U\ A\ \mathit{ra}) \to \mathit{abs}\ (\mathit{next\text{-}state}\ s) \equiv \mathit{next\text{-}state'}\ (\mathit{abs}\ s))
\end{aligned}
$$

Definition 3.3.2 (Abstraction)**.** *Let* U, V, A, B *be types and assume we have* (ra, next-state) : SRP U A *and* (ra', next-state') : SRP V B*. We say that* (ra', next-state') *is an abstraction of* (ra, next-state) *if there exists a* ***surjective***[6] *function* abs : RA-State U A ra → RA-State V B ra' *such that for all states* s : RA-State U A ra, abs (next-state s) ≡ next-state' (abs s)*. The type definition is as follows:*

$$
\begin{aligned}
&\mathit{Abstraction} : (U\ V\ A\ B : \mathit{Type}) \to \mathit{SRP}\ U\ A \to \mathit{SRP}\ V\ B \to \mathit{Type} \\
&\mathit{Abstraction}\ U\ V\ A\ B\ (\mathit{ra}, \mathit{next\text{-}state})\ (\mathit{ra}', \mathit{next\text{-}state}') = \\
&\quad (\mathit{abs} : \mathit{RA\text{-}State}\ U\ A\ \mathit{ra} \to \mathit{RA\text{-}State}\ V\ B\ \mathit{ra}') \\
&\quad \times (\mathit{abs}^{-1} : \mathit{Surjection}\ (\mathit{RA\text{-}State}\ U\ A\ \mathit{ra})\ (\mathit{RA\text{-}State}\ V\ B\ \mathit{ra}')\ \mathit{abs}) \\
&\quad \times ((s : \mathit{RA\text{-}State}\ U\ A\ \mathit{ra}) \to \mathit{abs}\ (\mathit{next\text{-}state}\ s) \equiv \mathit{next\text{-}state}'\ (\mathit{abs}\ s))
\end{aligned}
$$

Definition 3.3.3 (Behavioral Equivalence)**.** *Let* U, V, A, B *be types and assume we have* (ra, next-state) : SRP U A *and* (ra', next-state') : SRP V B*. We say that* (ra, next-state) *is behaviorally equivalent to* (ra', next-state') *if there exists an equivalence* (abs, rev-abs, rev-abs-abs, abs-rev-abs) : Equiv (RA-State U A ra) (RA-State V B ra') *such that for all states* s : RA-State U A ra, abs (next-state s) ≡ next-state' (abs s)*. The type definition is as follows:*

$$
\begin{aligned}
&\mathit{Equivalence} : (U\ V\ A\ B : \mathit{Type}) \to \mathit{SRP}\ U\ A \to \mathit{SRP}\ V\ B \to \mathit{Type} \\
&\mathit{Equivalence}\ U\ V\ A\ B\ (\mathit{ra}, \mathit{next\text{-}state})\ (\mathit{ra}', \mathit{next\text{-}state}') = \\
&\quad ((\mathit{abs}, \mathit{rev\text{-}abs}, \mathit{rev\text{-}abs\text{-}abs}, \mathit{abs\text{-}rev\text{-}abs}) : \mathit{Equiv}\ (\mathit{RA\text{-}State}\ U\ A\ \mathit{ra})\ (\mathit{RA\text{-}State}\ V\ B\ \mathit{ra}')) \\
&\quad \times ((s : \mathit{RA\text{-}State}\ U\ A\ \mathit{ra}) \to \mathit{abs}\ (\mathit{next\text{-}state}\ s) \equiv \mathit{next\text{-}state}'\ (\mathit{abs}\ s))
\end{aligned}
$$

All three definitions take the form of dependent products. Throughout this work, we will refer to the first component of these type as the abstraction function, and the last component

[6] Let A, B be types. For a surjective function $f : A \to B$, the proof of surjectivity, with type Surjection A B f, is a pseudoinverse $f^{-1} : (b : B) \to \mathsf{hfiber}\ A\ B\ f\ b$. The hfiber A B f b type is defined as a dependent pair, where the first component is a term of type A, and the second component is a proof that $f\ a \equiv b$. As such, $\lambda x.\ \mathsf{fst}\ (f^{-1} x)$ is a right-inverse of f, and $\lambda x.\ \mathsf{snd}\ (f^{-1} x)$ is a proof that $\lambda x.\ \mathsf{fst}\ (f^{-1} x)$ is indeed a right-inverse.

as the proof of semantic equivalence. Since bijective maps are surjective, and both satisfy the minimum requirements of a function, we get that Behavioral Equivalence $\implies$ Abstraction $\implies$ Quasi-abstraction for free.

We include the definition of quasi-abstractions for completeness. In general, we would like to avoid working with quasi-abstractions because the definition captures undesirable behaviors. For example, a consequence of the definition is that any SRP with a solution is a quasi-abstraction of any other SRP, where the function that maps all states of the latter SRP to the distinguished solution of the former SRP serves as the necessary abstraction function. Since we map every state to a solution, semantic equivalence is trivially achieved. However, this quasi-abstraction is unhelpful in capturing the behavior of the SRP being abstracted. Therefore, we will attempt to work with relations that are at least full abstractions whenever possible, and we will typically only use quasi-abstractions to highlight whenever fewer assumptions than those needed for full abstractions suffice in a given proof.

3.3.2 Relations between SRP Solutions

Once we have established an abstraction relation between SRPs, an additional feature that we would like to characterize would be how the solutions of the two SRPs relate to each other. To characterize solutions, we define three different notions below:

Definition 3.3.4 (Soundness)**.** *Let U, V, A, B be types and assume we have (ra, next-state) : SRP U A and (ra', next-state') : SRP V B. Further, assume that we have (abs, sem-eq) : Quasi-abstraction U V A B (ra, next-state) (ra', next-state'). We say that (ra', next-state') is a sound abstraction of (ra, next-state) if for all solutions (s, eq) : SRP-Solution U A (ra, next-state), we have that abs s is stable with respect to next-state'. In other words, the abstraction function abs maps solutions of (ra, next-state) to solutions of (ra', next-state').*

Showing that an abstraction relation is sound allows us to reason about the solutions of a concrete SRP by examining and/or proving properties of the solutions of an abstract SRP.

The following notions require that we have at least a full abstraction, but allow for more interesting scenarios.

Definition 3.3.5 (Similarity)**.** *Let U, V, A, B be types and assume we have (ra, next-state) : SRP U A and (ra', next-state') : SRP V B. Further, assume that we have (abs, abs^{-1}, sem-eq) : Abstraction U V A B (ra, next-state) (ra', next-state'). We say that (ra', next-state') is similar to (ra, next-state) if for all solutions (s, eq) : SRP-Solution U A (ra, next-state), we have that abs s is stable with respect to next-state', and for all solutions (s', eq') : SRP-Solution V B (ra', next-state'), we have that fst (abs^{-1}s) is stable with respect to next-state. In other words, the abstraction function abs is also surjective when restricted to solutions.*

Showing that an abstract SRP simulates a concrete SRP implies that there are no "extraneous" states or solutions that were created as a result of the abstraction. This gives us a notion of "losslessness" of abstraction. The next definition expands on this even more.

Definition 3.3.6 (Bisimilarity)**.** *Let U, V, A, B be types and assume we have (ra, next-state) : SRP U A and (ra', next-state') : SRP V B. Further, assume that we have (abs, abs^{-1}, sem-eq) : Abstraction U V A B (ra, next-state) (ra', next-state'). We say that (ra', next-state') is bisimilar to (ra, next-state) if for all solutions (s, eq) : SRP-Solution U A (ra, next-state), we have that abs s is stable with respect to next-state', and for all solutions (s', eq') : SRP-Solution V B (ra', next-state'), we have that fst (abs^{-1}s) is stable with respect to next-state, and furthermore, when restricted to solutions, the function abs is bijective*[7]*.*

Bisimilarity is the strongest notion of a lossless abstraction (among those that we will be exploring) that is short of a direct equivalence relation. If one is interested in the solutions of SRPs, then bisimilarity represents perfect losslessness of relevant information, while still allowing for potential simplification of the overall problem. Similar to the network state relations, one can also easily see that by definition, Bisimilarity $\implies$ Similarity $\implies$ Soundness.

[7]As a technicality, this is slightly weaker than saying that abs induces a bijection between the solution types SRP-Solution U A (ra, next-state) and SRP-Solution V B (ra', next-state') because we do not require mapping the proofs of stability, only the states themselves.

3.3.3 Algebraic Properties of the Relations

One can easily verify that all of the above relations are reflexive and transitive. Furthermore, behavioral equivalence is also symmetric. Verifying this property property is a little more involved, so we give the proof below:

Lemma 3.3.7 (Behavioral Equivalence Symmetry)**.** *Let U, V, A, B be types and assume we have (ra, next-state) : SRP U A and (ra', next-state') : SRP V B. Then, Equivalence U V A B (ra, next-state) (ra', next-state') implies Equivalence V U B A (ra', next-state') (ra, next-state).*

Proof. Assume we have ((abs, rev-abs, rev-abs-abs, abs-rev-abs), sem-eq) : Equivalence U V A B (ra, next-state) (ra', next-state'). Note that (rev-abs, abs, abs-rev-abs, rev-abs-abs) : Equiv (RA-State V B ra') (RA-State U A ra). We prove reverse semantic equivalence, with type (s' : RA-State V B ra') → rev-abs (next-state' s') ≡ next-state (rev-abs s') as follows: We use abs-rev-abs : (s : RA-State U A ra) → abs (rev-abs s) ≡ s to get rev-abs (next-state' s') ≡ rev-abs (next-state' (abs (rev-abs s'))). By sem-eq, we get rev-abs (next-state' (rev-abs (abs s'))) ≡ rev-abs (abs (next-state (rev-abs s'))). Finally, by rev-abs-abs : (s' : RA-State V B ra') → rev-abs (abs s') ≡ s', we get rev-abs (abs (next-state (rev-abs s'))) ≡ next-state (rev-abs s'). Composing the three paths gives us the desired result. □

Additionally, one can verify that, when restricted to finite non-empty routing algebras (i.e. routing algebras with a finite but non-zero number of possible states), the full abstraction relation for SRPs is anti-symmetric under the behavioral equivalence, and is thus a partial order. This follows from the fact that surjective functions are automatically bijective when mapping between finite types of the same size. Furthermore, we know that the equivalence class (where behaviorally-equivalent SRPs are identified) that acts as the the top element of this partial order is the one containing all SRPs with a single possible state[8].

[8]If there exists exactly one possible network state, then there exists exactly one semantics (the identity semantics), and thus one SRP, for the network, where the distinguished state is the sole solution. Therefore, for any other SRP, the function mapping all states to the distinguished state is the unique abstraction function, and furthermore, the abstraction is both trivially surjective and trivially exhibits semantic equivalence. Incidentally, it is also trivially similar.

4 Results

Using the proposed model above, we have been able to prove several theorems about the relations between stable routing problems in cubical type theory. We present the results, as well as proof sketches below:

4.1 Basic Results

First, we present some basic results about what relationships between states imply about relationships between solutions.

Theorem 4.1.1. *Let U, V, A, B be types and assume we have (ra, next-state) : SRP U A and (ra', next-state') : SRP V B. Further, assume that we have (abs, sem-eq) : Quasi-abstraction U V A B (ra, next-state) (ra', next-state'). Then, (ra', next-state') is a sound abstraction of (ra, next-state). In other words, all quasi-abstractions are sound.*

Proof. Assume we have (s, eq) : SRP-Solution U A ra. By congruence on eq, we have that abs s ≡ abs (next-state s). By sem-eq, we have that abs (next-state s) ≡ next-state' (abs s). Composing the two gives the desired result. □

Theorem 4.1.2. *Let U, V, A, B be types and assume we have (ra, next-state) : SRP U A and (ra', next-state') : SRP V B. Further, assume that we have ((abs, rev-abs, rev-abs-abs, abs-rev-abs), sem-eq) : Equivalence U V A B (ra, next-state) (ra', next-state'). Then, (ra', next-state') is bisimilar to (ra, next-state). In other words, all behaviorally equivalent SRPs are bisimilar to one another.*

Proof. Note that an equivalence is also a surjection: The pseudoinverse of abs can be defined with λx. (rev-abs x, abs-rev-abs x) : Surjection (RA-State U A ra) (RA-State V B ra') abs. Therefore, the premises are satisfied. To prove the first half of bisimilarity, simply use the above theorem: We know that equivalences are trivially abstractions, and that equivalences are symmetric, so simply applying the previous theorem on the forward and reverse

equivalences gives us the first half of bisimilarity. The second half is directly implied by rev-abs-abs and abs-rev-abs, i.e. since abs is a bijection for all states, it is trivially a bijection when restricted to solutions, since solutions are states. □

Next, we will take a look at ways of deriving equivalent SRPs from paths between types, which, by univalence, we know are simply equivalences between types. First, we explore paths between attribute types:

Theorem 4.1.3. *Let V, A, B be types and assume we have (ra, next-state) : SRP V A and p : A ≡ B. Then:*

1. *We can construct p^* : SRP V A ≡ SRP V B.*
2. *(ra, next-state) is behaviorally equivalent to transport (SRP V A) (SRP V B) p^* (ra, next-state).*
3. *If V is Fin (suc n) for some n : ℕ and next-state ≡ std-sem n A ra, then snd (transport (SRP V A) (SRP V B) p^* (ra, next-state)) ≡ std-sem n B (fst (transport (SRP V A) (SRP V B) p^* (ra, next-state))), i.e. if the original SRP is standard, then the transported network is also standard.*

Proof.

1. We can define p^* as follows:

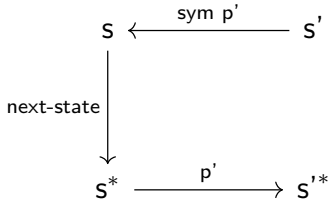
$$p^* : \mathsf{SRP\ V\ A} \equiv \mathsf{SRP\ V\ B}$$
$$p^* = \langle i \rangle\ (\mathsf{SRP\ V\ (p\ i)})$$

2. Consider a path between states defined as follows:

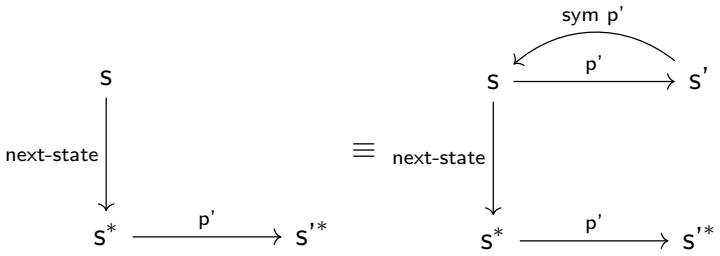
$$\mathsf{p'} : (\mathsf{V} \to \mathsf{A}) \equiv (\mathsf{V} \to \mathsf{B})$$

$$\mathsf{p'} = \langle \mathsf{i} \rangle\ (\mathsf{V} \to (\mathsf{p}\ \mathsf{i}))$$

By univalence, we can convert p' to an equivalence. This will serve as the first component of the behavioral equivalence proof. The second component, semantic equivalence, directly follows, since the RA semantics and the network states are both transported along paths that are derived from the path p. The transported semantics snd (transport (SRP V A) (SRP V B) p^* (ra, next-state)) essentially carries out the following operation:



Therefore, semantic equivalence in this case is asking whether the the following operations are equal (up to a path):



which is trivially true.

3. The only difference between snd (transport (SRP V A) (SRP V B) p^* (ra, std-sem n A ra)) and std-sem n B (fst (transport (SRP V A) (SRP V B) p^* (ra, std-sem n A ra))) is that the former transports the whole expression at once, and the latter transports each argument to the function std-sem separately. However, since all of these transports are

along the same base path p, they are path equivalent. Note that this is irrespective of how std-sem actually computes.

□

Next, we will explore how paths between vertex types induce equivalent SRPs, where we will see some of the limitations of cubical type theory.

Theorem 4.1.4. *Let U, V, A be types and assume we have (ra, next-state) : SRP U A and p : U ≡ V. Then:*

1. *We can construct p^* : SRP U A ≡ SRP V A.*
2. *(ra, next-state) is behaviorally equivalent to transport (SRP U A) (SRP V A) p^*(ra, next-state).*
3. *If U and V are Fin (suc n) for some n : ℕ, p : Fin (suc n) ≡ Fin (suc n) is a swap of two consecutive finite set elements*[9]*, and next-state ≡ std-sem n A ra, then snd (transport (SRP U A) (SRP V A) p^* (ra, next-state)) ≡ std-sem n B (fst (transport (SRP U A) (SRP V A) p^* (ra, next-state))).*
4. *If U and V are Fin (suc n) for some n : ℕ, p : Fin (suc n) ≡ Fin (suc n) is **any path**, and next-state ≡ std-sem n A ra, then snd (transport (SRP U A) (SRP V A) p^* (ra, next-state)) ≡ std-sem n B (fst (transport (SRP U A) (SRP V A) p^* (ra, next-state))).*

Proof.

1. Analogous to proof of Theorem 4.1.3 case 1.
2. Analogous to proof of Theorem 4.1.3 case 2.

[9]The canonical forms of the terms of the inductive type Fin n are the natural numbers from 0 to n - 1. By consecutive, we refer to a swap of k and k + 1, for k between 0 and n - 2, inclusive.

3. This case is not analogous to Theorem 4.1.3 case 3. This is because the term std-sem is not defined uniformly for all finite set vertex types in the same way that it is defined uniformly for all attribute types. Assume that p swaps k and k + 1, for some k ≤ n. Since the transported semantics snd (transport (SRP U A) (SRP V A) p^* (ra, std-sem n A ra)) uses the original semantics std-sem n A ra to compute, it will attempt to gather information from vertex k + 1 of the transported RA fst (transport (SRP U A) (SRP V A) p^* (ra, std-sem n A ra)) before vertex k in the "inner loop" of the computation[10]. Therefore, the transported semantics do not compute in the same way as the standard semantics std-sem n B (fst (transport (SRP U A) (SRP V A) p^* (ra, std-sem n A ra))). Nevertheless, we can prove that these two functions are equal extensionally (i.e. when given the same input, they compute equal outputs, despite the order in which they compute). We prove this in three steps:

   (1) For all inputs, the computations done after the first k - 1 iterations of the "inner loop" are equal: We prove this by induction. Let v be any vertex. In the beginning, s′[v] gets the same initial value in the initialization vector in both computations. All vertices less than k are unaffected by the swap, so the computation in each iteration is equal. By induction, all iterations up to k - 1 compute equally.

   (2) For all inputs, the computations done after the first k + 1 iterations of the "inner loop" are equal: Let v be any vertex. Let G be the original graph and ⊕ be the merge operation, which is the same for both RAs since only vertices are changed by the transport. There are four cases to consider:

      (A) G k v ≡ false ∧ G (k + 1) v ≡ false: In both computations, nothing happens. Therefore, they still remain equal.

      (B) G k v ≡ false ∧ G (k + 1) v ≡ true: In both computations, we merge with the information from the corresponding vertex that is connected and do nothing for the other vertex. Therefore, they still remain equal.

[10]The outer loop is unaffected by this technicality.

(C) G k v ≡ true ∧ G (k + 1) v ≡ false: Analogous to (B).

(D) G k v ≡ true ∧ G (k + 1) v ≡ true: Let a denote the attribute in s′[v] after k-1 iterations. Let b, c denote the attributes that are sent along the directed edges k → v and k + 1 → v, respectively, in the original RA. Then, the question reduces to asking whether (a ⊕ b) ⊕ c ≡ (a ⊕ c) ⊕ b. Under assumption of commutativity and associativity of ⊕, this is true.

(3) For all inputs, the computations are equal: We prove this by induction analogously to step (1). All vertices greater than k + 1 are unaffected by the swap, so using the result from step (2) as the base case, we can complete the induction.

4. This is a corollary of 3. Since paths represent equivalences, p : Fin (suc n) ≡ Fin (suc n) is simply a permutation on Fin (suc n). For any two permutation on m elements, it is always possible to transform from one to the other using at most $\binom{m}{2}$ adjacent swaps, and furthermore, bubble sort gives an algorithm for computing such a sequence of swaps (i.e. one can use the target permutation to define a total order on the elements and use bubble sort to sort according to that total order). Therefore, the statement in 3 extends to all paths of type Fin (suc n) ≡ Fin (suc n).

□

4.2 Topology-Preserving Abstractions

In this section, we discuss a method for constructing global SRP abstractions (i.e. abstractions defined in Section 3.3) using local abstractions on attributes. We will only consider standard SRPs. The method used in this section is inspired by the Bonsai algorithm [4] and by abstract interpretations of routing networks [5]. First, let us define the necessary ingredients.

Definition 4.2.1 (Isotopological Pair)**.** *Let V, A, B be types, and assume we have (G, I, ⊕, f) : RA V A and (G′, I′, ⊕′, f′) : RA V B. We say that (G, I, ⊕, f) and (G′, I′, ⊕′, f′) are an*

isotopological pair if $G = G'$ *definitionally. To enforce this, we write the following:*

$$\begin{aligned}
&\mathit{Isotopological\text{-}Pair} : \mathit{Type} \to \mathit{Type} \to \mathit{Type} \to \mathit{Type} \\
&\mathit{Isotopological\text{-}Pair}\ V\ A\ B = \\
&\quad (\mathit{Graph}\ V) \\
&\quad \times ((V \to A) \times (V \to B)) \\
&\quad \times ((A \to A \to A) \times (B \to B \to B)) \\
&\quad \times ((V \to V \to A \to A) \times (V \to V \to B \to B))
\end{aligned}$$

Definition 4.2.2 (Local Abstraction Conditions)**.** *Let* V, A, B *be types, and assume we have* $(G, (I, I'), (\oplus, \oplus'), (f, f')) : \mathit{Isotopological\text{-}Pair}\ V\ A\ B$ *and a function* $h : A \to B$*. We say that* $(G, (I, I'), (\oplus, \oplus'), (f, f'))$ *satisfies local abstraction conditions under* h *if:*

1. *(*I*-equivalence): For all vertices* $v : V$*, we have that* $I\ v \equiv h\ (I'\ v)$*.*
2. *(*$\oplus$*-equivalence): For all attributes* $a, a' : A$*, we have that* $h\ (a \oplus a') \equiv (h\ a) \oplus' (h\ a')$*.*
3. *(*f*-equivalence): For all vertices* $u, v : V$*, and all attributes* $a : A$*, we have that whenever* $G\ u\ v$ *is* true*,* $h\ (f\ u\ v\ a) \equiv f'\ u\ v\ (h\ a)$*.*

Furthermore, we refer to h *as the attribute abstraction function.*

We refer to the above as local abstractions because they only depend on the local topology of the network, rather than the topology of the whole network. Now, we present the main results of this section, which is the coincidence of local and global abstractions under standard semantics:

Theorem 4.2.3. *Let* A, B *be types and* $n : \mathbb{N}$*, and assume we have* $(G, (I, I'), (\oplus, \oplus'), (f, f')) : \mathit{Isotopological\text{-}Pair}\ (\mathit{Fin}\ (\mathsf{suc}\ n))\ A\ B$ *and a function* $h : A \to B$*. Further, assume that* $(G, (I, I'), (\oplus, \oplus'), (f, f'))$ *satisfies local abstraction conditions under* h*. Then,* $((G, I', \oplus', f'), \mathit{std\text{-}sem}\ n\ B\ (G, I', \oplus', f'))$ *is a quasi-abstraction of* $((G, I, \oplus, f), \mathit{std\text{-}sem}\ n\ A\ (G, I, \oplus, f))$*.*

Proof. The first half of the proof is to construct a state abstraction function abs. We construct the following:

$$\mathsf{abs} : \mathsf{RA\text{-}State\ V\ A\ (G,\ I,\ \oplus,\ f)} \to \mathsf{RA\text{-}State\ V\ B\ (G,\ I',\ \oplus',\ f')}$$

$$\mathsf{abs\ s\ v} = \mathsf{h\ (s\ v)}$$

The second half of the proof is to show semantic equivalence under abs, i.e. we must show that for all states $\mathsf{s} : \mathsf{RA\text{-}State\ V\ A\ (G,\ I,\ \oplus,\ f)}$ target vertices $\mathsf{v} : \mathsf{Fin\ (suc\ n)}$, we have that $\mathsf{h\ (std\text{-}sem\ n\ A\ (G,\ I,\ \oplus,\ f)\ s\ v)} \equiv \mathsf{std\text{-}sem\ n\ B\ (G,\ I',\ \oplus',\ f')\ (\lambda x.\ h\ (s\ x))\ v}$. We prove this by induction on the source vertices $\mathsf{u} : \mathsf{Fin\ (suc\ n)}$ in the "inner loop" of the standard semantics. For all target vertices $\mathsf{v} : \mathsf{Fin\ (suc\ n)}$, the base case of the induction is given by I-equivalence. The inductive case is given by $\oplus$-equivalence and f-equivalence, as well as by the induction hypothesis (note that $(\lambda x.\ \mathsf{h\ (s\ x)})\ \mathsf{v}$ directly β-reduces to $\mathsf{h\ (s\ v)}$, thus allowing the application of f-equivalence). □

To get a full abstraction, the only additional property we need is that the attribute abstraction function h itself is surjective. In practice, this is always possible by using the canonical surjection of the attribute abstraction function (i.e. we restrict the range type of attribute abstraction function to be the minimal type containing its image). We give a brief argument below:

Theorem 4.2.4. *Let A, B be types and* $\mathsf{n} : \mathbb{N}$*, and assume we have* $\mathsf{(G,\ (I,\ I'),\ (\oplus,\ \oplus'),\ (f,\ f'))} : \mathsf{Isotopological\text{-}Pair\ (Fin\ (suc\ n))\ A\ B}$ *and a* ***surjective*** *function* $\mathsf{h} : A \to B$ *with pseudoinverse* $\mathsf{h}^{-1} : \mathsf{Surjection\ A\ B\ h}$*. Further, assume that* $\mathsf{(G,\ (I,\ I'),\ (\oplus,\ \oplus'),\ (f,\ f'))}$ *satisfies local abstraction conditions under* h*. Then,* $\mathsf{((G,\ I',\ \oplus',\ f'),\ std\text{-}sem\ n\ B\ (G,\ I',\ \oplus',\ f'))}$ *is a abstraction of* $\mathsf{((G,\ I,\ \oplus,\ f),\ std\text{-}sem\ n\ A\ (G,\ I,\ \oplus,\ f))}$*.*

Proof. We use the same construction for the abstraction function as above. Surjectivity of the abstraction function follows from the surjectivity of h, since the attribute type is covariant for network states. The proof of semantic equivalence is the same as above. □

4.3 Network Compression

In this section, we explore abstractions of SRPs that reduce the number of network nodes in the RA. We give a set of conditions that, when satisfied, allow us to compress two network nodes into one and yield an abstracted SRP that is bisimilar to the original (i.e. all solutions to the SRP are preserved), but with one fewer node. Similar to the previous section, we assume that the SRP's strictly use the standard semantics. This section is also inspired by the Bonsai algorithm [4]. In this section, we hope to highlight the power of using cubical type theory to carry out proofs. We start by defining the ingredients we need.

Definition 4.3.1 (Fin^* Higher Inductive Type Family)**.** *Let $n : \mathbb{N}$ be a natural number. Then, $Fin^*\ n$ is the type whose terms are the same as $Fin\ (suc\ (suc\ n))$, but in which 0 and 1 are identified with a path. We define this using the previously-defined $AddPath$ type:*

$$Fin^* : \mathbb{N} \to Type$$

$$Fin^*\ n = AddPath\ (Fin\ (suc\ (suc\ n)))\ 0\ 1$$

Next, we prove that this HIT is path equal to the finite set type with one less term, and this will serve as the basis for our vertex compression proof.

Lemma 4.3.2. *For all $n : \mathbb{N}$, we have that $Fin^*\ n \equiv Fin\ (suc\ n)$.*

Proof. We first prove that the two types are equivalent. For the forward map f, we define it higher-inductively as follows:

$$\mathsf{f} : \mathsf{Fin}^*\ \mathsf{n} \to \mathsf{Fin}\ (\mathsf{suc}\ \mathsf{n})$$

$$\mathsf{f}\ (\mathsf{in}\ 0) = 0$$

$$\mathsf{f}\ (\mathsf{in}\ (\mathsf{fsuc}\ \mathsf{x})) = \mathsf{x}$$

$$\mathsf{f}\ (\mathsf{same}\ \mathsf{i}) = 0$$

Since f is mapping out of the HIT, it must map the path same to a path in the target type. Since Fin (suc n) has no non-trivial paths, we must map it to a constant path. Here, we chose the constant path $\langle i \rangle\ 0 : 0 \equiv 0$ (1 is fsuc 0 definitionally). For the reverse map g, we chose the straightforward $g = \lambda x.\ \mathsf{in}\ x$. Next, we prove the equivalence. For $gf : (x^* : \mathsf{Fin}^*\ n \to g\ (f\ x^*) \equiv x^*$, we offer the following proof by higher induction:

$$
\begin{aligned}
&gf : (x^* : \mathsf{Fin}^*\ n) \to g\ (f\ x^*) \equiv x^* \\
&gf\ (\mathsf{in}\ 0) = \mathsf{sym}\ \mathsf{same} \\
&gf\ (\mathsf{in}\ (\mathsf{fsuc}\ n)) = \mathsf{refl} \\
&gf\ (\mathsf{same}\ i) = \langle j \rangle\ \mathsf{same}\ (\sim j \vee i)
\end{aligned}
$$

Of note is that g (f (in 0)) evaluates to in 1, so in the first case, we must provide a proof of in 1 ≡ in 0, which is same : in 0 ≡ in 1 reversed. In the higher inductive case, we must provide a proof that in 1 ≡ same i, which we do using a connection. One can interpret the expression ∼j ∨ i as a Boolean-like expression, where the values of the dimension type, i0 and i1, play the role of false and true, respectively. When j is equal to i0 at the left boundary of the path, the expression evaluates to i1, giving us same i1, which is definitionally in 1. When j is equal to i1 at the right boundary of the path, the expression evaluates to i, giving us same i, which is exactly the required proof. The other half of the proof, fg, is given by reflexivity. Finally, we use univalence to convert this equivalence into a path. □

To obtain the final result, we employ the following proof strategy: Given an SRP defined on vertex type Fin (suc (suc n)) for some $n : \mathbb{N}$, define an analogous SRP (i.e. an analogous RA, as well as an analogous semantics) on the vertex type $\mathsf{Fin}^*\ n$. This SRP must satisfy two properties:

1. It must be bisimilar to the original SRP, and

2. After being transported along the vertex path established in the above lemma, its seman-

tics must be equal to the standard semantics.

By Theorem 4.1.4, we have that the transported SRP is equivalent to the analogous SRP (i.e. the one defined on vertex type $\mathsf{Fin}^*\ \mathsf{n}$) in the sense of Definition 3.3.3, which, by Theorem 4.1.2 implies that it is also bisimilar. Then, we complete the proof by using the transitive property of bisimilarity.

We proceed by first defining the necessary conditions for defining the analogous SRP.

Definition 4.3.3 (Compression Conditions)**.** *Let $\mathsf{n} : \mathbb{N}$ be a natural number and A be a type. Assume we have ((G, I, ⊕, f), std-sem (suc n) A (G, I, ⊕, f)) : SRP (Fin (suc (suc n))) A. We say that ((G, I, ⊕, f), std-sem (suc n) A ra) satisfies the 0/1 compression conditions if:*

1. *(0/1 G-equivalence): For all vertices $\mathsf{v} : \mathsf{Fin}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{n}))$, we have that $\mathsf{G}\ 0\ \mathsf{v} \equiv \mathsf{G}\ 1\ \mathsf{v}$ and that $\mathsf{G}\ \mathsf{v}\ 0 \equiv \mathsf{G}\ \mathsf{v}\ 1$.*

2. *(0/1 I-equivalence): We have that $\mathsf{I}\ 0 \equiv \mathsf{I}\ 1$.*

3. *(0/1 f-equivalence): For all vertices $\mathsf{v} : \mathsf{Fin}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{n}))$ and all attributes $\mathsf{a} : \mathsf{A}$, we have that $\mathsf{f}\ 0\ \mathsf{v}\ \mathsf{a} \equiv \mathsf{f}\ 1\ \mathsf{v}\ \mathsf{a}$ and that $\mathsf{f}\ \mathsf{v}\ 0\ \mathsf{a} \equiv \mathsf{f}\ \mathsf{v}\ 1\ \mathsf{a}$.*

4. *(0/1 f-distribution-over-⊕): For all vertices $\mathsf{v} : \mathsf{Fin}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{n}))$ and all attributes $\mathsf{a}\ \mathsf{a}' : \mathsf{A}$, we have that $\mathsf{f}\ 0\ \mathsf{v}\ \mathsf{a} \oplus \mathsf{f}\ 1\ \mathsf{v}\ \mathsf{a}' \equiv \mathsf{f}\ 1\ \mathsf{v}\ (\mathsf{a} \oplus \mathsf{a}')$*[11]

5. *(⊕-idempotence): For all attributes $\mathsf{a} : \mathsf{A}$, we have that $\mathsf{a} \oplus \mathsf{a} \equiv \mathsf{a}$.*

6. *(A-set): The attribute type A is a homotopical set, i.e. there exists at most one path (up to homotopy) between any two terms of type A.*

Now, we are able to define the analogous SRP, which we will refer to as the compression proposal.

Definition 4.3.4 (Compression Proposal)**.** *Let $\mathsf{n} : \mathbb{N}$ be a natural number and A be a type. Assume we have ((G, I, ⊕, f), std-sem (suc n) A (G, I, ⊕, f)) : SRP (Fin (suc (suc n))) A such*

[11] Equivalently, we could have f 0 v (a ⊕ a'), but f 1 v (a ⊕ a') simplifies the proof for technical reasons.

that it satisfies the 0/1 compression conditions. We say that $((G^*, l^*, \oplus^*, f^*)$, alt-sem n A $(G^*, l^*, \oplus^*, f^*))$ *is the compression proposal of* $((G, l, \oplus, f)$, std-sem (suc n) A $(G, l, \oplus, f))$, *whenever its components are defined as follows:*

G^* : Graph $(Fin^*\ n)$ – *Defined by higher induction on both the source and target vertices. In the case of zero bound dimension variables (i.e. the point case)* G^* (in u) (in v), *define it to be exactly* G u v. *In the higher inductive cases with one bound dimension variable (i.e. the line case), map the line in* Fin^* n *to the path given by 0/1* G*-equivalence. In the case with two bound dimension variables (i.e. the square case* G^* (same i) (same j)*), map the square in* Fin^* n *to the trivial square in* Bool, *as* Bool *is a homotopical set.*

l^* : $Fin^*\ n \to A$ – *Defined by higher induction on the vertex. In the point case, directly use* l *to map. In the line case, map the line to the path given by 0/1* l*-equivalence.*

$\oplus^*$: $A \to A \to A$ – *Define as* $\oplus$ *directly.*

f^* : $Fin^*\ n \to Fin^*\ n \to A \to A$ – *Define by higher induction on both the source and target vertices. In the point case, define directly with* f. *In the line case, map the line to the path given by 0/1* f*-equivalence. In the square case, map the square to the trivial square in* A *by using the fact that* A *is a homotopical set.*

alt-sem : $(n : \mathbb{N})$ (A : Type) $(ra^* :$ SRP $(Fin^*\ n)$ A$) \to$ RA-State $(Fin^*\ n)$ A $ra^* \to$
RA-State $(Fin^*\ n)$ A ra^*
– *Define analogously to the standard semantics, except in the "inner loop," skip vertex* in 0 *and start with vertex* in 1 *and count up to vertex* in (fsuc n) *in ascending order. No higher induction is needed for this definition.*

Next, we prove that the compression proposal is an abstraction of and is bisimilar to the original SRP.

Theorem 4.3.5. *Let* $n : \mathbb{N}$ *be a natural number and* A *be a type. Assume we have* $((G, l, \oplus, f)$, std-sem (suc n) A $(G, l, \oplus, f)) :$ SRP (Fin (suc (suc n))) A *such that it satisfies the 0/1 com-*

pression conditions. Further, let $((G^*, I^*, \oplus^*, f^*), \textsf{alt-sem}\ n\ A\ (G^*, I^*, \oplus^*, f^*))$ *be the compression proposal of* $((G, I, \oplus, f), \textsf{std-sem}\ (\textsf{suc}\ n)\ A\ (G, I, \oplus, f))$*. Then,* $((G^*, I^*, \oplus^*, f^*), \textsf{alt-sem}\ n\ A\ (G^*, I^*, \oplus^*, f^*))$ *is an abstraction of and is bisimilar to* $((G, I, \oplus, f), \textsf{std-sem}\ (\textsf{suc}\ n)\ A\ (G, I, \oplus, f))$*.*

Proof. First, we must define an abstraction function. We give the following higher inductive definition:

$$\textsf{abs} : (\textsf{Fin}\ (\textsf{suc}\ (\textsf{suc}\ n)) \to A) \to \textsf{Fin}^*\ n \to A$$

$$\textsf{abs}\ s\ (\textsf{in}\ 0) = s\ 0 \oplus s\ 1$$

$$\textsf{abs}\ s\ (\textsf{in}\ 1) = s\ 0 \oplus s\ 1$$

$$\textsf{abs}\ s\ (\textsf{in}\ (\textsf{fsuc}\ (\textsf{fsuc}\ x))) = s\ (\textsf{fsuc}\ (\textsf{fsuc}\ x))$$

$$\textsf{abs}\ s\ (\textsf{same}\ i) = s\ 0 \oplus s\ 1$$

Next, we must show that the two SRPs are semantically equivalent under the abstraction function $\textsf{abs}$, i.e. for all states s of the original SRP and vertices v^* of the compression proposal, we have that $\textsf{abs}\ (\textsf{std-sem}\ (\textsf{suc}\ n)\ A\ (G, I, \oplus, f)\ s)\ v^* \equiv \textsf{alt-sem}\ n\ A\ (G^*, I^*, \oplus^*, f^*)\ (\textsf{abs}\ s)\ v^*$. We prove this in three steps.

(1) We show that for all states $s : \textsf{Fin}\ (\textsf{suc}\ (\textsf{suc}\ n)) \to A$, we have that $\textsf{std-sem}\ (\textsf{suc}\ n)\ A\ (G, I, \oplus, f)\ s\ 0 \equiv \textsf{std-sem}\ (\textsf{suc}\ n)\ A\ (G, I, \oplus, f)\ s\ 1$. We prove this by induction on the "inner loop" of $\textsf{std-sem}$. The base case is given by $0/1$ I-equivalence. The inductive case is given by $0/1$ G- and $0/1$ f-equivalences, as well as by the induction hypothesis.

(2) We show that for all $v : \textsf{Fin}\ (\textsf{suc}\ (\textsf{suc}\ n))$, we have that $\textsf{std-sem}\ (\textsf{suc}\ n)\ A\ (G, I, \oplus, f)\ s\ v \equiv \textsf{alt-sem}\ n\ A\ (G^*, I^*, \oplus^*, f^*)\ (\textsf{abs}\ s)\ (\textsf{in}\ v)$[12]. We prove this by induction on the inner loops of the two semantics simultaneously. Note that $\textsf{std-sem}$ and $\textsf{alt-sem}$ start their "inner loops" at different counts (0 vs. $\textsf{in}\ 1$, respectively). For this proof, we will consider the

[12]Here, what we are doing is decoupling the bulk of the proof obligation from v^*, thereby allowing us to use induction on v, rather than higher induction on v^*. We patch it back together in (3).

case when loops are at vertex 1/in 1 to be the base case of the induction. In the point case of the higher induction and in the base case of the inner loop induction, there are two possibilities to consider:

(A) G 0 v ≡ false ∧ G 1 v ≡ false: No transfers have occurred, so we must show l v ≡ l* (in v). They are equal by definition.

(B) G 0 v ≡ true ∧ G 1 v ≡ true: Transfers have occurred. Note that transfers occurred from both vertices 0 and 1 for std-sem, but only for vertex in 1 for alt-sem because we skipped in 0. Therefore, we must show (l v ⊕ f 0 v (s 0)) ⊕ f 1 v (s 1) ≡ l* (in v) ⊕ f* (in 1) (in v) (abs s (in 1)). Note that the right hand side definitionally reduces to l v ⊕ f 1 v (s 0 ⊕ s 1). Therefore, we prove this equality by using the associative property of ⊕, together with 0/1 f-distribution-over-⊕.

The inductive case is directly provable using only the induction hypothesis, as abs s directly passes the along the corresponding value from s.

(3) Finally, we prove the overall statement by higher induction on the target vertex v*. For the point cases in 0 and in 1, we use (1), as well as ⊕-idempotence to show that abs applied on the left hand side is just the identity (up to a conversion from in v to v), and we use (2) to equate it to the right hand side. For the other point cases, abs is definitionally the identity (up to a conversion from in v to v), and so (2) alone is enough. Finally, for the line case, we are asked to produce a square in A, which we do using the fact that A is a homotopical set.

Next, to show bisimilarity, we must define a canonical pseudoinverse for the abs. We will choose rev-abs = λs*. λv. s* (in v) as the reverse map, and we prove that it maps to something in the pre-image by higher induction. For the point cases in 0 and in 1, we must show that s* (in 0) ⊕ s* (in 1) is equal to s* (in 0) (respectively s* (in 1)). Since s* is a map out of the HIT, it must respect the path given by same, so s* (in 0) ≡ s* (in 1) (witnessed by congruence on same). But since they are path equal, we can apply ⊕-idempotence to

get that $\mathsf{s}^*\ (\mathsf{in}\ 0) \oplus \mathsf{s}^*\ (\mathsf{in}\ 1)$ path equals $\mathsf{s}^*\ (\mathsf{in}\ 0)$ (respectively $\mathsf{s}^*\ (\mathsf{in}\ 1)$). For the other point cases, the abstraction and its pseudoinverse cancel definitionally. For the line case, we use a similar technique to the point cases.

By Theorem 4.1.1, we know that solutions of the original SRP map to solutions of the compression proposal. By the surjection proof that we just carried out, we know that $\mathsf{abs} \circ \mathsf{rev\text{-}abs}$ is the identity (up to a path) for all states of the compression proposal, which in particular include the solutions. Furthermore, we observe that due to $\oplus$-idempotence, $\mathsf{rev\text{-}abs} \circ \mathsf{abs}$ is the identity (up to a path) for all states s of the original SRP whenever $\mathsf{s}\ 0 \equiv \mathsf{s}\ 1$. In particular, all solutions to the original SRP have this property, since they are equal to their own next state, and we observed previously in (1) that the next state of any state of the original SRP must have the property that $\mathsf{s}\ 0 \equiv \mathsf{s}\ 1$. Therefore, when restricted to solutions, $\mathsf{rev\text{-}abs} \circ \mathsf{abs}$ is the identity. Finally, we show that solutions of the compression proposal map back to solutions of the original SRP by using a similar technique to the one used to show reverse semantic equivalence in Lemma 3.3.7 (the difference is exactly that $\mathsf{rev\text{-}abs} \circ \mathsf{abs}$ is only sometimes the identity function, i.e. when restricted to solutions), followed by the technique in Theorem 4.1.1 to go from "reverse semantic equivalence" to reverse soundness. □

Finally, all that is left is to verify that after the compression proposal is transported along the path $\mathsf{Fin}^*\ \mathsf{n} \equiv \mathsf{Fin}\ (\mathsf{suc}\ \mathsf{n})$ given in Lemma 5.3.2, the semantics of the transported SRP is equal to the standard semantics. From the definition of $\mathsf{alt\text{-}sem}$, one can see that this is trivially true, i.e. the forward map f maps $\mathsf{in}\ (\mathsf{fsuc}\ \mathsf{x})$ to x, so the transported $\mathsf{alt\text{-}sem}$ starts counting its inner loop from 0 and goes through all of $\mathsf{Fin}\ (\mathsf{suc}\ \mathsf{n})$ in ascending order, which is exactly how $\mathsf{std\text{-}sem}$ computes.

5 Discussion

5.1 Applications of the Cubical SRP Model

In this section, we discuss a few ways in which we can use the SRP model that we have developed in cubical type theory, together with the theorems about it that have been proven, to construct useful algorithms for studying routing network.

5.1.1 Anonymization Schemes

Sometimes, we may wish to let a third party study properties of routing network that we control, such as connectivity or reachability of network nodes, but we do not want reveal any private or sensitive information, such as IP addresses, to such third party. In order to protect such information, we could, in whatever medium we choose to encode our routing network, additionally choose to anonymize the IP addresses, i.e. by changing all the IP addresses to random addresses. However, to preserve the ability for a third party to study the network, we must replace all instances of the same address with the same randomly generated address. Likewise, two addresses that are originally distinct must remain distinct after anonymization.

If we encode the routing network nodes, topology, and routing protocol using the routing algebra defined in our SRP model, we can very easily also perform an anonymizations using the cubical techniques, since an anonymization scheme is nothing more than a bijection between two spaces of IP addresses. Below is an sketch of how one might go about performing an anonymization:

We begin by defining a type for the concrete IP addresses:

$$\mathsf{IP\text{-}Addr} : \mathsf{Type}$$

$$\mathsf{IP\text{-}Addr} = \ldots$$

Next, we can define a type for the attributes. Attributes may depend on the IP addresses, so we define something with the following signature:

$$\mathsf{Attr} : (\mathsf{Addr} : \mathsf{Type}) \to \mathsf{Type}$$
$$\mathsf{Attr}\ \mathsf{Addr} = \ldots$$

Then, we encode the routing network into a routing algebra as follows:

$$\mathsf{routing\text{-}network} : \mathsf{RA}\ \mathsf{IP\text{-}Addr}\ (\mathsf{Attr}\ \mathsf{IP\text{-}Addr})$$
$$\mathsf{routing\text{-}network} = \ldots$$

Afterward, we can define a type of anonymized IP addresses, together with an anonymization scheme that doubles as a proof that the two IP address types are equivalent:

$$\mathsf{IP\text{-}Addr'} : \mathsf{Type}$$
$$\mathsf{IP\text{-}Addr'} = \ldots$$
$$\mathsf{ip\text{-}eq} : \mathsf{IP\text{-}Addr} \equiv \mathsf{IP\text{-}Addr'}$$
$$\mathsf{ip\text{-}eq} = \ldots$$

Finally, to perform the anonymization, simply transport routing-network along the path defined by ip-eq, as follows:

$$\mathsf{anon\text{-}network} : \mathsf{RA}\ \mathsf{IP\text{-}Addr'}\ (\mathsf{Attr}\ \mathsf{IP\text{-}Addr'})$$
$$\mathsf{anon\text{-}network} = \mathsf{transport}\ (\mathsf{RA}\ \mathsf{IP\text{-}Addr}\ (\mathsf{Attr}\ \mathsf{IP\text{-}Addr}))\ (\mathsf{RA}\ \mathsf{IP\text{-}Addr'}\ (\mathsf{Attr}\ \mathsf{IP\text{-}Addr'}))$$
$$(\langle \mathsf{i} \rangle\ (\mathsf{RA}\ (\mathsf{ip\text{-}eq}\ \mathsf{i})\ (\mathsf{Attr}\ (\mathsf{ip\text{-}eq}\ \mathsf{i}))))\ \mathsf{routing\text{-}network}$$

Analogously, we can use transport to anonymize any network state, which we can then, for example, use an interpreter for computational cubical type theory to compute exactly

and give to a third party. Furthermore, since the anonymized network is produced through transports along the vertex and attribute types, we have the additional guarantee by Theorems 4.1.1 and 3.3.3 that the anonymized network is behaviorally equivalent to the concrete network for any semantics that we wish to assign to it[13].

5.1.2 Step-wise Abstraction Algorithm

Another way in which we could use our results is to construct an abstraction algorithm to reduce the complexity of a routing algebra. Suppose that we are working with a fairly large routing network with many attributes, as well as redundancies in the topology, and suppose that we are interested in studying a specific property, such as reachability of network nodes. Then, one can reduce the size of the problem by abstracting away many unnecessary details. Suppose the original network is modeled by a standard SRP. The theorems that we have presented in Section 5 allow us to do the following (which is inspired by the Bonsai algorithm [4] and abstract interpretations [5]):

1. First, abstract the attributes so that only relevant information is preserved, and redefine the routing algebra accordingly so that local abstraction conditions are satisfied.

2. Next, identify nodes in the simplified graph that satisfy the compression conditions and combine them together.

3. Repeat step 2 until no more compression is possible.

In step 1, soundness is guaranteed by Theorem 4.2.4. In step 2, our work in Section 5.3 show that the compression will yield a bisimilar network to the one obtained after step 1. Furthermore, by Theorem 4.1.4, we can expand the result in Section 5.3 to allow to compress any two nodes by simply renaming them to $\mathsf{0}$ and $\mathsf{1}$. Furthermore, the postcondition of compression in Section 5.3 is that we are left with a routing algebra that is one node

[13]Technically, anon-network is produced by a single transport of the vertex and attribute types simultaneously, but it is not difficult to show that this is path equivalent to the case where we transport them one at a time.

smaller, but still uses standard semantics, which is exactly the precondition for compression. Therefore, this justifies step 3, where we can continue to compress the network one node at a time. Finally, by the transitive property of abstraction/soundness, we are guaranteed to get a network that is at least a sound abstraction of the original. This implies that any property that holds for all solutions of the final abstract network, such as reachability, will analogously hold for the original network.

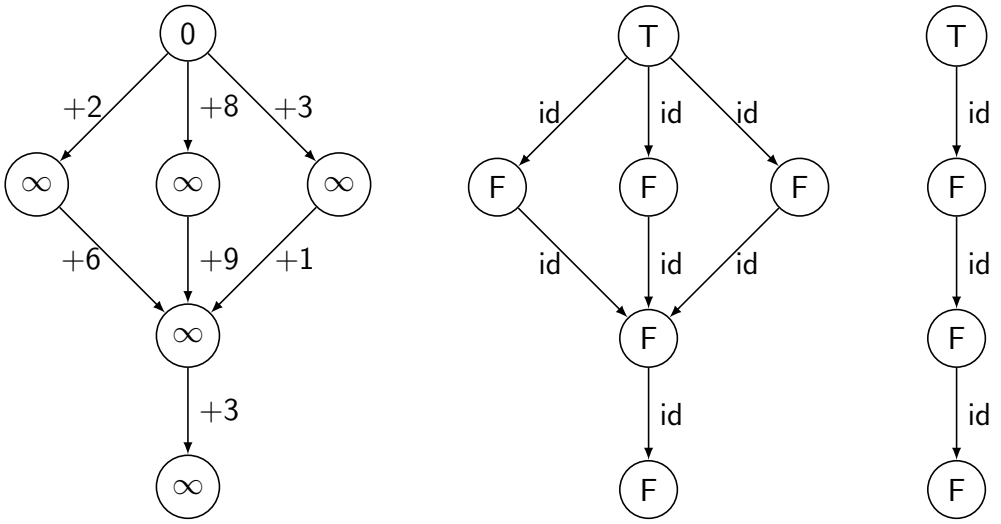


Figure 2: Step-wise Compression

Figure 2 shows this algorithm in action. We use the same graph as before. First, we choose the the attribute abstraction function $\mathsf{h} : \mathbb{N}_\infty \to \mathsf{Bool}$ that maps all finite numbers to true (denotes by T) and infinity to false (denoted by F). We note that this function is surjective. If we define the merge operation as logical or and the transfer functions for all edges as the identity function, then the new SRP locally (and therefore globally) abstracts the original SRP. After performing this abstraction, we see that the middle three nodes are now compressible, and so we compress them in turn to yield the final abstract SRP on the right. Figure 3 shows the solution of the original SRP compared to the solution of the abstract SRP. It turns out that the attribute abstraction step does not introduce any new solutions in this case, and so the unique solution to the original SRP is perfectly abstracted by the unique solution to the abstract SRP. Specifically, we see that reachability, denoted

by a finite attribute in the concrete case and by true in the abstract case, is preserved for all nodes in the respective solutions, keeping in mind that the second node in the abstract SRP represents all three parallel nodes in the concrete SRP.

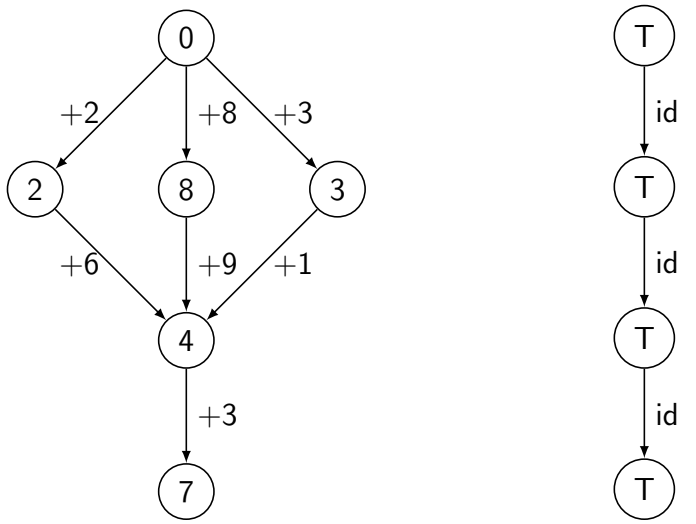


Figure 3: Comparing Solutions

5.2 Using Cubical Type Theory

For this project, we chose to use a cubical type theory to model and reason about the SRP. The choice of using cubical type theory as a logic system comes with many advantages and some disadvantages over using a more traditional logic system, such as Martin-Löf type theory.

5.2.1 Advantages of Modeling in Cubical Type Theory

One key advantage of using a cubical type theory is gaining access to univalence for free. The concept of univalence is built-in for cubical type theories, and so it makes reasoning about behavioral equivalence of SRPs much easier. For example, Theorems 4.1.3 and 4.1.4 (parts 1 and 2) are fairly easy to prove in a cubical type theory because we are able to treat vertex and attribute equivalences as type equalities, which everything in cubical type theory

must respect by default. An analogous proof in Martin-Löf type theory would require one to define and reason about transports along type equivalences individually for each type (e.g. what it would mean to transport a function along an equivalence for types that appear in covariant vs. contravariant positions), while cubical type theory would provides us with tools to reason about all transports uniformly.

Furthermore, cubical type theory also allows us to define higher inductive types, which is not definable is Martin-Löf type theory. The use of a higher inductive type was crucial in making the proof of compression bisimilarity in Section 5.3 much easier to carry out by allowing us to more easily modularize the proof obligations. For example, using a higher inductive type as an intermediary, we were able to separate reasoning about the necessary conditions to generate a bisimilarity from the details of the node compression itself. Furthermore, we in fact get compression for free by simply using transport, which incidentally also allowed us to reuse Theorem 4.1.4 in our overall argument for compression bisimilarity.

5.2.2 Disadvantages of Modeling in Cubical Type Theory

One disadvantage of using a cubical type theory is the pervasiveness of "off-by-a-path" computations. For example, in cubical type theory, in the absence of concrete type definitions, transport always computes up to a path, regardless of how trivial the path may be. Therefore, something as simple as transport A A refl a where a : A does not reduce definitionally to a if the type A is not known concretely[14]. The Agda proof checker will reject expressions which are "off by a path," leading to somewhat significant effort spent looking for ways to fix such expressions.

Furthermore, having path-based equalities also, at times, greatly increases the complexity of the proof obligation of simple statements. For example, most of the inductive types that are used in the cubical SRP model are strict sets, i.e. they have no inductively-defined path constructors. Proving decidability of equality for these types is trivial in Martin-Löf type

[14]Conversely, if, for example, we know that A is $\mathbb{N}$ and that a is 0, then the transport does reduce, though this may not be true for all concrete types.

theory (provided that the constructors only depend on other types with decidable equality, or a smaller term of the same type); one can simply do induction on the structure of the type. However, in cubical type theory, proving even the basic statement of $\mathsf{true} \not\equiv \mathsf{false}$ (let alone decidable equality) is non-trivial and requires appealing to the propositional nature of the paths in strict sets.

Proof-relevance of equalities also adds to the complexity of the logic. In the proof of compression bisimilarity, we added the assumption that the attribute type is a homotopical set in order to avoid dealing with proof-relevance of equalities among attributes in the higher inductive setting, since unlike the path constructor same, higher-order paths are not immediately relevant to the proof technique.

6 Future Work

There are many ideas left to be explored in the future. For example, more work can be done to formalize the algebraic properties of the various relations between SRPs defined in this work. So far, we have come up with the fact that abstractions form a partial order among finite non-empty SRPs, and furthermore we have identified the appropriate top element of this partial order. One thing that is has not yet been explored is whether the abstraction relation forms a $\vee$-semilattice, i.e. whether there exists a unique least upper-bound for any two finite non-empty SRPs.

Another interesting question that remains to be answered is whether there exist any useful characterizations of SRPs which are similar but not bisimilar. So far, we have seen that the unit SRP is similar to all finite non-empty SRPs; however, the unit SRP is degenerate in the sense that it carries no real information and is therefore not useful as a practical abstraction. It may be possible to show that a compression of redundant attributes in the attribute type, under appropriate compression conditions, results in an abstract SRP which is strictly similar to the original.

Finally, there is the task of formalizing existing theory, such as the Bonsai algorithm, in this cubical-type-theoretic model of the stable routing problem in order to show its correctness. The network compression algorithm outlined in this work always produces an abstraction that is semantically equivalent to the original SRP; however, it is yet to be proven whether the abstraction produced by the Bonsai algorithm satisfies this strict criterion or not. Perhaps some care is needed either to characterize the conditions under which Bonsai produces a semantically equivalent SRP, or to relax our definition of semantic equivalence to accommodate the Bonsai algorithm.

There is still much to be explored in the intersection of computer science and cubical type theory. I hope that whoever reads this will be inspired to continue this study into the potential of applying cubical techniques to related problems.

7 Acknowledgments

I would like to thank my advisors Matthew Weaver and Prof. David Walker for meeting with me every week, for giving me directions to explore, and for giving me much needed feedback throughout this project over the past year. I would also like to thank Prof. Aarti Gupta for agreeing to provide feedback as the second reader for this write-up and Prof. Brian W. Kernighan for listening to and providing feedback for my presentation.

References

[1] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[2] Timothy G. Griffin and João Lus Sobrinho. Metarouting. 2009.

[3] João Luís Sobrinho. An algebraic theory of dynamic network routing. *IEEE/ACM Transactions on Networking*, 13:1160–1173, 2005.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *SIGCOMM*, 2018.

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Abstract interpretation of distributed network control planes. In *POPL*, 2020.

[6] Anders Mortberg. Cubical agda. `https://homotopytypetheory.org/2018/12/06/cubical-agda/`, 2018.

[7] Ulf Norell et al. Agda's documentation, 2019.

[8] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. 04 2002.

[9] Michael A. Warren. The strict -groupoid interpretation of type theory. 2011.

[10] Steve Awodey and Michael A. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 146(1):4555, Jan 2009.

[11] Marc Bezem, Thierry Coquand, and Simon Huber. The univalence axiom in cubical sets, 2017.

[12] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom, 2016.

[13] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Kuen-Bang Hou (Favonia), Robert Harper, and Daniel R. Licata. Cartesian cubical type theory. `https://www.cs.cmu.edu/~rwh/papers/uniform/uniform.pdf`, 2017.

[14] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26, pages 107–128, 2014.

[15] Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. Cartesian cubical computational type theory: Constructive reasoning with paths and equalities. *Computer Science Logic*, 2018.

[16] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Kuen-Bang Hou (Favonia), Robert Harper, and Daniel R. Licata. Syntax and models of cartesian cubical type theory. `https://github.com/dlicata335/cart-cube/blob/master/cart-cube.pdf`, 2017.

Appendix

All of the formalization in Agda for this project can be found at `https://github.com/coolfan/cos-iw-2019`.