Avenir: Managing Data Plane Diversity with Control Plane Synthesis

Eric Hayden Campbell
Cornell

William T. Hallahan
Yale

Priya Srikumar
Cornell

Carmelo Cascone
ONF

Jed Liu
Intel

Vignesh Ramamurthy
Infosys

Hossein Hojjat*
Tehran & TeIAS

Ruzica Piskac
Yale

Robert Soulé
Yale

Nate Foster
Cornell

Abstract

The classical conception of software-defined networking (SDN) is based on an attractive myth: a logically centralized controller manages a collection of homogeneous data planes. In reality, however, SDN control planes must deal with significant diversity in hardware, drivers, interfaces, and protocols, all of which contribute to idiosyncratic differences in forwarding behavior that must be dealt with by hand.

To manage this heterogeneity, we propose Avenir, a synthesis tool that automatically generates control-plane operations to ensure uniform behavior across a variety of data planes. Our approach uses counter-example guided inductive synthesis and sketching, adding network-specific optimizations that exploit domain insights to accelerate the search. We prove that Avenir's synthesis algorithm generates correct solutions and always finds a solution, if one exists. We have built a prototype implementation of Avenir using OCaml and Z3 and evaluated its performance on realistic scenarios for the ONOS SDN controller and on a collection of benchmarks that illustrate the cost of retargeting a control plane from one pipeline to another. Our evaluation demonstrates that Avenir can manage data plane heterogeneity with modest overheads.

1 Introduction

The network control plane plays a similar role in modern systems as a classical OS kernel. It manages resources such as end-to-end forwarding paths, maps incoming traffic onto those paths, and enforces policy such as ensuring isolation between tenants in a public cloud.

One challenge that complicates the design of the control plane is dealing with data plane heterogeneity. Much as an OS kernel manages hardware resources for a variety of peripherals, the network control plane manages hardware resources for a variety of data planes. Most network operators purchase equipment from multiple manufacturers to avoid lock-in, which results in devices with heterogeneous feature sets, and even devices manufactured by the same vendor tend to evolve over time. This heterogeneity manifests as complexity throughout the control plane, appearing in low-level drivers and SDKs, device OSes (e.g., SONiC [42], FBOSS [5], Stratum [45]), higher-level APIs (e.g., OpenFlow [23], OpenConfig [30], P4Runtime [6]), and even network applications.

As an example, switches based on Broadcom ASICs such as Trident2, Tomahawk and Qumran-MX all expose an OpenFlow-like API to SDN controllers (or more precisely, the OF-DPA [32] abstraction). However, due to differences in the chips, the API behaves in subtly different ways on various devices. For instance, the Termination MAC table, which determines whether to route packets or bridge them, appears in all three devices but behaves differently on Trident2/Tomahawk versus Qumran-MX—the former supports matching on the ingress port while the latter does not. This discrepancy has led to bugs: before a special case was added to the ONOS controller, multicast traffic on Qumran-MX devices was flooded out on all ports rather than being forwarded to the proper multicast groups [34].

This anecdote is just one example of a more pervasive problem. The OF-DPA API specification [32] is more than 150 pages of English prose. The ONOS development team took two years to validate Qumran-MX switches and certify them as production-ready. This effort included multiple iterations of testing and bug fixing to port the Tomahawk driver to Qumran-MX, even though the devices come from the same vendor, implement the same protocols, and expose the same control plane abstractions. In practice, the problem of mapping abstract specifications of forwarding behavior down to real-world targets seems too hard to solve by hand.

Control Plane Synthesis. This paper presents a different approach to managing data plane diversity. Rather than relying on careful engineers to manually craft bug-free mappings from high-level abstractions to low-level targets, we show how to automate this task using program synthesis. More precisely, we develop Avenir, a system that automatically translates control plane operations written against an abstract forwarding specification (e.g., OF-DPA), into lower-level operations for a physical target (e.g., Qumran-MX).

*Work performed at Cornell.

Our approach proceeds in two steps. First, we use the P4 language [4] to model the behavior of the abstract and target devices. Although P4 was originally designed as a domain-specific language for programming devices like Barefoot's Tofino switch, it is also being used as a specification language for fixed-function devices (e.g., at Google [47]). For our purposes, what matters is that P4 provides a precise, bit-level specification of data plane behavior that can be mechanized using an SMT solver [21]. Hence, when P4 is not sufficiently expressive to model the pipelines' behavior, our approach should still be applicable. For example, one could work with other packet-processing languages like NPL, eBPF, or vendor SDKs. Second, we use *counterexample guided inductive synthesis* (CEGIS) [41] to translate the abstract control plane operations, such as inserting entries into a match-action table, into equivalent physical operations. Our synthesis algorithm is provably *sound* (i.e., if it succeeds, the abstract and target behaviors are guaranteed to be equivalent) and *complete* (i.e., if there a translation for a given operation, Avenir finds it).

At a technical level, we exploit the insight that data plane devices are fundamentally simple. When modeled as programs, they lack complex features like pointers and loops (parser state machines and uses of recirculation can be finitely unrolled in practice). Although data planes exhibit complexity in other dimensions, such as the number of protocols or table entries they support, the amount of processing they perform on any given packet is limited. Hence, it is possible to model their behavior using simple, loop-free programs that are amenable to analysis using automated solvers. In particular, P4's match-action tables can be treated as *program sketches*—i.e., programs populated with unknown variables called *holes*. The CEGIS loop synthesizes table operations by inductively filling in the program's holes. The controller interacts with these tables incrementally: table entries are usually not changed wholesale, but in small batches. We incrementally synthesize individual control plane operations rather than full tables, which greatly improves Avenir's efficiency.

However, even if one does synthesis incrementally, scaling up to real-world programs remains a significant challenge. Program synthesis has often been used in offline settings, where performance is not a critical concern. However, a typical control plane might modify a table every few milliseconds. To enable online operation, Avenir incorporates heuristic optimizations such as ignoring existing table rules (when possible), and learning "templates" that cache repeated patterns and avoid unnecessary calls to the SMT solver.

Implementation and Evaluation. We have built an implementation of Avenir in OCaml and Z3, and evaluated its effectiveness and scalability. In particular, we used Avenir to perform a "reboot" load test from the ONOS controller with moderate overhead: ONOS takes 15 minutes to generate 40k abstract IPv6 forwarding rules while our tool translates the insertions to a Broadcom pipeline in about 12 minutes. We conducted a series of experiments in which we retarget control

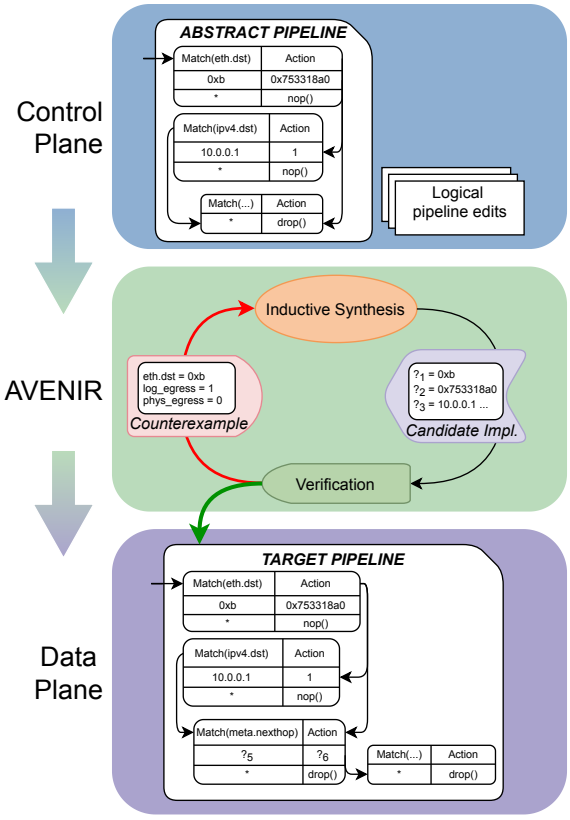

Figure 1: Avenir maps control plane operations for an abstract pipeline into corresponding operations for a target using sketch-based synthesis. The synthesis loop alternates between verifying the correctness of a candidate implementation and learning from counterexamples to generate a better one; the holes (e.g., $?_5$) in the target sketch denote missing values that are filled in using an SMT solver.

planes from one pipeline to another, and show that generated rules successfully forward packets on the Bmv2 software switch. Finally, to assess Avenir's scalability, we ran experiments on synthetic microbenchmarks.

Contributions. This paper presents Avenir, a practical control plane synthesis tool based on the following contributions:

- We present synthesis algorithm that incrementally computes changes to data plane operations, motivated by examples in real-world control planes.
- We formalize our synthesis algorithm and prove (in the appendix) that it is sound and complete.
- We present optimizations that leverage incrementality and domain insights to accelerate synthesis.
- We discuss an implementation and show through case studies and microbenchmarks that Avenir synthesizes control plane operations correctly with modest overheads.

L2_fwd	
eth.dst	**Action**
ABB28FC	set_out(5)

L3_fwd	
ipv4.dst	**Action**
10.0.0.1	set_out(8)

(a) Pipeline 1, with L2 and L3 forwarding for the original, homogeneous network.

L2_fwd	
eth.dst	**Action**
$?_1$	$?_2$

L3_fwd	
ipv4.dst	**Action**
10.0.0.1	set_meta(8)
$?_5$	$?_6$

LAG	
meta	**Action**
8	set_out(8)
$?_3$	$?_4$

(b) Pipeline 2, with a level of metadata indirection, and "holes" filled in. During synthesis, Avenir solves for these unknowns and concludes that $?_1 = \mathsf{ABB28FC}$, $?_2 = \mathsf{set\_meta}(5)$, $?_3 = 5$, $?_4 = \mathsf{set\_out}(5)$.

L2_fwd	
eth.dst	**Action**
ABB28FC	set_m2(5)

L3_fwd	
ipv4.dst	**Action**
10.0.0.1	set_m3(8)

LAG	
m2, m3	**Action**
5, 8	set_out(8)
5, ∗	set_out(5)
∗, 8	set_out(8)

(c) Pipeline 3, which introduces two additional metadata fields.

fwd_table	
eth.dst, ipv4.dst	**Action**
∗, 10.0.0.1	set_port(8)
ABB28FC, ∗	set_port(5)

(d) The second abstract pipeline which implements a "one big table."

$(Pipe_1 \Rightarrow Pipe_2)$

$e \mapsto$ e.table e.keys set_meta(e.out)
LAG e.out set_out(e.out)

$(Pipe_1 \Rightarrow Pipe_3)$

e in L2 $\mapsto$ L2 e.keys set_meta(e.out)
LAG $(e.\text{out}, *)$ set_out(e.out)
e in L3 $\mapsto$ L3 e.keys set_meta(e.out)
LAG $(*, e.\text{out})$ set_out(e.out)
L3 $(r.\text{m1}, e.\text{out})$ set_out($e.out$)
for every existing row r in LAG

(e) Translations from Pipeline 1 to Pipelines 2 and 3.

$(OBT \Rightarrow Pipe_1)$

e if eth.dst $= * \mapsto$ L2 e.ipv4.dst set_out(e.out)
e if eth.ipv4 $= * \mapsto$ L3 e.eth.dst set_out(e.out)
e otherwise $\mapsto$ Failure

$(OBT \Rightarrow Pipe_2)$

e if eth.dst $= * \mapsto$ L2 e.ipv4.dst set_out(e.out)
e if eth.ipv4 $= * \mapsto$ L3 e.eth.dst set_meta(e.out),
LAG $(e.\text{out}, *)$ set_out(e.out)
e otherwise $\mapsto$ Failure

$(OBT \Rightarrow Pipe_3)$

e if eth.dst $= * \mapsto$ L2 e.ipv4.dst set_out(e.out)
e if eth.ipv4 $= * \mapsto$ L3 e.eth.dst set_meta(e.out),
LAG $(*, e.\text{out})$ set_out(e.out)
LAG $(r.\text{m1}, e.\text{out})$ set_out(e.out)
for every existing row r in LAG
e otherwise $\mapsto$ Failure

(f) Translations from "one big table" to Pipelines 1 to 3.

Figure 2: Pipelines used in example scenario.

2 Background and Motivation

As shown in Figure 1, Avenir sits between the controller and the data plane, exposing an interface based on an abstract pipeline to the SDN control plane. It intercepts the control operations, translates them to the target pipeline, and passes results to the switch agent to install on the target device. Note that because Avenir works with an abstract notion of a pipeline, it could be used at multiple levels of abstraction—e.g., to implement a driver for a given switch, an abstraction layer like SAI, or even at higher layers of the SDN controller. Likewise, because Avenir operates on switch-by-switch granularity, it can expose different abstract pipelines for different targets. Avenir's synthesis algorithm is sound and its solutions are formally verified, which eliminates the potential for subtle bugs caused by the inherent complexity of the problem, assuming the specifications are correct. Avenir's algorithm is also complete—i.e., given sufficient time, it is guaranteed to find a correct sequence of target operations if it exists.

Status Quo: Manual Control Plane Mappings. Consider a simple running example based on ONOS that illustrates the need for a control plane synthesis tool. Suppose that each switch implements the simple L2-L3 pipeline in Figure 2a. In this pipeline, the output port is set based on the Ethernet and IPv4 destination addresses in the corresponding tables.

As the network matures, its engineers decide to add additional physical data planes—e.g., to incorporate a new generation of hardware or to avoid vendor lock-in. For instance, the pipeline, shown in Figure 2b, adds a layer of metadata indirection to the physical device to support link aggregation.

To avoid disrupting the control plane, which likely consists of hundreds of thousands of lines of code,[1] the engineers write a *driver* that translates operations written for Pipeline 1 into operations for Pipeline 2. In this case, the driver, shown in Figure 2e, is relatively simple: for each rule, it simply copies the output port into meta and inserts a row into the LAG table effectively copying the value of meta into the output port.

Now, suppose the engineers decide to support a third pipeline (Figure 2c), which sets a separate metadata field in each table. The translation (Figure 2e), is also simple, but requires some care to write—in particular, the L3 table's forwarding decision must always be preferred in the LAG table.

Finally, suppose the engineers want to migrate their original pipeline to a *one big table* abstraction (Figure 2d), similar to OpenFlow. Now, the engineers need to make code changes to all three translations (Figure 2f).

[1] ONOS has currently about 611k lines of Java code [28, 37].

Of course, the ONOS engineers could compose the translations from the one big table to the first pipeline, and on to the other pipelines. However as more and more logical and physical tables are added, managing a complex cascade of translations would become unwieldy, and hard to maintain.

Control Plane Synthesis with Avenir. Avenir improves upon the state of the art—i.e., writing manual translations—by automating the translation of rules from an abstract pipeline to a target pipeline. Of course, the programmer still needs to write programs that capture the behavior of both pipelines, and that's a non-trivial task. But we believe this should be less challenging than actually writing the translations—akin to describing source and target languages vs. writing a compiler.

To see how this is done, let's explore how Avenir translates abstract Pipeline 1 L2 insertions into Pipeline 2 insertions. First, assume, as shown in Figure 2b, that the L3 table is populated with rules that match on the IPv4 address (10.0.0.1) and set the metadata to (8), and the LAG table matches on that metadata and forwards out port 8. Consider inserting a single rule into the abstract Pipeline 1 L2 table that matches on $\mathsf{eth.dst} = \mathsf{ABB28FC}$ and sets the outport to 5. To reflect this update in Pipeline 2, we then need to solve for the unknowns, written as (?) in Figure 2b. These unknowns model the answers to questions like "Which tables need modification?" and "What should the matches/actions/action data be?"

More formally, the unknowns (?) represent a special kind of variable we instrument our program with, called a *hole*. Programs instrumented with holes are called *sketches*. We heuristically search for a valuation of these holes that makes the behaviors of the two pipelines equivalent. In this example, we could set $?_1 = \mathsf{ABB28FC}$, $?_2 = \mathsf{set\_meta}(5)$, $?_3 = 5$, and $?_4 = \mathsf{set\_out}(5)$. Since we do not need to insert a rule in the L3 table, we do not need to find values for these holes. In practice, holes can only be assigned values, not code snippets, like we are doing here for $?_2$ and $?_4$. We will see how to construct these sketches in detail in Section 3.2, and we will introduce our synthesis algorithm in Sections 3.3, 4.1 and 5.2.

As a strawman, we might consider an offline approach, where we synthesize the driver code once-and-for-all that translates any abstract operation into equivalent target operations. However, there are many cases (e.g., Figure 2f) where there is no translation that works for all abstract operations, this synthesis algorithm would fail to produce any solution in many cases where Avenir would succeed. Avenir's online solution allows for a more dynamic and flexible approach.

Incrementality and Optimizations. The key challenge in making Avenir practical is scaling up to handle real-world programs, which typically have at least dozens of tables with thousands of rules. Avenir needs to potentially compute a translation on every abstract control plane operation, so it must be responsive. As another strawman, imagine an approach that computes a full set of table rules on every control plane operation. This strategy might be workable when the tables have only a few rules, e.g., recomputing the existing match in Pipeline 2's L3 table, but it would quickly become a bottleneck if there were say, tens of thousands of rules in L3. Hence, we employ an incremental approach in which we synthesize "deltas" consisting of small batches of control plane operations rather than full tables. By only considering the most recent insertion or deletion into a table, we can often reuse previous solutions and avoid redundant recomputation.

Going a step further, we can cache "templates" derived from previous solutions to help translate future operations. For example, on the next insertion into L2, we can try to reuse the same stucture by inserting into L2_fwd and LAG, with actions set_meta and set_out, forcing the argument to set_meta to equal the LAG table match.

3 Control Plane Synthesis

Our synthesis algorithm is based on CEGIS [40]. The core of CEGIS is a loop with two main components: verification and inductive synthesis. In each iteration of the loop, a candidate implementation is run through the verification component to check correctness. If verification fails, a counterexample trace is produced, allowing the inductive synthesis component to learn from this failure to generate a better candidate. The loop terminates when verification ultimately succeeds.

In our setting, the CEGIS loop is run for each insertion into the abstract pipeline. Inductive synthesis produces candidate control plane implementations for the target pipeline, and verification checks whether the behavior of the two pipelines are equivalent. The rest of this section discusses the CEGIS components in detail. Section 4 discusses optimizations that make this approach efficient and scalable.

3.1 Basic Definitions and Verification

The *verification* component of the CEGIS loop determines whether the synthesized control plane operations implement the same packet-processing behavior on the target pipeline as on the abstract pipeline. We model packets as finite maps from a fixed set of header and metadata fields to bit vectors, and say two packets are equal and write $pkt = pkt'$ if they agree on all header fields. Packets have a direct interpretation as a boolean formula: for headers Hdr and a list $\vec{x} \subseteq \mathsf{Hdr}$, we write $pkt[\vec{x}]$ to mean $\bigwedge_{x \in \vec{x}} x = pkt.x$.

Syntax and Semantics. In Figure 3, we define the syntax of pipelines. A pipeline program is a just a command $c \in \mathsf{Cmd}$, that denotes a packet processing function, which we write $[\![c]\!] : \mathsf{Pkt} \to \mathsf{Pkt}$. Pipeline programs can contain bitvector expressions $e \in \mathsf{Expr}$ and boolean expressions $b \in \mathsf{Bool}$. Given a bitvector $[n]_s$ of length s, we use "wraparound" semantics when values n larger than $2^s - 1$ overflow. We often omit subscripts when s is clear from context or use evocative notation.

There are a few ways to compositionally build a pipeline program. First, fields f can be assigned values via the command $f := e$, which updates the packet pkt to $pkt\{f \mapsto n\}$, where e evaluates to n in pkt. Further, commands can be sequenced, $c_1; c_2$, which first executes c_1 then c_2. We can also

$$
\begin{array}{rcll}
c & ::= & & (c \in \mathsf{Cmd}) \\
& | & f := e & \textit{Assignment(*)} \\
& | & c;c & \textit{Sequence(*)} \\
& | & \mathsf{if} \overrightarrow{b \to c} \ \mathsf{fi} & \textit{Guarded Commands(*)} \\
& | & \mathsf{apply}\ t & \textit{Table Application} \\
a & ::= & & (a \in \mathsf{Act}) \\
& | & \lambda\,(\vec{x}).\ c(*) & \textit{Function} \\
t & ::= & \left\{ \begin{array}{l} \textit{name} : \mathsf{Name}; \\ \textit{keys} : \mathsf{Name}^+; \\ \textit{actions} : \mathsf{Act}^+; \\ \textit{default} : \mathsf{Act} \end{array} \right\} & \textit{Table Definition} \\
\delta,\varepsilon & ::= & & (\delta \in \mathsf{Edit}) \\
& | & \mathsf{A}_x(\rho) & \textit{Insertion} \\
& | & \mathsf{D}_x(n) & \textit{Deletion} \\
\rho & \in & \mathsf{Row} = \mathsf{List}[\mathsf{BitVec}] \times \mathsf{List}[\mathsf{BitVec}] \times \mathbb{N} & \\
\tau,\sigma & \in & \mathsf{Inst} = \mathsf{Name} \to \mathsf{List}[\mathsf{Row}] & \\
v & ::= & [n]_n & \textit{Bitvector}\ (v \in \mathsf{BitVec}) \\
h & ::= & \left\{ \begin{array}{l} \textit{name} : \mathsf{Name}; \\ \textit{width} : \mathbb{N} \end{array} \right\} & \textit{Header Field}\ (h \in \mathsf{Hdr}) \\
m & ::= & \left\{ \begin{array}{l} \textit{name} : \mathsf{Name}; \\ \textit{width} : \mathbb{N} \end{array} \right\} & \textit{Metadata Field}\ (h \in \mathsf{Meta}) \\
f & \in & \mathsf{Hdr} \cup \mathsf{Meta} & \\
x & \in & \mathsf{Name} & \\
n & \in & \mathbb{N} &
\end{array}
$$

Figure 3: Pipeline syntax. Actions vary under starred variants

use conditional control flow, written $\mathsf{if}\ b_1 \to c_1\ \ldots\ b_n \to c_n\ \mathsf{fi}$, which executes command c_i on the incoming packet *pkt* for the smallest-indexed b_i that evaluates to tt on *pkt*. These conditionals are similar to Dijkstra-style guarded commands [9]. Finally, table application $\mathsf{apply}(t)$ executes match-action table t. Tables are represented as records, where $t.\textit{name}$ is table's name; $t.\textit{keys}$ is a list of packet headers referred to by name; $t.\textit{actions}$ is the list of actions (which are lexically-scoped anonymous functions $\lambda(\vec{x}).c$); and $t.\textit{default}$ is the command that is executed when the table is missed. Only certain commands c may occur inside an action (denoted with a $(*)$ in Figure 3)—e.g., table application is not allowed.

Notice that tables have no way of referring to their entries. To represent entries in a table t, we maintain a *table instantiation* $\tau : \mathsf{Name} \to \mathsf{List}[\mathsf{Row}]$, alongside the syntactic pipeline, which maps table names to their row lists. We write Inst for the set of all instantiations. We refer to the pair (c,τ) as the *pipeline state*. A row $\rho \in \mathsf{Row}$ is a triple $\rho = (\vec{m},\vec{d},a)$, where $\vec{m}$ are matches, a is the action index and $\vec{d}$ is the action data.

We can define a source-to-source syntactic transformation $\mathsf{subst}(c,\tau)$ that replaces every occurence of $\mathsf{apply}(t)$ in c with a guarded command encoding the rows of the table $\vec{\rho} = \tau(t.\textit{name})$, as follows, where the ith row $\rho_i = (\vec{m}_i,\vec{d}_i,a_i)$:

$$
\mathsf{if}\ \left(\begin{array}{ccl}
t.\textit{keys} = \vec{m}_0 & \to & t.\textit{action}[a_0](\vec{d}_0) \\
& \cdots & \\
t.\textit{keys} = \vec{m}_n & \to & t.\textit{action}[a_n](\vec{d}_n) \\
\mathsf{tt} & \to & t.\textit{default}
\end{array} \right)\ \mathsf{fi}
$$

HOLE	DESCRIPTION
$?\mathsf{Del}_{t,i} = 1$	Delete row i in table t
$?\mathsf{Add}_{t,j} = 1$	Add j rows to table t
$?\mathsf{Act}_{t,j} = i$	New Row j in table t (if added), has action i
$?\mathsf{k}_{t,j} = v$	New Row j in table t (if added), matches header k with v
$?\mathsf{d}_{t,j,i} = v$	New Row j in table t (if added with action i) has action data for parameter d set to v

Figure 4: Summary of holes used in sketching.

We say that a row $(\vec{m},\vec{d},a)$ is well-defined for a table t when $|\vec{m}| = |t.\textit{keys}|$, $a < |t.\textit{actions}|$, and for the parameters $\vec{x}$ of $t.\textit{actions}[a]$, $|\vec{d}| = |\vec{x}|$. Further, an instance is well-defined when all of its rows are well-defined for their tables, and a command is well-defined when no two tables have the same name. We assume that commands and instantiations are well-defined, and that there are no bit-width mismatches: both are easy to check statically.

Finally, we have control plane edits ($\delta \in \mathsf{Edit}$), which are operations that allow the control plane to modify table instantiations. We interpret them as functions, i.e., $\delta(\tau) \in \mathsf{Inst}$. There are two kinds of edits: insertions and deletions. For a given instance τ, an insertion $\mathsf{A}_x(\rho)(\tau)$ appends ρ to the end of $\tau(x)$ (meaning it has the lowest priority). If $\tau(x)$ has a row ρ' with the same matches as ρ, the inserted row is dropped. A deletion $\mathsf{D}_x(i)(\tau)$ removes the ith element from $\tau(x)$.

Now that we know how to interpret pipelines as functions, we say $c_1 = c_2$ when they are functionally equivalent. To check this condition, we use predicate transformer semantics to generate a verification condition [13], written $c_1 \equiv c_2$, which we check using an SMT solver, by running $\mathsf{CheckSat}(c_1 \not\equiv c_2)$. If the solver returns UNSAT, we conclude the programs are equivalent. Otherwise, it returns SAT as well as a model that encodes a counterexample χ—i.e., an input and output packet pair χ that demonstrates different behavior in the abstract and physical programs, writing χ_0 and χ_1 for the input and output packets respectively. It is easy to prove that this validity check implies functional equivalence.

Theorem 1. *For every pair of pipelines c_1, c_2, if $c_1 = c_2$ then $\mathsf{CheckSat}(c_1 \not\equiv c_2) = \mathsf{UNSAT}$, and if $c_1 \neq c_2$ then $\mathsf{CheckSat}(c_1 \not\equiv c_2) = \mathsf{Sat}\ \chi$.*

Proof. By soundness of verification conditions with respect to the denotational semantics of guarded commands [9, 13]. □

3.2 Synthesizing Candidates via Sketches

To propose new candidate programs for verification, we use a technique called Sketching [41]. A *sketch* is a command containing special variables called *holes*. Aside from holes for values (i.e., ?k for match keys and ?d for action data), which we introduced in Section 2, we also need holes for table entries, corresponding to deletions (?Del), insertions (?Add) and action choice (?Act). The meaning of these holes is described in Figure 4.

$$\mathsf{fix\_cex}(p,\sigma,\chi,n,\vec{x}) \triangleq \chi_0[\vec{x}] \Rightarrow wp\,(\mathsf{instr}(p,\sigma,\cdot,n),\chi_1[\vec{x}])$$
$$\mathsf{model}(p,\sigma,X) \triangleq \mathsf{CheckSat}\left(\forall \vec{x}.\, \bigwedge_{\chi\in X} \mathsf{fix\_cex}(p,\sigma,\chi,|X|,\vec{x})\right)$$

Figure 5: The model function. In the above, the vector $\vec{x}$ is all of the non-hole variables that occur in the formula.

To compute a candidate solution in our CEGIS loop, we first instrument the program with holes. We write $\mathsf{instr}(c,\tau,\vec{\delta},n)$ to describe the program $\vec{\delta}(\tau(c))$ with deletion holes for every row in τ, and holes for n row insertions. We do not add deletion holes for insertions in $\vec{\delta}$, which is crucial for the completeness theorem (Section 4). We lift this function from tables to programs in the obvious way.

Consider the L2 table from pipelines 1 and 2. To instrument it with holes, allowing for a single insertion, we would insert a deletion hole for the existing rule and a single row of insertion holes, yielding the following sketch:

	Match(eth.dst)	**Action**
?Del $= 0$	ABB28FC	set_out(5)
?Add $= 1$	?eth.dst	if ?Act $= 0 \rightarrow$ set_out(?p) ?Act $= 1 \rightarrow$ drop() fi

A possible model for these holes that matches the destination MAC address with $00:00:00:00:00:00$ and drops the packet, is $\{?\mathsf{Del} \mapsto 0, ?\mathsf{eth.dst} \mapsto 0, ?\mathsf{Act} \mapsto 1\}$. Note that ?p is irrelevant, so we omit it from the model.

Of course, sketches represent a vast search space of edits: every existing table row can be deleted, and up to n rows can be inserted. Blindly searching through this space would not scale in practice. Instead, we learn from counterexamples to help guide the search toward a solution.

3.3 Counterexample-Guided Search

When the solver determines that a proposed candidate pipeline is not equivalent to the abstract pipeline, it generates a counterexample χ that encodes an input-output packet pair. This pair corresponds to a behavior of the abstract switch that is not replicated in the candidate or vice versa. We can use this counterexample to guide our search. More formally, we use the weakest precondition $wp(c,\varphi)$ whose satisfying models are inputs that, after executing c, yield outputs satisfying φ.

The fix_cex function constructs the formula $\chi_0[\vec{x}] \Rightarrow wp(s,\chi_1[\vec{x}])$ for the sketch $s = \mathsf{instr}(p,\sigma,\cdot,|X|)$. The formula identifies edits that when applied to the physical pipeline state (p,σ) produce the input-output behavior indicated by χ.

The function model in Figure 5 lifts fix_cex over all counterexamples X that have been seen so far. Notice that we only instrument the physical pipeline with $|X|$ insertion holes since each counterexample hits at most one rule in each table.

3.4 Synthesis Algorithm

The full synthesis algorithm is presented in Figure 6. Given a abstract pipeline l, a target pipeline p, an abstract table instantiation τ, a target table instantiation σ, a sequence of physical edits $\vec{\delta}$, and a set of counterexamples X, $\mathsf{cegis}(l,p,\tau,\sigma,\vec{\delta},X)$ produces a sequence of edits $\vec{\varepsilon}$ such that $\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\sigma))$ if one exists. We initially call the algorithm with $\vec{\delta} = []$ and $X = \{\}$. First, we call the SMT solver to check for equality. If the programs are equal, we are done, and return $\vec{\delta}$. Otherwise, we get a counterexample χ and solve for new edits by augmenting X with χ, applying the edits to the target pipeline and calling model. If it returns UNSAT, there is no way to make the programs equivalent and we fail. Otherwise, we get a new sequence of edits and keep searching.

```
cegis(l, p, τ, σ, δ⃗, X) ≜
  match CheckSat(subst(l, τ) ≢ subst(p, δ⃗(σ))) with
    | UNSAT → Ok δ⃗
    | SAT →
        match model(p, δ⃗(σ), {χ} ∪ X) with
          | UNSAT → Fail
          | SAT δ⃗′ → cegis(l, p, τ, σ, δ⃗′, {χ} ∪ X)
```

Figure 6: Simple Algorithm for Control Plane Synthesis.

3.5 Formal Properties

Next we establish two formal properties for our synthesis algorithm: soundness and completeness. Soundness means that synthesized target operations produce equivalent behavior.

Theorem 2 (Soundness)**.** *For every* $l,p \in \mathsf{Cmd}$, $\tau,\sigma \in \mathsf{Inst}$, $\vec{\delta} \in \mathsf{List[Edit]}$, *and* $X \subseteq [\![\mathsf{subst}(l,\tau)]\!] \cap [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$ *if* $\mathsf{cegis}(l,p,\tau,\sigma,\vec{\delta},X) = \mathsf{Ok}\,\vec{\varepsilon}$ *then* $\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\sigma))$.

Proof. Follows from Theorem 1. □

Completeness says that if a solution exists, then our synthesis algorithm will (eventually) find it.

Theorem 3 (Completeness)**.** *For every* $l,p \in \mathsf{Cmd}$, $\tau,\sigma \in \mathsf{Inst}$, $\vec{\delta} \in \mathsf{List[Edit]}$, *and* $X \subseteq [\![\mathsf{subst}(l,\tau)]\!] \cap [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$, *if* $\exists \vec{\delta}' \in \mathsf{List[Edit]}.\ \mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}'(\sigma))$ *then* $\exists \vec{\delta}'' \in \mathsf{List[Edit]}.\ \mathsf{cegis}(l,p,\tau,\sigma,\vec{\delta},X) = \mathsf{Ok}\ \vec{\delta}''$ *and* $\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}''(\sigma))$.

Proof. By induction on the size of $\mathsf{Pkt} \setminus \pi_1(X)$. □

Limitations The main limitation of this first synthesis algorithm is that the number of queries is bounded by the number of counterexamples—i.e., every possible packet. Given an MTU of n, there could be as many as 2^n packets.

4 A Scalable Solution: Incremental Synthesis

To obtain a scalable synthesis algorithm, we first exploit the insight that the control plane operates in an incremental fashion—i.e., before each control plane operation, the data

	Match(eth.dst)	**Action**
$?\mathsf{Del}_0 = 0$	ABB28FC	set_out(5)
$?\mathsf{Add}_0 = 1$	$?\mathsf{eth.dst}_0$	if $?\mathsf{Act}_0 = 0 \rightarrow \mathsf{set\_out}(?\mathsf{p}_0)$ $?\mathsf{Act}_0 = 1 \rightarrow \mathsf{drop}()$ fi
$?\mathsf{Add}_1 = 1$	$?\mathsf{eth.dst}_1$	if $?\mathsf{Act}_1 = 0 \rightarrow \mathsf{set\_out}(?\mathsf{p}_1)$ $?\mathsf{Act}_1 = 1 \rightarrow \mathsf{drop}()$ fi

	Match(ip.dst)	**Action**
$?\mathsf{Del}_2 = 0$	10.0.0.1	set_out(8)
$?\mathsf{Del}_1 = 0$	8.8.8.8	set_out(47)
$?\mathsf{Add}_3 = 1$	$?\mathsf{ipv4.dst}_3$	if $?\mathsf{Act}_2 = 0 \rightarrow \mathsf{set\_out}(?\mathsf{p}_2)$ $?\mathsf{Act}_2 = 1 \rightarrow \mathsf{drop}()$ fi
$?\mathsf{Add}_3 = 1$	$?\mathsf{ipv4.dst}_3$	if $?\mathsf{Act}_3 = 0 \rightarrow \mathsf{set\_out}(?\mathsf{p}_4)$ $?\mathsf{Act}_3 = 1 \rightarrow \mathsf{drop}()$ fi

(a) Basic Sketch: Satisfiable for packets that hit L2's first row and L3's second.

	Match(eth.dst)	**Action**
$?\mathsf{Del}_0 = 0$	ABB28FC	set_out(5)
$?\mathsf{Add}_0 = 1$	$?\mathsf{eth.dst}_0$	if $?\mathsf{Act}_0 = 0 \rightarrow \mathsf{set\_out}(?\mathsf{p}_0)$ $?\mathsf{Act}_0 = 1 \rightarrow \mathsf{drop}()$ fi

	Match(ip.dst)	**Action**
$?\mathsf{Del}_2 = 0$	10.0.0.1	set_out(8)
	8.8.8.8	set_out(47)
$?\mathsf{Add}_1 = 1$	$?\mathsf{ipv4.dst}_1$	if $?\mathsf{Act}_1 = 0 \rightarrow \mathsf{set\_out}(?\mathsf{p}_1)$ $?\mathsf{Act}_1 = 1 \rightarrow \mathsf{drop}()$ fi

(b) Incremental Sketch: Unsatisfiable for packets that hit L2's first row and L3's second, which triggers backtracking, remembering that the previously-synthesized edit was incorrect.

Figure 7: Examples of basic and incremental sketches for Pipeline 1.

planes are already equivalent, so we only need to handle incremental changes to the abstract program, such as adding or deleting a rule. In the common case, we do not have to resynthesize all of the previously generated rules. However, some care is needed as certain control plane operations do require deleting previously installed rules.

4.1 Single Counterexample-Guided Search

Our first enhancement to the basic synthesis algorithm is to only add insertion holes to solve for the most recent counterexample, and only add deletion holes for state that existed before synthesis began, which greatly reduces the number of holes we need to produce as we explore the space. Instead of instrumenting the program with insertion holes for every counterexample, we only do it for the most recent one.

Consider again the L2 and L3 tables from pipelines 1 with the initial state depicted in Figure 2a. We want to synthesize edits that send Ethernet packets that miss in the L2_fwd with destination DECAFBAD out on port 47. Suppose the first counterexample has input packet $\chi_0 = \{\mathsf{eth.dst} \mapsto \mathsf{DECAFBAD}, \mathsf{ipv4.dst} \mapsto 8.8.8.8\}$, and output packet $\chi_1 = \chi_0\{\mathsf{out} \mapsto 47\}$. Let's say on the first iteration we produce the (incorrect) edit to L2_fwd that maps $\mathsf{ipv4.dst} = 8.8.8.8$ to $\mathsf{set\_out}(47)$, and the verification step will provide a new counterexample.

Suppose the next counterexample has input packet $\chi'_0 = \{\mathsf{eth.dst} \mapsto \mathsf{ABB28FC}, \mathsf{ipv4.dst} \mapsto 8.8.8.8\}$, and output packet $\chi'_1 = \chi'_0\{\mathsf{out} \mapsto 5\}$. Now the simple algorithm will produce the sketch in Figure 7a, which can be solved by deleting the already inserted row ($?\mathsf{Del}_1 = 1$) and adding the single required row to the L2 table ($?\mathsf{Add}_0 = 1$, $?\mathsf{eth.dst}_0 = \mathsf{DECAFBAD}$, $?\mathsf{Act}_0 = 0$, $?\mathsf{p}_0 = 47$, and remaining Add/Del holes disabled).

In contrast, the incremental search will first create the unsatisfiable sketch shown in Figure 7b. There is no way to fill its holes to satisfy the above counterexample. We backtrack with the knowledge that $?\mathsf{ipv4.dst} \neq 8.8.8.8$ and attempt to solve the original sketch with respect to the original counterexample, and the only remaining solution is correct.

First, notice that the final simple sketch uses 21 holes, whereas each incremental sketch uses only 10. On the other hand, the incremental search sends 3 sketches to the solver as opposed to the simple search, which only sends 2. Why do we want to send *more* queries to Z3 instead of less? This is a result of the NP-completeness of SAT/SMT solving. Solving more formulae with fewer variables is often faster than solving fewer formulae with more variables. Here, the search space size for the 3 incremental sketches is approximately $3 \cdot |B|^{10}$, whereas for "simple" query it is approximately $|B|^{21}$, where $|B|$ is the size of the bitvector domain.

Further, observe that the incremental sketches we send will *always* have 10 holes, independent of the number of counterexamples, whereas the simple sketch will continue to add holes as the number of counterexamples grows.

We formalize this new incremental model-finding function model' in Figure 8. It is defined in term of a satisfiability check for a conjunction of three sub-formulas. The first conjunct uses a modified fix_cex function that instruments the program with one addition hole per table. The second conjunct, φ, limits the search by preventing models from reoccurring. The final conjunct is a search oracle $\mathsf{HEURISTIC}()$ that computes restrictions on the search space. The only constraints on $\mathsf{HEURISTIC}()$ are that it must not add covered rules or previously-deleted rules (to avoid looping), and it must not permanently preclude any solution (to ensure completeness).

$$\mathsf{fix\_cex}(p,\sigma,\vec{\delta},\vec{x},\chi) \triangleq pkt[\vec{x}] \Rightarrow wp(\mathsf{instr}(p,\sigma,\vec{\delta},1),pkt'[\vec{x}])$$

$$\mathsf{model}'(p,\sigma,\vec{\delta},\chi,\varphi) \triangleq \mathsf{SAT}\left(\begin{array}{l}\forall\vec{x}.\mathsf{fix\_cex}(p,\sigma,\vec{\delta},\vec{x},\chi))\\ \wedge\ \varphi \wedge \mathsf{HEURISTIC}()\end{array}\right)$$

Figure 8: The model′ function computes edits to physical state (p,σ) to accomodate the counterexample χ. The oracle soundly restricts the search space.

```
cegis(l, p, τ, δ, σ) ≜ verify(l, p, δ(τ), σ, [])
verify(l, p, τ, σ, δ⃗) ≜
  match CheckSat(subst(l, τ) ≢ subst(p, δ⃗(σ))) with
    | UNSAT → Ok δ⃗
    | SAT χ → solve(l, p, τ, σ, δ⃗, χ, tt)
solve(l, p, τ, σ, δ⃗, χ, φ) ≜
  match model′(p, σ, δ⃗, χ, φ) with
    | UNSAT → Fail
    | SAT δ⃗′ →
      match verify(l, p, τ, σ, δ⃗ ∘ δ⃗′) with
        | Ok δ⃗″ → Ok δ⃗″
        | Fail → solve(l, p, τ, σ, δ⃗, χ, φ ∧ ¬δ⃗′)
```

Figure 9: The incremental backtracking CEGIS algorithm.

4.2 Incremental Synthesis Algorithm

We present our incremental synthesis algorithm in Figure 9. It comprises two mutually recursive functions: verify, which checks the verification condition and solve, which generates new models. Both functions take the same arguments: the abstract and target programs and instantiations ((l,τ) and (p,σ) respectively), and a sequence of edits to the target program $\vec{\delta}$. They either return Ok $\vec{\delta}'$, where $\vec{\delta}$ is the prefix of $\vec{\delta}'$ and $\mathsf{CheckSat}(\mathsf{subst}(l,\tau) \not\equiv \mathsf{subst}(p,\vec{\delta}'(\sigma))) = \mathsf{UNSAT}$, or Fail, if there is no such $\vec{\delta}'$. The cegis function is the "main" method. It takes the abstract and target pipelines (l and p) and instantiations (τ and σ) as arguments, as well as the abstract edit δ. It then applies δ to τ and invokes verify with no target edits.

The verify function resembles the cegis function of Section 3. It first checks whether the programs are equal, and if so returns Ok $\vec{\delta}$. Otherwise it calls solve with an initial counterexample χ and an unrestricted model, which searches for an edit to make the programs equivalent.

The solve function takes the standard arguments, with the addition of a counterexample χ and the model space restriction formula φ, which keeps track of failed solutions for χ, to prevent repetition. First, model′ searches for a target edit that corrects the behavior for the counterexample. If none exists, we return Fail, indicating that there is no sequence of equivalent target edits with the prefix $\vec{\delta}$. Otherwise, model′ provides a model $\vec{\delta}'$. In this case we extend the running sequence of edits to $\vec{\delta} \circ \vec{\delta}'$ and call back to verify. If successful, we return the result, otherwise we preclude $\vec{\delta}'$ from the space of possible models φ (writing $\neg\vec{\delta}'$ for the negation of valuations that produce $\vec{\delta}'$.) Then we recursively call solve and continue searching within this restricted space of models.

4.3 Formal Properties

We prove that the incremental algorithm is also sound and complete. As with the simpler algorithm, the proof of soundness follows by the correctness of the verification condition.

Theorem 4 (Incremental Soundness)**.** *For every* $l,p \in \mathsf{Cmd}$, $\tau,\sigma \in \mathsf{Inst}$, $\delta \in \mathsf{Edit}$, $X \subseteq [\![\mathsf{subst}(l,\tau)]\!] \cap [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$, *if* $\mathsf{cegis}(l,p,\tau,\sigma,\delta,X) = \mathsf{Ok}\ \ \vec{\varepsilon}$, *then* $\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\sigma))$.

Proof. Again, the result follows from Theorem 1. □

As in the simple synthesis algorithm, incremental completness relies on the finite domain, which here is the product of two finite domains: (1) sequences of reachable edits that do not redundantly add and delete a rule, and (2) the number of valuations for the holes introduced by the instr function.

Theorem 5 (Incremental Completeness)**.** *For every abstract program* l*, target program* p*, abstract instantiation* τ*, target instantiation* σ *and abstract edit* δ *if* $\exists\vec{\varepsilon} \in \mathsf{List}[\mathsf{Edit}].\ \mathsf{subst}(l,\delta(\tau)) = \mathsf{subst}(p,\vec{\varepsilon}(\sigma))$ *then* $\exists\vec{\delta}' \in \mathsf{List}[\mathsf{Edit}].\ \mathsf{cegis}(l,p,\tau,\delta,\sigma,[]) = \mathsf{Ok}\ \vec{\delta}'$ *and* $\mathsf{subst}(l,\delta(\tau)) = \mathsf{subst}(p,\vec{\delta}'(\sigma))$.

Proof. By strong outer induction on the size of the reachable non-deleting edit sequences, and strong inner induction on the (lexicographically ordered) size of the counterexample set and the number of models in each model space. □

Theorem 5 proves that Avenir translates abstract operations given unbounded resources. In practice, Avenir's effectiveness relies on heuristics and optimizations.

5 Heuristics and Optimizations

Avenir offers a number of heuristic optimizations designed to help it scale to larger networks. Interestingly, these optimizations need not be sound. We introduce a run-time check for soundness and revert the optimization if it fails. We focus on two classes of optimizations: verification and model finding.

5.1 Exploiting Incrementality

The key to scalable synthesis is to adopt an incremental approach and focus on edits, while incorporating further optimizations within the verification and synthesis steps.

Fast Counterexamples. In the incremental setting, we know that a new abstract insertion δ must be the cause of any semantic difference with the target pipeline. We symbolically compute packets that hit δ via an SMT query that gives us a potential counterexample packet *pkt*. We use the denotational semantics to check whether *pkt* is a real counterexample. If *pkt* doesn't induce different behavior we retry the query (in

practice 10 times) until we either obtain a true counterexample, or resort back to the standard equivalence check.

Program Slicing. We leverage the incrementality assumption to use program slicing to verify only the part of the program that changes. This isn't always sound, so we check that the abstract edits are reachable iff the target edits are. We also have a faster and stronger constraint that checks that the abstract and target matches are disjoint from the extant rules. If both conditions fail, we run the full equivalence check. In practice, slicing composes with constant propagation and dead code elimination to normalize the queries.

Query Templates. The queries produced using program slicing are often syntactically similar. So when we see two validity queries that only differ in their specific concrete values, we try to abstract those concrete values into a universally quantified variable. We then check whether that more-general query is valid. If it is, we add it to a cache of templates, otherwise we continue in a CEGIS loop by negating the valuation of the quantified variables and trying again. Whenever we get future queries that are instances of the template, we can return VALID without having to consult the SMT solver.

Translation Templates. As with queries, we can cache translations of operations by generalizing over their concrete values to obtain a template. The template observes the way that concrete values are mapped from previously-seen abstract insertions into their equivalent target insertions, and structurally replicates that mapping on the new abstract insertion. It also observes the cache of translations for differing constants and generates unused constants for new rules which optimizes for metadata patterns like in Figures 2b and 2c. Note that before adding a solution to the cache, Avenir optionally reduces its size, by heuristically removing superfluous target edits, which improves the generality of the solution. When no template applies, Avenir relies on a heuristic-guided search.

5.2 Model-Finding Heuristics

Now we describe the implementation of the HEURISTIC() oracle, which abstracts a combination of heuristics. In our formalization, we assume that the heuristics are always complete. However in practice, many of Avenir's individual heuristics are not; when a given combination fails, we disable some and try again with a different combination. This search through the heuristics is currently hard-coded, but we plan to support user control of the search strategy and custom heuristics. We describe the heuristics useful in our experiments here.

Ternary and Optional Matching. In the previous sections, we only inserted holes to generate exact matches. We can generate ternary matches for a match key k, which allows us to represent, say, a wild-carded IPv4 source address in only a single row (rather than 2^{32} exact-match rows). To do this, we generate a pair of holes ?k and $?k_{\texttt{mask}}$ and encode the match as $k\&?k_{\texttt{mask}} = ?k$. To eliminate duplicate keys we also enforce the constraint $?k\&?k_{\texttt{mask}} = ?k$. For optional matches, we restrict the masks to be all 1s or all 0s.

Exact and Mask Hints. When a row is inserted into the abstract pipeline, the non-wildcarded keys K of that row are likely relevant in classifying packets. So, we force the relevance of matches on fields in K, either by copying the abstract match values into the target edits (which is very optimistic), or by forcing their masks (if masking is enabled) to be all 1s.

Action Hints. Given a counterexample (pkt_0, pkt_1), we can observe the variables that change in the abstract program, i.e., $\Delta = \{x \mid pkt_0.x \neq pkt_1.x\}$, and ensure that all edits have actions that can influence the value of some variable in Δ.

Other Optimizations. Our final collection of optimizations are based on intuitive heuristics that arise often in practice.

- **Reachable Adds.** We force synthesized models to be reachable using the counterexample driving the search.
- **Prefer Adds.** We try to find a solution that does not require deleting existing rules.
- **Prefer Non Zero Models.** We enforce $?k \neq 0 \neq ?d$ for all key and data holes, unless they are wildcarded.
- **Bounded Edits.** We restrict the search space so that backtracking is triggered beyond specified limits.
- **Previous Counterexamples.** We try to synthesize rules that do not violate previously-seen counterexamples.

6 Implementation

We implemented Avenir in approximately 11K lines [37] of OCaml code that interfaces with Z3 [8]. Our implementation accepts a description of an abstract and a target pipeline, sequences of insertions to both programs (to construct the initial state), as well as a sequence of abstract edits to synthesize. Avenir then produces a sequence of edits to the target program (or fails if no such sequence exists). All of the optimizations described in 5 are configurable as command line flags. In our implemention, we use an efficient encoding of the weakest precondition [13], which has linear size for the programs in our internal syntax.

P4 Program Encoding. The front-end of our implementation supports a large subset of P4, via an encoding from P4's control blocks into Avenir's internal syntax. This translation resembles previous work on verifying P4 programs [21]. Of course, P4 is a larger language than Avenir's syntax. We support more complex P4 language constructs by desugaring them into sequences of internal commands.

We currently assume that all of the data plane programs use the same parser and headers. Hence, in cases where a mapping only exists due to invariants enforced by the parser—e.g., that a packet cannot simultaneously have IPv4 and IPv6 headers—these assumptions must be manually encoded as annotations. We also ignore match kinds and assume all matches are either exact, ternary or optional, depending on command line flags. Finally, we manually encode certain device-specific behaviors

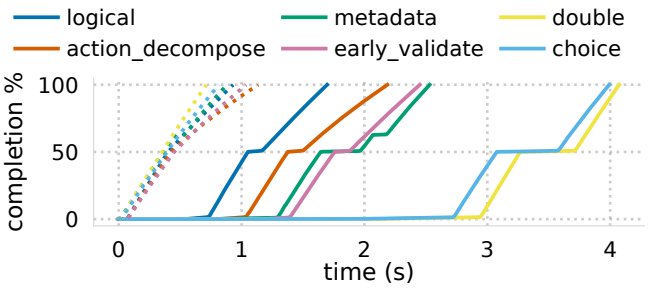

Figure 10: Retargeting case study: solid lines show cold-start completion %; dotted lines show hot-start completion %.

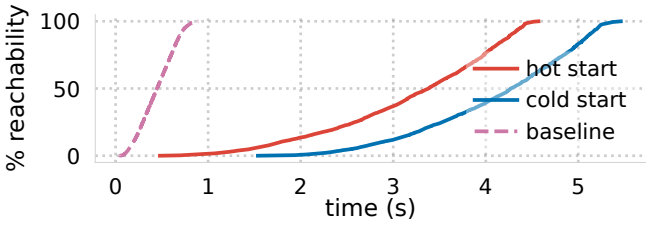

Figure 11: Proportion of all pairs of 64 hosts connected in a star topology that have completed a successful IPv4 ping.

such as the initial value fields and the drop port value. Our implementation is on GitHub[2] under an open-source license.

7 Evaluation

To evaluate Avenir, we demonstrate its functionality under a variety of synthetic and realistic scenarios, and measured its performance against hand-written baselines. First, we show how Avenir can automatically retarget a given abstract pipeline to multiple target pipelines (Section 7.1). Second, we pass packets through the Bmv2 software switch using the generated rules, which both shows they are correct and quantifies Avenir's performance when installing rules for multiple hosts (Section 7.2). Third, we present a case study consisting of a realistic workload drawn from the Trellis data center fabric, running on top of the ONOS SDN controller [2, 29] (Section 7.3). Finally, we study Avenir's scalability via a suite of microbenchmarks (Section 7.4). Our evaluation pays particular attention to the caches, as these are particularly important to obtain good performance.

Summary of Results. Overall, our evaluation shows that, in a variety of cases, Avenir can translate large numbers of rules efficiently. The retargeting, emulation, and ONOS experiments show that Avenir is effective at mapping to and from a variety of programs, and demonstrate that the caching optimizations are highly effective at reducing overheads.

7.1 Retargeting Study

Avenir allows operators to expose a single pipeline abstraction to the control plane, while implementing the forwarding logic over a myriad of physical devices. We demonstrate this use case via a retargeting study, where we retarget an initial program onto a variety of different target pipelines.

The logical program logical.p4 is a simple L2-L3 pipeline followed by a PUNT table that performs packet validation on all headers and metadata. We describe 5 additional target pipelines in terms of the changes to logical.p4:

(early_validate.p4) Replaces the PUNT table of logical.p4 with an ACL that can only match on addresses. Adds a validation table prior to the L2 table that matches on the validity of IPv4 and the TTL field and conditionally applies the rest of the pipeline.

(action_decomp.p4) Decomposes the L3 table into two tables, (1) a forward table that matches on the IPv4 destination and sets the output port, (2) a rewrite table that matches on the IPv4 destination and performs MAC rewriting.

(metadata.p4) Instead of setting the output port, the L2 and L3 tables set a metadata field. This metadata field is mapped to the output port in the nexthop table, which is applied between the L2 and L3 tables.

(double.p4) Applies all three tables in the pipeline twice.

(choice.p4) Introduces a staging table that sets a metadata variable to select between copies of the abstract pipeline.

We used Avenir to translate 1,001 logical.p4 insertions (1 into PUNT for TTL checking, 500 into L2 for Ethernet destination forwarding, and 500 into L3 for IPv4 destination forwarding and MAC rewriting). We show completion graphs for each target in Figure 10.

There are a few things to notice. Every line has an "elbow" at the 50% mark on the y-axis, after which the slope decreases. This represents the transition between parts of the workload. The L3 insertions are slower, because the L2 table is already populated with 500 rules, and slicing has to deal with larger tables. Further, these rules may cause the query template cache to miss: the second "elbow" on the metadata line indicates where the query cache's synthesis engine was able to successfully abstract.

To further demonstrate the power of our template caches, we compare our "cold-start" synthesis (solid lines), where the caches are empty, with "hot-start" synthesis (dotted lines), where the caches are fully populated. We achieve this by running Avenir on the same data twice, without resetting the caches, and logging performance for the second run. The massive performance increase is seen in Figure 10. Network operators concerned with nondeterministic runtimes associated with synthesis can manually populate their caches.

7.2 Network Emulation

We use Avenir to program the entries of a programmable software switch (bmv2) running in a network emulator (mininet). We configure 64 hosts in a star topology connected to a single

[2] Available at https://github.com/cornell-netlab/avenir

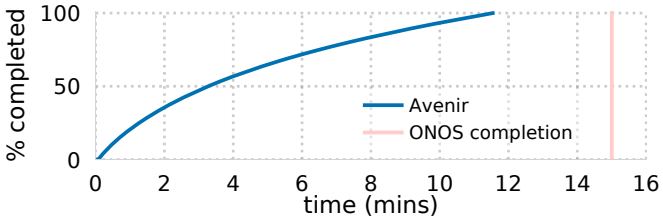

Figure 12: Completion graph for mapping 40k fabric.p4 IPv6 route insertions onto bcm.p4; ONOS takes around 15 min.

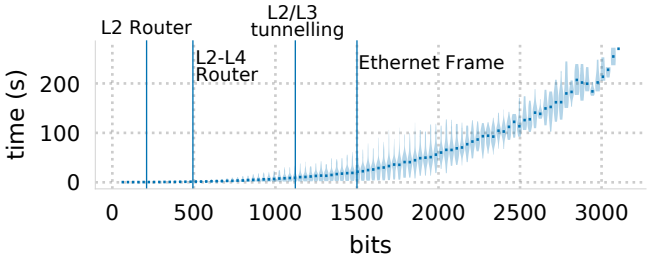

Figure 13: Program bits vs time to translate 100 edits. The vertical lines estimate the sizes of common router programs.

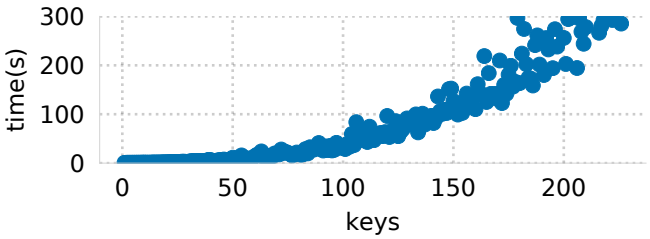

Figure 14: Classifier Scaling. We fixed the number of 32-bit output variables to 8, and varied the number of keys.

switch, and install rules to establish all-pairs ping connectivity. The P4 program running on the software switch is the simple_router.p4 program from the Bmv2 repository. The abstract program is a modified version that joins together the L3 rewriting and forwarding tables into one.

We generate rules required to establish all-pairs connectivity into the logical program and use Avenir to synthesize the equivalent edits into simple_router.p4. We then report the time of the first successful ping between each pair of hosts. We compare Avenir cold-cache run with a manually generated sequence of rule insertions and a pre-populated hot-cache, the results are depicted in Figure 11.

7.3 Case Study: Trellis & ONOS

Trellis [46] is a set of production-grade SDN apps running on ONOS [2,29] to provide control plane logic for multi-purpose L2/L3 leaf-spine fabrics of OF-DPA Broadcom switches. Internally, Trellis uses an ONOS API called FlowObjective, designed to allow portability of apps across different switches by abstracting common L2/L3 functionalities. Trellis controls switches by writing FlowObjectives, which are translated by an ONOS driver into OpenFlow messages for OF-DPA tables. Finally, OF-DPA translates OpenFlow messages to Broadcom SDK calls to populate ASIC-specific tables.

We evaluated Avenir on real-world P4 programs that represent the outermost layers of the architecture described above. The fabric.p4 [12] P4 program was created by the ONOS developers to support Trellis on programmable switches. It is designed to simplify control plane operations, and for this reason it closely resembles the FlowObjective API. Likewise, bcm.p4 [27] abstracts tables from the Broadcom SDK, and was created for Stratum [45], an open source switch agent that uses P4 to model control APIs.

We then collected 40k IPv6 route insertions into fabric.p4 corresponding to a switch reboot load test designed by ONOS engineers. Avenir synthesized insertions into bcm.p4 that equivalently process the IPv6 header and egress specification.

Since Avenir does not process the parser, we simulated its behavior by manually setting the validity bit of the IPv6 header to true, and the IPv4 and MPLS headers to false. Further, the P4 specification [7] leaves the initial values of metadata headers undefined; we manually zero-initialize the metadata fields (a behavior that can be specified for many P4 targets via a compile time flag).

Further, we modified the l3_fwd in bcm.p4 by swapping the IPv6 matches for IPv4 matches; otherwise there wouldn't have been a valid translation. Finally since Avenir works with parsed headers, we systematically renamed headers in bcm.p4 to match fabric.p4.

The results are shown in Figure 12. According to its engineers, ONOS computes and installs these 40k IPv6 routes over a period of about 15 minutes. This figure includes Trellis' route computation logic, the translation itself and the installation of rules onto the physical target devices. Figure 12 shows that Avenir translates these 40k routes into bcm.p4 pipeline in just under 12 minutes. However, it is unclear what conclusions to draw about overhead, because we don't know how ONOS' translation logic performs. In the (unlikely) best case, we would have no overhead. In the (also unlikely) worst case, we would nearly double the runtime. The real performance would likely be somewhere between these extremes.

7.4 Microbenchmarks

To assess Avenir's scalability, we procedurally generated a collection of microbenchmarks that explore two independent variables, the number of 32-bit input variables I and the number of 32-bit output variables O. For simplicity, the input and output variables sets are distinct.

The abstract pipeline has one table that matches on all of the input variables, and assigns one of the output variables. The target pipeline first matches on all output variables and assigns a metadata value m. This initial staging table is followed by a sequence of O output tables. Table i in this sequence matches solely on m and optionally assigns an output variable.

The results are shown in Figure 13. The x-axis shows the number of bits in the abstract program (i.e., $32(I+O)$) and the y axis shows the time in seconds to translate 100 random abstract edits. The violins show the timing distribution marked with median value. The variation comes from the random generation and from the variation in I and O.

Since networking programs are usually classifier-heavy, we also fixed the number of 32-bit output variables to 8, and varied the size of the classifier. The results are in Figure 14.

Of course, it's difficult to make general claims about the scalability of Avenir's approach, which incorporates numerous heuristics. Nevertheless, it does seem that the complexity increases exponentially with the number of bits, as is expected for a tool that relies on a black-box solver. Target pipelines with different structure than the regular, repeated structure in our microbenchmarks may behave differently.

8 Limitations and Future Work

We discuss two limitations to Avenir's methodology: the cost of formally specificying the abstract and target pipelines, and the run-time overheads of our heuristic search.

The biggest threat to Avenir's use is the requirement that pipelines be formally specified. The work required to develop a formal specification can be significant, and there is no guarantee that a given specification of a pipeline will accurately describe its run-time behavior. Of course, these concerns can be side-stepped if the pipelines are already programmed in P4. But more generally we would need tools for generating specifications and testing conformance. We plan to explore such tools in future work.

Another limitation is our use of heuristic search. The evaluation shows many situations in which Avenir works efficiently, but there are also situations in which it fails to terminate in a reasonable time. For example, to translate from $Pipe_1$ to *OBT* in Figure 2, Avenir maintains a cross product of L2_fwd and L3_fwd, which requires quadratic operations, and causes incremental heursitics to fail. Expanding the effective scope of Avenir's search is future work. We also plan to explore optimal notions of synthesis—e.g., finding the smallest solution.

9 Related Work

Synthesis. Avenir is based on Sketching [39], wherein the programmer is allowed to insert unknown "holes" into a program that are filled using CEGIS [40]. Sketching has been used to build a code generator for packet-processing switch pipelines [16]. NetComplete [10] allows network operators to express their intent by sketching parts of the intended configuration for refactoring or updating purposes. Our novelty is to use sketching to synthesize control plane mappings.

Another use of synthesis is to generate implementations from high-level specifications, e.g., stratified Datalog [11], regular expressions with uninterpreted functions [35], first-order logic constraints [3], and LTL [22].

P4 Verification. There are several recent projects on verifying P4 program properties. Lopes et al. developed an operational semantics for P4 and developed a verification tool based on Datalog which can check program equivalence [24]. P4K presented an operational semantics for P4 using the K framework [19]. p4pktgen uses symbolic execution to generate test cases for P4 programs [26]. ASSERT-P4 translates P4 to a C-like representation, and then symbolically executes the program [15]. Vera [43] uses SymNet [44] as a symbolic execution framework to verify P4 programs. p4v [21] uses symbolic techniques to avoid run-time source traversals.

Network Virtualization. There are many SDN controllers, such as POX [33], NOX [17], and Open Daylight [31]. A few of them specifically target the problem of flow rule composition, including the Frenetic language and controller [14] and Pyretic [25]. Other efforts have focused on network virtualization, i.e., mapping abstract specifications down to target realizations, such as ONIX [20]. FlowVisor [36], CoVisor [18] and the NetKAT compiler [38]. Among this work, Avenir is unique in developing an approach to managing heterogeneous abstract and target pipelines.

10 Conclusion

This paper presented Avenir, a tool that automatically synthesizes control plane operations to ensure uniform behavior across a variety of physical data planes. Avenir uses a counterexample guided inductive synthesis algorithm based on a novel application of sketches to data plane programs. Our evaluation demonstrates that Avenir correctly synthesizes control plane operations with modest overheads.

Acknowledgments

The authors wish to thank Andy Fingerhut for suggesting that we explore the idea of control plane synthesis, and for many helpful discussions as we developed Avenir. Thanks also to Charles Chan and Pier Luigi Ventre from ONF for the discussion on their experience supporting OF-DPA switches in ONOS and Trellis. This work is supported by NSF GRFP grant number DGE-1650441, NSF CCF-1553168, NSF FMiTF-1918396, DARPA HR001120C0107, and gifts from Alibaba, Fujitsu, Infosys, and VMware.

References

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic Foundations for Networks. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA (POPL)*, pages 113–126, January 2014.

[2] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an open, distributed SDN OS. In *ACM Workshop on Hot Topics in Software Defined Networking, Chicago, Illinois (HotSDN)*, pages 1–6, August 2014.

[3] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin T. Vechev. Config2Spec: Mining network specifications from network configurations. In Ranjita Bhagwan and George Porter, editors, *In USENIX Symposium on Networked Systems Design and Implementation, Santa Clara, CA (NSDI)*, pages 969–984, February 2020.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review (CCR)*, 44(3):87–95, July 2014.

[5] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. FBOSS: Building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, Budapest, Hungary (SIGCOMM)*, pages 342–356, August 2018.

[6] P4 Language Consortium. P4Runtime. https://p4.org/p4-runtime/, October 2017. Accessed March, 2021.

[7] P4 Language Consortium. P416 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf, 2018. Accessed March, 2021.

[8] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary (TACAS)*, pages 337–340. Springer, March 2008.

[9] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM (CACM)*, 18(8):453–457, August 1975.

[10] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *USENIX Symposium on Networked Systems Design and Implementation, Renton, WA (NSDI)*, pages 579–594, April 2018.

[11] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Network-Wide Configuration Synthesis. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification, Heidelberg, Germany (CAV)*, volume 10427 of *Lecture Notes in Computer Science*, pages 261–281, 2017.

[12] fabric.p4 source code from ONOS v2.2.2. https://github.com/opennetworkinglab/onos/blob/2.2.2/pipelines/fabric/impl/src/main/resources/fabric.p4, 2020. Accessed March, 2021.

[13] Cormac Flanagan and James B Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK (POPL)*, pages 193–205, 2001.

[14] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *ACM SIGPLAN International Conference on Functional Programming, Tokyo, Japan (ICFP)*, pages 279–291, September 2011.

[15] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering Bugs in P4 Programs with Assertion-based Verification. In *ACM SIGCOMM Symposium on Software Defined Networking Research, Los Angeles, CA (SOSR)*, pages 4:1–4:7, March 2018.

[16] Xiangyu Gao, Taegyun Kim, Michael Dean Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch Code Generation using Program Synthesis. In *ACM Special Interest Group on Data Communication, Virtual Event, USA (SIGCOMM)*, pages 44–61, August 2020.

[17] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review (CCR)*, 38(3):105–110, July 2008.

[18] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *USENIX Symposium on Networked Systems Design and Implementation, Oakland, CA (NSDI)*, pages 87–101, May 2015.

[19] Ali Kheradmand and Grigore Rosu. P4K: A Formal Semantics of P4 and Applications. *Computing Research Repository (CoRR)*, abs/1804.01468, 2018.

[20] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *USENIX Conference on Operating Systems Design and Implementation, Vancouver, BC (OSDI)*, pages 351–364, October 2010.

[21] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4V: Practical Verification for Programmable Data Planes. In *ACM Special Interest Group on Data Communication, Budapest, Hungary (SIGCOMM)*, pages 490–503, August 2018.

[22] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cerný. Event-driven network programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, Santa Barbara, CA (PLDI)*, pages 369–385, 2016.

[23] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review (CCR)*, 38(2):69–74, March 2008.

[24] Nick McKeown, Dan Talayco, George Varghese, Nuno Lopes, Nikolaj Bjørner, and Andrey Rybalchenko. Automatically verifying reachability and well-formedness in P4 Networks. Technical Report MSR-TR-2016-65, September 2016.

[25] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software-Defined Networks. In *USENIX Symposium on Networked Systems Design and Implementation, Lombard, IL (NSDI)*, pages 1–14, April 2013.

[26] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. p4pktgen: Automated Test Case Generation for P4 Programs. In *ACM SIGCOMM Symposium on SDN Research, Los Angeles, CA (SOSR)*, pages 5:1–5:7, March 2018.

[27] Brian O'Connor, Yi Tseng, Maximilian Pudelko, Carmelo Cascone, Abhilash Endurthi, You Wang, Alireza Ghaffarkhah, Devjit Gopalpur, Tom Everman, Tomek Madejski, et al. Using P4 on Fixed-Pipeline and Programmable Stratum Switches. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Los Angeles, CA (ANCS)*, pages 1–2, October 2019.

[28] ONOS : Open Network Operating System. https://github.com/opennetworkinglab/onos/commit/b7b79af9702f03c1286b8f2f9d98e6b87b29c467. Accessed March, 2021.

[29] Open Network Operating System (ONOS) SDN controller for SDN/NFV solutions. https://www.opennetworking.org/onos. Accessed March, 2021.

[30] OpenConfig. https://www.openconfig.net. Accessed March, 2021.

[31] OpenDaylight. https://www.opendaylight.org. Accessed March, 2021.

[32] OpenFlow-Data Plane Abstraction Networking Software. https://www.broadcom.com/products/ethernet-connectivity/software/of-dpa. Accessed March, 2021.

[33] The POX OpenFlow Controller. https://github.com/noxrepo/pox/. Accessed March, 2021.

[34] QMX switches require the unicast flow being installed before multicast flow in TMAC table. https://github.com/opennetworkinglab/onos/commit/45b69ab951915a4211a. Accessed March, 2021.

[35] Shambwaditya Saha, Santhosh Prabhu, and P. Madhusudan. NetGen: synthesizing data-plane configurations for network policies. In Jennifer Rexford and Amin Vahdat, editors, *ACM SIGCOMM Symposium on Software Defined Networking Research, Santa Clara, CA (SOSR)*, pages 17:1–17:6, June 2015.

[36] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the Production Network Be the Testbed? In *USENIX Conference on Operating Systems Design and Implementation, Vancouver, BC (OSDI)*, pages 365–378, October 2010.

[37] SLOCCount. https://dwheeler.com/sloccount/. Accessed March, 2021.

[38] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. A Fast Compiler for NetKAT. In *ACM SIGPLAN International Conference on Functional Programming, Vancouver, BV (2015)*, pages 328–341, August 2015.

[39] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, 2008.

[40] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching Concurrent Data Structures. In *ACM SIGPLAN Notices*, pages 136–148, June 2008.

[41] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial Sketching for Finite Programs. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA (ASPLOS)*, pages 404–415, 2006.

[42] SONiC. https://azure.github.io/SONiC/. Accessed March, 2021.

[43] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 Programs with Vera. In *ACM Special Interest Group on Data Communication, Budapest, Hungary (SIGCOMM)*, pages 518–532, 2018.

[44] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. SymNet: Scalable Symbolic Execution for Modern Networks. In *ACM Special Interest Group on Data Communication, Florianopolis, Brazil (SIGCOMM)*, pages 314–327, August 2016.

[45] Stratum: enabling the era of next generation SDN. https://www.opennetworking.org/stratum/. Accessed March, 2021.

[46] Trellis platform brief. https://www.opennetworking.org/wp-content/uploads/2019/09/TrellisPlatformBrief.pdf. Accessed March, 2021.

[47] Konstantin Weitz, Stefan Heule, Waqar Mohsin, Lorenzo Vicisano, and Amin Vahdat. Leveraging P4 for Fixed-Function Switches. In *P4 Workshop 2019*, 2019.

A Formal Details

The grammar of bit-vector expressions and boolean formulae is described in Figure 15.

$$
\begin{array}{rcll}
e & ::= & & (e \in \mathsf{Expr}) \\
& | & v & \textit{Bit Vector} \\
& | & f & \textit{Field} \\
& | & e - e & \textit{Subtraction} \\
& | & e + e & \textit{Addition} \\
& | & e \,\&\, e & \textit{Bitwise And} \\
\\
b & ::= & & (b \in \mathsf{Bool}) \\
& | & \mathsf{tt} & \textit{Truth} \\
& | & \mathsf{ff} & \textit{Falsehood} \\
& | & e = e & \textit{Equality} \\
& | & e < e & \textit{Inequality} \\
& | & \neg b & \textit{Negation} \\
& | & b \wedge b & \textit{Conjuction} \\
& | & b \vee b & \textit{Disjunction}
\end{array}
$$

Figure 15: Syntax of BitVector expressions and boolean formulae.

The semantics of edits are defined below

$$
\begin{array}{rcl}
\mathsf{A}_x(\rho)\ \tau & \triangleq & \begin{cases} \tau\{x \mapsto \tau(x)@[\rho]\}, & \forall \rho \in \tau(x), \\ & \pi_1(\rho) \neq \pi_1(\rho') \\ \tau, & \textit{otherwise} \end{cases} \\
\mathsf{D}_x(i)\ \tau & \triangleq & \tau\{x \mapsto \tau(x)[0:i]@\tau(x)[i+1:]\} \\
\vec{\delta}\ \tau & \triangleq & \delta_1 \circ \cdots \circ \delta_n\ \ \tau
\end{array}
$$

Instantiations. A table is populated by rows in Row. A single row $\rho = (\vec{m}, \vec{d}, a) \in \mathsf{Row}$ is a tuple comprising a sequence of match values $\vec{m}$, a sequence of action data $\vec{d}$, and an action index i. We write $\pi_1(\rho) = \vec{m}$, $\pi_2(\rho)$ for $\vec{d}$, and $\pi_3(\rho)$ for a.

Note that our instantiations only allow exact matches on data. This does not affect the generality of our formal results, since exact matches can easily encode ternary and lpm matches (with untenable blowup, of course). Using these more compact matches is an optimization we describe in Section 5.

A row is *well-formed* for a table t if $|\vec{m}| = t.keys$, $i < |t.actions|$, and when $t.actions[i] = \lambda\vec{x}.\ c$, then $|\vec{x}| = |\vec{d}|$. In practice, there are additional typing constraints regarding the sizes of the bitvectors, but as we've abstracted bitvectors to naturals in this exposition, we can set that bookkeeping aside. We assume henceforth that all rows are well-formed for the tables into which they are being installed.

An instantiation $\tau \in \mathsf{Inst}$ is a function from table name to sequences of rows that describes the given state of the tables in a pipeline, i.e., given a table t, $\tau(t.name)$ gives us the sequence of rows in the table. We often write $\tau(t)$ as convenient shorthand. Moving forwards we will use τ to describe instantiations for abstract programs and σ to describe instances for physical programs.

An instantiation is well-formed if every row in every table is well-formed

Now that we have pipelines c and instantiations τ, we can define how to combine them via the function $\mathsf{subst}(c, \tau)$, which produces another command with no tables in it. Effectively we replace a table t with a guarded command that checks each row $(\vec{m}, \vec{d}, i)$ in sequence. A single row is translated to the guarded command: $\mathsf{encKeys}(t.keys, \vec{m}) \to t.actions[i](\vec{d})$, where the encKeys function We present this formally in Figure 16. Since instances τ are total functions, $\tau(t)$ will always be defined.

We call the command $c' = \mathsf{subst}(c, \tau)$ an *instantiated pipeline*. Note that all of the table applications have been encoded away and we have a simple loop-free command.

Interpretation. We can interpret instantiated pipelines as functions on packets. A packet comes in and then a (possibly) different packet goes out. Similar to other formalisms of packet processing functions we define packets to be valuations on the headers and metadata [1]: packets are defined as finite maps $\mathsf{Hdr} \cup \mathsf{Meta} \rightharpoonup (\mathsf{BitVec})$. Operations on packets *pkt* are the standard ones: the empty packet is written $\{\}$; to update or set the value of $h \in \mathsf{Hdr}$ to $[n]_s \in \mathsf{BitVec}$ with $h.size = s$, write $pkt\{h \mapsto [n]_s\}$, otherwise the update is undefined; to access the value of x, write $pkt.x$. The set of defined names x in a packet *pkt* is denoted $\mathrm{dom}(pkt)$. A packet is *well-formed* when it can be constructed by a series of defined updates. In what follows, we assume all updates are well-formed and all packets well-defined: specifically, that Pkt the set of well-defined elements of $\mathsf{Hdr} \rightharpoonup (\mathsf{BitVec})$.

The semantics of commands on a packet *pkt* are straightforward. The assignment operation $x := e$ first evaluates e to a value n in the environment defined by the packet *pkt*, and then returns the packet $pkt\{x \mapsto n\}$. The sequence operator $(c_1; c_2)$ is simply interpreted as functional composition: the output of $[\![c_1]\!]\ pkt$ is passed into $[\![c_2]\!]$.

The semantics of guarded commands are broken into two cases. First, if there are no rows in the selection, i.e., if fi, the command is interpreted as the identity function, otherwise, if there is at least one row in the selection (i.e., if $(b \to c)\ \overrightarrow{b \to c}$ fi), then b is evaluated in the packet environment. If it evaluates to tt, then execute c, i.e., $[\![c]\!]\ pkt$, otherwise, if b evaluates to ff, we simply check the remaining rows in the guarded command, i.e $[\![\text{if } \overrightarrow{b \to c} \text{ fi}]\!]\ pkt$.

Finally, the denotation of a table is simply its default action.

We leave the semantics of our expressions ($\mathcal{E}[\![-]\!]$) and booleans ($\mathcal{B}[\![-]\!]$) undefined, as their definitions are standard.

The logical encoding uses the predicate transformer semantics of GCL programs [9]. Our semantics are identical to Dijkstra's, with the exception of our guarded commands. Ours are definitionally mutually exclusive, his are nondeterministic. To remedy this we simply negate the preceding guards, and then apply the weakest precondition function, *wp*.

$$
\begin{array}{rcl}
\Phi & \triangleq & \bigwedge\{h.name = x' \mid h \in\ x' \text{ fresh}\} \\
c_1 \equiv c_2 & \triangleq & wp(c_1,\Phi) \Leftrightarrow wp(c_2,\Phi) \\
\mathsf{ISVALID}(\varphi) & \triangleq & \mathsf{SAT}(\neg\varphi)
\end{array}
$$

Figure 18: The verification condition for determining when two programs implement the same function

$$
\begin{array}{l}
\mathsf{encKeys} :: \mathsf{List}[\mathsf{Hdr}] \times \mathsf{List}[\mathsf{BitVec}] \to \mathsf{Bool} \\
\mathsf{encKeys}(\vec{k},\vec{m}) \quad \triangleq \quad \bigwedge_{0\le i<|t.keys|} k_i = m_i \\
\\
\mathsf{subst}(c,\tau) :: \mathsf{Cmd} \\
\mathsf{subst}(x := e,\tau) \quad \triangleq \quad x := e \\
\mathsf{subst}(c;c,\tau) \quad \triangleq \quad \mathsf{subst}(c,\tau);\mathsf{subst}(c,\tau) \\
\mathsf{subst}(\mathsf{if}\ \overrightarrow{b \to c}\ \mathsf{fi},\tau) \quad \triangleq \quad \mathsf{if}\ \overrightarrow{b \to \mathsf{subst}(c,\tau)}\ \mathsf{fi} \\
\mathsf{subst}(\mathsf{apply}\ t,\tau) \quad \triangleq \\
\quad \mathsf{if} \\
\quad\ \mathsf{encKeys}(t.keys,\vec{m}_1) \to t.actions[a_1](\vec{d}_1) \\
\qquad\qquad\qquad \vdots \\
\quad\ \mathsf{encKeys}(t.keys,\vec{m}_n) \to t.actions[a_n](\vec{d}_n) \\
\quad \mathsf{fi} \\
\\
\quad \text{where } (\vec{m}_1,\vec{d}_1,a_1),\ldots,(\vec{m}_n,\vec{d}_n,a_n) = \tau(t)
\end{array}
$$

Figure 16: Semantics of Table Instances

$$
\begin{array}{l}
[\![c]\!] :: \mathsf{Pkt} \to \mathsf{Pkt} \\
[\![x := e]\!]\ pkt \quad \triangleq \quad pkt\{x \mapsto \mathcal{E}[\![e]\!]\ pkt\} \\
[\![c_1;c_2]\!]\ pkt \quad \triangleq \quad [\![c_2]\!] \circ [\![c_1]\!]\ pkt \\
[\![\mathsf{if}\ \ \mathsf{fi}]\!]\ pkt \quad \triangleq \quad pkt \\
[\![\mathsf{if}\ (b \to c)(\overrightarrow{b \to c})\mathsf{fi}]\!]\ pkt \\
\qquad\qquad \triangleq \begin{cases} [\![c]\!]\ pkt & \mathcal{B}[\![b]\!]\ pkt = \mathsf{tt} \\ [\![\mathsf{if}\ \overrightarrow{b \to c}\ \mathsf{fi}]\!]\ pkt & \text{otherwise} \end{cases} \\
[\![\mathsf{apply}(t)]\!]\ pkt \quad \triangleq \quad [\![t.default]\!]\ pkt
\end{array}
$$

Figure 17: Semantics of programs

The verification condition is defined in Figure 18. Leveraging the connection between weakest preconditions and denotational semantics [9], we can define equivalence pipelines c_1 and c_2 by taking the weakest precondition with respect to a particular formula Φ, which equates every header variable to a fresh symbolic variable. The free variable will then capture the input value of a program, and the symbolic variable will capture the output value of the program.

As an example, consider the following command c:

$$
\begin{array}{lcll}
\mathsf{if} & x = 1 & \to & x := 5 \\
 & \mathsf{tt} & \to & x := 9 \\
\mathsf{fi} & & &
\end{array}
$$

Then the Φ corresponding to this command is $x = x'$. Then $wp(c,\Phi)$ is equivalent to

$$
\begin{array}{r}
(x = 1 \Rightarrow 5 = x') \\
\wedge (x \neq 1 \Rightarrow 9 = x')
\end{array}
$$

The variable x captures the input conditions (either $x = 1$ or not), and the x's capture the output varibles: when the input x value is one, the output will be 5; otherwise, it will be 9.

Instrumentation The instr function formalizes how to add holes to a table. We write else if to help delineate elements in the selection list.

$$
\begin{array}{l}
\mathsf{instr}(t,\tau,\vec{\delta},n) \quad \triangleq \\
\quad \mathsf{if}\ t.keys = \vec{m}_i \wedge ?\mathtt{Del}_{t.name,i} = 0 \to t.actions[a_i](\vec{d}_i) \\
\qquad\qquad\qquad \text{for } (\vec{m}_i,\vec{d}_i,a_i) \in \tau(t) \text{ if } \mathsf{D}(t.name,i) \notin \vec{\delta} \\
\quad \mathsf{else\ if}\ t.keys = \vec{m} \to t.actions[a_i](\vec{d}) \\
\qquad\qquad\qquad \text{for } \mathsf{A}(t.name,(\vec{m}_i,\vec{d}_i,a_i)) \in \vec{\delta} \\
\quad \mathsf{else\ if} \\
\quad \left(\begin{array}{l} \bigwedge_{k \in t.keys} k = ?\mathtt{k}_{t.name,j} \\ \wedge ?\mathtt{Add}_{t.name,j} = 1 \\ \wedge \bigwedge_{k \le j} ?\mathtt{Add}_{t.name,k} = 1 \\ \wedge ?\mathtt{ActId}_{t.name,j} = i \end{array} \right) \to t.action[i](\overrightarrow{?d_{t.name,j,i}}) \\
\qquad\qquad\qquad \text{for } 0 \le i < |t.actions| \text{ and } 0 \le j < n \\
\quad \mathsf{fi}
\end{array}
$$

Figure 19: The instr function formalizes how to add holes to a tables

B Proof of Completeness

Lemma 1 (Hits Restrict)**.** *For every program $p \in \mathsf{Cmd}$, instance $\sigma \in \mathsf{Inst}$, and set of counter examples X such that for every $(\chi_0,\chi_1) \in X$, $[\![\mathsf{subst}(p,\sigma)]\!]\ \chi_0 = \chi_1$, then there exists σ' such that $X \subseteq [\![\mathsf{subst}(p,\sigma')]\!]$ and $|\sigma'(t)| \leq |X|$ for every $t \in \mathsf{Tables}(p)$.*

Proof. Proceed by induction on the structure of p:

[$(p = x := e)$] Trivial because $\mathsf{Tables}(x := e) = \emptyset$ and for every σ, $\mathsf{subst}(x := e, \sigma) = x := e$.

[$p = (p_1; p_2)$] Assume $p = p_1; p_2$. Introduce X and σ as above.

Decompose X across p_1 and p_2 such that

$$\begin{array}{rcl} X_1 & \triangleq & \{(\chi_0,\chi_1) \mid [\![\mathsf{subst}(p_1,\sigma)]\!]\ \chi_0 = \chi_1, \chi_0 \in \pi_1(X)\} \\ X_2 & \triangleq & \{(\chi_1,\chi_2) \mid [\![\mathsf{subst}(p_2,\sigma)]\!]\ \chi_1 = \chi_2, \chi_1 \in \pi_2(X_1)\} \end{array}$$

Notice that $X_2 \circ X_1 = X$.

Now by the IH on p_1, σ and X_1 we get σ_1 such that every rule is hit by a counterexample in X_1,

$$\forall (\chi_0,\chi_1) \in X_1, [\![\mathsf{subst}(p_1,\sigma_1)]\!]\ \chi_0 = \chi_1$$

and $|\sigma_1(t)| \leq |X_1|$ for all $t \in \mathsf{Tables}(p_1)$. Similarly, by the IH on p_2, σ and X_2, we get σ_2 such that every rule is hit by a counterexample in X_2.

$$\forall (\chi_1,\chi_2) \in X_2, [\![\mathsf{subst}(p_2,\sigma_2)]\!]\ \chi_1 = \chi_2$$

and $|\sigma_2(t)| \leq |X_1|$ for all $t \in \mathsf{Tables}(p_2)$.

Now we can construct σ' such that

$$\sigma'(t) = \begin{cases} \sigma_1(t), & \text{if } t \in \mathsf{Tables}(p_1) \\ \sigma_2(t), & \text{if } t \in \mathsf{Tables}(p_2) \end{cases}$$

Observe that $\sigma|_{\mathsf{Tables}(p_i)} = \sigma_i$ for $i = 1,2$. Now we show the two remaining properties.

- Since $X = X_2 \circ X_1$ is a function, we know $\forall t \in \mathsf{Tables}(p)$, $|\sigma_1(t)| \leq |X_1| \leq |X|$ and $|\sigma_2(t)| \leq |X_2| \leq |X|$, and the table names in p_1 and p_2 are disjoint. Conclude that $\forall t \in \mathsf{Tables}(p), |\sigma'(t)| \leq |X|$.
- Show $X \subseteq [\![\mathsf{subst}(p_1; p_2, \sigma')]\!]$. Our IHs give

$$\begin{array}{l} X_1 \subseteq [\![\mathsf{subst}(p_1,\sigma_1)]\!] \\ X_2 \subseteq [\![\mathsf{subst}(p_2,\sigma_2)]\!] \end{array}$$

By definition,

$$[\![\mathsf{subst}(p_1; p_2, \sigma')]\!] = [\![\mathsf{subst}(p_2,\sigma')]\!] \circ [\![\mathsf{subst}(p_1,\sigma')]\!]$$

Then,

$$[\![\mathsf{subst}(p_2,\sigma')]\!] \circ [\![\mathsf{subst}(p_1,\sigma')]\!] = [\![\mathsf{subst}(p_2,\sigma_2)]\!] \circ [\![\mathsf{subst}(p_1,\sigma_1)]\!]$$

by the disjointness of table names. The result follows.

[$p = \mathsf{if}\ \overrightarrow{b \to p}\ \mathsf{fi}$] Similar, but n-ary.

[$p = \mathsf{apply}(t)$] Assume that $p = \mathsf{apply}(t)$ for some table t. The corresponding rows for the table are $\sigma(t) = \vec{\rho}$. Compute a subsequence $\vec{\rho}'$ of $\vec{\rho}$ such that $\vec{\rho}'$ contains ρ_i iff there is some input packet in X that hits ρ_i. Create an instantiation $\sigma' = \sigma\{t \mapsto \vec{\rho}'\}$. Since each the rules are equality matches (and hence disjoint), $[\![\mathsf{subst}(\mathsf{apply}(t),\sigma')]\!]\ \chi_0 = \chi_1$ for every $(\chi_0,\chi_1) \in X$. Since some $\chi_0 \in \pi_1(X)$ may miss, conclude that $|\sigma'(t)| \leq |X|$. □

Lemma 2 (Model Solution)**.** *For every $p \in \mathsf{Cmd}$, every $\sigma \in \mathsf{Inst}$, and every $X \in \mathsf{Pkt}^2$, if there exists $\vec{\delta} \in \mathsf{List}[\mathsf{Edit}]$ such that $X \subseteq [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$, then*

$$\mathsf{SAT}\ \vec{\delta}' = \mathsf{model}(p,\sigma,X)$$

and

$$X \subseteq [\![\mathsf{subst}(p,\vec{\delta}'(\sigma))]\!]$$

Proof. Let $\vec{\delta}$ be such that $X \subseteq [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$.

Lemma 1 gives us σ' from $\vec{\delta}(\sigma)$ such that for every table $t \in \mathrm{dom}(\sigma')$, $|\sigma'(t)| \leq |X|$, and $X \subseteq [\![\mathsf{subst}(p,\sigma')]\!]$.

Construct the following witness to the query in Figure 5

$$\left\{ \begin{array}{l|l} ?\mathtt{Del}_{x,i} \mapsto 1, & t \in \mathsf{Tables}(p), \\ ?\mathtt{Add}_{x,j} \mapsto 1, & x = t.name \\ ?\mathtt{ActId}_{x,j} \mapsto a, & 0 \leq i < |\sigma(t)|, \\ \overrightarrow{?k_{x,j} \mapsto m}, & 0 \leq j < |\sigma'(t)| \\ \overrightarrow{?d_{x,j,i} \mapsto d} & (\vec{m},\vec{d},a) = \sigma'(t)[j] \end{array} \right\}$$

which corresponds to the rules

$$\begin{array}{rcll} d_{t,i} & \triangleq & \mathsf{D}(t,i), & \forall t \in \mathsf{Tables}(p), |\sigma(t)| > i \geq 0 \\ a_{t,j} & \triangleq & \mathsf{A}(t,\sigma'(t)[j]), & \forall t \in \mathsf{Tables}(p), 0 \leq j < |\sigma'(t)| \\ \vec{\delta}' & \triangleq & \vec{d} \cdot \vec{a} & \end{array}$$

Note that $\vec{d}_t$ is sorted from highest index to lowest index. Observe that $\vec{\delta}'(\sigma) = \vec{a}(\vec{d}(\sigma))\sigma = \vec{a}(\cdot) = \sigma'$, so we can conclude that $\forall (\chi_0,\chi_1) \in X, [\![\mathsf{subst}(p,\vec{\delta}'(\sigma))]\!]\ \chi_0 = \chi_1$, and we're done. □

Theorem 6 (Completeness)**.** *For every logical program* $l \in \mathsf{Cmd}$*, physical program* $p \in \mathsf{Cmd}$*, logical instantiation* $\tau \in \mathsf{Inst}$*, physical instantiation* $\sigma \in \mathsf{Inst}$*, sequence of physical edits* $\vec{\delta} \in \mathsf{List}[\mathsf{Edit}]$*, and every set* X *s.t.*

$$X \subseteq [\![\mathsf{subst}(l,\tau)]\!] \cap [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!],$$

if

$$\exists \vec{\varepsilon} \in \mathsf{List}[\mathsf{Edit}].\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\sigma))$$

then

$$\mathsf{Ok}\ \vec{\delta}' = \mathsf{cegis}(l,p,\tau,\sigma,\vec{\delta},X)$$

and

$$\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}'(\sigma))$$

Proof. Proceed by induction on $|\mathsf{Pkt} \setminus \pi_1(X)|$.

Base Case. $(\pi_1(X) = \mathsf{Pkt})$. The definition of functional subset gives us

$$\forall(\chi_1,\chi_2) \in X.[\![\mathsf{subst}(l,\tau)]\!]\ \chi_1 = \chi_2 = [\![\mathsf{subst}(p,\vec{\delta(\sigma)})]\!]\ \chi_1$$

which reduces to

$$\forall \chi_1 \in \pi_1(X).[\![\mathsf{subst}(l,\tau)]\!]\ \chi_1 = [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]\ \chi_1.$$

Then by the assumption that $\pi_1(X) = \mathsf{Pkt}$, this is just

$$\forall \chi_1 \in \mathsf{Pkt}.[\![\mathsf{subst}(l,\tau)]\!]\ \chi_1 = [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]\ \chi_1$$

Conclude, by definition, that

$$\mathsf{subst}(l,\tau) = \mathsf{subst}(l,\vec{\delta})$$

Inductive Step. $(\pi_1(X) \subseteq \mathsf{Pkt})$. If $\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}(\sigma))$, then we're done, so assume instead that we have a counterexample $\chi = (\chi_0,\chi_1)$ such that

$$[\![\mathsf{subst}(l,\tau)]\!]\ \chi_0 = \chi_1 \neq [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$$

which tells us that that $\chi \notin X$.

Our main assumption tells us that there is some $\vec{\varepsilon}$ such that

$$\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\sigma))$$

By definition we have

$$\forall pkt \in \mathsf{Pkt}.[\![\mathsf{subst}(l,\tau)]\!]\ pkt = [\![\mathsf{subst}(p,\vec{\varepsilon}(\sigma))]\!]\ pkt.$$

which, since $\pi_1(\{\chi\} \cup X) \subseteq \mathsf{Pkt}$, means that

$$\forall pkt \in \{\chi\} \cup X.[\![\mathsf{subst}(l,\tau)]\!]\ pkt = [\![\mathsf{subst}(p,\vec{\varepsilon}(\sigma))]\!]\ pkt,$$

Then Lemma 2 gives us a model $\vec{\delta}'$ such that

$$\forall pkt \in \{\chi\} \cup X.[\![\mathsf{subst}(l,\tau)]\!]\ pkt = [\![\mathsf{subst}(p,\vec{\delta}'(\sigma))]\!]\ pkt$$

The result follows by IH on $\{\chi\} \cup X$ as $|\{\chi\} \cup X| > |X|$. □

C Proof of Completeness for Incremental Synthesis

Definition 1 (Minimal Sequence)**.** *Given an instantiation* τ*, a sequence of edits* $\vec{\delta}$ *is minimal iff for every table* $t \in \mathrm{dom}(\tau)$*, for every* $\vec{\delta}'$ *s.t.* $\vec{\delta}(\sigma) = \vec{\delta}'(\sigma)$*,* $|\vec{\delta}| \leq |\vec{\delta}'|$.

Definition 2 (Minimal Extension)**.** *Given a command* c*, an instantiation* τ*, and a sequence of edits* $\vec{\delta}$*, another sequence* $\vec{\delta}'$ *is called a* minimal extension *of* $\vec{\delta}$ *when* $\vec{\delta} \circ \vec{\delta}'$ *is minimal.*

Lemma 3 (Finite Minimal Sequences)**.** *Given a command* c*, and an instantiation* τ*, the set of minimal sequences is finite.*

Proof. Proceed by induction on the structure of c.

($h := e$) There there is one minimal instatiation: $[]$.

($c_1; c_2$) By IHs, the set of minimal sequences corresponding to c_1 and τ is a finite D_1, and the set of minimal sequences corresponding to c_1 and τ is a finite D_2. The set of minimal sequences corresponding to $c_1; c_2$ is the set of all interleaving of all sequences in D_1 and sequences of D_2, which is finite.

(if $\overrightarrow{b \to c}$ fi) This is similar to the previous case except $n$ary.

($\mathsf{apply}(t)$) First we show that the space of functions that t can represent is finite. Since $t.keys$ is finite, and each header in $t.keys$ has a finite domain of values, so the domain of matches is finite. A similar argument shows that the domain of actions is finite, and so the domain of functions is finite. Further for each of these functions, there are finitely many ways of representing them (all of the permutations of the rules).

This means that there are finitely many minimal sequences when $\tau(t) = []$, simply install each of these, with no duplicates, since removing the duplicates would result in a smaller sequence of edits.

When $\tau(t) \neq []$ there are also finitely many minimal sequences. They are those that delete as few rules in $\tau(t)$ as necessary and then install the missing rules (if any are needed): they never delete a rule and then reinstall the same rule, and then never install a rule only to delete it later. □

Lemma 4 (Finite Minimal Extensions)**.** *For a command* c *and an instantiation* τ*, and a sequence of edits* $\vec{\delta}$*, there are finitely many minimal extensions of* $\vec{\delta}$*.*

Proof. There are finitely many minimal extensions of c and τ. Some of those have $\vec{\delta}$ as a prefix; there are finitely many of them. □

Definition 3 (φ-Preclusion)**.** *For a program c, instantiation* τ*, and a packet pair* $\chi \notin [\![\mathsf{subst}(c,\tau)]\!]$*, we say that* φ precludes $\vec{\delta}$ *when for every* $\vec{\delta}' \models \varphi$ *that is also a prefix of* $\vec{\delta}$*,* $\chi \notin [\![\mathsf{subst}(p,\vec{\delta}(\tau))]\!]$*. We say that* φ precludes a solution*, when it precludes every* $\vec{\delta}$*.*

Definition 4 (Zip Function)**.** *Let* $\mathsf{zip}(\vec{x},\vec{y})$ *be the function that simultaneously iterates through* $\vec{x}$ *and* $\vec{y}$ *and returns a sequence of the element-wise pairs, i.e.,* $\overrightarrow{(x,y)}$*. The function is undefined if* $|\vec{x}| \neq |\vec{y}|$*.*

Lemma 5 (Model Finding)**.** *For every logical program l, physical program p, logical instantiation* τ*, phsyical instantiation* σ*, sequence of physical edits* $\vec{\delta}$*, counterexample* $\chi \in [\![\mathsf{subst}(l,\tau)]\!] \setminus [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$*, and formula* $\varphi \in \mathsf{Bool}$ *such that for every* $\vec{\delta}' \models \neg\varphi$ *st* $\chi \in [\![\mathsf{subst}(p,\vec{\delta}'(\vec{\delta}(\sigma)))]\!]$*, there is no extension* $\vec{\delta}''$ *such that* $\mathsf{subst}(l,\tau) = \mathsf{subst}(l,\vec{\delta}''(\vec{\delta}'(\vec{\delta}(\sigma))))$*, then if there exists* $\vec{\varepsilon}$ *not precluded by* φ *such that*

$$\mathsf{subst}(p,\tau) = \mathsf{subst}(l,\vec{\varepsilon}(\vec{\delta}(\sigma)))$$

, and $\vec{\delta}\circ\vec{\varepsilon}$ *is minimal, then*

$$\mathsf{Sat}\ \vec{\delta}'' = \mathsf{model}'(p,\sigma,\delta,\chi,\varphi)$$

Proof. Let l, p, τ, σ, $\vec{\delta}$, χ, and φ be given. Construct the following model for the query in $\mathsf{model}'(p,\sigma,\delta,\chi,\varphi)$:

We accumulate a model for every table $t \in \mathsf{Tables}(p)$. If $\pi_1(\chi)$ in $\mathsf{subst}(p,\vec{\varepsilon}(\vec{\delta}(\sigma)))$ hits the ith row ρ_i, then there are several cases

MISS If $\pi_1(\chi)$ misses in $\mathsf{subst}(p,\vec{\delta}(\sigma))$, then add

$$\left\{\begin{array}{l} ?\mathsf{AddRowTo}_{t,1} \mapsto 1 \\ ?\mathsf{Act}_{t,1} \mapsto \rho_i.action \\ \overrightarrow{?\mathsf{k}_{t,1}} \mapsto \rho_i.keys \\ \overrightarrow{?\mathsf{d}_{t,\rho.action,1}} \mapsto \rho_i.data \end{array}\right\}$$

HITCORRECT If $\pi_1(\chi)$ hits the jth row ρ_j of $(\vec{\delta}(\sigma))(t)$ in $\mathsf{subst}(p,\vec{\delta}(\sigma))$, and $\rho_i = \rho_j$ then do nothing.

HITWRONG If $\pi_1(\chi)$ hits the jth row of $(\vec{\delta}(\sigma))(t)$ in $\mathsf{subst}(p,\vec{\delta}(\sigma))$ and $\rho_j \neq \rho_i$,

$$\left\{\begin{array}{l} ?\mathsf{AddRowTo}_{t,1} \mapsto 1 \\ ?\mathsf{Act}_{t,1} \mapsto \rho_i.action \\ \overrightarrow{?\mathsf{k}_{t,1}} \mapsto \rho_i.keys \\ \overrightarrow{?\mathsf{d}_{t,\rho.action,1}} \mapsto \rho_i.data \end{array}\right\}$$

From here we extract queries from the model as before, sorting the deletions by decreasing index to get $\vec{\delta}''$. Note that χ hits a syntactically equivalent rule in $\mathsf{subst}(p,\vec{\delta}''(\vec{\delta})(\sigma))$ as in $\mathsf{subst}(l,\vec{\varepsilon}(\vec{\delta}(\sigma)))$. We conclude $\chi \in [\![\mathsf{subst}(p,\vec{\delta}''(\vec{\delta}(\sigma)))]\!]$ □

Lemma 6 (Nontrivial Models)**.** *For every physical program p, physical instance* σ*, sequence of edits* $\vec{\delta}$*, formula* φ*, and counterexample* $\chi \in \mathsf{Pkt}^2$ *such that* $\chi \notin [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$*, if* $\mathsf{Sat}\ \vec{\delta}'' = \mathsf{model}'(p,\sigma,\vec{\delta},\chi,\varphi)$*, then* $\vec{\delta}'' \neq []$*.*

Proof. Let $p \in \mathsf{Cmd}$, $\sigma \in \mathsf{Inst}$, $\vec{\delta} \in \mathsf{List}[\mathsf{Edit}]$ and φ and $\chi \in \mathsf{Pkt}^2$ be given. Assume $\mathsf{Sat}\ \vec{\delta}'' = \mathsf{model}'(p,\sigma,\vec{\delta},\varphi)$.

Prove the contrapositive, that if $\vec{\delta}'' = []$, then $\chi \in [\![\mathsf{subst}(p,\sigma)]\!]$. Assume $\vec{\delta}'' = []$. Then the query in $\mathsf{model}(p,\sigma,\vec{\delta},\chi,\varphi)$ is satisfiable with all deletion and insertion holes set to zero. This means that the following query is also satisfiable (where $\chi = (pkt,pkt')$):

$$\mathsf{SAT}\left(\forall\vec{x}.\varphi \wedge \left(\overrightarrow{\chi_1.x = x}\right) \Rightarrow wp\left(\mathsf{subst}(p,\vec{\delta}(\sigma)),\left(\overrightarrow{pkt'.x = x}\right)\right)\right)$$

By the correspondence between *wp* and the denotational semantics, conclude that $\chi \in [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$. □

Proposition 1 (Oracle Constraints)**.** *For a given physical program p, edit sequence* δ *and instance* σ*,*

$$\mathsf{HEURISTIC}() \Rightarrow \bigwedge_{t\in\mathsf{Tables}(p)} ?\mathsf{Add}_{t,1} = 1 \Rightarrow \bigwedge_{(\vec{m},\vec{d},\vec{a})\in\vec{\delta}(\sigma)(t)} \neg\left(\overrightarrow{?\mathsf{k}_{t,1} = m}\right)$$

and

$$\mathsf{HEURISTIC}() \Rightarrow \bigwedge_{\substack{t\in\mathsf{Tables}(p)\\ 0\le i<|\sigma(t)|\\ (\vec{m},\vec{d},a)=\sigma(t)[i]}} ?\mathsf{Del}_{t,i} = 1 \Rightarrow \neg\left(\begin{array}{l} \overrightarrow{?\mathsf{k}_{t,1} = m} \\ \wedge ?\mathsf{Act}_{t,l} = a \\ \wedge \overrightarrow{?\mathsf{d}_{t,a,1} = d} \end{array}\right)$$

Lemma 7 (Reachable Edits)**.** *For every program p, instantiation* σ*, edit sequence* $\vec{\delta}$*, counterexample* χ *and* $\varphi \in \mathsf{Bool}$*, if*

$$\mathsf{Sat}\ \vec{\delta}' = \mathsf{model}'(p,\sigma,\vec{\delta},\chi,\varphi)$$

then the insertions in $\vec{\delta}$ *are reachable.*

Proof. By Proposition 1. □

Lemma 8 (Deletes Not Resurrected)**.** *For every program p, instantiation* σ*, edit sequence* $\vec{\delta}$*, counterexample* χ *and* $\varphi \in \mathsf{Bool}$*, if*

$$\mathsf{Sat}\ \vec{\delta}' = \mathsf{model}'(p,\sigma,\vec{\delta},\chi,\varphi)$$

then the insertions in $\vec{\delta}$ *are reachable.*

Proof. By Proposition 1. □

Lemma 9 (Minimal Models)**.** *For every physical program p, physical instance* σ*, sequence of edits* $\vec{\delta}$*, formula* φ*, and counterexample* $\chi \in \mathsf{Pkt}^2$ *such that* $\chi \notin [\![\mathsf{subst}(p,\sigma)]\!]$*, if* $\mathsf{Sat}\ \vec{\delta}'' = \mathsf{model}'(p,\sigma,\vec{\delta},\varphi)$*, then* $\vec{\delta}\circ\vec{\delta}''$ *is minimal.*

Proof. Let p, σ, $\vec{\delta}$, φ, χ be given. Assume Sat $\vec{\delta}'' = \mathsf{model}'(p,\sigma,\vec{\delta},\varphi)$.

Consider another sequence of edits $\vec{\delta}'$ such that there is some σ' such that $\vec{\delta}''(\vec{\delta}(\sigma)) = \sigma' = \vec{\delta}'(\sigma)$. Show that $|\vec{\delta}\circ\vec{\delta}''| \leq |\vec{\delta}'|$.

We prove two propositions.

(ADD) Assume that there were some insertion $\delta_i = \mathsf{A}(t,\rho) \in \vec{\delta}\circ\vec{\delta}''$ that doesn't occur in $\vec{\delta}'$. If $\delta_i \in \vec{\delta}$, then $\rho \in \sigma'(t)$. By minimality of $\vec{\delta}$, and $\rho \notin \sigma'(t)$. So $\sigma' \neq \vec{\delta}(\sigma)$, which is a contradiction. If $\delta_i \in \vec{\delta}''$, then $\rho \in \sigma'(t)$, and $\rho \notin \sigma(t)$, because model$'$ always produces reachable edits (Lemma 7). Consequently $\vec{\delta}'(\sigma) \neq \sigma'$ which is a contradiction.

So we know that δ_i has a corresponding edit $\delta'_j \in \vec{\delta}'$. Assume that there is another edit $\delta_k \in \vec{\delta}\circ\vec{\delta}$ that corresponds to δ'_j. This is impossible by Lemma 7 and by the minimality of $\vec{\delta}$.

(DEL) Assume that there were some edit $\delta_j = \mathsf{D}(t,i) \in \vec{\delta}\circ\vec{\delta}''$ deletes some row ρ that occurs in $\vec{\delta}'(\sigma)$: ie. $((\delta_{i-1}\circ\cdots\circ\delta_1)(\sigma))(t)[i] = \rho \notin \sigma'(t)$ and $\rho \in \vec{\delta}'(\sigma)$.

If $\delta_i \in \vec{\delta}''$, we know, by construction, that δ_i deletes a row in $\sigma(t)$. Further, Lemma 8 says $\vec{\delta}'(\sigma) \neq \sigma'$, which is a contradiction.

So we know that δ_i has a corresponding edit $\delta'_j \in \vec{\delta}'$. Assume that there is another edit $\delta_k \in \vec{\delta}\circ\vec{\delta}$ that corresponds to δ'_j. This is impossible because rows can only be deleted once.

Since every edit in $\vec{\delta}\circ\vec{\delta}''$ has a corresponding unique edit in δ'. Conclude that $\vec{\delta}\circ\vec{\delta}'' \subseteq \vec{\delta}'$. The result follows. □

Lemma 10 (Completeness)**.** *For every logical program l, every physical program p, every logical instantiation τ, every physical instantiation σ and every sequence of physical edits $\vec{\delta}$, then, the following properties hold:*

1. *if there exists a sequence of physical edits $\vec{\delta}'$ such that*

$$\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}'(\vec{\delta}(\sigma)))$$

then

$$\mathsf{Ok}\ \vec{\delta}'' = \mathsf{verify}(l,p,\tau,\sigma,\vec{\delta})$$

and

$$\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}''(\sigma))$$

2. *For every $\chi \in [\![\mathsf{subst}(l,\tau)]\!] \setminus [\![\mathsf{subst}(p,\vec{\delta}(\sigma))]\!]$, and every $\varphi \in \mathsf{Bool}$ such that if $\vec{\delta}' \models \neg\varphi$ and $[\![\mathsf{subst}(l,\tau)]\!]\ \pi_1(\chi) = \pi_2(\chi) = [\![\mathsf{subst}(p,\vec{\delta}'(\vec{\delta}(\sigma)))]\!]\ \pi_1(\chi)$, there is no extension $\vec{\delta}''$ such that $\mathsf{subst}(l,\tau) = \mathsf{subst}(l,\vec{\delta}''(\vec{\delta}'(\vec{\delta}(\sigma))))$ then if there exists a non-empty, minimal sequence of physical edits $\vec{\varepsilon}$ not precluded by φ such that*

$$\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\vec{\delta}(\sigma)))$$

then

$$\mathsf{Ok}\ \vec{\delta}'' = \mathsf{solve}(l,p,\tau,\sigma,\vec{\delta},\chi,X,\varphi)$$

and

$$\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\delta}''(\sigma))$$

Proof. First we justify the finiteness of our inner inductive measure. There is a finite number of models to every model$'$ query, simply because there are finitely many holes, each of which has finite domain. Since φ is composed of the same set of variables, it is also finite.

Let l, p, τ and σ be given. Proceed by induction on the number of nonempty minimal extensions of $\vec{\delta}$. Lemma 4 shows this measure is well-formed.

BASE CASE There are no nonempty minimal extensions of $\vec{\delta}$. Consider each proposition separately

1. Let $\vec{\varepsilon}$ be a nonempty sequence of edits such that $\mathsf{subst}(l,\tau) = \mathsf{subst}(l,\vec{\varepsilon}(\sigma))$. However, $\vec{\delta}$ has no nonempty minimal extensions, so $\mathsf{subst}(l,\tau) = \mathsf{subst}(l,\vec{\delta}(\sigma)))$. Entering the verify function, observe that by Theorem 1 CheckSat($\mathsf{subst}(l,\tau) \not\equiv \mathsf{subst}(l,\vec{\delta}(\sigma))$) will be UNSAT, and we're done by Theorem 4.
2. Vacuous, there is no such $\vec{\varepsilon}$.

INDUCTIVE STEP There are nonempty minimal extensions of $\vec{\delta}$.

First we prove proposition 2 by strong induction on the number of models for φ.

BASE CASE There are no models for φ, i.e., φ is unsatisfiable. Consequently, the call to model$'$ fails to produce a model. This is a contradiction by Lemma 5.

INDUCTIVE STEP There are n models for φ Let X and χ be given. Assume $\vec{\varepsilon}$ exists such that $\mathsf{subst}(l,\tau) = \mathsf{subst}(p,\vec{\varepsilon}(\vec{\delta}(\sigma)))$.

Now, we get a model by calling model$'(p,\sigma,\vec{\delta},\chi,\varphi)$, and there are two cases.

i. Assume the result is UNSAT. This is a contradiction by Lemma 5.
ii. Assume the result is Sat $\vec{\delta}''$. By Lemma 6, $|\vec{\delta}''|$ is nonempty. Consider two cases:
Case 1. Assume there exists some extension of $\vec{\delta}\circ\vec{\delta}''$ that is a solution. Then the outer IH proves that the verify call produces a solution.
Case 2. Assume there is no extension of $\vec{\delta}\circ\vec{\delta}''$ that is a solution. Then verify$(l,p,\tau,\sigma,\vec{\delta}\circ\vec{\delta}'')$

returns Fail. Then, since $\vec{\delta}'' \models \varphi$, and $\vec{\delta} \not\models \neg\vec{\delta}$, the number of models for $\varphi \wedge \neg\vec{\delta}''$ is strictly less than n. Further, we know that there is an way to extend $\vec{\delta}$ that is a solution, namely $\vec{\varepsilon}$. We also know that $\vec{\varepsilon}$ isn't precluded by φ, by assumption, finally we also know that $\vec{\varepsilon}$ isn't precluded by $\neg\vec{\delta}''$, because of our assumption that $\vec{\delta} \circ \vec{\delta}''$ cannot be extended to a solution. These final conditions witness the preconditions of the inner IH, which proves the result.

✓

Now prove proposition 1. Let $\vec{\varepsilon}$ be such that

$$\mathsf{subst}(l,\tau) = \mathsf{subst}(l,\vec{\varepsilon}(\vec{\delta}(\sigma)))$$

There are two cases, either $\mathsf{subst}(l,\tau) = \mathsf{subst}(l,\vec{\delta}\sigma)$ or not. In the former case, we are done by Theorem 1. In the latter, we will get a counterexample χ (by Theorem 1), such that

$$\chi \notin \mathsf{subst}(l,\tau) \cap \mathsf{subst}(l,\vec{\delta}(\sigma))$$

Then the result follows as a special case of the preceeding proof of proposition 2. □