How to Avoid Making a Billion-Dollar Mistake: Type-Safe Data Plane Programming with SafeP4

Anonymous Authors

Abstract

The P4 programming language offers high-level, declarative abstractions that bring the flexibility of software to the domain of networking. Unfortunately, the main abstraction used to represent packet data in P4—header types—lacks basic safety guarantees. Over the last few years, experience with an increasing number of P4 programs has shown the risks of the unsafe approach, which often leads to subtle software bugs.

This paper proposes SafeP4, a domain-specific language for programmable data planes in which all packet data is guaranteed to have a well-defined meaning and satisfy essential safety guarantees. We equip SafeP4 with a formal semantics and a static type system that statically guarantees header validity—a major source of safety bugs according to our analysis of real-world P4 programs. Statically ensuring header validity is challenging because the set of valid headers can be modified at runtime, making it a dynamic program property. Our type system achieves static safety by using a form of path-sensitive reasoning that tracks dynamic information from conditional statements, routing tables, and the control plane. Our empirical evaluation shows that SafeP4's type system can effectively eliminate common failures in many real-world programs.

2012 ACM Subject Classification Software and its engineering → Formal language definitions, Networks → Programming interfaces

Keywords and phrases P4, data plane programming, type systems

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

> I couldn't resist the temptation to put in a null reference [...] This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
>
> —Tony Hoare

Modern languages offer high-level abstractions that provide numerous benefits to programmers. Features such as type systems, structured control flow, objects, modules, etc. make it possible to express rich computations in terms of high-level structures and algorithms rather than machine-level code. Increasingly, many languages also offer fundamental safety guarantees—e.g., well–typed programs do not go wrong [21]—that make entire categories of programming errors simply impossible.

Unfortunately, although computer networks are critical infrastructure, providing provide the essential communication fabric that underpins nearly all modern systems, most network devices today are programmed using low-level languages that lack even basic safety guarantees. Unsurprisingly, networks are unreliable and remarkably insecure—e.g., the first step in the majority of cyberattacks involves compromising a router or other device [24, 17].

Over the past decade, there has been a remarkable shift to more flexible platforms in which the functionality of the network is specified in software. Early efforts related to

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:45
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

software-defined networking (SDN) [19, 6], focused on the control software that computes routes, balances load, and enforces security policies, and modeled the data plane as a simple pipeline of routing tables operating on a fixed set of packet formats. However, there has been recent interest in allowing the functionality of the data plane itself to be specified as a program—e.g., to implement new protocols, make more efficient use of hardware resources, or even relocate application-level functionality into the network [13, 12]. In particular, the P4 language [4] enables the functionality of a data plane to be programmed in terms of declarative abstractions such as header types, packet parsers, routing tables, and structured control flow that a compiler maps down to an underlying target device.

Unfortunately, while a number of P4's features were clearly inspired by designs found in modern languages, the central abstraction for representing packet data, header types, lacks basic safety guarantees. To a first approximation, a P4 header type can be thought of as a record with a field for each component of the header. For example, the header type for an IPv4 packet, would have a 4-bit version field, an 8-bit time-to-live field, two 32-bit fields for the source and destination addresses, and so on.

Unfortunately, according to the P4 language specification, instances of a header type may either be valid or invalid, and reading or writing an invalid header yields an undefined result. In practice, reading or writing an invalid header can lead to a variety of problems including dropping the packet when it should be forwarded, or even leaking information from one packet to the next! It also breaks portability, since the same program may behave differently when executed on different targets.

The choice to model the semantics of header types in an unsafe way was intended to make the language easier to implement on high-speed routers, which typically have limited amounts of memory. A typical P4 program might specify behavior for several dozen different protocols, but any particular packet is likely to contain only a small handfull of headers. It follows that if the compiler only needs to represent the valid headers at run-time, then memory requirements can be reduced. However, while it may have benefits for language implementers, the design is a disaster for programmers—it repeats Hoare's "billion-dollar mistake," and bakes an unsafe feature deep into the design of a language that has the potential to become the de-facto standard in a (multi) billion-dollar industry.

This paper investigates the design of a domain-specific language for programmable data planes in which all packet data is guaranteed to have a well-defined meaning and satisfy basic safety guarantees. In particular, we present SafeP4, a language with a precise semantics and a static type system that guarantees the validity of all headers read or written by the program. Although the type system is based on standard features, there are several aspects of its design that stand out. First, to facilitate tracking dependencies between headers—e.g. if the TCP header is valid, then the IPv4 will also be valid—SafeP4 has an expressive algebra of types that keeps tracks of fine-grained validity information. Second, to enable SafeP4 to accommodate the growing collection of extant P4 programs with only modest modifications, it uses a path-sensitive type system that incorporates information from conditional statements, routing tables, and the control plane to precisely track validity.

To evaluate our design for SafeP4, we formalized the language and its type system in a core calculus and proved the usual progress and preservation theorems. We also implemented the SafeP4 type system in an OCaml prototype, P4Check, and applied it to a suite of open-source programs found on GitHub such as `switch.p4`, a large P4 program that implements the features found in modern data center switches (specifically, it includes over four dozen different switching, routing, and tunneling protocols, as well as multicast, access control lists, among other features). We categorize common failures and, for programs that fail to type check, identify the root causes and apply repairs to make them well typed. We find that most programs can be repaired with low effort from programmers, typically by applying a modest number of simple repairs.

Overall, the main contributions of this paper are as follows:

- We propose SafeP4, a type-safe enhancement of the P4 language that eliminates all errors related to header validity.
- We formalize the syntax and semantics of SafeP4 in a core calculus and prove that the type system is sound.
- We implement our type checker in an OCaml prototype, P4Check.
- We evaluate our type system empirically on over a dozen real-world P4 programs and identify common errors and repairs.

The rest of this paper is organized as follows. Section 2 provides a more detailed introduction to P4 and elaborates on the problems this work addresses. Section 3 presents the design, operational semantics and type system of SafeP4 and reports our type safety result. The results of evaluating SafeP4 in the wild are presented in Section 4. Section 5 surveys related work and Section 6 summaries the paper and outlines topics for future work.

2 Background and Problem Statement

This section introduces the main features of P4 and highlights the problems caused by the unsafe semantics for header types using examples.

2.1 P4 Language

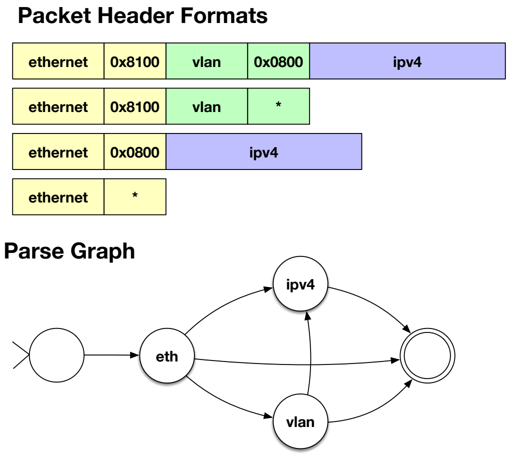
A P4 program comprises header type definitions, a parse graph, table and action declarations, and control flow specifying the order in which tables are applied to network packets.

Header Types and Instances The internal structure of each packet header are specified using P4's header types. For example, the first few lines of the following snippet of code:

```
header_type ethernet_t {
  fields {
    dstAddr: 48;
    srcAddr: 48;
    etherType: 16;
  }
}
header ethernet_t ethernet;
header ethernet_t inner_ethernet;
```

declare a header type (`ethernet_t`) for the Ethernet protocol with fields `dstAddr`, `srcAddr`, and `etherType`. The integer literals indicate the bit width of each field. The next two lines declare two `ethernet_t` instances, each with a distinct name (`ethernet` and `inner_ethernet`) and global scope. Ordinary packets usually have a single Ethernet header, but a tunneling protocol might maintain a second header for the nested packet.

Parsers The order in which packet headers are read from the stream of bits that represents the input packet is specified using P4's parsers, which are modeled using finite state machines. The code within each state may extract bits from the input stream, modify header instances, conditionally branch, and transition either to another state or into the `ingress` control, which has the effect of accepting the packet. Figure 1 depicts a visual representation of a parse graph for three common headers: Ethernet, VLAN, and IPv4. The instance `ethernet`




```
parser start {
  return parse_eth;
}
parser parse_eth {
 extract(ethernet);
 return select(latest.etherType){
   0x8100 : parse_vlan;
   0x0800 : parse_ipv4;
   default: ingress;
  }
}
parser parse_vlan {
  extract vlan {
  return select(latest.etherType){
    0x0800: parse_ipv4;
    default: ingress;
  }
}
parser parse_ipv4 {
  extract(ipv4);
  return ingress;
}
```

Figure 1 (Left) Header formats and parse graph that extracts an Ethernet header optionally followed by VLAN and/or IPv4 headers. (Right) P4 code implementing the same parser.

```
table forward {
 reads {
  ipv4 : valid;
  vlan : valid;
  ipv4.dstAddr: ternary;
 }
 actions = {
  nop;
  next_hop;
  remove;
 }
 default_action : nop();
}
```

Runtime Contents of `forward`

Pattern			Action	
`ipv4`	`vlan`	`ipv4.dstAddr`	Name	Data
`1`	`0`	`10.0.0.0/24`	`next_hop`	m
`0`	`1`	`0.0.0.0/0`	`remove`	

Figure 2 P4 tables.

is extracted first, optionally followed by a `vlan` instance, or an `ipv4` instance, or both. Extracting into an header instance populates its fields with bits from the input stream, advances the stream, and then marks the instance as valid.

Tables and Actions In most P4 programs, the bulk of the processing of each packet is performed using tables and actions. A table is defined in terms of (i) state it reads to determine a matching entry (if any), (ii) actions it may execute, and (iii) an optional default action it executes if no matching entry is found. For example, Figure 2 declares the table `forward` with a `reads` declaration that checks whether the `ipv4` and `vlan` instances are valid, and performs ternary matching on the `dstAddr` field of the `ipv4` instance. Tables are populated at run-time by the control plane, with entries that contain a pattern, an action, and action data. The pattern specifies the bits that should be compared against the values read in the table, the action is the name of a function, defined elsewhere in the P4 program, and the action data are values that serve as the arguments to that function.

Operationally, to process a packet, a table firsts scans its entries to locate a matching entry. If it finds an entry, the table is said to hit, and it executes the associated action.

```
action next_hop(src, dst) {
 modify_field(ethernet.srcAddr, src);
 modify_field(ethernet.dstAddr, dst);
 subtract_from_field(ipv4.ttl, 1);
}
```

```
action remove() {
 modify_field(ethernet.etherType,
              vlan.etherType);
 remove_header(vlan);
}
```

Figure 3 P4 actions.

Otherwise, if there are no matching entries, the packet is said to miss in the table and it executes the `default_action` (or an implicit no-op if the table lacks a default). For example, in Figure 2, the `forward` table is shown populated with two rules. The first rule tests whether `ipv4` is valid, `vlan` is invalid and then applies `next_hop` to an argument m (which stands for an Ethernet MAC address). The second rule checks that `ipv4` is invalid, then that `vlan` is valid, and then performs a ternary match using the all-wildcard pattern `0.0.0.0/0`.

Actions are functions containing sequences of primitive commands that perform operations such as adding and removing headers, assigning a value to a field, adding one field to another, etc. For example, Figure 3 depicts the `next_hop` and `remove` actions. The `next_hop` action assigns its argument `src`, which is provided by the control plane, to the `srcAddr` field of the `ethernet` instance while `remove` copies the `etherType` field from the `vlan` instance to the `ethernet` instance, and then removes the `vlan` instance. The `modify_field`$(h.f, e)$ primitive evaluates e and stores the result in header field $h.f$—e.g., `next_hop` decrements `ipv4.ttl` by one and saves the result. The `remove_header` primitive invalidates the header provided as an argument, and behaves like a no-op if the header is already invalid.

Control Control blocks use standard control-flow constructs to execute a pipeline of tables in sequence. They manage the order and conditions under which each table is executed. The `apply` command executes a table and conditionals branch on a boolean expression such as the validity of a header instance.

```
control ingress {
 if(valid(ipv4) or valid(vlan)) {
  apply(forward);
 }
}
```

This code applies the `forward` table only if one of `ipv4` or `vlan` is valid.

2.2 Common Bugs in P4 Programs

Having introduced the basic features of P4, we now present five categories of bugs found in open-source programs that arise due to reading and writing invalid headers—the main problem that SafeP4 addresses. There is one category for each of the following syntactic constructs: (1) parsers, (2) controls, (3) table reads, (4) table actions, and (5) default actions.

To identify the bugs we surveyed a benchmark suite of 15 research and industrial P4 programs that are publicly available on GitHub and compile to the BMv2 [22] backend. Later, in Section 4, we will report the number of occurrences of each of these categories in our benchmark suite detected by our approach.[1]

[1] We focus on $P4_{14}$ programs in this paper, but the issues we address also persist in the latest version of the langauge, $P4_{16}$. We did not consider $P4_{16}$ due to the smaller number of programs that are available, and for the pragmatic reason that our tool reuses an existing $P4_{14}$ front-end.

```
/* UNSAFE */



parser parse_ethernet {
 extract(ethernet);
 return select(latest.etherType) {
  0x0800 : parse_ipv4;
  default : ingress;

 }
}
parser parse_ipv4 {
 extract(ipv4);
 return select(latest.protocol) {
  6 : parse_tcp;
  default : ingress;

 }
}
```

```
/* SAFE */
parser_exception unsupported {
  parser_drop;
}
parser parse_ethernet {
 extract(ethernet);
 return select(latest.etherType) {
  0x0800 : parse_ipv4;
  default :
   parser_error unsupported;
  }
}
parser parse_ipv4 {
 extract(ipv4);
 return select(latest.protocol) {
 6 : parse_tcp;
 default : parser_error unsupported;
  }
}
```

```
parser parse_tcp {
  extract(tcp);
  return ingress;
}
```

```
control ingress {
  if (tcp.syn == 1 and tcp.ack ==
       1) {
      ...
}
```

Figure 4 Left: unsafe code in NetHCF; Right: our type-safe fix; Bottom: common code.

2.2.1 Parser Bugs

The first class of errors is due to the parser being too conservative about dropping malformed packets, which increases the set of headers that may be invalid in the control pipeline. In most programs, the parser chooses which headers to `extract` based on the fields of previously-extracted headers. For example, after extracting the IPv4 header, it is common to branch on the `protocol` field to decide whether to extract the TCP (`0x06`) or UDP (`0x11`) header. However, these alternatives do not capture all possible values for the `protocol` field—e.g., consider an OSPF packet (`0x59`). A common programming pattern is to add a `default` case to ensure that all possible packets are handled. However, the oft-forgotten next step is to drop the offending packet in the `ingress` pipeline! Failure to do so leads to undefined behavior, whenever the programmer attempts to access the TCP or UDP header.

An example from the NetHCF [31, 2] codebase illustrates this bug. NetHCF is a research tool designed to combat TCP spoofing. The program detects and drops spoofed packets by learning allowable hop-count values for the IPv4 address of each host and discarding packets from those hosts that have invalid hop-counts. As shown in Figure 4, the parser handles IPv4-TCP packets and redirects all other packets to the `ingress` control. Unfortunately, the `ingress` control executes no check on whether `tcp` is valid before accessing access `tcp.syn` to check whether it is equal to `1`. This is unsafe since `tcp` is not guaranteed to be valid.

To fix this bug, we can define a parser exception, `unsupported`, with an handler that drops packets, thereby protecting the `ingress` from having to handle unexpected packets.

2.2.2 Control Bugs

Another common bug occurs when a table is executed in a context in which the instances referenced by that table are not guaranteed to be valid. This bug can be seen in the open-source code for NetCache [11, 13], a P4 system that implements a load-balancing cache.

```
/* UNSAFE */
control ingress {

 process_cache();
 process_value();

 apply(ipv4_route);
}
```

```
/* SAFE */
control ingress {
  if (valid(nc_hdr)) {
     process_cache();
     process_value();
 }
 apply(ipv4_route);
}
```

```
control process_cache {
    apply(check_cache_exist);
    ...
}
```

```
table check_cache_exist {
    reads { nc_hdr.key }
    actions { ... }
}
```

Figure 5 Left: unsafe code in NetCache; Right: our type-safe fix; Bottom: Common code

The parser for NetCache reserves a specific port (8888) to handle its special-purpose traffic, a condition that is built into the parser, which extracts `nc_hdr` (i.e., the NetCache-specific header) only when UDP traffic arrives from port 8888. Otherwise it performs standard L2 and L3 routing. Unfortunately, the `ingress` control node tries to access `nc_hdr` before checking that it is valid! Specifically, the `reads` declaration for `check_cache_exists` table, which is executed first, presupposes that `nc_hdr` is valid. The invocation of the `process_value` table (not shown) contains another instance of the same bug.

To fix these bugs, we can wrap the calls to `process_cache` and `process_value` in an conditional that checks the validity of the header `nc_hdr`. This ensures that `nc_hdr` is valid when `process_cache` refers it.

2.2.3 Table Reads Bugs

A similar bug arises in programs that contain tables that first match on the validity of certain header instances before matching on the fields of those instances. The advantage of this approach is that multiple types of packets can be processed in a single table, which saves memory. However, if implemented incorrectly, it can lead to a bug, in which the `reads` declaration matches on the `exact` bits in a field from a header that may not be valid!

The `switch.p4` program is a "realistic production switch" [16] developed by Barefoot Networks, meant to be used "as-is, or as a starting point for more advanced switches" [16]. It provides many core network features, such as L2 switching, L3 routing, LAG, ECMP, VLAN, NVGRE, Geneve, GRE, ACL, and MPLS, among others [16, 23]. An archetypal example of table reads bugs is the `port_vlan_mapping` table of Switch (Figure 6). This table is invoked in a context, where it is not known which of the VLAN tags is valid, despite containing references to both `vlan_tag_[0]` and `vlan_tag_[1]` in its `reads` declaration, so the programmer has guarded the references to `vlan_tag_[`i`].vid` with keys that test the validity of `vlan_tag_[`i`]`, for $i = 1, 2$.

Guarding the accesses with valid matches in this way allows the control plane to install rules with keys such as $(0, 0, 1, 2)$, which checks the validity of each header before accessing them to perform the match. However, it is impossible, with the table as it is written, for the control plane to install a rule that will avoid reading the value of an invalid header. The first match that is performed at runtime is to verify that the `vlan_tag_[0]` instance is invalid. Then the `exact` match kind will try and check that `vlan_tag_[0].vid` $= 0$, even though the instance is invalid! This attempt to access an invalid header is results in undefined behavior, and is therefore a bug.

```
/* UNSAFE */
table port_vlan_mapping {
 reads {
  vlan_tag_[0] : valid;
  vlan_tag_[0].vid : exact;
  vlan_tag_[1] : valid;
  vlan_tag_[1].vid : exact;
 } ...
}
```

```
/* SAFE */
table port_vlan_mapping {
 reads {
  vlan_tag_[0] : valid;
  vlan_tag_[0].vid : ternary;
  vlan_tag_[1] : valid;
  vlan_tag_[1].vid : ternary;
 } ...
}
```

Figure 6 Left: a table in `switch.p4` with unprotected conditional reads; Right: our type-safe fix.

Its worthy to note that this code is not actually buggy on some targets—in particular, if invalid headers are initialized with `0`. However, this doesn't conform to the language specification, and therefore isn't portable to other hardware devices.

The naive solution to fix this bug is to refactor the table into four different tables (one for each combination of validity bits) and then check the validity of each header before the tables are invoked. This is a perfectly safe fix, but it can result in an exponential blowup in the number of tables, which is clearly undesirable for efficiency reasons, and because it complicates the control plane.

Fortunately, rather than factoring the table into four tables, we can replace the `exact` match-kinds with `ternary` match-kinds, which permit matching with wildcards. Since matching a wildcard does not evaluate the expression, then match rules like $(\texttt{0}, *, \texttt{0}, *)$ will cause the switch to safely skip evaluation of `vlan_tag_[`i`].vid`.

In order for this solution to typecheck, we need to assume that the control plane is well-behaved—i.e. that it will install wildcards for the `ternary` matches whenever the header is invalid. In our implementation, we print a warning whenever we make this kind of assumption so that the programmer can confirm that the control plane is well-behaved.

2.2.4 Table Action Bugs

Another prevalent bug, in our experience, arises when distinct actions in a table require different (and possible mutually exclusive) headers to be valid. This can lead to two problems: (i) the control plane can populate the table with unsafe match-action rules, and (i) there may be no validity checks that we can add to the control to make all of the actions typecheck.

The `fabric_ingress_dst_lkp` table (Figure 7) in `switch.p4` provides an example of this misbehavior. The `fabric_ingress_dst_lkp` table reads the value of `fabric_hdr.dstDevice` and then invokes one of several actions: `term_cpu_packet`, `term_fabric_unicast_packet`, or `term_fabric_multicast_packet`. Respectively, these actions require the `fabric_hdr_cpu`, `fabric_hdr_unicast`, and `fabric_hdr_multicast` (respectively) headers to be valid. Unfortunately the validity of these headers is mutually exclusive[2].

Since `fabric_header_cpu`, `fabric_header_unicast`, and `fabric_header_multicast` are mutually exclusive, there is no context that makes this table safe. The only facility the table provides to determine which action should be called is `fabric_hdr.dstDevice`. However, the parser chooses which header to parse based on the value of `fabric_hdr.packetType`, and the portion of the program that precedes the call to `fabric_ingress_dst_lkp` does not establish any relationship between `fabric_hdr.dstDevice` and `fabric_hdr.dst_device`. In fact

[2] There are other actions in the real `fabric_ingress_dst_lkp`, but these three actions demonstrate the core of the problem.

```
/* UNSAFE */
table fabric_ingress_dst_lkp {
 reads {
  fabric_hdr.dstDevice : exact;
 }


 actions {
  term_cpu_packet;
  term_fabric_unicast_packet;
  term_fabric_multicast_packet;
 }
}
```

```
/* SAFE */
table fabric_ingress_dst_lkp {
 reads {
  fabric_hdr.dstDevice : exact;
  fabric_hdr_cpu : valid;
  fabric_hdr_unicast: valid;
  fabric_hdr_multicast: valid;
 }
 actions {
  term_cpu_packet;
  term_fabric_unicast_packet;
  term_fabric_multicast_packet;
 }
}
```

Figure 7 Left: unsafe code in `switch.p4`; Right: our type-safe fix.

the program does not even reference these locations before the call to `fabric_ingress_dst_lkp`. Hence, the correctness of this table relies on well-formed input packets, which is not consistent with real switches, which can receive any sequence of bits that arrive "on the wire."

To fix this bug there are two possible solutions: (1) refactor the table into three tables whose applications are guarded with validity checks for the required headers, or (2) include validity matches in the reads declaration. In general, to avoid the exponential blowup in the number of on-switch tables, we proceed with option (2) as shown on the right side of Figure 7.

In order to type check this solution, we need to make an assumption about the way the control plane will populate the table. Concretely, if an action a only typechecks if a header h is valid, and h is not necessarily valid when the table is applied, we assume that the control plane will only call a if h is matched as valid. For example, `fabric_hdr_cpu` is not known to be valid when (the fixed version of) `fabric_ingress_dst_lkp` is applied, so we assume that the control plane will only call action `term_cpu_packet` when `fabric_hdr_cpu` is matched as valid. Again, our implementation prints these assumptions as warnings to the programmer, so they can confirm that the control plane will satisfy these assumptions.

2.2.5 Default Action Bugs

Finally, the *default action* errors occur when the programmer incorrectly assumes that a table performs some action when a packet misses. The NetCache program (described in Section 2.2.2) exhibits an example of this bug, too. The bug is shown in Figure 8, where the table `add_value_header_1` is expected to make the `nc_value_1` header valid, which is done in the `add_value_header_1_act` action. The control plane may refuse to add any rules to the table, which would cause all packets to miss, meaning that the `add_value_header_1_act` action would never be called and `nc_value_1` may not be valid. To fix this error, we simply set the default action for the table to `add_value_header_1_act`, which will force the table to remove the header no matter what the controller does.

2.3 A Typing Discipline to Eliminate Invalid References

In this paper, we propose a type system to increase the safety of P4 programs by detecting and preventing the classes of bugs defined in Section 2.2. These classes of bugs all manifest when a program attempts to access an invalid header—differentiating themselves only in their syntactic provenance. The type system that we present in the next section uses a path-sensitive analysis and occurrence typing [30], to keep track of which headers are guaranteed

```
/* UNSAFE */
table add_value_header_1 {
 actions {
  add_value_header_1_act;
 }

}
```

```
/* SAFE */
table add_value_header_1 {
 actions {
  add_value_header_1_act;
 }
 default_action :
    add_value_header_1_act();
}
```

Figure 8 Left: unsafe code in NetCache; Right: our type-safe fix.

```
if (ethernet.etherType == 0x0800) {
 apply(ipv4_table);
} else if (ethernet.etherType == 0x086DD) {
 apply(ipv6_table);
}
```

```
if (valid(ipv4)) {
 apply(ipv4_table);
} else if (valid(ipv6)) {
 apply(ipv6_table);
}
```

Figure 9 Left: data-dependent header validation; Right: syntactic header validation.

to be available at any program point, and rejects programs that reference headers that might be uninitialized; thus, preventing all references to invalid headers.

Of course, in general, the problem of deciding header-validity can depend on arbitrary data, so a simple type system cannot hope to fully determine all scenarios when an instance will be valid. Indeed, programmers often use a variety of data-dependent checks to ensure safety. For instance, the control snippet shown on the left-hand side of Figure 9 will not not produce undefined behavior, given a parser that chooses between parsing an `ipv4` header when `ethernet.etherType` is `0x0800`, an `ipv6` header when `ethernet.etherType` is `0x86DD`, and throws a parser error otherwise.

While this code is safe in this very specific context, it quickly becomes unsafe when ported to other contexts. For example in `switch.p4`, which performs tunneling, the egress control node copies the `inner_ethernet` header into the `ethernet`; however the `inner_ethernet` header may not be valid at the program point where the copy is performed. This behavior is left undefined [7], so targets are free to read arbitrary bits, in which case it could decide to call the `ipv4_table` despite `ipv4` being invalid!

To improve the maintainability and portability of the code, we can replace the data-dependent checks with validity checks, as illustrated by the control snippet shown on the right-hand side of Figure 9. The validity checks assert precisely the preconditions for calling each table, so that no matter what context this code snippet is called in, it is impossible for the `ipv4_table` to be called when the `ipv4` header is invalid.

In the next section, we develop a core calculus for SafeP4 with a type system that eliminates references to invalid headers, encouraging programers to replace data-dependent checks with header-validity checks.

s

3 SafeP4

This section discusses our design goals for SafeP4 and the choices we made to accommodate them, and formalizes the language's syntax, small-step semantics, and type system.

3.1 Design

Our primary design goal for SafeP4 is to develop a core calculus that models the main features of $P4_{14}$ and $P4_{16}$, while guaranteeing that all data from packet headers is manipulated in a safe and well-defined manner. We draw inspiration from Featherweight Java [10]—i.e., we model the essential features of P4, but prune away unnecessary complexity. The result is a minimal and elegant calculus that is easy to reason about, but can still express a large number of real-world data plane programs. For instance, P4 and SafeP4 both achieve protocol independence by allowing the programmer to specify the types of packet headers and their order in the bit stream. Similarly, SafeP4 mimics P4's use of tables to interface with the controller and decide, at runtime what actions to execute. Hence, we model the integral aspect of the interface between the control plane and the data plane.

So what features does SafeP4 prune away? We omit a number of constructs that are secondary to how packets are processed—e.g., `field_list_calculations`, `parser_exceptions`, `counters`, `meters`, etc. It would be relatively straightforward to add these to the calculus—indeed, most are already handled in our prototype—at the cost of making it more complicated. We also modify or distill several aspects of P4. For instance, P4 separates the parsing phase and the control phase. Rather than unnecessarily complicating the syntax of SafeP4, we allow the syntactic objects that represent parsers and controls to be freely mixed. We make a similar simplification in actions, informally enforcing which primitive commands can be invoked within actions (e.g., field modification, but not conditionals).

Another challenge arises in trying to model core behaviors of both $P4_{14}$ and $P4_{16}$, in that they each have different type systems and behaviors for evaluating expressions! Our calculus abstracts away expression typing and syntax variants by assuming that we are given a set of constants k that can represent values like `0` or `True`, or operators such as `&&` and `?:`. We also assume that these operators are assigned appropriate (i.e., sound) types. With these features in hand, one can instantiate our type system over arbitrary constants.

Another departure from P4 is related to *add* command, which presents a complication for our expression types. The analogous `add_header` action in P4 simply modifies the validity bit, without initializing any of the fields. This means that accessing any of the header fields before they have been manually initialized reads a non-deterministic value. Our calculus neatly sidesteps this issue by defining the semantics of the $add(h)$ primitive to initialize each of the fields of h to a default value. We assume that along with our type constants there is a function *init* that accepts a header type η and produces a header instance of type η with all fields set to their default value. Note that we could have instead modified our type system to keep track of the definedness of header fields as well as their validity. However, for simplicity we choose to focus on header validity in this paper.

The portion of our type system that analyzes header validity, requires some way of keeping track of which headers are valid. Naively, we can keep track of a set of which headers are guaranteed to be valid on all program paths, and reject programs that reference headers not in this set. However, this coarse-grained approach would lead to a large number of false positives. For instance, the parser shown in Figure 1 parses an `ethernet` header and then either boots to `ingress` or parses an `ipv4` header and then either proceeds to the `ingress` or parses an `vlan` header. Hence, at the `ingress` node, the only header that is guaranteed to be valid is the `ethernet` header! However, it is certainly safe to write an `ingress` program that references the `vlan` header after checking it was valid. To reflect this in the type system we introduce a special construct called $valid(h)\ c_1\ else\ c_2$, which executes c_1 if h is valid and c_2 otherwise. When we type check this command, following previous work on occurrence typing [30], we check c_1 with the additional fact that h is valid, and we check c_2 with the additional fact that c_2 is valid.

Even with this enhancement, this type system would still be overly restrictive. To see why, let us augment the parser from Figure 1 with the ability to parse TCP and UDP packets: after parsing the `ipv4` header, the parser can optionally extract the `vlan`, `tcp`, or `udp` header and then boot to ingress. Now suppose that we have a table `tcp_table` that refers to both `ipv4` and `tcp` in its `reads` declaration, and that `tcp_table` is (unsafely) applied immediately in the `ingress`. Because the validity of `tcp` implies the validity of `ipv4`, it should be safe to check the validity of `tcp` and then apply `tcp_table`. However, using the representation of valid headers as a set, we would need to ascertain the validity of `ipv4` and of `tcp`.

To solve this problem, we enrich our type representation to keep track of dependencies between headers. More specifically, rather than representing all headers guaranteed to be valid in a set, we use a finer-grained representation—a set of sets of headers that might be valid at the current program point. For a given header reference to be safe, it must to be a member of all possible sets of headers—i.e., it must be valid on all paths through the program that reach the reference.

Overall, the combination of an expressive language of types and a simple version of of occurrence typing allows us to capture dependencies between headers and perform useful static analysis of the dynamic property of header validity.

The final challenge with formally modelling P4 is its interface with the control-plane, which populates the tables and provides arguments to the actions. While the control-plane only methodology for managing switch behavior is to populate the match-action tables with forwarding entries, it is perfectly capable of producing undefined behavior. However, if we assume that the controller is well-intentioned, we can prove the safety of more programs.

In our formalization, to streamline the presentation, we model the control plane as a function $\mathcal{CA}(t, H) = (a_i, \bar{v})$ that takes in a table t and the current headers H and produces the action to call a_i and the (possibly empty) action data arguments $\bar{v}$. We also use a function $\mathcal{CV}(t) = (\bar{S}, \bar{e})$ that analyzes a table t and produces a list of match key expressions $\bar{e}$ that must be evaluated when the table is invoked, and a list of sets of valid headers $\bar{S}$, one set for each action, that can be safely assumed valid when the entries are populated by the control plane. Together, these functions model the runtime interface between the switch and the controller. In order to prove progress and preservation, we assume that $\mathcal{CV}$ and $\mathcal{CA}$ satisfy three simple correctness properties—see Appendix C.1 for details.

3.2 Syntax

The syntax of SafeP4 is shown in Figure 10. To lighten the notation, we write $\bar{x}$ as shorthand for a (possibly empty) sequence $x_1, ..., x_n$.

A SafeP4 program consists of a sequence of declarations $\bar{d}$ and a command c. The set of declarations includes header types, header instances, and tables. Header type declarations describe the format of individual headers and are defined in terms of a name and a sequence of field declarations. The notation "$f : \tau$" indicates that field f has type τ. We let η range over header types. A header instance declaration assigns a name h to a header type η. The map $\mathcal{HT}$ encodes the (global) mapping between header instances and header types. Table declarations $t(\bar{e}, \bar{a})$, are defined in terms of a sequence of match-key expressions $\bar{e}$ read in the table, and a sequence of actions $\bar{a}$. The notation $t.reads$ denotes the expressions and $t.actions$ denotes the actions.

Actions are written as (uncurried) λ-abstractions. An action $\lambda\bar{x}.\ c$ declares a (possibly empty) sequence of parameters, drawn from a fresh set of names, which are in scope for the command c. The run-time arguments for actions are chosen by the control plane. Note that

Commands

c ::=

$\mid$	$extract(h)$	EXTRACTION
$\mid$	$emit(h)$	DEPARSING
$\mid$	$c_1; c_2$	SEQUENCE*
$\mid$	$if(e)\ c_1\ else\ c_2$	CONDITIONAL
$\mid$	$valid(h)\ c_1\ else\ c_2$	VALIDITY
$\mid$	$t.apply()$	APPLICATION
$\mid$	$skip$	SKIP
$\mid$	$add(h)$	ADDITION*
$\mid$	$remove(h)$	REMOVAL*
$\mid$	$h.f = e$	MODIFICATION*

Actions

$a ::= \lambda\bar{x}.c$ ACTION

Expressions

e ::=

$\mid$	v	VALUES
$\mid$	$h.f$	HEADER FIELD
$\mid$	x	VARIABLE
$\mid$	k^n	CONSTANT

Declarations

d ::=

$\mid$	$t(\bar{e}, \bar{a})$	TABLE
$\mid$	$\eta\ \{\overline{f : \tau}\}$	HEADER TYPE
$\mid$	$h \mapsto \eta$	INSTANTIATION

Program

$\mathcal{P}$ ::= $(\bar{d}, c)$ PROGRAM

Values

$v \in V$

Constants

$k \in K$

Header Types

Θ ::=

$\mid$	0	CONTRADICTION
$\mid$	1	EMPTY
$\mid$	h	INSTANCE
$\mid$	$\Theta_1 \cdot \Theta_2$	CONCATENATION
$\mid$	$\Theta_1 + \Theta_2$	CHOICE

Action Types

$\alpha ::= \bar{\tau} \to \Theta$

Expression Types

τ ::= `Bool` $\mid \bar{\tau} \to \tau \mid \ldots$

Figure 10 Syntax of SafeP4

we artificially restrict the commands that can be called in the body of the action to addition, removal, modification and sequence; these actions are identified with an asterisk in Figure 10.

The calculus provides commands for extracting (*extract*), creating (*add*), removing (*remove*), and modifying ($h.f = e$) header instances. The *emit* command serializes a packet header back into a bit sequence (*emit*). The *if*-statement conditionally executes one of two commands based on the value of a boolean condition. Similarly, the *valid*-statement branches on the validity of h. Table application commands ($t.apply()$) are used to invoke a table t in the current state. The *skip* command is a no-op.

The only built-in expressions in SafeP4 are variables x and header fields, written $h.f$. We let v range over values and assume a collection of n-ary constant operators $k^n \in K$.

For simplicity, we assume that every header referenced in an expression has a corresponding instance declaration. We also assume that header instance names h, header type names η, variable names x, and table names t are drawn from disjoint sets of names H,E,V, and T respectively and that each name is declared only once.

3.3 Type System

SafeP4 provides two main kinds of types, basic types τ and header types Θ as shown in Figure 10. We assume that the set of basic types includes booleans (for conditionals) as well as tuples and function types (for actions).

A header type Θ represents a set of possible co-valid header instances. The type 0 denotes

$$
\begin{aligned}
[\![0]\!] &= \{\} \\
[\![1]\!] &= \{\{\}\} \\
[\![h]\!] &= \{\{h\}\} \\
[\![\Theta_1 \cdot \Theta_2]\!] &= [\![\Theta_1]\!] \bullet [\![\Theta_2]\!] \\
[\![\Theta_1 + \Theta_2]\!] &= [\![\Theta_1]\!] \cup [\![\Theta_2]\!]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}(h, f_i) &= \tau_i && \textit{Field lookup} \\
\mathcal{A}(a) &= \lambda\bar{x} : \bar{\tau}.\ c && \textit{Action lookup} \\
\mathcal{CA}(H, t) &= (a_i, \bar{v}) && \textit{Control-plane actions} \\
\mathcal{CV}(t) &= (\bar{S}, \bar{e}) && \textit{Control-plane validity}
\end{aligned}
$$

Figure 11 Semantics of header types (left) and auxiliary functions (right).

the empty set. This type arises when there are unsatisfiable assumptions about which headers are valid. The type 1 denotes the singleton denoting the empty set of headers. It describes the type of the initial state of the program. The type h denotes a singleton set, $\{\{h\}\}$—i.e., states where only h is valid. The type $\Theta_1 \cdot \Theta_1$ denotes the set obtained by combining headers from Θ_1 and Θ_2—i.e., a product or concatenation. Finally, the type $\Theta_1 + \Theta_2$ denotes the union of Θ_1 or Θ_2, which intuitively, represents an alternative.

The semantics of header types, $[\![\Theta]\!]$, is defined by the equations in Figure 11. Intuitively, each subset represents one alternative set of headers that may be valid. For example, the header type $\texttt{eth} \cdot (\texttt{ipv4} + 1)$ denotes the set $\{\{\texttt{eth}, \texttt{ipv4}\}, \{\texttt{eth}\}\}$.

To formulate the typing rules for SafeP4, we also define a set of operations on header types: `Restrict`, `NegRestrict`, `Includes`, `Remove`, and `Empty`. The restrict operator `Restrict` Θ h recursively traverses Θ and keeps only those choices in which h is contained, mapping all others to 0. Semantically this has the effect of throwing out the subsets of $[\![\Theta]\!]$ that do not contain h. Dually `NegRestrict` Θ h produces only those choices/subsets where h is invalid. `Includes` Θ h traverses Θ and checks that h is always valid. Semantically this says that h is a member of every element of $[\![\Theta]\!]$. `Remove` Θ h removes h from every path, which means, semantically that it removes h from ever element of $[\![\Theta]\!]$. Finally, `Empty` Θ checks whether Θ denotes the empty set. An in-depth treatment of these operators, with proofs of all of these claims can be found in Appendix B.

3.3.1 Typing Judgement

The main typing judgement has the form $\Gamma \vdash c : \Theta \Rrightarrow \Theta'$, which means that in variable context Γ, if c is executed in the header context Θ, then a header instance type Θ' is assigned. Intuitively, Θ encodes the sets of headers that may be valid when type checking a command. Γ is a standard type environment which maps variables x to type τ. If there exists Θ' such that $\Gamma \vdash c : \Theta \Rrightarrow \Theta'$, we say that c is well-typed in Θ.

The typing rules rely on several auxiliary definitions shown in Figure 11. The field type lookup function $\mathcal{F}(h, f_i)$ returns the type assigned to a field f_i in header h. The action lookup function $\mathcal{A}(a)$ returns the action definition $\lambda\bar{x} : \bar{\tau}.\ c$ for action a. Finally, the function $\mathcal{CA}(t, H)$ computes the run-time actions for table t, while $\mathcal{CV}(t)$ computes t's assumptions about validity. Both of these are assumed to be instantiated by the control plane in a way that satisfies basic correctness properties—see Appendix C.1.

The typing rules for commands are presented in Figure 12. The rule T-Zero gives a command an arbitrary output type if the input type is empty. The rules T-Skip are T-Seq are standard. The rule T-If a path-sensitive union type between the type computed for each branch. The rule T-IfValid is similar, but leverages knowledge about the validity of

$$\frac{\texttt{Empty}\ \Theta_1}{\Gamma \vdash c : \Theta_1 \Rrightarrow \Theta_2}\ \text{T-Zero} \qquad \frac{}{\Gamma \vdash \mathit{skip} : \Theta \Rrightarrow \Theta}\ \text{T-Skip}$$

$$\frac{\Gamma \vdash c_1 : \Theta \Rrightarrow \Theta_1 \qquad \Gamma \vdash c_2 : \Theta_1 \Rrightarrow \Theta_2}{\Gamma \vdash c_1; c_2 : \Theta \Rrightarrow \Theta_2}\ \text{T-Seq}$$

$$\frac{\Gamma; \Theta \vdash e : \mathit{Bool} \qquad \Gamma \vdash c_1 : \Theta \Rrightarrow \Theta_1 \qquad \Gamma \vdash c_2 : \Theta \Rrightarrow \Theta_2}{\Gamma \vdash \mathit{if}\ (e)\ c_1\ \mathit{else}\ c_2 : \Theta \Rrightarrow \Theta_1 + \Theta_2}\ \text{T-If}$$

$$\frac{\Gamma \vdash c_1 : \texttt{Restrict}\ \Theta\ h \Rrightarrow \Theta_1 \qquad \Gamma \vdash c_2 : \texttt{NegRestrict}\ \Theta\ h \Rrightarrow \Theta_2}{\Gamma \vdash \mathit{valid}(h)\ c_1\ \mathit{else}\ c_2 : \Theta \Rrightarrow \Theta_1 + \Theta_2}\ \text{T-IfValid}$$

$$\frac{\texttt{Includes}\ \Theta\ h \qquad \mathcal{F}(h, f) = \tau_i \qquad \Gamma; \Theta \vdash e : \tau_i}{\Gamma \vdash h.f = e : \Theta \Rrightarrow \Theta}\ \text{T-Mod}$$

$$\frac{}{\Gamma \vdash \mathit{extract}(h) : \Theta \Rrightarrow \Theta \cdot h}\ \text{T-Extr} \qquad \frac{}{\Gamma \vdash \mathit{emit}(h) : \Theta \Rrightarrow \Theta}\ \text{T-Emit}$$

$$\frac{}{\Gamma \vdash \mathit{add}(h) : \Theta \Rrightarrow \Theta \cdot h}\ \text{T-Add} \qquad \frac{}{\Gamma \vdash \mathit{remove}(h) : \Theta \Rrightarrow \texttt{Remove}\ \Theta\ h}\ \text{T-Rem}$$

$$\frac{\mathcal{CV}(t) = (\bar{S}, \bar{e}) \qquad t.\mathit{actions} = \bar{a} \qquad \cdot; \Theta \vdash e_i : \tau_i \quad \text{for } e_i \in \bar{e} \qquad \texttt{Restrict}\ \Theta\ S_i \vdash a_i : \bar{\tau}_i \to \Theta'_i \quad \text{for } a_i \in \bar{a}}{\Gamma \vdash t.\mathit{apply}() : \Theta \Rrightarrow \left(\sum \Theta'_i\right)}\ \text{T-Apply}$$

Figure 12 Command typing Rules for SafeP4

$$\frac{\Gamma, \bar{x} : \bar{\tau} \vdash c : \Theta \Rrightarrow \Theta'}{\Gamma; \Theta \vdash \lambda\ \bar{x} : \bar{\tau}.c : \bar{\tau} \to \Theta'}\ \text{(T-Action)}$$

Figure 13 Action typing rule for SafeP4

h. So the true branch c_1 is checked in the context $\texttt{Restrict}\ \Theta\ h$, and the false branch c_2 is checked in the context $\texttt{NegRestrict}\ \Theta\ h$. The top-level output type is the union of the resulting output types for c_1 and c_2. The rule T-Mod checks that h is guaranteed to be valid using the `Includes` operator, and uses the auxiliary function $\mathcal{F}$ to obtain the type assigned to $h.f$. Note that the set of valid headers does not change when evaluating an assignment, so the output and input types are identical. The rules T-Extr and T-Add assign header extractions and header additions the type $\Theta \cdot h$, reflecting the fact that h is valid after the command executes. Emitting packet headers does not change the set of valid headers, which is captured by rule T-Emit. The typing rule T-Rem uses the `Remove` operator to remove h from the input type Θ. Finally, the rule T-Apply checks table applications. To understand how it works, let us first consider a simpler but less precise typing rule:

$$\frac{t.\mathit{reads} = \bar{e} \qquad \cdot; \Theta \vdash e_i : \tau_i \quad \text{for } e_i \in \bar{e} \qquad t.\mathit{actions} = \bar{a} \qquad \cdot; \Theta \vdash a_i : \bar{\tau}_i \to \Theta'_i \quad \text{for } a_i \in \bar{a}}{\cdot \vdash t.\mathit{apply}() : \Theta \Rrightarrow \left(\sum \Theta'_i\right)}$$

Intuitively, this rule says that to type check an table application, we check each expression it reads and each of its actions. The final header type is the union of the types computed for

$$\frac{\texttt{typeof}(k) = \bar{\tau} \to \tau' \qquad \Gamma;\Theta \vdash e_i : \tau_i}{\Gamma;\Theta \vdash k(\bar{e}) : \tau'}\ \text{T-Const} \qquad \frac{x : \tau \in \Gamma}{\Gamma;\Theta \vdash x : \tau}\ \text{T-Var} \qquad \frac{\texttt{Includes}\ \Theta\ h \qquad \mathcal{F}(h,f) = \tau}{\Gamma;\Theta \vdash h.f : \tau}\ \text{T-Field} \quad (\text{T-Field})$$

Figure 14 Expression typing rules for SafeP4

the actions. To put it another way, it models table application as a non-deterministic choice between its actions. However, while this rule is sound, it is overly conservative. In particular, it does not model the fact that the control plane often uses header validity bits to control which actions are executed. Hence, The actual typing rule, T-Apply, is parameterized on a function $\mathcal{CV}(t)$ that models the choices made by the control plane, returning for each action a_i, a set of headers S_i that can be assumed valid when type checking a_i, as well as a subset of the expressions read by the table — e.g., excluding expressions that can be wildcarded when certain validity bits are false.

The typing judgement for actions (Figure 13) is of the form $\Gamma;\Theta \vdash a : \bar{\tau} \to \Theta$, meaning that a has type $\bar{\tau} \to \Theta$ in variable context Γ and header context Θ. Given a variable context Γ and header type Θ, an action $\lambda\bar{x}.c$ encodes a function of type $\bar{\tau} \to \Theta'$, so long as the body c is well-typed in the context where Γ is extended with $x_i : \tau_i$ for every i.

The typing rules for expressions are shown in Figure 14. Constants are typechecked according to rule T-Constant, as long as each expression that is passed as an argument to the constant k has the type required by the `typeof` function. The rule T-Var is standard.

3.4 Operational Semantics

We now present the small-step operational semantics of SafeP4. We define the operational semantics for commands in terms of four-tuples $\langle I, O, H, c\rangle$, where I is the input bit stream (which is assumed to be infinite for simplicity), O is the output bit stream, H is a map that associates each valid header instance with a records containing the values of each field, and c is the command to be evaluated. The reduction rules are presented in Figure 15.

The command $\mathit{extract}(h)$ evaluates via the rule E-Extr, which looks up the header type in $\mathcal{HT}$ and then invokes corresponding deserialization function. The deserialized header value v is added to to the map of valid header instances, H. For example, assuming the header type $\eta = \{f : \mathit{bit}\langle 3\rangle;\ g : \mathit{bit}\langle 2\rangle;\}$ has two fields f and g and $I = 11000B$ where B is the rest of the bit stream following, then $\mathit{deserialize}_\eta(I) = (\{f = \mathit{110};\ g = \mathit{00};\}, B)$.

The rule E-Emit serializes a header instance h back into a bit stream. It first looks up the corresponding header type and header value in the header table $\mathcal{HT}$ and the map of valid headers respectively. The header value is then passed to the serialization function for the header type to produce a bit sequence that is appended to the output bit stream. Similarly, we assume that a serialization function is defined for every header type, which takes the bit values of the fields of a header value and concatenates them to produce a single bit sequence. We adopt the semantics of P4 with respect to emitting invalid headers. Emitting an invalid header instance, i.e., a header instance which has not been added or extracted, has just a no-op without any effect on the output bit stream (rule E-EmitInvalid). Notice also that the header remains unchanged in H.

Sequential composition reduces left to right, i.e., the left command needs to be reduced to *skip* before the right command can be reduced (rule E-Seq). The evaluation of conditionals (rules E-If, E-IfTrue, E-IfFalse) is standard. Both E-Seq and E-IF are relegated to the

E-Extr
$$\frac{\mathcal{HT}(h) = \eta \qquad deserialize_\eta(I) = (v, I')}{\langle I, O, H, extract(h)\rangle \to \langle I', O, H[h \mapsto v], skip\rangle}$$

E-Mod
$$\frac{H(h) = r \qquad r' = \{r \ with \ f = v\}}{\langle I, O, H, h.f = v\rangle \to \langle I, O, H[h \mapsto r'], skip\rangle}$$

E-Emit
$$\frac{\mathcal{HT}(h) = \eta \qquad serialize_\eta(H(h)) = \bar{B}}{\langle I, O, H, emit(h)\rangle \to \langle I, O.\bar{B}, H, skip\rangle}$$

E-Apply
$$\frac{\mathcal{CA}(H, t) = (a_i, \bar{v}) \qquad \mathcal{A}(a_i) = \lambda\bar{x}.c_i}{\langle I, O, H, t.apply()\rangle \to \langle I, O, H, c_i[\bar{v}/\bar{x}]\rangle}$$

E-EmitInvalid
$$\frac{h \notin dom(H)}{\langle I, O, H, emit(h)\rangle \to \langle I, O, H, skip\rangle}$$

E-Add
$$\frac{\mathcal{HT}(h) = \eta \qquad init_\eta = v}{\langle I, O, H, add(h)\rangle \to \langle I, O, H[h \mapsto v], skip\rangle}$$

E-IfValidTrue
$$\frac{h \in dom(H)}{\langle I, O, H, valid(h) \ c_1 \ else \ c_2\rangle \to \langle I, O, H, c_1\rangle}$$

E-AddValid
$$\frac{h \in dom(H)}{\langle I, O, H, add(h)\rangle \to \langle I, O, H, skip\rangle}$$

E-IfValidFalse
$$\frac{h \notin dom(H)}{\langle I, O, H, valid(h) \ c_1 \ else \ c_2\rangle \to \langle I, O, H, c_2\rangle}$$

E-Rem
$$\frac{}{\langle I, O, H, remove(h)\rangle \to \langle I, O, H \setminus h, skip\rangle}$$

Figure 15 Selected rules of the operational semantics of SafeP4; the elided rules are standard and can be found in Appendix A.

E-Const
$$\frac{[\![k]\!](v_1, ..., v_n) = v}{\langle H, k(v_1, ..., v_n)\rangle \to v}$$

E-Field
$$\frac{H(h) = \{f_1 : n_1, ..., f_k : n_k\}}{\langle H, h.f_i\rangle \to n_i}$$

Figure 16 Selected rules of the operational semantics for expressions.

appendix for brevity. The rules for validity checks (E-IfValidTrue, E-IfValidFalse) step to the true branch if $h \in dom(H)$ and to the false branch otherwise.

Table application commands are evaluated according to rule E-TApply. We first invoke the control plane function $\mathcal{CA}(H, t)$, which determines an action a_i and action data v. Then we use $\mathcal{A}$ to lookup the definition of a_i, yielding $\lambda\bar{x} : \bar{\tau}.\ c_i$ and step to $c_i[\bar{v}/\bar{x}]$. Note that for simplicity, we model the evaluation of expressions read by the table in the the control-plane function $\mathcal{CA}$ rather than in the calculus.

The rule E-Add evaluates addition commands $add(h)$. Similar to header extraction, the $init_\eta()$ function produces a header instance v of type η with all fields set to a default value and extends the map H with $h \mapsto v$. Note that according to E-Add-Exist, if the header instance is already valid, $add(h)$ does nothing. Finally, the rule E-Rem removes the header from the map H. Again, if a header h is already invalid, removing it has no effect.

The semantics for expressions is given in Figure 16. We assume that there is an evaluation function for constants $[\![k]\!](\bar{v}) = v$ that is well-behaved—i.e., if $\texttt{typeof}(k) = \bar{\tau} \to \tau'$ and $\overline{v : \tau}$, then $.;. \vdash [\![k]\!](\bar{v}) : \tau'$. We use these facts to prove progress and preservation.

We define the semantics of expressions using tuples $\langle H, e\rangle$, where H is the same map used in the semantics of commands and e is the expression to evaluate. The rule E-Const evaluates constants (we omit the obvious congruence rule) and rule E-Field reduces header field expressions to the value stored in H for the respective field.

$$\textsc{Ent-Empty}\ \frac{}{\cdot \models 1} \qquad \textsc{Ent-Inst}\ \frac{dom(H) = \{h\}}{H \models h} \qquad \textsc{Ent-Seq}\ \frac{H_1 \models \Theta_1 \quad H_2 \models \Theta_2}{H_1 \cup H_2 \models \Theta_1 \cdot \Theta_2} \qquad \textsc{Ent-ChoiceL}\ \frac{H \models \Theta_1}{H \models \Theta_1 + \Theta_2} \qquad \textsc{Ent-ChoiceR}\ \frac{H \models \Theta_2}{H \models \Theta_1 + \Theta_2}$$

Figure 17 The *Entailment* relation between header instances and header instance types

3.5 Safety of SafeP4

We prove safety in terms of progress and preservation. Both theorems make use of the relation $H \models \Theta$ which intuitively holds if H is described by Θ. The formal definition, as given in Figure 17, satisfies $H \models \Theta$ if and only if $dom(H) \in [\![\Theta]\!]$.

We prove type safety via progress and preservation theorems. The respective proofs are mostly straightforward for our system–we highlight the unusual and nontrivial cases below an relegate the full proofs to the appendix.

▶ **Theorem 1** (Progress). *If* $\cdot \vdash c : \Theta \Rightarrow \Theta'$ *and* $H \models \Theta$, *then either,*

- $c = skip$, *or*
- $\exists \langle I', O', H', c' \rangle.\ \langle I, O, H, c \rangle \rightarrow \langle I', O', H', c' \rangle$.

Intuitively, progress says that a well-typed command is fully reduced or can take a step.

▶ **Theorem 2** (Preservation). *If* $\Gamma \vdash c : \Theta_1 \Rightarrow \Theta_2$ *and* $\langle I, O, H, c \rangle \rightarrow \langle I', O', H', c' \rangle$, *where* $H \models \Theta_1$, *then* $\exists \Theta_1', \Theta_2'.\ \Gamma \vdash c : \Theta_1' \Rightarrow \Theta_2'$ *where* $H' \models \Theta_1'$ *and* $\Theta_2' < \Theta_2$.

More interestingly, preservation says that if a command c is well-typed with input type Θ_1 and output type Θ_2, and c evaluates to c' in a single step, then there exists an input type Θ_1' and an output type Θ_2' that make c' well-typed. To make the inductive proof go through, we also need to prove that Θ_1' describes the same maps of header instance H as Θ_1, and Θ_2' is semantically contained in Θ_2. (These conditions are somewhat reminiscent of conditions found in languages with subtyping.)

Proof. By induction on a derivation of $\Gamma \vdash c : \Theta_1 \Rightarrow \Theta_2$, with a case analysis on the last rule used. We focus on two of the most interesting cases. See Appendix C for the full proof.

Case T-IfValid: $c = valid(h)\ c_1\ else\ c_2$ and $\Gamma \vdash c_1 : \texttt{Restrict}\ \Theta_1\ h \Rightarrow \Theta_{12}$ and $\Gamma \vdash c_2 : \texttt{NegRestrict}\ \Theta_1\ h \Rightarrow \Theta_{22}$ and $\Theta_2 = \Theta_{12} + \Theta_{22}$.

There are two evaluation rules that apply to c, E-IfValidTrue and E-IfValidFalse

Subcase E-IfValidTrue: $c' = c_1$ and $h \in dom(H)$ and $H' = H$.

Let $\Theta_1' = \texttt{Restrict}\ \Theta_1\ h$ and $\Theta_2' = \Theta_{12}$. We have $\Gamma \vdash c' : \Theta_1' \Rightarrow \Theta_2'$ by assumption, we have $H \models \Theta_1'$ by Lemma 17, and we have $\Theta_2' < \Theta_2$ by the definition of $<$ and the semantics of union.

Subcase E-IfValidFalse: $c' = c_2$ and $h \notin dom(H)$ and $H' = H$.

Symmetric to the previous case.

Case T-Apply: $c = t.apply()$ and $\mathcal{CV}(t) = (\bar{S}, \bar{e})$ and $t.actions = \bar{a}$ and $\cdot; \Theta \vdash e_i : \tau_i$ for $e_i \in \bar{e}$ and $\texttt{Restrict}\ \Theta_1\ S_i \vdash a_i : \bar{\tau}_i \rightarrow \Theta_i'$ for $a_i \in a$ and $\Theta_2 = \sum(\Theta_i')$

Only one evaluation rule that applies to c, E-Apply. It follows that $\mathcal{CA}(H, t) = (a_i, \bar{v})$, and $c' = c_i[\bar{v}/\bar{x}]$ where $\mathcal{A}(a_i) = \lambda \bar{x}.\ c_i$. By inverting `T-Action`, we have $\Gamma, \bar{x} : \bar{\tau}_i; \vdash c_i :$

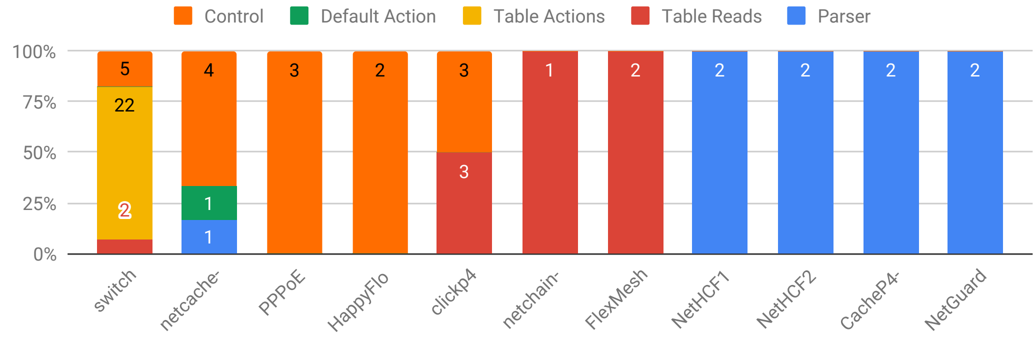


Figure 18 Proportional frequencies of each bug type per-program. The raw number of bugs for each program and category is reported at the top of each stacked bar.

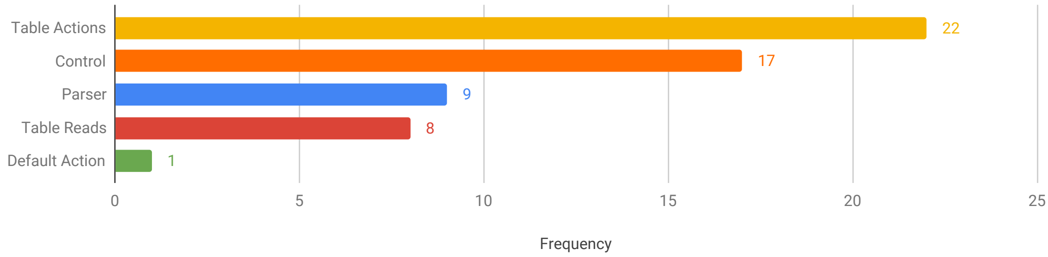


Figure 19 Frequency of each bug across all programs. The raw number of bugs in each category is reported to the right of the bar

$\texttt{Restrict}\ \Theta\ S_i \Mapsto \Theta'_i$. By Proposition 14, we have $\cdot; \cdot \vdash \bar{v} : \bar{\tau}_i$. By the substitution lemma, we have $\Gamma \vdash c_i[\bar{w}/\bar{x}] : \texttt{Restrict}\ \Theta\ S_i \Mapsto \Theta'_i$. Let $\Theta'_1 = \texttt{Restrict}\ \Theta\ S_i$ and $\Theta'_2 = \Theta'_i$. We have shown that $\Gamma \vdash c' : \Theta'_1 \Mapsto \Theta'_2$, we have that $H' \models \Theta'_1$ by Proposition 15, and we have $\Theta'_2 < \Theta_2$ by the definition of $<$ and the semantics of union types. ◀

4 Experience (Evaluation)

We implemented our type system in a tool called P4Check that automatically checks P4 programs and reports violations of the type system presented in Figure 12. P4Check uses the front-end of `p4v` [18] and handles the full $\mathrm{P4}_{14}$ language. Our key findings, which are reported in detail below, show (i) that our type system finds bugs "in the wild" and (ii) that the programmer effort needed to repair programs to pass our type checker is modest.

4.1 Overview of Bugs in the Wild

We ran P4Check on 15 open source $\mathrm{P4}_{14}$ programs[3] designed for research and industrial use. The subject programs are of varying sizes and complexities—ranging from 143 to 9060 lines of code. Our criteria for selecting programs was: (1) each program had to be open

[3] We chose to check $\mathrm{P4}_{14}$ instead of $\mathrm{P4}_{16}$, since there were more realistic open-source programs available on GitHub in $\mathrm{P4}_{14}$. In fact, we could find no $\mathrm{P4}_{16}$ program that comes close to the size and complexity if `switch.p4`, which is written in $\mathrm{P4}_{14}$.

source, (2) available on GitHub, and (3) compile without errors, (4) and be written either by industrial teams developing production code or by researchers implementing standard or novel network functionality in P4 (i.e., we excluded programs primarily used for teaching). Out of the 15 subject programs only 4 passed our type checker, all of which were simple implementations of routers or DDoS mitigation that accepted only a small number of packet types and were relatively small (188 - 635 lines of code). For the remaining 11 programs (industrial and research) our checker found 418 type checking violations overall.

Frequently, multiple violations produced by P4Check have the same bug as their root cause. For example, if a single action `rewrite_ipv4` that rewrites fields `srcAddr` and `dstAddr` for an `ipv4` header is called in a context that cannot prove that `ipv4` is valid, then both references to `ipv4.srcAddr` and `ipv4.dstAddr` will be reported as violations, even though they are due to the same *control* bug (Section 2.2.2)—namely that `rewrite_ipv4` was not called in a context that could prove the validity of `ipv4`. To address this issue, we applied another metric to quantify the number of bugs (inspired by the method proposed by others [15]): we equate the number of bugs in each program with the number of bug *fixes* required to make the program in question pass our type checker. Using this metric, we counted 58 bugs.

We classified the bugs according to the classes described in Section 2.2. Figure 18 depicts the per-program breakdown of the frequency of each bug class, and Figure 19 depicts the overall frequency of each bug. Notice that even though table action bugs were the most frequent bug (with 22 occurrences), they were only found in a single program (`switch.p4`). These bugs are especially prevalent in this program because of its heavy reliance on correct control-plane configuration. Conversely, there were 9 occurrences across 5 programs for both parser bugs and table reads bugs.

Readers familiar with previous work on `p4v` [18], a recent P4 verification tool may notice that we detected no default action bugs for the `switch.p4` program, while `p4v` reported many! The reasons for this are twofold. First, `p4v` allows programmers to verify arbitrarily complex propositions in Dijkstra's guarded command language, which means that it can express fine-grained conditions on tables and relationships between them. In contrast, we make heuristic assumptions about P4 programs that automatically eliminate many bugs, including some default action bugs. Second, our repairs are often coarse-grained and may enforce a stronger guarantee on the program than may be necessary; using first-order logic annotations, `p4v` programmers manually specify the most liberal (and hence complex) assumptions.

We make no claims about the completeness of our taxonomy. For example, we found one instance, in the HappyFlowFriends program, where the programmer had mistakenly instantiated metadata m as a header, and consequently did not parse m (since metadata is always valid) causing m to (ironically) always be invalid.

4.2 P4Check in Action

We reprise the canonical examples of each class of bugs from Section 2.2, describing how P4Check detects them and discussing ways to fix them.

4.2.1 Parser Bugfixes

Recall Figure 4, which exhibits the parser bug. The bug occurs because the parser, which extracts IPv4-TCP packets, boots unexpected packets (such as IPv6 or UDP packets) directly to `ingress`, which then assumes that both the `ipv4` and `tcp` headers are valid, even though the parser does not guarantee this fact.

```
./h.p4, line 350, cols 12-21: error tcp not guaranteed to be valid
./h.p4, line 118, cols 8-16: error ipv4 not guaranteed to be valid
./h.p4, line 101, cols 42-50: error ipv4 not guaranteed to be valid
./h.p4, line 320, cols 8-15: error tcp not guaranteed to be valid
./h.p4, line 362, cols 12-19:error tcp not guaranteed to be valid
./h.p4, line 362, cols 29-36: error tcp not guaranteed to be valid
./h.p4, line 295, cols 60-69: error tcp not guaranteed to be valid
./h.p4, line 107, cols 8-16: error ipv4 not guaranteed to be valid
./h.p4, line 101, cols 42-50: error ipv4 not guaranteed to be valid
./h.p4, line 163, cols 8-16: error ipv4 not guaranteed to be valid
./h.p4, line 101, cols 42-50: error ipv4 not guaranteed to be valid
```

```
./h.p4, line 350, cols 12-21: error tcp not guaranteed to be valid
./h.p4, line 320, cols 8-15: error tcp not guaranteed to be valid
./h.p4, line 362, cols 12-19: error tcp not guaranteed to be valid
./h.p4, line 362, cols 29-36: error tcp not guaranteed to be valid
./h.p4, line 295, cols 60-69: error tcp not guaranteed to be valid
```

Figure 20 Curated output from P4Check for the parser bug in NetHCF before (above) and after (below) modifying `parse_ethernet`

In terms of our type system, the parser produces packets of type $\texttt{ethernet} \cdot (1 + \texttt{ipv4} \cdot (1 + \texttt{tcp}))$; however the control only handles packets of type $\texttt{ethernet} \cdot \texttt{ipv4} \cdot \texttt{tcp}$. Hence, when typecheck this example, P4Check reports every reference to `tcp` and `ipv4` in the whole program as a violation of the type system. As shown in the top half Figure 20, we get an error message at every reference to `ipv4` or `tcp`. The ubiquity of the reports intimates a mismatch between the parsing and the control types, which gives the programer a hint as how to fix the problem.

When we modify the `default` clause in `parse_ethernet`, as in Figure 4, and run our tool again, all of the `ipv4` violations are removed, as shown in the bottom half of Figure 20. Then fixing the `parse_ipv4` parser, as in Figure 4, causes our tool to output no violations; we count each of these fixes to be a separate bug. Now, the type on entering the `ingress` control function is $\texttt{ethernet} \cdot \texttt{ipv4} \cdot \texttt{tcp}$, so all subsequent references to `ipv4` and `tcp` will be safe.

4.2.2 Control Bugfixes

The control bug occurs when the incoming type presents a choice between two headers that is not handled by subsequent code. The exemplar presented in Figure 5 presents a parser that produces the following type Θ:

$$\Theta = \texttt{ethernet} \cdot (1 + \texttt{ipv4} \cdot (1 + \texttt{udp} \cdot (1 + \texttt{nc\_hdr} \cdot \tau) + \texttt{tcp})),$$

where τ is a widely branching type representing caching operations. Notice $\texttt{Includes}\ \Theta\ \texttt{nc\_hdr}$ is false. However, the control nodes `process_cache` and `process_value` only type check in contexts where $\texttt{Includes}\ \Theta\ \texttt{nc\_hdr}$ is true. P4Check reports type violations at every reference to `nc_hdr`. Fixing this error is simply a matter of wrapping the `process_cache()` call in a validity check as demonstrated in Figure 5. Since NetCache handles TCP and UDP packets as well as its special-purpose packets, we simply continue on and apply the IPv4 routing table if the validity check for `nc_hdr` fails.[4]

[4] Astute readers may detect a parser bug in this example. Hint, the `ipv4_route` table requires $\texttt{Includes}\ \Theta\ \texttt{ipv4}$ where Θ is type where it is applied.

```
port.p4, line 248, cols 8-24: warning: assuming either vlan_tag_[0]
    matched as valid or vlan_tag_[0].vid wildcarded

port.p4, line 250, cols 8-24: warning: assuming either vlan_tag_[1]
    matched as valid or vlan_tag_[1].vid wildcarded
```

```
fabric.p4 line 42, cols 41-67: warning: assuming fabric_header_cpu
    matched as valid for rules with action terminate_cpu_packet

fabric.p4, line 57, cols 17-54: warning: assuming fabric_header_unicast
     matched as valid for rules with action
    terminate_fabric_unicast_packet

fabric.p4, line 81, cols 17-56: warning: assuming
    fabric_header_multicast matched as valid for rules with action
    terminate_fabric_multicast_packet
```

Figure 21 Warnings printed after fixing `switch.p4`'s reads bug (top), and its actions bug (bottom)

4.2.3 Table Reads Bugfixes

Table reads errors, like the one in Figure 6, occur when a header h is included in the `reads` declaration of a table t with match kind k, and h is not guaranteed to be valid at the call site of t, and if $h \notin \texttt{valid\_reads}(t)$ or the match-kind of $k \neq \texttt{ternary}$.

In the case of the `port_vlan_mapping` table in Figure 6, there is a valid bit for both `vlan_tag_[0]` and `vlan_tag_[1]`, both of which are followed by `exact` matches. To solve this problem, we need to use the `ternary` match-kind instead, which allows the use of wildcard matching. When a field is matched with a wildcard, the table does not attempt to compute the value of the `reads` expression; instead, the table short-circuits and skips the check entirely.

This fix assumes that the controller is well behaved and fills the `vlan_tag_[0].vid` with a wildcard whenever `vlan_tag_[0]` is matched as invalid (and similarly for `vlan_tag_[1]`)s. This also what the SafeP4 type system does. P4Check prints warnings describing these assumptions to the programmer (top of Figure 21), giving them properties against which to check their control plane implementation.

4.2.4 Table Action Bugfixes

The table actions bugs, which are exemplified by the table `fabric_ingress_dst_lkp` from Figure 7, can be fixed by modifying the `reads` declaration in the table. Recall that the parser will parse exactly one of the headers `fabric_hdr_cpu`, `fabric_hdr_unicast` and `fabric_hdr_multicast`, which means that when the table is applied at type Θ, exactly one of `Includes` Θ `fabric_hdr_i` for $i \in \{\texttt{cpu}, \texttt{unicast}, \texttt{multicast}\}$ will hold. Now, the action `term_cpu_packet` typechecks only with the (nonempty) type `Restrict` Θ `fabric_hdr_cpu`, and the actions `term_fabric_`i`_packet` only typecheck with the (nonempty) types `Restrict` Θ `term_fabric_`i`_packet` for $i = \texttt{unicast}, \texttt{multicast}$. P4Check suggests that this is the cause of the bug since it reports type violations for all of the references to these three headers in the control paths following from the application of `fabric_ingress_dst_lkp`.

The optimal[5] fix here is to augment the `reads` declaration to include a validity check for

---

[5] The other fix would be to refactor the single into multiple tables, each of which is guarded by separate validity checks. However, combining this kind of logic in a single table helps to conserve on-switch memory, so in striving to change the behavior of the program as little as possible, we propose modifying the table reads.

each contentious header. We then assume that the controller is well-behaved enough to only call actions when their required headers are valid, allowing us to typecheck each action in the appropriate type restriction. P4Check alerts the programmer whenever it makes such an assumption. We show these warnings for the fixed version of `fabric_ingress_dst_lkp` below the line in Figure 21.

4.2.5 Default Action Bugfix

The default action bug occurs when a programmer creates a wrapper table for an action that modifies the type, and forgets to force the table to call that action when the packet misses. The `add_value_header_1` table from Figure 8 wraps the action `add_value_header_1_act`, which calls the single line `add_header(nc_value_1)`.

The default action, when left unspecified, is `nop`, which means that if the pre-application type was Θ, then the post-application type is $\Theta + \Theta \cdot \texttt{nc\_value\_1}$, which does not include `nc_value_1`. Hence, P4Check reports every subsequent reference (on this code path) to `nc_header_1` to be a type violation.

To fix this bug, we need to set the default action to `add_value_1`—this makes the post-application type $\Theta \cdot \texttt{nc\_value\_1} + \Theta \cdot \texttt{nc\_value\_1} = \Theta \cdot \texttt{nc\_value\_1}$, which includes `nc_value_1`, thus allowing the subsequent code to typecheck.

4.3 Overhead

It is important to evaluate two kinds of overhead when considering a static type system: overhead on programmers and on the underlying implementation.

Typically, adding a static type system to a dynamic type system requires more work for the programmer—the field of gradual typing is devoted breaking the gargantuan task into smaller commit-sized chunks [5]. Surprisingly, in our experience, migrating real-world P4 code to pass the SafeP4 type system required only modest programmer effort.

To qualitatively evaluate the effort required to change an unsafe program into a safe one using our type system, we manually fixed all of the bugs that we detected. The programs that had bugs required us to edit between 0.10% and 1.4% of the lines of code. The one exception was PPPoE_using_P4, which was a 143 line program that required 6 line-edits (4%), all of which were validity checks. Conversely, `switch.p4` required 34 line edits, the greatest observed number, but this only accounted for 0.37% of the total lines of code in the program.

Each class of bugs has a simple one-to-two line fix, as described in Section 4.2: adding a validity check, adding a default action, or slightly from the parser. Each of these changes was straightforward to identify and simple to make.

Another possible concern is that that extending tables with extra read expressions, or adding run-time validity checks to controls, might impose a heavy cost on implementations, especially on hardware. Although we have not yet performed an extensive study of the impact on compiled code, based on the size and complexity of the annotations we added, we believe the additional cost should be quite low. Overall, given the large number of potential bugs located by P4Check, we believe the assurance one gains about safety properties by using a static type system makes the costs well worth it.

5 Related Work

Probably the most closely related work to SafeP4 is `p4v` [18]. Unlike SafeP4, which is based on a static type system, `p4v` uses Dikstra's approach to program verification based on predicate transformer semantics. To model the behavior of the control plane, `p4v` uses first-order annotations. SafeP4's typing rule for table application is inspired by this idea, but adopts simple heuristics rather than requiring logical annotations.

Both tools be used to verify safety properties of data planes modelled in P4—e.g., that no read or write operations are possible on an invalid header. However, unlike SafeP4, `p4v` does not define a formal semantics of the P4 language and hence does not formally prove the soundness of the approach. As it is often the case when comparing approaches based on types to those based on program verification, `p4v` can check more complex properties, including architectural invariants and program-specific properties—e.g., that the IPv4 time-to-live field is correctly decremented on every packet. However, in general, it requires annotating the program with formal specifications both for the correctness property itself and to model the behavior of the control plane.

McKeown et al. developed an operational semantics for P4 [20], which is translated to Datalog to verify safety properties and to check program equivalence. An operational semantics for P4 was also developed in the K framework [25], yielding a symbolic model checker and deductive verification tool [14]. Vera [28] models the semantics of P4 by translation to SymNet [29], and develops a symbolic execution engine for verifying a variety of properties, including header validity.

Compared to SafeP4, these approaches do not use their formalization of P4 as a foundation for defining a type system that addresses common bugs. To the best of our knowledge, SafeP4 is the first formal calculus for a P4-like packet processing language that provides correct-by-construction guarantees of header safety properties.

Other languages have used type systems to rule our safety problems due to null references. For example, NullAway [27] analyzes all Java programs annotated with `@Nullable` annotations, making path-sensitive deductions about which references may be null. Similar to the validity checks in SafeP4, NullAway analyses conditionals for null checks of the form `var != null` using data flow analysis.

Looking further afield, PacLang [9] is a concurrent packet-processing language that uses a linear type system to allow multiple references to a given packet within a single thread. PacLang and SafeP4 share the use of a type system for verifying safety properties but they differ in the kind of properties they address and, hence, the kind of type system they employ for this purpose. In addition, the primary focus in PacLang is on efficient compilation whereas SafeP4 is concerned with ensuring the type safety of header data manipulated by the program.

Domino [26] is a domain-specific language for data plane algorithms supporting *packet transactions*—i.e., blocks of code that are guaranteed to be atomic and isolated from other transactions. In Domino, the programmer defines the operations needed for each packet without worrying about other in-flight packets. If it succeeds, the compiler guarantees performance at the line rate supported on programmable switches. Overall, Domino focuses on transactional guarantees and concurrency rather than header safety properties.

BPF+ [3] and eEBPF [8] are packet-processing frameworks that can be used to extend the kernel networking stack with custom functionality. The modern eBPF framework is based on machine-level programming model, but it uses a virtual machine and code verifier to ensure a variety of basic safety properties. Much of the recent work on eBPF focuses on techniques such as just-in-time compilation to achieve good performance.

SNAP [1] is a language for stateful packet processing based on P4. It offers a programming model with global state registers that are distributed across many physical switches while optimizing for various criteria, such as minimizing congestion. More specifically, the compiler analyses read/write dependencies to automatically optimize the placement of state and the routing of traffic across the underlying physical topology.

Of course, there is a long tradition of formal calculi that aim to capture some aspect of computation and make it amenable for mathematical reasoning. The design of SafeP4 is directly inspired by Featherweight Java [10], which stands out for its elegant formalization of a real-world languages in an extensible core calculus.

6 Conclusion

P4 provides a powerful programming model for network devices based on high-level and declarative abstractions. Unfortunately, P4 lacks basic safety guarantees, which often lead to a variety of bugs in practice. This paper proposes SafeP4, a domain-specific language for programmable data planes that comes equipped with a formal semantics and a static type system which ensures that every read or write to a header at run-time will be safe. Under the covers, SafeP4 uses a rich set of types that tracks dependencies beween headers, as well as a path-sensitive analysis and domain-specific heuristics that model common idioms for programming control planes and minimize false positives. Our experiments using an OCaml prototype and a suite of open-source programs found on GitHub show that most P4 applications can be made safe with minimal programming effort. We hope that our work can help lay the foundation for future enhancements to P4 as well as the next generation of data plane languages. In the future, we plan to explore enriching SafeP4's type system to track additional properties, investigate correct-by-construction techniques for writing control-plane code, and develop a compiler for the language.

References

1. Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 29–43, New York, NY, USA, 2016. ACM. URL: `http://doi.acm.org/10.1145/2934872.2934892`, `doi:10.1145/2934872.2934892`.
2. Jiasong Bai, Jun Bi, Menghao Zhang, and Guanyu Li. Filtering spoofed ip traffic using switching asics. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 51–53. ACM, 2018.
3. Andrew Begel, Steven McCanne, and Susan L. Graham. Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 123–134, New York, NY, USA, 1999. ACM. URL: `http://doi.acm.org/10.1145/316188.316214`, `doi:10.1145/316188.316214`.
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. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. URL: `http://doi.acm.org/10.1145/2656877.2656890`, `doi:10.1145/2656877.2656890`.
5. John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. Migrating gradual types. *Proceedings of the ACM on Programming Languages*, 2(POPL):15, 2017.

6 Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 1–12. ACM, 2007.

7 P4 Language Consortium. P4 language specification, version 1.0.4. Technical report, Available at https://p4.org/specs/, 2017.

8 Jonathan Corbet. Bpf: the universal in-kernel virtual machine, May 2014. Available at `https://lwn.net/Articles/599755/`,.

9 Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In David Schmidt, editor, *Programming Languages and Systems*, pages 204–218, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

10 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. URL: `http://doi.acm.org/10.1145/503502.503505`, `doi:10.1145/503502.503505`.

11 Xin Jin. netcache-p4, Mar 2018. URL: `https://github.com/netx-repo/netcache-p4`.

12 Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-rtt coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2018. Best paper award.

13 Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2017.

14 Ali Kheradmand and Grigore Roşu. P4k: A formal semantics of p4 and applications. Technical Report https://arxiv.org/abs/1804.01468, University of Illinois at Urbana-Champaign, April 2018.

15 George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2018.

16 Chaitanya Kodeboyina. An open-source p4 switch with sai support, Jun 2015. URL: `https://p4.org/p4/an-open-source-p4-switch-with-sai-support.html`.

17 Rahul Kumar and BB Gupta. Stepping stone detection techniques: Classification and state-of-the-art. In *Proceedings of the international conference on recent cognizance in wireless communication & image processing*, pages 523–533. Springer, 2016.

18 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 *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 490–503, New York, NY, USA, 2018. ACM. URL: `http://doi.acm.org/10.1145/3230543.3230582`, `doi:10.1145/3230543.3230582`.

19 Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. URL: `http://doi.acm.org/10.1145/1355734.1355746`, `doi:10.1145/1355734.1355746`.

20 Nick McKeown, Dan Talayco, George Varghese, Nuno Lopes, Nikolaj Bjorner, and Andrey Rybalchenko. Automatically verifying reachability and well-formedness in p4 networks. Technical report, September 2016. URL: `https://www.microsoft.com/en-us/research/publication/automatically-verifying-reachability-well-formedness-p4-networks/`.

21 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, dec 1978.

22 Barefoot Networks. Behavioral model, Dec 2018. URL: `https://github.com/p4lang/behavioral-model`.

23 Barefoot Networks. Switch, Jan 2018. URL: `https://github.com/p4lang/switch`.

24 Tj OConnor, William Enck, W Michael Petullo, and Akash Verma. Pivotwall: Sdn-based information flow control. In *Proceedings of the Symposium on SDN Research*, page 3. ACM, 2018.

25 Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. `doi:10.1016/j.jlap.2010.03.012`.

26 Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 15–28, New York, NY, USA, 2016. ACM. URL: `http://doi.acm.org/10.1145/2934872.2934900`, `doi:10.1145/2934872.2934900`.

27 Manu Sridharan. Engineering nullaway, uber's open source tool for detecting nullpointerexceptions on android, Dec 2018. URL: `https://eng.uber.com/nullaway/`.

28 Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging p4 programs with Vera. In *ACM SIGCOMM*, pages 518–532, New York, NY, USA, 2018. ACM. URL: `http://doi.acm.org/10.1145/3230543.3230548`, `doi:10.1145/3230543.3230548`.

29 Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM*, pages 314–327, New York, NY, USA, 2016. ACM. URL: `http://doi.acm.org/10.1145/2934872.2934881`, `doi:10.1145/2934872.2934881`.

30 Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM. URL: `http://doi.acm.org/10.1145/1863543.1863561`, `doi:10.1145/1863543.1863561`.

31 Menghao Zhang. Anti-spoof, Nov 2018. URL: `https://github.com/zhangmenghao/Anti-spoof`.

A Additional Operational Semantics rules

This section presents additional evaluation rules.

E-Seq

$$\frac{}{\langle I, O, H, \mathit{skip}; c_2\rangle \rightarrow \langle I, O, H, c_2\rangle}$$

E-Seq1

$$\frac{\langle I, O, H, c_1\rangle \rightarrow \langle I', O', H', c_1'\rangle}{\langle I, O, H, c_1; c_2\rangle \rightarrow \langle I', O', H', c_1'; c_2\rangle}$$

E-If

$$\frac{\langle H, e\rangle \rightarrow e'}{\langle I, O, H, \mathit{if}\ (e)\ \mathit{then}\ c_1\ \mathit{else}\ c_2\rangle \rightarrow \langle I, O, H, \mathit{if}\ (e')\ \mathit{then}\ c_1\ \mathit{else}\ c_2\rangle}$$

E-IfTrue

$$\frac{}{\langle I, O, H, \mathit{if}\ (\mathit{true})\ \mathit{then}\ c_1\ \mathit{else}\ c_2\rangle \rightarrow \langle I, O, H, c_1\rangle}$$

E-IfFalse

$$\frac{}{\langle I, O, H, \mathit{if}\ (\mathit{false})\ \mathit{then}\ c_1\ \mathit{else}\ c_2\rangle \rightarrow \langle I, O, H, c_2\rangle}$$

B Operations on header types

This section presents an in-depth treatment of the operations defined on header instance types. In the following we assume that S ranges over elements of the domain $\mathcal{P}(\mathcal{P}(H))$.

Restriction The restrict operator $\texttt{Restrict}\ \Theta\ h$ recusively traverses Θ and keeps only those choices in which h is contained, zeroing out the others. Semantically this has the effect of throwing out the subsets of $[\![\Theta]\!]$ that do not contain h, i.e., we define restriction semantically as $S|h \triangleq \{hs | hs \in S \wedge h \in hs\}$. Syntactically we define restriction by induction on Θ as shown in Figure 22. The equivalence of the syntactic and the semantic definition is captured

$$
\begin{aligned}
\texttt{Restrict}\ 0\ h &\triangleq 0 \\
\texttt{Restrict}\ 1\ h &\triangleq 0 \\
\texttt{Restrict}\ g\ h &\triangleq \begin{cases} g & \text{if } g = h \\ 0 & \text{otherwise} \end{cases} \\
\texttt{Restrict}\ (\Theta_1 \cdot \Theta_2)\ h &\triangleq ((\texttt{Restrict}\ \Theta_1\ h) \cdot \Theta_2) + (\Theta_1 \cdot (\texttt{Restrict}\ \Theta_2\ h)) \\
\texttt{Restrict}\ (\Theta_1 + \Theta_2)\ h &\triangleq (\texttt{Restrict}\ \Theta_1\ h) + (\texttt{Restrict}\ \Theta_2\ h)
\end{aligned}
$$

Figure 22 Syntactic definition of the $\texttt{Restrict}\ \Theta\ h$ operator.

by Lemma 3.

▶ **Lemma 3.** $[\![\Theta]\!]|h == [\![\texttt{Restrict}\ \Theta\ h]\!]$.

Proof. By induction on Θ.
Case $\Theta = 0$:

$$
\begin{aligned}
&[\![0]\!]|h \\
&= \{\}|h && \text{by definition of } [\![.]\!] \\
&= \{hs | hs \in \{\} \wedge h \in hs\} && \text{by definition of } .|h \\
&= \{\} && \text{by set theory} \\
&= [\![0]\!] && \text{by definition of } [\![.]\!] \\
&= [\![\texttt{Restrict}\ 0\ h]\!] && \text{by definition of } \texttt{Restrict}\ .\ h
\end{aligned}
$$

Case $\Theta = 1$:

$$
\begin{aligned}
&[\![1]\!]|h \\
&= \{\{\}\}|h && \text{by definition of } [\![.]\!] \\
&= \{hs | hs \in \{\{\}\} \wedge h \in hs\} && \text{by definition of } .|h \\
&= \{\} && \text{by set theory} \\
&= [\![0]\!] && \text{by definition of } [\![.]\!] \\
&= [\![\texttt{Restrict}\ 1\ h]\!] && \text{by definition of } \texttt{Restrict}\ .\ h
\end{aligned}
$$

Case $\Theta = g$:

$$
\begin{aligned}
& [\![g]\!]|h \\
&= \{\{g\}\}|h && \text{by definition of } [\![.]\!] \\
&= \{hs | hs \in \{\{g\}\} \wedge h \in hs\} && \text{by definition of } .|h \\
& \textit{Subcase } h = g \\
&= \{\{g\}\} && \text{by set theory} \\
&= [\![g]\!] && \text{by definition of } [\![.]\!] \\
&= [\![\texttt{Restrict}\ g\ h]\!] && \text{by assumption } h = g \text{ and by definition of } \texttt{Restrict}\ .\ h \\
& \textit{Subcase } h \neq g \\
&= \{\} && \text{by set theory} \\
&= [\![0]\!] && \text{by definition of } [\![.]\!] \\
&= [\![\texttt{Restrict}\ g\ h]\!] && \text{by definition of } \texttt{Restrict}\ .\ h \text{ and by assumption } h \neq g
\end{aligned}
$$

Case $\Theta = \Theta_1 \cdot \Theta_2$:

$$
\begin{aligned}
& [\![\Theta_1 \cdot \Theta_2]\!]|h \\
&= \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!]\}|h && \text{by definition of } [\![.]\!] \\
&= \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!] \wedge h \in (hs_1 \cup hs_2)\} && \text{by definition of } .|h \\
&= \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!] \wedge h \in hs_1\} \cup && \text{by set theory} \\
&\quad \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!] \wedge h \in hs_2\} \\
&= \{hs_1 \cup hs_2 | (hs_1 \in [\![\Theta_1]\!] \wedge h \in hs_1) \wedge hs_2 \in [\![\Theta_2]\!]\} \cup && \text{by logic and set theory} \\
&\quad \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge (hs_2 \in [\![\Theta_2]\!] \wedge h \in hs_2)\} \\
&= \{hs_1 | hs_1 \in [\![\Theta_1]\!] \wedge h \in hs_1\} \bullet \{hs_2 | hs_2 \in [\![\Theta_2]\!]\} \cup && \text{by definition of } S_1 \bullet S_2 \\
&\quad \{hs_1 | hs_1 \in [\![\Theta_1]\!]\} \bullet \{hs_2 | hs_2 \in [\![\Theta_2]\!] \wedge h \in hs_2\} \\
&= \{hs_1 | hs_1 \in [\![\Theta_1]\!]\}|h \bullet \{hs_2 | hs_2 \in [\![\Theta_2]\!]\} \cup && \text{by definition of } .|h \\
&\quad \{hs_1 | hs_1 \in [\![\Theta_1]\!]\} \bullet \{hs_2 | hs_2 \in [\![\Theta_2]\!]\}|h \\
&= [\![\Theta_1]\!]|h \bullet [\![\Theta_2]\!] \cup [\![\Theta_1]\!] \bullet [\![\Theta_2]\!]|h && \text{by definition of } [\![.]\!] \\
&= [\![\texttt{Restrict}\ \Theta_1\ h]\!] \bullet [\![\Theta_2]\!] \cup [\![\Theta_1]\!] \bullet [\![\texttt{Restrict}\ \Theta_2\ h]\!] && \text{by induction hypothesis} \\
&= [\![\texttt{Restrict}\ \Theta_1\ h \cdot \Theta_2 + \Theta_1 \cdot \texttt{Restrict}\ \Theta_2\ h]\!] && \text{by definition of } S_1 \bullet S_2 \text{ and } [\![.]\!] \\
&= [\![\texttt{Restrict}\ (\Theta_1 \cdot \Theta_2)\ h]\!] && \text{by definition of } \texttt{Restrict}\ .\ h
\end{aligned}
$$

Case $\Theta = \Theta_1 + \Theta_2$:

$$
\begin{aligned}
& [\![\Theta_1 + \Theta_2]\!]|h \\
&= ([\![\Theta_1]\!] \cup [\![\Theta_2]\!])|h && \text{by definition of } [\![.]\!] \\
&= \{hs | hs \in ([\![\Theta_1]\!] \cup [\![\Theta_2]\!]) \wedge h \in hs\} && \text{by definition of } .|h \\
&= \{hs_1 | hs_1 \in [\![\Theta_1]\!] \wedge h \in hs_1\} \cup \{hs_2 | hs_2 \in [\![\Theta_2]\!] \wedge h \in hs_2\} && \text{by set theory} \\
&= [\![\Theta_1]\!]|h \cup [\![\Theta_2]\!]|h && \text{by definition of } .|h \\
&= [\![\texttt{Restrict}\ \Theta_1\ h]\!] \cup [\![\texttt{Restrict}\ \Theta_2\ h]\!] && \text{by induction hypothesis} \\
&= [\![\texttt{Restrict}\ \Theta_1\ h + \texttt{Restrict}\ \Theta_2\ h]\!] && \text{by definition of } [\![.]\!] \\
&= [\![\texttt{Restrict}\ (\Theta_1 + \Theta_2)\ h]\!] && \text{by definition of } \texttt{Restrict}\ .\ h
\end{aligned}
$$

◀

Negated Restriction Dually to the restrict operator, $\texttt{NegRestrict}\ \Theta\ h$ produces only those choices/subsets where h is invalid. Semantically, negated restriction is defined as $S|\neg h \triangleq \{hs | hs \in S \wedge h \notin hs\}$. Syntactically we define *Negated Restriction* by induction on Θ as shown in Figure 23.

$$
\begin{aligned}
\texttt{NegRestrict}\ 0\ h &\triangleq 0 \\
\texttt{NegRestrict}\ 1\ h &\triangleq 1 \\
\texttt{NegRestrict}\ g\ h &\triangleq \begin{cases} 0 & \text{if } g = h \\ g & \text{otherwise} \end{cases} \\
\texttt{NegRestrict}\ (\Theta_1 \cdot \Theta_2)\ h &\triangleq (\texttt{NegRestrict}\ \Theta_1\ h) \cdot (\texttt{NegRestrict}\ \Theta_2\ h) \\
\texttt{NegRestrict}\ (\Theta_1 + \Theta_2)\ h &\triangleq (\texttt{NegRestrict}\ \Theta_1\ h) + (\texttt{NegRestrict}\ \Theta_2\ h)
\end{aligned}
$$

Figure 23 Syntactic definition of the $\texttt{NegRestrict}\ \Theta\ h$ operator

The equivalence of the syntactic and semantic definition is captured by Lemma 4.

▶ **Lemma 4.** $[\![\Theta]\!]|\neg h == [\![\texttt{NegRestrict}\ \Theta\ h]\!]$.

Proof. By induction on Θ.

Case $\Theta = 0$:

$$
\begin{aligned}
& [\![0]\!]|\neg h \\
&= \{\}|\neg h && \text{by definition of } [\![.]\!] \\
&= \{hs | hs \in \{\} \wedge h \notin hs\} && \text{by definition of } .|\neg h \\
&= \{\} && \text{by set theory} \\
&= [\![0]\!] && \text{by definition of } [\![.]\!] \\
&= [\![\texttt{NegRestrict}\ 0\ h]\!] && \text{by definition of } \texttt{NegRestrict}\ .\ h
\end{aligned}
$$

Case $\Theta = 1$:

$$
\begin{aligned}
& [\![1]\!]|\neg h \\
&= \{\{\}\}|\neg h && \text{by definition of } [\![.]\!] \\
&= \{hs | hs \in \{\{\}\} \wedge h \notin hs\} && \text{by definition of } .|\neg h \\
&= \{\{\}\} && \text{by set theory} \\
&= [\![1]\!] && \text{by definition of } [\![.]\!] \\
&= [\![\texttt{NegRestrict}\ 1\ h]\!] && \text{by definition of } \texttt{NegRestrict}\ .\ h
\end{aligned}
$$

Case $\Theta = g$:

$$
\begin{aligned}
& [\![g]\!]|\neg h \\
&= \{\{g\}\}|\neg h && \text{by definition of } [\![.]\!] \\
&= \{hs | hs \in \{\{g\}\} \wedge h \notin hs\} && \text{by definition of } .|\neg h \\
& \textit{Subcase } h = g \\
&= \{\} && \text{by set theory} \\
&= [\![0]\!] && \text{by definition of } [\![.]\!] \\
&= [\![\texttt{NegRestrict } 0 \ h]\!] && \text{by assumption } h = g \text{ and by definition of } \texttt{NegRestrict} \ . \ h \\
& \textit{Subcase } h \neq g \\
&= \{\{g\}\} && \text{by set theory} \\
&= [\![g]\!] && \text{by definition of } [\![.]\!] \\
&= [\![\texttt{NegRestrict } g \ h]\!] && \text{by definition of } \texttt{NegRestrict} \ . \ h \text{ and by assumption } h \neq g
\end{aligned}
$$

Case $\Theta = \Theta_1 \cdot \Theta_2$:

$$
\begin{aligned}
& [\![\Theta_1 \cdot \Theta_2]\!]|\neg h \\
&= ([\![\Theta_1]\!] \bullet [\![\Theta_2]\!])|\neg h && \text{by definition of } S_1 \bullet S_2 \\
&= \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2[\!]\}|\neg h && \text{by definition of } [\![.]\!] \\
&= \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!] \wedge h \notin (hs_1 \cup hs_2)\} && \text{by definition of } .|\neg h \\
&= \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!] \wedge h \notin hs_1 \wedge h \notin hs_2)\} && \text{by set theory and logic} \\
&= \{hs_1 \cup hs_2 | (hs_1 \in [\![\Theta_1]\!] \wedge h \notin hs_1) \wedge (hs_2 \in [\![\Theta_2]\!] \wedge h \notin hs_2)\} && \text{by set theory and logic} \\
&= \{hs_1 | hs_1 \in [\![\Theta_1]\!] \wedge h \notin hs_1\} \bullet \{hs_2 | hs_2 \in [\![\Theta_2]\!] \wedge h \notin hs_2\} && \text{by definition of } S_1 \bullet S_2 \\
&= [\![\Theta_1]\!]|\neg h \bullet [\![\Theta_2]\!]|\neg h && \text{by definition of } .|\neg h \\
&= [\![\texttt{NegRestrict } \Theta_1 \ h]\!] \bullet [\![\texttt{NegRestrict } \Theta_2 \ h]\!] && \text{by induction hypothesis} \\
&= [\![(\texttt{NegRestrict } \Theta_1 \ h) \cdot (\texttt{NegRestrict } \Theta_2 \ h)]\!] && \text{By definition of } [\![.]\!] \\
&= [\![\texttt{NegRestrict } (\Theta_1 \ \cdot \Theta_2) \ h]\!] && \text{by definition of } \texttt{NegRestrict} \ . \ h
\end{aligned}
$$

Case $\Theta = \Theta_1 + \Theta_2$:

$$
\begin{aligned}
& [\![\Theta_1 + \Theta_2]\!]|\neg h \\
&= ([\![\Theta_1]\!] \cup [\![\Theta_2]\!])|\neg h && \text{by definition of } [\![.]\!] \\
&= \{hs | hs \in ([\![\Theta_1]\!] \cup [\![\Theta_2]\!]) \wedge h \notin hs\} && \text{by definition of } .|\neg h \\
&= \{hs_1 | hs_1 \in [\![\Theta_1]\!] \wedge h \notin hs_1\} \cup \{hs_2 | hs_2 \in [\![\Theta_2]\!] \wedge h \notin hs_2\} && \text{by set theory} \\
&= [\![\Theta_1]\!]|\neg h \cup [\![\Theta_2]\!]|\neg h && \text{by definition of } .|\neg h \\
&= [\![\texttt{NegRestrict } \Theta_1 \ h]\!] \cup [\![\texttt{NegRestrict } \Theta_2 \ h]\!] && \text{by induction hypothesis} \\
&= [\![\texttt{NegRestrict } \Theta_1 \ h + \texttt{NegRestrict } \Theta_2 \ h]\!] && \text{by definition of } [\![.]\!] \\
&= [\![\texttt{NegRestrict } (\Theta_1 + \Theta_2) \ h]\!] && \text{by definition of } \texttt{NegRestrict} \ . \ \neg h
\end{aligned}
$$

◀

Inclusion `Inclusion` Θ h traverses Θ and checks to make sure that h is valid in every path. Semantically this says that h is a member of every element of $[\![\Theta]\!]$, i.e., $h \sqsubset S \triangleq \bigwedge(hs \in S \wedge h \in hs)$. Syntactically we define *Inclusion* by induction on Θ as shown in Figure 24.

$$
\begin{aligned}
\texttt{Includes}\ 0\ h &\triangleq \texttt{false}\\
\texttt{Includes}\ 1\ h &\triangleq \texttt{false}\\
\texttt{Includes}\ g\ h &\triangleq \begin{cases} \mathit{true} & \text{if } g = h\\ \mathit{false} & \text{otherwise}\end{cases}\\
\texttt{Includes}\ (\Theta_1 \cdot \Theta_2)\ h &\triangleq (\texttt{Includes}\ \Theta_1\ h) \vee (\texttt{Includes}\ \Theta_2\ h)\\
\texttt{Includes}\ (\Theta_1 + \Theta_2)\ h &\triangleq (\texttt{Includes}\ \Theta_1\ h) \wedge (\texttt{Includes}\ \Theta_2\ h)
\end{aligned}
$$

Figure 24 Syntactic definition of the `Includes` Θ h operator

The equivalence of the syntactic and semantic definition is captured by Lemma 5.

▶ **Lemma 5.** $\forall hs \in S.h \in hs ==$ `Includes` Θ h.

Proof. By induction on Θ.
Case $\Theta = 0$:

$$
\begin{aligned}
& h \sqsubset [\![0]\!] \\
&= \bigwedge(hs \in \{\} \wedge h \in hs) && \text{by definition of } [\![.]\!]\\
&= \texttt{false} && \text{by logic and set theory}\\
&= \texttt{Includes}\ 0\ h && \text{by definition of } \texttt{Includes}\ .\ h
\end{aligned}
$$

Case $\Theta = 1$:

$$
\begin{aligned}
& h \sqsubset [\![1]\!] \\
&= \bigwedge(hs \in \{\{\}\} \wedge h \in hs) && \text{by definition of } [\![.]\!]\\
&= \texttt{false} && \text{by logic and set theory}\\
&= \texttt{Includes}\ 1\ h && \text{by definition of } \texttt{Includes}\ .\ h
\end{aligned}
$$

Case $\Theta = g$:

$$
\begin{aligned}
& h \sqsubset [\![g]\!] \\
&= \bigwedge(hs \in \{\{g\}\} \wedge h \in hs) && \text{by definition of } [\![.]\!]\\
& \quad \textit{Subcase } h = g \\
&= \texttt{true} && \text{by logic and set theory}\\
&= \texttt{Includes}\ g\ h && \text{by definition of } (\texttt{Includes}\ .\ h) \text{ and assumption } h = g\\
& \quad \textit{Subcase } h \neq g \\
&= \texttt{false} && \text{by logic and set theory}\\
&= \texttt{Includes}\ g\ h && \text{by definition of } (\texttt{Includes}\ .\ h) \text{ and assumption } h \neq g
\end{aligned}
$$

Case $\Theta = \Theta_1 \cdot \Theta_2$:

$$\begin{aligned}
& h \sqsubset [\![\Theta_1 \cdot \Theta_2]\!] && \\
&= h \sqsubset ([\![\Theta_1]\!] \bullet [\![\Theta_2]\!]) && \text{by definition of } S_1 \bullet S_2 \\
&= h \sqsubset \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!]\} && \text{by definition of } [\![.]\!] \\
&= h \sqsubset \{hs_1 | hs_1 \in [\![\Theta_1]\!]\} \vee h \sqsubset \{hs_2 | hs_2 \in [\![\Theta_2]\!]\} && \text{by set theory and logic} \\
&= \bigwedge(hs_1 \in [\![\Theta_1]\!] \wedge h \in hs_1) \vee \bigwedge(hs_2 \in [\![\Theta_2]\!] \wedge h \in hs_2) && \text{by definition of } h \sqsubset . \\
&= h \sqsubset [\![\Theta_1]\!] \vee h \sqsubset [\![\Theta_2]\!] && \text{by definition of } [\![.]\!] \\
&= (\texttt{Includes}\ \Theta_1\ h) \vee (\texttt{Includes}\ \Theta_2\ h) && \text{by induction hypothesis} \\
&= (\texttt{Includes}\ \Theta_1 \cdot \Theta_2\ h) && \text{by definition of } \texttt{Includes}\ .\ h
\end{aligned}$$

Case $\Theta = \Theta_1 + \Theta_2$:

$$\begin{aligned}
& h \sqsubset [\![\Theta_1 + \Theta_2]\!] && \\
&= h \sqsubset ([\![\Theta_1]\!] \cup [\![\Theta_2]\!]) && \text{by definition of } [\![.]\!] \\
&= \bigwedge(hs \in ([\![\Theta_1]\!] \cup [\![\Theta_2]\!]) \wedge h \in hs) && \text{by definition of } h \sqsubset . \\
&= \bigwedge(hs_1 \in [\![\Theta_1]\!] \wedge h \in hs_1 \wedge hs_2 \in [\![\Theta_2]\!] \wedge h \in hs_2) && \text{by set theory and logic} \\
&= \bigwedge(hs_1 \in [\![\Theta_1]\!] \wedge h \in hs_1) \wedge \bigwedge(hs_2 \in [\![\Theta_2]\!] \wedge h \in hs_2) && \text{by set theory and logic} \\
&= h \sqsubset [\![\Theta_1]\!] \wedge h \sqsubset [\![\Theta_2]\!] && \text{by definition of } h \sqsubset . \\
&= (\texttt{Includes}\ \Theta_1\ h) \wedge (\texttt{Includes}\ \Theta_2\ h) && \text{by induction hypothesis} \\
&= (\texttt{Includes}\ (\Theta_1 + \Theta_2)\ h) && \text{by definition of } \texttt{Includes}\ .\ h
\end{aligned}$$

◀

Removal `Remove` Θ h removes h from every path, which means, semantically that it removes h from ever element of $[\![\Theta]\!]$, i.e., $S \setminus h \triangleq \{hs | hs \in S \wedge hs \setminus \{h\}\}$. Syntactically we define *Removal* by induction on Θ as shown in Figure 25.

$$\begin{aligned}
\texttt{Remove}\ 0\ h &\triangleq 0 \\
\texttt{Remove}\ 1\ h &\triangleq 1 \\
\texttt{Remove}\ g\ h &\triangleq \begin{cases} 1 & \text{if } g = h \\ g & \text{otherwise} \end{cases} \\
\texttt{Remove}\ (\Theta_1 \cdot \Theta_2)\ h &\triangleq (\texttt{Remove}\ \Theta_1\ h) \cdot (\texttt{Remove}\ \Theta_2\ h) \\
\texttt{Remove}\ (\Theta_1 + \Theta_2)\ h &\triangleq (\texttt{Remove}\ \Theta_1\ h) + (\texttt{Remove}\ \Theta_2\ h)
\end{aligned}$$

Figure 25 Syntactic definition of the `Remove` Θ h operator

The equivalence of the syntactic and semantic definition is captured by Lemma 6.

▶ **Lemma 6.** $[\![\Theta]\!] \setminus h == [\![\texttt{Remove}\ \Theta\ h]\!]$.

Proof. By induction on Θ.

Case $\Theta = 0$:

$$\begin{aligned}
&\llbracket 0 \rrbracket \setminus h \\
&= \{\} \setminus h && \text{by definition of } \llbracket . \rrbracket \\
&= \{hs | hs \in \{\} \wedge hs \setminus \{h\}\} && \text{by definition of } . \setminus h \\
&= \{\} && \text{by set theory} \\
&= \llbracket 0 \rrbracket && \text{by definition of } \llbracket . \rrbracket \\
&= \llbracket \texttt{Remove } 0\ h \rrbracket && \text{by definition of } \texttt{Remove . } h
\end{aligned}$$

Case $\Theta = 1$:

$$\begin{aligned}
&\llbracket 1 \rrbracket \setminus h \\
&= \{\{\}\} \setminus h && \text{by definition of } \llbracket . \rrbracket \\
&= \{hs | hs \in \{\{\}\} \wedge hs \setminus \{h\}\} && \text{by definition of } . \setminus h \\
&= \{\{\}\} && \text{by set theory} \\
&= \llbracket 1 \rrbracket && \text{by definition of } \llbracket . \rrbracket \\
&= \llbracket \texttt{Remove } 1\ h \rrbracket && \text{by definition of } \texttt{Remove . } h
\end{aligned}$$

Case $\Theta = g$:

$$\begin{aligned}
&\llbracket g \rrbracket \setminus h \\
&= \{\{g\}\} \setminus h && \text{by definition of } \llbracket . \rrbracket \\
&= \{hs | hs \in \{\{g\}\} \wedge hs \setminus \{h\}\} && \text{by definition of } . \setminus h \\
&\quad \textit{Subcase } h = g \\
&= \{\{\}\} && \text{by set theory} \\
&= \llbracket 1 \rrbracket && \text{by definition of } \llbracket . \rrbracket \\
&= \llbracket \texttt{Remove } 1\ h \rrbracket && \text{by definition of } \texttt{Remove . } h \\
&\quad \textit{Subcase } h \neq g \\
&= \{\{g\}\} && \text{by set theory} \\
&= \llbracket g \rrbracket && \text{by definition of } \llbracket . \rrbracket \\
&= \llbracket \texttt{Remove } g\ h \rrbracket && \text{by definition of } \texttt{Remove . } h
\end{aligned}$$

Case $\Theta = \Theta_1 \cdot \Theta_2$:

$$
\begin{aligned}
& [\![\Theta_1 \cdot \Theta_2]\!] \setminus h \\
&= ([\![\Theta_1]\!] \bullet [\![\Theta_2]\!]) \setminus h && \text{by definition of } S_1 \bullet S_2 \\
&= \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!]\} \setminus h && \text{by definition of } [\![.]\!] \\
&= \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!] \wedge (hs_1 \cup hs_2) \setminus h\} && \text{by definition of } . \setminus h \\
&= \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!] \wedge hs_1 \setminus h \wedge hs_2 \setminus h)\} && \text{by set theory and logic} \\
&= \{hs_1 \cup hs_2 | (hs_1 \in [\![\Theta_1]\!] \wedge hs_1 \setminus h) \wedge (hs_2 \in [\![\Theta_2]\!] \wedge hs_2 \setminus h)\} && \text{by set theory and logic} \\
&= \{hs_1 | hs_1 \in [\![\Theta_1]\!] \wedge hs_1 \setminus h\} \bullet \{hs_2 | hs_2 \in [\![\Theta_2]\!] \wedge hs_2 \setminus h\} && \text{by definition of } S_1 \bullet S_2 \\
&= [\![\Theta_1]\!] \setminus h \bullet [\![\Theta_2]\!] \setminus h && \text{by definition of } . \setminus h \\
&= [\![\texttt{Remove } \Theta_1 \ h]\!] \bullet [\![\texttt{Remove } \Theta_2 \ h]\!] && \text{by induction hypothesis} \\
&= [\![(\texttt{Remove } \Theta_1 \ h) \cdot (\texttt{Remove } \Theta_2 \ h)]\!] && \text{By definition of } [\![.]\!] \\
&= [\![\texttt{Remove } (\Theta_1 \ \cdot \Theta_2) \ h]\!] && \text{by definition of } \texttt{Remove } . \ h
\end{aligned}
$$

Case $\Theta = \Theta_1 + \Theta_2$:

$$
\begin{aligned}
& [\![\Theta_1 + \Theta_2]\!] \setminus h \\
&= ([\![\Theta_1]\!] \cup [\![\Theta_2]\!]) \setminus h && \text{by definition of } [\![.]\!] \\
&= \{hs | hs \in ([\![\Theta_1]\!] \cup [\![\Theta_2]\!]) \wedge hs \setminus \{h\}\} && \text{by definition of } . \setminus h \\
&= \{hs | hs \in ([\![\Theta_1]\!] \cup [\![\Theta_2]\!]) \wedge hs \setminus \{h\}\} && \text{by definition of } . \setminus h \\
&= \{hs_1 | hs_1 \in [\![\Theta_1]\!] \wedge hs_1 \setminus \{h\}\} \cup && \text{by logic and set theory} \\
& \quad \{hs_2 | hs_2 \in [\![\Theta_2]\!] \wedge hs_2 \setminus \{h\}\} \\
&= [\![\Theta_1]\!] \setminus h \cup [\![\Theta_2]\!] \setminus h && \text{by definition of } . \setminus h \\
&= [\![\texttt{Remove } \Theta_1 \ h]\!] \cup [\![\texttt{Remove } \Theta_2 \ h]\!] && \text{by induction hypothesis} \\
&= [\![(\texttt{Remove } \Theta_1 \ h) \cdot ([\![\texttt{Remove } \Theta_2 \ h)]\!] && \text{by definition of } [\![.]\!] \\
&= [\![\texttt{Remove } (\Theta_1 \cdot \Theta_2) \ h]\!] && \text{by definition of } \texttt{Remove } . \ h
\end{aligned}
$$

◀

Emptiness `Empty` Θ checks if Θ is semantically empty. Syntactically we define *Empty* by induction on Θ as shown in Figure 26.

$$
\begin{aligned}
\texttt{Empty } 0 &\triangleq \mathit{true} \\
\texttt{Empty } 1 &\triangleq \mathit{false} \\
\texttt{Empty } h &\triangleq \mathit{false} \\
\texttt{Empty } (\Theta_1 \cdot \Theta_2) &\triangleq \texttt{Empty } \Theta_1 \wedge \texttt{Empty } \Theta_2 \\
\texttt{Empty } (\Theta_1 + \Theta_2) &\triangleq \texttt{Empty } \Theta_1 \wedge \texttt{Empty } \Theta_2
\end{aligned}
$$

Figure 26 Syntactic definition of the `Remove` Θ h operator

The equivalence of the syntactic and semantic definition is captured by Lemma 7.

▶ **Lemma 7.** $[\![\Theta]\!] == \{\}$ *if and only if* `Empty` Θ.

Proof. By induction on Θ.
Case $\Theta = 0$: We have $[\![0[\![= \{\}$ and *Empty* $0 = \mathit{true}$.
Case $\Theta = 1$:
We have $[\![1[\![\neq \{\}$ and *Empty* $1 = \mathit{false}$.
Case $\Theta = h$:
We have $[\![h[\![\neq \{\}$ and *Empty* $h = \mathit{false}$.
Case $\Theta = \Theta_1 \cdot \Theta_2$: By definition we have $[\![\Theta_1 \cdot \Theta_2]\!] = [\![\Theta_1]\!] \bullet [\![\Theta_2]\!]$ which is equal to $\{s_1 \cup s_2 \mid s_1 \in [\![\Theta_1]\!] \wedge s_2 \in [\![\Theta_2]\!]\}$. It follows that $[\![\Theta_1 \cdot \Theta_2[\![\neq \{\}$ iff $[\![\Theta_1[\![\neq \{\}$ and $[\![\Theta_2[\![\neq \{\}$. By induction hypothesis, we have $[\![\Theta_1[\![\neq \{\}$ if and only if *Empty* $\Theta_1 = \mathit{true}$, and $[\![\Theta_2[\![\neq \{\}$ if and only if *Empty* $\Theta_2 = \mathit{true}$. The result follows as *Empty* $(\Theta_1 \cdot \Theta_2) =$ `Empty` $\Theta_1 \wedge$ `Empty` Θ_2.
Case $\Theta = \Theta_1 + \Theta_2$: By definition we have $[\![\Theta_1 + \Theta_2]\!] = [\![\Theta_1]\!] \cup [\![\Theta_2]\!]$. It follows that $[\![\Theta_1 \cdot \Theta_2[\![\neq \{\}$ iff $[\![\Theta_1[\![\neq \{\}$ and $[\![\Theta_2[\![\neq \{\}$. By induction hypothesis, we have $[\![\Theta_1[\![\neq \{\}$ if and only if *Empty* $\Theta_1 = \mathit{true}$, and $[\![\Theta_2[\![\neq \{\}$ if and only if *Empty* $\Theta_2 = \mathit{true}$. The result follows as *Empty* $(\Theta_1 + \Theta_2) =$ `Empty` $\Theta_1 \wedge$ `Empty` Θ_2. ◀

C Safety of SafeP4

We prove safety in terms of progress and preservation. Both theorems make use of the relation $H \models \Theta$ as defined in Figure 17. The empty header instance map only entails the empty header instance type 1 (Rule Ent-Empty). If a header instance h is contained in the map of valid header instances H, H entails the header instance type h (Rule Ent-Inst). The sequence type $\Theta_1 \cdot \Theta_2$ is entailed by the distinct union of the maps entailing Θ_1 and Θ_2 respectively (Rule Ent-Seq) and the choice type $\Theta_1 + \Theta_2$ is entailed either by the map entailing Θ_1 or the map entailing Θ_2 (Rules Ent-ChoiceL and Ent-ChoiceR).

We prove progress and preservation only for commands. For expressions we formulate these properties as additional lemmas (Lemmas 8 and 9). The respective proofs are straightforward for our system.

▶ **Lemma 8** (Expression Progress). *If* $\cdot;\Theta \vdash e : t$ *and* $H \models \Theta$*, then either* e *is a value or* $\exists e'.\langle H, e\rangle \rightarrow e'$*.*

▶ **Lemma 9** (Expression Preservation). *If* $\Gamma;\Theta \vdash e : t$ *and* $H \models \Theta$ *and* $\langle H, e\rangle \rightarrow e'$ *then* $\Gamma;\Theta \vdash e' : t$*.*

▶ **Lemma 10** (Expression Substitution). *If* $\Gamma, x : \tau; \Theta \vdash e : \tau'$ *and* $\cdot;\cdot \vdash \bar{v} : \bar{\tau}$ *then* $\Gamma;\Theta \vdash e[\bar{v}/\bar{x}] : \tau'$

▶ **Lemma 11.** *If* $H \models \Theta$ *then* $\mathit{dom}(H) \in [\![\Theta]\!]$*.*

Proof. By induction on Theta.
Case $\Theta = 0$: The case immediately holds as $H \models 0$ is a contradiction.
Case $\Theta = 1$: By inversion of *Entailment*, $H = \cdot$, and so $\mathit{dom}(H) = \{\} \in [\![1]\!] = \{\{\}\}$.
Case $\Theta = h$: By inversion of *Entailment*, $\mathit{dom}(H) = \{h\} \in [\![h]\!] = \{\{h\}\}$.
Case $\Theta = \Theta_1 \cdot \Theta_2$: By inversion of *Entailment*, $H = H_1 \cup H_2, H_1 \models \Theta_1, H_2 \models \Theta_2$.
By induction hypothesis, $\mathit{dom}(H_1) \in [\![\Theta_1]\!]$ and $\mathit{dom}(H_2) \in [\![\Theta_2]\!]$.
By set theory, $\mathit{dom}(H) = \mathit{dom}(H_1) \cup \mathit{dom}(H_2)$
By induction hypothesis, $\mathit{dom}(H_1) \in [\![\Theta_1]\!]$ and $\mathit{dom}(H_2) \in [\![\Theta_2]\!]$.
By definition of $[\![.]\!]$ and $(\bullet)$, $[\![\Theta]\!] = [\![\Theta_1]\!] \bullet [\![\Theta_2]\!] = \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!]\}$ and

therefore $dom(H_1) \cup dom(H_2) \in \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!]\}$, i.e., $dom(H) \in [\![\Theta]\!]$.
Case $\Theta = \Theta_1 + \Theta_2$: By inversion of *Entailment*, either $H \models \Theta_1$ or $H \models \Theta_2$.
Subcase $H \models \Theta_1$: By the induction hypothesis, $dom(H) \in [\![\Theta_1]\!]$. and by set theory $dom(H) \in [\![\Theta_1]\!] \cup [\![\Theta_2]\!]$.
Subcase $H \models \Theta_2$: Symmetric to the previous subcase. ◀

▶ **Lemma 12.** *If $H \models \Theta$ and* `Includes` Θ h*, then $h \in dom(H)$.*

Proof. By induction on Θ.
Case $\Theta = 0$: The case immediately holds as $H \models 0$ is a contradiction.
Case $\Theta = 1$: By inversion of *Entailment*, $H = \cdot$ The case immediately holds, as `Includes` Θ h is a contradiction.
Case $\Theta = g$:
By inversion of *Entailment*, $dom(H) = \{g\}$
By assumption `Includes` Θ h, $h = g$.`Includes` $\{g\}$ g, i.e., $h \in dom(H)$.
Case $\Theta = \Theta_1 \cdot \Theta_2$:
By inversion of *Entailment*, $H = H_1 \cup H_2, H_1 \models \Theta_1, H_2 \models \Theta_2$.
By set theory $dom(H) = dom(H_1) \cup dom(H_2)$
By definition of *Inclusion* and by assumption `Includes` Θ h, `Includes` Θ_1 $h \vee$ `Includes` Θ_2 h
Subcase `Includes` Θ_1 h: By induction hypothesis, $h \in dom(H_1)$ and by assumption $dom(H_1) \subseteq dom(H)$, we can conclude $h \in dom(H)$.
Subcase `Includes` Θ_2 h: Symmetric to the previous subcase.
Case $\Theta = \Theta_1 + \Theta_2$:
By inversion of *Entailment*, either $H \models \Theta_1$ or $H \models \Theta_2$.
By definition of *Inclusion* and by assumption `Includes` Θ h, `Includes` Θ_1 h and `Includes` Θ_2 h.
Subcase $H \models \Theta_1$: By induction hypothesis, we can conclude $h \in dom(H)$.
Subcase $H \models \Theta_2$: Symmetric to the previous subcase. ◀

C.1 Control Plane Assumptions

The following propositions model the assumptions about the control plane functions $\mathcal{CA}$ and $\mathcal{CV}$ that are required to prove type safety.

▶ **Proposition 13** (Control Plane Reads)**.** If $H \models \Theta$ and $\mathcal{CV}(t) = (\bar{e}, \bar{S})$ and $\Gamma; \Theta \vdash e_i : \tau_i$ for $e_i \in \bar{e}$ then $\mathcal{CA}(t, H) = (a_i, \bar{v})$.

▶ **Proposition 14** (Control Plane Action Data)**.** If $H \models \Theta$ and $\mathcal{CA}(t, H) = (a_i, \bar{v})$ and $\mathcal{A}(a_i) = \lambda\bar{x} : t\bar{a}u.\ c_i$ then $\cdot; \cdot \vdash \bar{v} : t\bar{a}u$

▶ **Proposition 15** (Control Plane Assumptions)**.** If $H \models \Theta$ and $\mathcal{CA}(t, H) = (a_i, \bar{v})$ and $\mathcal{CV}(t) = \bar{S}$ then $H \models$ `Restrict` Θ S.

C.2 Progress

▶ **Theorem 16** (Progress)**.** *If $\cdot \vdash c : \Theta \Rrightarrow \Theta'$ and $H \models \Theta$, then either $c = skip$ or $\exists \langle I', O', H', c' \rangle.\ \langle I, O, H, c \rangle \rightarrow \langle I', O', H', c' \rangle$*

Proof. By induction on typing derivations of $\cdot \vdash c : \Theta \Rrightarrow \Theta'$.

Case T-Skip: $c = skip$

Immediate.

Case T-EXTR: $c = \mathit{extract}(h)$

Let $(I', v) = \mathit{deserialize}_\eta(I)$ and $O' = O$ and $H' = H[h \mapsto v]$ and $c' = \mathit{skip}$. The result follows by E-SKIP.

Case T-EMIT: $c = \mathit{emit}(h)$

If $h \notin \mathit{dom}(H)$, let $I' = I$ and $O' = O$ and $H' = H$, and $c' = \mathit{skip}$. The result follows by E-EMITINVALID. Otherwise, $h \in \mathit{dom}(H)$. Let $H(h) = v$ and $\bar{B} = \mathit{serialize}_\eta(v)$ and $I' = I$ and $O' = O.\bar{B}$ and $H' = H$ and $c' = \mathit{skip}$. The result follows by E-EMIT.

Case T-SEQ: $c = c_1; c_2$ and $\cdot \vdash c_1 : \Theta \Rrightarrow \Theta_1$ and $\cdot \vdash c_2 : \Theta_1 \Rrightarrow \Theta_2$

By induction hypothesis, c_1 is either *skip* or there is some $\langle I', O', H', c_1' \rangle$, such that $\langle I, O, H, c_1 \rangle \to \langle I', O', H', c_1' \rangle$.
If $c = \mathit{skip}$, let $I' = I$ and $O' = O$ and $H' = H$ and $c' = c_2$. The result follows by E-SEQ. Otherwise, the result follows by E-SEQ1.

Case T-IF: $c = \mathit{if}\ (e)\ \mathit{then}\ c_1\ \mathit{else}\ c_2$ and $\cdot; \Theta \vdash e : \mathit{Bool}$ and $\cdot \vdash c_1 : \Theta \Rrightarrow \Theta_1$ and $\cdot \vdash c_2 : \Theta \Rrightarrow \Theta_2$

By the progress theorem for expressions, we have that e is either `true`, `false`, or there is some e' such that $\langle H, e \rangle \to e'$.

Subcase $e =$ `true`: Let $I' = I$ and $O' = O$ and $H' = H$ and $c' = c_1$. The result follows by E-IFTRUE.

Subcase $e =$ `false`: Symmetric to the previous case.

Subcase $\langle H, e \rangle \to e'$: Let $I' = I$ and $O' = O$ and $H' = H$ and $c' = \mathit{if}\ (e')\ c_1\ c_2$. The result follows by E-IF.

Case T-IFVALID: $c = \mathit{if\_valid}\ (e)\ \mathit{then}\ c_1\ \mathit{else}\ c_2$

If $h \in \mathit{dom}(H)$, let $I' = I$ and $O' = O$ and $H' = H$ and $c' = c_1$. The result follows by E-IFVALIDTRUE Otherwise, $h \notin \mathit{dom}(H)$. Let $I' = I$ and $O' = O$ and $H' = H$ and $c' = c_2$. The result follows by E-IFVALIDFALSE

Case T-APPLY: $c = t.\mathit{apply}()$

By Proposition 13, we have $\mathcal{CA}(t, H) = (a_i, \bar{v})$. Let $\mathcal{A}(a) = \lambda\bar{x} : t\bar{a}u.\ c_i$. Let $I' = I$ and $O' = O$ and $H' = H$ and $c' = c_i[\bar{v}/\bar{x}]$. The result follows by E-APPLY.

Case T-ADD: $c = \mathit{add}(h)$

If $h \in \mathit{dom}(H)$, let $I' = I$ and $O' = O$ and $H' = H$ and $c' = \mathit{skip}$. The result follows by E-ADDVALID. Otherwise, $h \notin \mathit{dom}(H)$. Let $v = \mathit{init}_\eta$ and $I' = I$ and $O' = O$ and $H' = H[h \mapsto v]$ and $c' = \mathit{skip}$. The result follows by E-ADD

Case T-REMOVE: $c = \mathit{remove}(h)$

Let $I' = I$ and $O' = O$ and $H' = H \setminus h$ and $c' = \mathit{skip}$. The result follows by E-REMOVE.

Case T-MOD: $c = h.f = e$ and `Includes` $\Theta\ h$ and $\mathcal{F}(h, f) = \tau_i$ and $\cdot; \Theta \vdash e : \tau_i$

By the progress rule for expressions, either e is a value or there is some e' such that $\langle H, e \rangle \to e'$.

Subcase $e = v$: By Lemma 12: $h \in \mathit{dom}(H)$. Let $r = H(h)$ and $r' = \{r\ \mathit{with}\ f = v\}$. Also let $I' = I$ and $O' = O$ and $H' = H[h \mapsto r']$ and $c' = \mathit{skip}$. The result follows by E-MOD.

Subcase $\langle H, e \rangle \to e'$: Let $I' = I$ and $O' = O$ and $H' = H$ and $c' = h.f = e'$. The result follows by E-MOD1.

Case T-ZERO: `Empty` Θ_1

By Lemma 11, we have $\mathit{dom}(H) \in [\![\Theta_1]\!]$. By Lemma 7, we have $[\![\Theta_1]\!] = \{\}$, which is a contradiction. ◀

C.3 Preservation

▶ **Lemma 17.** *If $H \models \Theta$ and $h \in dom(H)$ then $H \models \texttt{Restrict}\ \Theta\ h$.*

Proof. By induction on Θ.
Case $\Theta = 0$: The case immediately holds as $H \models 0$ is a contradiction.
Case $\Theta = 1$: By inversion of *Entailment*, $H = \cdot$. The case immediately holds as $h \in dom(\cdot)$ is a contradiction.
Case $\Theta = g$: By inversion of *Entailment*, $dom(H) = \{g\}$, and so $h = g$.
By definition of *Restriction* $\texttt{Restrict}\ \Theta\ h = \texttt{Restrict}\ g\ g = g$. By Ent-Inst $H \models g$, i.e., $H \models \Theta$.
Case $\Theta = \Theta_1 \cdot \Theta_2$: By inversion of *Entailment* $H = H_1 \cup H_2, H_1 \models \Theta_1, H_2 \models \Theta_2$. By $h \in dom(H)$, either $h \in dom(H_1)$ or $h \in dom(H_2)$.
Subcase $h \in dom(H_1)$: By the induction hypothesis, we have $H_1 \models \texttt{Restrict}\ \Theta_1\ h$. By Ent-Seq, we have $H_1 \cup H_2 \models \texttt{Restrict}\ \Theta_1\ h \cdot \Theta_2$. By Ent-ChoiceL, we have $H_1 \cup H_2 \models (\texttt{Restrict}\ \Theta_1\ h \cdot \Theta_2) + (\Theta_1 \cdot \texttt{Restrict}\ \Theta_2\ h)$ which finishes the case.
Subcase $h \in dom(H_2)$: Symmetric to the previous subcase.
Case $\Theta = \Theta_1 + \Theta_2$: By inversion of *Entailment*, either $H \models \Theta_1$ or $H \models \Theta_2$.
Subcase $H \models \Theta_1$: By the induction hypothesis, we have $H \models \texttt{Restrict}\ \Theta_1\ h$. By Ent-ChoiceL, $H \models \texttt{Restrict}\ \Theta_1\ h + \texttt{Restrict}\ \Theta_2\ h$.
Subcase $H \models \Theta_2$: Symmetric to the previous subcase. ◀

▶ **Lemma 18** (NegRestrict-Domain-Entail)**.** *If $H \models \Theta$ and $h \notin dom(H)$ then $H \models \texttt{NegRestrict}\ \Theta\ h$.*

Proof. By induction on Θ.
Case $\Theta = 0$: The case immediately holds as $H \models 0$ is a contradiction.
Case $\Theta = 1$: By inversion of *Entailment*, $H = \cdot$. By definition of *Negated Restriction*, $\texttt{NegRestrict}\ \Theta\ h = \texttt{NegRestrict}\ 1\ h = 1$. By Ent-Empty $\cdot \models 1$, i.e., $H \models \texttt{NegRestrict}\ \Theta\ h$.
Case $\Theta = g$: By inversion of *Entailment*, $dom(H) = \{g\}$. By the induction hypothesis $h \neq g$. By definition of *Restriction* $\texttt{NegRestrict}\ \Theta\ h = \texttt{NegRestrict}\ g\ h = g$. By Ent-Inst $H \models g$, i.e., $H \models \texttt{NegRestrict}\ \Theta\ h$.
Case $\Theta = \Theta_1 \cdot \Theta_2$: By inversion of *Entailment*, $H = H_1 \cup H_2, H_1 \models \Theta_1, H_2 \models \Theta_2$. By $h \notin dom(H)$, $h \notin dom(H_1)$ and $h \notin dom(H_2)$. By the induction hypothesis, $H_1 \models \texttt{NegRestrict}\ \Theta_1\ h$ and $H_2 \models \texttt{NegRestrict}\ \Theta_2\ h$. By Ent-Seq, $H_1 \cup H_2 \models \texttt{NegRestrict}\ \Theta_1\ h \cdot \texttt{NegRestrict}\ \Theta_2\ h$ which finishes the case.
Case $\Theta = \Theta_1 + \Theta_2$: By inversion of *Entailment*, either $H \models \Theta_1$ or $H \models \Theta_2$.
Subcase $H \models \Theta_1$: By the induction hypothesis, we have $H \models \texttt{NegRestrict}\ \Theta_1\ h$. By Ent-ChoiceL, $H \models \texttt{NegRestrict}\ \Theta_1\ h + \texttt{NegRestrict}\ \Theta_2\ h$.
Subcase $H \models \Theta_2$: Symmetric to the previous subcase. ◀

▶ **Lemma 19** (Substitution)**.** *If $\Gamma, x : \tau; \Theta \vdash c : \Theta'$ and $\cdot; \cdot \vdash v : \tau$ then $\Gamma; \Theta \vdash c : \Theta'$*

Proof. By straightforward induction on the derivation $\Gamma, x : \tau; \Theta \vdash c : \Theta'$. ◀

▶ **Lemma 20** (Entails-Add)**.** *If $H \models \Theta$ then $H[h \mapsto v] \models \Theta \cdot h$*

Proof. We analyze two cases.

Case $h \in dom(H)$: By the assumption of the case, we have $dom(H) = dom(H[h \mapsto v])$. Let $H_1 = H[h \mapsto v]$ and $H_2 = \{h \mapsto v\}$. Observe that $H[h \mapsto v] = H_1 \cup H_2$. By

Lemma 22, we have that $H_1 \models \Theta$. By ENT-INST we have $H_2 \models h$. By ENT-SEQ we have $H[h \mapsto v] \models \Theta \cdot h$.

Case hnot $\in dom(H)$: Let $H_1 = H$ and $H_2 = \{h \mapsto v\}$. Observe that $H[h \mapsto v] = H_1 \cup H_2$. By assumption we have $H_1 \models \Theta$. By ENT-INST we have $H_2 \models h$. By ENT-SEQ we have $H[h \mapsto v] \models \Theta \cdot h$.

◀

▶ **Lemma 21** (Entails-Remove). *If* $H \models \Theta$ *then* $H \setminus h \models \texttt{Remove}\ \Theta\ h$.

Proof. By induction on Θ.
Case $\Theta = 0$: The case immediately holds, as $H \models 0$ is a contradiction.

Case $\Theta = 1$: By inversion of *Entailment*, $H = \cdot$. By set theory, $\cdot \backslash h = \cdot$ and $\texttt{Remove}\ 1\ h = 1$. By ENT-EMPTY, $\cdot \models 1$.

Case $\Theta = g$: By inversion of *Entailment*, $dom(H) = \{g\}$.
Subcase $g = h$: By set theory $H \setminus h = \cdot$. By definition of *Remove*, $\texttt{Remove}\ \Theta\ h = 1$. By ENT-EMPTY, $\cdot \models 1$, which concludes the case.
Subcase $g \neq h$: By set theory $H \setminus h = H$. By definition of *Remove*, $\texttt{Remove}\ \Theta\ h = g$. By assumption, $H \models \Theta$, which concludes the case.

Case $\Theta = \Theta_1 \cdot \Theta_2$: By inversion of *Entailment*, $H = H_1 \cup H_2, H_1 \models \Theta_1, H_2 \models \Theta_2$. By induction hypothesis, $H_1 \setminus h \models \texttt{Remove}\ \Theta_1\ h$ and $H_2 \setminus h \models \texttt{Remove}\ \Theta_2\ h$. By set theory, $H_1 \setminus h\ \cup H_2 \setminus h = (H_1 \cup H_2) \setminus h$. By definition of *Removal*, $\texttt{Remove}\ \Theta_1\ h \cdot \texttt{Remove}\ \Theta_2\ h = \texttt{Remove}\ (\Theta_1 \cdot \Theta_2)\ h$. By ENT-SEQ, $(H_1 \cup H_2) \setminus h \models \texttt{Remove}\ (\Theta_1 \cdot \Theta_2)\ h$.

Case $\Theta = \Theta_1 + \Theta_2$: By inversion of *Entailment*, either $H \models \Theta_1$ or $H \models \Theta_2$. By definition of *Removal*, $\texttt{Remove}\ \Theta_1\ h + \texttt{Remove}\ \Theta_2\ h = \texttt{Remove}\ (\Theta_1 + \Theta_2)\ h$. *Subcase* $H \models \Theta_1$: By induction hypothesis, $H \setminus h \models \Theta_1 \setminus h$. By ENT-CHOICEL, applied to $H \setminus h \models \texttt{Remove}\ \Theta_1\ h$, and $\texttt{Remove}\ \Theta_2\ h$, we can conclude $H \setminus h \models \texttt{Remove}\ \Theta_1\ h + \texttt{Remove}\ \Theta_2\ h$. By definition of *Removal*, $H \setminus h \models \texttt{Remove}\ (\Theta_1 + \Theta_2)\ h$. ◀

▶ **Lemma 22.** *If* $H \models \Theta$ *and* $dom(H) = dom(H')$ *then* $H' \models \Theta$.

Proof. By induction on Θ.
Case $\Theta = 0$: The case immediately holds as $H \models 0$ is a contradiction.
Case $\Theta = 1$: By inversion of *Entailment* $H = \cdot$. By assumption $dom(H) = dom(H')$, $H' = \cdot$ and by ENT-EMPTY, $H' \models \Theta$.
Case $\Theta = g$: By inversion of *Entailment*, $dom(H) = \{g\}$. By assumption $dom(H) = dom(H')$ and by Ent-Inst, $H' \models \Theta$.
Case $\Theta = \Theta_1 \cdot \Theta_2$: By inversion of *Entailmment*, $H = H_1 \cup H_2, H_1 \models \Theta_1, H_2 \models \Theta_2$. By set theory, $dom(H) = dom(H_1) \cup dom(H_2)$. By induction hypothesis if $dom(H_1') = dom(H_1)$ and $dom(H_2') = dom(H_2)$, then $H_1' \models \Theta_1, H_2' \models \Theta_2$. By ENT-SEQ, $H' = H_1' \cup H_2' \models \Theta_1 \cdot \Theta_2$.
Case $\Theta = \Theta_1 + \Theta_2$: By inversion of *Entailment*, either $H \models \Theta_1$ or $H \models \Theta_2$.
Subcase $H \models \Theta_1$: By induction hypothesis, we have $H' \models \Theta_1$. By ENT-CHOICEL, $H' \models \Theta_1 + \Theta_2$.
Subcase $H \models \Theta_2$: Symmetric to the previous subcase. ◀

We define $\Theta_1 < \Theta_2 \triangleq [\![\Theta_1]\!] \subseteq [\![\Theta_2]\!]$, i.e., for every $S \in [\![\Theta_1]\!], S \in [\![\Theta_2]\!]$.

▶ **Lemma 23.** *If* $\Theta_1' < \Theta_1$ *then* $\Theta_1 \cdot h < \Theta_1 \cdot h$.

Proof. We calculate as follows:

1. $[\![\Theta_1']\!] \subseteq [\![\Theta_1]\!]$ by (B) and the definition of $<$.

2. $[\![\Theta_1' \cdot h]\!] == [\![\Theta_1']\!] \bullet \{\{h\}\}$ by definition of $[\![.]\!]$
3. $[\![\Theta_1 \cdot h]\!] == [\![\Theta_1]\!] \bullet \{\{h\}\}$ by definition of $[\![.]\!]$
4. Let $S \in [\![\Theta_1']\!] \bullet \{\{h\}\}$.
5. $S = S' \cup \{h\}$, where $S' \in [\![\Theta_1']\!]$ by def of $\bullet$.
6. By 1., $S' \in [\![\Theta_1]\!]$
7. By set theory, $S' \cup \{h\} \in [\![\Theta_1]\!] \bullet \{\{h\}\}$.
8. Then $[\![\Theta_1']\!] \bullet \{\{h\}\} \subseteq [\![\Theta_1]\!] \bullet \{\{h\}\}$

◀

▶ **Lemma 24.** *If* $\Theta_1' < \Theta_1$ *then* $[\![\texttt{Remove}\ \Theta_1'\ h]\!] \subseteq [\![\texttt{Remove}\ \Theta_1\ h]\!]$.

Proof. Since $[\![\texttt{Remove}\ \Theta_1'\ h]\!] == [\![\Theta_1']\!] \setminus h$ and $[\![\texttt{Remove}\ \Theta_1\ h]\!] == [\![\Theta_1]\!] \setminus h$ by Lemma 6, we can equivalently show that $[\![\Theta_1']\!] \setminus h \subseteq [\![\Theta_1]\!] \setminus h$, which follows from set theory. ◀

▶ **Lemma 25.** *If* $\Theta_1' < \Theta_1$ *then* $[\![\texttt{Restrict}\ \Theta_1'\ h]\!] \subseteq [\![\texttt{Restrict}\ \Theta_1\ h]\!]$

Proof. By Lemma 3, $[\![\texttt{Restrict}\ \Theta_1'\ h]\!] == [\![\Theta_1']\!]|h$ and $[\![\texttt{Restrict}\ \Theta_1\ h]\!] == [\![\Theta_1]\!]|h$. By set theory, $[\![\Theta_1']\!]|h \subseteq [\![\Theta_1]\!]|h$ when $[\![\Theta_1']\!] \subseteq [\![\Theta_1]\!]$, so we are done. ◀

▶ **Lemma 26.** *If* $\Theta_1' < \Theta_1$ *then* $[\![\texttt{NegRestrict}\ \Theta_1'\ h]\!] \subseteq [\![\texttt{NegRestrict}\ \Theta_1\ h]\!]$

Proof. By Lemma 4, $[\![\texttt{NegRestrict}\ \Theta_1'\ h]\!] == [\![\Theta_1']\!]|\neg h$ and $[\![\texttt{NegRestrict}\ \Theta_1\ h]\!] == [\![\Theta_1]\!]|\neg h$. By set theory, $[\![\Theta_1']\!]|\neg h \subseteq [\![\Theta_1]\!]|\neg h$ when $[\![\Theta_1']\!] \subseteq [\![\Theta_1]\!]$, so we are done. ◀

▶ **Lemma 27.** *If* $\Theta' < \Theta$ *and* $\texttt{Includes}\ \Theta_1\ h$ *then* $\texttt{Includes}\ \Theta_1'\ h$.

Proof. By Lemma 5, $\texttt{Includes}\ \Theta_1'\ h = h \sqsubset \Theta_1'$. By the same lemma, $\texttt{Includes}\ \Theta_1\ h = h \sqsubset \Theta_1$. Let $S \in [\![\Theta_1']\!]$ to show $h \in S$ and hence $h \sqsubset \Theta_1'$. Since $[\![\Theta_1']\!] \subseteq [\![\Theta_1]\!]$, by assumption and definition of $<$, then $S \in [\![\Theta_1]\!]$. Since $h \sqsubset \Theta_1$, conclude $h \in S$ and we are done. ◀

▶ **Lemma 28.** *If* $\Theta_a' < \Theta_a$ *and* $\Theta_b' < \Theta_b$ *then* $\Theta_a' + \Theta_b' < \Theta_a + \Theta_b$.

Proof. We have to show that $[\![\Theta_a' + \Theta_b']\!] \subseteq [\![\Theta_a + \Theta_b]\!]$ when $\Theta_a' < \Theta_a$ and $\Theta_b' < \Theta_b$. By definition of $[\![.]\!]$ we can equally show that $[\![\Theta_a']\!] \cup [\![\Theta_b']\!] \subseteq [\![\Theta_a]\!] \cup [\![\Theta_b]\!]$, which follows from set theory. ◀

▶ **Lemma 29.** *If* $\Gamma; \Theta_1 \vdash a : \bar{\tau} \to \Theta_2$ *and* $\Theta_1' < \Theta_1$, *then* $\exists \Theta_2'. \Gamma; \Theta_1' \vdash a : \bar{\tau} \to \Theta_2'$ *and* $\Theta_2' < \Theta_2$.

Proof. There is only one way to have concluded that $\Gamma; \Theta_1 \vdash a : \bar{\tau} \to \Theta_2$: via the [T-Action] rule, which gives us two facts: we know $a - \lambda\bar{x} : \bar{\tau}.c$, and $\Gamma, \bar{x} : \bar{\tau} \vdash \Theta_2 : \Theta_1 \Mapsto$.

Since this c is an action command, is only generated by the add, remove, modification and sequence commands. So we perform a limited induction on the structure of c:

Case $c = add(h)$. The only typing rule that applies is T-Add, so we know $\Theta_2 = \Theta_1 \cdot h$. Now let $\Theta_2' = \Theta_1' \cdot h$. Then T-Add shows $\Gamma, \bar{x} : \bar{\tau} \vdash add(h) : \Theta_1' \Mapsto \Theta_1' \cdot h$. Then $\Theta_1' \cdot h < \Theta_1 \cdot h$ follows by Lemma 23, and we are done.

Case $c = remove(h)$. The only typing rule that could have applied is T-Remove, so we know that $\Theta_2 = \texttt{Remove}\ \Theta_1\ h$. Let $\Theta_2' = \texttt{Remove}\ \Theta_1'\ h$. Then T-Remove shows $\Gamma, \bar{x} : \bar{\tau} \vdash remove(h) : \Theta_1' \Mapsto \mathit{Remove}\ \Theta_1\ h$. Then $\mathit{Remove}\ \Theta_1'\ h < \mathit{Remove}\ \Theta_1\ h$ by Lemma 24.

Case $c = h.f = v$. The only typing rule that could have applied is T-Mod, so we know that $\Theta_2 = \Theta_1$, Let $\Theta_2' = \Theta_1'$, which proves $\Theta_2' < \Theta_2$ by assumption.

We know by our case assumtion that $\Gamma, \bar{x} : \bar{\tau}; \Theta_1 \vdash e : \mathcal{F}(h, f)$ and `Includes` $\Theta 1$ h. By T-Mod, we only need to show that (1) $\Gamma, \bar{x} : \bar{\tau} \vdash e : \mathcal{F}(h, f)$ and (2) `Includes` Θ_1' h. (1) follows by Lemma 30, and (2) follows by Lemma 27.

Case $c = c_1; c_2$. The only rule that could have applied is T-Seq, so we know that $\Gamma, \bar{x} : \bar{\tau} \vdash c_1 : \Theta_1 \Rrightarrow \Theta_{11}$, and $\Gamma, \bar{x} : \bar{\tau} \vdash c_2 : \Theta_{11} \Rrightarrow \Theta_2$, and $\Theta_1' \leq \Theta_1$.

The inductive hypothesis on c_1 gives us a $\Theta_{11}' < \Theta_{11}$ such that $\Gamma, \bar{x} : \bar{\tau} \vdash c_1 : \Theta_1' \Rrightarrow T_{11}'$.

The inductive hypothesis on c_2 gives us a $\Theta_2' < \Theta_2$ such that $\Gamma, \bar{x} : \bar{\tau} \vdash c_2 : \Theta_{11}' \Rrightarrow \Theta_2'$.

The result follows by `T-Seq`.

◀

▶ **Lemma 30.** *If* $\Gamma; \Theta \vdash e : \tau$ *and* $\Theta' < \Theta$*, then* $\Gamma; \Theta' \vdash e : \tau$*.*

Proof. By induction on the typing derivation.

Case T-Constant We know $e = k(\bar{e}), \Gamma; \Theta \vdash e_i : \tau_i$ for all i, `typeof`$(k) = t\bar{a}u \to \tau$ and $\Theta' < \Theta$. By induction hypothesis, $\Gamma; \Theta' \vdash e_i : \tau_i$ for all i and we are done by T-Constant.

Case T-Var We know $e = x, x : \tau \in \Gamma$, and $\Theta' < \Theta$. We are done by T-Var.

Case T-Field We know $e = h.f$, `Includes` Θ h and $\Theta' < \Theta$. By Lemma 27, we know `Includes Θ′ h` and the result follows by T-Field. ◀

▶ **Lemma 31.** *If* $\Gamma \vdash c : \Theta_1 \Rrightarrow \Theta_2$ *and* $\Theta_1' < \Theta_1$*, then* $\exists \Theta_2'. \Gamma \vdash c : \Theta_1' \Rrightarrow \Theta_2'$ *and* $\Theta_2' < \Theta_2$*.*

Proof. By induction on a derivation of $\Gamma \vdash c : \Theta_1 \Rrightarrow \Theta_2$. We refer to assumptions $\Gamma \vdash c : \Theta_1 \Rrightarrow \Theta_2$ and $\Theta_1' < \Theta_1$ as (A) and (B) respectively. Similarly, we use (1) and (2) to refer to the proof goals $\exists \Theta_2'. \Gamma \vdash c : \Theta_1' \Rrightarrow \Theta_2'$ and $\Theta_2' < \Theta_2$ respectively.

Case T-Zero: By assumption, we have `Empty` Θ_1. By Lemmas 7 and 27 we have `Empty` Θ_1'. Let $\Theta_2' = \Theta_2$. We have $\Gamma \vdash c : \Theta_1' \Rrightarrow \Theta_2$ by T-Zero, proving (1), and $\Theta_2' < \Theta_2$ by reflexivity, proving (2).

Case T-Skip: We know $c = \mathit{skip}$ and $\Theta_2 = \Theta_1$ and $\Theta_1' < \Theta_1$. Let $\Theta_2' = \Theta_1'$. Then by assumption (B) $\Theta_2' = \Theta_1' < \Theta_1 = \Theta_2$, proving (2) and $\Gamma \vdash \mathit{skip} : \Theta_1' \Rrightarrow \Theta_1'$ by T-Skip, proving (1).

Case T-Emit: We know $c = \mathit{emit}(h)$ and $\Theta_2 = \Theta_1$ and $\Theta_1' < \Theta_1$. Let $\Theta_2' = \Theta_1'$. Then by assumption (B), $\Theta_2' = \Theta_1' < \Theta_1 = \Theta_2$, proving (2) and $\Gamma \vdash \mathit{emit}(h) : \Theta_1' \Rrightarrow \Theta_1'$ by T-Emit, proving (1).

Case T-Add: We know $c = \mathit{add}(h)$ and $\Theta_2 = \Theta_1 \cdot h$ and $\Theta_1' < \Theta_1$. (1) follows since we can prove $\Gamma \vdash \mathit{add}(h) : \Theta_1' \Rrightarrow \Theta_1' \cdot h$ by T-Add. (2), i.e., $\Theta_1' \cdot h < \Theta_1 \cdot h$, follows from Lemma 23.

Case T-Extr: Similar to case T-Add. We know $c = \mathit{extract}(h)$ and $\Theta_2 = \Theta_1 \cdot h$ and $\Theta_1' < \Theta_1$. Let $\Theta_2' = \Theta_1' \cdot h$. (1) follows since we can prove $\Gamma \vdash \mathit{extract}(h) : \Theta_1' \Rrightarrow \Theta_1' \cdot h$ by T-Extract. (2) follows by Lemma 23.

Case T-Rem: We know $c = \mathit{remove}(h)$ and $\Theta_2 =$ `Remove` Θ_1 h and $\Theta_1' < \Theta_1$. Let $\Theta_2' =$ `Remove` Θ_1' h. (1) follows by T-Rem and for (2) we have to show that `Remove` Θ_1' $h <$ `Remove` Θ_1 h, which follows from Lemma 24.

Case T-Mod: We know $c = h.f = e$ and $\Theta_2 = \Theta_1$ and $\Theta_1' < \Theta_1$. Let $\Theta_2' = \Theta_1'$. If $[\![\Theta_1']\!] = [\![0]\!]$ then $\Theta_1 == 0$ by idempotent semiring equality and (1) follows by T-Zero. Otherwise $[\![\Theta_1']\!]$ is nonempty. To show (1) we need to show

(a) `Includes` Θ_1' h,
(b) $\mathcal{F}(h, f) = \tau$,
(c) $\Gamma; \Theta_1 \vdash e : \tau$

(b) and (c) follow from the assumption that the previous rule in the typing derivation was T-MOD. This inversion also gives us `Includes` $\Theta_1\ h$. To show (a) we calculate as follows. $h \sqsubset [\![\Theta_1]\!]$ by Lemma 5, i.e. $h \in S$ for every $S \in [\![\Theta_1]\!]$ by definition. Since $[\![\Theta'_1]\!] \subseteq [\![\Theta_1]\!]$, $h \in S$ for every $S \in [\![\Theta'_1]\!]$, by set theory. By definition we get $h \sqsubset [\![\Theta'_1]\!]$. By Lemma 5, we can conclude `Includes` $\Theta'_1\ h$.

Case T-SEQ: We know $c = c_1; c_2$ and $\Gamma \vdash c_1 : \Theta_1 \Rrightarrow \Theta_{11}$ and $\Gamma \vdash c_2 : \Theta_{11} \Rrightarrow \Theta_2$ and $\Theta'_1 < \Theta_1$. By induction hypothesis, $\exists \Theta'_{11}.\Gamma \vdash c_1 : \Theta'_1 \Rrightarrow \Theta'_{11}$ and $\Theta'_{11} < \Theta_{11}$. Again, by induction hypothesis, $\exists \Theta'_2.\Gamma \vdash c_2 : \Theta'_{11} \Rrightarrow \Theta'_2$ (proving 1) and $\Theta'_2 < \Theta_2$ (proving 2) which concludes the case.

Case T-IFVALID: We know $c = \mathit{valid}(h)\ c_1\ \mathit{else}\ c_2$ and $\Gamma \vdash c_1 : \texttt{Restrict}\ \Theta_1\ h \Rrightarrow \Theta_t, \Gamma \vdash c_2 : \texttt{NegRestrict}\ \Theta_1\ h \Rrightarrow \Theta_f, \Theta_2 = \Theta_t + \Theta_f$, and $\Theta'_1 < \Theta_1$. Let $\Theta'_2 = \texttt{Restrict}\ \Theta'_1\ h + \texttt{NegRestrict}\ \Theta'_1\ h$. (1) is immediate from T-IFVALID. (2) follows from Lemmas 25, 26 and 28.

Case T-IF: We know $c = \mathit{if}\ (e)\ c_1\ \mathit{else}\ c_2, \Gamma \vdash c_1 : \Theta_1 \Rrightarrow \Theta_{11}, \Gamma \vdash c_2 : \Theta_1 \Rrightarrow \Theta_{12}, \Gamma; \Theta_1 \vdash e : \mathit{Bool}$ and $\Theta'_1 < \Theta_1$. By induction hypothesis, there exists Θ'_{11} such that (1a) $\Gamma \vdash c_1 : \Theta'_1 \Rrightarrow \Theta'_{11}$ and (2a) $\Theta'_{11} < \Theta_{11}$. Also by induction hypothesis, there exists Θ'_{12} such that (1b) $\Gamma \vdash c_2 : \Theta'_1 \Rrightarrow \Theta'_{12}$ and (2b) $\Theta'_{12} < \Theta_{12}$. Let $\Theta'_2 = \Theta'_{11} + \Theta'_{12}$. (1) follows by T-IF (1a), (1b), and the fact that $\Gamma; \Theta_1 \vdash e : \mathit{Bool}$. (2) follows by Lemma 28, (2a), (2b).

Case T-APPLY: We know $c = t.\mathit{apply}(), \Theta_2 = \Theta_{11} + \Theta_{12} + ... + \Theta_{1n}, t.\mathit{actions} = a_1 + a_2 + ... + a_n, \cdot; \Theta_1 \vdash e_j : \tau_j$ for $j = 1, ..., m$, $\mathcal{V}(t) = (S_1...S_n, e_1...e_m)$ and $\texttt{Restrict}\Theta_1 S_i \vdash a_i : \bar{\tau}_i \to \Theta_{1i}$. We want to construc $\Theta'_2 < \Theta_2$ such that $\Gamma \vdash t.\mathit{apply}() : \Theta'_1 \Rrightarrow \Theta'_2$. By repeated application of Lemma 25, $\texttt{Restrict}\ \Theta'_1\ S_i < \texttt{Restrict}\ \Theta_1\ S_i$. For every i apply Lemma 29 which gives us $\Gamma; \texttt{Restrict}\ \Theta'_1\ S_i \vdash a : \bar{\tau} \to \Theta_{1_i}$ and $\Theta'_{1i} < \Theta_{1i}$. Let $\Theta'_2 = \sum_i \Theta'_{1i}$. (2) follows by T-APPLY. To show (1), i.e., $\Theta'_2 = \sum_i \Theta'_{1i} < \sum_i \Theta_{1i} = \Theta_2$. We know $\Theta'_{1i} < \Theta_{1i}$ for all i. The result follows by repeated application of Lemma 28. ◀

▶ **Theorem 32** (Preservation). *If* $\Gamma \vdash c : \Theta_1 \Rrightarrow \Theta_2$ *and* $\langle I, O, H, c\rangle \to \langle I', O', H', c'\rangle$*, where* $H \models \Theta_1$*, then* $\exists \Theta'_1, \Theta'_2.\ \Gamma \vdash c : \Theta'_1 \Rrightarrow \Theta'_2$ *where* $H' \models \Theta'_1$ *and* $\Theta'_2 < \Theta_2$.

Proof. By induction on a derivation of $\Gamma \vdash c : \Theta_1 \Rrightarrow \Theta_2$, with a case analysis on the last rule used.

Case T-SKIP: $c = \mathit{skip}$ and $\Theta_2 = \Theta_1$

Vauously holds as there is no c' such that $\langle I, O, H, c\rangle \to \langle I', O', H', c'\rangle$.

Case T-EXTR: $c = \mathit{extract}(h)$ and $\Theta_2 = \Theta_1 \cdot h$

The only evaluation rule that applies to c is E-EXTR, so we also have $c' = \mathit{skup}$ and $\mathcal{HT}(h) = \eta$ and $H' = H[h \mapsto v]$ where $\mathit{deserialize}_\eta = (v, I')$. Let $\Theta'_1 = \Theta'_2 = \Theta_2$. We have $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ by by T-SKIP, we have $H' \models \Theta'_2$ by Lemma 20, and we have $\Theta'_2 < \Theta_2$ by reflexivity.

Case T-EMIT: $c = \mathit{emit}(h)$ and $\Theta_2 = \Theta_1$.

There are two evaluation rules that apply to c, E-EMIT and E-EMITINVALID. In either case, $c' = \mathit{skip}$ and $H' = H$. Let $\Theta'_1 = \Theta'_2 = \Theta_1$. We have $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ by T-SKIP, we have $H' \models \Theta'_1$ by assumption, and we have $\Theta'_2 < \Theta_2$ by reflexivity.

Case T-SEQ: $c = c_1; c_2$ and $\Gamma \vdash c_1 : \Theta_1 \Rrightarrow \Theta_{12}$ and $\Gamma \vdash c_2 : \Theta_{12} \Rrightarrow \Theta_2$

There are two evaluation rules that apply to c, E-SEQ1 and E-SEQ.

Subcase E-SEQ: $c' = c_2$ and $H' = H$

By inversion of $\Gamma \vdash c_1 : \Theta_1 \Rrightarrow \Theta_{12}$ we have $\Theta_{12} = \Theta_1$. Let $\Theta'_1 = \Theta_1$ and $\Theta'_2 = \Theta_2$. We have $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ by assumption, we have $H \models \Theta'_1$ also by assumption, and $\Theta'_2 < \Theta_2$ by reflexivity.

Subcase E-SEQ: $c' = c_1'; c_2$ and $\langle I, O, H, c_1\rangle \rightarrow \langle I', O', H', c_1'\rangle$.

By IH we have $\Gamma \vdash c_1 : \Theta_1' \Rrightarrow \Theta_{12}'$ such that $H' \models \Theta_1'$ and $\Theta_{12}' < \Theta_{12}$. By Lemma 31 we have $\Gamma \vdash c_2 : \Theta_{12}' \Rrightarrow \Theta_2$ for some $\Theta_2' < \Theta_2$. We have $\Gamma \vdash c_1; c_2 : \Theta_1' \Rrightarrow \Theta_2'$ by T-SEQ, which finishes the case.

Case T-IF: $c = \mathit{if}\ (e)\ c_1\ \mathit{else}\ c_2$ and $\Gamma; \Theta_1 \vdash e : \mathit{Bool}$ and $\Gamma \vdash c_1 : \Theta_1 \Rrightarrow \Theta_{12}$ and $\Gamma \vdash c_2 : \Theta_1 \Rrightarrow \Theta_{22}$ and $\Theta_2 = \Theta_{12} + \Theta_{22}$.

There are two evaluation rules that apply to c, E-IF, E-IFTRUE, and E-IFFALSE.

Subcase E-IF: $c' = \mathit{if}\ (e')\ c_1\ \mathit{else}\ c_2$ and $H' = H$

Let $\Theta_1' = \Theta_1$ and $\Theta_2' = \Theta_2$. We have $\Gamma \vdash \mathit{if}\ e\ c_1\ c_2 : \Theta_1' \Rrightarrow \Theta_2'$ by T-IF, we have $H \models \Theta_1$ by assumption, and we have $\Theta_2 < \Theta_2'$ by reflexivity.

Subcase E-IFTRUE: $c' = c_1$ and $H' = H$.

Let $\Theta_1' = \Theta_1$ and $\Theta_2' = \Theta_{12}$. We have $\Gamma \vdash c' : \Theta_1' \Rrightarrow \Theta_2'$ by assumption, we have $H \models \Theta_1'$ also by assumption, and we have $\Theta_2' < \Theta_2$ by the definition of $<$ and the semantics of types.

Subcase E-IFTRUE: $c' = c_2$ and $H' = H$.

Symmetric to the previous case.

Case T-IFVALID: $c = \mathit{valid}(h)\ c_1\ \mathit{else}\ c_2$ and $\Gamma \vdash c_1 : \texttt{Restrict}\ \Theta_1\ h \Rrightarrow \Theta_{12}$ and $\Gamma \vdash c_2 : \texttt{NegRestrict}\ \Theta_1\ h \Rrightarrow \Theta_{22}$ and $\Theta_2 = \Theta_{12} + \Theta_{22}$.

There are two evaluation rules that apply to c, E-IFVALIDTRUE and E-IFVALIDFALSE

Subcase E-IFVALIDTRUE: $c' = c_1$ and $h \in \mathit{dom}(H)$ and $H' = H$.

Let $\Theta_1' = \texttt{Restrict}\ Theta_1\ h$ and $\Theta_2' = \Theta_{12}$. We have $\Gamma \vdash c' : \Theta_1' \Rrightarrow \Theta_2'$ by assumption, we have $H \models \Theta_1'$ by Lemma 17, and we have $\Theta_2' < \Theta_2$ by the definition of $<$ and semantics of types.

Subcase E-IFVALIDFALSE: $c' = c_2$ and $h \notin \mathit{dom}(H)$ and $H' = H$.

Symmetric to the previous case.

Case T-APPLY: $c = t.\mathit{apply}()$ and $\mathcal{CV}(t) = (\bar{S}, \bar{e})$ and $t.\mathit{actions} = \bar{a}$ and $\cdot; \Theta \vdash e_i : \tau_i$ for $e_i \in \bar{e}$ and $\texttt{Restrict}\ \Theta_1\ S_i \vdash a_i : \bar{\tau}_i \rightarrow \Theta_i'$ for $a_i \in a$ and $\Theta_2 = \sum(\Theta_i')$

There is only one evaluation rule that applies to c, E-APPLY. It follows that $\mathcal{CA}(H, t) = (a_i, \bar{v})$, and $c' = c_i[\bar{v}/\bar{x}]$ where $\mathcal{A}(a_i) = \lambda \bar{x}.\ c_i$. Next, inverting `T-Action`, we have $\Gamma, \bar{x} : \bar{\tau}_i; \vdash c_i : \texttt{Restrict}\ \Theta\ S_i \Rrightarrow \Theta_i'$. By Proposition 14, we have $\cdot; \cdot \vdash \bar{v} : \bar{\tau}_i$. Hence, by the substitution lemma, we have $\Gamma \vdash c_i[\bar{w}/\bar{x}] : \texttt{Restrict}\ \Theta\ S_i \Rrightarrow \Theta_i'$. Let $\Theta_1' = \texttt{Restrict}\ \Theta\ S_i$ and $\Theta_2' = \Theta_i'$. We have already shown that $\Gamma \vdash c' : \Theta_1' \Rrightarrow \Theta_2'$, we have that $H' \models \Theta_1'$ by Proposition 15, and we have $\Theta_2' < \Theta_2$ by the definition of $<$ and the semantics of union types.

Case T-ADD: $c = \mathit{add}(h)$ and $\Theta_2 = \Theta_1 \cdot h$

There are two evaluation rules that apply to c, E-ADD and E-ADDVALID.

Subcase E-ADD: $c' = \mathit{skip}$ and $\mathcal{HT}(h) = \eta$ and $\mathit{init}_\eta = v$ and $H' = H[h \mapsto v]$

Let $\Theta_1' = \Theta_2' = \Theta_2$. We have $\Gamma \vdash c' : \Theta_1' \Rrightarrow \Theta_2'$ by T-SKIP, we have $H' \models \Theta_1'$ by Lemma 20, and we have $\Theta_2' < Theta_2$ by reflexivity.

Subcase E-ADDVALID: $c' = \mathit{skip}$ and $H' = H$

Let $\Theta_1' = \Theta_2' = \Theta_2$. We have $\Gamma \vdash c' : \Theta_1' \Rrightarrow \Theta_2'$ by T-SKIP, we have $H' \models \Theta_1'$ by Lemma 22 and 20 since $\mathit{dom}(H') = \mathit{dom}(H[h \mapsto v])$ for any v, and we have $\Theta_2' < \Theta_2$ by reflexivity.

Case T-Rem: $c = \mathit{remove}(h)$ and $\Theta_2 = \mathtt{Remove}\ \Theta_1\ h$

There is only one evaluation rule that applies to c, E-Rem, so we have $c' = \mathit{skip}$ and $H' = H \setminus h$. Let $\Theta_1' = \Theta_2' = \mathtt{Remove}\ \Theta\ h$. We have $\Gamma \vdash c' : \Theta_1' \Rrightarrow \Theta'2$ by T-Skip, we have $H' \models \Theta_1'$ by Lemma 21, and we have $\Theta_2' < \Theta_2$ by reflexivity.

Case T-Mod: $c = h.f = e$ and $\mathtt{Includes}\ \Theta_1\ h$ and $\mathcal{HT}(h, f) = \tau_i$ and $\cdot; \Theta_1 \vdash e : \tau_i$ and $\Theta_2 = \Theta_1$

There are two evaluation rules that applies to c, E-Mod1 and E-Mod.

Subcase E-Mod1: $c' = h.f = e'$ and $e \rightarrow e'$ and $H' = H$

By preservation for expressions we have $\cdot; \Theta_1 \vdash e' : \tau_i$. Let $\Theta_1' = \Theta_2' = \Theta_1$. W We have $\Gamma \vdash c' : \Theta_1' \Rrightarrow \Theta_2'$ by T-Mod, we have $H' \models \Theta_1'$ by assumption, and we have $\Theta_2' < \Theta_2$ by reflexivity.

Subcase E-Mod: $c' = \mathit{skip}$ and $\mathit{dom}(H') = \mathit{dom}(H)$

Let $\Theta_1' = \Theta_2' = \Theta_1$. We have $\Gamma \vdash c' : \Theta_1' \Rrightarrow \Theta_2'$ by T-Skip, we have $H' \models \Theta_1'$ by Lemma 22, and we have $\Theta_2' < \Theta_2$ by reflexivity.

Case T-Zero: $\mathtt{Empty}\ \Theta_1$

By Lemma 11, we have $\mathit{dom}(H) \in [\![\Theta_1]\!]$. By Lemma 7, we have $[\![\Theta_1]\!] = \{\}$, which is a contradiction. ◀