Dependently-Typed Data Plane Programming

MATTHIAS EICHHOLZ, Technical University of Darmstadt, Germany
ERIC HAYDEN CAMPBELL, Cornell University, USA
MATTHIAS KREBS, Technical University of Darmstadt, Germany
NATE FOSTER, Cornell University, USA
MIRA MEZINI, Technical University of Darmstadt, Germany

Programming languages like P4 enable specifying the behavior of network data planes in software. However, with increasingly powerful and complex applications running in the network, the risk of faults also increases. Hence, there is growing recognition of the need for methods and tools to statically verify the correctness of P4 code, especially as the language lacks basic safety guarantees. Type systems are a lightweight and compositional way to establish program properties, but there is a significant gap between the kinds of properties that can be proved using simple type systems (e.g., SafeP4 [Eichholz et al. 2019]) and those that can be obtained using full-blown verification tools (e.g., `p4v` [Liu et al. 2018]). In this paper, we close this gap by developing Π4, a dependently-typed version of P4 based on decidable refinements. We motivate the design of Π4, prove the soundness of its type system, develop an SMT-based implementation, and present case studies that illustrate its applicability to a variety of data plane programs.

CCS Concepts: • **Software and its engineering → Formal language definitions**; • **Networks → Programming interfaces**.

Additional Key Words and Phrases: Software-Defined Networking, P4, Dependent Types

ACM Reference Format:
Matthias Eichholz, Eric Hayden Campbell, Matthias Krebs, Nate Foster, and Mira Mezini. 2022. Dependently-Typed Data Plane Programming. *Proc. ACM Program. Lang.* 6, POPL, Article 40 (January 2022), 28 pages. https://doi.org/10.1145/3498701

1 INTRODUCTION

Computer networks are becoming increasingly programmable as languages like P4 [Bosshart et al. 2014] make it possible to specify the behavior of data planes in software. With the increased availability of programmable devices, a number of powerful and complex applications having become viable, ranging from novel network protocols to full-blown in-network computation (e.g., executing application-level storage queries using network devices [Jin et al. 2017]). But as the complexity of these applications increases, so does the risk of faults, especially as P4's main abstraction for representing packet data—namely header types—lacks basic safety guarantees. Experience with a growing number of programs has shown the risks of the unsafe approach, which often leads to subtle software bugs [Eichholz et al. 2019; Liu et al. 2018]. This is clearly unacceptable,

Authors' addresses: Matthias Eichholz, Technical University of Darmstadt, Germany, eichholz@cs.tu-darmstadt.de; Eric Hayden Campbell, Cornell University, USA, ehc86@cornell.edu; Matthias Krebs, Technical University of Darmstadt, Germany, krebs@cs.tu-darmstadt.de; Nate Foster, Cornell University, USA, jnfoster@cs.cornell.edu; Mira Mezini, Technical University of Darmstadt, Germany, mezini@cs.tu-darmstadt.de.

2475-1421/2022/1-ART40
https://doi.org/10.1145/3498701

given the crucial role that networks play in nearly all modern systems. Hence, we need methods and tools to statically verify the correctness of data plane programs.

Today, most data plane verification tools [Dumitrescu et al. 2020; Liu et al. 2018; Stoenescu et al. 2018] are monolithic in nature. For example, `p4v` [Liu et al. 2018], which is based on software model checking, operates on whole programs. But while monolithic approaches have certain advantages—e.g., they minimize the need for programmer annotations—they also have downsides. The most fundamental limitation is the inherent tension with modular design—it is difficult to accommodate an "open-world" model, in which third-party components are plugged into existing programs. For instance, an equipment vendor might want to provide a "base" program that implements standard packet-processing functionality like Ethernet switching, and allow customers to add custom functionality of their own design [Baldi 2019; Gao et al. 2020; Soni et al. [n.d.]]. Composable approaches to data plane programming require compositional reasoning methods [Beckett and Mahajan 2020].

Type systems are a lightweight and compositional way to establish program properties—i.e., the types for individual components document assumptions about the components they rely upon as well as the guarantees they offer. However, somewhat surprisingly, types have rarely been applied in the realm of network programming, and the few exceptions [Eichholz et al. 2019; Ennals et al. 2004; Muthukrishnan et al. 2010] are simple type systems with limited expressive power. For example, SafeP4 [Eichholz et al. 2019] uses regular types [Castagna et al. 2014; Gapeyev and Pierce 2003; Hosoya and Pierce 2003] and path-sensitive occurrence typing [Tobin-Hochstadt and Felleisen 2010] to reason about basic safety properties, but it cannot capture richer program properties (e.g., whether the IPv4 and IPv6 headers are only ever accessed on mutually exclusive execution paths), or track the values of individual fields (e.g., whether EtherType equals to `0x0800`, which indicates an IPv4 packet, or to `0x86DD`, which indicates an IPv6 packet). The inability of SafeP4 to reason about the values being manipulated by the program significantly limits its expressiveness. In general, there is a significant gap between the kinds of properties that can be checked using type systems like SafeP4 and full-blown verification tools like `p4v`.

Thus, it is natural to ask whether we can design a compositional type system that has the same expressive power as data plane verification tools. This paper answers this question in the affirmative, by presenting Π4—a dependently-typed version of P4. Π4 fits with the trend of recently proposed dependently-typed languages [Condit et al. 2007; Rondon et al. 2008; Vazou et al. 2014; Xi and Pfenning 1999] that are blurring the line between type checking and theorem proving. For instance, Liquid Haskell [Rondon et al. 2008; Vazou et al. 2014] allows programmers to smoothly shift from properties that can be checked with traditional typing disciplines to more sophisticated ones that go beyond simple syntactic checks. Under the hood, Liquid Haskell uses an SMT solver to automatically discharge the logical proof obligations generated during type checking.

Yet, thus far, the dependently-typed approach has not been explored for network programming. In this paper, we demonstrate that data plane programs are a "killer application" for dependent types. On the one hand, they clearly need precise types, as most programs rely on intricate packet formats (e.g., so-called "type-length-value" encodings, where the first few bits determine the type, length, and structure of the bits that follow). On the other hand, data plane programs are fundamentally simple (e.g., P4 does not support pointers or loops) and lack the kinds of complex features that often make precise type systems complicated to design and implement.

Our main contribution lies in exploring and addressing the subtle challenges that arise in developing a dependent type system for the P4 programming language, including balancing the tradeoffs between expressiveness and decidability. Π4 features a combination of refinement types, dependent function types, union types, and explicit substitutions. This combination is key to retain precision during type checking—e.g., we can compute exact types for conditionals, thereby having

access to an accurate type at any program point. Moreover, our design enables precise typing in the presence of domain-specific features that combine packet serialization and deserialization operations with imperative control-flow. To this end, Π4 combines a dependent sum type with a novel "chomp" operation that computes the type that remains after extracting bits from a packet buffer. We formally prove that Π4's type system is sound via standard progress and preservation theorems.

The chomp operator is reminiscent of regular expression derivatives [Brzozowski 1964]. To the best of our knowledge, with a few notable exceptions (e.g., work by McBride [McBride 2001]) derivative-like operations have not been extensively studied the context of dependent type systems. Thus, beyond providing an elegant solution to a practical problem, we believe that Π4's innovative combination of dependent types with regular types and the possibility to compute derivatives of types is of general theoretical interest and may be useful in other domains—e.g., one potential direction is verified serializers and deserializers [Delaware et al. 2019; Ramananandro et al. 2019].

We have built a prototype implementation of Π4 in OCaml and the Z3 SMT solver. The type checker determines whether a Π4 program has a given type by checking the validity of a series of logical formulae using an SMT solver. We encode types into the effectively propositional fragment of first-order logic over fixed-width bit vectors, which facilitates automating subtyping and equivalence checks. We prove (cf. Theorem 4.1) that this logical fragment is sufficient for checking our types under the assumption that the types written by the programmer denote finite sets. We believe this is a reasonable assumption, because networks enforce a *maximum transmission unit* (MTU) (i.e., a bound on the size of packets constraining the maximum number of bits that network switches can receive or transmit[1]) which bounds the size of bit vectors we need to consider in the encoding. In the presence of an MTU, our types denote finite sets, which can be enumerated to decide the key judgments (i.e., subtyping, size constraints, and inclusion checks).

Using our Π4 prototype, we develop several case studies, demonstrating that Π4 is capable of expressing and (modularly) reasoning about properties from the literature ranging from basic safety to intricate invariants: parser round-tripping, protocol conformance, determined forwarding, etc. We selected properties that are also studied by recent data plane verification tools like `p4v` or Vera, which indicates that Π4 is capable of covering properties of interest to the networking community. However, we leave a careful study of the practical utility of Π4 (e.g., with larger examples and user studies) to future work.

Overall, the contributions of our work are as follows:

- Section 2 motivates dependently-typed data plane programming.
- Section 3 presents Π4, a dependently-typed core of P4, featuring a combination expressive types for describing structures (regular types as well as decidable refinement and dependent function types) combined with a bit-by-bit "chomp" operator.
- Section 3 develops a semantic proof of soundness for Π4's type system.
- Section 4 defines a decidable algorithmic type system for Π4.
- Section 5 and Section 6 discusses case studies using Π4's type system to check realistic program properties.

2 BACKGROUND

Most networks are based on a division of labor between two components: the control plane and the data plane. The control plane, usually implemented in software, is responsible for performing tasks such as learning the topology, computing network-wide forwarding paths, managing shared

[1]The MTU is often set to 1500 bytes.

```
1  parser P(packet_in p, out hdrs h) {
2    state start {
3      p.extract(h.ethernet);
4      transition select(
5        h.ethernet.etherType) {
6          0x0800: parse_ipv4;
7          default: accept;
8      }
9    }
10   state parse_ipv4 {
11     p.extract(h.ipv4);
12     transition accept;
13   }
14 }
15 control Ingress(inout hdrs h) {
16   apply {
17     if (h.ipv4.src == 10.10.10.10) {
18       drop();
19     }
20   }
21 }
```

Fig. 1. Unsafe P4 program: IPv4 is not guaranteed to be valid in the ingress.

resources like bandwidth, and enforcing security policies. The control plane can either be realized using distributed routing protocols (e.g., in a traditional network), or as a logically-centralized program (e.g., in a software-defined network). The data plane, often implemented in hardware or with highly-optimized software, is responsible for forwarding packets. It parses packets into collections of headers, performs lookups in routing tables, filters traffic using access control lists, applies queueing policies, and ultimately drops, copies, or forwards the packet to the next device.

P4 is a domain-specific programming language for specifying the behavior of network data planes. It is designed to be used with programmable devices such as PISA switches [Bosshart et al. 2013], FPGAs [Ibanez et al. 2019; Wang et al. 2017], or software devices (e.g., eBPF [Høiland-Jørgensen et al. 2018]). The language is based on a pipeline abstraction: given an input packet it executes a sequence of blocks of code, one per pipeline component, to produce the outputs. Each pipeline component is either a "parser," which consists of a state machine that maps binary packets into typed representations, or a "control," which consists of a sequence of imperative commands. To interface with the control plane, P4 programs may contain "match-action" tables, which contain dynamically reconfigurable entries, each corresponding to a fixed block of code.

Unfortunately, P4 is an unsafe language that does not prevent programmers from writing programs such as the one shown in Figure 1. The program begins by parsing the Ethernet header. Then, if the Ethernet header contains the appropriate EtherType (`0x0800`), it also parses the IPv4 header. Next, in the ingress control, if the IPv4 source address matches a specified address, the packet is marked to be dropped. However, there is no guarantee that the IPv4 header will be a well-defined value—e.g., if the EtherType is `0x08DD`, indicating an IPv6 packet, the value produced by reading the IPv4 source address (Line 17) will be undefined, making the behavior of the program non-deterministic and possibly different than what the programmer intended.

SafeP4 addresses the lack of basic safety guarantees for P4 using a simple type system [Eichholz et al. 2019]. Specifically, its type system keeps track of the set of valid header instances at each statement. For example, starting from the empty heap with no header instances valid, SafeP4 computes the type of the program above to be `ether.ipv4 + ether` after parsing. This type reflects that on the first program path both Ethernet and IPv4 are valid, but on the second program path only Ethernet is valid. Thus, when type checking the if-condition in the ingress code, the type checker knows that IPv4 may be invalid and rejects the program. A simple fix (shown in Figure 2a) that makes the program safe is to add an explicit validity check before accessing the IPv4 header. Because it is aware of the semantics of the `isValid` command, the SafeP4 type checker computes the type before the access to be `ether.ipv4`—i.e., IPv4 is guaranteed to be valid.

```
control Ingress(inout headers h) {
  apply {
    if(h.ipv4.isValid()) {
      if(h.ipv4.src == 10.10.10.10) {
        drop();
}}}}
```

(a) Explicit validity check

```
control Ingress(inout headers h) {
  apply {
    if(h.ether.etherType == 0x0800) {
      if(h.ipv4.src == 10.10.10.10) {
        drop();
}}}}
```

(b) Implicit validity check

Fig. 2. Safe implementation of ingress

In practice, however, relying on explicit validity checks is not sufficient. For example, consider the code shown in Figure 2b. Recall that, given the parser above, the IPv4 header will be present if the EtherType is `0x0800`. Hence, the ingress control can be safely executed. Yet, SafeP4's type checker rejects the program because the type system is not expressive enough to capture the dependency between the EtherType value and IPv4's validity.

To address this problem, Π4 employs a dependent type system [Xi and Pfenning 1999], in which we can compute a precise type for the program after parsing:

$$(x : \{y : \epsilon \mid |\mathsf{y.pkt_{in}}| > 272\}) \longrightarrow \left(\begin{array}{l} \Sigma y : \mathtt{ether}.\{z : \mathtt{ipv4} \mid y.\mathtt{ether.etherType} == 0x0800\} \\ + \{z : \mathtt{ether} \mid z.\mathtt{ether.etherType} \neq 0x0800\}) \end{array} \right)$$

Intuitively, this type says that, starting with the empty heap ($y : \epsilon$) and a packet buffer that has at least enough bits to extract both the Ethernet and the IPv4 header ($|y.pkt_{in}| > 272$), the parser ends in one of two possible states: (1) either both Ethernet and IPv4 are valid (Σy: `ether`.$\{z$: `ipv4` | ...$\}$), if the EtherType is equal to `0x0800` (note how z : `ipv4` is conditioned by y.`ether.etherType` == $0x0800$), or (2) just Ethernet is valid, if EtherType is not equal to `0x0800`. When checking the ingress control, the type checker uses the predicate `ether.etherType` == `0x0800` from the conditional to derive the set of valid header instances, which, in this case, includes IPv4. Thus, accessing the IPv4 source address is safe and the program correctly passes the type checker.

While the output type is admittedly notationally heavy—a common feature in precise type systems—note that the programmer is not forced to write down the most precise type! Π4 only requires the annotated type to be sufficiently precise to capture basic safety guarantees and other desired invariants. For example, in a program where only the Ethernet header needs to be valid at the end of the parser, they could use the type $(x : \{y : \epsilon \mid |\mathsf{y.pkt_{in}}| > 272\}) \longrightarrow \mathtt{ether}\sim$, which indicates that, at the end of the parser, at least Ethernet is valid (and possibly others, too).

This example illustrates how Π4's type system statically checks intricate safety properties with high precision. Sections 5 and 6 present more case studies showing how Π4's type system can be used to check practical properties of interest.

3 DEPENDENT TYPES FOR P4 (Π4)

This section introduces the design, syntax, and semantics of Π4, a core calculus modeled after P4 and equipped with dependent types.

3.1 Design of Π4

Π4 focuses on the aspects of the P4 programming language that benefit from dependent types, (e.g., parsing, deparsing, validity, and control flow) and omits features that add clutter (e.g., externs, registers, checksums, hashing, and packages). Following `p4v` [Liu et al. 2018], we do not explicitly model match-action tables and instead use ghost state and conditionals to encode them (see

Section 6). Consequently, Π4 is a loop-free[2] imperative language with a few domain-specific primitive commands: *extract*(ι), *remit*(ι), *add*(ι), and *reset*.

In P4, the `emit`(ι) primitive serializes a header instance ι into a bitstring and prepends it to the outgoing packet payload, *only if* ι *is valid*, otherwise it does nothing. To simplify typing rules and semantics, Π4 provides the primitive *remit*(ι), which *really* emits ι if it is valid, and otherwise gets stuck. Hence, `emit`(ι) can be encoded as syntactic sugar: *if*(ι.*valid*) *then remit*(ι) *else skip*. Another superficial difference is that we model header field accesses as direct bit-slices into the instance (to avoid another layer of indirection in our semantics)—i.e., `eth.srcAddr` is written `eth`$[48:96]$.

More substantially, Π4 diverges from P4 in the way it handles instance validity. A header instance is valid in two cases: (i) if it has been extracted from the packet (which automatically populates the instance with the appropriate bits) or (ii) if its validity bit has manually been set using the `setValid()` method (which does nothing to the instance itself). In P4, reads to uninitialized variables produce undefined values, so a common programming practice is to follow a call to `setValid()` with a sequence of assignments to the header fields—thereby avoiding undefined reads. In Π4, rather than forcing the programmer to manually write default values, the *add*(ι) command sets ι.*valid* to true, and sets ι a pre-determined default value (say 0).[3] If required, P4's behavior could be encoded using an extra 1-bit header to independently track the validity of the instance and initialization of its fields.

We also introduce a new primitive, *reset*, which models the behavior of P4 between pipeline stages. In many switch architectures [Bosshart et al. 2013], packets are deparsed and then reparsed between pipelines—e.g., after `ingress` and before `egress`. The *reset* command encodes the behavior of the inner step: it combines the deparsed bits with the packet's unparsed payload and passes it along as the input to the next stage. The *reset* command would also be useful to reason about invariants across multiple switch programs, although we don't explore that use in this paper.

Finally, in designing Π4, our primary goal is to enable data plane programmers to make use of dependent types to verify useful program properties in a compositional way and without having to write manual proofs. To enable modular reasoning, we need a way to annotate (and modularly check) programs with types. We annotate a program c with a type σ using an ascription operator: c *as* σ. The ascription has no effect on the runtime behavior of the code (i.e., c *as* σ always just steps to c). It does, however, indicate a program point where type checking should occur. Hence, we can independently typecheck c at type σ and then use σ when checking the rest of the program.

Intuition for Π4*'s type system.* Next, we give an intuition for Π4's types. A command c is always assigned a dependent function type $(x : \tau_1) \to \tau_2$, where x may occur in τ_2. This design allows us to relate the input and output values of commands expressed in the *heap types* τ_1 and τ_2. For example, we may want to ensure that the Ethernet header has the same value after being deparsed, reset, and then reparsed. To express equations like this, we use refinement types $\{y : \tau \mid \varphi\}$, where φ is a formula in the logic of variable-width bit vectors with concatenation and length operators. In this example, we could say that the Ethernet header is unchanged by using the output type $\{y : \tau_2 \mid x.\texttt{eth} = y.\texttt{eth}\}$.

We also often need to reference intermediate formulae, so we introduce substitution types, written $\tau_2[x \mapsto \tau_1]$, where x may occur in τ_2 but not in τ_1. In such a type, τ_1 may represent the type at any earlier point in the program. Π4 also supports fine-grained path-dependent reasoning via union types $(\tau_1 + \tau_2)$. It is also convenient to have trivial ($\top$) and absurd ($\varnothing$) types.

[2]P4 allows loops within parsers, but because programs are restricted to finite state, the language specification allows implementations to unroll loops.

[3]The difference here is similar to the difference between C's `malloc` (analogous to P4's semantics) and `calloc` (analogous to Π4's semantics).

$$
\begin{array}{lll}
\tau & ::= \varnothing \mid \top \mid \Sigma x\colon \tau.\tau \mid \tau + \tau \mid \{x : \tau \mid \varphi\} \mid \tau[x \mapsto \tau] & \text{(heap types)} \\
\sigma & ::= \mathbb{N} \mid \mathbb{B} \mid \mathsf{BV} \mid (x : \tau) \to \tau & \text{(base types)} \\
\varphi & ::= e = e \mid e > e \mid \varphi \wedge \varphi \mid \neg\varphi \mid x.\iota.\mathit{valid} \mid \mathsf{true} \mid \mathsf{false} & \text{(formulae)} \\
e & ::= n \mid bv \mid |x.p| \mid e + e \mid e@e \mid x.p \mid x.p[l\!:\!r] \mid x.\iota[l\!:\!r] & \text{(expressions)} \\
bv & ::= \langle\rangle \mid 0 :: bv \mid 1 :: bv \mid \mathsf{b_n} :: bv & \text{(bit vectors)} \\
p & ::= \mathit{pkt}_{in} \mid \mathit{pkt}_{out} & \text{(packets)} \\
c & ::= \mathit{extract}(\iota) \mid \mathit{if}(\varphi)\ c\ \mathit{else}\ c \mid c; c \mid \iota.f := e \mid \mathit{remit}(\iota) & \text{(commands)} \\
 & \quad \mathit{skip} \mid \mathit{reset} \mid \mathit{add}(\iota) \mid c\ \mathit{as}\ (x : \tau) \to \tau & \\
d & ::= \eta\ \{\overline{f : \mathsf{BV}}\} \mid \iota \mapsto \eta & \text{(declarations)} \\
P & ::= (\overline{d}, c) & \text{(programs)}
\end{array}
$$

Fig. 3. Syntax of Π4

Finally, the design of our type system is also informed by the need to model parsing operations. Specifically, we must ensure that the refinements on the input type and on the output type remain consistent after bits have been shuffled around by a command. For example, given an input type $\{x : \top \mid x.\mathit{pkt}_{in}[0:8] == 4 \wedge |x.\mathit{pkt}_{in}| > |\mathtt{ipv4}|\}$, where $x.\mathit{pkt}_{in}$ represents the incoming packet, and the command $\mathit{extract}(\mathtt{ipv4})$, the output type should reflect that the `ipv4` header instance is now valid, that $\mathtt{ipv4}[0:8]$ is 4, and that $x.\mathit{pkt}_{in}$ may have no more bits remaining. Π4 accomplishes this using two key mechanisms: (1) a dependent sum type $\Sigma x : \tau_1.\tau_2$ that computes the disjoint union of the valid instances in τ_1 and τ_2 and concatenates the incoming and outgoing packets (Section 3.3), and (2) a refinement transformer, chomp, that manipulates input refinements to be consistent with the extraction operation (see Section 3.5).

3.2 Syntax

Figure 3 shows the syntax of Π4. Boolean formulae φ include literals, equality (=) and validity of instances ($x.\iota.\mathit{valid}$), conjunction ($\wedge$), and negation ($\neg$). Expressions e include naturals, bit vectors, packet length ($|x.p|$), addition (+), concatenation (@), packet access ($x.p$) and slices of packets ($x.p[l\!:\!r]$) and instances ($x.\iota[l\!:\!r]$).

To ease the notation, we write $x.\iota[l]$ instead of $x.\iota[l\!:\!l+1]$ for bit-wise access, $x.\iota.f$ instead of $x.\iota[l\!:\!r]$ for ranges matching header instance fields, $x.\iota$ instead of $x.\iota[0\!:\!\mathrm{sizeof}(\iota)]$, $x.\iota[n\!:\!]$ for the remaining bits of $x.\iota$ starting from bit $n+1$, and similarly for the corresponding formulae involving packet variables $x.p$. We use a list-like encoding of bit vectors. A bit vector is either the empty bit vector ($\langle\rangle$) or a concatenation of bits. We assume that bit variables $\mathsf{b_n}$ are not part of the surface syntax and are only used internally. For singleton bit vectors, we write $\langle b\rangle$ instead of $b :: \langle\rangle$.

We write $\epsilon \triangleq \{x : \top \mid \bigwedge_{\iota \in \mathit{dom}(\mathcal{HT})} \neg x.\iota.\mathit{valid}\}$ for the type denoting the empty heap on which no header instances are valid, $\iota \triangleq \{x : \top \mid x.\iota.\mathit{valid} \wedge \bigwedge_{\iota' \in \mathit{dom}(\mathcal{HT}), \iota' \neq \iota} \neg x.\iota'.\mathit{valid}\}$ for the type denoting the heap exclusively containing instance ι, and $\iota^{\sim} \triangleq \{x : \top \mid x.\iota.\mathit{valid}\}$ for the type denoting the heap on which at least instance ι is guaranteed to be valid.

For formulae, we write $x \equiv y$ (respectively $x \equiv_\iota y$) as syntactic sugar for the boolean predicates capturing strict equality (respectively instance equality) between the heaps bound to x and y. Strict equality requires that both the input and output packets are equivalent as well as all instances contained in the heap—i.e., $x.\mathit{pkt}_{in} = y.\mathit{pkt}_{in} \wedge x.\mathit{pkt}_{out} = y.\mathit{pkt}_{out} \wedge \bigwedge_{\iota \in \mathit{dom}(\mathcal{HT})} x.\iota = y.\iota$, while instance equality only requires that the instances are equivalent in both heaps. We write $x.\iota.\mathit{valid} = y.\iota.\mathit{valid}$ as syntactic sugar for $(x.\iota.\mathit{valid} \wedge y.\iota.\mathit{valid}) \vee (\neg x.\iota.\mathit{valid} \wedge \neg y.\iota.\mathit{valid})$. We use standard encodings using negation and conjunction for logical connectives like $\vee$ or $\Rightarrow$.

$$
\begin{aligned}
[\![\tau]\!]_{\mathcal{E}} &\subseteq \mathcal{H} \\
[\![\varnothing]\!]_{\mathcal{E}} &= \{\} \\
[\![\top]\!]_{\mathcal{E}} &= \mathcal{H} \\
[\![\tau_1 + \tau_2]\!]_{\mathcal{E}} &= [\![\tau_1]\!]_{\mathcal{E}} \cup [\![\tau_2]\!]_{\mathcal{E}} \\
[\![\Sigma x\colon \tau_1.\tau_2]\!]_{\mathcal{E}} &= \{h_1 +\!\!+ h_2 \mid h_1 \in [\![\tau_1]\!]_{\mathcal{E}} \wedge h_2 \in [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h_1]}\} \\
[\![\tau_1[x \mapsto \tau_2]]\!]_{\mathcal{E}} &= \{h \mid h_2 \in [\![\tau_2]\!]_{\mathcal{E}} \wedge h \in [\![\tau_1]\!]_{\mathcal{E}[x \mapsto h_2]}\} \\
[\![\{x : \tau \mid \varphi\}]\!]_{\mathcal{E}} &= \{h \mid h \in [\![\tau]\!]_{\mathcal{E}} \wedge [\![\varphi]\!]_{\mathcal{E}[x \mapsto h]} = \mathsf{true}\}
\end{aligned}
$$

Fig. 4. Semantics of heap types

We write $\overline{x}$ as a shorthand for a possibly empty sequence $x_1, ..., x_n$. A program consists of a sequence of declarations $\overline{d}$ and a command c. Declarations d include header type declarations $\eta\ \{\overline{f : \mathrm{BV}}\}$ and header instance declarations $\iota \mapsto \eta$. Header type declarations specify the format of network packet headers. They are defined in terms of a name and a sequence of field declarations, where each field is itself defined in terms of a field name and a type. We write $f : \mathrm{BV}$ to denote that field f has a bit vector type BV. With η ranging over header types, the instance declaration $\iota \mapsto \eta$ assigns the name ι to header type η. The global mapping between header instances and header types is stored in the so-called header table $\mathcal{HT}$. We assume that names of header instances and header types are drawn from disjoint sets of names and that each name is declared only once.

Π4 provides commands for parsing (*extract*), creating (*add*) and modifying ($\iota.f := t$) header instances. The *remit* command serializes a header instance into a bit sequence. The *reset* command resets the program state—in particular, the packet buffers. Commands can be sequentially composed ($c_1; c_2$), *skip* is a no-op, and the *if*-command conditionally executes one out of two commands based on the value of the boolean formula φ. We assume that formulae and expressions used in commands are implicitly prefixed with a variable named heap, but we often omit it in the surface syntax. For example, we write *if* (`ether`.*valid*) *then extract*(`ipv4`) *else skip* instead of *if* (`heap.ether`.*valid*) *then extract*(`heap.ipv4`) *else skip*. Π4 provides modular reasoning via type ascription (c *as* $(x : \tau_1) \to \tau_2$). Finally, we assume that every header referenced in a program has a corresponding instance declaration—this could be enforced statically using a simple analysis.

3.3 Type System

As shown in Figure 3, there are two categories of types, base types σ and heap types τ. Base types include natural numbers, bit vectors, booleans, and dependent function types. Heap types τ represent sets of heaps, where each element in the set describes a different program path. The goal is to capture bit-level dependencies between header instances and the incoming and outgoing packet in the type system. A heap h in the set of heaps $\mathcal{H}$ describes a possible system state, consisting of the incoming and outgoing packet and the set of valid header instances. We model heaps as maps from names to bit vectors. A heap contains two special entries pkt_{in} and pkt_{out} representing the incoming and outgoing packet buffers, as well as mappings from instance names to bit vectors for each valid header instance. The semantics of types is shown in Figure 4. Heap types are evaluated in an environment $\mathcal{E}$, which is a mapping from variable names to heaps. The environment models other heaps available in the current scope upon which the current header type may depend.

The type $\varnothing$ denotes the empty set. It is used in situations where there are unsatisfiable assumptions involving the header instances or the incoming and outgoing packet buffers.

$$[\![e]\!]_{\mathcal{E}} \in \mathsf{BV} \cup \mathbb{N}$$

$$[\![|x.p|]\!]_{\mathcal{E}} = \begin{cases} 0 & \text{if } \mathcal{E}(x)(p) = \langle\rangle \\ n & \text{if } \mathcal{E}(x)(p) = \langle b_1, ..., b_n\rangle \\ \bot & \text{otherwise} \end{cases}$$

$$[\![x.p]\!]_{\mathcal{E}} = \begin{cases} \langle b_1, ..., b_n\rangle & \text{if } \mathcal{E}(x)(p) = \langle b_1, ..., b_n\rangle \\ \bot & \text{otherwise} \end{cases}$$

$$[\![x.p[n:m]]\!]_{\mathcal{E}} = \begin{cases} \langle b_n, ..., b_{m-1}\rangle & \text{if } [\![x.p]\!]_{\mathcal{E}} = \langle b_0, ..., b_k\rangle \wedge \\ & 0 \leq n < m \leq k+1 \\ \bot & \text{otherwise} \end{cases}$$

$$[\![x.\iota[n:m]]\!]_{\mathcal{E}} = \begin{cases} \langle b_n, ..., b_{m-1}\rangle & \text{if } [\![x.\iota]\!]_{\mathcal{E}} = \langle b_0, ..., b_k\rangle \wedge \\ & 0 \leq n < m \leq k+1 \\ \bot & \text{otherwise} \end{cases}$$

Fig. 5. Selected cases of the semantics of expressions

The top type $\top$ denotes the set of all possible heaps. The choice type $\tau_1 + \tau_2$ denotes the union of the sets of heaps represented by τ_1 and τ_2. The dependent pair $\Sigma x : \tau_1.\tau_2$ denotes the concatenation of heaps from τ_1 and τ_2, where heaps described by τ_2 may depend on heaps from τ_1. The concatenation $h = h_1 +\!\!+ h_2$ of two heaps h_1 and h_2 requires that header instances contained in h_1 and h_2 are disjoint. The resulting heap contains all instances from h_1 and from h_2, with pkt_{in} and pkt_{out} the concatenation of respective bit vectors in h_1 and h_2. The explicit substitution $\tau_1[x \mapsto \tau_2]$ denotes the set of heaps obtained by evaluating τ_1 for every heap described by τ_2. Finally, the refinement type $\{x : \tau \mid e\}$ denotes the set of heaps described by τ for which the predicate e holds.

The refinement predicate φ is evaluated in the same type of environment as heap types. Formulae evaluate to a boolean value, i.e., $[\![\varphi]\!]_{\mathcal{E}} \in \mathbb{B}$. The semantics of expression equality ($e_1 = e_2$) is defined as the semantic equality between expressions e_1 and e_2. If the semantics of one of the expressions is undefined, expression equality always evaluates to false. The semantics of expression comparison ($e_1 > e_2$) is defined analogously. Instance validity $x.\iota.valid$ evaluates to true if header instance ι is contained in the heap bound to x in $\mathcal{E}$, otherwise it evaluates to false. The remaining operations have standard semantics.

The semantics of expressions is defined in Figure 5. For brevity, we omit standard cases. Expressions evaluate to either a bit vector or a natural number. For the semantic addition (+) and bit vector concatenation operator (@), we assume that as soon as evaluating one operand results in an error ($\bot$) the whole expression also evaluates to $\bot$. To evaluate the length of a packet $|x.p|$, we compute the length of the bit vector of pkt_{in} or pkt_{out} respectively in the heap bound to x in the environment. If no heap is bound to variable name x, the expression evaluates to $\bot$. The semantics of bit vectors is as expected, except for variables, which we look up from a designated location in the environment. If no binding for the bit variable exists, it evaluates to $\bot$. The semantics of bit vector concatenation is standard. A packet access $x.p$ looks up the respective entry from the heap bound to variable x in $\mathcal{E}$. A packet slice $x.p[l:r]$ is evaluated in the same way, but additionally the designated slice is obtained from the bit vector. Again, if the variable is not bound or an index is greater than the length of the bit vector, the expression evaluates to $\bot$. The semantics of instance slices $x.\iota[l:r]$ is defined similarly but the lookup occurs on header instance ι. We interpret slices as

$$\textsc{E-Extract}\quad \frac{\mathcal{HT}(\iota) = \eta \qquad deserialize_\eta(I) = (v, I')}{\langle I, O, H, extract(\iota)\rangle \longrightarrow \langle I', O, H[\iota \mapsto v], skip\rangle}$$

$$\textsc{E-Remit}\quad \frac{\iota \in dom(H) \qquad \mathcal{HT}(\iota) = \eta \qquad serialize_\eta(H(\iota)) = bv}{\langle I, O, H, remit(\iota)\rangle \longrightarrow \langle I, O :: bv, H, skip\rangle}$$

$$\textsc{E-Mod}\quad \frac{H(\iota) = r \qquad r' \triangleq \{r\ with\ f = v\}}{\langle I, O, H, \iota.f := v\rangle \longrightarrow \langle I, O, H[\iota \mapsto r'], skip\rangle}$$

$$\textsc{E-Reset}\quad \frac{I' = O@I}{\langle I, O, H, reset\rangle \longrightarrow \langle I', \langle\rangle, [], skip\rangle}$$

$$\textsc{E-Add}\quad \frac{\iota \notin dom(H) \qquad \mathcal{HT}(\iota) = \eta \qquad init_\eta = v}{\langle I, O, H, add(\iota)\rangle \longrightarrow \langle I, O, H[\iota \mapsto v], skip\rangle}$$

$$\textsc{E-Ascribe}\quad \frac{}{\langle I, O, H, c\ as\ (x : \tau_1) \longrightarrow \tau_2\rangle \longrightarrow \langle I, O, H, c\rangle}$$

Fig. 6. Small-step operational semantics of Π4

half-open intervals, where the left bound is included and the right bound is excluded. For example, given a bit vector $bv = 1010$ we have $bv[1{:}4] = 010$.

We define two semantic operations on heap types: *inclusion* and *exclusion* of instances. The first, Includes $\Gamma\ \tau\ \iota$, traverses τ and checks that instance ι is valid in every path. Semantically this says that ι is a member of every element of $[\![\tau]\!]_{\mathcal{E}}$—i.e., if $\mathcal{E} \models \Gamma$, then $\forall h \in [\![\tau]\!]_{\mathcal{E}}.\iota \in dom(h)$. The second, Excludes $\Gamma\ \tau\ \iota$, traverses τ and checks that instance ι is invalid in every path. Semantically this says that ι is no member of every element of $[\![\tau]\!]_{\mathcal{E}}$—i.e., if $\mathcal{E} \models \Gamma$, then $\forall h \in [\![\tau]\!]_{\mathcal{E}}.\iota \notin dom(h)$.

3.4 Operational semantics

The small-step operational semantics of Π4, is shown in Figure 6. It is defined in terms of a four-tuple $\langle I, O, H, c\rangle$, where I and O are the bitstrings for the incoming and outgoing packets respectively, H is a map that relates instance names to records containing the field values, and c is a command.

The $extract(\iota)$ command (E-Extract) first looks up the header type from the header table ($\mathcal{HT}$), and uses a deserialization function to copy the appropriate number of bits from the input packet into the deserialized representation of the instance v. This value is added to the map of valid header instances H. We assume there exists a deserialization function for every header instance. For example, assuming $I = 110011B$, where B is the rest of the bitstring, and $\eta = \{f : 4; g : 2\}$, then $deserialize_\eta(I) = (\{f = 1100; g = 11\}, B)$.

The $remit(\iota)$ command (rule E-Remit) requires that the header instance is valid—i.e., it is contained in H. Similar to E-Extract, we assume there is a serialization function for every heap type, which turns a record representing the instance back into a bit sequence. For example, $serialize_\eta(\{f = 1100; g = 11\}) = 11011$. The serialized bit sequence is appended to the end of the outgoing packet. Both the input packet and the set of valid headers remain unchanged.

The rule E-Mod defines the semantics of assigning a value to a header field. Assuming r is the record storing the values of the fields, an updated record r' with the modified field value is stored in H. The input and output packets remain unchanged. If the assigned expression is not a value, it is reduced first.

The rules for sequencing (E-Seq, E-Seq1) are standard. Sequences of commands evaluate from left to right—i.e., the left-hand command is reduced to skip before the right-hand command is evaluated. The evaluation rules for conditionals (E-If, E-IfTrue, E-IfFalse) are also standard. All standard rules are omitted.

The rule E-Reset defines the semantics of the command *reset*. It would be invoked between the ingress and egress pipelines, when the packet emitted by the ingress becomes the input packet for the egress. Operationally, the bits contained in the output packet are prepended to the bits of the input packet. This concatenated bit sequence serves as the new input packet. The output packet is emptied and all valid header instances are discarded.

The rule E-Add initializes a header instance if it is not already valid. The evaluation is similar to rule E-Extract, except that no bits are taken from the input bitstring. Instead we assume that there exists an initialization function $init_\eta$ for every heap type η —similar to *deserialize*—that initializes all fields of an instance to a fixed value. If an instance is already valid, this operation is a no-op. An ascribed command c *as* σ (rule E-Ascribe) evaluates to c trivially, without modifying the heap.

3.5 Typing Judgement

The typing judgement has the form $\Gamma \vdash c : (x : \tau_1) \to \tau_2$. Intuitively, type τ_1 describes the input heap and τ_2 describes the output heap obtained after the execution of c. Γ is a variable context that maps variable names to heap types and is used to capture additional dependencies of the input type. If a command typechecks in a context where x maps to τ (i.e., $\Gamma, x : \tau \vdash c : (x : \tau_1) \to \tau_2$) it means that given some heap described by type τ on which the input heap might depend, executing c on the input heap described by τ_1 will result in a heap described by τ_2.

The typing rules are presented in Figure 7. The typing rule T-Extract captures that ι must be valid after an *extract* command is executed and the input packet of type τ_1 provides enough bits for the instance ($\text{sizeof}_{pkt_{in}}(\tau) \geq n$ iff $\forall \mathcal{E}, h \in [\![\tau]\!]_{\mathcal{E}}, |h(pkt_{in})| \geq n$). Intuitively, the chomp operator ensures that the output type reflects that the first n bits, where n is the number of bits contained in header instance ι, are removed from the input packet and copied into instance ι.

The typing rule for sequencing T-Seq is mostly standard, with one peculiarity: because our typing judgement assigns dependent function types to commands, the result type τ_{22} of command c_2 might depend on its input type τ_{12}—i.e., variable y might appear free in τ_{22}. Hence, we must also capture the type τ_{12} in the result type. The typing rule T-Skip is standard, except that it strictly enforces that the heaps described by the output type are equivalent to the heaps described by the input type. To typecheck the command *remit*, we check whether the instance to be emitted is guaranteed to be valid in the input type. The assigned output type ensures that emitting a header instance appends the value of the instance to the end of the outgoing packet (second projection of the assigned Σ-type) but leaves the input packet and all other validity information unchanged (first projection of the assigned Σ-type). The rule T-Reset resets all assumptions about header validity, empties pkt_{out} and refines pkt_{in} to be the concatenation of pkt_{out} and pkt_{in} of the input type. In the output type, we use a Σ-type to model the concatenation.

The rule T-If typechecks each branch of the conditional with the additional assumption that the condition φ holds respectively does not hold. The resulting type is a path-sensitive union type, which includes the types of both paths. By default, all variables in formula φ in the command are bound to heap. To turn φ into a refinement on a type, we substitute every occurrence of heap with the respective binder of the type we want to refine. We write $\varphi[x/\text{heap}]$ to denote the formula obtained from φ in which heap is replaced with x. For example, if the command is *if*(`ether.etherType = 0x0800`) *then extract*(`ipv4`) *else skip*, we typecheck the then-branch with type $(x : \{y : \tau_1 \mid y.\texttt{ether.etherType} = \texttt{0x0800}\}) \to \tau_{12}$. The full command is checked with type $(x : \tau_1) \to \{y : \tau_{12} \mid x.\texttt{ether.etherType} = \texttt{0x0800}\} + \{y : \tau_{22} \mid \neg x.\texttt{ether.etherType} = \texttt{0x0800}\}$.

To typecheck a modification of an instance field, the typing rule T-Mod first checks if the instance to be modified is guaranteed to be valid in the input type. The output type is similar to the strongest-postcondition of the input type: everything in the output type is the same as in x, except for the modified instance field $y.\iota.f$, which must be equal to $e[x/\text{heap}]$.

$$\textsc{T-Extract}\quad \frac{\Gamma \vdash \mathsf{sizeof}_{pkt_{in}}(\tau_1) \geq \mathsf{sizeof}(\iota) \qquad \varphi_1 \triangleq z.pkt_{in} = z.pkt_{out} = \langle\rangle \qquad \varphi_2 \triangleq y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x}{\Gamma \vdash \mathit{extract}(\iota) : (x : \tau_1) \to \Sigma y\colon \{z : \iota \mid \varphi_1\}.\{z : \mathsf{chomp}(\tau_1, \iota, y) \mid \varphi_2\}}$$

$$\textsc{T-Seq}\quad \frac{\Gamma \vdash c_1 : (x : \tau_1) \to \tau_{12} \qquad \Gamma, x : \tau_1 \vdash c_2 : (y : \tau_{12}) \to \tau_{22}}{\Gamma \vdash c_1; c_2 : (x : \tau_1) \to \tau_{22}[y \mapsto \tau_{12}]} \qquad \textsc{T-Skip}\quad \frac{\tau_2 \triangleq \{y : \tau_1 \mid y \equiv x\}}{\Gamma \vdash \mathit{skip} : (x : \tau_1) \to \tau_2}$$

$$\textsc{T-Remit}\quad \frac{\mathsf{Includes}\ \Gamma\ \tau_1\ \iota \qquad \varphi \triangleq z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota}{\Gamma \vdash \mathit{remit}(\iota) : (x : \tau_1) \to \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \epsilon \mid \varphi\}}$$

$$\textsc{T-Reset}\quad \frac{\varphi_1 \triangleq z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out} \qquad \varphi_2 \triangleq z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{in}}{\Gamma \vdash \mathit{reset} : (x\colon \tau_1) \to \Sigma y\colon \{z : \epsilon \mid \varphi_1\}.\{z : \epsilon \mid \varphi_2\}} \qquad \textsc{T-Ascribe}\quad \frac{\Gamma \vdash c : \sigma}{\Gamma \vdash c\ \mathit{as}\ \sigma : \sigma}$$

$$\textsc{T-If}\quad \frac{\cdot; \tau \vdash \varphi : \mathbb{B} \qquad \Gamma \vdash c_1 : (x\colon \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}) \to \tau_{12} \qquad \Gamma \vdash c_2 : (x\colon \{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\}) \to \tau_{22}}{\Gamma \vdash \mathit{if}(\varphi)\ c_1\ \mathit{else}\ c_2 : (x\colon \tau_1) \to \{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\}}$$

$$\textsc{T-Mod}\quad \frac{\begin{array}{c}\mathsf{Includes}\ \Gamma\ \tau_1\ \iota \\ \mathcal{F}(\iota, f) = \mathsf{BV} \qquad \cdot; \tau_1 \vdash e : \mathsf{BV} \qquad \varphi_{pkt} \triangleq y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out} \\ \varphi_\iota \triangleq \forall \kappa \in \mathit{dom}(\mathcal{HT}).\ \iota \neq \kappa \to y.\kappa = x.\kappa \wedge \qquad \varphi_f \triangleq \forall g \in \mathit{dom}(\mathcal{HT}(\iota)).\ f \neq g \to y.\iota.g = x.\iota.g\end{array}}{\Gamma \vdash \iota.f := e : (x\colon \tau_1) \to \{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = e[x/\mathsf{heap}]\}}$$

$$\textsc{T-Add}\quad \frac{\mathsf{Excludes}\ \Gamma\ \tau_1\ \iota \qquad \mathit{init}_{\mathcal{HT}(\iota)} = v \qquad \varphi \triangleq z.pkt_{in} = z.pkt_{out} = \langle\rangle \wedge z.\iota = v}{\Gamma \vdash \mathit{add}(\iota) : (x : \tau_1) \to \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \iota \mid \varphi\}} \qquad \textsc{T-Sub}\quad \frac{\Gamma \vdash \tau_1 <: \tau_3 \qquad \Gamma, x\colon \tau_1 \vdash \tau_4 <: \tau_2 \qquad \Gamma \vdash c : (x\colon \tau_3) \to \tau_4}{\Gamma \vdash c : (x\colon \tau_1) \to \tau_2}$$

Fig. 7. Command typing rules for Π4.

Rule T-Add first checks that the instance is not yet included in the type and assigns an output type that reflects that all information from the input type τ_1 are retained and just instance ι is added. The typing rule for ascription T-Ascribe is standard. The typing rule for subsumption T-Sub is also standard. We write $\Gamma \vdash \tau_1 <: \tau_2$ to denote the subtyping check between τ_1 and τ_2. The contexts Γ_1 and Γ_2 capture external dependencies of τ_1 and τ_2 respectively.

We take a semantic approach for defining subtyping as shown in the left of Figure 8. Type τ_1 with context Γ_1 is a subtype of type τ_2 with context Γ_2, if and only if for all environments $\mathcal{E}_1$ and $\mathcal{E}_2$ such that $\mathcal{E}_1$ entails the context Γ_1 for subtype τ_1 and $\mathcal{E}_2$ entails the context Γ_2 for supertype τ_2, the

$$\Gamma \vdash \tau_1 <: \tau_2 \overset{\Delta}{\Leftrightarrow} \forall \mathcal{E} \models \Gamma. [\![\tau_1]\!]_{\mathcal{E}} \subseteq [\![\tau_2]\!]_{\mathcal{E}} \qquad \mathcal{E} \models \Gamma \overset{\Delta}{\Leftrightarrow} \forall x_i \in dom(\Gamma). \mathcal{E}(x_i) = h_i \wedge h_i \models_{\mathcal{E}} \Gamma(x_i)$$

Fig. 8. Left: Subtyping. Right: Entailment between environments and subtyping contexts.

$$\textsc{Ent-Top}\ \frac{}{(I,O,H) \models_{\mathcal{E}} \top} \qquad \textsc{Ent-ChoiceL}\ \frac{(I,O,H) \models_{\mathcal{E}} \tau_1}{(I,O,H) \models_{\mathcal{E}} \tau_1 + \tau_2} \qquad \textsc{Ent-ChoiceR}\ \frac{(I,O,H) \models_{\mathcal{E}} \tau_2}{(I,O,H) \models_{\mathcal{E}} \tau_1 + \tau_2}$$

$$\textsc{Ent-Refine}\ \frac{(I,O,H) \models_{\mathcal{E}} \tau \qquad [\![\varphi]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} = \mathsf{true}}{(I,O,H) \models_{\mathcal{E}} \{x : \tau \mid \varphi\}}$$

$$\textsc{Ent-Sigma}\ \frac{(I_1,O_1,H_1) \models_{\mathcal{E}} \tau_1 \qquad (I_2,O_2,H_2) \models_{\mathcal{E}[x \mapsto (I_1,O_1,H_1)]} \tau_2}{(I_1@I_2, O_1@O_2, H_1 \cup H_2) \models_{\mathcal{E}} \Sigma x : \tau_1.\tau_2}$$

$$\textsc{Ent-Subst}\ \frac{(I_2,O_2,H_2) \models_{\mathcal{E}} \tau_2 \qquad (I,O,H) \models_{\mathcal{E}[x \mapsto (I_2,O_2,H_2)]} \tau_1}{(I,O,H) \models_{\mathcal{E}} \tau_1[x \mapsto \tau_2]}$$

Fig. 9. Entailment between heaps and heap types.

set of heaps described by τ_1 evaluated in environment $\mathcal{E}_1$ is a subset of the set of heaps described by τ_2 evaluated in environment $\mathcal{E}_2$.

The entailment between environments and typing contexts is defined in the right of Figure 8. An environment $\mathcal{E}$ entails a context Γ, iff for every mapping from a variable name x_i to some heap type τ_i in Γ there exists a mapping from variable x_i to some heap h_i in environment $\mathcal{E}$ and that heap h_i entails type τ_i. The entailment between a heap and a type is defined in Figure 9. A heap $H[pkt_{in} \mapsto I, pkt_{out} \mapsto O]$, in short (I, O, H) entails a type τ, if it is contained in the type.

3.6 Chomp

When an instance ι is extracted, sizeof(ι) bits are moved from the input bitstring to the instance—we call this *chomping*. To reflect it in the type that we assign to an extract command, we define a syntactic operation chomp that transforms a heap type into the heap type that would result from extracting an instance.

We first specify a semantic chomp operation on a single heap ($\mathsf{chomp}^{\Downarrow}(h, n)$) in Definition 3.1—it removes the first n bits from the input packet in heap h.

Definition 3.1 (Semantic Chomp). $\mathsf{chomp}^{\Downarrow}(h, n) = h[pkt_{in} \mapsto h(pkt_{in})[n:]]$

Intuitively, syntactic chomp lifts $\mathsf{chomp}^{\Downarrow}$ to heap types (written formally in Lemma 3.2). For example, given a header instance A of type `A_t { f: 2 }`, $\mathsf{chomp}(\{x : \epsilon \mid x.pkt_{in}[0:2] = 11\}, A, y) = \{x : \epsilon \mid y.A[0{:}2] = 11\}$, i.e., because header instance A contains two bits, the first two bits are moved from pkt_{in} to instance A, bound by y. It turns out that we can define chomp via a simple syntactic transformation. To do this, we first define a single-bit operation, chomp_1, that processes only a single bit. Then chomp recursively lifts chomp_1 to the appropriate length.

3.6.1 Chomp$_1$. To chomp a single bit from a heap type, we will need to update all references to the length, as well as to the first bit of pkt_{in}. This computation resembles Brzozowski derivatives [Brzozowski 1964]. For the complete definition of chomp_1, we refer the reader to the companion technical report. Here we provide some intuition for how it works. Semantically, $\mathsf{chomp}_1(\tau, \mathrm{b}_0)$ transforms for each heap h denoted by a heap type τ, into the heap $h[pkt_{in} \mapsto h(pkt_{in})[1:]]$. The variable b_0 is a placeholder corresponding to the missing bit. Then, a helper function $\mathsf{heapRef}_1$ replaces the

placeholder bits introduced by chomp_1 with references to the extracted bit. In particular, the i-th call to heapRef_1, replaces variable b_i with $x.\iota[i-1:i]$.

Syntactically, when chomping a heap type τ we update each occurrence of pkt_{in} in a refinement, if that occurrence describes the first bit of pkt_{in} of a heap in the semantics of τ. Types $\varnothing$ and $\top$ are not affected by chomping. For a choice type $\tau_1 + \tau_2$, chomp_1 is applied to both τ_1 and τ_2 individually, as each branch of the choice type describes isolated heaps of τ. In the substitution type, $\tau_1[x \mapsto \tau_2]$, only τ_1 is chomped, as τ_2 only captures information relevant for evaluating refinements.

For the refinement type $\{x : \tau_1 \mid \varphi\}$, chomp is applied recursively to τ_1 and all references to the first bit of pkt_{in} as well as the length of pkt_{in} are updated accordingly. We increment numeric expressions referencing $x.pkt_{in}$ (e.g. the refinement $|x.pkt_{in}|$ becomes $|x.pkt_{in}| + 1$ and we prepend the placeholder bit b_n for bit-vector expressions referencing $x.pkt_{in}$).

When we apply chomp_1 to type $\Sigma x : \tau_1.\tau_2$, we have to distinguish two cases, either the input packet described by τ_1 contains at least one bit or the input packet described by τ_1 is empty. In the first case, chomp_1 removes the first bit of pkt_{in} in τ_1 and in the second case it removes the first bit of pkt_{in} in τ_2. If we chomp in τ_1 we need to update all refinements to $x.pkt_{in}$ in τ_2, as τ_1 is bound to x in τ_2; otherwise chomping could cause contradictions between refinements referencing the same component. Similar to the computation of a Brzozowski derivative of a product, the result is the union of the type obtained by chomping τ_1 and τ_2 respectively, where we additionally assert in the second case that pkt_{in} of τ_1 must be empty.

Example. Given type $\tau = \Sigma x : \{y : \epsilon \mid |y.pkt_{in}| = 1\}.\{z : \epsilon \mid |x.pkt_{in}| = 1\}$

$$\begin{aligned}\text{chomp}_1(\tau, \text{b}_0) = \Sigma x : \{y : \epsilon \mid |y.pkt_{in}| + 1 = 1\}.\{z : \epsilon \mid |x.pkt_{in}| + 1 = 1\} + \\ \Sigma x : \{y : \epsilon \mid |y.pkt_{in}| = 1 \wedge |y.pkt_{in}| = 0\}.\{z : \epsilon \mid |x.pkt_{in}| = 1\}\end{aligned}$$

Example. Given a type $\tau = \{x : \{y : \iota \mid |y.pkt_{in}| = 8\} \mid x.pkt_{in}[0:8] = x.\iota[4:12]\}$

$$\text{chomp}_1(\tau, b_0) = \{x : \{y : \iota \mid |y.pkt_{in}| + 1 = 8\} \mid b_0 :: x.pkt_{in}[0:7] = x.\iota[4:12]\}$$

Example. Given a header instance A and a heap type $\tau = \{x : \epsilon \mid \text{b}_0 :: \langle\rangle @ x.pkt_{in}[0] = 10\}$. The first call to heapRef_1 returns type $\{x : \epsilon \mid (y.\iota[0] @ \langle\rangle) @ x.pkt_{in}[0] = 10\}$.

3.6.2 Correctness of Chomp. We prove that chomp is correct with respect to $\text{chomp}^{\Downarrow}$. Specifically, Lemma 3.2 states that—given some heap $h \in [\![\tau]\!]_{\mathcal{E}}$—there exists a corresponding heap h' in the semantics of the chomped type that is equivalent to the heap obtained after applying $\text{chomp}^{\Downarrow}$ to h. Since chomp adds a refinement on $x.\iota$, we evaluate the chomped type in an environment, where x maps to the heap in which ι contains the first $\text{sizeof}(\iota)$ bits from $h(pkt_{in})$. This reflects the intuition that chomp populates the header instance ι with the first $\text{sizeof}(\iota)$ bits from the input packet.

Lemma 3.2 (Semantic Chomp). *If x does not appear free in τ, then for all heaps $h \in [\![\tau]\!]_{\mathcal{E}}$ where $|h(pkt_{in})| \geq \text{sizeof}(\iota)$, there exists $h' \in [\![\text{chomp}(\tau, \iota, x)]\!]_{\mathcal{E}'}$ such that $h' = \text{chomp}^{\Downarrow}(h, \text{sizeof}(\iota))$ where $\mathcal{E}' = \mathcal{E}[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto h(pkt_{in})[0 : \text{sizeof}(\iota)]])]$.*

3.7 Safety of Π4

We prove safety of Π4 in terms of standard progress and preservation theorems. That is, well-typed programs do not get stuck and when well-typed programs are evaluated, they remain well typed. Both theorems make use of the entailment relation defined in Figure 9.

Theorem 3.3 (Progress). *If $\cdot \vdash c : (x : \tau_1) \to \tau_2$ and $(I, O, H) \models \tau_1$, then either $c = skip$ or $\exists \langle I', O', H', c' \rangle . \langle I, O, H, c \rangle \to \langle I', O', H', c' \rangle$.*

Proof. By induction on the typing derivation. For details, see the companion technical report. □

As usual, progress says that if a command is well-typed, it is either *skip* or can take a step.

Theorem 3.4 (Preservation). *If* $\Gamma \vdash c : (x : \tau_1) \rightarrow \tau_2$, $\langle I, O, H, c\rangle \rightarrow \langle I', O', H', c'\rangle$, *and* $\mathcal{E} \models \Gamma$ *and* $(I, O, H) \models_{\mathcal{E}} \tau_1$, *then there exists* $\Gamma', \mathcal{E}', x', \tau_1', \tau_2'$, *such that* $\Gamma' \vdash c' : (x' : \tau_1') \rightarrow \tau_2'$ *and* $\mathcal{E}' \models \Gamma'$ *and* $(I', O', H') \models_{\mathcal{E}'} \tau_1'$ *and* $[\![\tau_2']\!]_{\mathcal{E}'[x' \mapsto (I', O', H')]} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto (I, O, H)]}$

Proof. By induction on the typing derivation. For details, see the companion technical report. □

The preservation theorem says that if a command is a well-typed command c that can step to c', and a heap entails the input type τ_1, then c' is well-typed from τ_1' to τ_2' for some τ_1' and τ_2' such that the final heap entails τ_1' and the set of heaps described by τ_2' is a subset of the heaps denoted by τ_2 with their respective input heaps bound to variable x.

4 IMPLEMENTATION

We have built a prototype implementation of Π4's type system in OCaml and Z3. Under the hood, it uses an encoding of Π4's types into a decidable theory of first-order logic, facilitating use of an SMT solver to automatically discharge the various side conditions that arise during type checking. We describe the algorithmic type system and its decidability, some optimizations we use to simplify our types, and our $P4_{16}$ frontend.

4.1 Algorithmic Type System and Decidability

For our implementation, we define an algorithmic version of our type system whose rules are mostly identical to the rules from our declarative type system. Figure 10 shows two selected algorithmic typing rules that demonstrate the key differences from our declarative system. Many of the typing rules have semantic conditions that must be checked during type checking. In the algorithmic type system, we encode these constraints as subtype constraints. For example, when we type-check the command *add* (ι), we must check that the newly added instance is not already valid in the input type τ_1, i.e., Excludes $\Gamma\ \tau_1\ \iota$. As shown by rule T-Add-Algo in Figure 10, Excludes $\Gamma\ \tau_1\ \iota$ becomes the subtype check $\Gamma \vdash \tau_1 <: \{x : \top \mid \neg x.\iota.valid\}$. Similarly, rule T-Mod and T-Remit require Includes $\Gamma\ \tau_1\ \iota$, which becomes $\Gamma \vdash \tau_1 <: \{x : \top \mid x.\iota.valid\}$ in T-Mod-Algo and T-Remit-Algo respectively. The check $\text{sizeof}_{pkt_{in}}(\tau) \geq \text{sizeof}(\iota)$ required by T-Extract becomes $\Gamma \vdash \tau_1 <: \{x : \top \mid |x.pkt_{in}| \geq \text{sizeof}(\iota)\}$ in rule T-Extract-Algo.

The second major difference is the rule for type ascription T-Ascribe-Algo. In our implementation we check if the input type τ_1' is a subtype of the ascribed input type τ_1. We then use the ascribed input type to compute an output type τ_2. Finally, we check if the computed output type is a subtype of the ascribed output type. Note, that our type checking algorithm can be used to obtain a weak form of type inference. Given an input type that describes the state before the execution, our algorithm computes an output type, which describes the state after the execution of the program. However, a full-blown treatment of type reconstruction (such as the one used by Liquid Haskell [Vazou et al. 2018]) is left for future work.

We convert every check $\Gamma \vdash \tau_1 <: \tau_2$ into a formula in the theory of fixed-width bit vectors. This is largely straightforward, except for the encoding of pkt_{in} and pkt_{out}, which may be arbitrarily long. However, network switches have a maximum number of bits that they can receive or transmit, called the maximum transmission unit (MTU). So when compiling a P4 program to a given switch, we know that the transmitted packets must be smaller than MTU. We exploit this fact to prove a bound on the size of the bit vectors that must be considered.

More formally, we say that a type τ is bounded by N in a context Γ, written $\Gamma \vdash \tau \leq N$, iff for every $\mathcal{E} \models \Gamma$, and $h \in [\![\tau]\!]_{\mathcal{E}}$, $|h(pkt_{in})| + |h(pkt_{out})| \leq N$. We need to bound both $h(pkt_{in})$ and $h(pkt_{out})$ by N because (as seen in the *reset* command), the emitted packet is $h(pkt_{out})@h(pkt_{in})$.

$$\textsc{T-Add-Algo}\quad \frac{\Gamma \vdash \tau_1 <: \{x : \top \mid \neg x.\iota.valid\} \qquad init_{\mathcal{HT}(\iota)} = v}{\Gamma \vdash add(\iota) : (x : \tau_1) \rightsquigarrow \Sigma y : \{z : \tau_1 \mid z \equiv x\}.\{x : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle \wedge z.\iota = v\}}$$

$$\textsc{T-Ascribe-Algo}\quad \frac{\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2' \qquad \Gamma \vdash \tau_1' <: \tau_1 \qquad \Gamma, (x : \tau_1') \vdash \tau_2' <: \tau_2}{\Gamma \vdash c \ as \ (x : \tau_1) \to \tau_2 : (x : \tau_1') \rightsquigarrow \tau_2}$$

Fig. 10. Selected rules of the algorithmic type system

Theorem 4.1 (*MTU-Bound*) says that given an algorithmic typing judgement on a program c, for which the input type and all ascribed types in c respect the MTU N, the output type will require no more than $N + \mathsf{emit}(c)$ bits, where $\mathsf{emit}(c) \in \mathbb{N}$ is the number of bits that could possibly be emitted in c. The details are shown in the companion technical report. Note that even though the real input type is constrained by the same MTU N, intermediate states may require more than just N bits. *MTU-Bound* shows that $N + \mathsf{emit}(c)$ suffices as the maximum combined width of pkt_{in} and pkt_{out}.

Theorem 4.1 (MTU-Bound). *For every Γ, c, x, τ_1, τ_2, and N, if $\Gamma \vdash \tau_1 \leq N$ and $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2$ and every ascribed type in c is also bounded by N, then $\Gamma, (x : \tau_1) \vdash \tau_2 \leq N + \mathsf{emit}(c)$.*

Proof. By induction on the typing derivation. For details, see the companion technical report. □

Theorem 4.2 establishes the correctness of the algorithmic typing relation. It states that a program c typechecks in the declarative system with type $(x : \tau_1) \to \tau_2$ if and only if it also typechecks in the algorithmic system with type $(x : \tau_1) \rightsquigarrow \tau_2'$ and the output type of the algorithmic system τ_2' is a subtype of the output type τ_2 of the declarative system.

Theorem 4.2 (Algorithmic Typing Correctness). *For all Γ, c, x, τ_1, and τ_2, where x is not free in τ_1, $\Gamma \vdash c : (x : \tau_1) \to \tau_2$ if and only if there is some τ_2' such that $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2'$, and $\Gamma, (x : \tau_1) \vdash \tau_2' <: \tau_2$.*

Proof. By induction on the typing derivation. For details, see the companion technical report. □

With Theorems 4.2 and 4.1 in hand, it is straightforward to show the decidability of the declarative type system, i.e., that $\Gamma \vdash c : (x : \tau_1) \to \tau_2$ is decidable (cf. Theorem 4.3). Theorem 4.2 allows us to equivalently show that typechecking the command in the algorithmic type system—i.e., $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2'$—terminates and that checking $\Gamma, (x : \tau_1) \vdash \tau_2' <: \tau_2$ terminates, which follows by finite enumeration using the bounds guaranteed by Theorem 4.1.

Theorem 4.3 (Decidability). *If Γ, τ_1, τ_2 and every ascribed type in c are bounded by the MTU N, then $\Gamma \vdash c : (x : \tau_1) \to \tau_2$ is decidable.*

Proof. Proof by *Algorithmic Typing Correctness*, *MTU-Bound* and by induction on the typing derivation. For details, see the companion technical report. □

4.2 Rewriting Optimizations

The final major difference between the declarative type system and our implementation is that we exploit two type equivalences to eliminate Σ-types and chomp. We exploit the fact that Σ-types can be written using refinement and substitution types. In other words, in any context Γ, the type $\Sigma x : \tau_1.\tau_2$ is equivalent to

$$\left\{ x{:}\top \;\middle|\; \left(\begin{array}{l} x.pkt_{in} = l.pkt_{in}@r.pkt_{in} \wedge \\ x.pkt_{out} = l.pkt_{out}@r.pkt_{out} \end{array} \right) \wedge \bigwedge_{\iota \in dom(\mathcal{HT})} \left(\begin{array}{l} x.\iota.valid = l.\iota.valid \oplus r.\iota.valid \wedge \\ l.\iota.valid \implies x.\iota = l.\iota \wedge \\ r.\iota.valid \implies x.\iota = r.\iota \end{array} \right) \right\} \begin{array}{l}[r \mapsto \tau_2] \\ [l \mapsto \tau_1]\end{array}$$

We also can eliminate occurrences of chomp produced by extractions. Observe that in the context where $(x : \tau)$, the following two types are equivalent

$$\Sigma y : \left\{ z : \iota \;\middle|\; \begin{array}{l} z.pkt_{in} = \langle\rangle \wedge \\ z.pkt_{out} = \langle\rangle \end{array} \right\} . \left\{ z : \mathsf{chomp}(\tau, \iota, y) \;\middle|\; \left(\begin{array}{l} y.\iota@z.pkt_{in} = x.pkt_{in} \wedge \\ z.pkt_{out} = x.pkt_{out} \end{array} \right) \wedge z \equiv_\iota x \right\}$$

$$\doteq$$

$$\{y : \top \mid y.\iota.valid \wedge x.pkt_{in} = y.\iota@y.pkt_{in} \wedge x.pkt_{out} = y.pkt_{out} \wedge \textstyle\bigwedge_{\kappa \in dom(\mathcal{HT}) \wedge \kappa \neq \iota} y.\kappa = x.\kappa\}$$

Because these types are *equivalent* we can give *extract*(ι) commands the following type:

$$(x : \tau) \longrightarrow \{y : \top \mid y.\iota.valid \wedge x.pkt_{in} = y.\iota@y.pkt_{in} \wedge x.pkt_{out} = y.pkt_{out} \wedge \bigwedge_{\kappa \in dom(\mathcal{HT}) \wedge \kappa \neq \iota} y.\kappa = x.\kappa\}$$

This optimization, along with similar changes to the types for *add*(ι) and *remit*(ι) greatly reduce the size of the generated Z3 formulae, making typechecking tractable.

4.3 Beyond the core calculus

Our prototype is equipped with a $P4_{16}$ frontend that uses Petr4's parser [Doenges et al. 2021] to translate a subset of type-annotated $P4_{16}$ programs into Π4 programs. We leverage P4's builtin annotation mechanism to allow users to annotate control and parser blocks with types using the custom `@pi4`(σ) annotation, where σ is the desired type. We also provide convenience notation such as `@pi4_roundtrip`(τ), which ensures, as elaborated in Section 5.3 that the composition of deparser, *reset*, and parser has type $(x : \tau) \longrightarrow \{y : \top \mid x \equiv y\}$.

5 CHECKING NETWORK INVARIANTS

We now show that dependent types are a good match for P4, by demonstrating that Π4's type system can be used to (i) check real network protocol invariants and (ii) verify a variety of basic and advanced safety properties. We showcase properties that are also studied in the context of other P4 verification tools [Liu et al. 2018; Stoenescu et al. 2018]. All examples, in this section and the next, have been implemented in our Π4 prototype.

In most P4 programs, packet-forwarding behavior is specified using a predefined record of type `standard_metadata_t`. In particular, the `egress_spec` field instructs the switch to forward the packet out on a specific port. We assume that the field is initialized to `0x00`, indicating that no forwarding decision has been made, and that by setting the field to `0x1FF` (i.e., the largest unsigned 9-bit value), the switch can be instructed to drop the packet.[4] For simplicity, we treat all P4 metadata as an ordinary header instance called `stdmeta`.

5.1 Protocol conformance

We start with examples showing how Π4's type system can be used to ensure that a program conforms with standard network protocols.

[4]In the examples that follow, we use bitvector literals assuming that they are implicitly cast to the appropriate widths (following $P4_{16}$'s casting semantics). The implementation, however, requires these to be explicit lengths, e.g. `0b111111111` instead of `0x1FF` for a 9 bit field.

```
/* Unsafe */
if(ipv4.valid) {
  stdmeta.egress_spec := 0x1;
  ipv4.ttl := ipv4.ttl - 0x1
}
```

```
/* Safe */
if(ipv4.valid) {
  if(ipv4.ttl == 0x00) {
    stdmeta.egress_spec := 0x1FF
  } else {
    stdmeta.egress_spec := 0x1;
    ipv4.ttl := ipv4.ttl - 0x1
  }
}
```

```
(x:{y:ipv4~ | y.meta.valid}) ->
    {y:ipv4~ | y.meta.valid && (x.ipv4.ttl==0x0 => y.meta.egress_spec==0x1FF)}
```

Fig. 11. IPv4 TTL example. Top left: doesn't typecheck; top right: typechecks; bottom: Π4 type encoding the TTL invariant.

```
/* Unsafe */
extract(ether);
if(ether.etherType == 0x0800) {
  extract(ipv4)
}
```

```
/* Safe */
extract(ether);
if(ether.etherType == 0x0800) {
  extract(ipv4);
  if(ipv4.ihl != 0x5) {
    extract(ipv4opt)
  }
}
```

```
(x:{y:ϵ|y.pkt_in.length > 280}) ->
    {y:T|((y.ipv4.valid && y.ipv4.ihl != 0x5) => y.ipv4opt.valid) &&
        ((y.ipv4.valid && y.ipv4.ihl == 0x5) => !y.ipv4opt.valid)}
```

Fig. 12. IPv4 Options example. Top left: doesn't typecheck; right: typechecks; bottom: Π4 type encoding the IPv4-Option specification.

IPv4 — Time To Live. For Internet Protocol (IP) packets, the time to live (TTL) limits how often a packet can be forwarded from one network switch to another. Every time a packet is forwarded, TTL is decremented; when TTL is zero before the packet has reached its destination, forwarding halts to eliminate the risk of infinite loops.[5] The code snippet in the top left side of Figure 11 violates the property because the packet is always forwarded on the same port while TTL is decremented. We can detect this violation by checking the program with the type at the bottom of Figure 11, which reads as: Starting in a heap where at least IPv4 is valid, after executing the ingress code, still at least IPv4 is valid and if the IPv4 TTL is zero, the value of `egress_spec` indicates that the packet will be dropped. The program in the top right of Figure 11 successfully typechecks with the type.

IPv4 Options. The standard IPv4 header consists of at least 160 bits, but it may also carry additional data in optional fields. The *Internet Header Length (IHL)* field specifies the length of the header as multiples of 32 and indicates whether additional data is available. The minimum IHL is 5 ($5*32 = 160$) and the maximum is 15. Due to their flexibility, IP options are notoriously difficult to parse, and many real-world network devices handle them incorrectly. We can use Π4's type system to ensure that we also extract the IPv4 options from the input packet, whenever IPv4 is valid and IHL > 5. The type shown in the bottom of Figure 12 states that executing the parser in the empty heap where enough bits are available to extract Ethernet, IPv4 and IPv4 options, produces a heap satisfying the constraint that when IPv4 is valid and IHL is 5, IPv4 options are not valid, and when IPv4 is

[5]Strictly speaking, IPv4 requires a special ICMP message to be returned to the sender to indicate the error.

```
/* Unsafe */
extract(ether);
extract(ipv4)
```

```
/* Safe */
extract(h.ether);
if(ether.etherType == 0x0800) {
  extract(ipv4)
}
```

```
(x: {y:ϵ | y.pkt_in.length > 272}) ->
          {y:T | y.ipv4.valid => y.ether.etherType == 0x0800}
```

Fig. 13. Header dependency example. Top left: doesn't typecheck; top right: typechecks; bottom: Π4 type encoding IPv4's dependency on Ethernet.

valid and IHL > 5, then IPv4 options are valid. Figure 12 shows one example where this property is violated (top left) and one where it holds (top right).

Header Dependencies. Most protocols have some way of keeping track of what other protocols are encapsulated in the payload of a packet—i.e., which header follows next. The correspondence between field values and protocols is typically defined as part of the protocol standard. For example, an Ethernet frame uses the EtherType field (written `ether.etherType`) for this purpose: a value of `0x0800` indicates that the next header is an IPv4 header, while a value of `0x86dd` indicates that the next header is an IPv6 header. This is specified in the type at the bottom of Figure 13. The code snippet on the top left of Figure 13 violates the dependency between the IPv4 header and the EtherType field of the Ethernet header. Our type checker detects this by checking that executing the parser on an empty heap with enough bits to extract both Ethernet and IPv4, produces a heap with either an invalid or a valid IPv4 header and an EtherType value of `0x0800`. The code on the top right of Figure 13 fixes the error by only extracting `ipv4` when `ether.etherType` is `0x0800`.

5.2 Basic Safety Properties

Π4's type system can be also used to ensure safety properties. We discussed how it detects accesses to invalid header instances in Section 2. Here we present an example showing how it can be used to enforce *determined forwarding* [Liu et al. 2018; Stoenescu et al. 2018]. Typical P4 programs contain thousands of paths on which a packet can be processed. To avoid situations where packets are dropped unexpectedly, a desirable invariant is that each program path contains an explicit forwarding decision—i.e., packets are either forwarded on some switch port or dropped. Our type checker is able to detect violations of this property. The type in the bottom of Figure 14 shows one way of encoding this specification as a type. Under the assumption that the egress specification is initialized with a dummy value of `0x0`, the type asserts that it is, at some point, modified during the execution of the pipeline, i.e., a forwarding decision is made for every packet. The program on the top left of Figure 14 fails to typecheck with the type, because the egress specification is unset for packets with `ipv4.dst` equal to `0x0a0a0a0a`. The program on the top right of Figure 14 fixes this issue via an else-case that assigns the egress specification to `0x1FF`.

5.3 Parser-Deparser Compatibility

A P4 program typically defines the parser, controls for ingress and egress pipelines, and the deparser.[6] In practice, parsing and deparsing may also happen between the ingress and egress stages—i.e., the deparser code is additionally executed at the end of the ingress followed by the parser code, before the egress. In such cases, it is important to ensure that data intended to be

[6]Why this four-phase structure? Having separate ingress and egress pipelines allows packet processing to occur both before and after packets are scheduled, typically using one or more queues.

```
1  /* Unsafe */
2  if(ipv4.valid) {
3    if(ipv4.dst != 0x0a0a0a0a) {
4      stdmeta.egress_spec := 0x1FF
5    }
6  }
```

```
1  /* Safe */
2  if(ipv4.valid) {
3    if(ipv4.dst != 0x0A0A0A0A) {
4      stdmeta.egress_spec := 0x1
5    } else {
6      stdmeta.egress_spec := 0x1FF
7    }
8  }
```

```
(x:{y:ipv4~| y.stdmeta.valid}) -> {y:⊤| y.stdmeta.egress_spec != 0x0}
```

Fig. 14. Determined forwarding example. Top left: ill-typed; top right: well-typed; bottom: determined forwarding specification encoded as Π4 type.

carried from ingress to egress is serialized and deserialized correctly. Otherwise, headers may be unexpectedly removed from the packet.

For example, assume that the parser shown in Figure 15 successfully parses the Ethernet and IPv4 headers from the input packet, but not a VLAN header. From the code we can conclude that EtherType must be $0x0800$. Let's further assume that the programmer intends the ingress control in the middle right of Figure 15. After parsing, the switch checks if a VLAN header is present. If a VLAN header was already parsed from the input packet, no changes are made. Otherwise a VLAN header is added (Line 27) and the EtherType of the Ethernet header is updated accordingly. If an IPv4 header is present, the EtherType must be updated accordingly (Line 31) to obtain a protocol-conformant packet. Now, assume that the programmer forgot the statement on Line 31, i.e., didn't update `ether.etherType` (this unsafe example is in the left of Figure 15). After running the deparser at the end of ingress, all three headers are serialized: The first 112 bits correspond to the Ethernet header, followed by 32 bits of the VLAN header, and another 160 bits of the IPv4 header. Since the programmer forgot to update EtherType, bits 96 to 112 contain the value `0x0800`. If the parser runs with this bitstream as the input, it will first parse the Ethernet header, then look at the `etherType` and given the value $0x0800$, it will continue to parse the IPv4 header. Hence, the bits of the VLAN header are parsed as an IPv4 header, leading to a corrupted packet.

To avoid such errors, we want to enforce the invariant that all instances valid at the end of ingress are equivalent to those obtained after deparsing and re-parsing. We instruct our type checker to verify this property via the type on Line 36, by checking the whole program with a type that ensures there are enough bits to parse the headers, shown on Line 1.

5.4 Mutual Exclusion of Headers

The parser shown on the top of Figure 16 conditionally parses either IPv4 or IPv6. Because only one of the paths is taken at runtime, it should never happen that both instances are valid at the same time. This property might be exploited in an implementation, allowing the same memory to be used to store both headers. In this small example, it is easy to see that this invariant holds. But in larger programs it is difficult to track which header instances are valid on which execution paths. We can check that the property continues to hold in the ingress in the middle of Figure 16 using the type in the ascription on Line 1. The ingress code in the middle left of Figure 16 exemplifies a violation of the property: If a packet enters the control block with a valid IPv4 header, it will leave with both a valid IPv4 and a valid IPv6 header; a violation of our property. The code on the middle right is safe because it includes a conditional that explicitly checks the validity of IPv4 before adding IPv6. The combination of union types and refinement types makes our type system capable of such precise path-dependent reasoning.

```
Parser ≜
  extract(ether);
  if(ether.etherType == 0x8100) {
    extract(vlan);
    if(vlan.etherType == 0x0800) {
      extract(ipv4)
    }
  } else {
    if(ether.etherType == 0x0800) {
      extract(ipv4)
    }
  }

Deparser ≜
  if (ether.valid) { remit(ether) };
  if (vlan.valid) { remit(vlan) };
  if (ipv4.valid) { remit(ipv4) }
```

```
UnsafeIngress ≜
  if(!vlan.valid) {
    add(vlan);
    vlan.etherType := 0x0;
    if(ipv4.valid) {
      vlan.etherType := 0x0800
    }
  };

SafeIngress ≜
  if(!vlan.valid) {
    add(vlan);
    vlan.etherType := 0x0;
    ether.etherType := 0x8100;
    if(ipv4.valid) {
      vlan.etherType := 0x0800;
    }
  }
```

```
Parser; Ingress; /* Ingress is either UnsafeIngress or SafeIngress  */
(Deparser; remit; Parser) as
    (x:{z:ether~|z.ether.etherType == 0x8100 && z.vlan.valid &&
                 (z.ipv4.valid <=> z.vlan.etherType == 0x0800) &&
                 z.pkt_out.length == 0 &&
                 z.pkt_in.length > 0}) -> {y:T| x === y}
```

```
(x:{y:ϵ|y.pkt_out.length == 0 && y.pkt_in.length > 304}) -> T
```

Fig. 15. Roundtripping Definitions. Top left: common parser and deparser; top right: unsafe and safe ingress code; middle: the pipeline, which typechecks with `Ingress` ↦ `UnsafeIngress`, but not with `Ingress` ↦ `UnsafeIngress`; bottom: the type at which to check the full pipeline.

```
(extract(ether);
if (ether.etherType == 0x86dd) { extract(ipv6) }
else { if(ether.etherType == 0x0800) { extract(ipv4) } })
  as (x:{y:ϵ|y.pkt_in.length>432}) -> {y:ether~|!(y.ipv4.valid&&y.ipv6.valid)};
Ingress /* Can be SafeIngress or UnsafeIngress */
  as (x: {y:ether~|!(y.ipv4.valid && y.ipv6.valid)}) ->
        {y:ether~|!(y.ipv4.valid && y.ipv6.valid)});
if (ether.valid) { remit(ether) };
if (ipv4.valid) { remit(ipv4) };
if (ipv6.valid) { remit(ipv6) }
```

```
UnsafeIngress ≜
  add(ipv6)
  ether.etherType := 0x86DD
```

```
SafeIngress ≜
  if(ipv4.valid) {
    add(ipv6);
    ether.etherType := 0x86DD
  }
```

```
(x:{y:ϵ|y.pkt_in.length>432}) -> T
```

Fig. 16. Mutual exclusion example: IPv4 and IPv6 should never be simultaneously valid. Top: Common pipeline; middle left: unsafe `Ingress` code; middle right: safe `Ingress` code; bottom: whole program type.

5.5 Limitations

There are a few P4 features that our Π4 prototype does not support, mostly because they pose challenges to SMT-based approaches to verification: hash functions, externs (a kind of foreign

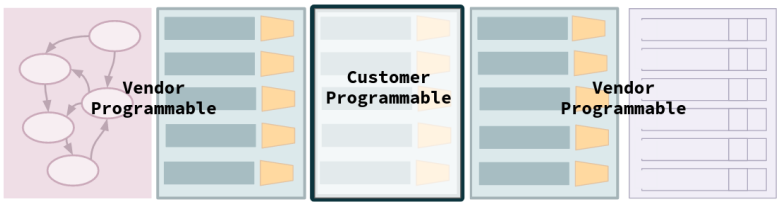

Fig. 17. Modular router design

```
1  extract(vlan);
2  Ingress /* Customer Specified. Default, Overwrite, Table, or UnsafeActions */
3    as (x:Σx:stdmeta.ipv4) -> {y:Σy:stdmeta~.ipv4~|y.vlan == x.vlan};
4  remit(vlan)
```

Fig. 18. Instantiation of modular router design; the parser (*Pre*) and deparser (*Post*) are provided by the vendor, the `Ingress` code is provided by the customer.

function interface into the hardware), and registers. The unpredictability of hash functions is difficult to verify, but we can either over-approximate them as uninterpreted functions, or use a more fine-grained approach such as concolic verification. Externs either need to be annotated with specific types, or over-approximated as uninterpreted functions. Registers are on-switch state that can be modified by the packet or the controller and persists between packets. This is tricky to represent in the semantics and has some distributed computing concerns. We could over-approximate the behavior by havoc-ing the values every time the register is read.

6 MODULARITY REASONING WITH Π4

An emerging design pattern for data plane switches is partial programmability, e.g., Cisco's daPIPE [Baldi 2019], which is designed for the Nexus 3400 switch [Cisco 2018]. The idea is that a device vendor provides a partially-implemented pipeline together with a set of program points where customers can inject custom code as shown in Figure 17. The designer requires that customer programs satisfy certain properties, but in current architectures, they are not automatically checked.

To illustrate, consider a deployment of the customizable pipeline in a campus network where network engineers want to experiment with in-band network telemetry (INT) without perturbing the VLAN tag, which is used to enforce security policies. Let's say there are four classes of traffic, `Visitor`, `Student`, `Faculty`, and `Staff`, each with unique VLAN identifiers. We want to ensure that no matter how `Ingress` is instantiated in the left of Figure 18, it cannot cause students and visitors to acquire privileges of faculty or staff—e.g., this might leak confidential data. With Π4, we can design a modular system that statically checks invariants on customer programs. Practically, we can ensure that VLAN is not changed by checking that customer's code has a type like: $(x : \tau) \rightarrow \{y : \tau' \mid x.\texttt{vlan.vid} = y.\texttt{vlan.vid}\}$, where the τ and τ' are appropriate for the specific pipeline. We check, once-and-for-all, that the surrounding switch code composes with this type, and incrementally check that the customer code has this type (for an appropriate τ).

6.1 Specifying an Invariant

To further illustrate, consider the toy example shown the right of Figure 18, which has a VLAN instance (32 bits) and the standard metadata used in the P4 switch model (325 bits), including a 9-bit egress specification `stdmeta.egress_spec`, and a 12-bit vlan tag field, `vlan.vid`. The control flow simply extracts the VLAN instance, executes the modular `Ingress` control, and then emits

```
5  Default ≜ skip

6  Table ≜
7    add(_vlan_table);
8    if (_vlan_table.vid_key ==
9        vlan.vid){
10     if (_vlan_table.act == 0b0){
11       stdmeta.egress_spec := 0x1FF
12     } else {
13       stdmeta.egress_spec := 0x1
14     }
15   }
```

```
16 Overwrite ≜ vlan.vid := Faculty

17 UnsafeActions ≜
18   add(_vlan_table);
19   if (_vlan_table.vid_key ==
20       vlan.vid) {
21     if (_vlan_table.act == 0b0) {
22       vlan.vid := Faculty
23     } else { vlan.vid := Staff }
24   } else {
25     vlan.vid := Visitor
26   }
```

Fig. 19. A collection of safe and unsafe customer implementations for the Ingress module from Figure 18. Top Left: *Default*; Top Right: *Overwrite*; Bottom Left: *Table*; Bottom Right: *Unsafe Actions*

the VLAN header. We want the program to typecheck with type $(x : \{x : \texttt{stdmeta} \mid |x.pkt_{in}| > 32\}) \to \Sigma y : \texttt{stdmeta~.vlan~}$.

In this example, the type ascription provides compositional reasoning—i.e., we don't need to re-check that the whole pipeline is well-typed. Instead, we check once-and-for all that *extract*(vlan) has type $(x : \{x : \texttt{stdmeta} \mid |x.pkt_{in}| > 32\}) \to (x : (\Sigma x : \texttt{stdmeta.ipv4}))$, and that *remit*(vlan) has type $(x : (\{x : \Sigma x : \texttt{stdmeta.ipv4} \mid x.\texttt{vlan} = y.\texttt{vlan}\})) \to \Sigma y : \texttt{stdmeta~.vlan~}$ in context $(y : \Sigma x : \texttt{stdmeta.ipv4})$. Both are easy to check.

Now, when we swap in different implementations for Ingress, we only need to check that it has its ascribed type on Line 3 of Figure 18, without rechecking the surrounding code. With the infrastructure Π4's type system provides, network engineers can make changes to their experimental module Ingress and check its compatibility with the switch without re-checking the feasibility of the whole switch in a modular fashion.

6.2 Checking Customer Programs

We now consider a collection of customer programs as shown in Figure 19 that an engineer may want to install into the switch and how Π4 prevents security vulnerabilities by ensuring the customer code has the type annotated on Line 3 of Figure 18.

Default. Consider the empty program, shown in the top left of Figure 18, which would surely be the default behavior when the programmer hasn't written any code yet. To typecheck this no-op module, we check that *skip* has the ascribed type, which it clearly does, since it does not change the value of the program.

Overwrite. Conversely, if the customer were to install a blatantly incorrect program as the one in the top right of Figure 18, which always overwrites the VLAN tag with the identifier reserved for faculty members, the type system complains that the following subtyping check fails—when x.vlan.vid is, say, Student, the two types denote disjoint sets of heaps.

$$\begin{aligned}&(x : \Sigma y : \texttt{stdmeta.vlan})\\ &\quad \vdash \{w : \Sigma y : \texttt{stdmeta~.vlan~} \mid w.\texttt{vlan.vid} = \texttt{Faculty}\}\\ &\qquad <: \{w : \Sigma y : \texttt{stdmeta~.vlan~} \mid w.\texttt{vlan.vid} = x.\texttt{vlan.vid}\},\end{aligned}$$

Table. We model match-action tables using an encoding similar to the one used in p4v [Liu et al. 2018], where we create an extra header that captures the keys and selected action. Consider the vlan table on the bottom left of Figure 18, which matches on vlan.vid and selects one of two actions: the first sets the egress_spec to 0x1FF; the second sets it to 0x001. To encode this table,

we create a new header `_vlan_table` with a 12-bit field `vid_key` and a 1-bit field `act`, modelling the table application via the code shown in the bottom left of Figure 19. This will typecheck since no branch of the code modifies the `vlan.vid` field, and `_vlan_table` is permitted to be valid.

Unsafe Actions. Now consider a `vlan` table where each action *does* modify the VLAN id. For example, the table shown in the bottom right of Figure 18 can either set the VID to one of {`Faculty`, `Staff`}, or, if the packet misses in the table, to `Visitor`. Here, the VLAN id is clobbered whenever this table is applied, triggering a violation of the subset check just as with *Overwrite*.

7 RELATED WORK

Formal Reasoning for P4 Programs. A number of verification approaches have been proposed for P4 programs. `p4v` [Liu et al. 2018] applies classical techniques based on predicate transformer semantics to achieve monolithic verification of P4 programs. Vera [Stoenescu et al. 2018] and P4-Assert [Freire et al. 2018] are symbolic execution engines for P4. The `bf4` tool [Dumitrescu et al. 2020] follows the approach of `p4v`, but also attempts to infer control-plane constraints that are sufficiently strong to establish correctness, and offers heuristics for repairing programs when verification fails. SafeP4 [Eichholz et al. 2019] uses a simple type system to track header validity. Petr4 [Doenges et al. 2021] develops a formal semantics for P4 but does not itself offer verification tools. In constrast to this earlier work, Π4 uses dependent types and offers compositional verification.

Dependent Types. Early work by Xi and Pfenning [1999] showed how dependent types could be used to eliminate run-time safety checks—e.g., array bounds checks in imperative programs. Xanadu [Xi 2000] adds dependent typing to imperative programming, but does not capture the effect of mutations in the type. Xi and Harper [2001] later showed how dependent types could be applied to assembly code. Deputy [Condit et al. 2007] used dependent types to reason about complex, heap-allocated data structures. Similar to Deputy, Π4's typing rule for modification of header fields is also inspired by the Hoare axiom for assignment. Π4 is different in that typechecking has no effect on the run time and also supports path-sensitive reasoning. Similar to Π4, Hoare Type Theory (HTT) [Nanevski et al. 2006] statically tracks how the heap evolves during execution. Typing of computations in HTT is similar to the dependent function types Π4– the type captures the state before and after execution, possibly relating the output type with the input type. In our domain, this requires bit-by-bit transformations on the input type, provided by chomp. Other type systems like Ynot [Nanevski et al. 2008], FCSL [Nanevski et al. 2014], and F* [Swamy et al. 2016] provide dependent types for low-level imperative programming. They target general functional verification and often require manual programs-as-proofs to do so. Π4 is designed with domain-specific properties of network programming in mind and is fully automatic.

Solver-Aided Tools. Recent work on dependently-typed languages has focused on automation, building on advances in SAT/SMT solvers to make dependent types usable by ordinary programs. A prominent example is Liquid Haskell [Rondon et al. 2008], which extends Haskell with decidable refinement types. Under the hood, proof obligations generated during type checking are transparently handled by an SMT solver. Just as Liquid Haskell requires its refinements to be in the theory of quantifier-free integer linear arithmetic in order to be decidable, Π4 stipulates that types must denote finite sets—a restriction justified by the domain. This assumption lets us encode types into the effectively propositional fragment of first-order logic over bit vectors.

Formalizing Protocols. Another line of work focuses on language-based specifications of protocols. CMU's FoxNet project used SML to specify the behavior of an entire networking stack [Biagioni et al. 1994]. McCann and Chandra [2000] used a type-based approach to give abstract specifications of protocols. Grammar-based tools such as PADS [Fisher and Gruber 2005], Narcissus [Delaware

et al. 2019], and Yakker [Jim et al. 2010], enable specifying the syntax of complex, dependent formats including network protocols, and provide tools for serializing and deserializing data. They focus exclusively on deriving correct parsers from a typed representation of data and may be suitable to describe the header formats and the parser, but there is no equivalent to our chomp operator that allows us to statically capture how the input packet changes during parsing.

8 CONCLUSION AND FUTURE WORK

This paper presented Π4—the first dependently-typed language for data plane programming—a domain with difficult challenges where programming language theory can have a big impact. In particular, low-level data plane languages like P4 seem to be a sweet spot for dependent types. On the one hand, precise types are necessary because critical correctness properties often hinge on intricate, bit-level packet formats, where the first few bits of a packet determine the format, the length, and the processing of the following ones. On the other hand, a high degree of automation is possible due to the restricted nature of the language, which does not support pointers, loops, or other features that often complicate very precise type systems. Yet, thus far, dependent typing has not been explored for data plane programs—the community has relied on verification tools that lack compositional reasoning.

Π4's type system is innovative in its combination of refinement types, dependent function types, a limited form of regular types, including unions, explicit substitutions, and a primitive "chomp" operation, reminiscent of regular expression derivatives [Brzozowski 1964], which can be used to give a precise type to P4's parsing constructs. It is capable of statically checking advanced properties of data plane programs that combine packet serialization and deserialization operations with imperative control-flow. Under the hood, an SMT solver automatically discharges the formulas generated during type checking without requiring any manual proof. We define Π4 formally and prove type soundness and decidability. Our case studies demonstrate how Π4 supports modular reasoning in scenarios ranging from basic safety properties to intricate invariants.

There are a number of interesting directions for future work. We plan to investigate connections that our verified approach to parsing using derivatives may have to other domains, e.g., verified serializers and deserializers like EverParse [Ramananandro et al. 2019] and Narcissus [Delaware et al. 2019]. Another direction is to consider the effect of modularity on verification times; if a tool incrementally caches verification results for ascribed code blocks, it would only have to check the portions of the code that change between runs of the typechecker. Some preliminary experiments indicate that modular typechecking offers significant benefits, but an empirical study to answer this question carefully is left for future work, after we have optimized our prototype. We also plan to extend Π4 to handle more complicated features of P4 perhaps requiring concolic techniques [Godefroid et al. 2005]. Further, understanding whether dependent types are the right interface for modular verification of dataplane programs is important. In fact, user studies investigating the appropriate typing interfaces, such as gradual typing and type inference, would be important for guiding the design of impactful systems for modular data plane verification.

ACKNOWLEDGMENTS

We are grateful to the POPL reviewers for their careful feedback and many suggestions for improving this paper. Our work has been supported in part by the German Research Foundation (DFG) as part of the Collaborative Research Center (CRC) 1053 MAKI, by the National Research Center for Applied Cybersecurity ATHENE, by the National Science Foundation under grant FMiTF-1918396 as well as a Graduate Research Fellowship, the Defense Advanced Research Projects Agency under Contract HR001120C0107, and gifts from Keysight and InfoSys.

REFERENCES

M. Baldi. 2019. daPIPE a Data Plane Incremental Programming Environment. In *Symposium on Architectures for Networking and Communications Systems (ANCS)*. 1–6. https://doi.org/10.1109/ANCS.2019.8901893

Ryan Beckett and Ratul Mahajan. 2020. A General Framework for Compositional Network Modeling. In *Workshop on Hot Topics in Networks (HotNets)*. 8–15. https://doi.org/10.1145/3422604.3425930

Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. 1994. Signatures for a Network Protocol Stack: A Systems Application of Standard ML. In *Conference on LISP and Functional Programming (LFP)*. 55–64. https://doi.org/10.1145/182409.182431

Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Computer Communications Review (CCR)* (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Conference of the Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 99–110. https://doi.org/10.1145/2486001.2486011

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. https://doi.org/10.1145/321239.321249

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 2014. Polymorphic Functions with Set-Theoretic Types: Part 1: Syntax, Semantics, and Evaluation. In *Symposium on Principles of Programming Languages (POPL)*. 5–17. https://doi.org/10.1145/2535838.2535840

Cisco. 2018. Cisco Nexus 3000 Series Switches. https://www.cisco.com/c/en/us/products/switches/nexus-3000-series-switches/index.html

Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. 2007. Dependent Types for Low-Level Programming. In *European Symposium on Programming (ESOP)*. 520–535. https://doi.org/10.1007/978-3-540-71316-6_35

Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats. *Proceedings of the ACM Programming Languages (PACMPL)* 3, ICFP, Article 82 (July 2019). https://doi.org/10.1145/3341686

Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. 2021. Petr4: Formal Foundations for P4 Data Planes. *Proceedings of the ACM on Programming Languages (PACMPL)* 5, POPL, Article 41 (Jan. 2021). https://doi.org/10.1145/3434322

Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. 2020. Bf4: Towards Bug-Free P4 Programs. In *Conference of the Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 571–585. https://doi.org/10.1145/3387514.3405888

Matthias Eichholz, Eric Campbell, Nate Foster, Guido Salvaneschi, and Mira Mezini. 2019. How to Avoid Making a Billion-Dollar Mistake: Type-Safe Data Plane Programming with SafeP4. In *European Conference on Object-Oriented Programming (ECOOP)*. 12:1–12:28. https://doi.org/10.4230/LIPIcs.ECOOP.2019.12

Robert Ennals, Richard Sharp, and Alan Mycroft. 2004. Linear Types for Packet Processing. In *European Symposium on Programming (ESOP)*. 204–218. https://doi.org/10.1007/978-3-540-24725-8_15

Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-Specific Language for Processing Ad Hoc Data. In *Conference on Programming Language Design and Implementation (PLDI)*. 295–304. https://doi.org/10.1145/1065010.1065046

Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Uncovering Bugs in P4 Programs with Assertion-Based Verification. In *Symposium on SDN Research (SOSR)*. Article 4, 7 pages. https://doi.org/10.1145/3185467.3185499

Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *Conference of the Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 435–450. https://doi.org/10.1145/3387514.3405879

Vladimir Gapeyev and Benjamin C. Pierce. 2003. Regular Object Types. In *European Conference on Object-Oriented Programming (ECOOP)*. 151–175. https://doi.org/10.1007/978-3-540-45070-2_8

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*. 213–223. https://doi.org/10.1145/1065010.1065036

Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. 54–66. https://doi.org/10.1145/3281411.3281443

Haruo Hosoya and Benjamin C. Pierce. 2003. XDuce: A Statically Typed XML Processing Language. *ACM Transactions on Internet Technology (TOIT)* 3, 2 (May 2003), 117–148. https://doi.org/10.1145/767193.767195

Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *Symposium on Field-Programmable Gate Arrays (FPGA)*. 1–9. https://doi.org/10.1145/3289602.3293924

Trevor Jim, Yitzhak Mandelbaum, and David Walker. 2010. Semantics and Algorithms for Data-Dependent Grammars. In *Symposium on Principles of Programming Languages (POPL)*. 417–430. https://doi.org/10.1145/1706299.1706347

Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Symposium on Operating Systems Principles (SOSP)*. 121–136. https://doi.org/10.1145/3132747.3132764

Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. p4v: Practical Verification for Programmable Data Planes. In *Conference of the Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 490–503. https://doi.org/10.1145/3230543.3230582

Conor McBride. 2001. The derivative of a regular type is its type of one-hole contexts. Extended abstract, available at http://strictlypositive.org/diff.pdf.

Peter J. McCann and Satish Chandra. 2000. Packet Types: Abstract Specification of Network Protocol Messages. In *Conference of the Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 321–333. https://doi.org/10.1145/347059.347563

Chitra Muthukrishnan, Vern Paxson, Mark Allman, and Aditya Akella. 2010. Using Strongly Typed Networking to Architect for Tussle. In *Workshop on Hot Topics in Networks (HotNets)*. Article 9, 6 pages. https://doi.org/10.1145/1868447.1868456

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *European Symposium on Programming (ESOP)*. 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and Separation in Hoare Type Theory. In *International Conference on Functional Programming (ICFP)*. 62–73. https://doi.org/10.1145/1159803.1159812

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: Dependent Types for Imperative Programs. In *International Conference on Functional Programming (ICFP*. 229–240. https://doi.org/10.1145/1411204.1411237

Tahina Ramananandro, Antoine Delignat-Lavaud, Cedric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *USENIX Security Symposium (USENIX Security)*. 1465–1482. https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Conference on Programming Language Design and Implementation (PLDI)*. 159–169. https://doi.org/10.1145/1375581.1375602

Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. [n.d.]. Composing Dataplane Programs with μP4. In *Conference of the Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 329–343. https://doi.org/10.1145/3387514.3405872

Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 Programs with Vera. In *Conference of the Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 518–532. https://doi.org/10.1145/3230543.3230548

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Symposium on Principles of Programming Languages (POPL)*. 256–270. https://doi.org/10.1145/2837614.2837655

Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical Types for Untyped Languages. In *International Conference on Functional Programming (ICFP)*. 117–128. https://doi.org/10.1145/1863543.1863561

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *International Conference on Functional Programming (ICFP)*. 269–282. https://doi.org/10.1145/2628136.2628161

Niki Vazou, Éric Tanter, and David Van Horn. 2018. Gradual Liquid Type Inference. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, OOPSLA, Article 132 (Oct. 2018), 25 pages. https://doi.org/10.1145/3276502

Han Wang, Robert Soulé, Huynh Tu Dang, Ki-Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A Rapid Prototyping Framework for P4. In *Symposium on SDN Research (SOSR)*. 122–135. https://doi.org/10.1145/3050220.3050234

Hongwei Xi. 2000. Imperative programming with dependent types. In *Symposium on Logic in Computer Science (LICS)*. 375–387. https://doi.org/10.1109/LICS.2000.855785

Hongwei Xi and Robert Harper. 2001. A Dependently Typed Assembly Language. In *International Conference on Functional Programming (ICFP)*. 169–180. https://doi.org/10.1145/507635.507657

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Symposium on Principles of Programming Languages (POPL)*. 214–227.