TECHNISCHE UNIVERSITÄT DARMSTADT

Type-Safe Data Plane Programming

vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

M.Sc. Matthias Eichholz

geboren in Oberhausen

Gutachter

Prof. Dr.-Ing. Mira Mezini
Technische Universität Darmstadt

Prof. Nate Foster
Cornell University, USA

Darmstadt, 2022

Matthias Eichholz: Type-Safe Data Plane Programming
Darmstadt, Technische Universität Darmstadt
Tag der mündlichen Prüfung: 12.10.2022
Jahr der Veröffentlichung der Dissertation auf TUprints: 2022
URN: urn:nbn:de:tuda-tuprints-228736

Abstract

Since the mid-1990s, there have been efforts to enable more flexible processing of network packets by making packet processing programmable. With the advent of software-defined networking (SDN), this idea has now become a reality. Early approaches initially focused on control plane programming, with the goal of implementing centralized network policies at a high level of abstraction without having to use low-level, device-specific configuration mechanisms. For this purpose, various network programming languages have been developed, which provide correctness guarantees and make the formal verification of network policies possible.

More recently, it is also possible to program the network data plane. Being able to define the structure of network packet headers freely, opens up a whole new range of applications, from implementing new network protocols up to moving application logic directly into the network. Until today, the P4 language has become the de facto standard for programming data planes. While P4 provides declarative abstractions for programming data planes, P4 lacks basic safety guarantees to help avoid errors and implement correct applications for the data plane.

Modern programming languages use static type systems to provide languages with basic safety guarantees that completely eliminate the occurrence of entire categories of errors. Surprisingly, however, the use of type systems in the field of network programming has hardly been investigated. This dissertation investigates what appropriate type systems must look like in order to provide data plane programming languages—in particular, P4—with static correctness guarantees. As a first step, we present SafeP4, a domain-specific language for programmable data planes that is equipped with a static type system that guarantees that all headers that are read or written are valid, which is a common cause of errors. We then present Π4, whose type system is based on dependent types and is thus able to bridge the gap in terms of expressiveness between SafeP4 and full-fledged verification tools. At the same time, Π4 enables modular verification of programs.

Our evaluation using open source programs confirms that accessing invalid packet headers is a common source of errors in practice and that the SafeP4's type system is capable of identifying buggy programs. Using case studies, we show that Π4's type system is capable of expressing and verifying a variety of real-world correctness properties.

Zusammenfassung

Seit Mitte der 1990er Jahre gibt es Bestrebungen, eine flexiblere Verarbeitung von Netzwerkpaketen zu ermöglichen, indem Paketverarbeitung programmierbar wird. Mit dem Aufkommen von Software-Defined-Networking (SDN) ist diese Idee nun Realität geworden. Frühe Ansätze haben sich zunächst auf die Programmierung der Control-Plane konzentriert, mit dem Ziel, auf hohem Abstraktionsniveau zentrale Netzwerkrichtlinien zu realisieren, ohne systemnahe, gerätespezifische Konfigurationsmechanismen nutzen zu müssen. Hierfür entstanden diverse Programmiersprachen, die Korrektheitsgarantien gewähren und die formale Verifikation der Netzwerkrichtlinien ermöglichen.

In jüngster Zeit ist es zudem auch möglich die Netzwerk-Data-Plane zu programmieren. Durch die Möglichkeit die Struktur von Netzwerkpaketheadern frei zu definieren, eröffnen sich eine Vielzahl neuer Anwendungsmöglichkeiten, von der Implementierung neuartiger Netzwerkprotokolle bis hin zur Auslagerung von Anwendungsfunktionalität direkt in das Netzwerk. Bis heute hat sich die Sprache P4 als Standard für die Programmierung von Data-Planes durchgesetzt. Zwar bietet P4 deklarative Abstraktionen für die Programmierung von Data-Planes, allerdings fehlen P4 grundlegende Sicherheitsgarantien, die dabei helfen Fehler zu vermeiden und korrekte Anwendungen für die Data-Plane zu implementieren.

Modernen Programmiersprachen nutzen statische Typsysteme, um Sprachen mit grundlegenden Sicherheitsgarantien auszustatten, die das Auftreten ganzer Fehlerkategorien vollständig ausschließen. Überraschenderweise wurde der Einsatz von Typsysteme im Bereich der Netzwerkprogrammierung jedoch bislang kaum untersucht. Diese Dissertation untersucht wie geeignete Typsysteme aussehen müssen, um Data-Plane-Programmiersprachen – insbesondere P4 – mit statischen Korrektheitsgarantien auszustatten. Im ersten Schritt präsentieren wir SafeP4, eine domänenspezifische Sprache für programmierbare Data-Planes, die mit einem statischen Typsystem ausgestattet ist, das garantiert, dass alle Header, die gelesen oder geschrieben werden gültig sind, was eine häufige Ursache für Fehler ist. Im zweiten Schritt präsentieren wir Π4, dessen Typsystem auf Dependent-Types basiert und damit in der Lage ist, die Lücke hinsichtlich der Ausdrucksstärke zwischen SafeP4 und vollwertigen Verifikationswerkzeugen zu schließen. Gleichzeitig ermöglicht Π4 die modulare Verifikation von Programmen.

Unsere Auswertung anhand von Open-Source-Programmen bestätigt, dass der Zugriff auf ungültige Paketheader in der Praxis eine häufige Fehlerquelle ist und das SafeP4s Typsystem in der Lage ist, fehlerhafte Programme zu identifizieren. Anhand von Fallstudien zeigen wir zudem, dass Π4s Typsystem imstande ist eine Vielzahl praktisch relevanter Korrektheitseigenschaften auszudrücken und zu verifizieren.

Acknowledgements

First and foremost, I thank my advisor Mira Mezini, who made it possible for me to pursue a Phd in the first place and who helped me to ultimately achieve this goal. Thank you, Mira, for the freedom to follow my own research ideas. However, I am at least as grateful to Nate Foster, without whom many aspects of this work would probably have developed quite differently. Thank you, Nate, I really learned a lot from you about what it needs to be a PL researcher.

I also like to thank the remaining members of my PhD committee Reiner Hähnle, Zsolt István, and Marie-Christine Jakobs. Furthermore, I like to thank many others whom I met and worked with during my time as a Phd student. In particular, I thank Eric Hayden Campbell for his contributions to the papers that form the basis of this thesis, and for the many discussions about formalizations and proofs. I thank my co-authors and fellow PhD students Marcel Blöcher, Matthias Krebs, Johannes Krude, Katharina Keller, and Artur Sterz. I thank all members of the Software Technology Group especially my long-time office colleagues Ragnar Mogk, Pascal Weisenburger and Mirko Köhler for the numerous conversations, discussions and the occasional, very entertaining rants. Thanks to Gudrun Harris and Claudia Roßmann, who always had your back when it came to bureaucracy. I also thank former group member Guido Salvaneschi for his support at the beginning of my time as a PhD student.

I thank my family, especially my parents and my sister for always being there for me and believing in me. Last but not least, I like to thank my wife Christin and my little son Jonas. Thank you, Christin, for accompanying and supporting me all these years through all the ups and downs—it is finally done, but now for real. Thank you, Jonas, for always brightening up my day with your little smile. I love you.

Contents

Part I

Prologue

CHAPTER 1

Introduction

For more than a quarter of a century, the idea has existed to overcome the limitations imposed by the static nature of networks by providing programmability inside the network [Smi+96; FRZ13]. Until today, with a major focus on interoperability, new developments such as new network protocols usually have to undergo years of standardization [Ten+97; WGT98; TW07; Cal06] before being adopted by hardware vendors and being usable in practical deployments, which overall slows down innovation in the field of networking. The functionality of most network devices is strongly linked to the underlying hardware and the available functionality is solely dictated by hardware vendors. Implementations are usually proprietary, and network administrators have only limited configuration mechanisms available to adapt the functionality of the network device to their needs. As such, there has long been a desire to implement new network services or tailor packet processing functionality to the needs of applications running on top of the network, but the original approaches have not caught on.

For a little over a decade, the way how we can configure networks has changed significantly, and today this idea is finally becoming a reality. There is a shift towards more flexible platforms, which allow to specify the behavior of the network in software. These platforms are based on the idea of breaking the tight coupling between deciding where to send packets and actually forwarding packets [Yan+04; Lak+04], which results in a logically centralized control plane and a separate data plane.

Early efforts related to software-defined networking (SDN) focused on the control plane software [Cas+07; McK+08]. These approaches made it possible to write programs that determine how routes through the network are computed, load is balanced or how security policies are enforced. The data plane was modeled as a simple pipeline that operates on a fixed set of packet formats. In practice, it has been found that a fixed number of available packet headers that can be accessed is not sufficient [Bos+14]. To overcome the limitations imposed by a fixed data plane, there is a recent interest in allowing the functionality of the data plane itself to be specified as a program. The goal is to provide flexible mechanisms that allow arbitrary packet headers to be extracted and processed. This opens up a whole new range of applications since it allows network programmers to implement new network protocols, make more efficient use of hardware resources or even relocate application-level functionality into the network [Jin+17; Jin+18].

In particular, the P4 language is becoming the de facto standard for programming data planes. It enables the functionality of the data plane to be programmed in terms of declarative abstractions such as header types, packet parsers, match-action tables, and structured control flow that a compiler maps down to an underlying target device.

1.1 Problem Statement

Today, computer networks play a more important role than ever before, since they provide the communication fabric for nearly all modern software systems. Unfortunately networks are still programmed using low-level languages that lack basic safety guarantees. Unsurprisingly this results in networks being unreliable and remarkably insecure—e.g., the first step in a cyberattack often involves compromising a router or other network device [KG16; OC0+18].

While a number of P4's features were clearly inspired by designs found in modern languages, the central abstraction for representing packet data—header types—lacks basic safety guarantees. Header types are used to describe the structure of packet headers, which—as a first approximation—can be thought of as a record, with one entry for each header field. For example, the header type for an IPv4 packet would have a 4-bit version field, an 8-bit time-to-live field, two 32-bit fields for the source and destination addresses, and so on. According to the P4 language specification [P416], an instance of a header type may either be valid or invalid. If the instance is valid, then all operations reading or writing the header instance produce a defined value, but if it is invalid, then operations yield undefined results. In practice, programs that manipulate invalid headers can exhibit a variety of faults including dropping the packet when it should be forwarded or even leaking information from one packet to the next. In addition, such programs are also not portable, since their behavior can vary when executed on different targets.

The choice to model the semantics of header types in an unsafe way was intended to make the language easier to implement on high-speed routers, which have often limited amounts of memory. A typical P4 program might specify the behavior for several dozen different protocols, but any particular packet is likely to contain only a small handful of headers. Consequently, if the compiler only needs to represent the valid headers at run-time, then memory requirements can be reduced. However, while it may have benefits for language implementers, the design is a disaster for programmers. It repeats Hoare's "mistake" [Hoa09] and bakes an unsafe feature deep into the design of a language that has the potential to become the de facto standard in a multi-billion-dollar industry.

As programmable data planes become more prevalent and more complex applications are implemented inside the network, the more the risk of bugs increases. In the past, various verification tools have been developed to statically detect errors resulting from insecure access of header instances. We believe that basic safety guarantees should be part of the programming language instead. Modern programming languages offer features such as type systems, structured control flow, objects, modules, etc. that make it possible to express rich computations in terms of high-level abstractions rather than machine-level code. Increasingly, many languages also offer fundamental safety guarantees—e.g., well-typed programs do not go wrong [Mil78]—that make entire categories of programming errors simply impossible.

Type systems are a lightweight and compositional way to establish program properties. Types for individual components document assumptions about the components they rely upon as well as the guarantees they offer. However, the use of type systems in the realm of network programming has barely been investigated until now. For simple

properties, such as handing the access to uninitialized memory locations, no additional program annotations are necessary in contrast to existing verification tools.

The compositionality inherent to type systems, enables modular verification of programs. Existing verification tools are monolithic and do not provide any support for modular verification of programs, which was not a problem in the past, since P4 programs were also mostly monolithic and mostly still are. Although pre-processors are used to separate the program code for individual sections of the packet processing pipeline, P4 programs still contain a complete description of the processing pipeline. Increasingly, however, attempts are being made to make data plane programs modular [Gao+20; Son+20], with the aim of creating libraries for programming data planes from which data plane programs can be composed and that functionality can be extended in a modular fashion, such that programmers can describe the intended behavior in terms of high-level abstractions instead of low-level, platform-specific language constructs.

In summary, the P4 language is used to program critical infrastructure, although it lacks basic safety guarantees. Thus, with the steady proliferation of programmable data planes, the risk of network devices exhibiting unexpected errors is increasing. Future versions of P4 or new languages for programming data planes should therefore be equipped with suitable type systems that provide the programmer with necessary safety guarantees already at compile time.

1.2 State of the Art

In order to provide a better understanding of the proposed solution, we first provide an overview of the current state of network programming and network program verification.

Network Programming Languages

With OpenFlow [McK+08], the idea of a network operating system [Gud+08] emerged that provides abstractions for the resources of the underlying network in a manner similar to conventional operating systems providing access to system resources. The goal was to provide programmers with higher abstractions for configuring networks so that network policies no longer had to be realized by configuring individual devices using low-level configuration mechanisms. On this basis, a multitude of programming languages emerged, which raised the level of abstraction even further.

Various declarative programming paradigms such as logic programming [Hin+09], functional reactive programming [VH11; VKF12] or tierless programming [Nel+14] were used, with the goal of avoiding errors through complex interactions between packet-handling rules, for example, when composing network policies [Mon+13], for stateful packet processing [Ara+16], consistent network policy updates [McC+16] or to enable formal reasoning [And+14; Kim+15].

In contrast, data planes are currently programmed primarily with low-level languages [Bos+14; Bro19]. Both μP4 [Son+20] and Lyra [Gao+20] aim to provide high-level abstractions for the data plane, similar to the high-level languages for the control plane, to make code portable between architectures and to enable composition. However, neither approach is suitable for specifying and verifying correctness properties for data planes.

Verification of Data Plane Programs

Until now, only dedicated verification tools have been used for the verification of data planes. Various verification tools have been developed in the recent years to statically check the correctness of data planes, using a number of different verification techniques. p4v [Liu+18] applies classical techniques based on predicate transformer semantics. Vera [Sto+18] and Assert-P4 [Fre+18; Nev+18] are symbolic execution engines for P4. p4pktgen [Nöt+18] uses symbolic execution to automatically generate test cases for P4 programs. The bf4 tool [Dum+20] follows the approach pioneered in 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. P4K [KR18] provides a formal semantics of P4 in the K framework [RŞ10] and thus, can make use of the verification tools provided by the K framework. P4AIG [Nou+19] statically verifies programmable data planes using sequential circuits—a hardware verification technique. In contrast, P4RL [Shu+19] uses a dynamic approach—fuzz testing—for the verification of P4 programs.

Having to resort to a separate verification tool increases the entry barrier for verification compared to when verification mechanisms are part of the programming language itself, as it is the case with type systems. In addition, none of the previously mentioned verification tools allows to modularly reason about data planes.

Type Systems for Networks

Surprisingly, the use of type systems in the context of network programming has hardly been investigated so far, although the use of strongly typed languages for programming packet processing systems was already investigated in the 1990s in the context of Active Networks [Hic+98; Ale+98], with the aim of providing the programmer with static correctness guarantees for programs executed inside the network.

PacLang [ESM04] is an imperative, concurrent language for expressing packet processing applications for Network Processors [All+03]. It uses a linear type system to ensure that a packet is never processed by multiple threads simultaneously and can only be processed by multiple threads if it was transferred between threads beforehand.

Muthukrishnan et al. proposed strongly typed networking [Mut+10], where packets carry additional type information that describes how the receiver of the data will interpret it. Network entities can then reject traffic based on this information if not enough context is provided to correctly carry out its functionality.

In a position paper [GGW15], Gaboardi et al. propose the use of a simple type system to ensure that rules installed by an SDN controller are compatible with the underlying match-action table. For other properties they recognize the need for refinement type systems or dependent type systems without providing a concrete solution. However, all the above systems use simple type systems that are not suitable for providing static correctness guarantees for data plane programs written in P4.

1.3 The Thesis in a Nutshell

For programming data planes, we lack programming languages that provide programmers even with basic safety guarantees. The development of a variety of verification tools clearly shows that there is a need to statically verify the correctness of data planes. The goal of this thesis is therefore to develop suitable type systems that are able to verify

the correctness of data planes, starting from basic security properties up to application-specific properties.

Thesis Statement

> *Type systems are well suited to equip data plane programming languages with safety guarantees, which can be used to verify a wide range of safety properties.*

To validate this thesis, we develop two type systems for P4, currently the most widely used programming language for data planes. Our first contribution is SafeP4, a domain-specific language (DSL) that models the main abstractions of P4 and comes with a static type system that guarantees header validity. While header validity is a common cause of safety bugs, it is not sufficient to match the expressive power of full-fledged verification tools. As a secondary contribution, we therefore develop Π4, a dependently-typed version of P4, which allows us to express and verify rich network properties while retaining the compositionality inherent to type systems.

1.3.1 SafeP4

To address header validity bugs, we design SafeP4, a domain-specific language for programmable data planes, which has a static type system that guarantees that all headers read or written by the program are guaranteed to be valid. SafeP4 models all the essential features of the P4 language, but prunes away unnecessary complexity, which results in a minimal calculus that is easy to reason about, but can still express numerous real-world data plane programs. The type system has an expressive algebra of so-called *header types*, by means of which it is possible to describe precisely which headers are valid on a certain program path. The type checker can automatically reject all programs that attempt to read or write headers that are not valid on all program paths leading to a certain program statement.

The challenge in designing the type system is that header validity is actually a dynamic property, because the set of valid headers can be modified at runtime. Data plane programs do not fully describe the functionality of the data plane. Instead, the functionality of the data plane is largely determined by the control plane, in the form of packet processing rules used to populate match action tables. Depending on the installed rules and the contents of the network packet that is processed, packet headers can thus be added or removed. To still be able to achieve static safety, SafeP4 uses a path-sensitive type system that incorporates information from conditional statements, forwarding tables and the control plane to precisely track header validity.

The type system was implemented in the form of P4Check, a static analysis tool that allows P4 programs to be analyzed without having to re-implement them using the DSL provided by SafeP4. Our evaluation shows that SafeP4's type system is capable of discovering numerous unsafe header accesses in real-world P4 programs, both from academia and the industry.

1.3.2 Π4

With the design of Π4, we pursue two goals: on the one hand, we want to close the gap between SafeP4 and full-fledged data plane verification tools, and on the other

hand, we want to use the compositionality inherent to type systems to enable modular verification of data planes.

Even though validity bugs are a common source of errors, there are a variety of other safety properties that SafeP4 cannot express. The main reason is that SafeP4's type system is not able to capture values of individual fields, which is an essential requirement for a multitude of safety properties, since network packets often rely on so-called type-length-value encoding where the first bits determine the type, length, and structure of subsequent bits.

For this reason, we resort to a more powerful typing discipline in Π4's type system—dependent types—which allow us to define types based on program values. Π4 extends SafeP4's header types to *heap types*, which not only capture which instances are valid, but also the shape of header instances and the incoming and outgoing network packet down to the bit-level. Overall Π4 features a combination of dependent function types, dependent pairs, refinement types, union types and explicit substitutions, which enables precise typing in the presence of domain-specific features that combine packet serialization and de-serialization with imperative control flow.

In the design of Π4's type system, we manage to balance the tradeoffs between expressiveness and decidability. By encoding types into the effectively propositional fragment of first-order logic over fixed-width bit vectors, we achieve automated subtyping and equivalence checks. This relieves the programmer from the burden of writing manual proofs, which is often the case for dependent type systems.

1.4 Contributions

The main contribution of this thesis is to demonstrate that type systems are well-suited to verify the correctness of data planes and that it is not necessary to sacrifice safety guarantees in favor of a simpler implementation. On the contrary, P4's constrained programming model lends itself perfectly to the use of expressive typing disciplines. As part of the design, implementation and evaluation of SafeP4 and Π4, this thesis makes the following individual contributions.

- We formalize SafeP4, a core calculus that models the core features of P4 and is equipped with a type system based on *header types*—a limited form of regular types—that statically guarantees header validity and prove its type system sound.

- With the formalization of SafeP4's type system, we show how the dynamic behavior of the control plane can be approximated and thus the dynamic property of *header validity* can be made statically verifiable.

- We formalize Π4, a dependently-typed version of the P4 language that combines dependent function types with *heap types*—a combination of refinement types, a limited form of regular types and explicit substitutions. We show how precise typing can be enabled in the presence of domain-specific language features that combine packet serialization and de-serialization with imperative control flow. Again, we prove soundness for Π4's type system and in addition, we prove that type checking is decidable.

- With our chomp operator that computes the type that remains after extracting bits from a packet buffer, we demonstrate how we can compute derivatives of regular types in the presence of dependent types.

- We provide an implementation of Π4, which provides automated subtyping and equivalence checks by encoding *heap types* into a decidable theory of first-order logic.
- We provide a classification of which of P4's language constructs are susceptible to validity bugs and exemplify how to avoid corresponding bugs.
- We evaluate SafeP4 by checking a set of open-source P4 programs, both from industry and academia, against SafeP4's typing rules. Our results confirm that header validity bugs are a common source of errors, more than 70% of programs examined contained at least one such bug. Our evaluation further shows that fixing header validity bugs usually entails only low overhead for the programmer.
- We evaluate the expressiveness of Π4 with a set of case studies, which demonstrate that Π4 is capable of expressing and checking properties that were also addressed by other data plane verification tools. In an additional case study, we further demonstrate how Π4 can be used to reason about modular data plane programs. Finally, we evaluate the runtime performance of Π4's type checker on a set of open-source P4 programs.

1.5 List of Publications

The contributions of this thesis appeared previously in the following publications at peer-reviewed conferences. Parts of them are used verbatim.

[Eic+19] Matthias Eichholz et al. "How to Avoid Making a Billion-Dollar Mistake: Type-Safe Data Plane Programming with SafeP4". In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 12:1–12:28. DOI: 10.4230/LIPIcs.ECOOP.2019.12.

[Eic+22] Matthias Eichholz et al. "Dependently-Typed Data Plane Programming". In: *Proceedings of the ACM on Programming Languages* 6.POPL (2022). DOI: 10.1145/3498701.

I furthermore (co)authored the following peer-reviewed conference publications and workshop papers, which are not part of this thesis.

[Eic16] Matthias Eichholz. "Language Support for Verifiable SDNs". In: *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. SPLASH Companion 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 9–11.

[ESM18] Matthias Eichholz, Guido Salvaneschi, and Mira Mezini. "Towards Safe Modular Composition of Network Functions". In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. Programming'18 Companion. Nice, France: Association for Computing Machinery, 2018, pp. 81–86.

[Blö+19] Marcel Blöcher et al. "GRASS: Generic Reactive Application-Specific Scheduling". In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 21–30.

[Kru+19a] Johannes Krude et al. "Online Reprogrammable Multi Tenant Switches". In: *Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*. ENCP '19. Orlando, FL, USA: Association for Computing Machinery, 2019, pp. 1–8.

[Kru+19b] Johannes Krude et al. "Optimizing Data Plane Programs for the Network". In: *Proceedings of the ACM SIGCOMM 2019 Workshop on Networking and Programming Languages*. NetPL'19. Beijing, China: Association for Computing Machinery, 2019, p. 1.

[Ste+21] Artur Sterz et al. "ReactiFi: Reactive Programming of Wi-Fi Firmware on Mobile Devices". In: *The Art, Science, and Engineering of Programming* 5 (2021), p. 4.

1.6 Structure of the Thesis

Although the work presented in this thesis is my own, research is still ultimately a collaborative process and I have been fortunate to work with several co-authors, who have helped to shape and improve the texts, concepts and solutions presented. In the following I will state which parts of this thesis are based on which publication. In addition, I will mention which contributions were mainly made by my co-authors.

Chapter 2 provides background on software-defined networking and network programmability. We summarize the central language abstractions of P4 and show what differences exist between the two main language versions $P4_{14}$ and $P4_{16}$, which are mainly of syntactic nature. Chapter 3 classifies common bugs that arise in P4 programs caused by accessing invalid header instances and shows examples of how to fix these bugs. The chapter is based on [Eic+19] where the classification was contributed by Eric Hayden Campbell. Chapter 4 formalizes SafeP4 and proves soundness for its type system. It is based on the work published at ECOOP [Eic+19]. Chapter 5 provides the formalization of Π4, including the definition of our chomp operator and proves Π4's type system to be safe and decidable. This chapter is based on the work published at POPL [Eic+22]. Chapter 6 describes the central insights underlying the implementation of Π4, including the algorithmic typing, the SMT encoding and necessary optimizations. This chapter is in parts also based on the work published at POPL [Eic+22]. Chapter 7 provides evaluation results that show that bugs due to invalid header references arise in various industrial and academic data plane programs and is based on the work published at ECOOP [Eic+22]. The implementation of the prototype was done by Nate Foster, since he had access to a proprietary frontend for the P4 language and the evaluation was mostly performed by Eric Hayden Campbell. Chapter 8 is also based on [Eic+22] and describes several case studies that show that Π4 is well-suited to express a wide range of network properties also addressed by verification efforts from the networking community. Chapter 9 evaluates Π4's runtime performance when checking real P4 programs. Chapter 10 concludes this thesis and discusses future research directions.

CHAPTER 2

Background

This chapter introduces how programmability inside the network has emerged, leading to the current state of programmable data planes. We then show how the behavior of packet processing pipelines can be specified using the language abstractions provided by the P4 language.

2.1 Programmable Packet Processing

The functionality of network devices is based on a division of labor between two components, the control plane and the data plane. The control plane is responsible for deciding where to forward packets, for example by calculating routes, distributing load or enforcing security policies, while the data plane comprises specialized hardware capable of efficiently processing network packets at line rate. Until today, the majority of network devices exhibit a close tie between the control and data plane, as vendors control and closely coordinate both the hardware and software. This complicates a variety of network management tasks. For example, despite the distributed nature of network control software, the majority of these devices must be configured individually via vendor-specific configuration mechanisms—which usually differ between vendors and sometimes even between devices from the same vendor—making it difficult to debug network-wide configurations.

Moreover, the control software is usually limited to standardized network protocols to ensure interoperability between devices from different vendors. Customizing the behavior of network devices is only possible to a limited extent, but it is not readily possible to simply replace the software, for example, to implement new functions such as new network protocols. As a result, while network hardware continues to evolve, innovation at the control software level is slowed down. Before new developments are adopted by vendors, it usually takes several years for them to go through standardization processes. For example, the standardization of IPv6 started as early as 1998 [DH98] and was not completed until mid-2017 [DH17].

In order to accelerate innovations within the network, the idea of programmable networks therefore first emerged in the mid-1990s. With the growing success of the Internet, researchers wanted to address the limitation that implementing and testing new network protocols was not possible in real networks, but instead limited to small lab

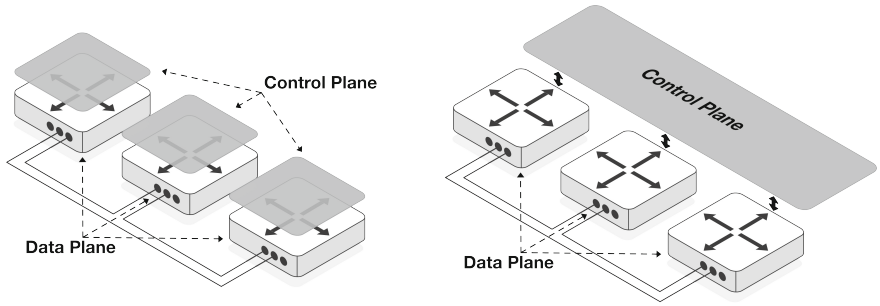

Figure 2.1: Control plane and data plane in traditional networks (left) in comparison to a logically centralized control plane used in SDN (right).

deployments or simulations. Active Networks [BCZ97; Ale+98; TW07] therefore aimed at making packet processing on network nodes adaptable, such that new protocols could be implemented with little effort and packet processing could even be adapted to the requirements of individual applications. This was achieved by either providing network switches with a set of functions that a packet could execute at runtime as needed or by sending the code to be executed directly with the network packets. With the Packet Language for Active Networks (PLAN) [Hic+98], there were even efforts to establish safety guarantees as a fundamental part of the language by means of static and dynamic typing. For many the proposed approaches were too radical and as a result the vision of programmable networks did not catch on at the time.

The topic of programmable networks first gained considerable attention in the late 2000s with the advent of software-defined networks and, in particular, the development of OpenFlow [McK+08]. OpenFlow managed to find a balance between the vision of programmable networks and practical deployability by building on existing hardware, which lead to a fast adoption both in academia and industry. As visualized in Figure 2.1, SDN is based on a fundamental change in network architecture and builds on two key ideas developed in previous years: the separation between the control and data plane and the consolidation of the previously distributed control plane into a logically centralized control plane. By removing the tight coupling between the control plane and the data plane, it became possible to program the control plane software independently. Henceforth, the control plane software runs on general-purpose machines while the data plane consists of dumb network switches that interact with the control plane via well-defined APIs such as OpenFlow. At the heart of an OpenFlow switch are *match-action tables*, whose entries form a set of rules that determine how network packets are processed. Each rule consists of (1) a *pattern* that identifies a set of packets based on a fixed set of packet headers, (2) a list of *actions* that are applied to packets for which the pattern matches as well as (3) a *priority* for disambiguating rules and (4) *counters* for tracking the total size of packets processed using the respective rule. Depending on the set of installed rules, the switch can fulfill different roles, for example that of a router, a firewall or a load balancer. Via a dedicated communication channel between the switch and the controller, the controller can install new rules at runtime, but switches can also issue packets to be processed on the controller, for example, in case that none of the installed rules matches.

The development of OpenFlow was closely followed by the idea of a network operating system, which provides the programmer with a unified interface that abstracts over the resources of the network (e.g. switches), thus enabling network-wide policies

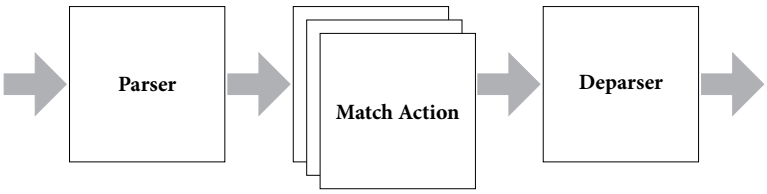

Figure 2.2: P4's abstract forwarding model.

and making low-level, device-specific configurations obsolete. The task of network operating system was taken over by a variety of controller platforms that were developed in the following period. Furthermore, a multitude of programming languages based on declarative programming paradigms such as logic programming [Hin+09], functional reactive programming [VH11; VKF12] or tierless programming [Nel+14] emerged, which raised the level of abstraction even further. The goal of these languages was to avoid errors through complex interactions between packet-handling rules, for example, when composing network policies [Mon+13], for stateful packet processing [Ara+16], consistent network policy updates [McC+16] or to enable formal reasoning [And+14; Kim+15].

As OpenFlow became more widely adopted, it became apparent that matching on a set of predefined packet headers is not sufficient in practice to satisfy all application domains, which initially led to the standard being extended several times to provide new packet headers and eventually to efforts to make the data plane programmable as well. While packet processing is still based on match-action tables, a central component in these systems is a programmable parser, which allows defining new packet headers as well as the order in which these are read from the packet being processed. This opens up a variety of new applications, from new network protocols to more efficient use of hardware resources to offloading application logic into the network [Jin+17; Jin+18]. The P4 language [Bos+14] has emerged as the de facto standard for programming data planes.

2.2 The P4 Language

P4 is a domain-specific language for specifying the behavior of network data planes. It provides declarative abstractions to describe how network devices process packets, i.e., arbitrary sequence of bits that can be divided into (1) a set of pre-determined headers that determine how the packet will be forwarded through the network, and (2) a payload that encodes application-level data. P4 is designed to be protocol-independent, which means that it handles packets with standard header formats (e.g., Ethernet, IP, TCP, etc.) as well as packets with custom header formats defined by the programmer.

By now, different kinds of devices can be programmed with P4, including PISA switches [Bos+13], FPGAs [Iba+19; Wan+17] or software devices, e.g., eBPF [Høi+18]. Because individual devices can differ significantly in their internal structure–for example, a programmable network interface card uses a different processing pipeline than a switch—P4 uses the concept of architectures to abstract over hardware details. A P4 architecture constitutes the programming model. It determines the programmable blocks and which functions are available to the programmer to interact with the hardware. The architecture is specified by the hardware vendor, which can either be a standard architecture such as the *Portable Switch Architecture* (PSA) [PSA16], or it can be a custom

```
header ethernet_t {
  bit<48> dstAddr;
  bit<48> srcAddr;
  bit<16> etherType;
}
```

Figure 2.3: Header type declarations in $P4_{16}$.

```
1 struct headers {
2   ethernet_t ethernet;
3   ethernet_t inner_ethernet;
4   vlan_t[2] vlan;
5 }
```

Figure 2.4: Declaration of header instances in $P4_{16}$.

one. The PSA, for example, provides for a total of six programmable blocks, one parser, one match action stage and one deparser each for both the ingress and egress processing stages.

Simplified, P4 programs follow the abstract forwarding model shown in Figure 2.2, which generalizes how packets are processed in different forwarding devices. A P4 program first *parses* the headers in the input packet into a typed representation, which together with various metadata forms a global, per-packet state that is shared between the pipeline stages. Next, it uses a *match-action pipeline* to compute a transformation on those headers—e.g., modifying fields, adding headers, or removing them. Finally, a *deparser* serializes the headers back into a packet, which can be output to the next device. The rest of this section describes P4's typed representation, how the parsers, and deparsers convert between packets and this typed representation, and how control flows through the match-action pipeline.

2.2.1 Header Types and Header Instances

The packet headers on which a P4 program operates are referred to as *header instances* while *header types* specify the internal representation of packet data. For example, the code snippet shown in Figure 2.3 declares a header type (`ethernet_t`) for the Ethernet header with fields for the destination (`dstAddr`) and source (`srcAddr`) addresses and the so-called *EtherType* (`etherType`). Each field in a header type declaration is annotated with a type. The type `bit<W>` indicates that the field is a bit vector of size W.

The code snippet in Figure 2.4 shows how headers are instantiated. Lines 2 and 3 define two header instances `ethernet` and `inner_ethernet` that share the same structure defined by header type `ethernet_t`. As shown, multiple header instances can be derived from the same header type, but since header instances in P4 are globally scoped, instance names must be unique. Besides header instance declarations, P4 also allows defining fixed-size arrays of headers with the same type, so-called *header stacks*. On Line 4 in Figure 2.4, a header stack `vlan` that can hold two instances of type `vlan_t` is defined. Both header stack instances can be referenced using the notation `vlan[0]` and `vlan[1]` respectively. They both are equivalent to a non-stack header instance.

In addition to fields that have been explicitly declared as part of a header type declaration, all header instances automatically include an additional Boolean *validity* field. Fields with Boolean type are implemented as bit vectors of size one, which is why we speak of the validity bit of a header instance. If the value of the field is *true*, we say that the header instance is *valid*, otherwise it is *invalid*. By default, header instances are invalid. The validity bit can be manipulated by the parser or by explicitly marking a header instance as valid or invalid. Each header stack instance has its own validity bit and adding or removing elements from the stack does only change the number of *valid* headers in the stack but not the number of headers.

2.2.2 Metadata

Metadata is per-packet state that is generated during the execution of a P4 program. Metadata behaves like header instances, i.e., individual fields can be read and written, but metadata has no validity bit and is therefore always valid. A distinction is made between user-defined metadata and intrinsic metadata. The structure of user-defined metadata is determined by the programmer and can be thought of as a set of temporary variables that can be modified by the program. In contrast, intrinsic metadata is provided by the architecture and holds information about the incoming packet such as the input port a packet arrives at. In addition, intrinsic metadata performs the task of a control register, i.e. by setting certain fields the programmer determines how the hardware processes the packet. For example, in the architecture used by $P4_{14}$, the programmer determines on which port the packet will be forwarded by setting the `egress_spec` field.

2.2.3 Parsers

A P4 parser specifies the order in which header instances are extracted from the input packet using a simple abstraction based on finite state machines. Figure 2.5 shows an implementation of a parser for three common headers Ethernet, VLAN and IPv4. Every parser contains at least one start state named *start* and two final states, *accept* and *reject*. The final state *accept* indicates successful parsing while the state *reject* indicates a parsing failure. For example, the parser in Figure 2.5 transitions to the acceptance state if none of the explicitly specified cases matches (Lines 7 and 14) or after parsing the IPv4 header (Line 19). When extracting into a header instance, bits are copied from the input packet into the header instance, which is as a result marked as valid. The `select` statement is P4's version of a switch statement and allows to transition to a different parser state based on the contents of a previously extracted header.

The parser first extracts the instance `ethernet`, optionally followed by a `vlan` instance, or an `ipv4` instance, or both. Figure 2.6 depicts the headers accepted by the parser. If the EtherType field of the Ethernet header contains any value except 0x0800 or 0x8100, no other headers except for Ethernet are extracted from the packet. If the EtherType is 0x0800, the IPv4 header is also be extracted. Similarly, if the EtherType is 0x8100, the VLAN header is extracted instead. In this case, the value of the EtherType field of the VLAN header determines whether the IPv4 header is additionally extracted or not. The corresponding parse graph is shown in Figure 2.7, where the final state on the right represents the *accept* state. When the parser reaches the *accept* state, processing transitions into the *ingress* pipeline, which begins the match-action processing.

```
parser Parser(packet_in packet, out headers hdr, ...) {
  state start {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
        0x0800: parse_ipv4;
        0x8100: parse_vlan;
        default: accept;
      }
  }
  state parse_vlan {
    packet.extract(hdr.vlan);
    transition select(hdr.vlan.etherType) {
      0x0800: parse_ipv4;
      default: accept;
    }
  }
  state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition accept;
  }
}
```

Figure 2.5: P4 code implementing a parser in $P4_{16}$.

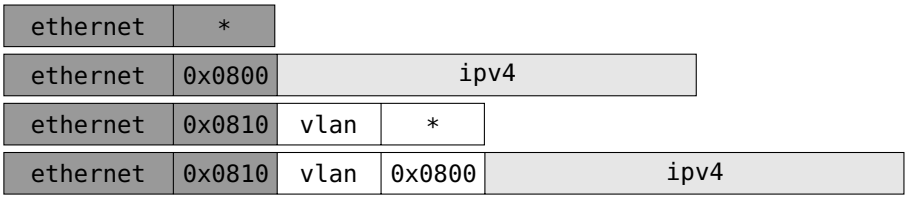

Figure 2.6: Headers accepted by the parsers in Figures 2.5 and 2.13.

2.2.4 Tables and Actions

The bulk of the processing for each packet in a P4 program is performed using *match-action tables*, a central data structure the match-action pipeline relies on. Match-action tables encode conditional processing, more specifically, the table first looks up the values being tested against a list of possible entries, and then executes a further piece of code depending on which entry (if any) matched. However, unlike standard conditionals, the entries in a match-action table are not known at compile-time. Rather, they are inserted and removed at run-time by the control plane, which may be logically centralized (as in a software-defined network), or it may operate as a distributed protocol (as in a conventional network).

As shown in Figure 2.8, a table is defined in terms of (1) the data it reads to determine a matching entry (if any) (2) the *actions* it may execute, and (3) an optional *default action* it executes if no matching entry is found. In $P4_{16}$ the data read by the table is specified

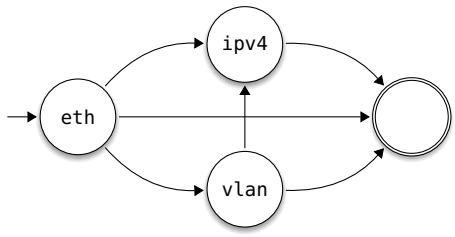

Figure 2.7: State machine of the parsers described by code in Figures 2.5 and 2.13.

```
1  table forward {
2    key = {
3      hdr.ipv4.isValid(): exact;
4      hdr.vlan.isValid(): exact;
5      hdr.ipv4.dstAddr: ternary;
6    }
7    actions = {
8      nop;
9      next_hop;
10     remove;
11   }
12   default_action = nop;
13 }
```

Figure 2.8: Table declaration in $P4_{16}$. `forward` reads the validity of the `ipv4` and `vlan` header instances and the `dstAddr` field of the `ipv4` header instance, and calls one of its actions: `nop`, `next_hop`, or `remove`.

using the keyword `key`. Table `forward` shown in Figure 2.8 reads the validity of the `ipv4` and `vlan` header instances and the `dstAddr` field of the `ipv4` header instance.

A table also specifies the *match-kind* that describes how each header field should match with the patterns provided by the control plane. The $P4_{16}$ core library provides three predefined match-kinds, *exact*, *ternary* and *lpm*. An *exact* match requires the bits in the packet be exactly equivalent to the bits in the controller-installed pattern. A *ternary* match allows wildcards in arbitrary positions, so the controller-installed pattern `0*` would match bit sequences `00` and `01`. A *longest prefix* match (*lpm*) selects the table entry where the largest number of leading bits of the specified header field match those in the table entry. For example, let us assume that a table performs a longest prefix match on the IPv4 destination address and the table contains two entries with addresses `192.168.178.80/28` and `192.168.178.0/24`. Both IP address ranges each contain the address `192.168.178.88`, but due to the longer matching prefix, the table entry with the address `192.168.178.80/28` is selected for a packet containing the destination address `192.168.178.88`.

The behavior of a table depends on the entries installed at run-time by the control-plane. Each table entry contains a match pattern, an action, and action data. Intuitively, the match pattern specifies the bits that should be used to match values, the action is the name of a pre-defined function (such as the ones in Figure 2.10), and the action data are the arguments to that function. Operationally, to process a packet, a table first

Pattern			Action	
ipv4	vlan	ipv4.dstAddr	**Name**	**Data**
1	0	10.0.0.*	next_hop	d,p
0	1	*	remove	

Figure 2.9: Runtime contents of `forward`.

```
action next_hop(bit<48> dst, bit<9> port) {
  hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
  hdr.ethernet.dstAddr = dst;
  hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
  standard_meta.egress_spec = port;
}
```

```
action remove() {
  hdr.ethernet.etherType = hdr.vlan.etherType;
  hdr.vlan.setInvalid();
}
```

Figure 2.10: P4 actions.

scans its entries to locate the first matching entry. If such a matching entry is found, the packet is said to "hit" in the table, and the associated action is executed. Otherwise, if no matching entry is found, the packet is said to "miss" in the table, and the *default action* (which is a no-op if unspecified) is executed.

Figure 2.9 provides an example of how the entries of the `forward` table populated by the control plane might look light. The first rule tests whether `ipv4` is valid, `vlan` is invalid, and the first 24 bits of `ipv4.dstAddr` equal `10.0.0`, and then applies `next_hop` with arguments d and p, which stand for destination address and port. The second rule checks that `ipv4` is invalid, then that `vlan` is valid, and skips evaluating the value of `ipv4.dstAddr` (since it is wildcarded), to finally apply the `remove` action.

Actions are functions containing sequences of primitive commands that perform operations such as adding and removing headers, assigning values to fields, etc. For example, Figure 2.10 depicts two actions `next_hop` and `remove`. The `next_hop` action first updates the Ethernet source with the current Ethernet destination, then updates the Ethernet destination with action data from the controller, then decrements the IPv4 TTL (time-to-live) field and finally sets the outgoing port, also with data provided by the controller. The `remove` action copies the data contained in the EtherType field from the VLAN header instance to the Ethernet header instance and invalidates the VLAN header.

2.2.5 Control

In $P4_{16}$, tables and actions are declared as part of control blocks. In addition, control blocks define in which order and under which conditions packet headers are transformed by match action tables. The body (`apply`) of a control block is an imperative program that uses standard control flow constructs to describe a pipeline of match action tables. The *ingress* control block usually begins to execute as soon as the parser completes. The

```
1 control Ingress(inout headers hdr, ...) {
2   // table and declarations
3   ...
4   apply {
5     if(hdr.ipv4.isValid() || hdr.vlan.isValid()) {
6       forward.apply();
7     }
8   }
9 }
```

Figure 2.11: Ingress control block in $P4_{16}$.

```
control Deparser(packet_out packet, in headers hdr) {
  packet.emit(hdr.ethernet);
  packet.emit(hdr.vlan);
  packet.emit(hdr.ipv4);
}
```

Figure 2.12: $P4_{16}$ deparser.

ingress control shown in Figure 2.11 conditionally executes the table `forward` (Line 6) if one of IPv4 or VLAN is valid (Line 5).

2.2.6 Deparser

The deparser reassembles the final output packet after all processing has been done by serializing each valid header instance in some order. In $P4_{16}$, a control block is used to describe the deparsed headers and their order as shown in Figure 2.12. In the example, the deparser produces a packet with Ethernet, VLAN, and IPv4, in that order. Every emit statement contains an implicit validity check, so headers are only emitted if they are actually valid, otherwise emit behaves like a no-op instruction.

2.2.7 Externs

In addition to the core language, P4 provides so-called *externs*, a foreign-function interface into the hardware. Externs allow P4 programs to use functionality that cannot be expressed with P4 itself. P4 programs can interact with externs via well-defined APIs and change their internal state, but the internal behavior of an extern itself is fixed and dictated by the hardware. Which externs are available depends on the architecture of the hardware. Examples for externs are checksum units, or stateful elements such as counters, meters and registers that allow to save state between packets.

2.2.8 P4 Language Versions

There are two major versions of the P4 language, $P4_{14}$ and the newer revision $P4_{16}$. With $P4_{16}$, the language designers have attempted to reduce the complexity of the language into a core language that provides fewer distinguished language constructs. In

```
1  parser start {
2    extract(ethernet);
3    return select(latest.etherType) {
4      0x0800: parse_ipv4;
5      0x8100: parse_vlan;
6      default: ingress;
7    }
8  }
9  parser parse_vlan {
10   extract(vlan);
11   return select(latest.etherType) {
12     0x0800: parse_ipv4;
13     default: ingress;
14   }
15 }
16 parser parse_ipv4 {
17   extract(ipv4);
18   return ingress;
19 }
```

Figure 2.13: P4 code implementing a parser in $P4_{14}$.

explaining the central language abstractions, we so far concentrated on $P4_{16}$, however, since Chapters 3 and 4 refer to language version 14, we will now briefly point out the main syntactic differences.

Figure 2.13 shows the same parser as Figure 2.5, but this time implemented in $P4_{14}$. Parser states are specified using the keyword `parser` and parser states do not transition to a next state but rather return the next parser to be executed. Accordingly, in $P4_{14}$ the end of the parsing phase is not expressed by a transition to an acceptance state, but by the parser returning as the next state the name of a control block to be executed afterwards. For example, Lines 6, 13 and 18 in Figure 2.13 indicate that after the parsing phase, the *ingress* control should be executed.

The code in Figure 2.14 shows how the table from Figure 2.8 is declared in $P4_{14}$, which is overall analogous to $P4_{16}$. The major syntactic difference is that match keys are specified using the keyword `reads`. $P4_{14}$ also provides additional match-kinds such as *valid* for matching on the validity bit of header instances. Tables and actions are declared outside of control blocks, but otherwise control blocks are similar to $P4_{16}$ as shown in Figure 2.15.

In $P4_{14}$ the compiler automatically generated the deparser from the parser, and while the concept of externs was already present, stateful elements were part of the core language.

2.3 Chapter Summary

This chapter outlined how the idea of programmable networks started in the mid-1990s with Active Networks and how modern software-defined networks simplify network

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

Figure 2.14: Table declaration in $P4_{14}$.

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

Figure 2.15: Control block implemented in $P4_{14}$.

management by separating the network control plane from the data plane. While early efforts focused on the programmability of the control plane, more recently it is also possible to program the data plane, for which the P4 language has established itself as the de facto standard. The second part of the chapter provided a deeper insight into the P4 language. Starting from P4's abstract forwarding model consisting of the parser, the match action pipeline and the deparser, we showed how these can be programmed with the abstractions provided by P4.

CHAPTER 3

Common Header Validity Bugs

While a number of P4's features were inspired by designs found in modern languages, the central abstraction for representing packet data—header types—lacks basic safety guarantees. As discussed in the previous chapter, an instance of a header type may either be valid or invalid. According to the P4 language specification, if the instance is valid, then all operations produces a defined value, but if it is invalid, then reading or writing a field yields an undefined result.

In practice, this has serious consequences for P4 programmers, since they must be careful not to read or write invalid headers, because programs that manipulate invalid headers can exhibit a variety of faults including dropping the packet when it should be forwarded, or even leaking information from one packet to the next. In addition, such programs are also not portable, since their behavior can vary when executed on different targets.

In this chapter we present five categories of bugs found in open-source P4 programs that arise due to reading and writing invalid headers. To identify the bugs we surveyed a benchmark suite of 15 research and industrial P4 programs that are publicly available on GitHub and compile to the BMv2 [Net18] backend. We categorize bugs based on the following syntactic constructs: (1) parsers, (2) controls, (3) table reads, (4) table actions, and (5) default actions. For each bug, we also present a possible type-safe fix.

For this survey, we exclusively considered programs implemented in $P4_{14}$ because only few $P4_{16}$ programs were publicly available at that time, however, the same bug categories persist to exist in $P4_{16}$.

3.1 Parser Bugs

The first class of errors is caused by the parser being too conservative about dropping malformed packets, which increases the set of headers that may be invalid in the control pipeline. In most programs, the parser chooses which headers to extract based on the fields of previously-extracted headers using P4's *select* statement. Programmers often fail to handle packets falling through to the *default* case of these select statements.

Figure 3.1 illustrates this bug using an example from the codebase of the research tool NetHCF [Bai+18; Li+19]—a tool designed to combat TCP spoofing. After extracting the

```
1  /* Unsafe */
2  parser parse_ethernet {
3    extract(ethernet);
4    return select(latest.etherType) {
5      0x0800: parse_ipv4;
6      default: ingress;
7    }
8  }
9  parser parse_ipv4 {
10   extract(ipv4);
11   return select(latest.protocol) {
12     6: parse_tcp;
13     default: ingress;
14   }
15 }
16 parser parse_tcp {
17   extract(tcp);
18   return ingress;
19 }
```

```
1  control ingress {
2    if(tcp.syn == 1 and ...) {...}
3  }
```

Figure 3.1: Parser bug example in NetHCF: parser (top) and ingress control (bottom).

Ethernet and IPv4 headers, the parser handles TCP packets in `parse_ipv4` (Line 12) based on the IPv4 protocol field and redirects all other packets to the ingress control, exemplary shown in the bottom of Figure 3.1. However, the ingress control does not check whether TCP is valid before accessing `tcp.syn` to check whether it is equal to 1. This is unsafe since `tcp` is not guaranteed to be valid even though it is required to be valid in the ingress control.

A possible fix of this bug is to throw an error during parsing, to stop the processing of the current packet and drop it instead. In $P4_{14}$, this can be achieved by defining a parser exception (Line 2 in Figure 3.2) with a handler that drops packets (Line 3 in Figure 3.2). This handler protects the ingress control from having to handle unexpected packets as shown in Lines 10 and 18.

In $P4_{16}$ we have to resort to a different mechanism, since parser exceptions are no longer part of the language. Instead, we can use the `verify` statement, a simple form of error handling, as shown on Line 7 in Figure 3.3. If the boolean expression used as the first argument evaluates to false, the parser immediately transitions to the *reject* state, which causes parsing to terminate immediately and sets the error state of the parser to the error used as second argument (`Unsupported`). However, the actual runtime behavior of this implementation highly depends on the target it is executed on. It is up to the target's implementation whether a rejected packet is actually dropped or just an error flag is set, and therefore the code is not portable. In the latter case, to achieve an

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

Figure 3.2: Fixing parser bugs in $P4_{14}$ using parser exceptions.

equivalent runtime behavior to the $P4_{14}$ fix, we could check for an error in the ingress and manually drop the packet.

In general, however, this fix might not be the best solution, because it alters the original behavior of the program. However, without knowing the programmer's intention, it is generally not possible to automatically repair a program with undefined behavior.

3.2 Control Bugs

Another common bug occurs when a table is executed in a context in which the instances referenced by that table are not guaranteed to be valid. This bug can be seen in the source code of NetCache [Jin18; Jin+17], a system that uses P4 to implement a load-balancing cache. The parser reserves a specific port (8888) to handle special-purpose traffic. If UDP traffic arrives at port 8888, the parser extracts the NetCache-specific header `nc_hdr`. Otherwise, it performs standard L2 and L3 routing.

As shown in the top-left of Figure 3.4, in the ingress control, the `process_cache` control block is executed, which itself is shown in the bottom of Figure 3.4. The latter defines and applies table `check_cache_exist`, which reads field `nc_hdr.key` as part of the match key. However, it is never checked that `nc_hdr` is actually valid. The invocation of the `process_value` table (not shown) contains another instance of the same bug. As shown in the top-right of Figure 3.4, to fix these bugs, we can wrap the calls to `process_cache` and `process_value` in a conditional that checks the validity of the header `nc_hdr`. This ensures that `nc_hdr` is valid when `process_cache` refers to it.

```
/* Safe */
error { Unsupported }

parser Parser(packet_in packet, out headers hdr, ...) {
  state start {
    packet.extract(hdr.ethernet);
    verify(hdr.ethernet.etherType == 0x0800, error.
        Unsupported)
    transition select(hdr.ethernet.etherType) {
      0x0800: parse_ipv4;
      default: accept;
    }
  }
  state parse_ipv4 {
    packet.extract(hdr.ipv4);
    verify(hdr.ipv4.protocol == 6, error.Unsupported)
    transition select(hdr.ipv4.protocol) {
      6: parse_tcp;
      default: accept;
    }
  }
  ...
}
```

Figure 3.3: Fixing parser bugs in $P4_{16}$ using the `verify` statement.

3.3 Table Reads Bugs

A similar bug arises in programs that contain tables that first match on the validity of certain header instances before matching on the fields of those instances. The advantage of this approach is that multiple types of packets can be processed in a single table, which saves memory. However, if implemented incorrectly, this programming pattern can lead to a bug, in which the match key reads bits from a header that may not be valid. An example of this bug is exhibited by switch.p4, a "realistic production switch" developed by Barefoot Networks, meant to be used "as-is, or as a starting point for more advanced switches" [Kod15].

Table `port_vlan_mapping`, shown in the top of Figure 3.5, shows an archetypal example of a table reads bug. This table is invoked in a context where it is unknown which of the VLAN tags is valid, despite containing references to both `vlan_tag_[0]` and `vlan_tag_[1]` in the match key declaration. The references to header fields `vlan_tag_[i].vid` are guarded with keys that test the validity of `vlan_tag_[i]`, for $i = 0, 1$. However, as written, it is impossible for the control plane to install a rule that will always avoid reading the value of an invalid header. The first match will check whether instance `vlan_tag_[0]` is invalid, which is safe. However, the very next match will try to read the value of field `vlan_tag_[0].vid`, even when the instance is invalid. This attempt to access an invalid header results in undefined behavior, and

```
/* Unsafe */
control ingress {

  process_cache();
  process_value();

  apply(ipv4_route);
}
```

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

```
table check_cache_exist {
  reads { nc_hdr.key : exact }
  actions { ... }
}
control process_cache {
  apply(check_cache_exist);
}
```

Figure 3.4: Control bug example in NetCache (top-left), type-safe fix (top-right) and common code (bottom).

is therefore a bug. It is worthy to note that this code is not actually buggy on some targets—in particular, on targets where invalid headers are initialized with 0. However, 0-initialization is not prescribed by the language specification, and therefore this code is not portable across targets.

The naive solution to fix this bug is to refactor the table into four different tables (one for each combination of validity bits) and then check the validity of each header before the tables are invoked. While this fix is perfectly safe, it can result in a combinatorial blowup in the number of tables, which is clearly undesirable both for efficiency reasons and because it requires modifying the control plane. Fortunately, rather than factoring the table into four tables, we can replace the exact match-kinds with ternary match-kinds as shown in the bottom of Figure 3.5, which permit matching with wildcards. In particular, the control plane can install rules that match invalid instances using an all-wildcard patterns, which is safe. In order for this solution to be an actual fix, we also need to assume that the control plane is well-behaved—i.e. that it will install wildcards for the ternary matches whenever the header is invalid.

3.4 Table Action Bugs

Another common bug arises when distinct actions in a table require different (and possible mutually exclusive) headers to be valid. This can lead to two problems: (1) the control plane can populate the table with unsafe match-action rules, and (2) there may be no validity checks that we can add to the control to make all the actions type-check.

Table `fabric_ingress_dst_lkp` shown in the top of Figure 3.6 provides an example of this bug[1]. It reads the value of header field `fabric_hdr.dstDevice` and

[1]There are other actions in the real implementation, but these three actions demonstrate the core of the problem.

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

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

Figure 3.5: Table reads bug in switch.p4 (top) and type-safe fix (bottom).

then invokes one of several actions: (1) `term_cpu_packet` (2) `term_fabric_unicast_packet`, or (3) `term_fabric_multicast_packet`. These actions require that the headers (1) `fabric_hdr_cpu`, (2) `fabric_hdr_unicast`, and (3) `fabric_hdr_multicast` respectively are valid. However, the validity of these headers is mutually exclusive.

Since all three headers are mutually exclusive, there is no single context that makes this table safe. The only facility the table provides to determine which action should be called is `fabric_hdr.dstDevice`. However, the P4 program doesn't establish a relationship between the value of `fabric_hdr.dstDevice` and the validity of any of these three header instances. So, the behavior of this table is only well-defined when the input packets are well-formed, an unreasonable expectation for real switches, which may receive *any* sequence of bits "on the wire."

We fix this bug by including validity matches in the match key, as shown in the bottom of Figure 3.6. Similar to the fix presented in Section 3.3, this solution avoids combinatorial blowup and extensive control plane refactoring. Again, we need to make an assumption about the way the control plane will populate the table. Concretely, if an action a is only safe to execute if a header h is valid, and h is not necessarily valid when the table is applied, we assume that the control plane will only call a if h is matched as valid. For example, `fabric_hdr_cpu` is not known to be valid when (the fixed version of) `fabric_ingress_dst_lkp` is applied, so we assume that the control plane will only call action `term_cpu_packet` when `fabric_hdr_cpu` is matched as valid.

```
/* Unsafe */
table fabric_ingress_dst_lkp {
  reads {
    h.fabric_hdr.dstDevice: exact;
  }
  actions {
    term_cpu_packet;
    term_fabric_unicast_packet;
    term_fabric_multicast_packet;
  }
}
```

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

Figure 3.6: Table action bug in `switch.p4` (top) and type-safe fix (bottom).

3.5 Default Action Bugs

Default action bugs occur when the programmer incorrectly assumes that a table performs some action when a packet misses. The implementation of NetCache, which we introduced in Section 3.2, exhibits an example of this bug. The bug is shown in the top of Figure 3.7, where table `add_value_header_1` is expected to make the `nc_value_1` header valid, which is done in the `add_value_header_1_act` action. The control plane may refuse to add any rules to the table, which would cause all packets to miss, meaning that the `add_value_header_1_act` action would never be called and header `nc_value_1` may not be valid. To fix this error, as shown in the bottom of Figure 3.7, we simply set the default action for the table to `add_value_header_1_act`, which will force the table to add the header no matter what rules the controller installs.

3.6 Chapter Summary

In this chapter, we presented five of P4's language constructs that are potentially vulnerable to validity bugs and what possible bug fixes for these bugs might look like. We

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

```
/* Safe */
table add_value_header_1 {
  actions {
    add_value_header_1_act;
  }
  default_action = add_value_header_1_act();
}
```

Figure 3.7: Default action bug in NetCache: unsafe code missing a default action (top) and type-safe fix (bottom).

do not claim that our taxonomy is complete and, moreover, the proposed bug fixes may change the behavior of the original program, but this cannot be avoided without knowing the programmer's intentions.

Header validity bugs can be caused by the parser when packages that should not be considered are handled inadequately. Additionally, they can also occur in various places related to match action tables. Invalid headers can be accessed if headers a table matches on are not guaranteed to be valid in the respective context, but also if in such a case, the programmer wants to ensure validity by means of an additional validity match, but which is not implemented correctly. Furthermore, bugs can occur when actions require that certain—possibly mutually exclusive—headers are valid without the program making a connection between the validity of these headers and the reads-expressions of the table defining the actions, or when it is incorrectly assumed that a header is made valid by an action without any guarantee that this action will ever be executed.

Part II

Typed Data Plane Programming

CHAPTER 4

A Typing Discipline to Ensure Header Validity

While the way headers are represented in P4 has advantages for language implementers, the design is a disaster for programmers. As we saw in the previous chapter, a variety of subtle bugs can creep in at various points in a P4 program when invalid headers are accessed, which makes it challenging for programmers to write correct data plane programs. The fact that accessing invalid headers returns undefined values is comparable to the existence of null references in various general-purpose programming languages. Computer scientist Tony Hoare once called his invention of the null reference a billion-dollar mistake [Hoa09]. Computer networks provide the foundation for the majority of today's software systems. By embedding an insecure language feature deep into the design of a language that is about to become the standard in a billion-dollar industry, we are about to repeat Hoare's "mistake".

In this chapter, we look at how we can incorporate header safety guarantees into the language, thus enabling a correct-by-construction approach. We present SafeP4, a domain-specific language for programmable data planes in which all packet data is guaranteed to have a well-defined meaning and satisfy essential safety guarantees. We equip SafeP4 with a static type system that statically guarantees header validity, which relies on an expressive algebra of so-called *header types* that tracks validity information at a fine level of granularity. One of the main challenges here is that P4 programs do not completely specify the behavior of data planes. Part of the behavior is determined by the match-action rules installed by the control plane at runtime. Since the available rules can be altered at runtime, and as a consequence the set of valid headers can dynamically change, header validity becomes a dynamic program property. As a result, SafeP4's type system employs a form of path-sensitive reasoning that tracks dynamic information from conditional statements, routing tables and the control plane. We formalize the syntax and semantics of SafeP4 in a core calculus and prove that the type system is sound.

4.1 Design

We made four key design decisions when designing SafeP4. First, we represent only the core of P4, i.e., the features relevant to packet processing. Second, we simplify and generalize certain aspects to avoid unnecessary complications in the calculus. Third, we use a set of sets of headers to capture valid header instances per program path to avoid an overly restrictive type system and fourth, we assume that the control plane is well-behaved.

Core calculus In the design of SafeP4, we draw inspiration from Featherweight Java [IPW01], i.e., we model the essential features of P4, but prune away unnecessary complexity. The result is a minimal calculus that is easy to reason about, but can still express numerous real-world data plane programs. Our calculus is protocol-independent by allowing the programmer to specify the types of packet headers and their order in the bit stream. Also, SafeP4 mimics P4's use of tables to interface with the control-plan and decide which actions to execute at run-time.

Simplification and generalization We omit a number of constructs that are secondary to how packets are processed, including parser exceptions, counters, meters, etc., however, it would be relatively straightforward to add these to the calculus. Also, compared to P4, SafeP4 does not enforce a strict separation between the parsing phase and the control phase. Rather than unnecessarily complicating the syntax of SafeP4, we allow the syntactic objects that represent parsers and controls to be freely mixed. Similarly, we only enforce informally which primitive commands can be invoked within actions, e.g., field modifications but not conditionals.

We also deviate from P4 with regard to the *add* command. In $P4_{16}$, the analogous operation `setValid` (respectively `add_header` in $P4_{14}$) only modifies the validity bit of a header instances. Accordingly, accessing an added header instance returns a non-deterministic value if the header instance's fields have not been manually initialized beforehand. Instead of complicating our type system by additionally capturing whether header fields are defined, we define the semantics of our *add(h)* command to initialize each field of header instance h with a default value. We assume that in addition to our type constants there exists a function *init* that accepts a header type η and returns a header instance of type η with all fields set to their default value.

Last, we want to model the core behavior of both $P4_{14}$ and $P4_{16}$, however, both use different type systems and evaluation behaviors for expressions. We therefore abstract away expression typing and syntax variants by assuming that we are given a set of n-ary constants k that can represent values like 0 or `true` (0-ary), or unary and binary operators such as negation or logical conjunction, etc. We further assume that these operators are assigned sound types. With these features in hand, one can instantiate our type system over arbitrary constants.

Granularity of types To analyze the validity of header instances, our type system needs a way to capture which header instances are valid. Naively, we can keep track of a set of headers, which are guaranteed to be valid on all program paths and reject all programs that reference headers not included in this set. However, this coarse-grained approach would generate a large number of false positives.

For example, consider the parser implementation in Figure 4.1. The parser extracts an Ethernet header and then either parses a VLAN header or proceeds to the ingress.

```
1  parser start {
2    extract(ethernet);
3    return select(latest.etherType) {
4      0x8100 : parse_vlan;
5      default: ingress;
6    }
7  }
8  parser parse_vlan {
9    extract(vlan);
10   return ingress;
11 }
```

Figure 4.1: Parser extracting Ethernet and optionally VLAN.

```
1 control ingress {
2   if(valid(vlan)) {
3     modify_field(ethernet.etherType, vlan.etherType);
4     remove_header(vlan);
5   }
6 }
```

Figure 4.2: Ingress program

Hence, at the beginning of the ingress, only Ethernet is guaranteed to be valid. However, it is certainly safe to write an ingress program that references the VLAN header after checking that it is valid as shown in Figure 4.2.

To reflect this in the type system, we introduce a special construct $valid(h)\ c_1\ else\ c_2$, which executes c_1 if h is valid and c_2 otherwise. When we type check this command, following previous work on occurrence typing [TF10], we check c_1 with the additional fact that h is valid, and we check c_2 with the additional fact that h is not valid. Despite this enhancement, our type system would still be overly restrictive. For example, the parser shown in Figure 4.3, first extracts Ethernet and then boots into ingress or extracts IPv4. If IPv4 is valid, the parser optionally extracts TCP or UDP.

Now, suppose that we have an ingress control that defines a table `tcp_table` that refers to both IPv4 and TCP in its key expression and that only checks the validity of TCP before applying the table as shown in Figure 4.4. In general, the validity of TCP also implies the validity of IPv4, so it should be safe to apply table `tcp_table` after checking only the validity of TCP. However, with the representation of valid headers as a set, the type checker would still reject the program, because at the point where `tcp_table` is applied (Line 11), only Ethernet and TCP would be guaranteed to be valid. Instead, we would need to explicitly check that both IPv4 and TCP are valid.

We solve the problem by using a more fine-grained type representation—namely a set of sets of headers instead of a set—to capture header instances guaranteed to be valid and their dependencies. Each inner set contains all headers that might be valid at the current program point for some program path. For a given header reference to be safe, it must be a member of all possible sets of headers, i.e., it must be valid on all paths

```
parser start {
  extract(ethernet);
  return select(latest.etherType) {
    0x0800: parse_ipv4;
    default: ingress;
  }
}
parser parse_ipv4 {
  extract(ipv4);
  return select(latest.protocol) {
    0x6: parse_tcp;
    0x11: parse_udp;
    default: ingress;
  }
}
parser parse_tcp { ... }
parser parse_udp { ... }
```

Figure 4.3: Parser extracting Ethernet, optionally followed by IPv4 and if IPv4 is valid, optionally followed by TCP or UDP.

```
table tcp_table {
  reads {
    ipv4.dstAddr: lpm;
    tcp.port: exact;
  }
  actions { ... }
}

control ingress {
  if(valid(tcp)) {
    apply(tcp_table);
  }
}
```

Figure 4.4: Ingress control that applies table `tcp_table`, which reads headers `ipv4` and `tcp`.

through the program that reach the reference.

Control plane In the formalization of SafeP4, we model the control plane as a function that—given a table and the currently valid headers—returns the action to call and the (possibly empty) action data arguments. Also, a second function analyzes the table and produces for each action a set of valid headers that can be safely assumed valid when the entries are populated by the control plane. From the table declaration

and the header instances that can be assumed valid, based on the match-kinds, we can derive a list of match key expression that must be evaluated when the table is invoked. Together, these functions model the run-time interface between the switch and the controller. We assume that the control plane interface satisfies three simple correctness properties: (1) the control plane can safely install table entries that never read invalid headers, (2) the action data provided by the control plane has the types expected by the action, and (3) the control plane will only assume valid headers for an action that are valid for a given packet.

4.2 Syntax

The syntax of SafeP4 is shown in Figure 4.5. To lighten the notation, we write $\overline{x}$ as shorthand for a (possibly empty) sequence $x_1, \ldots, x_n$. A SafeP4 program consists of a sequence of declarations $\overline{d}$ and a command c. Declarations include tables, header types and header instances.

A table declaration $t(\overline{h}, \overline{(e, m)}, \overline{a})$ is defined in terms of a sequence of valid-match header instances $\overline{h}$, a sequence of match-key expressions $\overline{(e, m)}$, where e is an expression and m is the match-kind[1] used to match this expression and a sequence of actions $\overline{a}$. The notation $t.valids$ denotes the valid-match instances, $t.reads$ denotes the expressions, and $t.actions$ denotes the actions. For example, the following table declaration

$$
\begin{aligned}
&forward((ipv4, vlan), ((ipv4.dstAddr, ternary)), (\\
&\quad skip,\\
&\quad \lambda s, d.ethernet.srcAddr = s; ethernet.dstAddr = d,\\
&\quad ethernet.etherType = vlan.etherType; remove(vlan)\\
&))
\end{aligned}
$$

corresponds to the following P4 table, assuming action `nop` is—as the name suggests—a no-op instruction, action `next_hop` sets the Ethernet source and destination addresses to values provided by the control plane and action `remove` populates the EtherType field of the Ethernet header with the value contained in EtherType field of the VLAN header and ultimately removes the VLAN header.

```
1  table forward {
2    reads {
3      ipv4 : valid;
4      vlan : valid;
5      ipv4.dstAddr: ternary;
6    }
7    actions {
8      nop;
9      next_hop;
10     remove;
11   }
12 }
```

[1] In this work we focus on exact and ternary matches as well as matches on the validity bit.

Commands

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

Actions

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

Expressions

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

Declarations

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

Match Kinds

$m \in \{exact, ternary\}$

Constants

$k^n \in K$

Program

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

Values

$v \in V$

Header Types

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

Action Types

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

Expression Types

$\tau ::= \texttt{Bool}$
$\mid\ \overline{\tau} \to \tau$
$\mid\ \cdots$

Figure 4.5: Syntax of SafeP4

Actions are written as (uncurried) λ-abstractions. An action $\lambda\overline{x}.\ c$ declares a (possibly empty) sequence of parameters, drawn from a fresh set of names, which are in scope for the command c. The run-time arguments for actions (action data) are provided by the control plane. We artificially restrict the commands that can be called in the body of the action to addition, removal of headers, modification of header fields and sequence. These commands are identified with an asterisk in Figure 4.5.

Header type declarations describe the format of individual headers and are defined in terms of a name and a sequence of field declarations. The notation "$f : \tau$" indicates that field f has type τ. We let η range over header types. A header instance declaration assigns a name h to a header type η. The map $\mathcal{HT}$ encodes the (global) mapping between header instances and header types.

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

The only built-in expressions in SafeP4 are variables x and header fields, written $h.f$. We let v range over values and assume a collection of n-ary constant operators $k^n \in K$. For simplicity, we assume that every header referenced in an expression has a corresponding instance declaration. We also assume that header instance names h, header type names η, variable names x, and table names t are drawn from disjoint sets

$$
\begin{aligned}
[\![\Theta]\!] &\subseteq \mathcal{P}(Header) \\
[\![0]\!] &= \{\} \\
[\![1]\!] &= \{\{\}\} \\
[\![h]\!] &= \{\{h\}\} \\
[\![\Theta_1 \cdot \Theta_2]\!] &= [\![\Theta_1]\!] \bullet [\![\Theta_2]\!] \\
[\![\Theta_1 + \Theta_2]\!] &= [\![\Theta_1]\!] \cup [\![\Theta_2]\!]
\end{aligned}
$$

Figure 4.6: Semantics of header types.

of names H,E,V, and T respectively and that each name is declared only once.

SafeP4 provides three kinds of types, header types Θ, expression types τ and action types α. We assume that the set of expression types includes booleans (for conditionals) as well as tuples and function types.

4.3 Static Semantics

SafeP4 uses a path-sensitive analysis, coupled with occurrence typing to keep track of which headers are guaranteed to be available at any program point. The type system rejects programs that reference headers that might be uninitialized, thus, preventing all references to invalid headers.

Semantics of types A header type Θ represents a set of header instances that may be valid at the same time. The type 0 denotes the empty set. This type arises when there are unsatisfiable assumptions about which headers are valid. The type 1 denotes the empty singleton set of headers. It describes the initial state of the program where no headers are valid. The type h denotes a singleton set, $\{\{h\}\}$, i.e., states where only h is valid. The type $\Theta_1 \cdot \Theta_2$ denotes the set obtained by combining headers from Θ_1 and Θ_2, i.e., a product or concatenation ($H_1 \bullet H_2 = \{h_1 \cup h_2 \mid h_1 \in H_1 \wedge h_2 \in H_2\}$). Finally, the type $\Theta_1 + \Theta_2$ denotes the union of Θ_1 or Θ_2, which intuitively represents an alternative.

The semantics of header types, $[\![\Theta]\!]$, is defined by the equations in Figure 4.6. Intuitively, each subset represents one alternative set of headers that may be valid. For example, the header type $\texttt{eth} \cdot (\texttt{ipv4} + 1)$ denotes the set $\{\{\texttt{eth}, \texttt{ipv4}\}, \{\texttt{eth}\}\}$.

Typing judgment We use different typing judgments for command typing, expression typing and action typing. The typing judgment for commands has the form $\Gamma \vdash c : \Theta_1 \Rrightarrow \Theta_2$, which means that in variable context Γ, if c is executed in the header context Θ_1, then a header instance type Θ_2 is assigned. Intuitively, Θ_1 encodes the sets of headers that may be valid when type checking a command. Γ is a standard type environment which maps variables x to type τ. If there exists Θ_2 such that $\Gamma \vdash c : \Theta_1 \Rrightarrow \Theta_2$, we say that c is well-typed in Θ_1.

The typing judgment for expressions and actions has the form $\Gamma; \Theta \vdash e : \tau$ and $\Gamma; \Theta \vdash a : \overline{\tau} \to \Theta$ respectively, meaning that expression e has type τ respectively that action a has type $\overline{\tau} \to \Theta$ in variable context Γ and header context Θ.

4.3.1 Operations on header types

To formulate the typing rules for SafeP4, we first define a set of operations on header types, namely *restriction*, *negated restriction*, *inclusion*, *removal* and *emptiness*. In the following we assume that S ranges over elements of the domain $\mathcal{P}(\mathcal{P}(Headers))$.

Restriction The operator $\texttt{Restrict}\ \Theta\ h$ recursively traverses Θ and keeps only those choices in which h is contained, zeroing out the others. Semantically this has the effect of throwing out the subsets of $[\![\Theta]\!]$ that do not contain h, i.e., we define restriction semantically as $S|_h \triangleq \{hs \mid hs \in S \wedge h \in hs\}$. Syntactically we define restriction by induction on Θ as follows:

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

Lemma 4.1 captures the equivalence of the syntactic and the semantic definition.

Lemma 4.1 (Restrict Equal). $[\![\Theta]\!]|_h = [\![\texttt{Restrict}\ \Theta\ h]\!]$

Proof. By induction on Θ. □

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

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

Lemma 4.2 captures the equivalence of the syntactic and the semantic definition.

Lemma 4.2. *(NegRestrict Equal)* $[\![\Theta]\!]|_{\neg h} = [\![\texttt{NegRestrict}\ \Theta\ h]\!]$

Proof. By induction on Θ. □

Inclusion `Includes` Θ h traverses Θ and checks if h is valid in every path. Semantically this says that h is a member of every element of $[\![\Theta]\!]$, i.e., $h \sqsubset S \triangleq \forall hs \in S.h \in hs$. Syntactically we define inclusion by induction on Θ:

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

Lemma A.3 captures the equivalence of the syntactic and the semantic definition.

Lemma 4.3. *(Includes Equal)* $\forall hs \in [\![\Theta]\!].h \in hs = \texttt{Includes}\ \Theta\ h$

Proof. By induction on Θ. □

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

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

Lemma A.4 captures the equivalence of the syntactic and the semantic definition.

Lemma 4.4. *(Remove Equal)* $[\![\Theta]\!] \setminus h = [\![\texttt{Remove}\ \Theta\ h]\!]$

Proof. By induction on Θ. □

Emptiness `Empty` Θ checks if Θ is semantically empty. Syntactically we define emptiness by induction on Θ:

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

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

$$\mathcal{F}(h, f_i) = \tau_i \qquad \textit{Field type lookup}$$
$$\mathcal{A}(a) = \lambda\overline{x} : \overline{\tau}.\ c \qquad \textit{Action lookup}$$
$$\mathcal{H}(e) = \overline{h} \qquad \textit{Referenced Header instances}$$
$$\mathcal{CA}(t, H) = (a_i, \overline{v}) \qquad \textit{Control-plane actions}$$
$$\mathcal{CV}(t) = \overline{S} \qquad \textit{Control-plane validity}$$

$$\mathsf{maskable}(t, e, \textit{exact}) \triangleq \textit{false}$$
$$\mathsf{maskable}(t, e, \textit{ternary}) \triangleq \mathcal{H}(e) \subseteq t.\textit{valids}$$

Figure 4.7: Auxiliary functions.

Lemma 4.5. $[\![\Theta]\!] == \{\}$ *if and only if* `Empty` Θ.

Proof. By induction on Θ. □

4.3.2 Typing rules

SafeP4's typing rules rely on several auxiliary functions shown in Figure 4.7. The field type lookup function $\mathcal{F}(h, f_i)$ returns the type assigned to a field f_i in header instance h by first looking up the corresponding header type η from the global header table and then looking up the field type from the header type declaration. The action lookup function $\mathcal{A}(a)$ returns the action definition $\lambda\overline{x} : \overline{\tau}.\ c$ for action a and $\mathcal{H}(e)$ returns the header instances $\overline{h}$ referenced by expression e. Given a table t and the currently valid headers H, the function $\mathcal{CA}(t, H)$ returns the run-time action to call a_i and the (possibly empty) action data arguments $\overline{v}$. The function $\mathcal{CV}(t)$ analyzes table t and produces a list of sets of valid headers $\overline{S}$—one set for each action—that can be safely assumed valid when the entries are populated by the control plane. Both, $\mathcal{CA}(t, H)$ and $\mathcal{CV}(t)$ are assumed to be instantiated by the control plane.

The function $\mathsf{maskable}(t, e, m)$ takes in a table t, a match key expression e and a match-kind m and checks whether expression e must be evaluated when table t is invoked. If maskable evaluates to *true*, it means that the expression does not need to be evaluated. If the match-kind is *exact*, e always needs to be evaluated and if the match-kind is *ternary*, e only needs to be evaluated if it references at least one header instance that is not part of the valid-match header instances ($t.\textit{valids}$).

Command typing The typing rules for commands are presented in Figure 4.8. The rule T-Zero gives a command an arbitrary output type if the input type is empty. The rules T-Skip and T-Seq are standard. The rule T-If a path-sensitive union type between the type computed for each branch. The rule T-If Valid is similar, but leverages knowledge about the validity of h. So the true branch c_1 is checked in the context `Restrict` Θ h, and the false branch c_2 is checked in the context `NegRestrict` Θ h. The top-level output type is the union of the resulting output types for c_1 and c_2. The rule T-Mod checks that h is guaranteed to be valid using the `Includes` operator, and uses the auxiliary function $\mathcal{F}$ to obtain the type assigned to $h.f$. The set of valid headers does not change when evaluating an assignment, so the output and input types are identical.

T-Zero
$$\frac{\texttt{Empty}\ \Theta_1}{\Gamma \vdash c : \Theta_1 \Rrightarrow \Theta_2}$$

T-Skip
$$\frac{}{\Gamma \vdash skip : \Theta \Rrightarrow \Theta}$$

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

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

T-If Valid
$$\frac{\begin{array}{c}\Gamma \vdash c_1 : \texttt{Restrict}\ \Theta\ h \Rrightarrow \Theta_1 \\ \Gamma \vdash c_2 : \texttt{NegRestrict}\ \Theta\ h \Rrightarrow \Theta_2\end{array}}{\Gamma \vdash valid(h)\ c_1\ else\ c_2 : \Theta \Rrightarrow \Theta_1 + \Theta_2}$$

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

T-Extr
$$\frac{}{\Gamma \vdash extract(h) : \Theta \Rrightarrow \Theta \cdot h}$$

T-Add
$$\frac{}{\Gamma \vdash add(h) : \Theta \Rrightarrow \Theta \cdot h}$$

T-Rem
$$\frac{}{\Gamma \vdash remove(h) : \Theta \Rrightarrow \texttt{Remove}\ \Theta\ h}$$

T-Emit
$$\frac{}{\Gamma \vdash emit(h) : \Theta \Rrightarrow \Theta}$$

T-Apply
$$\frac{\begin{array}{c}\mathcal{CV}(t) = \overline{S} \qquad t.actions = \overline{a} \\ \overline{e} = \{e_j \mid (e_j, m_j) \in t.reads \wedge \neg \mathsf{maskable}(t, e_j, m_j)\} \\ \cdot; \Theta \vdash e_j : \tau_j \quad \text{for } e_j \in \overline{e} \\ \cdot; \texttt{Restrict}\ \Theta\ S_i \vdash a_i : \overline{\tau}_i \to \Theta'_i \quad \text{for } a_i \in \overline{a}\end{array}}{\Gamma \vdash t.apply() : \Theta \Rrightarrow \left(\sum_{a_i \in \overline{a}} \Theta'_i\right)}$$

Figure 4.8: Command typing rules for SafeP4.

The rules T-Extr and T-Add assign header extractions and header additions the type $\Theta \cdot h$, reflecting the fact that h is valid after the command executes. Emitting packet headers does not change the set of valid headers, which is captured by rule T-Emit. The typing rule T-Rem uses the `Remove` operator to remove h from the input type Θ.

Finally, the rule T-Apply checks table applications. To understand how it works, let us first consider a simpler, but less precise, typing rule:

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

Intuitively, this rule says that to type check a table application, we check each expression it reads and each of its actions. The final header type is the union of the types computed for the actions. This rule models table application as a non-deterministic choice between its actions. However, while this rule is sound, it is overly conservative. In particular, it does not model the fact that the control plane often uses header validity bits to control which actions are executed.

Hence, the actual typing rule, T-Apply, makes use of function $\mathcal{CV}(t)$ to first obtain for each action a_i, a set of headers S_i that can be assumed valid when type checking

```
1 table port_vlan_mapping {
2   reads {
3     vlan[0] : valid;
4     vlan[0].vid : ternary;
5     vlan[1] : valid;
6     vlan[1].vid : ternary;
7   } ...
8 }
```

Figure 4.9: Expressions `vlan[0].vid` and `vlan[1].vid` must be wildcarded when `vlan[0]` respectively `vlan[1]` are invalid. The typechecker can warn the programmer about these assumptions for the table to be safe.

$$\textsc{T-Const}\quad \frac{\texttt{typeof}(k) = \overline{\tau} \to \tau' \qquad \Gamma; \Theta \vdash e_i : \tau_i}{\Gamma; \Theta \vdash k(\overline{e}) : \tau'} \qquad \textsc{T-Var}\quad \frac{x : \tau \in \Gamma}{\Gamma; \Theta \vdash x : \tau}$$

$$\textsc{T-Field}\quad \frac{\texttt{Includes}\ \Theta\ h \qquad \mathcal{F}(h, f) = \tau}{\Gamma; \Theta \vdash h.f : \tau}$$

Figure 4.10: Expression typing rules for SafeP4.

a_i. From the match key expressions of the table declaration and the headers assumed valid, we can derive a subset of the expressions read by the table, e.g., excluding expressions that can be wildcarded when certain validity bits are false using the function $\mathsf{maskable}(t, e, m)$. Consequently, we only need to type check these expressions. For example, given the table in Figure 4.9, if an action a_j is matched by the rule $(0, *, 0, *)$, both S_j and e_j are empty. Just like the simplified rule, the final header type is the union of the types computed for the actions, however, we additionally restrict the header context with the header instances assumed to be valid. Here we lift the `Restrict` operator to sets of header instances:

$$\begin{aligned} \texttt{Restrict}\ \Theta\ \{\} &\triangleq \Theta \\ \texttt{Restrict}\ \Theta\ \{h_1, ..., h_n\} &\triangleq \texttt{Restrict}\ (\texttt{Restrict}\ (\texttt{Restrict}\ \Theta\ h_1)\ ...)\ h_n \end{aligned}$$

Expression typing The typing rules for expressions are shown in Figure 4.10. Constants are type-checked according to rule T-Constant, as long as each expression that is passed as an argument to the constant k has the type required by the `typeof` function. The rule T-Var is standard. Rule T-Field checks that header instance h is guaranteed to be valid and assigns the type obtained from the field type lookup function $\mathcal{F}$.

Action typing Given a variable context Γ and header type Θ, an action $\lambda\overline{x} : \overline{\tau}.\ c$ encodes a function of type $\overline{\tau} \to \Theta'$, so long as the body c is well-typed in the context where Γ is extended with $x_i : \tau_i$ for every i.

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

Figure 4.11: Action typing rule for SafeP4.

4.4 Dynamic Semantics

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

For the definition of the operational semantics, we assume that for each declared header type η there exists a deserialization function ($deserialize_\eta$), a serialization function ($serialize_\eta$) and an initialization function ($init_\eta$). The function $deserialize_\eta(I) = (v, I')$ takes in the input bit stream I and returns a header value v populated with bits from I as well as the rest of the input bit stream. For example, assuming the header type $\eta = \{f : bit\langle 3 \rangle;\ g : bit\langle 2 \rangle;\}$ has two fields f and g and $I = 11000B$ where B is the rest of the bit stream following, then $deserialize_\eta(I) = (\{f = 110;\ g = 00;\}, B)$. The serialize function is the corresponding counterpart. It takes the bit values of the fields of a header value and concatenates them in the order in which they are defined to produce a single bit sequence. For example, calling *serialize* on the header value $\{f = 110;\ g = 00;\}$ returns the bit string 11000. Finally, the function $init_\eta$ returns a header instance of type η with all fields set to a default value.

Semantics of commands The reduction rules are presented in Figure 4.12 and Figure 4.13. The command $extract(h)$ evaluates via the rule E-Extr, which looks up the header type in the global header table $\mathcal{HT}$ and then invokes the corresponding deserialization function. The deserialized header value v is added to the map of valid header instances, H and evaluation continues with the remaining input bit stream I'. The rules for validity checks step to the true branch if $h \in dom(H)$ (E-If ValidTrue) and to the false branch otherwise (E-If ValidFalse). The rule E-Rem removes the header from the map H. If a header h is already invalid, removing it has no effect. Modification of header fields is evaluated according to rules E-Mod and E-Mod1. If the assigned expression is fully reduced, the respective field in the header value is updated with the new value in H. Otherwise, the assigned expression is reduced first. Table application commands are evaluated according to rule E-Apply. We first invoke the control plane function $\mathcal{CA}(t, H)$ to determine an action a_i and action data v. Then we use $\mathcal{A}$ to look up the definition of a_i, yielding $\lambda \overline{x} : \overline{\tau}.\ c_i$ and step to $c_i[\overline{v}/\overline{x}]$. Note that for simplicity, we model the evaluation of expressions read by the table using the control-plane function $\mathcal{CA}$. The rule E-Add evaluates addition commands $add(h)$. Similar to header extraction, we first obtain the header type η of the instance h and then use the function $init_\eta$ function to obtain a header instance v of type η with all fields set to a default value and extend the map H with $h \mapsto v$. Note that according to E-AddValid, if the header instance is already valid, $add(h)$ does nothing. The rule E-Emit serializes a header instance h back into a bit stream. It first looks up the corresponding header type and header value in the header table $\mathcal{HT}$ and the map of valid headers respectively. The header value is then passed to the serialization function for the header type to produce a bit sequence

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

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

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

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

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

E-Mod1
$$\frac{\langle H, e\rangle \to e'}{\langle I, O, H, h.f = e\rangle \to \langle I, O, H, h.f = e'\rangle}$$

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

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

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

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

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

Figure 4.12: Operational semantics of commands.

that is appended to the output bit stream. We adopt the semantics of P4 with respect to emitting invalid headers. Emitting an invalid header instance—i.e., a header instance which has not been added or extracted—has no effect on the output bit stream (rule E-EmitInvalid). Notice also that the header remains unchanged in H.

Sequential composition (cf. Figure 4.13) reduces left to right, i.e., the left command needs to be reduced to *skip* (rule E-Seq1) before the right command can be reduced (rule E-Seq). The evaluation of conditionals (rules E-If, E-If True, E-If False) is standard. The expression in the condition is first evaluated to a value. If the condition is true, the conditional steps to command c_1 otherwise to command c_2.

Semantics of expressions The semantics for expressions is defined in Figure 4.14. It is defined in terms of tuples $\langle H, e\rangle$, where H is the same map used in the semantics of commands and e is the expression to evaluate. The rule E-Field reduces header field expressions to the value stored in the heap H for the respective field. To evaluate

E-Seq
$$\frac{}{\langle I, O, H, skip; c_2\rangle \to \langle I, O, H, c_2\rangle}$$

E-Seq1
$$\frac{\langle I, O, H, c_1\rangle \to \langle I', O', H', c_1'\rangle}{\langle I, O, H, c_1; c_2\rangle \to \langle I', O', H', c_1'; c_2\rangle}$$

E-If
$$\frac{\langle H, e\rangle \to e'}{\langle I, O, H, if\ (e)\ then\ c_1\ else\ c_2\rangle \to \langle I, O, H, if\ (e')\ then\ c_1\ else\ c_2\rangle}$$

E-IfTrue
$$\frac{}{\langle I, O, H, if\ (true)\ then\ c_1\ else\ c_2\rangle \to \langle I, O, H, c_1\rangle}$$

E-IfFalse
$$\frac{}{\langle I, O, H, if\ (false)\ then\ c_1\ else\ c_2\rangle \to \langle I, O, H, c_2\rangle}$$

Figure 4.13: Operational semantics of commands (continued).

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

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

E-Const-Cong
$$\frac{\langle H, e_i\rangle \to e_i'}{\langle H, k(v_1, ..., v_{i-1}, e_i, ..., e_n)\rangle \to k(v_1, ..., v_{i-1}, e_i', ..., e_n)}$$

Figure 4.14: Operational semantics for expressions.

constants via the rule E-Const, we assume that there is an evaluation function for constants $[\![k]\!](\overline{v}) = v$ that is well-behaved—i.e., if $\texttt{typeof}(k) = \overline{\tau} \to \tau'$ and $\overline{v : \tau}$, then $\cdot;\cdot \vdash [\![k]\!](\overline{v}) : \tau'$. Arguments passed to constants are evaluated left to right (rule E-Const-Cong).

4.5 Safety

We prove safety in terms of progress and preservation [WF94]. Both theorems make use of the relation $H \vDash \Theta$ which intuitively holds if H is described by Θ. The formal definition, as given in Figure 4.15, satisfies $H \vDash \Theta$ if and only if $dom(H) \in [\![\Theta]\!]$.

The empty header instance map only entails the empty header instance type 1 (Rule Ent-Empty). If a header instance h is contained in the map of valid header instances H, H entails the header instance type h (Rule Ent-Inst). The sequence type $\Theta_1 \cdot \Theta_2$ is entailed by the distinct union of the maps entailing Θ_1 and Θ_2 respectively (Rule Ent-Seq) and the choice type $\Theta_1 + \Theta_2$ is entailed either by the map entailing Θ_1 or the map entailing Θ_2 (Rules Ent-ChoiceL and Ent-ChoiceR).

We prove progress and preservation only for commands. For expressions, we formulate these properties as additional lemmas (Lemmas 4.6 and 4.7). The respective proofs

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

$$\textsc{Ent-ChoiceL}\quad \frac{H \vDash \Theta_1}{H \vDash \Theta_1 + \Theta_2} \qquad \textsc{Ent-ChoiceR}\quad \frac{H \vDash \Theta_2}{H \vDash \Theta_1 + \Theta_2}$$

Figure 4.15: The *Entailment* relation between header instances and header instance types

are straightforward for our system.

Lemma 4.6 (Expression Progress). *If* $\cdot; \Theta \vdash e : \tau$ *and* $H \vDash \Theta$*, then either* e *is a value or* $\exists e'.\langle H, e\rangle \to e'$.

Lemma 4.7 (Expression Preservation). *If* $\Gamma; \Theta \vdash e : \tau$ *and* $H \vDash \Theta$ *and* $\langle H, e\rangle \to e'$ *then* $\Gamma; \Theta \vdash e' : \tau$.

To prove both theorems we also need the following propositions that model our assumptions about the functions $\mathcal{CA}$ and $\mathcal{CV}$ modeling the control plane.

Proposition 1 (Control Plane Reads). *If* $H \vDash \Theta$ *and* $\mathcal{CV}(t) = \bar{S}$ *and* $\bar{e} = \{e_j \mid (e_j, m_j) \in t.reads() \wedge \neg\mathsf{maskable}(t, e_j, m_j)\}$ *and* $\Gamma; \Theta \vdash e_j : \tau_j$ *for* $e_j \in \bar{e}$ *then* $\mathcal{CA}(t, H) = (a_i, \bar{v})$.

Proposition 2 (Control Plane Action Data). *If* $H \vDash \Theta$ *and* $\mathcal{CA}(t, H) = (a_i, \bar{v})$ *and* $\mathcal{A}(a_i) = \lambda\bar{x} : \bar{\tau}.\ c_i$ *then* $\cdot; \cdot \vdash \bar{v} : \bar{\tau}$

Proposition 3 (Control Plane Assumptions). *If* $H \vDash \Theta$ *and* $\mathcal{CA}(t, H) = (a_i, \bar{v})$ *and* $\mathcal{CV}(t) = \bar{S}$ *then* $H \vDash \texttt{Restrict}\ \Theta\ S_i$.

Proposition 1 captures the first assumption—i.e., the control plane can safely install table entries that never read invalid headers, Proposition 2 captures the second assumption—i.e., the action data provided by the control plane has the types expected by the action and Proposition 3 captures the third assumption—i.e., the control plane will only assume valid headers for an action that are valid for a given packet.

Theorem 4.8 (Progress). *If* $\cdot \vdash c : \Theta_1 \Rrightarrow \Theta_2$ *and* $H \vDash \Theta_1$*, 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. □

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

Theorem 4.9 (Preservation). *If* $\Gamma \vdash c : \Theta_1 \Rrightarrow \Theta_2$ *and* $\langle I, O, H, c\rangle \to \langle I', O', H', c'\rangle$*, where* $H \vDash \Theta_1$*, then* $\exists\Theta_1', \Theta_2'.\ \Gamma \vdash c : \Theta_1' \Rrightarrow \Theta_2'$ *where* $H' \vDash \Theta_1'$ *and* $\Theta_2' < \Theta_2$.

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

□

Preservation says that if a command c is well-typed with input type Θ_1 and output type Θ_2 and c evaluates to c' in a single step, then there exists an input type Θ_1' and an output type Θ_2' that make c' well-typed, and Θ_1' describes the same maps of header instances H as Θ_1, and Θ_2' is semantically contained in Θ_2. We define syntactic containment to be $\Theta_1 < \Theta_2 \triangleq [\![\Theta_1]\!] \subseteq [\![\Theta_2]\!]$.

4.6 Related Work

Formal Reasoning for P4 Programs With respect to verifying the correctness of P4 programs, probably the most closely related work to SafeP4 is p4v [Liu+18]. Unlike SafeP4, which is based on a static type system, p4v uses Dijkstra's approach to program verification based on predicate transformer semantics. To model the behavior of the control plane, p4v uses first-order annotations. SafeP4's typing rule for table application is inspired by this idea, but adopts simple heuristics—e.g., we only assume that the control plane is well-behaved—rather than requiring logical annotations. Both p4v and P4Check can be used to verify safety properties of data planes modelled in P4—e.g., that no read or write operations are possible on an invalid header. As it is often the case when comparing approaches based on types to those based on program verification, p4v can check more complex properties, including architectural invariants and program-specific properties—e.g., that the IPv4 time-to-live field is correctly decremented on every packet. However, in general, it requires annotating the program with formal specifications both for the correctness property itself and to model the behavior of the control plane. McKeown et al. developed an operational semantics for P4 [McK+16], which is translated to Datalog to verify safety properties and to check program equivalence. An operational semantics for P4 was also developed in the K framework [RŞ10], yielding a symbolic model checker and deductive verification tool [KR18]. Vera [Sto+18] models the semantics of P4 by translation to SymNet [Sto+16], and develops a symbolic execution engine for verifying a variety of properties, including header validity. Compared to SafeP4, these approaches do not use their formalization of P4 as a foundation for defining a type system that addresses common bugs. To the best of our knowledge, SafeP4 is the first formal calculus for a P4-like packet processing language that provides correct-by-construction guarantees of header safety properties.

Ensuring Null-Safety Other languages have used type systems to rule our safety problems due to null references. For example, NullAway [Sri18] analyzes all Java programs annotated with special annotations, making path-sensitive deductions about which references may be null. Similar to the validity checks in SafeP4, NullAway analyzes conditionals for null checks of the form `var != null` using data flow analysis.

Packet Processing Languages Looking further afield, PacLang [ESM04] is a concurrent packet-processing language that uses a linear type system to allow multiple references to a given packet within a single thread. PacLang and SafeP4 share the use of a type system for verifying safety properties, but they differ in the kind of properties they address and, hence, the kind of type system they employ for this purpose. In addition, the primary focus in PacLang is on efficient compilation whereas SafeP4 is concerned with ensuring safety of header data. Domino [Siv+16] is a domain-specific language for data plane algorithms supporting *packet transactions*—i.e., blocks of code that are guaranteed to be atomic and isolated from other transactions. In Domino, the programmer defines the operations needed for each packet without worrying about

other in-flight packets. In case of success, the compiler guarantees performance at the line rate supported on programmable switches. Overall, Domino focuses on transactional guarantees and concurrency rather than header safety properties. BPF+ [BMG99] and eEBPF [Cor14] are packet-processing frameworks that can be used to extend the kernel networking stack with custom functionality. The modern eBPF framework is based on machine-level programming model, but it uses a virtual machine and code verifier to ensure a variety of basic safety properties. Much of the recent work on eBPF focuses on techniques such as just-in-time compilation to achieve good performance. SNAP [Ara+16] is a language for stateful packet processing based on P4. It offers a programming model with global state registers that are distributed across many physical switches while optimizing for various criteria, such as minimizing congestion. More specifically, the compiler analyzes read/write dependencies to automatically optimize the placement of state and the routing of traffic across the underlying physical topology.

Taint Analysis and Formal Calculi While our approach to track validity is network-specific, it is similar to taint analysis [VIS96; HOM06; HDM14], which attempts to identify secure program parts that can be safely accessed. Of course, there is a long tradition of formal calculi that aim to capture some aspect of computation and make it amenable for mathematical reasoning. The design of SafeP4 is directly inspired by Featherweight Java [IPW01], which stands out for its elegant formalization of a real-world language in an extensible core calculus.

4.7 Chapter Summary

In this chapter we introduced SafeP4, a domain-specific language for program-mable data planes, which is equipped with a static type system that guarantees that all headers read or written are guaranteed to be valid. SafeP4 models the essential features of P4 but prunes away language constructs that are secondary to how packets are processed. To be compatible with both language versions $P4_{14}$ and $P4_{16}$, SafeP4 abstracts away expression typing and evaluation behaviors for expressions in terms of n-ary constants that are assumed to have sound types.

SafeP4 introduces the notion of *header types*, which allow the type system to statically capture which header instances are valid at a specific point in the program. To ensure that capturing valid headers is not too restrictive, SafeP4 employs occurrence typing, which allows the type checker to use more precise types depending on whether explicit validity checks succeed or fail. Furthermore, using a fine-grained representation of valid header instances based on sets of sets of headers, SafeP4 is able to capture valid header instances together with their dependencies per program path.

The problem of header validity being a dynamic property due to the interaction with the control plane is addressed by modeling the runtime interface between the switch and the controller using three functions. Given a table and the currently valid headers, the first function returns the action to call and the possibly empty action data arguments. The second function produces for each action of a table a set of valid headers that can be safely assumed valid when the entries are populated by the control plane. From the table declaration and the header instances that can be assumed valid, the third function derives a list of match key expressions that must be evaluated when the table is invoked. In addition, the control plane interface relies on three basic correctness properties that are assumed to hold: (1) the control plane can safely install table entries that never read invalid headers, (2) the action data provided by the control plane has the types expected

by the action, and (3) the control plane will only assume valid headers for an action that are valid for a given packet. These correctness properties allowed us to prove safety for SafeP4's type system.

CHAPTER 5

Dependently-Typed Data Plane Programming

In Chapter 4, we introduced SafeP4, a domain-specific language that uses a correct-by-construction approach to statically eliminate a variety of errors based on header validity. A major advantage of the presented approach is that no complex program annotations are required, as is the case with other data plane verification tools. However, because header validity is deeply baked into the system as a central correctness property, SafeP4 is limited in its ability to verify richer properties compared to other verification tools. In particular, SafeP4 is not able to capture individual values of the program state. For example, it is not possible to guarantee that headers are only ever accessed on mutually exclusive program paths (e.g., IPv4 and IPv6) or to track dependencies between headers based on header field values.

While type system-based approaches are so far limited in their expressiveness, they have one key advantage over many existing data plane verification tools, which is compositionality. Type systems are designed to enable compositional reasoning—i.e., the types for individual components document assumptions about the components they rely upon as well as the guarantees they offer. So far, there has been little effort in writing modular P4 code in which individual parts of the code can be reused, which has long been common practice in general-purpose programming languages. However, recently, there have been efforts to enable modular designs of data plane programs [Gao+20; Son+20], paving the way for an "open-world" model in which third-party components are embedded into existing programs.

The question therefore arises whether it is possible to combine the expressive power of fully-fledged verification tools with the compositional checking inherent to type systems. Dependently-typed languages [XP99; Con+07; RKJ08; Vaz+14] are increasingly blurring the line between type checking and theorem proving. For instance, Liquid Haskell [RKJ08; Vaz+14] allows programmers to smoothly shift from properties that can be checked with traditional typing disciplines to more sophisticated ones. Under the hood, an SMT solver automatically discharges the formulas generated during type checking without requiring manual proofs. So far, the dependently-typed approach has not yet been explored in the context of network programming.

```
1  parser MyParser(packet_in pkt, out headers hdr, ...) {
2    state start {
3      pkt.extract(hdr.ethernet);
4      transition select(hdr.ethernet.etherType) {
5        0x0800: parse_ipv4;
6        default: accept;
7      }
8    }
9    state parse_ipv4 {
10     pkt.extract(hdr.ipv4);
11     transition accept;
12   }
13 }
14
15 control MyIngress(inout headers hdr, ...) {
16   apply {
17     if (hdr.ipv4.isValid()) {
18       if (hdr.ipv4.src == 10.10.10.10) {
19         drop();
20       }
21     }
22   }
23 }
```

Figure 5.1: Safe program according to SafeP4, using a validity check in the ingress.

In this chapter, we present Π4, a dependently-typed core of the P4 language. For Π4's type system, we extend SafeP4's header types to *heap types*, which also capture the shape of valid packet headers as well as the shape of the incoming and outgoing packet. We formalize its syntax and semantics and show how we can enable precise typing in the presence of domain-specific features that combine packet serialization and deserialization operations with imperative control flow. For example, our novel chomp operator allows to precisely capture the effect of packet deserialization, i.e., it allows to compute the type that remains after extracting bits from a packet buffer. Finally, we prove safety for Π4's type system.

5.1 An Overview of Π4

To get a first understanding of how dependent types allow checking richer properties, this section provides a high-level overview of Π4. Let us consider the program shown in Figure 5.1. The parser (lines 1 to 13) extracts Ethernet, optionally followed by IPv4. The ingress uses an explicit validity check to make sure that IPv4 is valid before conditionally dropping the packet depending on the value of the IPv4 source address (lines 17-19).

This program will pass SafeP4's type checker. On the other hand, if we use the same parser but the implementation of the ingress control shown in Figure 5.2, SafeP4's type checker will reject the program. This might be surprising, because the parser guarantees

```
1     control MyIngress(inout headers hdr, ...) {
2         apply {
3             if (hdr.ethernet.etherType == 0x0800) {
4                 if (hdr.ipv4.src == 10.10.10.10) {
5                     drop();
6                 }
7             }
8         }
9     }
```

Figure 5.2: Ingress control using a data-dependent check to guarantee only valid headers are accessed.

that if the EtherType is equal to 0x0800, the IPv4 header is also valid, thus, the program is actually safe. However, the type computed for the parser $\texttt{ethernet} \cdot (\texttt{ipv4} + 1)$ does not guarantee that IPv4 is valid on all program paths and also the path-sensitive reasoning does not add additional assumptions about the validity of IPv4 in the ingress.

To address this problem, Π4 employs a dependent type system [XP99], in which we can compute a precise type for the program after parsing:

$$(x : \{y : \epsilon \mid |\texttt{y.pkt}_{\texttt{in}}| > 272\}) \rightarrow \left(\begin{array}{l} \Sigma y\colon \texttt{ether}.\{z : \texttt{ipv4} \mid y.\texttt{ether.etherType} == 0x0800\} + \\ \{y : \texttt{ether} \mid y.\texttt{ether.etherType} \neq 0x0800\} \end{array} \right)$$

Intuitively, this type says that, starting with the empty heap ($y : \epsilon$) with 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 (denoted by +): (1) both Ethernet and IPv4 are valid ($\Sigma y\colon \texttt{ether}.\{z : \texttt{ipv4} \mid ...\}$), if the EtherType is equal to 0x0800 (note how $z : \texttt{ipv4}$ is conditioned by $y.\texttt{ether.etherType} == 0x0800$) or (2) just Ethernet is valid, if EtherType is not equal to 0x0800. When checking the ingress control, the type checker can use the predicate $\texttt{ether.etherType} == 0x0800$ on 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— 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 is needed to be valid at the end of the parser, we can use the type $(x : \{y : \epsilon \mid |\texttt{y.pkt}_{\texttt{in}}| > 272\}) \rightarrow \texttt{ether}_{\approx}$, which indicates that at least Ethernet is valid but possibly others, too.

5.2 Design

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. We want to show that dependent types are a good match for data plane programming.

Deviation from P4 Similar to SafeP4, Π4 focuses on the unique aspects of the P4 programming language, which benefit from dependent types, (e.g., parsing, deparsing, validity, and control flow) and omits features that would simply add clutter (e.g., externs, registers, checksums, hashing, and pipelines). We can even get away without explicitly modeling tables. Following p4v [Liu+18], Π4 uses ghost state and conditionals to encode tables, which is discussed in detail in Section 8.3.2. Consequently, Π4 is a loop-free[1] imperative language with a few domain-specific primitive commands: *extract*(ι), *add*(ι), *remove*(ι), *remit*(ι), and *reset*.

P4's *emit* primitive serializes a header instance ι into a series of bits 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 expressed using the command *if* (ι.*valid*) *remit*(ι) *else skip*.

We follow the design decisions made in SafeP4 with respect to explicitly validating header instances. Again, to avoid dealing with undefined values from reads to uninitialized header instances, rather than forcing the programmer to manually write default values, the *add*(ι) command sets ι.*valid* to `true`, and assigns instance ι a pre-determined default value (say 0). 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 [Bos+13], 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.

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`$[48{:}96]$.

Type system Π4's type system is intended to promote modular reasoning, so 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 type check c with type σ and then use σ when checking the rest of the program.

We always assign a dependent function type $(x : \tau_1) \to \tau_2$ to a command c, where x may occur in τ_2. This design allows us to relate the input and output heaps of commands described by *heap types* τ_1 and τ_2 respectively. For example, we may want to ensure that the Ethernet header has the same value after being deparsed, followed by a reset, and then being parsed again. 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 type $\{y : \tau_2 \mid x.\texttt{eth} = y.\texttt{eth}\}$.

Granularity of types Our goal is to equip Π4's type system with the ability to capture information down to the bit level. In particular, we must ensure that the input type and output type remain consistent after bits have been shuffled around by a command, which is especially challenging when parsing. For example, given an input type $\{x : \top \mid x.pkt_{in}[0{:}8] = 0x04 \wedge |x.pkt_{in}| > 160\}$, where $x.pkt_{in}$ represents an incoming

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

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

Figure 5.3: Syntax of Π4

packet and the command *extract*(`ipv4`), the output type should reflect that the `ipv4` header instance is now valid, that `ipv4`[0:8] is 0x04, and that $x.pkt_{in}$ may have no more bits remaining. Π4 accomplishes this using two key mechanisms: (1) a dependent sum type $\Sigma x\colon \tau_1.\tau_2$ that computes the disjoint union of the valid instances in τ_1 and τ_2 and concatenates the incoming and outgoing packets and (2) a refinement transformer, chomp, that manipulates input refinements to be consistent with the extraction operation.

5.3 Syntax

Figure 5.3 shows the syntax of Π4. Boolean formulae φ include expression equality ($e_1 = e_2$), expression comparison ($e_1 > e_2$), conjunction ($\wedge$), negation ($\neg$), validity of instances ($x.\iota.valid$) and boolean literals true and false. Expressions e include naturals, bit vectors, packet length $|x.p|$, addition (+), concatenation ($e_1@e_2$), packet access $x.p$ and slices of packets ($x.p[l\!:\!r]$) and instances ($x.\iota[l\!:\!r]$). Packet accesses refer either to the input packet (pkt_{in}) or the output packet (pkt_{out}).

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 $x.\iota[\mathsf{0} : \mathsf{sizeof}(\iota)]$, and similarly for the corresponding expressions 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 $x \equiv y$ (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. It is formally defined as follows:

$$
\begin{aligned}
x \equiv y \triangleq\ & x.pkt_{in} = y.pkt_{in} \wedge x.pkt_{out} = y.pkt_{out} \wedge \\
& \bigwedge_{\iota \in dom(\mathcal{HT})} (x.\iota.valid \wedge y.\iota.valid \wedge x.\iota = y.\iota) \vee (\neg x.\iota.valid \wedge \neg y.\iota.valid)
\end{aligned}
$$

In contrast, instance equality only requires that the instances are equivalent in both heaps according to the following definition:

$$
x \equiv_\iota y \triangleq \bigwedge_{\iota \in dom(\mathcal{HT})} (x.\iota.valid \wedge y.\iota.valid \wedge x.\iota = y.\iota) \vee (\neg x.\iota.valid \wedge \neg y.\iota.valid)
$$

A program consists of a sequence of declarations $\overline{d}$ and a command c, where $\overline{x}$ is a shorthand for a possibly empty sequence $x_1, ..., x_n$. Declarations d include header type declarations $\eta\ \{\overline{f : \mathsf{BV}}\}$ 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 bit vector type. We write $f : \mathsf{BV}$ 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*), removing (*remove*) and modifying ($\iota.f := e$) 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 and all assumptions about header validity. The *if*-command conditionally executes one out of two commands based on the value of the boolean formulae φ. Commands can be sequentially composed ($c_1; c_2$), *skip* is a no-op, and with type ascription ($c\ as\ (x : \tau) \to \tau$) it is possible to explicitly assign a type to a command which it is assumed to have at the current point in the program. We assume that every header referenced in a program has a corresponding instance declaration—a property that could be enforced using a simple static analysis.

Both heap types and commands share the same syntactic categories of formulas and expressions. However, since there are no binders at the level of commands, we implicitly assume that formulas used as the condition for an if-statement as well as expressions assigned to header fields are implicitly prefixed with a variable named heap. We usually omit these binders in the surface syntax, for example, we write `if(ethernet.valid) ...else...` instead of `if(heap.ethernet.valid)...else...`.

Π4 provides two categories of types, base types σ and heap types τ. Base types include natural numbers ($\mathbb{N}$), booleans ($\mathbb{B}$), bit vectors (BV) and dependent function types ($(x : \tau) \to \tau$). Heap types include the bottom type ($\varnothing$), the top type ($\top$) and dependent pairs $\Sigma x : \tau_1.\tau_2$, where x may occur in τ_2. Π4 also supports fine-grained path-dependent reasoning via union types $(\tau_1 + \tau_2)$. Refinement types $\{x : \tau \mid \varphi\}$ allow to endow a type with a boolean predicate φ which is assumed to hold for all heaps described by type τ. We also often need to reference intermediate types, which is achieved with substitution types $\tau_2[x \mapsto \tau_1]$, where x may occur in τ_2. In such a type, τ_1 may represent the type at any earlier point in the program.

To ease the notation, we additionally define the types ϵ, ι and $\iota_{\approx}$. These types are only syntactic sugar and thus can be expressed by combinations of the previously described heap types. The type $\epsilon \triangleq \{x : \top \mid \bigwedge_{\iota \in dom(\mathcal{HT})} \neg x.\iota.valid\}$ describes the empty heap on which no header instances are valid. The type $\iota \triangleq \{x : \top \mid x.\iota.valid \wedge \bigwedge_{\iota' \in dom(\mathcal{HT}), \iota' \neq \iota} \neg x.\iota'.valid\}$ describes the heap on which only instance ι is valid, while $\iota_{\approx} \triangleq \{x : \top \mid x.\iota.valid\}$ describes the heap on which at least instance ι is guaranteed to be valid.

5.4 Well-formedness

We assume that all types, formulae and expressions are well-formed, i.e., they satisfy the following basic syntactic properties as defined in Figure 5.4, Figure 5.5 and Figure 5.6.

1. There are no free variables in types

$$\textsc{Wf-Nat}\ \frac{}{\mathsf{wf}\ \mathbb{N}} \qquad \textsc{Wf-Bool}\ \frac{}{\mathsf{wf}\ \mathbb{B}} \qquad \textsc{Wf-BV}\ \frac{}{\mathsf{wf}\ \mathsf{BV}} \qquad \textsc{Wf-Pi}\ \frac{\cdot \vdash \mathsf{wf}_\tau\ \tau_1 \qquad x \vdash \mathsf{wf}_\tau\ \tau_2}{\mathsf{wf}\ (x : \tau_1) \to \tau_2} \qquad \textsc{Wf-Nothing}\ \frac{}{\Gamma \vdash \mathsf{wf}_\tau\ \varnothing}$$

$$\textsc{Wf-Top}\ \frac{}{\Gamma \vdash \mathsf{wf}_\tau\ \top} \qquad \textsc{Wf-Choice}\ \frac{\Gamma \vdash \mathsf{wf}_\tau\ \tau_1 \qquad \Gamma \vdash \mathsf{wf}_\tau\ \tau_2}{\Gamma \vdash \mathsf{wf}_\tau\ \tau_1 + \tau_2} \qquad \textsc{Wf-Sigma}\ \frac{\Gamma \vdash \mathsf{wf}_\tau\ \tau_1 \qquad \Gamma, x \vdash \mathsf{wf}_\tau\ \tau_2}{\Gamma \vdash \mathsf{wf}_\tau\ \Sigma x\colon \tau_1.\tau_2}$$

$$\textsc{Wf-Ref}\ \frac{\Gamma \vdash \mathsf{wf}_\tau\ \tau \qquad \Gamma, x \vdash \mathsf{wf}_\varphi\ \varphi}{\Gamma \vdash \mathsf{wf}_\tau\ \{x : \tau \mid \varphi\}} \qquad \textsc{Wf-Subst}\ \frac{\Gamma, x \vdash \mathsf{wf}_\tau\ \tau_1 \qquad \Gamma \vdash \mathsf{wf}_\tau\ \tau_2}{\Gamma \vdash \mathsf{wf}_\tau\ \tau_1[x \mapsto \tau_2]}$$

Figure 5.4: Well-formedness of types.

$$\textsc{Wf-True}\ \frac{}{\Gamma \vdash \mathsf{wf}_\varphi\ \mathsf{true}} \qquad \textsc{Wf-False}\ \frac{}{\Gamma \vdash \mathsf{wf}_\varphi\ \mathsf{false}} \qquad \textsc{Wf-Valid}\ \frac{x \in \Gamma}{\Gamma \vdash \mathsf{wf}_\varphi\ x.\iota.valid} \qquad \textsc{Wf-Neg}\ \frac{\Gamma \vdash \mathsf{wf}_\varphi\ \varphi}{\Gamma \vdash \mathsf{wf}_\varphi\ \neg\varphi}$$

$$\textsc{Wf-Conj}\ \frac{\Gamma \vdash \mathsf{wf}_\varphi\ \varphi_1 \qquad \Gamma \vdash \mathsf{wf}_\varphi\ \varphi_2}{\Gamma \vdash \mathsf{wf}_\varphi\ \varphi_1 \wedge \varphi_2} \qquad \textsc{Wf-Eq}\ \frac{\Gamma \vdash \mathsf{wf}_e\ e_1 \qquad \Gamma \vdash \mathsf{wf}_e\ e_2}{\Gamma \vdash \mathsf{wf}_\varphi\ e_1 = e_2}$$

$$\textsc{Wf-Gt}\ \frac{\Gamma \vdash \mathsf{wf}_e\ e_1 \qquad \Gamma \vdash \mathsf{wf}_e\ e_2}{\Gamma \vdash \mathsf{wf}_\varphi\ e_1 > e_2}$$

Figure 5.5: Well-formedness of formulae.

2. The bounds of bit vector slices are positive and describe at least a range of length one.

3. The bounds of instance slices respect the statically known size of the instance

5.5 Dynamic Semantics

The operational semantics of Π4 is in many aspects similar to the operational semantics of SafeP4 (cf. Section 4.4). It is also defined in terms of a four-tuple $\langle I, O, H, c\rangle$, where I is the bit stream of the incoming packet, O is the bit stream of the outgoing packet, H is a map that relates instance names to records containing the field values, and c is a command. The rules of the operational semantics of Π4 are shown in Figure 5.8.

The semantics of the extract command (rule E-Extract) is identical to the semantics of the command in SafeP4. Again, we assume the existence of a deserialization function $deserialize_\eta$ (as defined in Section 4.4) that copies the appropriate number of bits from the input bit stream into the deserialized representation of the instance v leaving the remainder of the input bit stream I'. The deserialized value is added to the map of valid header instances H.

$$\textsc{Wf-Num}\quad \frac{}{\Gamma \vdash \mathsf{wf}_e\ n} \qquad \textsc{Wf-Bitvec}\quad \frac{}{\Gamma \vdash \mathsf{wf}_e\ bv} \qquad \textsc{Wf-Length}\quad \frac{x \in \Gamma}{\Gamma \vdash \mathsf{wf}_e\ |x.p|} \qquad \textsc{Wf-Plus}\quad \frac{\Gamma \vdash \mathsf{wf}_e\ e_1 \qquad \Gamma \vdash \mathsf{wf}_e\ e_2}{\Gamma \vdash \mathsf{wf}_e\ e_1 + e_2}$$

$$\textsc{Wf-Concat}\quad \frac{\Gamma \vdash \mathsf{wf}_e\ e_1 \qquad \Gamma \vdash \mathsf{wf}_e\ e_2}{\Gamma \vdash \mathsf{wf}_e\ e_1@e_2} \qquad \textsc{Wf-Packet}\quad \frac{x \in \Gamma}{\Gamma \vdash \mathsf{wf}_e\ x.p} \qquad \textsc{Wf-PacketSlice}\quad \frac{x \in \Gamma \qquad l \geq 0 \qquad l < r}{\Gamma \vdash \mathsf{wf}_e\ x.p[l:r]}$$

$$\textsc{Wf-InstanceSlice}\quad \frac{x \in \Gamma \qquad l \geq 0 \qquad l < r \qquad r \leq \mathsf{sizeof}(\iota)}{\Gamma \vdash \mathsf{wf}_e\ x.\iota[l:r]}$$

Figure 5.6: Well-formedness of expressions.

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

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

$$\textsc{E-Remove}\quad \frac{\iota \in dom(H)}{\langle I, O, H, remove(\iota)\rangle \to \langle I, O, H \setminus \iota, skip\rangle}$$

$$\textsc{E-Mod}\quad \frac{H(\iota) = r \qquad r' \triangleq \{r\ with\ f = v\}}{\langle I, O, H, \iota.f := v\rangle \to \langle I, O, H[\iota \mapsto r'], skip\rangle} \qquad \textsc{E-Mod1}\quad \frac{\langle I, O, H, t\rangle \to t'}{\langle I, O, H, h.f = t\rangle \to \langle I, O, H, h.f = t'\rangle}$$

$$\textsc{E-Remit}\quad \frac{\iota \in dom(H) \qquad \mathcal{HT}(\iota) = \eta \qquad serialize_\eta(H(\iota)) = bv}{\langle I, O, H, remit(\iota)\rangle \to \langle I, O :: bv, H, skip\rangle} \qquad \textsc{E-Reset}\quad \frac{I' = O@I}{\langle I, O, H, reset\rangle \to \langle I', \langle\rangle, [], skip\rangle}$$

Figure 5.7: Small-step operational semantics of Π4.

Command *add* evaluates by rule E-Add if the instance is not yet valid. The evaluation is similar to rule E-Extract, except that no bits are taken from the input bit stream. Again, we assume that there exists an initialization function $init_\eta$ for every heap type η that initializes all fields of an instance to a fixed value (cf. Section 4.4). If an instance is already valid, the program gets stuck.

Removing a header instance (rule E-Remove) requires that the header instance is valid—i.e., it is contained in H. While the input and output packet are not affected, the instance is removed from the map of valid header instances H, which is denoted by $H \setminus \iota$.

Command $remit(\iota)$ (rule E-Remit) also requires that the header instance is valid.

E-Seq

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

E-Seq1

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

E-Ascribe

$$\frac{}{\langle I, O, H, c\ as\ (x : \tau_1) \rightarrow \tau_2\rangle \rightarrow \langle I, O, H, c\rangle}$$

E-If

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

E-IfTrue

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

E-IfFalse

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

Figure 5.8: Additional rules of the small-step operational semantics of Π4.

Again, we assume there is a serialization function for every heap type (cf. Section 4.4), which turns a record representing the instance back into a bit sequence. 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. In contrast to SafeP4, the program gets stuck, if we try to emit a header instance that is currently not valid.

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 (rule E-Mod1).

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.

An ascribed command $c\ as\ \sigma$ (rule E-Ascribe) evaluates to c trivially, without modifying the heap. 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 are also standard. Conditionals reduce the boolean formula first (rule E-If) and if it evaluates to true, the then-branch (rule E-IfTrue) otherwise the else-branch (rule E-IfFalse) is evaluated.

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

Figure 5.9: Semantics of heap types

$$
\begin{bmatrix} a & \mapsto & 1011 \\ pkt_{in} & \mapsto & 1101 \\ pkt_{out} & \mapsto & \langle\rangle \end{bmatrix} \mathbin{++} \begin{bmatrix} b & \mapsto & 11 \\ pkt_{in} & \mapsto & 0 \\ pkt_{out} & \mapsto & 0000 \end{bmatrix} = \begin{bmatrix} a & \mapsto & 1011 \\ b & \mapsto & 11 \\ pkt_{in} & \mapsto & 11010 \\ pkt_{out} & \mapsto & 0000 \end{bmatrix}
$$

Figure 5.10: Example of a heap concatenation. The resulting heap contains the union of header instances while the input packet respectively output packet of the second heap is appended to the input packet respectively output packet of the first heap.

5.6 Static Semantics

Π4's type system is capable of capturing bit-level dependencies between header instances and the incoming and outgoing packet at any given program point. A heap h in the set of heaps $\mathcal{H}$ describes such 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 for every valid header instance a mapping from the instance name to a bit vector, as well as two special entries pkt_{in} and pkt_{out} representing the incoming and outgoing packet buffers.

Semantics of heap types Heap types τ represent sets of heaps, where each element in the set describes a different program path. Heap types are evaluated in an environment $\mathcal{E}$, which maps variables x,y,z to heaps and bit variables b_0, b_1, b_2 to single bits. The environment models other heaps available in the current scope upon which the current header type may depend.

The semantics of types is shown in Figure 5.9. 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. 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 $h = h_1 \mathbin{++} h_2$ 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} being the concatenation of respective bit vectors in h_1 and h_2 as exemplified in Figure 5.10.

$$
\begin{aligned}
[\![\varphi]\!]_{\mathcal{E}} &\in \mathbb{B} \\
[\![e_1 = e_2]\!]_{\mathcal{E}} &= [\![e_1]\!]_{\mathcal{E}} = [\![e_2]\!]_{\mathcal{E}} \\
[\![e_1 > e_2]\!]_{\mathcal{E}} &= [\![e_1]\!]_{\mathcal{E}} > [\![e_2]\!]_{\mathcal{E}} \\
[\![\varphi_1 \wedge \varphi_2]\!]_{\mathcal{E}} &= [\![\varphi_1]\!]_{\mathcal{E}} \wedge [\![\varphi_2]\!]_{\mathcal{E}} \\
[\![\neg\varphi]\!]_{\mathcal{E}} &= \neg[\![\varphi]\!]_{\mathcal{E}} \\
[\![x.\iota.valid]\!]_{\mathcal{E}} &= \iota \in dom(\mathcal{E}(x)) \\
[\![\mathsf{true}]\!] &= \mathit{true} \\
[\![\mathsf{false}]\!] &= \mathit{false}
\end{aligned}
$$

Figure 5.11: Semantics of formulae.

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 $\{x : \tau \mid \varphi\}$ denotes the set of heaps described by τ for which the predicate φ holds. The semantics of the additional heap types we have defined as syntactic sugar is defined as follows:

$$
\begin{aligned}
[\![\epsilon]\!]_{\mathcal{E}} &= \{h \mid \forall \iota.\iota \notin dom(h)\} \\
[\![\iota]\!]_{\mathcal{E}} &= \{h \mid \iota \in dom(h) \wedge \forall \iota' \in dom(\mathcal{HT}).\ \iota' \neq \iota \rightarrow \iota' \notin dom(h)\} \\
[\![\iota_{\approx}]\!]_{\mathcal{E}} &= \{h \mid \iota \in dom(h)\}
\end{aligned}
$$

Semantics of formulae The semantics of formulae is defined in Figure 5.11. Refinement predicates φ are 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. Similarly, the semantics of expression comparison ($e_1 > e_2$) is defined as the semantic comparison between expressions e_1 and e_2. Instance validity ($x.\iota.valid$) evaluates to *true*, if header instance ι is contained in the heap bound to x in environment $\mathcal{E}$, otherwise it evaluates to false. The semantics of conjunction, negation and the boolean literals true and false are standard.

Semantics of expressions The semantics of expressions is defined in Figure 5.12. Expressions evaluate to either a bit vector or a natural number. Again, we use the same evaluation environment $\mathcal{E}$. The semantics of naturals and bit vectors is standard, except for bit variables $\mathsf{b_n}$. In addition to bit literals 0 and 1, bit vectors can contain bit variables, which are looked up from the environment during evaluation. The semantic of addition is also standard. 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. The semantics of bit vector concatenation is as expected. If $[\![e_1]\!]_{\mathcal{E}} = \langle b_0, ..., b_n \rangle$ and $[\![e_2]\!]_{\mathcal{E}} = \langle b_{n+1}, ..., b_m \rangle$, then $[\![e_1 @ e_2]\!] = \langle b_0, ..., b_n, b_{n+1}, ..., b_m \rangle$.

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. The semantics of instance slices $x.\iota[l:r]$ is defined similarly, but the lookup occurs on header instance ι. Since the size and order of header fields is statically known, a field access $x.\iota.f$ is just a named slice on a header instance. We interpret slices as 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 $bv[1:4] = 010$.

$$
\begin{aligned}
[\![e]\!]_{\mathcal{E}} &\in \mathsf{BV} \cup \mathbb{N} \\
[\![n]\!]_{\mathcal{E}} &= n \\
[\![|x.p|]\!]_{\mathcal{E}} &= \begin{cases} 0 & \text{if } \mathcal{E}(x)(p) = \langle\rangle \\ n & \text{if } \mathcal{E}(x)(p) = \langle b_1, ..., b_n \rangle \end{cases} \\
[\![e_1 + e_2]\!]_{\mathcal{E}} &= [\![e_1]\!]_{\mathcal{E}} + [\![e_2]\!]_{\mathcal{E}} \\
[\![e_1 @ e_2]\!]_{\mathcal{E}} &= [\![e_1]\!]_{\mathcal{E}} @ [\![e_2]\!]_{\mathcal{E}} \\
[\![x.p]\!]_{\mathcal{E}} &= \begin{cases} \langle b_1, ..., b_n \rangle & \text{if } \mathcal{E}(x)(p) = \langle b_1, ..., b_n \rangle \end{cases} \\
[\![x.p[n:m]]\!]_{\mathcal{E}} &= \begin{cases} \langle b_n, ..., b_{m-1} \rangle & \text{if } [\![x.p]\!]_{\mathcal{E}} = \langle b_0, ..., b_k \rangle \wedge 0 \leq n < m \leq k+1 \end{cases} \\
[\![x.\iota[n:m]]\!]_{\mathcal{E}} &= \begin{cases} \langle b_n, ..., b_{m-1} \rangle & \text{if } [\![x.\iota]\!]_{\mathcal{E}} = \langle b_0, ..., b_k \rangle \wedge 0 \leq n < m \leq k+1 \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
[\![bv]\!]_{\mathcal{E}} &\in \mathsf{BV} \\
[\![\langle\rangle]\!]_{\mathcal{E}} &= \langle\rangle \\
[\![0 :: bv]\!]_{\mathcal{E}} &= \langle 0, b_1, ..., b_n \rangle && \text{if } [\![bv]\!]_{\mathcal{E}} = \langle b_1, ..., b_n \rangle \\
[\![1 :: bv]\!]_{\mathcal{E}} &= \langle 1, b_1, ..., b_n \rangle && \text{if } [\![bv]\!]_{\mathcal{E}} = \langle b_1, ..., b_n \rangle \\
[\![\mathsf{b_n} :: bv]\!]_{\mathcal{E}} &= \langle \mathcal{E}(\mathsf{b_n}), b_1, ..., b_m \rangle && \text{if } [\![bv]\!]_{\mathcal{E}} = \langle b_1, ..., b_m \rangle
\end{aligned}
$$

Figure 5.12: Semantics of expressions (top) and bit vectors (bottom).

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

Typing Judgment The typing judgement has the form $\Gamma \vdash c : (x : \tau_1) \rightarrow \tau_2$. Intuitively, type τ_1 describes the input heap and τ_2 describes the output heap obtained after the execution of command 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 y maps to τ (i.e., $\Gamma, y : \tau \vdash c : (x : \tau_1) \rightarrow \tau_2$) 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.

Subtyping We write $\Gamma \vdash \tau_1 <: \tau_2$ to denote the subtyping check between τ_1 and τ_2. Context Γ captures external dependencies of τ_1 and τ_2 respectively. We define subtyping semantically as follows:

$$\Gamma \vdash \tau_1 <: \tau_2 \triangleq \forall \mathcal{E}.\mathcal{E} \vDash \Gamma \implies [\![\tau_1]\!]_{\mathcal{E}} \subseteq [\![\tau_2]\!]_{\mathcal{E}}$$

Type τ_1 is a subtype of type τ_2 in context Γ, if and only if for any environment $\mathcal{E}$ such that environment $\mathcal{E}$ entails context Γ, the set of heaps described by τ_1 is a subset of the set of heaps described by τ_2 both evaluated in environment $\mathcal{E}$.

$$\textsc{Ent-Top}\quad \frac{}{(I,O,H) \vDash_{\mathcal{E}} \top}$$

$$\textsc{Ent-ChoiceL}\quad \frac{(I,O,H) \vDash_{\mathcal{E}} \tau_1}{(I,O,H) \vDash_{\mathcal{E}} \tau_1 + \tau_2}$$

$$\textsc{Ent-ChoiceR}\quad \frac{(I,O,H) \vDash_{\mathcal{E}} \tau_2}{(I,O,H) \vDash_{\mathcal{E}} \tau_1 + \tau_2}$$

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

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

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

Figure 5.13: Entailment between heaps and heap types.

The entailment between environment $\mathcal{E}$ and typing context Γ is formally defined as follows:

$$\mathcal{E} \vDash \Gamma \triangleq \forall x_i \in dom(\Gamma).\mathcal{E}(x_i) = h_i \wedge h_i \vDash_{\mathcal{E}} \Gamma(x_i)$$

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 relation between a heap and a type is defined in Figure 5.13. A heap $H[pkt_{in} \mapsto I, pkt_{out} \mapsto O]$, in short (I,O,H) entails a type τ, if the heap (I,O,H) is contained in the semantics of τ.

Command typing The typing rules for commands are shown in Figure 5.14 and Figure 5.15. The typing rule T-Extract first captures that in order to execute an *extract* command, the input packet must provide enough bits to populate the instance. The predicate $\mathsf{sizeof}_{pkt_{in}}(\tau) \geq n$ holds, if and only if the input packet in any heap described by type τ contains at least n bits, i.e., $\mathsf{sizeof}_{pkt_{in}}(\tau) \geq n$ iff $\forall \mathcal{E}, h \in [\![\tau]\!]_{\mathcal{E}}.|h(pkt_{in})| \geq n$.

Further, rule T-Extract also captures that instance ι must be valid after an *extract* command is executed. When a packet header is extracted, the first n bits—where n is the number of bits contained in the header instance—are removed from the input packet and copied into the instance. We need to reflect this change accordingly. This is the task of the chomp refinement transformer, which transforms our input type to obtain a new type, which reflects this change. The chomp operator will be discussed in detail in Section 5.7.

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 is retained and just instance ι is added, which is initialized with the value provided by initialization function $\mathsf{init}_{\mathcal{HT}(\iota)}$.

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. Helper function $\mathcal{F}(\iota, f)$ returns the bit vector type for field f in instance ι. While in our formalization we do not explicitly distinguish different bit vector types, this allows for a more fine-grained check if the assigned expression is compatible with the header field. Instead of just

T-Extract

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

T-Add

$$\frac{\mathsf{Excludes}\ \Gamma\ \tau_1\ \iota \qquad \mathsf{init}_{\mathcal{HT}(\iota)} = v \qquad \varphi \triangleq z.pkt_{in} = z.pkt_{out} = \langle\rangle \wedge z.\iota = v}{\Gamma \vdash add(\iota) : (x : \tau_1) \to \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \iota \mid \varphi\}}$$

T-Mod

$$\frac{\mathsf{Includes}\ \Gamma\ \tau_1\ \iota \qquad \mathcal{F}(\iota, f) = \mathsf{BV} \qquad \Gamma; \tau_1 \vdash e : \mathsf{BV} \qquad \varphi_{pkt} \triangleq y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out} \qquad \varphi_\iota \triangleq \forall \kappa \in dom(\mathcal{HT}).\ \kappa \neq \iota \implies y.\kappa = x.\kappa \wedge \qquad \varphi_f \triangleq \forall g \in dom(\mathcal{HT}(\iota)).\ g \neq f \implies y.\iota.g = x.\iota.g}{\Gamma \vdash \iota.f := e : (x : \tau_1) \to \{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = e[x/\mathsf{heap}]\}}$$

T-Remove

$$\frac{\mathsf{Includes}\ \Gamma\ \tau_1\ \iota \qquad \varphi_\iota \triangleq \forall \kappa \in dom(\mathcal{HT}).\kappa \neq \iota \implies y.\kappa = x.\kappa \qquad \varphi_{pkt} \triangleq y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out}}{\Gamma \vdash remove(\iota) : (x : \tau_1) \to \{y : \top \mid \varphi_\iota \wedge \varphi_{pkt} \wedge \neg y.\iota.valid\}}$$

T-Remit

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

T-Reset

$$\frac{\varphi_1 \triangleq z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out} \qquad \varphi_2 \triangleq z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{in}}{\Gamma \vdash reset : (x\colon \tau_1) \to \Sigma y\colon \{z : \epsilon \mid \varphi_1\}.\{z : \epsilon \mid \varphi_2\}}$$

Figure 5.14: Typing rules for domain-specific commands.

checking if the assigned expression is of type bit vector, we can additionally check that the bit-size of the assigned expression matches the bit-size of the header field we are assigning to. The output type is similar to the strongest post-condition 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 $e[x/\mathsf{heap}]$. Predicate φ_{pkt} ensures that the input and output packet remain unchanged, predicate φ_ι ensures that all instances beside instance ι remain unchanged and predicate φ_f ensures that all fields except for field f on instance ι remain unchanged. Rule T-Remove first checks that the instance to be removed is valid in the input type. The output type reflects that all other instances besides instance ι as well as the input packet and output packet remain unchanged. To typecheck the command T-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

T-If

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

T-Seq

$$\frac{\Gamma \vdash c_1 : (x : \tau_1) \to \tau_{12} \qquad \Gamma, x : \tau_1 \vdash c_2 : (y : \tau_{12}) \to \tau_{22}}{\Gamma \vdash c_1; c_2 : (x : \tau_1) \to \tau_{22}[y \mapsto \tau_{12}]}$$

T-Skip

$$\frac{\tau_2 \triangleq \{y : \tau_1 \mid y \equiv x\}}{\Gamma \vdash \mathit{skip} : (x : \tau_1) \to \tau_2}$$

T-Ascribe

$$\frac{\Gamma \vdash c : \sigma}{\Gamma \vdash c\ \mathit{as}\ \sigma : \sigma}$$

T-Sub

$$\frac{\Gamma \vdash \tau_1 <: \tau_3 \qquad \Gamma, x : \tau_1 \vdash \tau_4 <: \tau_2 \qquad \Gamma \vdash c : (x : \tau_3) \to \tau_4}{\Gamma \vdash c : (x : \tau_1) \to \tau_2}$$

Figure 5.15: Additional command typing rules.

validity information unchanged (first projection of the assigned Σ-type). Rule T-Reset resets all assumptions about header validity, empties the output packet pkt_{out} and sets the input packet 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.

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 (cf. Section 5.3) and since we want to use the formula as a refinement in the type, we have to adjust the binders accordingly. 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 $\varphi[x/\mathsf{heap}]$ to denote the formula obtained from φ in which all variables heap are substituted with x. For example, if the command is `if(ethernet.etherType = 0x0800) extract(ipv4) else skip`, we typecheck the then-branch with type $(x : \{y : \tau_1 \mid y.\texttt{ethernet.etherType} = 0x0800\}) \to \tau_{12}$. The full command is checked with type $(x : \tau_1) \to \{y : \tau_{12} \mid x.\texttt{ethernet.etherType} = 0x0800\} + \{y : \tau_{22} \mid \neg x.\texttt{ethernet.etherType} = 0x0800\}$.

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 also standard, except that it strictly enforces that the heaps described by the output type and input type respectively are equivalent. The typing rule for ascription T-Ascribe is standard. The typing rule for subsumption T-Sub is also standard. Since τ_4 can depend on the input type τ_3 and similarly, since τ_2 can depend on τ_1, we need to extend the typing context with the respective types. Because of the subtype relation between τ_1 and τ_3, all heaps that τ_4 can depend on are also described by type τ_1, thus for the subtyping check between types τ_4 and τ_2 we extend the type typing context with τ_1.

$$\begin{aligned}\mathsf{chomp} &: \tau \times \iota \times \mathcal{X} \to \tau \\ \mathsf{chomp}(\tau, \iota, x) &\triangleq \mathsf{chompRec}(\tau, \mathsf{sizeof}(\iota), x, \iota)\end{aligned}$$

where

$$\mathsf{chompRec} : \tau \times \mathbb{N} \times \mathcal{X} \times \iota \to \tau$$

$$\mathsf{chompRec}(\tau, n, x, \iota) \triangleq \begin{cases} \tau & \text{if } n = 0 \\ \text{let } \tau' = \mathsf{heapRef}_1(& \text{o/w } (\mathsf{b_n} \text{ fresh}) \\ \qquad\qquad \mathsf{chomp}_1(\tau, \mathsf{b_n}), \mathsf{b_n}, \iota, n) \text{ in} & \\ \mathsf{chompRec}(\tau', n-1, x, \iota) & \end{cases}$$

Figure 5.16: Definition of chomp.

$$\begin{aligned}\mathsf{chomp}_1 &: \tau \times B_n \to \tau \\ \mathsf{chomp}_1(\Sigma x\colon \tau_1.\tau_2, \mathsf{b_n}) &\triangleq \Sigma x\colon \mathsf{chomp}_1(\tau_1, \mathsf{b_n}).\mathsf{chompRef}_1(\tau_2, x, \mathsf{b_n}) + \\ &\qquad \Sigma x\colon \{y : \tau_1 \mid |y.pkt_{in}| = 0\}.\mathsf{chomp}_1(\tau_2, \mathsf{b_n}) \\ \mathsf{chomp}_1(\tau_1 + \tau_2, \mathsf{b_n}) &\triangleq \mathsf{chomp}_1(\tau_1, \mathsf{b_n}) + \mathsf{chomp}_1(\tau_2, b_0) \\ \mathsf{chomp}_1(\{x : \tau \mid e\}, \mathsf{b_n}) &\triangleq \{x : \mathsf{chomp}_1(\tau, \mathsf{b_n}) \mid \mathsf{chomp}_1^{\varphi}(e, x, \mathsf{b_n})\} \\ \mathsf{chomp}_1(\tau_1[x \mapsto \tau_2], \mathsf{b_n}) &\triangleq \mathsf{chomp}_1(\tau_1, \mathsf{b_n})[x \mapsto \tau_2] \\ \mathsf{chomp}_1(\tau, \_) &\triangleq \tau\end{aligned}$$

Figure 5.17: Definition of chomp_1.

5.7 Chomp

When an instance ι is extracted, $\mathsf{sizeof}(\iota)$ bits are moved from the input bit stream to the instance. We want this process, which we call chomping, to be reflected in the type that we assign to an extract command. For example, given a header instance $y.A$ of type $A_\eta = \{f : 2\}$, and heap type $\tau = \{x : \epsilon \mid x.pkt_{in}[0{:}2] = 11\}$, moving the first two bits of the input packet into instance A is captured by the type $\{x : \epsilon \mid y.A[0{:}2] = 11\}$. Chomping is a syntactic transformation of a heap type, which is defined as a bitwise operation consisting of (1) chomp_1, the consumption of exactly one bit from pkt_{in} and (2) $\mathsf{heapRef}_1$, the refinement of the consumed bit to the extracted instance. Finally, chomp (c.f. Figure 5.16) lifts the pairwise application of chomp_1 and $\mathsf{heapRef}_1$ to be applicable to whole instances.

5.7.1 Single-bit Chomp

For chomping off one bit from a heap type, we need to update references to the length, as well as to the first bit of pkt_{in}. The removal of one bit from pkt_{in} through chomp_1 resembles the computation of a Brzozowski derivative [Brz64]. We define 1-bit chomping for heap types (chomp_1), expressions (chomp_1^e) and formulae ($\mathsf{chomp}_1^{\varphi}$).

As shown in Figure 5.17, function chomp_1 takes two arguments, a heap type τ and a bit variable $\mathsf{b_n}$ out of the set of bit variables B_n and returns a heap type. Intuitively,

$$
\begin{aligned}
\mathsf{chompRef}_1 &: \tau \times \mathcal{X} \times B_n \to \tau \\
\mathsf{chompRef}_1(\Sigma x\colon \tau_1.\tau_2, x, \mathsf{b_n}) &\triangleq \Sigma x\colon \mathsf{chompRef}_1(\tau_1, x, \mathsf{b_n}).\mathsf{chompRef}_1(\tau_2, x, \mathsf{b_n}) \\
\mathsf{chompRef}_1(\tau_1 + \tau_2, x, \mathsf{b_n}) &\triangleq \mathsf{chompRef}_1(\tau_1, x, \mathsf{b_n}) + \mathsf{chompRef}_1(\tau_2, x, \mathsf{b_n}) \\
\mathsf{chompRef}_1(\{x : \tau \mid \varphi\}, x, \mathsf{b_n}) &\triangleq \{x : \mathsf{chompRef}_1(\tau, x, \mathsf{b_n}) \mid \mathsf{chomp}_1^{\varphi}(\varphi, x, \mathsf{b_n})\} \\
\mathsf{chompRef}_1(\tau_1[y \mapsto \tau_2], x, \mathsf{b_n}) &\triangleq \mathsf{chompRef}_1(\tau_1, x, \mathsf{b_n})[y \mapsto \mathsf{chompRef}(\tau_2, x, \mathsf{b_n})] \\
\mathsf{chompRef}_1(\tau, \_, \_) &\triangleq \tau
\end{aligned}
$$

Figure 5.18: Definition of $\mathsf{chompRef}_1$.

chomp_1 removes the first bit of the input packet and replaces all references to this bit with bit variable $\mathsf{b_n}$. Syntactically, when chomping a heap type τ we need to 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 $\tau = \tau_1 + \tau_2$ chomp_1 is applied to both types τ_1 and τ_2 individually, as each branch of the choice type is describing isolated heaps of τ. In the substitution type $\tau = \tau_1[x \mapsto \tau_2]$ only τ_1 is chomped, as τ_2 only captures information relevant for evaluating refinements. In the refinement type $\tau = \{x : \tau_1 \mid \varphi\}$ τ_1 is chomped as well as the formula φ. The binder x is used in the chomping of the expression, as x is the latest leftmost binder for pkt_{in}.

Chomping a sigma type $\tau = \Sigma x\colon \tau_1.\tau_2$ is a bit more involved than the previous cases, because of the concatenation semantics of pkt_{in}. Since we only chomp off exactly one bit, we need to consider two cases. First, the input packet in the left projection τ_1 of the Σ-type contains at least one bit and second, the input packet of the first projection is empty, i.e., the length is equal to zero. In the first case, a single bit is removed from τ_1, which requires that all references to $x.pkt_{in}$ in τ_2 must be updated as τ_1 is bound to x in τ_2. This is the responsibility of function $\mathsf{chompRef}$. Otherwise, chomping could cause contradictions between refinements referencing the same component.

For example, let us consider the heap type $\Sigma x : \{y : \epsilon \mid |y.pkt_{in}| = 1\}.\{z : \epsilon \mid |x.pkt_{in}| = 1\}$, where both refinements reference the first projection. To obtain an updated type that reflects the removal of the first bit of the input packet, chomp must update both refinements accordingly, i.e., $\mathsf{chomp}_1(\Sigma x\colon \{y : \epsilon \mid |y.pkt_{in}| = 1\}.\{z : \epsilon \mid |x.pkt_{in}| = 1\}, \mathsf{b_0}) = \Sigma x\colon \{y : \epsilon \mid |y.pkt_{in}| + 1 = 1\}.\{z : \epsilon \mid |x.pkt_{in}| + 1 = 1\}$.

In the second case, we remove the bit from the input packet of the second projection. When we chomp in τ_2 we use a refinement to assert the input packet pkt_{in} of τ_1 is actually empty (e.g. $\mathsf{chomp}_1(\Sigma x\colon \epsilon.\{y : \epsilon \mid |x.pkt_{in}| = 0\}, \mathsf{b_0}) = \Sigma x\colon \{y : \epsilon \mid |y.pkt_{in}| = 0\}.\{y : \epsilon \mid |x.pkt_{in}| = 0\}$). Finally, we combine both possibilities with a choice type.

As defined in Figure 5.18, the function $\mathsf{chompRef}_1$ takes three arguments, a heap type τ, a variable x out of the set of all variables $\mathcal{X}$ and a bit variable $\mathsf{b_n}$. Intuitively, $\mathsf{chompRef}_1$ updates all references to $x.pkt_{in}$ in τ and replaces all references to this bit with bit variable b_0. It is defined recursively on all heap types and for refinement types it passes on the execution to $\mathsf{chomp}_1^{\varphi}$.

Function $\mathsf{chomp}_1^{\varphi}$ (cf. Figure 5.19) takes three arguments, a formula φ, a variable x and a bit variable $\mathsf{b_n}$. Intuitively, $\mathsf{chomp}_1^{\varphi}$ updates via function chomp_1^{e} all expressions referencing $x.pkt_{in}$ in formula φ such that references to the first bit of $x.pkt_{in}$ are replaced with bit variable b_0 and references to the length of $x.pkt_{in}$ are increased by one. Formulae true, false and $x.\iota.valid$ are not affected by chomping and negation $\neg e$

$$
\begin{aligned}
\mathsf{chomp}_1^{\varphi} &: \varphi \times \mathcal{X} \times B_n \to \varphi \\
\mathsf{chomp}_1^{\varphi}(e_1 = e_2, x, \mathsf{b_n}) &\triangleq \mathsf{chomp}_1^{e}(e_1, x, \mathsf{b_n}) = \mathsf{chomp}_1^{e}(e_2, x, \mathsf{b_n}) \\
\mathsf{chomp}_1^{\varphi}(e_1 > e_2, x, \mathsf{b_n}) &\triangleq \mathsf{chomp}_1^{e}(e_1, x, \mathsf{b_n}) > \mathsf{chomp}_1^{e}(e_2, x, \mathsf{b_n}) \\
\mathsf{chomp}_1^{\varphi}(\varphi_1 \wedge \varphi_2, x, \mathsf{b_n}) &\triangleq \mathsf{chomp}_1^{\varphi}(\varphi_1, x, \mathsf{b_n}) \wedge \mathsf{chomp}^{\varphi}(\varphi_2, x, \mathsf{b_n}) \\
\mathsf{chomp}_1^{\varphi}(\neg\varphi, x, \mathsf{b_n}) &\triangleq \neg\mathsf{chomp}_1^{\varphi}(\varphi, x, \mathsf{b_n}) \\
\mathsf{chomp}_1^{\varphi}(\varphi, \_, \_) &\triangleq \varphi
\end{aligned}
$$

Figure 5.19: Definition of $\mathsf{chomp}_1^{\varphi}$.

$$
\begin{aligned}
\mathsf{chomp}_1^{e} &: e \times \mathcal{X} \times B_n \to e \\
\mathsf{chomp}_1^{e}(|x.pkt_{in}|, y, \_) &\triangleq \begin{cases} |x.pkt_{in}| + 1 & \text{if } x = y \\ |x.pkt_{in}| & \text{otherwise} \end{cases} \\
\mathsf{chomp}_1^{e}(x.pkt_{in}, y, \mathsf{b_n}) &\triangleq \begin{cases} \langle \mathsf{b_n} \rangle @ x.pkt_{in} & \text{if } x = y \\ x.pkt_{in} & \text{otherwise} \end{cases} \\
\mathsf{chomp}_1^{e}(x.pkt_{in}[l:r], y, \mathsf{b_n}) &\triangleq \begin{cases} x.pkt_{in}[l:r] & \text{if } x \neq y \\ \langle \mathsf{b_n} \rangle & \text{if } x = y \wedge r \leq 1 \\ \langle \mathsf{b_n} \rangle @ x.pkt_{in}[0:r-1] & \text{if } x = y \wedge l = 0 \\ x.pkt_{in}[l-1:r-1] & \text{if } x = y \wedge l \neq 0 \end{cases} \\
\mathsf{chomp}_1^{e}(n + m, y, \mathsf{b_n}) &\triangleq \mathsf{chomp}_1^{e}(n, y, \mathsf{b_n}) + \mathsf{chomp}_1^{e}(m, y, \mathsf{b_n}) \\
\mathsf{chomp}_1^{e}(bv_1 @ bv_2, y, \mathsf{b_n}) &\triangleq \mathsf{chomp}_1^{e}(bv_1, y, \mathsf{b_n}) @ \mathsf{chomp}_1^{e}(bv_2, y, \mathsf{b_n}) \\
\mathsf{chomp}_1^{e}(e, \_, \_) &\triangleq e
\end{aligned}
$$

Figure 5.20: Definition of chomp_1^{e}.

and conjunction $e_1 \wedge e_2$ are standard congruence rules. For expression equality $e_1 = e_2$ and expression comparison $e_1 > e_2$, chomp_1^{e} is applied to both expressions individually.

Figure 5.20 defines how expressions are transformed as part of chomp_1^{e}. Function chomp_1^{e} takes three arguments, an expression e, a variable x and a bit variable $\mathsf{b_n}$. Intuitively, chomping expressions with $\mathsf{chomp}_1^{e}(e, x, \mathsf{b_n})$ replaces references to the first bit of $x.pkt_{in}$ in e with $\mathsf{b_n}$ and references to the length of $x.pkt_{in}$ in e are incremented by one. Expressions e that do not reference $x.pkt_{in}$ or $|x.pkt_{in}|$, like $x.pkt_{out}$, numerals n and bit vectors bv are not affected by chomping, thus $\mathsf{chomp}_1^{e}(e, x, \mathsf{b_n}) = e$. If the expression is $x.pkt_{in}$ or $|x.pkt_{in}|$, but the binder does not match with the variable passed to chomp_1^{e}, the expression remains unchanged. The cases $e_1 @ e_2$ and $e_1 + e_2$ are standard congruence rules. We increment expressions $|x.pkt_{in}|$ by one if the binder matches with the variable argument passed to chomp_1^{e}. If $e = x.pkt_{in}$, we prepend bit variable $\mathsf{b_n}$, which effectively removes the first bit. For input packet slices ($x.pkt_{in}[l:r]$), the result of chomp_1^{e} depends on the bounds l and r. If the right bound is less or equal to one, the slice contains only a single bit, so we replace the whole slice with a bit vector only containing bit variable $\mathsf{b_n}$ (e.g., $\mathsf{chomp}_1^{e}(x.pkt_{in}[0:1], x, \mathsf{b_n}) = \langle \mathsf{b_n} \rangle$). If the left bound is zero, the bit variable $\mathsf{b_n}$ is prepended and the size of the slice is reduced by

one to retain the overall size of the expression (e.g., $\mathsf{chomp}_1^e(x.pkt_{in}[0:8], x, \mathsf{b_n}) = \langle \mathsf{b_n} \rangle @x.pkt_{in}[0:7]$). If the left bound is greater than zero, both bounds are decremented by one (e.g., $\mathsf{chomp}_1^e(x.pkt_{in}[4:8], x, \mathsf{b_0}) = x.pkt_{in}[3:7]$).

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

5.7.2 Instance Refinement

With $\mathsf{heapRef}_1$ we replace the placeholder bits introduced by chomp_1 with references to the extracted instance. As presented in Figure 5.21, the function $\mathsf{heapRef}_1$ takes five arguments, a heap type τ (we define the same function also for formulae and expressions), a bit variable $\mathsf{b_n}$, a variable x, a header instance ι and a number n. Intuitively, $\mathsf{heapRef}_1(\tau, \mathsf{b_n}, x, \iota, n)$ replaces $\mathsf{b_n}$ in τ with $x.\iota[m:m+1]$, where $m = \mathsf{sizeof}(\iota) - n$.

The only place where a bit variable can occur according to the syntax of $\Pi 4$ is in the bit vector construction. Each case except for expressions $b :: bv$ will either be the identity if the input is a value or a congruence rule if the input has sub-nodes. For a expression $b :: bv$ (1) if $b \neq \mathsf{b_n}$, we keep b and continue searching in bv or (2) if $b = \mathsf{b_n}$, we replace $\mathsf{b_n}$ with $x.\iota[\mathsf{sizeof}(\iota) - n]$ and continue searching in bv.

Example Given header instance A of type $A_\eta = \{f : 2\}$ and heap type $\tau = \{x : \epsilon \mid \langle b_0 \rangle @x.pkt_{in}[0] = 10\}$, $\mathsf{heapRef}_1(\tau, b_0, y, A, 2) = \{x : \epsilon \mid y.A[0]@x.pkt_{in}[0] = 10\}$.

5.7.3 Correctness of Chomp

We can prove that our definition of the chomp operator (cf. Figure 5.16) has the desired semantics. We first define a semantic chomp operation $\mathsf{chomp}^{\Downarrow}$ that—given a heap h and a number n—removes the first n bits from the input packet in heap h:

$$\mathsf{chomp}^{\Downarrow}(h, n) \triangleq h[pkt_{in} \mapsto h(pkt_{in})[n:]]$$

We can then prove the following lemma, which 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 heap obtained after applying $\mathsf{chomp}^{\Downarrow}$ to heap h. Since chomp adds a refinement on $x.\iota$, we have to evaluate the chomped type in an environment, where x maps to the heap in which ι contains the first $\mathsf{sizeof}(\iota)$ bits from $h(pkt_{in})$. This corresponds with the intuition that chomp populates the header instance ι with the first $\mathsf{sizeof}(\iota)$ bits from the input packet.

Lemma 5.1 (Semantic Chomp). *If x does not appear free in τ, then for all heaps $h \in [\![\tau]\!]_{\mathcal{E}}$ where $|h(pkt_{in})| \geq \mathsf{sizeof}(\iota)$, there exists $h' \in [\![\mathsf{chomp}(\tau, \iota, x)]\!]_{\mathcal{E}'}$ such that $h' = \mathsf{chomp}^{\Downarrow}(h, \mathsf{sizeof}(\iota))$ where $\mathcal{E}' = \mathcal{E}[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto h(pkt_{in})[0 : \mathsf{sizeof}(\iota)]])]$.*

Proof. By unfolding the definition of chomp and by induction on the number of bits consumed. The full proof can be found in Appendix A.2.1 (Lemma A.44). □

5.8 Safety

We prove safety of $\Pi 4$ in terms of standard progress and preservation theorems. That is, well-typed programs do not get stuck and when well-typed programs are evaluated,

$$
\begin{aligned}
\mathsf{heapRef}_1 &: \tau \times \mathsf{b_n} \times \mathcal{X} \times \iota \times \mathbb{N} \to \tau \\
\mathsf{heapRef}_1(\Sigma x\colon \tau_1.\tau_2, \mathsf{b_n}, y, \iota, n) &\triangleq \Sigma x\colon \mathsf{heapRef}_1(\tau_1, \mathsf{b_n}, y, \iota, n). \\
&\qquad \mathsf{heapRef}_1(\tau_2, \mathsf{b_n}, y, \iota, n) \\
\mathsf{heapRef}_1(\tau_1 + \tau_2, \mathsf{b_n}, y, \iota, n) &\triangleq \mathsf{heapRef}_1(\tau_1, \mathsf{b_n}, y, \iota, n) + \\
&\qquad \mathsf{heapRef}_1(\tau_2, \mathsf{b_n}, y, \iota, n) \\
\mathsf{heapRef}_1(\{x : \tau \mid \varphi\}, \mathsf{b_n}, y, \iota, n) &\triangleq \{x : \mathsf{heapRef}_1(\tau, \mathsf{b_n}, y, \iota, n) \mid \\
&\qquad \mathsf{heapRef}_1(\varphi, \mathsf{b_n}, y, \iota, n)\} \\
\mathsf{heapRef}_1(\tau_1[x \mapsto \tau_2], \mathsf{b_n}, y, \iota, n) &\triangleq \mathsf{heapRef}_1(\tau_1, \mathsf{b_n}, y, \iota, n)[x \mapsto \\
&\qquad \mathsf{heapRef}_1(\tau_2, \mathsf{b_n}, y, \iota, n)] \\
\mathsf{heapRef}_1(\tau, \_, \_, \_, \_) &\triangleq \tau
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{heapRef}_1 &: \varphi \times \mathsf{b_n} \times \mathcal{X} \times \iota \times n \to \varphi \\
\mathsf{heapRef}_1(e_1 = e_2, \mathsf{b_n}, x, \iota, n) &\triangleq \mathsf{heapRef}_1(e_1, \mathsf{b_n}, x, \iota, n) = \mathsf{heapRef}_1(e_2, \mathsf{b_n}, x, \iota, n) \\
\mathsf{heapRef}_1(e_1 > e_2, \mathsf{b_n}, x, \iota, n) &\triangleq \mathsf{heapRef}_1(e_1, \mathsf{b_n}, x, \iota, n) > \mathsf{heapRef}_1(e_2, \mathsf{b_n}, x, \iota, n) \\
\mathsf{heapRef}_1(\varphi_1 \wedge \varphi_2, \mathsf{b_n}, x, \iota, n) &\triangleq \mathsf{heapRef}_1(\varphi_1, \mathsf{b_n}, x, \iota, n) \wedge \mathsf{heapRef}_1(\varphi_2, \mathsf{b_n}, x, \iota, n) \\
\mathsf{heapRef}_1(\neg\varphi, \mathsf{b_n}, x, \iota, n) &\triangleq \neg\mathsf{heapRef}_1(\varphi, \mathsf{b_n}, x, \iota, n) \\
\mathsf{heapRef}_1(\varphi, \_, \_, \_, \_) &\triangleq \varphi
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{heapRef}_1 &: e \times \mathsf{b_n} \times \mathcal{X} \times \iota \times n \to e \\
\mathsf{heapRef}_1(b :: bv, \mathsf{b_n}, x, \iota, n) &\triangleq \begin{cases} x.\iota[\mathsf{sizeof}(\iota) - n : \mathsf{sizeof}(\iota) - n + 1]@ \\ \qquad \mathsf{heapRef}_1(bv, \mathsf{b_n}, x, \iota, n) & \text{if } b = \mathsf{b_n} \\ \langle b \rangle @\mathsf{heapRef}_1(bv, \mathsf{b_n}, x, \iota, n) & \text{otherwise} \end{cases} \\
\mathsf{heapRef}_1(e, \_, \_, \_) &\triangleq e
\end{aligned}
$$

Figure 5.21: Definition of $\mathsf{heapRef}_1$.

they remain well typed. Both theorems make use of the entailment relation defined in Figure 5.13.

Theorem 5.2 (Progress). *If* $\cdot \vdash c\colon (x : \tau_1) \to \tau_2$ *and* $(I, O, H) \vDash \tau_1$, *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. The full proof can be found in Appendix A.2.1 (Theorem A.35). □

The progress theorem states that if a command is well-typed, it is either *skip* or it can take a step.

Theorem 5.3 (Preservation). *If* $\Gamma \vdash c : (x : \tau_1) \to \tau_2$ *and* $\langle I, O, H, c \rangle \to \langle I', O', H', c' \rangle$ *and* $\mathcal{E} \vDash \Gamma$ *and* $(I, O, H) \vDash_{\mathcal{E}} \tau_1$, *then there exists* $\Gamma', \mathcal{E}', x', \tau_1', \tau_2'$, *such that* $\Gamma' \vdash c' :$

$(x' : \tau_1') \to \tau_2'$ *and* $\mathcal{E}' \vDash \Gamma'$ *and* $\Gamma \subseteq \Gamma'$ *and* $\mathcal{E} \subseteq \mathcal{E}'$ *and* $(I', O', H') \vDash_{\mathcal{E}'} \tau_1'$ *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. The full proof can be found in Appendix A.2.1 (Theorem A.54). □

Preservation says that if a command c is well-typed, command c can step to c', and the starting heap entails input type τ_1, then c' is well-typed with type $(x' : \tau_1') \to \tau_2'$ for some x', τ_1' and τ_2' s.t. the stepped heap entails τ_1' and the set of heaps described by τ_2' is a subset of the heaps described by τ_2.

5.9 Related Work

Formal Reasoning for P4 Programs A number of different approaches for verifying properties of P4 programs have been proposed in recent years. p4v [Liu+18] applies classical techniques based on predicate transformer semantics to achieve monolithic verification of P4 programs. In contrast, Π4 uses dependent types and offers compositional verification. Vera [Sto+18] and Assert-P4 [Fre+18; Nev+18] are symbolic execution engines for P4. The bf4 tool [Dum+20] follows the approach pioneered in 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. P4K [KR18] provides a formal semantics of P4 in the K framework [RŞ10] and thus, can make use of the verification tools provided by the K framework. Petr4 [Doe+21] develops a formal semantics for P4 but does not itself offer verification tools. In contrast, P4RL [Shu+19] uses a dynamic approach—fuzz testing—for the verification of P4 programs.

Dependent Types There is a long history of using dependent types to capture properties of low-level code. Early work by Xi and Pfenning [XP99] showed how dependent types could be used to eliminate run-time safety checks—e.g., array bounds checks in imperative programs. Xanadu [Xi00] adds dependent typing to imperative programming, but does not capture the effect of mutations in the type. Xi and Harper later showed how dependent types could be applied to assembly code [XH01].

Deputy [Con+07] 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 type checking has no effect on the run time, and it also supports path-sensitive reasoning.

Similar to Π4, Hoare Type Theory (HTT) [NMB06] statically tracks how the heap evolves during execution. Typing of computations in HTT is similar to the dependent function types Π4 uses for commands. 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 [Nan+08], FCSL [Nan+14], and F* [Swa+16] provide dependent types for low-level imperative programming. While these type systems target general functional verification and often require manual programs-as-proofs to do so, our type system is designed with domain-specific properties of network programming in mind and is fully automatic.

Solver-Aided Tools A key focus of recent work on dependently-typed language has been on automation. This work builds on recent advances in SAT/SMT solvers and is designed to make dependent types usable by ordinary programs. A prominent example

is Liquid Haskell [RKJ08]. It extends Haskell with refinement types, but imposes restrictions to ensure the refinements remain decidable. Under the hood, all proof obligations generated by Liquid Haskell during type checking are handled by an SMT solver in a way that is transparent to the programmer. Π4 draws inspiration from Liquid Haskell's decidable refinement types. However, our SMT encoding and our proof of correctness and decidability are novel. Liquid Haskell stipulates that its refinements must be in the theory of quantifier-free integer linear arithmetic in order to be decidable. We encode types into the effectively propositional fragment of first-order logic over bit vectors, which facilitates automatic subtyping and equivalence checks.

The Prototype Verification System (PVS) [ORS92; Owr+95] is an interactive theorem prover that similar to Π4 combines dependent types and refinement types in its specification language. Also, PVS automatically extracts and solves proof obligations during type checking, but in contrast to Π4, type checking is undecidable, which requires the user to get involved in complex cases.

Formalizing Protocols Another line of work has developed language-based specifications of protocols. CMU's FoxNet project used SML to specify the behavior of an entire networking stack [Bia+94]. McCann and Chandra used a type-based approach to give abstract specifications of protocols [MC00]. Grammar-based tools such as PADS [FG05], Narcissus[Del+19], and Yakker [JMW10], enable specifying the syntax of complex, dependent formats including network protocols, and also provide tools for serializing and deserializing data.

5.10 Chapter Summary

In this chapter, we introduced Π4, a dependently-typed version of the P4 language that aims at closing the gap between type-system-based approaches and full-fledged verification tools while enabling modular verification.

Featuring a combination of expressive types including refinement types, dependent pairs, union types and explicit substitutions, Π4 is able to capture precise assumptions about valid header instances and the incoming and outgoing packet as well as dependencies between them down to the bit level. Dependent function types make it possible to statically capture the effect of executing a command in a certain program state and to relate both program states before and after the execution. Π4 addresses the challenge of retaining precise types in the presence of packet deserialization, one central domain-specific feature of the P4 language, by resorting to the novel chomp operator, which computes a type that remains after extracting bits from a packet buffer.

Π4 relies on type ascription to enable modular checking of programs. By ascribing dependent function types, it is possible to describe the requirements a command has on the context in which it is executed as well as the guarantees the command provides to the outside via the input type and output type respectively. Each ascribed type indicates a program point where type checking should occur, thus, ascribed commands can be independently type-checked. It is then sufficient to rely on the ascribed output type to check the rest of the program. As a result, as we will see in Chapter 8, Π4 is capable of expressing and verifying a variety of rich network properties in a modular way.

CHAPTER 6

An Implementation of Π4

We have built a prototype implementation of Π4 in OCaml and Z3 [MB08], which comprises approximately 7600 lines of OCaml code. Our implementation provides two frontends, one based on *Menhir*, an LR(1) parser generator for the OCaml programming language and one based on Petr4's parser [Doe+21]. While the first allows the programmer to directly use the syntax described in Section 5.3, the latter allows processing P4 programs leveraging the built-in annotation mechanism. The programmer can annotate P4 programs using `@pi4` annotations to add custom type annotations to P4 code blocks that are then checked by Π4's type checker. Under the hood, our implementation uses an encoding of Π4's types into a decidable theory of first-order logic, facilitating an SMT solver to automatically discharge the various side conditions that arise during type checking.

We start this chapter by presenting the algorithmic type system of Π4. Next, we show how to encode Π4's types into a decidable theory of first-order logic, exploiting the fact that in practice, the size of network packets are bound by the *Maximum Transmission Unit* (MTU). With the algorithmic typing rules in place, we then formally prove that given an input type that respects the bounds imposed by the MTU, we can also prove a bound on the computed output types, which ultimately allows us to prove type checking to be decidable. Finally, we discuss how the algorithmic typing rules can be optimized to obtain a more efficient encoding and provide a brief overview of our P4 frontend.

6.1 Algorithmic Typing Rules

For our implementation, we define an algorithmic version of our type system (cf. Figures 6.1 and 6.2) whose rules are mostly identical to the rules from our declarative type system. The first difference to the declarative type system is that the semantic conditions that must be checked during type checking are encoded as subtype constraints. For example, when we type-check the command $add(\iota)$, we must check that the newly added instance is not already valid in the input type τ_1, i.e., $\mathsf{Excludes}\ \Gamma\ \tau_1\ \iota$. As shown by rule T-Add-Algo in Figure 6.1, $\mathsf{Excludes}\ \Gamma\ \tau_1\ \iota$ becomes the subtype check $\Gamma \vdash \tau_1 <: \{x : \top \mid \neg x.\iota.valid\}$. Similarly, rule T-Mod and T-Remit require $\mathsf{Includes}\ \Gamma\ \tau_1\ \iota$, which becomes $\Gamma \vdash \tau_1 <: \{x : \top \mid x.\iota.valid\}$ in T-Mod-Algo and T-

T-Extract-Algo

$$\frac{\Gamma \vdash \tau_1 <: \{x : \top \mid |x.pkt_{in}| \geq \mathsf{sizeof}(\iota)\} \qquad \varphi_1 \triangleq z.pkt_{in} = z.pkt_{out} = \langle\rangle \qquad \varphi_2 \triangleq y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x}{\Gamma \vdash extract(\iota) : (x : \tau_1) \rightsquigarrow \Sigma y\colon \{z : \iota \mid \varphi_1\}.\{z : \mathsf{chomp}(\tau_1, \iota, y) \mid \varphi_2\}}$$

T-Add-Algo

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

T-Remove-Algo

$$\frac{\Gamma \vdash \tau_1 <: \iota_\approx \qquad \varphi_\iota \triangleq \forall \kappa \in dom(\mathcal{HT}).\kappa \neq \iota \implies y.\kappa = x.\kappa \qquad \varphi_{pkt} \triangleq y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out}}{\Gamma \vdash remove(\iota) : (x : \tau_1) \rightsquigarrow \{y : \top \mid \varphi_\iota \wedge \varphi_{pkt} \wedge \neg y.\iota.valid\}}$$

T-Mod-Algo

$$\frac{\Gamma \vdash \tau <: \iota_\approx \qquad \mathcal{F}(\iota, f) = \mathsf{BV} \qquad \Gamma; \tau_1 \vdash e : \mathsf{BV} \qquad \varphi_{pkt} \triangleq y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out} \qquad \varphi_\iota \triangleq \forall \kappa \in dom(\mathcal{HT}).\ \iota \neq \kappa \Rightarrow y.\kappa = x.\kappa \qquad \varphi_f \triangleq \forall g \in dom(\mathcal{HT}(\iota)).\ f \neq g \Rightarrow y.\iota.g = x.\iota.g}{\Gamma \vdash \iota.f := e : (x : \tau_1) \rightsquigarrow \{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = e[x/\mathsf{heap}]\}}$$

T-Remit-Algo

$$\frac{\Gamma \vdash \tau_1 <: \iota_\approx \qquad \varphi \triangleq z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota}{\Gamma \vdash remit(\iota) : (x : \tau_1) \rightsquigarrow \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \epsilon \mid \varphi\}}$$

T-Reset-Algo

$$\frac{\varphi_1 \triangleq z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out} \qquad \varphi_2 \triangleq z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{in}}{\Gamma \vdash reset : (x : \tau_1) \rightsquigarrow \Sigma y\colon \{z : \epsilon \mid \varphi_1\}.\{z : \epsilon \mid \varphi_2\}}$$

Figure 6.1: Algorithmic typing rules for domain-specific commands.

Remit-Algo respectively. The check $\mathsf{sizeof}_{pkt_{in}}(\tau) \geq \mathsf{sizeof}(\iota)$ required by T-Extract becomes $\Gamma \vdash \tau_1 <: \{x : \top \mid |x.pkt_{in}| \geq \mathsf{sizeof}(\iota)\}$ 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 $\hat{\tau}_1$. We then use the ascribed input type to compute an output type τ_c. Finally, we check if the computed output type τ_c is a subtype of the ascribed output type $\hat{\tau}_2$. 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.

6.2 Decidability

An essential prerequisite to prove decidability of Π4's type checking is that subtyping checks are decidable. Since we encode types into bit vectors, this follows by finite enumeration, if we can show that the bit vectors are finite. Unfortunately, the pkt_{in} and

T-Seq-Algo
$$\frac{\Gamma \vdash c_1 : (x : \tau_1) \rightsquigarrow \tau_{12} \qquad \Gamma, (x : \tau_1) \vdash c_2 : (y : \tau_{12}) \rightsquigarrow \tau_{22}}{\Gamma \vdash c_1; c_2 : (x : \tau_1) \rightsquigarrow \tau_{22}[y \mapsto \tau_{12}]}$$

T-Skip-Algo
$$\frac{\tau_2 \triangleq \{y : \tau_1 \mid y \equiv x\}}{\Gamma \vdash \mathit{skip} : (x : \tau_1) \rightsquigarrow \tau_2}$$

T-Ascribe-Algo
$$\frac{\Gamma \vdash c : (x : \hat{\tau}_1) \rightsquigarrow \tau_c \qquad \Gamma \vdash \tau_1 <: \hat{\tau}_1 \qquad \Gamma, x : \hat{\tau}_1 \vdash \tau_c <: \hat{\tau}_2}{\Gamma \vdash c \ \mathit{as}\ (x : \hat{\tau}_1) \to \hat{\tau}_2 : (x : \tau_1) \rightsquigarrow \hat{\tau}_2}$$

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

Figure 6.2: Additional algorithmic typing rules.

pkt_{out} entries describe infinite bit vectors. However, in practice, packets are bounded by the *Maximum Transmission Unit* (MTU) and therefore, we can bound the number of bits required to represent heaps described by a certain heap type. More formally, we say that a heap type τ is bounded by N in a context Γ, written $\Gamma \vdash \tau \leq N$, iff for every heap described by type τ, the length of the input packet and output packet together is less or equal to N, which is captured by the following definition:

$$\Gamma \vdash \tau \leq N \triangleq \forall \mathcal{E} \vDash \Gamma, \forall h \in [\![\tau]\!]_{\mathcal{E}}, |h(pkt_{in})| + |h(pkt_{out})| \leq N$$

It is not sufficient to just require that $|h(pkt_{in})| \leq N$ and $|h(pkt_{out})| \leq N$, because as seen in case of the *reset* command, the resulting input packet is the concatenation of the input and output packet of the input heap, i.e., $h(pkt_{out})@h(pkt_{in})$, which might violate the constraint.

As a first step, we want to prove that if a program typechecks with a bound on its input type, then we can compute that maximum number of bits that we need to encode the output type. Ideally, this would be the same bound, however, it is possible for a program to emit more bits from the incoming packet than is allowed by the MTU. So we define a helper function $\mathsf{emit}(c) \in \mathbb{N}$ that over-approximates the maximum number of bits that could be emitted along any program path in c, which is defined in Figure 6.3. Command $\mathit{remit}(\iota)$ emits n bits, where n is the size of the header instance ι. The number of emitted bits of conditionals is at most the maximum of the bits emitted in the then-branch and the else-branch. Sequences emit the sum of emitted bits from both commands and for ascribed commands we take the number of bits emitted by the command itself. All other commands do not affect the number of emitted bits.

Theorem 6.1 captures the idea 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. Note that even though the input type is constrained by the MTU, intermediate states may require more than just N bits. This theorem shows that $N + \mathsf{emit}(c)$ suffices as the maximum combined width of pkt_{in} and pkt_{out}.

$$
\begin{array}{lcl}
\mathsf{emit}(\mathit{extract}(\iota)) & \triangleq & 0 \\
\mathsf{emit}(\mathit{if}\,(b)\ c_1\ \mathit{else}\ c_2) & \triangleq & \max(\mathsf{emit}(c_1), \mathsf{emit}(c_2)) \\
\mathsf{emit}(c_1; c_2) & \triangleq & \mathsf{emit}(c_1) + \mathsf{emit}(c_2) \\
\mathsf{emit}(\iota.f := e) & \triangleq & 0 \\
\mathsf{emit}(\mathit{remit}(\iota)) & \triangleq & \mathsf{sizeof}(\iota) \\
\mathsf{emit}(\mathit{skip}) & \triangleq & 0 \\
\mathsf{emit}(\mathit{reset}) & \triangleq & 0 \\
\mathsf{emit}(\mathit{remove}(\iota)) & \triangleq & 0 \\
\mathsf{emit}(\mathit{add}(\iota)) & \triangleq & 0 \\
\mathsf{emit}(c\ \mathit{as}\ \sigma) & \triangleq & \mathsf{emit}(c)
\end{array}
$$

Figure 6.3: $\mathsf{emit}(c) \in \mathbb{N}$ computes the maximum number of bits that can be emitted along any path in c.

Theorem 6.1 (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 Theorem A.78 in Appendix A.2.3. □

Theorem 6.2 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 6.2 (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 Theorem A.73 in Appendix A.2.3. □

With Theorems 6.1 and 6.2 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, which is stated by Theorem 6.3. Theorem 6.2 allows us to equivalently show that type checking the command in the algorithmic type system—i.e., $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2'$— and that checking $\Gamma, (x : \tau_1) \vdash \tau_2' <: \tau_2$ are decidable. We can prove the former by induction on the typing derivation, while the latter follows by finite enumeration using the bounds guaranteed by Theorem 6.1.

Theorem 6.3 (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 Theorem A.79. □

$$\bigwedge_{y \in dom(\Gamma)} \mathsf{smt}(y, \Gamma(y)) \wedge \mathsf{smt}(s, \tau_1) \wedge$$
$$\forall \mathsf{consts}(\mathsf{t}, \tau_2).\mathsf{smt}(t, \tau_2) \implies \neg \mathsf{equal}(s, t)$$

Figure 6.4: Encoding of subtyping check $\Gamma \vdash \tau_1 <: \tau_2$

6.3 SMT Encoding

In our implementation, we convert every subtyping check $\Gamma \vdash \tau_1 <: \tau_2$ into a formula in the theory of uninterpreted functions and (fixed-width) bit vectors (UFBV) [WHM13]. Again, we exploit the fact that the MTU limits the number of bits network switches can receive or transmit. So in our encoding, the MTU determines the size of the bit vectors.

Our encoding is based on the semantic notion of subtyping as defined in Section 5.6. Intuitively, type τ_1 is a subtype of τ_2, i.e., $\Gamma \vdash \tau_1 <: \tau_2$, if the set of heaps described by τ_1 is a subset of the set of heaps described by τ_2. Let S be the set of heaps described by τ_1 and T the set of heaps described by τ_2. Set S is a subset of set T ($S \subseteq T$) iff for all $s \in S$ there exists a $t \in T$ such that $s = t$—i.e., $\forall s \in S.\exists t \in T.s = t$. We want the formula to hold for all assignments of s and t, so we check if the formula is valid by checking that the negation of the formula $\exists s \in S.\forall t \in T.s \neq t$ is not satisfiable (*UNSAT*).

Accordingly, the subtype check $\Gamma \vdash \tau_1 <: \tau_2$ is encoded into the formula shown in Figure 6.4. Similar to the elements in the set of heaps described by subtype τ_1, heap types bound to variables in context Γ respectively are also existentially quantified. However, in the encoding, we can eliminate the existential quantifiers by skolemization. We declare a set of global constants for all heap types (function $\mathsf{consts}(x, \tau)$), which we use to model the heaps described by the types. We declare two constants for every header instance in the header table $\mathcal{HT}$, a boolean constant capturing the validity of the header instance (e.g. `x.ethernet.valid`) and a bit vector constant for the instance data (e.g. `x.ethernet`), where the size of the bit vector corresponds to the size of the instance. We declare two bit vector constants each for the input and output packet, one capturing the length of the packet (e.g., `x.pkt_in.length`) and one capturing the contents of the packet (e.g. `x.pkt_in`). The length of both bit vectors is determined by the MTU. The length bit vector has as many bits as needed to store the value of the MTU, the length of the packet bit vector is equal to the MTU.

In the encoding, we decompose heap types according to their recursive structure. We reference the respective sub-heaps via the binders introduced by the types, which is the reason why all declared constants are prefixed with a variable. For example, assuming subtype τ_1 has the form $\Sigma x : \{y : \top \mid ...\}.\{z : \top \mid ...\}$, the encoding will declare the following set of constants:

- $\{\mathtt{s}, \mathtt{x_l}, \mathtt{x_r}, \mathtt{y}, \mathtt{z}\}.\iota_0, \ldots, \{\mathtt{s}, \mathtt{x_l}, \mathtt{x_r}, \mathtt{y}, \mathtt{z}\}.\iota_\mathtt{n}$
- $\{\mathtt{s}, \mathtt{x_l}, \mathtt{x_r}, \mathtt{y}, \mathtt{z}\}.\iota_0.\mathtt{valid}, \ldots, \{\mathtt{s}, \mathtt{x_l}, \mathtt{x_r}, \mathtt{y}, \mathtt{z}\}.\iota_\mathtt{n}.\mathtt{valid}$
- $\{\mathtt{s}, \mathtt{x_l}, \mathtt{x_r}, \mathtt{y}, \mathtt{z}\}.\mathtt{pkt\_in}, \{\mathtt{s}, \mathtt{x_l}, \mathtt{x_r}, \mathtt{y}, \mathtt{z}\}.\mathtt{pkt\_in.length}$
- $\{\mathtt{s}, \mathtt{x_l}, \mathtt{x_r}, \mathtt{y}, \mathtt{z}\}.\mathtt{pkt\_out}, \{\mathtt{s}, \mathtt{x_l}, \mathtt{x_r}, \mathtt{y}, \mathtt{z}\}.\mathtt{pkt\_out.length}$

Note, we introduce an additional binder s for the subtype and t for the supertype to be able to refer to the top-level heap, which, for example, is necessary if the top-level type

$$
\begin{aligned}
\mathsf{smt}(x, \varnothing) &\triangleq \mathsf{false} \\
\mathsf{smt}(x, \top) &\triangleq \mathsf{true} \\
\mathsf{smt}(x, \tau_1 + \tau_2) &\triangleq \mathsf{smt}(x, \tau_1) \vee \mathsf{smt}(x, \tau_2) \\
\mathsf{smt}(x, \Sigma y\colon \tau_1.\tau_2) &\triangleq \mathsf{smt}(y_l, \tau_1) \wedge \mathsf{smt}(y_r, \tau_2) \wedge \\
&\quad\ \mathsf{pktbounds}(x) \wedge \\
&\quad\ \mathsf{pktbounds}(y_l) \wedge \\
&\quad\ \mathsf{pktbounds}(y_r) \wedge \\
&\quad\ \mathsf{append}(x, y_l, y_r) \\
\mathsf{smt}(x_0, \tau_0[x_1 \mapsto \tau_1]) &\triangleq \mathsf{smt}(x_0, \tau_0) \wedge \mathsf{smt}(x_1, \tau_1) \wedge \\
&\quad\ \mathsf{pktbounds}(x_0) \wedge \\
&\quad\ \mathsf{pktbounds}(x_1) \\
\mathsf{smt}(x, \{y : \tau \mid e\}) &\triangleq \mathsf{smt}(y, \tau) \wedge \mathsf{smt\_form}(e) \wedge \\
&\quad\ \mathsf{pktbounds}(x) \wedge \\
&\quad\ \mathsf{pktbounds}(y) \wedge \\
&\quad\ \mathsf{equal}(x, y)
\end{aligned}
$$

Figure 6.5: SMT encoding of heap types.

is a choice type. For Σ-types we introduce binders for both projections, in the example x_l and x_r[1].

Encoding of heap types As defined in Figure 6.5, the function $\mathsf{smt}(x, \tau)$ encodes heap type τ into a first-order logic formula that describes the heap identified by binder x, i.e., the formula adds assertions for constants prefixed with variable x. The types $\varnothing$ and $\top$ are encoded into boolean literals *false* and *true* respectively. A choice type $\tau_1 + \tau_2$ is encoded into the disjunction between formulas for τ_1 and τ_2. The encoding of a Σ-type $\Sigma y\colon \tau_1.\tau_2$ returns a conjunction between the formulas for τ_1 and τ_2, constraints on the input and output packet (function pktbounds) as well as the encoding of the concatenation of heaps described by τ_1 and τ_2 (function append). The auxiliary functions pktbounds and append are defined in Figure 6.6. Function $\mathsf{pktbounds}(x)$ asserts for the input and output packet of heap x that the length is less than the MTU $\mathcal{M}$ and that the value of the input and output packet is constrained by the packet length. Function $\mathsf{append}(x_0, x_1, x_2)$ returns a formula that describes the concatenation of heaps x_1 and x_2, where heap x_0 is the resulting heap. An instance ι is valid in x_0, if it is either valid in x_1 or x_2. Depending on the validity of instances, function hdreq asserts that instances in x_0 are equal to instances in x_1 and x_2 respectively. Function $\mathsf{append\_pkt}(x_0, x_1, x_2, p)$ asserts that the packet p in x_0 is the concatenation of packets p in x_1 and x_2. The packet length is the sum of the packet lengths in x_1 and x_2, but at most the MTU $\mathcal{M}$. If the total length is greater than zero but the packet in x_1 is empty, the resulting packet is determined by x_2. If the packet in x_1 is not empty, the result of the concatenation is obtained by computing the bitwise OR between the packet in x_1 and the packet

[1]This is a design decision we made; instead we could have used the binder of the Σ-type for the left projection and only introduce a fresh binder for the right projection.

$$
\begin{aligned}
\mathsf{hdreq}(x, y, \iota) &\triangleq y.\iota.valid \implies x.\iota = y.\iota \\
\mathsf{equal}(x, y) &\triangleq \bigwedge_{\iota \in dom(\mathcal{HT})} x.\iota = y.\iota \wedge x.\iota.valid = y.\iota.valid \wedge \\
&\quad \bigwedge_{p \in \{pkt_{in}, pkt_{out}\}} x.p = y.p \wedge x.p.length = y.p.length \\
\mathsf{pktbounds}(x) &\triangleq \bigwedge_{p \in \{pkt_{in}, pkt_{out}\}} (x.p.length \leq \mathcal{M} \wedge \\
&\qquad (x.p.length = 0 \vee x.p < 2^{x.p.length}))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{append\_pkt}(x_0, x_1, x_2, p) &\triangleq x_0.p.length = \mathsf{min}(x_1.p.length + x_2.p.length, \mathcal{M}) \wedge \\
&\quad (x_0.p.length = 0 \vee (x_1.p.length = 0 \wedge x_0.p = x_2.p) \vee \\
&\quad (x_1.p.length > 0 \wedge x_0.p = x_1.p \mid (x_2.p \ll x_1.p.length))) \\
\mathsf{append}(x_0, x_1, x_2) &\triangleq \bigwedge_{p \in \{pkt_{in}, pkt_{out}\}} \mathsf{append\_pkt}(x_0, x_1, x_2, p) \wedge \\
&\quad \bigwedge_{\iota \in dom(\mathcal{HT})} (x_0.\iota.valid = (x_1.\iota.valid \oplus x_2.\iota.valid) \wedge \\
&\qquad \mathsf{hdreq}(x_0, x_1, \iota) \wedge \mathsf{hdreq}(x_0, x_2, \iota))
\end{aligned}
$$

Figure 6.6: Helper functions for SMT encoding.

in x_2 shifted by the length of the packet in x_1. For example, if $x_1.pkt_{in}.length = 3$ and $x_1.pkt_{in} = 000101$ and $x_2.pkt_{in} = 000110$, $x_0.pkt_{in} = 000101|(000110 << 3) = 000101|110000 = 110101$. Note, in the formalization of $\Pi 4$, the least significant bit of a bit vector is the left-most bit, while in Z3, the right-most bit is the least significant one—i.e, in the formalization the concatenation of bit vectors $x_1.pkt_{in} = 101$ and $x_2.pkt_{in} = 011$ results in $x_1.pkt_{in}@x_2.pkt_{in} = 101011$.

The encoding of a substitution type $\tau_0[x_1 \mapsto \tau_1]$ returns a conjunction of the formulas for τ_0 and τ_1 as well as packet constraints on x_0 and x_1 The encoding of a refinement type $\{y : \tau \mid \varphi\}$ returns the conjunction between the formula for heap type τ and the formula for $\Pi 4$ formula φ, as well as constraints on the input and output packet and the equality between heaps x and y (function $\mathsf{equal}(x, y)$) with regard to instances and input and output packets.

Encoding of formulae The encoding of formulae is defined in Figure 6.7. Boolean literals, conjunctions and negations are encoded into their respective counterparts. The encoding of instance validity checks simply returns the respective constant. In the encoding of expression equality and expression comparison, we distinguish between arithmetic expressions and bit vector expressions. The comparison of arithmetic expressions is defined in Figure 6.8. When comparing two arithmetic expressions a_1 and a_2, we first compute the maximum value (function max_value) these expressions can have. For numerals, this corresponds to their value, the packet length can have at most the value of the MTU $\mathcal{M}$, and the maximum value of an addition is the sum of both maximum values. Next, we compute the minimum number n of bits required to represent the maximum value. Both terms are encoded into a fixed-width bit vector of size n (function arith). We write $[m]_n$ to denote a bit vector of size n with value m.

$$
\begin{aligned}
\mathsf{smt\_form}(\mathsf{true}) &\triangleq \mathit{true} \\
\mathsf{smt\_form}(\mathsf{false}) &\triangleq \mathit{false} \\
\mathsf{smt\_form}(\varphi_1 \wedge \varphi_2) &\triangleq \mathsf{smt\_form}(\varphi_1) \wedge \mathsf{smt\_form}(\varphi_2) \\
\mathsf{smt\_form}(\neg\varphi) &\triangleq \neg\mathsf{smt\_form}(\varphi) \\
\mathsf{smt\_form}(x.\iota.\mathit{valid}) &\triangleq x.\iota.\mathit{valid} \\
\mathsf{smt\_form}(a_1 = a_2) &\triangleq \mathsf{cmp\_arith}(a_1, a_2, =) \\
\mathsf{smt\_form}(a_1 > a_2) &\triangleq \mathsf{cmp\_arith}(a_1, a_2, >) \\
\mathsf{smt\_form}(bv_1 = bv_2) &\triangleq \mathsf{cmp\_bv}(bv_1, bv_2, =) \\
\mathsf{smt\_form}(bv_1 > bv_2) &\triangleq \mathsf{cmp\_bv}(bv_1, bv_2, >)
\end{aligned}
$$

Figure 6.7: SMT encoding of expressions.

$$
\begin{aligned}
\mathsf{arith}(m, n) &\triangleq [m]_n \\
\mathsf{arith}(|x.p|, n) &\triangleq [x.p.length]_n \\
\mathsf{arith}(a_1 + a_2, n) &\triangleq \mathsf{arith}(a_1, n) + \mathsf{arith}(a_2, n) \\
\\
\mathsf{max\_value}(m) &\triangleq m \\
\mathsf{max\_value}(|x.p|) &\triangleq \mathcal{M} \\
\mathsf{max\_value}(n + m) &\triangleq \mathsf{max\_value}(n) + \mathsf{max\_value}(m) \\
\\
\mathsf{cmp\_arith}(a_1, a_2, op) &\triangleq \mathsf{let}\ m_1 = \mathsf{max\_value}(a_1)\ \mathsf{in} \\
&\quad\ \mathsf{let}\ m_2 = \mathsf{max\_value}(a_2)\ \mathsf{in} \\
&\quad\ \mathsf{let}\ \mathit{len} = \mathsf{min\_bit\_width}(\mathsf{max}(m_1, m_2))\ \mathsf{in} \\
&\quad\ \mathsf{arith}(a_1, \mathit{len})\ op\ \mathsf{arith}(a_2, \mathit{len})
\end{aligned}
$$

Figure 6.8: SMT encoding of arithmetic terms and comparison of arithmetic terms.

Since packet length terms are encoded into the respective constant, it might happen that the target size is larger than the pre-declared size of the constant's bit vector type. In this case, we zero-pad the declared constant to produce a bit vector of the desired size. The comparison itself is straightforwardly encoded into a comparison of fixed-width bit vectors

Similarly, we define the comparison of bit vectors in Figure 6.9. The main challenge in the encoding of bit vector expressions is the concatenation of two bit vectors, especially, if a reference to either the input packet or the output packet is involved, e.g., $x.pkt_{in}$@1010. While the input packet (just as the output packet) is represented by a fixed-width bit vector in the encoding, the bit vector representing the packet length actually determines how many bits of the first actually contain meaningful data. For example, let us assume that the input packet is represented by an 8-bit bit vector, but only three bits are used, i.e., `x.pkt_in` = `00000111` and `x.pkt_in.length` = `3`. We call the first the *static size*

$$
\begin{aligned}
\mathsf{static\_size}(\langle\rangle) &\triangleq 0 \\
\mathsf{static\_size}(b :: bv) &\triangleq 1 + \mathsf{static\_size}(bv) \\
\mathsf{static\_size}(bv_1 @ bv_2) &\triangleq \min(\mathsf{static\_size}(bv_1) + \mathsf{static\_size}(bv_2), \mathcal{M}) \\
\mathsf{static\_size}(x.p[l:r]) &\triangleq r - l \\
\mathsf{static\_size}(x.\iota[l:r]) &\triangleq r - l \\
\mathsf{static\_size}(x.p) &\triangleq \mathcal{M}
\end{aligned}
$$

$$
\mathsf{bv2smt}(b :: bv) \triangleq \texttt{concat}\ \mathsf{bv2smt}(bv)\ [b]_1
$$

$$
\begin{aligned}
\mathsf{bits}(bv, n) &\triangleq [\mathsf{bv2smt}\ bv]_n, n \\
\mathsf{bits}(x.p, n) &\triangleq [x.p]_n, x.p.\mathit{length} \\
\mathsf{bits}(x.p[l:r], n) &\triangleq [(\texttt{extract}\ (r-1)\ l)\ x.p]_n, r - l \\
\mathsf{bits}(x.\iota[l:r], n) &\triangleq [(\texttt{extract}\ (r-1)\ l)\ x.\iota]_n, r - l \\
\mathsf{bits}(bv_1 @ bv_2, n) &\triangleq \mathsf{let}\ v_1, n_1 = \mathsf{bits}(bv_1, n)\ \mathsf{in} \\
&\quad\ \mathsf{let}\ v_2, n_2 = \mathsf{bits}(bv_2, n)\ \mathsf{in} \\
&\quad\ \begin{cases} v_2 & \text{if } n_1 = 0 \\ [v_1 \mid (v_2 \ll n_1)]_n, n_1 + n_2 & \text{otherwise} \end{cases} \\
\mathsf{cmp\_bv}(bv_1, bv_2, op) &\triangleq \mathsf{let}\ \mathit{len} = \max(\mathsf{static\_size}(bv_1), \mathsf{static\_size}(bv_2))\ \mathsf{in} \\
&\quad\ \mathsf{let}\ v_1, n_1 = \mathsf{bits}(bv_1, \mathit{len})\ \mathsf{in} \\
&\quad\ \mathsf{let}\ v_2, n_2 = \mathsf{bits}(bv_2, \mathit{len})\ \mathsf{in} \\
&\quad\ n_1 = n_2 \wedge (n_1 = 0 \vee v_1\ op\ v_2)
\end{aligned}
$$

Figure 6.9: SMT encoding of bit vector terms

and the latter the *dynamic size* of `x.pkt_in`. Inconsiderately concatenating both bit vectors results in `101000000111`, but instead we want the bits `1010` to follow directly after the three bits embodying the contents of `x.pkt_in`, i.e., `01010111`.

In the encoding of a comparison of two bit vectors bv_1 and bv_2 (function cmp_bv), we therefore first compute the maximum static size of both bit vectors *len*. We then encode both bv_1 and bv_2 into an SMT bit vector expression of size *len* and finally the formula for bit vector comparison asserts that the dynamic sizes of both bit vectors must be the same and the size is either zero or the comparison relation holds. As defined in the top of Figure 6.9, the static size of a bit vector value corresponds to the number of bits. The static size of a concatenation of two bit vectors is the sum of the static sizes of both bit vectors. The static size of a slice is determined by the number of bits the slice comprises and the static size of a packet bit vector is the MTU $\mathcal{M}$.

The encoding of bit vector terms is defined by function $\mathsf{bits}(e, n)$ that encodes a bit vector expression e into a bit vector of size n and additionally returns the dynamic size of the respective expression. Bit vector constants are encoded into bit vectors of size n. The only peculiarity is that we have to reverse the order of bits during the encoding because, as stated above, Π4's formalization uses a different bit order than Z3. Note, because Z3 does not support empty bit vectors, we cannot straightforwardly encode

the empty bit vector $\langle\rangle$. In the encoding we therefore handle cases dealing with empty bit vectors using assertions on the packet length. The encoding of packet references simply returns the respective constant, possibly zero-extended to match size n. The dynamic size is determined by the respective packet length constant. Slices on packets and instances are encoded into a bit vector extraction. The dynamic size in these cases is statically known and results from the size of the interval. For the concatenation of two bit vectors bv_1 and bv_2 we first encode both bit vectors into an SMT expression. If the dynamic size of bv_1 is zero, the result is the result of encoding bv_2. Otherwise, the concatenated bit vector is obtained by shifting the result of encoding bv_2 by the dynamic size of the first bit vector to the left and applying the bitwise inclusive OR operator to the resulting bit vector and the result of encoding bv_1.

6.4 Optimizations

The performance of the type checker depends mainly on how fast the SMT solver is able to perform subtyping checks. This results in two levers that can be used to optimize the performance: (1) the encoding itself and (2) the number of invocations of the SMT solver during the type checking process. In the following, we will look at a total of three such optimizations.

6.4.1 Optimizing the SMT Encoding

First, we consider two optimizations to improve the SMT encoding. We discuss how we can treat Σ-types as syntactic sugar to obtain a more efficient encoding, and also how we can inline substitution types to reduce the overall complexity of generated SMT queries.

Type Equivalences

The typing rules for parsing, adding and deparsing header instances as well as the rules for commands *reset* and *skip* have one significant drawback both in the declarative and in the algorithmic type system. The output type is obtained by extending the input type, mostly by means of a Σ-type with new header instances or bits in the output packet, which are added to the heap by the execution of the command. By carrying over the input type, the computed type grows larger and larger the longer the program is that is checked, and thus, subtyping checks become more expensive in terms of the time Z3 needs to solve formulae. Furthermore, the encoding of Σ-types presented above has been shown to incur significant overhead in practice, i.e., subtyping checks that include Σ-types take significantly longer.

Fortunately, we can assign equivalent types, which allows us to eliminate Σ-types and even the chomp-operator. To eliminate Σ-types, we exploit the fact that Σ-types can be expressed using refinement types and substitution types, which is captured by Lemma 6.4. We write $\Gamma \vdash \tau_1 \doteq \tau_2$ to denote the equivalence of types τ_1 and τ_2 in context Γ, more formally:

$$\Gamma \vdash \tau_1 \doteq \tau_2 \triangleq \forall \mathcal{E} \vDash \Gamma . [\![\tau_1]\!]_{\mathcal{E}} = [\![\tau_2]\!]_{\mathcal{E}}$$

Lemma 6.4 (Rewriting Sigma Types)**.** *In any context* Γ,

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

Proof. Proof each direction separately.

($\Rightarrow$) Let $\mathcal{E} \vDash \Gamma$ and let $h \in [\![\Sigma x\colon \tau_1.\tau_2]\!]_{\mathcal{E}}$. By the semantics of heap types, we know there exists h_1 and h_2 such that $h = h_1 +\!\!+\, h_2$ and $h_1 \in [\![\tau_1]\!]_{\mathcal{E}}$ and $h_2 \in [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h_1]}$. By definition of heap concatenation, $h(pkt_{in}) = h_1(pkt_{in})@h_2(pkt_{in})$ and also $h(pkt_{out}) = h_1(pkt_{out})@h_2(pkt_{out})$. Further, $dom(h)$ is the disjoint union of $dom(h_1)$ and $dom(h_2)$ such that if $\iota \in dom(h_i)$, $h(\iota) = h_i(\iota)$ for each $i = 1, 2$ and each $\iota \in dom(\mathcal{HT})$. The result follows by definition of the semantics.

($\Leftarrow$) Let $\mathcal{E} \vDash \Gamma$. By the definition of the semantics, it suffices to show, for $h_1 \in [\![\tau_1]\!]_{\mathcal{E}}$, and $h_2 \in [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h_1]}$, and $h \in [\![\top]\!]_{\mathcal{E}[x \mapsto h_1, r \mapsto h_2]}$ such that the above refinement holds for h, that $h \in [\![\Sigma x\colon \tau_1.\tau_2]\!]_{\mathcal{E}}$. By the semantics of heap types, it suffices to show that $h = h_1 +\!\!+\, h_2$. The refinement tells us that

- $h(pkt_{in}) = h_1(pkt_{in})@h_2(pkt_{in})$ and
- $h(pkt_{out}) = h_1(pkt_{out})@h_2(pkt_{out})$.

Further, $dom(h)$ is the disjoint union of $dom(h_1)$ and $dom(h_2)$ such that if $\iota \in dom(h_i)$, $h(\iota) = h_i(\iota)$ for each $i = 1, 2$ and each $\iota \in dom(\mathcal{HT})$.

□

Since the Σ-types used in the typing rules are even more specific due to various refinements, we are able to specify equivalent types that are even further simplified and do not need to introduce additional substitution types. Lemma 6.5 shows exemplarily for rule T-Extract, how we can construct a refinement type that is equivalent to the Σ-type used for the output type. Similarly, equivalent types can be specified for the *add*, *remit*, and *reset* commands.

Lemma 6.5 (Rewrite Sigma Extract)**.** *For all* Γ, x, τ *and* ι, *if* $\Gamma \vdash \mathsf{sizeof}_{pkt_{in}}(\tau) \geq \mathsf{sizeof}(\iota)$ *and* x *does not occur free in* τ, *then*

$$\Gamma, x : \tau \vdash \Sigma y : \left\{ z : \iota \;\middle|\; \begin{array}{l} z.pkt_{in} = \langle\rangle \wedge \\ z.pkt_{out} = \langle\rangle \end{array} \right\} . \left\{ z : \mathsf{chomp}(\tau, \iota, y) \;\middle|\; \begin{array}{l} y.\iota@z.pkt_{in} = x.pkt_{in} \wedge \\ z.pkt_{out} = x.pkt_{out} \wedge \\ z \equiv_{\iota} x \end{array} \right\} \doteq \left\{ y : \top \;\middle|\; \begin{array}{l} y.\iota.valid \wedge \bigwedge_{\kappa \in dom(\mathcal{HT}) \wedge \kappa \neq \iota} y.\kappa = x.\kappa \wedge \\ y.\iota@y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out} \end{array} \right\}$$

Proof. Again, we prove both directions separately. The result follows by the semantics of heap types and the relation between the semantics of chomped types and semantic chomp ($\mathsf{chomp}^{\Downarrow}$). The full proof can be found in Appendix A.2.4 (Lemma A.80) □

Lemma 6.6 shows that an equivalent type also exists for the output type of the rule T-Skip, which does not copy the input type into the output type, but instead uses $\top$ as the base type. Figure 6.10 summarizes all optimized algorithmic typing rules.

Lemma 6.6 (Rewriting Refinement Types)**.** *For Γ, τ, ι, x, y, such that x and y do not occur free in τ,*

$$\Gamma, (x : \tau) \vdash \{y : \tau \mid x \equiv y\} \doteq \{y : \top \mid x \equiv y\}$$

Proof. Prove each direction separately.

($\Rightarrow$) Let $\mathcal{E} \vDash \Gamma, (x : \tau)$. We know $\mathcal{E} = \mathcal{E}'[x \mapsto h_1]$ such that $h_1 \in [\![\tau]\!]_{\mathcal{E}'}$. Let $h_2 \in [\![\{y : \tau \mid x \equiv y\}]\!]_{\mathcal{E}'[x \mapsto h_1]}$. Then $h_2 \in [\![\tau]\!]_{\mathcal{E}'[x \mapsto h_1]}$ and $[\![x \equiv y]\!]_{\mathcal{E}'[x \mapsto h_1, y \mapsto h_2]} = \mathit{true}$. From the latter, we can conclude that $h_2 = h_1$. To show $h_2 \in [\![\{y : \top \mid y \equiv x\}]\!]_{\mathcal{E}'[x \mapsto h_1]}$, we have to show that $h_2 \in [\![\top]\!]_{\mathcal{E}'[x \mapsto h_1]}$, which is immediate, and $[\![x \equiv y]\!]_{\mathcal{E}'[x \mapsto h_1, y \mapsto h_2]} = \mathit{true}$, which immediately follows by the fact that $h_2 = h_1$.

($\Leftarrow$) Let $\mathcal{E} \vDash \Gamma, (x : \tau)$. We know $\mathcal{E} = \mathcal{E}'[x \mapsto h_1]$ such that $h_1 \in [\![\tau]\!]_{\mathcal{E}'}$. Let $h_2 \in [\![\{y : \top \mid x \equiv y\}]\!]_{\mathcal{E}'[x \mapsto h_1]}$. Then $h_2 \in [\![\top]\!]_{\mathcal{E}'[x \mapsto h_1]}$, and $[\![x \equiv y]\!]_{\mathcal{E}[x \mapsto h_1, y \mapsto h_2]} = \mathit{true}$. Observe that $h_1 = h_2$. To show that $h_2 \in [\![\{y : \tau \mid y \equiv x\}]\!]_{\mathcal{E}'[x \mapsto h_1]}$, we must show that $h_2 \in [\![\tau]\!]_{\mathcal{E}'[x \mapsto h_1]}$ and $[\![x \equiv y]\!]_{\mathcal{E}[x \mapsto h_1, y \mapsto h_2]} = \mathit{true}$. The first follows by assumption that $h_1 \in [\![\tau]\!]_{\mathcal{E}'}$ and the fact that $h_2 = h_1$. The second immediately follows from $h_2 = h_1$.

□

Substitution Inlining

Another possibility for optimization arises with respect to the typing rule for sequencing T-Seq(-Algo). As a reminder, since the output type τ_{22} of command c_2 might depend on the output type τ_{12} of c_1, we need to memorize τ_{12} for the final output type. In the formalization we achieve this in terms of a substitution type, which is elegant for the formalization, but in practice again leads to the complexity of the computed types growing with the length of the program. This in turn leads to an increase in the complexity of the SMT queries, since as shown at the beginning of Section 6.3, additional constants are generated for each binder, for which the SMT solver must solve additional constraints.

As it turns out, in most cases it is possible to eliminate the explicit substitution while obtaining an equivalent heap type. For example, let us consider the simple Π4 program shown in the following code listing and let us assume this program typechecks with some type $(x : \tau_{in}) \to \tau_{out}$.

```
1 extract(ether);
2 skip
```

Let us further assume that the header table only contains header instance `ether`. Given type τ_{in}, the type checker computes the following type τ_c, where—starting from the input heap bound to variable x—the refinement type bound to z describes the intermediate heap obtained after executing command *extract*(`ether`).

T-Extract-Algo-Opt

$$\frac{\begin{array}{c}\Gamma \vdash \tau_1 <: \{x : \top \mid |x.pkt_{in}| \geq \mathsf{sizeof}(\iota)\} \\ \varphi_\iota \triangleq y.\iota.valid \wedge y.\iota = x.pkt_{in}[\mathsf{o}{:}\mathsf{sizeof}(\iota)] \wedge \bigwedge_{\kappa \in dom(\mathcal{HT}) \wedge \kappa \neq \iota} y.\kappa = x.\kappa \\ \varphi_{pkt} \triangleq y.\iota@y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out}\end{array}}{\Gamma \vdash extract(\iota) : (x : \tau_1) \rightsquigarrow \{y : \top \mid \varphi_\iota \wedge \varphi_{pkt}\}}$$

T-Remit-Algo-Opt

$$\frac{\Gamma \vdash \tau_1 <: \iota_\approx \qquad \varphi_{pkt} \triangleq y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out}@x.\iota}{\Gamma \vdash remit(\iota) : (x : \tau_1) \rightsquigarrow \{y : \top \mid y \equiv_\iota x \wedge \varphi_{pkt}\}}$$

T-Reset-Algo-Opt

$$\frac{\varphi_{pkt} \triangleq y.pkt_{in} = x.pkt_{out}@x.pkt_{in} \wedge y.pkt_{out} = \langle\rangle}{\Gamma \vdash reset : (x : \tau_1) \rightsquigarrow \{y : \top \mid \bigwedge_{\iota \in dom(\mathcal{HT})} \neg y.\iota.valid \wedge \varphi_{pkt}\}}$$

T-Add-Algo-Opt

$$\frac{\begin{array}{c}\Gamma \vdash \tau_1 <: \{x : \top \mid \neg x.\iota.valid\} \\ init_{\mathcal{HT}(\iota)} = v \qquad \varphi_\iota \triangleq y.\iota.valid \wedge y.\iota = v \wedge \bigwedge_{\kappa \in dom(\mathcal{HT}) \wedge \kappa \neq \iota} y.\kappa = x.\kappa \\ \varphi_{pkt} \triangleq y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out}\end{array}}{\Gamma \vdash add(\iota) : (x : \tau_1) \rightsquigarrow \{y : \top \mid \varphi_\iota \wedge \varphi_{pkt}\}}$$

T-Skip-Algo-Opt

$$\frac{}{\Gamma \vdash skip : (x : \tau) \rightsquigarrow \{y : \top \mid y \equiv x\}}$$

Figure 6.10: Optimized algorithmic typing rules for Π4.

$$\begin{aligned}\tau_c = \{y : \top \mid\ & |y.pkt_{in}| = |z.pkt_{in}| \wedge \\ & y.pkt_{in} = z.pkt_{in} \wedge \\ & |y.pkt_{out}| = |z.pkt_{out}| \wedge \\ & y.pkt_{out} = z.pkt_{out} \wedge \\ & ((\neg y.\texttt{ether}.valid \wedge \neg z.\texttt{ether}.valid) \vee \\ & (y.\texttt{ether}.valid \wedge z.\texttt{ether}.valid \wedge y.\texttt{ether} = z.\texttt{ether}))\}[z \mapsto \\ & \quad \{v : \top \mid v.\texttt{ether}.valid \wedge \\ & \qquad v.\texttt{ether}@v.pkt_{in} = x.pkt_{in} \wedge \\ & \qquad |v.pkt_{in}| + 112 = |x.pkt_{in}| \wedge \\ & \qquad |v.pkt_{out}| = |x.pkt_{out}| \wedge \\ & \qquad v.pkt_{out} = x.pkt_{out}\}]\end{aligned}$$

Suppose we want to convince ourselves what length of the input packet this type allows. First, we see that the length of the input packet is determined by the refinement $|y.pkt_{in}| = |z.pkt_{in}|$. So intuitively, in the next step we replace the information about the length of the input packet that we have available in the type bound to variable z. In

the example, the refinement $|v.pkt_{in}| + 112 = |x.pkt_{in}|$ expresses that the input packet is 112 bits shorter than it is in the input heap (x). Accordingly, we can also use this information directly to describe the length of $y.pkt_{in}$, i.e. $|y.pkt_{in}| + 112 = |x.pkt_{in}|$. We can repeat this step for all refinements referring to the substituted type z to ultimately end up with the following equivalent type.

$$\begin{aligned}\Gamma, x : \tau_{in} \vdash \tau_c \doteq \{y : \top \mid & |y.pkt_{in}| + 112 = |x.pkt_{in}| \wedge \\ & x.pkt_{in}[0{:}112]@y.pkt_{in} = x.pkt_{in} \wedge \\ & |y.pkt_{out}| = |x.pkt_{out}| \wedge \\ & y.pkt_{out} = x.pkt_{out} \wedge \\ & y.\texttt{ether}.\mathit{valid} \wedge \\ & y.\texttt{ether} = x.pkt_{in}[0{:}112]\}\end{aligned}$$

With regard to the SMT encoding, only 12 constants are generated for this type instead of 24 constants for which the SMT solver must solve constraints. If we consider longer, non-trivial programs, this difference becomes even bigger, since the calculated substitution types grow with the program length. If we directly inline the resulting substitution types during type checking of command sequences, the size of computed types remains constant.

However, there are also substitution types that cannot be inlined. For example, let us assume we are in a context, where variable x binds to type $\{x : \epsilon \mid |x.pkt_{in}| > 55 \wedge |x.pkt_{out}| > 55\}$, i.e., the length of the input and output packet together is at least 112 bits.

$$\begin{aligned}&\{y : \top \mid y.\texttt{ether}.\mathit{valid} \wedge y.\texttt{ether} = z.pkt_{in}[0{:}112]\}[z \mapsto \\ &\quad \{v : \top \mid v.pkt_{in} = x.pkt_{out}@x.pkt_{in}\}]\end{aligned}$$

If we follow a similar approach before, we end up with the following equivalent type:

$$\{y : \top \mid y.\texttt{ether}.\mathit{valid} \wedge y.\texttt{ether} = (x.pkt_{out}@x.pkt_{in})[0{:}112]\}$$

Unfortunately, Π4's current syntax prohibits expressing generic slices on bit vectors. Currently, only slices on header instances and on the input or output packet are supported. Since the type bound to x does not specify how the 112 bits distributed over the input and output packet, there exists no equivalent type by means of packet slices. As a consequence, with Π4's current syntax it is not possible to come up with an algorithm that is able to inline arbitrary substitution types. We therefore outline below how we can design an inlining algorithm specifically adapted to the types computed by Π4's type checker.

We now discuss based on the commands available in Π4 and their respective typing rules[2] which peculiarities arise for the inlining. Note, while we assume in the typing rules that we can derive the length of pkt_{in} and pkt_{out} from the respective bit vectors, we have to handle them explicitly in our implementation and therefore consider them explicitly with respect to inlining. All typing rules adhere to the same structure, i.e., the respective output types define (1) which part of the heap remains unchanged and (2) which part of the heap has changed. The former consists of simple equalities such as $y.pkt_{in} = x.pkt_{in}$, which can be easily replaced syntactically, as we saw in the initial

[2]We consider the optimized algorithmic typing rules presented in Figure 6.10.

example. We therefore focus mainly on the refinements that describe the changes to the heap.

reset The output type of rule T-Reset-Algo-Opt manifests the example which we have discussed before that cannot be inlined given the current syntax of Π4. We therefore keep the substitution type around and only inline preceding and subsequent commands. For example, if we have a program $p; reset; s$, the resulting heap type is of the form $(\tau_s[y \mapsto \tau_r])[z \mapsto \tau_p]$, with τ_s being the inlined heap type for all subsequent commands, τ_r being the inline heap type for command *reset* and τ_p being the inlined heap type of all preceding commands.

c_1 *as* $(x : \tau_1) \to \tau_2$ As the ascribed output type τ_2 can be an arbitrary heap type, it may also contain statements that are not inlinable. Since there is currently no trivial solution to inline ascribed types as this would require a generic inlining approach, we skip the inlining for such commands. However, for a command c_1 *as* $(x : \tau_1) \to \tau_2; c_2$ it is still worth to inline the heap type for c_1, even though it will ultimately be replaced by the ascribed type τ_2.

skip Since rule T-Skip-Algo-Opt only asserts that the heap remains unchanged, we can straightforwardly inline all equalities as described before.

$add(\iota)$ The changes introduced by typing rule T-Add-Algo-Opt are captured by the following two refinements: $y.\iota.valid$ and $y.\iota = v$, where v is a bit vector initialized with a statically known value such as $\langle 0 \ldots 0 \rangle$. We can inline command sequences of the form $add(\iota); c_2$ by replacing validity checks referencing the substitution with the boolean literal *true*, and references to instance ι can be replaced with the value of v. If preceded by another command, i.e. $c_1; add(\iota)$, nothing needs to be inlined because the two expressions do not depend on the input type.

$remove(\iota)$ Analogous to the inlining of command $add(\iota)$, we can inline the refinement $\neg y.\iota.valid$ by replacing validity checks referencing the substitution with the boolean literal *false*.

$extract(\iota)$ If we leave out all refinements capturing the part of the heap that remains unchanged, the following type describes the changes introduced by command $extract(\iota)$, where $r = \mathsf{sizeof}(\iota)$.

$$\begin{aligned}\{y : \top \mid\ & y.\iota.valid \wedge \\ & y.\iota = x.pkt_{in}[0{:}r] \wedge \\ & x.pkt_{in}[0{:}r] @ y.pkt_{in} = x.pkt_{in} \wedge \\ & |y.pkt_{in}| + r = |x.pkt_{in}|\}\end{aligned}$$

As before, references to $y.\iota.valid$ are replaced by boolean literal *true*. References to $y.\iota$ are replaced by sub-slices on $x.pkt_{in}[0{:}r]$, depending on the fields defined for instance ι. As $extract(\iota)$ is the only command except for *reset* that alters pkt_{in}, and since we do not inline *reset* commands, the only interesting case arises from inlining two consecutive $extract(\iota)$ commands, for example, $extract(\iota_1); extract(\iota_2)$. With $r_1 = \mathsf{sizeof}(\iota_1)$, $r_2 = \mathsf{sizeof}(\iota_2)$, and binder x referencing the input type, the resulting substitution type with only the refinements on the input packet looks as

follows.

$$\{y : \top \mid z.pkt_{in}[0{:}r_2]@y.pkt_{in} = z.pkt_{in} \wedge \\ |y.pkt_{in}| + r_2 = |z.pkt_{in}|\}[z \mapsto \\ \{v : \top \mid x.pkt_{in}[0{:}r_1]@v.pkt_{in} = x.pkt_{in} \wedge \\ |v.pkt_{in}| + r_1 = |x.pkt_{in}|\}]$$

The inlined heap type captures that after executing the two consecutive extract commands, $x.pkt_{in}[r{:}]$ remains left of the input packet, which is captured by the following type:

$$\{y : \top \mid x.pkt_{in}[0{:}r_1]@x.pkt_{in}[r_1{:}r_1 + r_2]@y.pkt_{in} = x.pkt_{in} \wedge \\ |y.pkt_{in}| + r_1 + r_2 = |x.pkt_{in}|\}$$

with $r = r_1 + r_2$

$$\doteq \{y : \top \mid x.pkt_{in}[0{:}r]@y.pkt_{in} = x.pkt_{in} \wedge |y.pkt_{in}| + r = |x.pkt_{in}|\}$$

remit(ι) We can argue similarly as before for command *remit*(ι), which is the only command besides *reset* that modifies the output packet. Accordingly, the most interesting case results from a sequence of remit commands. Again, the following substitution type captures the changes to the heap with respect to the output packet after executing *remit*(ι_1); *remit*(ι_2).

$$\{y : \top \mid y.pkt_{out} = z.pkt_{out}@z.\iota_2 \wedge |y.pkt_{out}| = |z.pkt_{out}| + \mathsf{sizeof}(\iota_2)\}[z \mapsto \\ \{v : \top \mid v.pkt_{out} = x.pkt_{out}@x.\iota_1 \wedge |v.pkt_{out}| = |x.pkt_{out}| + \mathsf{sizeof}(\iota_1)\}]$$

After inlining, we obtain the following type, where again $r = r_1 + r_2$.

$$\{y : \top \mid y.pkt_{out} = x.pkt_{out}@x.\iota_1@x.\iota_2 \wedge |y.pkt_{out}| = |x.pkt_{out}| + r\}$$

$\iota.f := e$ Field assignments are the only commands that operate at the level of header fields. As captured by typing rule T-Mod-Algo, in the output heap only the value of field f of instance ι is changed, while the rest of the heap remains unchanged. This is captured by the formula $y.\iota[l : r] = e[x/\mathsf{heap}]$, where l and r are the respective field bounds. The inlining itself is mostly straightforward, because as we have seen before, we can simply substitute all references to the field respectively instance slice with the assigned expression. Note, however, that we must decompose refinements that refer to the full instance. For example, given header instance h with shape $\{a : 4; b : 4; c : 2\}$ and program $\mathtt{h.b} := 0b1011; skip$, we encounter the following substitution type.

$$\{y : \top \mid y.\mathtt{h} = z.\mathtt{h} \wedge \ldots\}[z \mapsto \{v : \top \mid v.\mathtt{h}[4{:}8] = 0b1011 \wedge v.\mathtt{h}[0{:}4] = x.\mathtt{h}[0{:}4] \ldots\}]$$

To obtain the inlined type, we must decompose $z.\mathtt{h}$ properly to end up with type $\{y : \top \mid y.\mathtt{h} = x.\mathtt{h}[0{:}4]@0b1011@x.\mathtt{h}[8{:}10]\}$.

if(φ) c_1 *else* c_2 Regardless of whether we consider programs c; *if*(φ) c_1 *else* c_2 or alternatively *if*(φ) c_1 *else* c_2; c, we must inline the type for command c in a union type (i.e., $(\tau_{then} + \tau_{else})[z \mapsto \tau_c]$) or we must inline a union type in the type for command c (i.e. $\tau_c[z \mapsto (\tau_{then} + \tau_{else})]$). In both cases, both branches of the union type can be inlined independently as described before. The end result in both cases is again a union type.

```
1  @pi4("(Parser;Ingress) as (x:{y:ϵ|y.pkt_in.length > 304}) ->
       {z:⊤|!z.vlan.valid}")
2  parser Parser(packet_in packet, out headers hdr, ...) {
3    state start {
4      packet.extract(hdr.ethernet);
5      transition select(hdr.ethernet.etherType) {
6        0x0800: parse_ipv4;
7        0x8100: parse_vlan;
8        default: accept;
9      }
10   }
11   state parse_vlan {
12     packet.extract(vlan);
13     transition select(hdr.vlan.etherType) {
14       0x0800: parse_ipv4;
15       default: accept;
16     }
17   }
18   state parse_ipv4 {
19     packet.extract(hdr.ipv4);
20     transition accept;
21   }
22 }
```

Figure 6.11: VLAN decapsulation example.

6.4.2 Reducing the Number of SMT Solver Invocations

If we look at the reasons for invoking the SMT solver during type checking, the most common reason is to check the validity of headers. By strictly following the formalization with the implementation, many of these checks become redundant. For example, let us consider a somewhat realistic P4 program shown in Figures 6.11 and 6.12. The parser first parses the Ethernet header, optionally followed by VLAN and IPv4. If the IPv4 header is valid in the ingress, the program then performs a basic forwarding operation and if the VLAN header is valid, it removes the VLAN header. As indicated by the `@pi4` annotation in Line 1 of Figure 6.11, we want to check the whole program, i.e., the parser followed by the ingress with type $(x : \{y : \epsilon \mid |y.pkt_{in}| > 304\}) \rightarrow \{z : \top \mid \neg z.\texttt{vlan}.\mathit{valid}\}$. This type asserts that starting in a heap where no headers are valid, and the input packet provides enough bits to extract all three header instances, we always end up in a heap for which it is guaranteed that the VLAN header is not present. The annotation mechanism provided by our P4 frontend will be explained in more detail in the next section.

Translating the P4 program into Π4's syntax and type checking the program with the annotated type, takes about 12 seconds[3], of which roughly 68% are used to check the validity of headers. A total of 18 validity checks are performed, each one resulting

[3] Assuming an MTU of 1500 bytes

```
control Ingress(inout headers hdr, ...) {
  action ipv4_forward(bit<48> dstAddr, bit<9> port) {
    standard_metadata.egress_spec = port;
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = dstAddr;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
  }
  action vlan_decap() {
    hdr.ethernet.etherType = hdr.vlan.etherType;
    hdr.vlan.setInvalid();
  }
  table ipv4_lpm {
    key = { hdr.ipv4.dstAddr: lpm; }
    actions = { ipv4_forward; }
  }
  apply {
    if (hdr.ipv4.isValid()) {
      ipv4_lpm.apply();
    }
    if (hdr.vlan.isValid()) {
      vlan_decap();
    }
  }
}
```

Figure 6.12: VLAN decapsulation example continued.

in a call to the SMT solver. If we look at the program more closely, it is noticeable that many of the checks are indeed redundant. For example, the parser guarantees that the Ethernet header is valid. If we cache this information, we save a call to the SMT solver for each subsequent read or write access to the Ethernet header, which are six in total for this example program. In total, we can save all but two solver calls for the example program this way.

Implementing this cache is mostly straightforward. If we successfully typecheck an *extract* or *add* command, we store the information that on the current program path the respective header instance is valid. After a *remove* command, we invalidate the instance in the cache. For conditionals, we apply a simple heuristic to the condition. If it consists of simple validity checks such as `hdr.ipv4.isValid()`, we add this information to the cache used for the *then* and *else* branches accordingly. Since the validity assumptions can diverge across both branches, we have to merge them for subsequent commands of conditionals. If both branches provide the same validity information for a specific instance, we keep the entry in the cache, otherwise we remove entries. After checking a *reset* command, we store the information that all header instances are invalid, and type ascriptions make it necessary to clear the entire cache. The reason for the latter is that when applying a type ascription, we check that the computed output type τ_c is a subtype of the ascribed output type τ_{asc}. This in turn means that the ascribed output type may

$$\alpha ::= \beta \; as \; (x : \tau) \to \tau \qquad \text{(annotations)}$$
$$\beta ::= P \mid C \mid reset \mid \beta; \beta \mid \beta \; as \; (x : \tau_1) \to \tau_2 \qquad \text{(annotation bodies)}$$

Figure 6.13: Syntax of `@pi4` annotations.

```
1 @pi4("(MyParser;MyIngress;MyDeparser) as (x:τ1) → τ2")
2 parser MyParser(packet_in p, ...) { ... }
3 control MyIngress(inout headers hdr, ...) { ... }
4 control MyDeparser(packet_out packet, ...) { ... }
```

Figure 6.14: The annotation on line 1 instructs the frontend to check the full pipeline consisting of the parser, ingress control and deparser with type $(x : \tau_1) \to \tau_2$.

be less specific than the actual type and as such we cannot transfer our knowledge about instance validity in τ_c to τ_{asc}. When performing the validity checks, we first check the cache for an entry. If no entry is available, we have to resort to the SMT solver and store the computed result in the cache.

6.5 P4 Frontend

With our P4 frontend we are able to automatically translate a part of the P4 language into Π4's syntax. In order to be able to flexibly check P4 programs for different properties, we use the annotation mechanism of the language to specify which parts of the program should be checked with which type. For this we use the custom annotation `@pi4("α")`, whose syntax is defined in Figure 6.13. An annotation α always consists of an annotation body β ascribed with a dependent function type. The annotation body β is either the name of a parser P or a control C, *reset*, a sequence of two annotation bodies, or again an ascribed annotation body. For example, the annotation in Line 1 of Figure 6.14 expresses that the complete pipeline described by the parser *MyParser*, the ingress control *MyIngress*, and the deparser *MyDeparser* should be checked with type $(x : \tau_1) \to \tau_2$. We allow the programmer to use certain unicode symbols in annotations, e.g., $\top$ or ϵ, but also more convenient notations, for example, && and || for conjunction and disjunction (instead of $\wedge$ and $\vee$) respectively. The parser and control blocks are translated into a Π4 command c and likewise *reset*, $\beta; \beta$ and $\beta \; as \; (x : \tau_1) \to \tau_2$ are translated into the corresponding Π4 commands.

Since Π4 is an imperative language, we unroll parsers during translation. We model match-action tables using an encoding similar to the one used in p4v [Liu+18], where we create an extra header that captures the match keys and selected action and action parameters. Figure 6.15 shows how table `routing_v6` declared in control `Forwarding` is translated. The table header type declaration in the lower half of Figure 6.15 contains a one bit field `act` for modeling the choice of the actions, since a single bit is sufficient to represent two actions. Field `set_next_id_next_id` represents the action parameter `next_id` for action `set_next_id` and field `ipv6_dst_addr_key` models the match key of table `routing_v6`. The table is then translated into a set of conditional commands, where the conditions encode the matching on the key and the non-deterministic choice of actions. There are limitations in the translation particularly with regard to

```
control Forwarding(...) {
  action set_next_id(bit<32> next_id) {
    meta.next_id = next_id;
  }
  table routing_v6 {
    key = { hdr.ipv6.dst_addr: exact; }
    actions = {
        set_next_id; NoAction;
    }
    default_action = NoAction();
  }
}
```

```
header_type Forwarding_routing_v6_table_t {
  act: 1;
  set_next_id_next_id: 32;
  ipv6_dst_addr_key: 128;
}
header Forwarding_routing_v6_table :
    Forwarding_routing_v6_table_t
if(Forwarding_routing_v6_table.ipv6_dst_addr_key == ipv6.
    dst_addr) {
  if(Forwarding_routing_v6_table.act == 0b0) {
    meta.next_id := Forwarding_routing_v6_table.
        set_next_id_next_id
  } else {
    if(Forwarding_routing_v6_table.act == 0b1) {
      skip
    }
  }
}
```

Figure 6.15: Translation of P4 tables into Π4: P4 table declaration (top) and result of translation (bottom)

externs such as stateful elements or hash computations, since these cannot currently be modeled with the syntax of Π4.

6.6 Chapter Summary

This chapter outlined the underlying details of the implementation of Π4's type checker. We started by defining the algorithmic version of Π4's type system. We exploited the fact that network switches are limited by the *Maximum Transmission Unit* (MTU) in the size of packets they can receive and transmit, solving the problem that our heap types describe possibly unbounded bit vectors. We were able to prove an upper bound on the

size of computed output types under the assumption that the input type as well as all ascribed types are bound by the MTU as well. Ultimately, this allowed us to prove our type checking algorithm to be decidable. The MTU also allows us to encode subtyping checks into a formula in the theory of uninterpreted functions and fixed-width bit vectors (UFBV), facilitating automated subtyping checks. Finally, we discussed several optimizations to speed up solving SMT queries in terms of simpler encodings, inlining substitution types and by reducing the number of SMT solver invocations by caching assumptions about header validity.

A formal proof that our encoding is correct is left for future work. Since the performance of the SMT solver also depends heavily on the size of the bit vectors used, a possible further optimization is to adapt the number of bits to the size of heap types used to check a program. However, this requires being able to compute a bound for any heap type. Currently, it is unclear if that is generally possible. In case it is, this would also allow us to remove the constraint from our decidability theorem that every ascribed type must be bounded.

Part III

Evaluation

CHAPTER 7

Header Validity Bugs in Real-world Programs

In Chapter 3, we identified five syntactic constructs of the P4 language that are vulnerable to header validity bugs. In Chapter 4, we then introduced SafeP4, a type system that statically ensures that all headers that are read or written, are valid. In this chapter, we want to answer whether SafeP4's type system is able to identify the different types of header validity bugs in real-world P4 programs. We also investigate how often these bugs occur in practice and how much effort is required by a programmer to fix header validity bugs. Accordingly, we formulate the following three research questions:

RQ1 Is SafeP4 able to detect header validity bugs in real-world programs?

RQ2 How often do the different kinds of bugs occur in practice?

RQ3 What is the overhead for fixing these bugs?

To evaluate the type system proposed in Chapter 4, we implemented P4Check, a tool that automatically checks whether P4 programs comply with the rules presented in Section 4.3 and reports the respective violations if not. Our implementation uses the frontend provided by p4v [Liu+18] and is able to cover the whole $P4_{14}$ language. We used P4Check to check 15 open source programs that differ in size and complexity, ranging from 143 lines of code to 9600 lines of code. We selected these programs based on the following criteria: (1) each program must be open-source, (2) publicly available on GitHub, (3) compile without errors and (4) contain either industrial or academic code that implements standard or novel network functionality, i.e., we excluded programs primarily used for teaching purposes.

Out of the 15 programs, only four did not violate any of our typing rules and passed the checker. These were primarily implementations of simple routers or DDoS mitigation mechanisms, only consisting of a few lines of code (188-635 lines) and only accepting a few packet types. For the remaining 11 programs, our tool found a total of 418 violations. The high number of violations stems from the fact that each individual violation is counted, even if multiple violations share the same root cause. For example, if a single

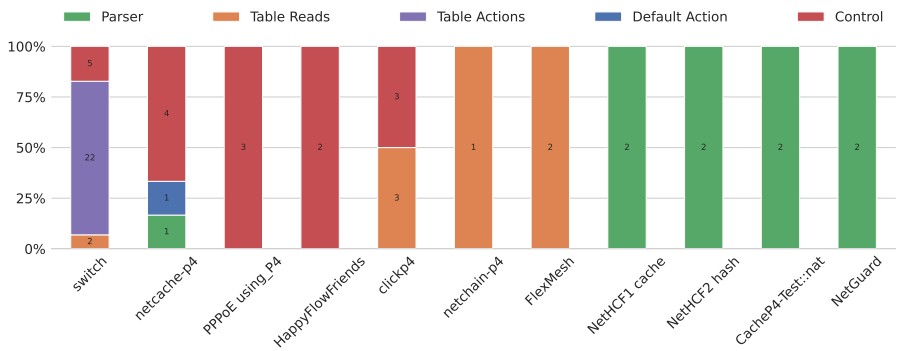

Figure 7.1: Proportional frequencies of each bug type per-program. The raw number of bugs for each program and category is reported on each stacked bar.

action modifies the source address and destination address field of an IPv4 header in a context that cannot prove that IPv4 is valid, then both references will be reported as violations, even though they are due to the same control bug. We therefore use another metric inspired by the metric proposed in [Kle+18]. We relate the number of bugs in each program to the number of bug fixes required for the program to successfully type-check. Using this metric, we counted 58 bugs.

We classified the bugs according to the categories presented in Chapter 3. Figure 7.1 shows which bug type occurs how often per program and Figure 7.2 shows how often the bugs from the different categories occur overall. It is particularly noticeable that although table action bugs occur most often numerically (22 times), there are only occurrences in one program (*switch.p4*). The reason for this is that *switch.p4* heavily relies on correct control plane configurations. In contrast, there were only nine respectively eight occurrences of parser bugs and table read bugs across five programs. Compared to p4v [Liu+18], P4Check was unable to detect any default action bugs in *switch.p4*, while p4v reported many of such bugs, which has two reasons. First, p4v allows programmers to specify complex properties that involve fine-grained conditions on tables and the relationships between tables. In contrast, SafeP4 makes various assumptions that rule out a variety of bugs, including some default action bugs. Second, our repairs are often coarse-grained, potentially forcing stronger guarantees on the program than necessary. In contrast, p4v uses first-order logic annotations, which allows the programmer to formulate the weakest and hence more complex assumptions.

7.1 Detecting and Repairing Bugs

We will now look at how P4Check detects bugs from the categories identified in Chapter 3 and how the repairs affect the computed types.

Parser bugfix We have shown an example of a parser bug in Figure 3.1. The bug occurs because the ingress expects packets where the IPv4 header and the TCP header are valid, but the parser does not guarantee that only such packets are successfully parsed. Instead, program execution switches to the ingress control, when the parser encounters an unexpected header.

Our type system is able to detect this bug because the ingress expects packets of type `ethernet · ipv4 · tcp`, i.e., it must be guaranteed that all three packet headers Ethernet,

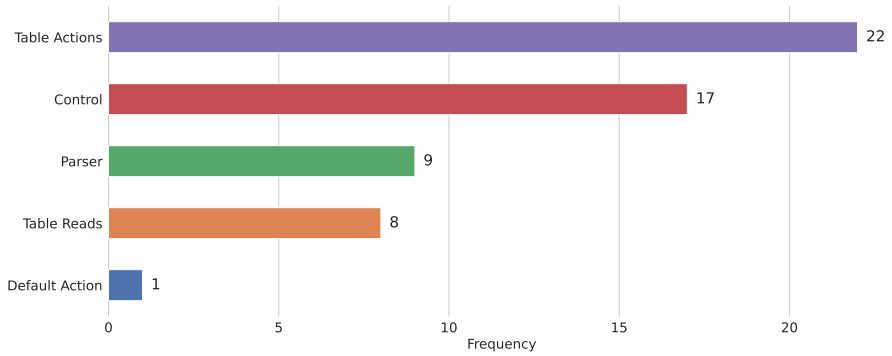

Figure 7.2: Frequency of each bug across all programs. The raw number of bugs in each category is reported to the right of the bar.

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

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

Figure 7.3: Curated output from P4Check for the parser bug in NetHCF before (above) and after (below) modifying `parse_ethernet`

IPv4 and TCP are valid on all program paths. However, the parser only produces packets of type $\texttt{ethernet} \cdot (1 + \texttt{ipv4} \cdot (1 + \texttt{tcp}))$, which means that only Ethernet is guaranteed to be valid, while the IPv4 header and the TCP header are both optional. Accordingly, P4Check reports each reference to the IPv4 header and TCP header as a violation of the type system as shown in the top half of Figure 7.3. The ubiquity of the reports intimates a mismatch between the parsing and the control types, which gives the programmer a hint as how to fix the problem.

If we fix the program step-by-step and replace in `parse_ethernet` the default clause with a parser exception as shown in Line 10 of Figure 3.2, the parser henceforth guarantees that Ethernet and IPv4 are valid in the ingress, i.e, parsed packets have type $\texttt{ethernet} \cdot \texttt{ipv4} \cdot (1 + \texttt{tcp})$. If we run the tool again, all violations related to the IPv4 header are removed from the output, as shown in the bottom of Figure 7.3. Additionally applying the second fix (cf. Line 18 of Figure 3.2) causes P4Check to output no violations, since the type upon entering the ingress control is $\texttt{ethernet} \cdot \texttt{ipv4} \cdot \texttt{tcp}$, resulting in

```
port.p4, line 248, cols 8-24: warning: assuming either vlan_tag_[0]
    matched as valid or vlan_tag_[0].vid wildcarded
port.p4, line 250, cols 8-24: warning: assuming either vlan_tag_[1]
    matched as valid or vlan_tag_[1].vid wildcarded
```

```
fabric.p4 line 42, cols 41-67: warning: assuming fabric_header_cpu
    matched as valid for rules with action terminate_cpu_packet
fabric.p4, line 57, cols 17-54: warning: assuming
    fabric_header_unicast matched as valid for rules with action
    terminate_fabric_unicast_packet
fabric.p4, line 81, cols 17-56: warning: assuming
    fabric_header_multicast matched as valid for rules with action
    terminate_fabric_multicast_packet
```

Figure 7.4: Warnings printed after fixing switch.p4's reads bug (top), and its actions bug (bottom)

all subsequent references to the IPv4 and TCP headers being safe.

Control bugfix Recall that a control bug occurs when the incoming type presents a choice between two instances, but subsequent code expects one of the instances to be valid. The control bug example in Figure 3.4 uses a parser that produces the type $\Theta = \texttt{ethernet} \cdot (1 + \texttt{ipv4} \cdot (1 + \texttt{udp} \cdot (1 + \texttt{nc\_hdr} \cdot \tau) + \texttt{tcp}))$, where τ is a type for caching operations. Especially, this type suggests that `Includes` Θ `nc_hdr` does not hold, however, controls `process_cache` and `process_value` only typecheck in contexts where `Includes` Θ `nc_hdr` is true.

Accordingly, P4Check reports type violations at every reference to `nc_hdr`. Fixing this error is simply a matter of wrapping both calls to `process_cache()` and `process_value()` in a validity check as demonstrated in the top right of Figure 3.4. As a result, the type inside the validity check becomes $\Theta = \texttt{ethernet} \cdot \texttt{ipv4} \cdot \texttt{udp} \cdot \texttt{nc\_hdr} \cdot \tau$ and thus `Includes` Θ `nc_hdr` is always true. As NetCache handles TCP and UDP packets as well as its special-purpose packets, we can't include the application of the IPv4 routing table in the validity check. This is another instance of a parser bug, as the type does not guarantee `Includes` Θ `ipv4`, which is required by table `ipv4_route`.

Table reads bugfix As shown in Figure 3.5, table reads errors occur when a header h is included in the *reads* declaration of a table t with match kind k, and h is not guaranteed to be valid at the call site of t, and if $h \notin \mathit{valid\_reads}(t)$ or the match-kind of $k \neq \mathit{ternary}$. In the case of the `port_vlan_mapping` table in Figure 3.6, there is a valid bit for both `vlan_tag_[0]` and `vlan_tag_[1]`, both of which are followed by *exact* matches. We fixed this bug by using a *ternary* match-kind instead, which allows the use of wildcard matching, since when a field is matched with a wildcard, the table does not attempt to compute the value of the *reads* expression.

However, this fix assumes that the controller is well-behaved and fills the table entry for `vlan_tag_[0].vid` with a wildcard whenever `vlan_tag_[0]` is matched as invalid (and similarly for `vlan_tag_[1]`). This also what SafeP4's type system does, with its maskable checks in typing rule T-Apply. P4Check prints warnings describing these assumptions to the programmer as shown in the top of Figure 7.4, giving them guidelines against which to check their control plane implementation.

Table action bugfix As exemplified with table `fabric_ingress_dst_lkp` in Figure 3.6, table actions bugs occur when at least one action cannot be safely executed in all scenarios. In the example, the parser parses exactly one of the three headers (1) `fabric_hdr_cpu`, (2) `fabric_hdr_unicast` and (3) `fabric_hdr_multicast`. Thus, when the table is applied at type Θ, exactly one of `Includes` Θ `fabric_hdr_i` for $i \in \{\texttt{cpu}, \texttt{unicast}, \texttt{multicast}\}$ will hold. Now, the action `term_cpu_packet` type-checks only with the (nonempty) type `Restrict` Θ `fabric_hdr_cpu`, and the actions `term_fabric_`i`_packet` only typecheck with the (nonempty) types `Restrict` Θ `term_fabric_`i`_packet` for i = `unicast`, `multicast`

P4Check suggests that this is the cause of the bug since it reports type violations for all references to these three headers in the control paths following from the application of `fabric_ingress_dst_lkp`. As shown in the bottom of Figure 3.6, the optimal fix is to augment the *reads* declaration to include a validity check for each contentious header. We then assume that the controller is well-behaved enough to only call actions when their required headers are valid, allowing us to typecheck each action in the appropriate type restriction. Again, P4Check alerts the programmer whenever it makes such an assumption. We show these warnings for the fixed version of `fabric_ingress_dst_lkp` in the bottom part of Figure 7.4

Default action bugfix Recall that default action bugs, such as the one shown in Figure 3.7, occur when a programmer creates a wrapper table for an action that modifies the type, and forgets to force the table to call that action when the packet misses. Table `add_value_header_1` wraps action `add_value_header_1_act`, which itself executes `add_header(nc_value_1)`. The default action, when left unspecified, is `nop`, which means that if the pre-application type was Θ, then the type after applying the table is $\Theta' = \Theta + \Theta \cdot \texttt{nc\_value\_1}$. For this type `Includes` Θ' `nc_value_1` is false, hence, P4Check reports every subsequent reference (on this code path) to `nc_header_1` to be a type violation. Fixing this bug by setting the default action to `add_value_1` makes the post-application type $\Theta \cdot \texttt{nc\_value\_1} + \Theta \cdot \texttt{nc\_value\_1} = \Theta \cdot \texttt{nc\_value\_1}$, and therefore `Includes` Θ' `nc_value_1` is true, which allows the subsequent code to typecheck.

7.2 Overhead

It is important to evaluate two kinds of overhead when considering a static type system: overhead on programmers and on the underlying implementation. Typically, adding a static type system to a dynamic type system requires more work for the programmer—the field of gradual typing is devoted breaking this task into smaller commit-sized chunks [Cam+17]. Surprisingly, in our experience, migrating real-world P4 code to pass the SafeP4 type system only required modest programmer effort.

To qualitatively evaluate the effort required to change an unsafe program into a safe one using our type system, we manually fixed all detected bugs. The programs that had bugs required us to edit between 0.10% and 1.4% of the lines of code. The one exception was *PPPoE_using_P4*, which was a 143 line program that required 6 line-edits (4%), all of which were validity checks. Conversely, *switch.p4* required 34 line edits, the greatest observed number, but this only accounted for 0.37% of the total lines of code in the program. Each class of bugs has a simple one-to-two line fix, as described in Section 7.1: adding a validity check, adding a default action, or slightly modifying the parser. Each of these changes was straightforward to identify and simple to make.

Another possible concern is that extending tables with extra read expressions, or adding run-time validity checks to controls, might impose a heavy cost on implementations, especially on hardware. Although we have not yet performed an extensive study of the impact on compiled code, based on the size and complexity of the annotations we added, we believe the additional cost should be quite low. We were able to compile our fixed version of *switch.p4* program to the Tofino architecture [Bos18] with only a modest increase in resource usage. Overall, given the large number of potential bugs located by P4Check, we believe the assurance one gains about safety properties by using a static type system makes the costs well worth it.

7.3 Chapter Summary

In this chapter, we have shown how we can use SafeP4's type system in practice to check real P4 programs for header validity bugs according to the taxonomy presented in Chapter 3 (*RQ1*). For this purpose we have shown exemplarily which effects buggy programs as well as the repaired versions of these programs have on the computed types. Validity bugs were present in the majority of the programs we examined, which clearly indicates that validity bugs are ubiquitous in practice. Number-wise, table-action bugs occurred most frequently, but only in a single program, which was significantly larger than the others in terms of the number of lines of code. If we look at how often certain bug categories appear in different programs, control and parser bugs predominate (*RQ2*). Our evaluation has shown that validity bugs can usually be fixed with little effort, which makes the effort for the programmer manageable. A first evaluation indicates that the additional code required to fix the header validity bugs does not result in significantly higher hardware resource consumption (*RQ3*).

CHAPTER 8

Expressivity of Π4

One of our main goals in designing Π4 was to create a type system that is capable of verifying relevant network properties beyond header validity but that retains at the same time the modularity inherent to type systems. In this chapter, we evaluate to what extent Π4 can bridge the gap between SafeP4 and full-fledged verification tools in terms of expressiveness, and whether Π4 is actually able to verify P4 programs in a modular fashion. We thereby answer the following two research questions:

RQ1 Can Π4 be used to verify practically relevant properties beyond header validity?

RQ2 Can Π4 be used to verify P4 programs in a modular fashion?

We start this chapter with a brief overview of properties discussed by other P4 verification tools ranging from basic safety properties to advanced safety properties. We then show how the most common of these properties and other practically relevant properties can be expressed using our heap types and verified by Π4's type checker. Finally, based on a case study, we investigate how we can facilitate Π4's type system to modularly verify P4 programs.

8.1 Survey

We surveyed the publications on recent P4 verification tools with respect to the network properties discussed. Table 8.1 lists seven properties that were discussed in the context of at least two tools. Other properties discussed are mainly application-specific and are therefore not considered further. For each of the verification tools including Π4, a checkmark indicates whether the respective property can be verified or not.

Header validity As already motivated in Chapter 3, accessing an invalid header instance in P4 yields undefined values, which in turn can result in a variety of subtle bugs. This problem has already been recognized, and as shown in Table 8.1, most P4 verification tools provide the ability to detect invalid headers accesses. As it was already the case with SafeP4, header validity is also a central safety property enforced by Π4's type system. The type checker rejects any program that attempts to access invalid headers.

	Header validity	Protocol conformance	Out-of-bounds accesses	Arithmetic overflow	Determined forwarding	Parser-Deparser compat.	Read-only fields
ASSERT-P4 [Fre+18]	✓	✓					
p4v [Liu+18]	✓	✓	✓	✓	✓	✓	✓
Vera [Sto+18]	✓	✓	✓	✓	✓		✓
P4RL [Shu+19]		✓				✓	
bf4[Dum+20]	✓		✓		✓	✓	
Π4	✓	✓			✓	✓	✓

Table 8.1: Common safety properties examined by other P4 verification tools.

The main difference is that Π4 is able to ensure header validity even in the presence of data-dependent validity checks. A corresponding example where the validity of the IPv4 header is checked depending on the EtherType field of the Ethernet header, has already been discussed in Section 5.1.

Protocol conformance To ensure interoperability between network devices of different vendors, network protocols are usually standardized. For example, these standards specify which values certain fields can contain, which other protocols are encapsulated in the payload of the packet depending on certain header fields, or how packets must be processed in certain situations. Protocol conformance is an umbrella term for a variety of properties that ensure that protocols are implemented correctly. A frequently considered example is the time-to-live (TTL) of an IPv4 packet, which limits how often a packet can be forwarded before it must be discarded by the switch. We show in Section 8.2.1 how we can express and verify protocol conformance such as the mentioned example using Π4's types.

Out-of-bound accesses P4's header stacks represent fixed-size arrays of headers. Accordingly, errors typical for arrays can occur, such as out-of-bounds accesses, which a programmer usually wants to rule out. Currently, Π4 lacks the support for header stacks. However, since this is not a fundamental limitation and avoiding out-of-bounds access is a prime example of the use of dependent types, it should be possible to extend Π4 accordingly in the future.

Arithmetic overflow Arithmetic operations in P4 do not detect overflows or underflows and should therefore be eliminated. Large values not fitting into a fixed number of bits will be cut off, which might lead to unexpected behavior of the program. Currently, it is not possible to check arithmetic overflows using Π4's type system. Since our heap

types capture values down to the bit level, it seems possible to support this property in the future.

Determined forwarding Another basic safety property, is *determined forwarding*. 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. We show in Section 8.2.2 how we can encode this invariant with Π4's heap types.

Parser-Deparser compatibility A P4 program typically defines the parser, controls for ingress and egress pipelines, and the deparser. The main reason for this four-phase structure is that separate ingress and egress pipelines allow packet processing to occur both before and after packets are scheduled, typically using one or more queues. 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 carried from ingress to egress is serialized and deserialized correctly. Otherwise, headers may be unexpectedly removed from the packet. We discuss this invariant in more detail in Section 8.2.3 and show how this invariant can be checked using Π4.

Read-only fields In P4 there are certain metadata fields that are read-only (e.g. the `egress_port`), yet the compiler does not rule out writes but instead silently ignores them. To prevent unexpected behavior of the program, it is therefore desirable to prohibit all write access to read-only fields. We show in Section 8.3 how an instance of this invariant can be checked using Π4.

8.2 Checking Network Invariants

We now show how Π4's type system can be used to check real network protocol invariants and verify a variety of basic and advanced safety properties as discussed before. In most P4 programs, the packet-forwarding behavior of the device is specified using a predefined record of type `standard_metadata_t`. In particular, the field `egress_spec` is used to instruct the switch to forward the packet 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 integer that can be encoded into 9 bits, which is the width of the field), the switch can be instructed to drop the packet. For simplicity, we treat P4 metadata as an ordinary header instance.

There is a general pattern that can be used to encode invariants in types. Given a program that typechecks with some type $(x : \tau_{in}) \rightarrow \tau_{out}$, we can instead typecheck the program with type $(x : \tau_{in}) \rightarrow \{y : \tau_{out} \mid \varphi_{inv}\}$, which refines the output type with an expression describing an invariant (φ_{inv}). By doing so, we effectively filter out all heaps for which the invariant does not hold. If the type computed by the type checker is not a subtype of the annotated output type, there must be some heap allowed by the program, for which the invariant does not hold.

```
/* Unsafe */
if(ipv4.valid) {
  stdmeta.egress_spec := 0x1 ;
  ipv4.ttl := ipv4.ttl - 1
}
```

```
/* Safe */
if(ipv4.valid) {
  if(ipv4.ttl == 0) {
    stdmeta.egress_spec := 0x1ff
  } else {
    stdmeta.egress_spec := 0x1;
    ipv4.ttl := ipv4.ttl - 1
  }
}
```

$$
\begin{aligned}
&(x : \{y : \texttt{ipv4}_{\approx} \mid y.\texttt{stdmeta}.\mathit{valid}\}) \rightarrow \\
&\quad \{y : \texttt{ipv4}_{\approx} \mid y.\texttt{stdmeta}.\mathit{valid} \wedge \\
&\qquad\qquad (y.\texttt{ipv4.ttl} = 0 \Longrightarrow y.\texttt{stdmeta.egress\_spec} = \texttt{0x1ff})\}
\end{aligned}
$$

Figure 8.1: IPv4 TTL example. Top: property violated; middle: property holds; bottom: Π4 type encoding the TTL invariant.

8.2.1 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.

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, the TTL is decremented; when the TTL is zero before the packet has reached its destination, forwarding halts to eliminate the risk of infinite loops.[1]

In the code snippet in the top of Figure 8.1, the intended behavior is violated, because the packet is always forwarded on the same port while the TTL is decremented. We can detect this violation by checking the program with the type shown in the bottom of Figure 8.1, which reads as follows: starting in a heap where at least IPv4 and the standard metadata is valid, after executing the ingress code, still at least IPv4 and the standard metadata is valid and if the IPv4 TTL is zero, the value of `egress_spec` indicates that the packet will be dropped. The middle of Figure 8.1 shows a program that successfully typechecks with the given 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

[1] Strictly speaking, IPv4 requires a special ICMP message to be returned to the sender to indicate the error, but here we will simply drop the packet.

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

```
/* Safe */
extract(ethernet);
if(ethernet.etherType == 0x0800) {
  extract(ipv4);
  if(ipv4.ihl != 0x5) {
    extract(ipv4_opt)
  }
}
```

$$
\begin{aligned}
&(x : \{y : \epsilon \mid |y.pkt_{in}| > 592\}) \rightarrow \\
&\quad \{y : \top \mid ((y.ipv4.valid \wedge y.ipv4.ihl \neq 5) \implies y.ipv4\_opt.valid) \wedge \\
&\qquad ((y.ipv4.valid \wedge y.ipv4.ihl = 5) \implies \neg y.ipv4\_opt.valid)\}
\end{aligned}
$$

Figure 8.2: IPv4 Options example. Top: property violated; middle: property holds; bottom: Π4 type encoding the IPv4-Option specification.

available. The minimum IHL is 5 ($5 * 32 = 160$) and the maximum value 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. Figure 8.2 provides one example where this property is violated (top) and one where it holds (middle). To rule out violations, we can check the respective programs with the type at the bottom of Figure 8.2. This type 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 if IPv4 is valid, then either IHL is 5 and IPv4 options are not valid, or IHL > 5 and IPv4 options are valid.

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 `ethernet.etherType`) for this purpose: a value of 0x0800 indicates that the next header is an IPv4 header, while, for example, a value of 0x86DD indicates that the next header is an IPv6 header. The code snippet in the top of Figure 8.3 violates the dependency between the IPv4 header and the EtherType field of the Ethernet header. Our type checker detects this violation by checking that executing the parser in an empty heap, with enough bits to extract both Ethernet and IPv4, produces a heap with either an invalid IPv4 header or a valid IPv4 header and an EtherType value of 0x0800, which is captured by the type shown in the bottom of Figure 8.3. An example of a program

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

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

$$(x : \{y : \epsilon \mid |y.pkt_{in}| > 272\}) \rightarrow \{y : \top \mid y.\texttt{ipv4}.valid \implies y.\texttt{ethernet.etherType} == 0x0800\}$$

Figure 8.3: Header dependency example. Top: property violated; middle: property holds; bottom: Π4 type encoding IPv4's dependency on Ethernet.

```
/* Unsafe */
if(ipv4.valid) {
  if(ipv4.dstAddr != 0x0a0a0a0a) {
    stdmeta.egress_spec := 0x1
  }
}
```

```
/* Safe */
if(ipv4.valid) {
  if(ipv4.dstAddr != 0x0a0a0a0a) {
    stdmeta.egress_spec := 0x1
  } else {
    stdmeta.egress_spec := 0x1ff
  }
}
```

$$(x : \{y : \texttt{ipv4}_{\approx} \mid y.\texttt{stdmeta}.valid\}) \rightarrow \{y : \top \mid y.\texttt{stdmeta.egress\_spec} \neq 0x0\}$$

Figure 8.4: Determined forwarding example. Top: property violated; middle: property holds; bottom: Π4 type encoding the determined forwarding specification.

that successfully typechecks with the given type is shown in the middle of Figure 8.3.

8.2.2 Determined Forwarding

Our type checker is able to detect violations in programs that do not make an explicit forwarding decision. For example, for the type shown at the bottom of Figure 8.4, the program shown in the top of Figure 8.4 violates this property, while the property holds for the program shown in the middle of the same figure. The type states that starting

in a heap where it is guaranteed that at least IPv4 and the standard metadata are valid, after executing the program, we end in a state, where for all program paths the field `stdmeta.egress_spec` does not contain the initial value anymore, which indicates that a forwarding decision was made.

8.2.3 Parser-Deparser Compatibility

We now show how Π4 can guarantee that deparsed packets can be correctly re-parsed without losing or corrupting information contained in packet headers. For example, assuming that the parser shown in Figure 8.5 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. Further, assuming that the programmer intends the ingress control shown in the middle right of Figure 8.5, 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 3) and the EtherType of the Ethernet header is updated accordingly. If an IPv4 header is present, the EtherType field of the VLAN header must also be updated (line 6) to obtain a protocol-conformant packet.

On the other hand, if the programmer forgot the statement on Line 4, i.e., they didn't update the `ethernet.etherType` field, serializing and deserializing the parsed headers will produce a corrupted packet. This unsafe example is shown on the middle left of Figure 8.5. 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. However, since the programmer forgot to update the EtherType, bits 96 to 112 still contain the value 0x0800. Hence, if the parser is run with this bit stream 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. As a result, 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. The code of the full pipeline is shown in the bottommost code snippet in Figure 8.5. The ascribed input type starting in Line 3 captures the assumptions about the state after executing the parser followed by the ingress control, i.e., both Ethernet and VLAN are guaranteed to be valid and the validity of IPv4 is indicated by the EtherType field of the VLAN header. The ascribed output type (Line 7) specifies the actual invariant, namely that executing the deparser followed by a reset statement and the parser produces a heap that is equivalent to the heap obtained after executing the parser followed by the ingress. We instruct our type checker to verify the property by checking the program with the type shown at the bottom of Figure 8.5, which ensures that there are enough bits to parse all possible headers.

8.2.4 Mutual Exclusion of Headers

Mutual exclusion is another property of interest that arises from the fact that the contents of specific header fields indicate which protocol header follows next and that each packet at runtime usually specifies only one such a protocol. In an implementation, we can take advantage of this property and use the same memory to store mutually exclusive headers. An example is the parser shown in the top of Figure 8.6 that conditionally parses either IPv4 or IPv6. Because only one of the paths is taken at runtime, it should

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

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

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

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

```
/* Ingress is either UnsafeIngress or SafeIngress */
Parser; Ingress;
(Deparser; reset; 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:⊤|x === y}
```

$$(x : \{y : \epsilon \mid |y.pkt_{out}| = 0 \wedge y.pkt_{in}.length > 304\}) \rightarrow \top$$

Figure 8.5: Roundtripping example. Common parser followed by the unsafe ingress code (left) and safe ingress code (right), followed by the deparser. The last code snippet shows the full pipeline, and type at the bottom is used to check the pipeline.

never happen that both instances are valid at the same time. 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 shown in Line 10 using the type in the annotation on line 11. This type ensures that starting from a heap in which Ethernet and optionally either IPv4 or IPv6 are valid, at the end of the ingress, it is still guaranteed that IPv4

```
1  (extract(ether);
2  if(ether.etherType == 0x86dd) {
3    extract(ipv6)
4  } else {
5    if(ether.etherType == 0x0800) {
6      extract(ipv4)
7    }
8  }) as (x:{y:ϵ|y.pkt_in.length > 432}) -> {y:ether~|!(y.ipv4.
     valid && y.ipv6.valid)};

10 Ingress /* Can be SafeIngress or UnsafeIngress*/
11   as (x:{y:ether~|!(y.ipv4.valid && y.ipv6.valid)}) ->
12         {y:ether~|!(y.ipv4.valid && y.ipv6.valid)};

14 if(ether.valid) {
15   remit(ether)
16 };
17 if(ipv4.valid) {
18   remit(ipv4)
19 };
20 if(ipv6.valid) {
21   remit(ipv6)
22 }
```

```
1 UnsafeIngress ≜
2   add(ipv6);
3   ether.etherType := 0x86dd
```

```
1 SafeIngress ≜
2   if(!ipv4.valid) {
3     add(ipv6);
4     ether.etherType = 0
          x86dd
5   }
```

$$(x : \{y : \epsilon \mid |y.pkt_{in}| > 432\}) \to \top$$

Figure 8.6: 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.

and IPv6 are not valid at the same time. The ingress code shown in the middle left of Figure 8.6 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 in the middle right is safe because it includes a conditional that explicitly checks the validity of IPv4 before adding IPv6.

Summary Π4's types are expressive enough to express a variety of properties beyond header validity, from protocol conformance to basic safety properties to complex proper-

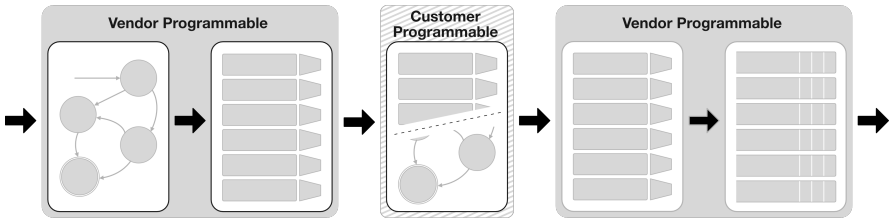

Figure 8.7: Hybrid switch architecture: fixed-function data plane in which the customer can execute custom code at pre-defined program points.

ties such as mutual exclusion of headers or the compatibility of the parser and deparser. This enables us to close the gap between previous type system-based approaches and existing verification tools for P4 (*RQ1*).

8.3 Designing for Modularity

One key advantage of P4 programmable devices over traditional network devices is that the functionality of the data plane can be tailored to the application needs. However, at the same time, this advantage presents new challenges for network administrators, for example, the entire functionality of such devices usually must be implemented from scratch, without the possibility to rely on proven implementations provided by vendors. Therefore, an emerging design pattern for data plane switches is partial programmability, such as Cisco's daPIPE [Bal19], which is designed for the Nexus 3400 switch [Cis21]. The idea is that a device vendor provides a partially-implemented pipeline together with a set of program points where the customer can inject custom code as shown in Figure 8.7. Since the P4 language was not designed for incremental programming, where functionality is added to an existing data plane program, several challenges arise. Most importantly, the base data plane program provided by the vendor should not be affected by the injected code. Because of trade secrets or too much effort required to understand the base program in detail, it is usually not an option that the vendor shares the implementation of the base program such that the programmer can understand what his program is allowed to do. A better approach is that vendors require that customer programs satisfy certain properties, but in current architectures, these properties are not automatically checked.

For example, 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 us assume that there are four classes of traffic, `Visitor`, `Student`, `Faculty`, and `Staff`, each with unique VLAN identifiers. We want to ensure that no matter how the customer-programmable part of the pipeline is instantiated, it cannot cause students and visitors to acquire the privileges of faculty or staff.

With Π4, we can design a modular system that checks invariants on customer programs to be integrated into vendor pipelines statically. Practically, we can ensure that the VLAN tag is not changed, by checking that the customer's code has a type like: $(x : \tau) \rightarrow \{y : \tau' \mid x.\texttt{vlan.vid} = y.\texttt{vlan.vid}\}$, where 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).

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

Figure 8.8: Instantiation of modular router design; the parser and deparser are provided by the vendor, the ingress is provided by the customer.

8.3.1 Specifying Invariants

In the following we will consider a very simple implementation of the example described above, show in Figure 8.8. We assume that the header instance `vlan` is the standard VLAN header (32 bits), including a 12-bit vlan tag field `vlan.vid`. We further assume that instance `stdmeta`, which is initially valid, provides access to the standard metadata (325 bits) used in the P4 switch model, including a 9-bit egress specification field `meta.egress_spec`.

The control flow simply extracts the VLAN instance, executes the modular ingress control, and then emits the VLAN header. The overall behavior of the program is captured by type $(x : \{y : \texttt{stdmeta} \mid |y.pkt_{in}| > 32\}) \to \Sigma y\colon \texttt{stdmeta}_{\approx}.\texttt{vlan}_{\approx}$. This type expresses that before executing the program `stdmeta` is valid and the input packet provides at least 32 bits, i.e., enough bits to extract the VLAN header. After the execution at least the metadata and VLAN headers are valid, but possibly also other headers added in the ingress. However, this type is too coarse-grained to guarantee that the ingress code will not change the parsed VLAN header.

Since both parser and deparser are provided by the vendor and stay unchanged, it is also not necessary to re-check that the whole pipeline is well-typed every time the customer implementation changes. Instead, we check once that the parser (Line 1 of Figure 8.8) is compatible with type $(x : \{y : \texttt{stdmeta} \mid |y.pkt_{in}| > 32\}) \to \Sigma y : \texttt{stdmeta.vlan}$ and that the deparser (Line 4 of Figure 8.8) is compatible with type $(x : \Sigma y\colon \texttt{stdmeta.vlan}) \to \Sigma y\colon \texttt{stdmeta}_{\approx}.\texttt{vlan}_{\approx}$. Of course, we could also provide a more specific output type for the deparser, for example, asserting that the content of the last 32 bits of the output packet is equal to the content of the VLAN instance.

When we swap in different implementations for the ingress control, we only need to check the ascribed type on Line 3 of Figure 8.8, without rechecking the surrounding code. This type defines the requirements the implementation of the ingress control must fulfill in order to be compatible with the rest of the pipeline, i.e, possibly new header instances can be added but the VLAN header must remain unchanged. 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.

8.3.2 Checking Customer Programs

We now consider a collection of customer programs 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 8.8.

```
Default ≜
  skip
```

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

```
Table ≜
  add(_vlan_table);
  if(_vlan_table.vid_key == vlan.vid) {
    if(_vlan_table.act == 0b0) {
      stdmeta.egress_spec := 0x1ff
    } else {
      stdmeta.egress_spec := 0x1
    }
  }
```

```
UnsafeActions ≜
  add(_vlan_table);
  if(_vlan_table.vid_key == vlan.vid) {
    if(_vlan_table.act == 0b0) {
      vlan.vid := Faculty
    } else {
      vlan.vid := Staff
    }
  } else {
    vlan.vid := Visitor
  }
```

Figure 8.9: A collection of safe and unsafe customer implementations for the `Ingress` module from Figure 8.8. Top Left: *Default*; Top Right: *Overwrite*; Bottom Left: *Table*; Bottom Right: *Unsafe Actions*

Default Consider the empty program, shown in the top of Figure 8.9, which would surely be the default behavior when the programmer has not written any code yet. To typecheck this no-op module, we check that command *skip* has the annotated type:

$$\cdot \vdash \texttt{skip} : (x : \Sigma y : \texttt{stdmeta.vlan}) \to \{y : \Sigma z : \texttt{stdmeta}_{\approx}.\texttt{vlan}_{\approx} \mid y.\texttt{vlan} = x.\texttt{vlan}\}$$

The command *skip* typechecks with this type by rules T-Skip and T-Sub, since

$$\begin{aligned}(x : \Sigma y : \texttt{stdmeta.vlan}) \vdash \{y : \Sigma z : \texttt{stdmeta.vlan} \mid y \equiv x\} \ <: \\ \{y : \Sigma z : \texttt{stdmeta}_{\approx}.\texttt{vlan}_{\approx} \mid y.\texttt{vlan} = x.\texttt{vlan}\}\end{aligned}$$

Overwrite Conversely, if the customer were to install an obviously incorrect program, such as the second one in Figure 8.9, which always overwrites the VLAN tag with the identifier reserved for faculty members, the type system complains that the following subtyping check fails:

```
1  control Ingress(...) {
2    action drp() {
3      stdmeta.egress_spec = 0x1FF;
4    }
5    action fwd() {
6      stdmeta.egress_spec = 1;
7    }
8
9    table vlan {
10     key = { vlan.vid : exact; }
11     actions = { drp; fwd; }
12   }
13
14   apply {
15     vlan.apply();
16   }
17 }
```

Figure 8.10: P4 table encoded by the program *Table* from Figure 8.9.

$$(x : \Sigma y\colon \texttt{stdmeta.vlan}) \vdash \{y : \Sigma z\colon \texttt{meta}_{\approx}.\texttt{vlan}_{\approx} \mid y.\texttt{vlan.vid} = \texttt{Faculty}\} \;<:\; \{y : \Sigma z\colon \texttt{stdmeta}_{\approx}.\texttt{vlan}_{\approx} \mid y.\texttt{vlan.vid} = x.\texttt{vlan.vid}\}$$

For example, if the VLAN tag of the incoming packet (x.`vlan.vid`) is `Student`, the two types denote disjoint sets of heaps.

Table The third program (*Table*) in Figure 8.9 encodes the P4 table shown in Figure 8.10, which matches on the value of header field `vlan.vid` and selects one of two actions: (1) `drp`, which sets the outgoing port (`egress_spec`) to 0x1FF, and (2) `fwd`, which sets it to 0x1.

As discussed in Section 6.5, to encode this table, we create a new header `_vlan_table` with a 12-bit field `vid_key` and a 1-bit field `act`. The field `_vlan_table.vid_key` represents the match key of the table, while the field `_vlan_table.act` encodes the different actions. Since in this example the table only provides two actions, a single bit is sufficient to encode both alternatives. Actions `drp` and `fwd` are represented by `_vlan_table.act = 0` and `_vlan_table.act = 1` respectively. This program will typecheck since no branch of the code modifies the `vlan.vid` field, and `_vlan_table` is permitted to be valid.

Unsafe actions Finally, let us consider the last program in Figure 8.9, which encodes the P4 table shown in Figure 8.11. This table provides three actions `fac`, `stf` and `vst`, each modifying the VLAN tag. The P4 annotation `@defaultonly` indicates that action `vst` can only be used as default action and never in the table.

Whenever the table is applied, the VLAN tag is overwritten, however, without respecting the VLAN tag of the incoming packet. This clearly violates the requirement

```
control Ingress(...) {
  action fac() {
    vlan.vid = Faculty;
  }
  action stf() {
    vlan.vid = Staff;
  }
  action vst() {
    vlan.vid = Visitor;
  }

  table vlan {
    key = { vlan.vid : exact; }
    actions = {
      fac; stf;
      vst @defaultonly;
    };
    default_action = vst;
  }

  apply {
    vlan.apply();
  }
}
```

Figure 8.11: P4 table encoded by the program *UnsafeActions* from Figure 8.9.

that the VLAN tag must be unchanged after executing the ingress program, triggering a violation of the subset check just as in the *Overwrite* example.

Summary This case study has shown how Π4 can be used to verify data plane programs in a modular way (*RQ2*). Type annotations allow expressing requirements that other modules must fulfill. If the implementation of modules changes, it is sufficient to re-check these modules in isolation according to their external requirements without having to re-check the entire pipeline.

8.4 Chapter Summary

In this chapter, we showed that Π4 is indeed expressive enough to verify a wide range of real-world network properties. Thus, Π4 is able to bridge the gap between type system-based approaches such as SafeP4 and full-fledged verification tools. At the same time, our approach allows programs to be verified modularly, which will likely benefit efforts to modularize P4 code in the future. There are still properties that Π4 does not yet support. As we have shown, these include, properties like array bound checks and arithmetic overflows, however we do not think this is a fundamental limitation since eliminating array bound checks is one of the standard examples discussed in the

dependent typing literature [XP98] and arithmetic overflows can be handled similarly. On the other hand, the verification of language features such as hash functions, externs, or registers poses a bigger challenge and is left for future work.

CHAPTER 9

Performance Evaluation

After exemplifying in the previous chapter that Π4's types are suitable for expressing and checking a variety of practically relevant properties, in this chapter we evaluate the runtime performance of Π4 using several real P4 programs. We thereby address the following research questions.

RQ1 How long does it take to verify that a certain property holds?

RQ2 What impact do the optimizations described in Section 6.4 have?

RQ3 What impact does the MTU and as such the size of bit vectors used in the encoding have?

RQ4 What is the impact of modular verification?

For our evaluation we used a collection of open-source programs and programs written by ourselves, which are summarized in Table 9.1. We used our P4 frontend to automatically translate P4 programs into Π4's syntax and then used the type checker to check annotated types. We were therefore limited to programs that can be translated into the syntax of Π4, for example, we exclude programs using persistent state (e.g. registers or counters). In some cases we have adapted the programs so that they could be translated, or we only translated parts automatically and parts by hand, while trying to preserve their semantics. For example, the program *ngsdn* is a slightly adapted version of the next generation SDN platform[1] tutorial, where we manually adjusted the control flow and removed the segment routing over IPv6 feature because the parser could not be fully translated. Similarly, we adapted the parser for program *fabric*—provided by the Open Network Operating System (ONOS)[2]—since it contains an infinite loop. In practice, the program relies on packets containing a specific header value that terminates the loop. However, this makes it impossible for us to unroll the parser, which is necessary for our translation to Π4. In addition, we only included the filtering and forwarding features and removed all remaining. To provide a better intuition about the different programs, Table 9.1 provides for each program (1) how many lines of P4 code it consists

[1] `https://opennetworking.org/ng-sdn/`
[2] `https://opennetworking.org/onos/`

Program	LoC P4	Parser states	Tables	Total header length (bytes)	Π4 Commands
multicast	116	2	1	38	23
basic	120	3	1	34	33
ecn	134	3	1	38	39
qos	157	3	1	38	41
vlan_decap	105	3	1	38	45
roundtrip	93	3	0	38	48
load_balance	170	4	3	38	49
basic_tunnel	156	4	2	38	55
ngsdn	421	10	6	88	239
fabric	537	15	6	86	541

Table 9.1: P4 programs used for the performance evaluation ordered by the number of Π4 commands after translation.

of, (2) how many states the parser comprises, (3) how many tables the program uses, (4) what is the longest sequence of packet headers that is parsed (in bytes), and (5) how many Π4 commands the translated program consists of.

9.1 Checking Header Validity

To address our first research question, as suggested by Liu et al. [Liu+18], we check all programs for header validity, as this property requires reasoning about all control-flow paths. Since header validity is a fundamental safety guarantee of Π4's type system, it is sufficient to check that the programs produce any valid output heap. We can capture this with the following type.

$$(x : \{y : \texttt{meta} \cdot \texttt{standard\_metadata} \mid |y.pkt_{in}| > \mathsf{LEN} \wedge |y.pkt_{out} = 0|\}) \rightarrow \top$$

This type states that, given an initial heap where both user-defined metadata and the intrinsic metadata provided by the target are valid[3], and enough bits are available in the input packet to extract the longest sequence of headers, and additionally assuming that the output packet is empty, the output heap is in the set of all possible heaps. For example, for program *fabric*, we use $\mathsf{LEN} = (86 * 8) - 1$ to assert that the input packet must at least provide 86 bytes = 688 bits.

We conducted all experiments on a workstation equipped with an Intel Core i7-6700K CPU and 32 GiB of RAM. Since we quickly ran out of memory while checking program *ngsdn*, we repeated the experiment on a server equipped with an AMD EPYC 7542 CPU and 512 GiB of RAM. In all other experiments, we obtained very similar results on both machines, and surprisingly, our workstation was even able to check programs faster in several cases.

For our first experiment, we used an MTU of 1500 bits, which is only one-eighth of the standard MTU, but is sufficient for our programs because the length of the input

[3] As explained in Section 2.2.2, this is the default behavior in P4

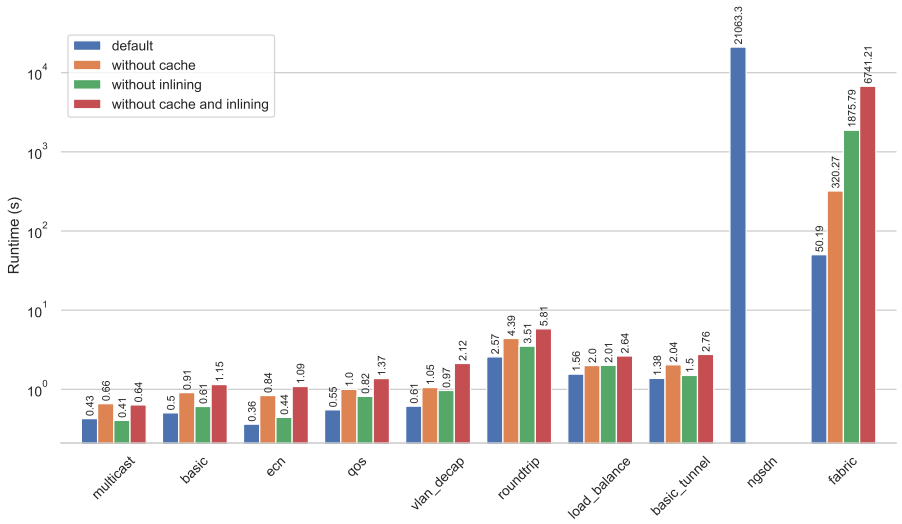

Figure 9.1: Time needed to check header validity for the programs listed in Table 9.1, with substitution inlining and validity caching enabled (*default*), without validity caching enabled, without substitution inlining enabled and with both optimizations disabled.

packet plus the length of the output packet—which is twice the total header length reported in Table 9.1—does not exceed this value. The results are given by the blue bars (*default*) in Figure 9.1. Since there are huge spans between the measured values, we use a logarithmic scale for the y-axis. However, since this distorts the actual size differences in the plot, we additionally give the absolute value above each bar. Most notably are the results for programs *ngsdn* and *fabric*. Intuitively, both programs take the longest to verify, as they are also the largest programs. Surprisingly, it takes almost 6 hours (5:51) to check *ngsdn*, while *fabric* with more than twice as many commands takes only about 50 seconds. The reason for this is that our type checker found a header validity bug in the implementation of *fabric*, and thus can terminate early.

Figure 9.2 shows a stripped-down excerpt from *fabric.p4* showing the offensive code. The ingress control first applies the filtering control and afterwards applies the forwarding control. The first control applies table `fwd_classifier`—shown in the top of Figure 9.2—which matches on the Ethernet type field and sets the forwarding type (`fwd_type`) accordingly. If the forwarding type is equal to the constant named `FWD_IPV6_UNICAST`, the forwarding control applies table `routing_v6`, which reads the IPv6 header. However, this program does not guarantee that the IPv6 header is actually valid in this case. After repairing the bug by guarding the invocation of table `routing_ipv6` with an additional validity check, we ran our checker again on the program, but—even on our server—eventually ran out of memory before the check was completed.

9.2 Effects of Optimizations on Runtime

In the next step, we evaluated whether the optimizations described in Section 6.4 actually have the desired effect, thus addressing *RQ2*. As a reminder, as a first optimization we reduce the size of the computed types by eliminating explicit type substitutions and the

```
action set_forwarding_type(fwd_type_t fwd_type) {
  meta.fwd_type = fwd_type;
}
table fwd_classifier {
  key = { hdr.eth_type.value: ternary; }
  actions = { set_forwarding_type; }
}
```

```
table routing_v6 {
  key = { hdr.ipv6.dst_addr: lpm; }
}
apply {
  if (meta.fwd_type == FWD_IPV6_UNICAST) {
    routing_v6.apply();
  }
}
```

Figure 9.2: Header validity bug in *fabric.p4*. Excerpt from control *Filtering* (top) and control *Forwarding* (bottom). The validity of `ipv6` in control *Forwarding* depends on correct control-plane entries in table `fwd_classifier`.

second optimization reduces the number of SMT solver calls by caching assumptions about header validity. The results can also be seen in Figure 9.1. Except for program *ngsdn*, we ran all programs respectively without the header validity cache but with substitution inlining, without substitution inlining but with the header validity cache, and without any optimization. Program *ngsdn* could not be checked at all without the optimizations.

As can be seen with program *multicast*, substitution inlining may introduce additional overhead for programs that can be checked in a short time. For smaller programs, the validity cache has a bigger impact on the runtime than substitution inlining, however, this reverses for lager programs as can be seen with program *fabric*.

9.3 Effects of the MTU on Runtime

Next, we evaluate the impact of MTU on the runtime of the type checker (*RQ3*). Again, we look at the time it takes our type checker to check header validity. We start with an MTU of 1500 bits and then increase the MTU in steps of 1500 bits, up to a value of 12000 bits = 1500 bytes. As can be seen in Figure 9.3, the runtime of the type checker increases with increasing MTU. For larger programs, the runtime increases significantly more with larger MTU values than for comparatively smaller programs. The program *roundtrip* is noticeable, which proportionally shows the largest increase in runtime. The reason is that the program uses the reset command, which prohibits the calculated types being completely inlined (cf. Section 6.4.1), which has a significant effect on the runtime of the SMT solver.

To better quantify the respective increase in runtime, Figure 9.4 additionally shows the percentage runtime increase compared to the next smaller MTU, i.e., how much does the runtime increase when we increase the MTU from 1500 bits to 3000 bits, or

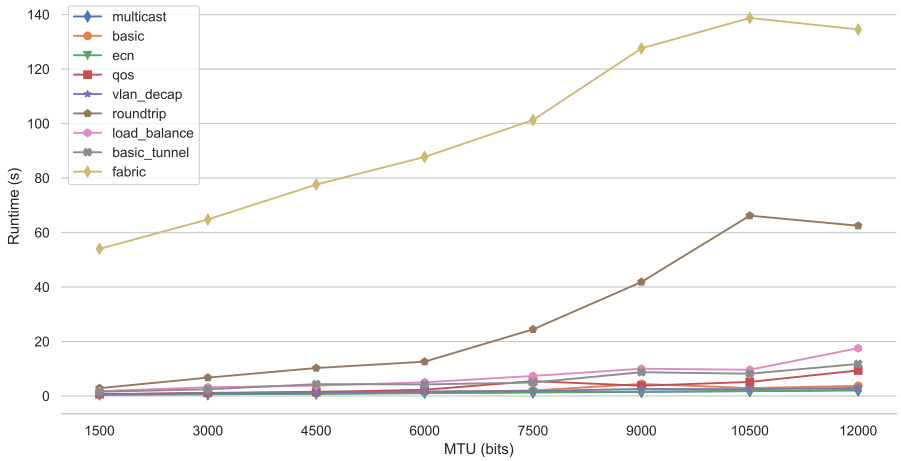

Figure 9.3: Overall time needed to check header validity with varying MTUs.

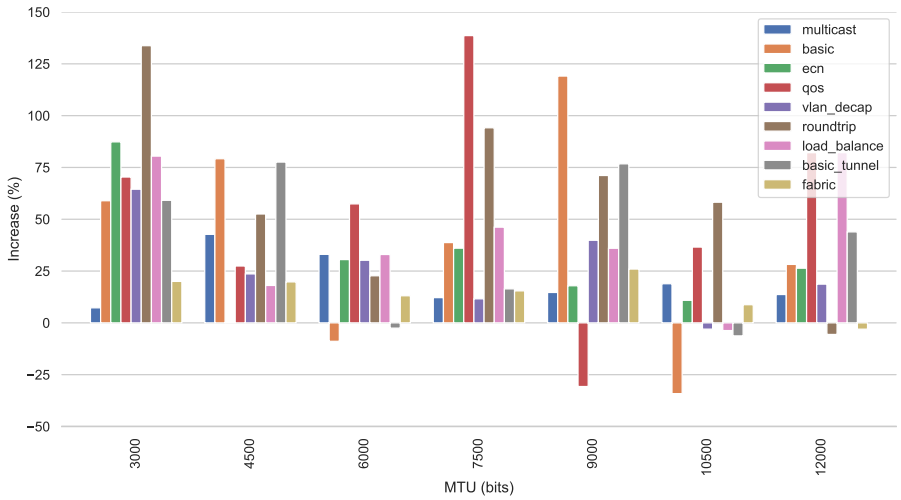

Figure 9.4: Percentage increase in runtime compared to the previous MTU value, starting from an MTU of 1500 bits. For example, checking program *roundtrip* with an MTU of 3000 bits increases the type checking time by roughly 130% compared to an MTU of 1500 bits.

from 3000 bits to 4500 bits, and so on. Overall, the runtime increases with a larger MTU value, but no direct conclusions can be drawn from the selected MTU to the increase in runtime. Not every increase in MTU results in the same percentage runtime increase. Also, the runtime does not necessarily increase to a greater extent with larger MTU values. In a few cases, a larger MTU value even leads to a reduced runtime compared to the next smaller MTU value.

Program	Parser	Ingress	Egress	Deparser	Complete
ngsdn	2.99	19.85	0.31	2.01	201.93
fabric	15.24	21.81	-	743.50	355.94

Table 9.2: Time in seconds needed to modularly verify header validity for programs *ngsdn* and *fabric*. *Parser*, *Ingress*, *Egress* and *Deparser* are the required times to check the respective pipeline stages in isolation. *Complete* indicates the time for the entire pipeline, where we ascribed each pipeline stage with a type that captures the requirements on the input heap and the guarantees for the resulting heap.

9.4 Modular Verification

A key aspect of Π4 is its support for modular verification. Therefore, we will now evaluate to which extent we can exploit modularity to make verification of large programs in particular achievable (*RQ4*). For this purpose we use program *ngsdn* and the fixed version of *fabric* without any validity bugs. Table 9.2 shows the results for checking each pipeline stage in isolation and for the whole pipeline. However, when checking the complete pipeline, we ascribe a type to every pipeline stage, which exactly captures the requirements it has on the input heap and the guarantees it offers with respect to the output heap.

For example, we use the following type to describe the parser of program *ngsdn*.

$$
\begin{aligned}
&(x : \{y : \texttt{standard\_metadata} \cdot \texttt{meta} \mid |y.pkt_{in}| > 703\}) \rightarrow \\
&\quad \{y : \top \mid y.\mathit{standard\_metadata.valid} \wedge \\
&\qquad\qquad y.\texttt{meta}.\mathit{valid} \wedge \\
&\qquad\qquad y.\texttt{ethernet}.\mathit{valid} \wedge \\
&\qquad\qquad \neg y.\mathit{cpu\_in.valid} \wedge \\
&\qquad\qquad y.\texttt{icmpv6}.\mathit{valid} \implies (y.\texttt{ipv6}.\mathit{valid} \wedge \\
&\qquad\qquad\quad (y.\texttt{icmpv6.type} = 0x87 \vee y.\texttt{icmpv6.type} = 0x88) \implies \\
&\qquad\qquad\quad y.\texttt{ndp}.\mathit{valid}) \wedge \\
&\qquad\qquad |y.pkt_{in}| \geq 0\}
\end{aligned}
$$

This type states that given an input packet that provides enough bits to extract all consecutive packet headers, the parser guarantees that afterwards the `ethernet` header will be parsed, but not the `cpu_in` header. Furthermore, if the `icmpv6` header is parsed, it is guaranteed that the `ipv6` header was also parsed and if the type field of header `icmpv6` contains the value 0x87 or 0x88 the `ndp` header will be additionally parsed. This type captures exactly the requirements of the ingress control to also successfully typecheck. We can proceed similarly for the egress and the deparser. For example the egress has the requirement that the header `cpu_in` must not be valid at the beginning, which is guaranteed by both the parser and the ingress. Overall, this approach allowed us to check the complete program *ngsdn* in about 3.5 minutes, which is a significant improvement over the 6 hours previously measured.

By proceeding analogously, we can also significantly reduce the time required to check program *fabric*. Note, the table does not provide a value for the egress because we have initially limited this program to a portion of the ingress control. Since the functionality of the ingress is described by two separate control blocks, we could also

check them in isolation. For example, using this approach, it took us only about 5 seconds to check the filtering part of the ingress control. It is surprising that it takes more than twice as long to check the deparser in isolation compared to the full pipeline. However, a detailed root cause analysis is left for future work.

9.5 Chapter Summary

In this chapter, we showed that Π4's SMT solver-based approach is suitable for checking real P4 applications. However, especially with regard to larger programs, it is essential to rely on Π4's capability to modularly check programs, but still our approach is currently not able to compete with tools like p4v [Liu+18] or Aquila [Tia+21]. It is crucial for an implementation of our SMT-based approach that the complexity of the SMT queries as well as the number of SMT solver invocations is reduced as much as possible, for example, by applying the optimizations we have proposed. Furthermore, a crucial factor with regard to the complexity of the SMT queries is the MTU used, which determines the maximum size of the encoded bit vectors. An interesting question that has remained open is whether it is possible to automatically compute the minimum required MTU for each program and annotated type while preserving the properties of our type system, thus avoiding unnecessary overhead.

Part IV

Epilogue

CHAPTER 10

Conclusion and Future Work

This dissertation confirms our initial hypothesis that type systems are well suited to equip data plane programming languages—in particular the P4 language—with safety guarantees, which makes it possible to verify a rich set of safety properties.

We started with the basic property of header validity where we were able to show that several of P4's language features are susceptible to bugs due to accessing invalid header instances. Based on these findings, we then designed SafeP4, a domain-specific language for programmable data planes whose static type system guarantees that all headers accessed are guaranteed to be valid. We were faced with the challenge that due to the interaction with the control plane, header validity becomes a dynamic property, which we addressed both by employing path-sensitive typing that incorporates information from forwarding table declarations and by assuming that the control plane satisfies three basic safety properties. Our evaluation showed that all the error categories we identified occur in real-world programs of varying sizes and that SafeP4's type system can be utilized to detect them without having to annotate the source code beforehand, which is a major difference to existing verification tools. In the second step we addressed the issue of SafeP4 being limited to checking header validity.

With Π4, we managed to close the gap between simple approaches such as SafeP4 and full-fledged verification tools with respect to the expressive power by resorting to the more powerful typing discipline of dependent types. Since type systems are a compositional way to establish program properties, our approach offers for the first time the possibility to verify rich correctness properties for data plane programs in a modular way. At the same time, we managed to automate subtype checks by encoding them into SMT queries in the theory of fixed-width bit vectors, relieving the programmer from writing manual proofs, which is common for dependently-typed systems.

With this dissertation we have laid the foundation for a wider adoption of type systems in the field of network programming—in particular for programmable data planes. This results in a variety of interesting directions in which our work can be expanded.

Integration into P4 The first direction in which our work can be extended is making our dependent type system practically usable. We have equipped our prototype implementation with a frontend based on Petr4 that allows programs written in the

P4 language to be parsed so that they can be annotated with types and type-checked. However, our frontend is not complete and requires two major enhancements.

The first is with respect to the support of more advanced language features. There are a few P4 features that our current prototype does not support, mostly because they pose challenges to SMT-based approaches to verification. The unpredictability of hash functions is difficult to verify. Besides over-approximating their behavior by representing them as uninterpreted functions, we can resort to a more fine-grained approach such as concolic verification [GKS05]. Registers are on-switch state that can be modified by the packet or the controller and persists between packets. Representing persistent state in our current semantics is tricky, since it involves distributed computing concerns and needs further investigation if one does not simply want to over-approximate their behavior, e.g., by making havoc of the values every time the register is read. More general, to enable typing of externs, the work on typing foreign function interfaces [FF08] might serve a starting point.

The second improvement concerns the integration of our type system into the P4 language. To guide the design of impactful systems for modular verification of data plans, it is necessary to provide a gradual transition from untyped to fully typed code. Gradual Typing [ST06; TF06] would allow P4 programmers to statically type parts of their code as needed. Approaches such as *migrational typing* [Cam+17] or the use of type inference [SV08] could support a gradual migration. How type inference, such as the one used by Liquid Haskell [VTV18], can be realized for Π4 is also future work.

In addition, the type-checking performance must be further optimized. Even though we have already implemented various optimizations, the overall time needed to check even simple programs is not sufficient for quick feedback during the development process. For this purpose, it can be examined whether special SMT tactics or an alternative encoding allow a more efficient solving of our subtyping constraints. An alternative could be an embedding of our type checker into a dependently-typed language that does not need to rely on an SMT solver at all.

Extend guarantees offered by types A second direction in which we can extend our work is to extend our type system to also cover the interaction between the data plane and the control plane. Even though the functionality of the data plane is not specified exclusively by the P4 program and depends to a large extent on the control plane, there are no mechanisms to ensure that no faulty rules—e.g., causing inconsistencies between the control plane and data plane [Shu+20]—are installed by the control plane.

It would therefore be interesting to investigate whether a specification of the data plane using the dependent types we have developed can be used to restrict the set of rules that can be installed by the control plane to conform to the network policy being implemented. For example, if the parser never extracts a particular packet header, this could result in the control plane program being forced to install a rule that ensures that appropriate packets are processed by the controller.

This would, however, require that both the data plane and the control plane are programmed together. Approaches from the field of *tierless programming* [Nel+14; RV18] might be suitable starting points. From the common description, a compiler could then generate platform-specific code, both for the data plane and the controller that implements the necessary guarantees.

Exploring Applications of chomp Another direction in which our work can be extended is with respect to our chomp operator. Here, it would be interesting to explore

what applications arise in other domains for our verified approach to parsing using derivatives. The domain of verified serializers and deserializers like EverParse [Ram+19] and Narcissus [Del+19] could be interesting. So far these approaches do not statically capture what remains after parsing parts of a certain input.

Bibliography

[Ale+98] D. S. Alexander et al. "The SwitchWare Active Network Architecture". In: *IEEE Network* 12.3 (May 1998), pp. 29–36. DOI: 10.1109/65.690959.

[All+03] J. R. Allen et al. "IBM PowerNP Network Processor: Hardware, Software, and Applications". In: *IBM Journal of Research and Development* 47.2-3 (2003), pp. 177–193. DOI: 10.1147/rd.472.0177.

[And+14] Carolyn Jane Anderson et al. "NetKAT: Semantic Foundations for Networks". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 113–126. DOI: 10.1145/2535838.2535862.

[Ara+16] Mina Tahmasbi Arashloo et al. "SNAP: Stateful Network-Wide Abstractions for Packet Processing". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016, pp. 29–43. DOI: 10.1145/2934872.2934892.

[Bai+18] Jiasong Bai et al. "Filtering Spoofed IP Traffic Using Switching ASICs". In: *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. ACM. 2018, pp. 51–53.

[Bal19] M. Baldi. "daPIPE - A Data Plane Incremental Programming Environment". In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2019, pp. 1–6. DOI: 10.1109/ANCS.2019.8901893.

[BCZ97] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. "An Architecture for Active Networking". In: *High Performance Networking VII: IFIP TC6 Seventh International Conference on High Performance Networks (HPN '97), 28th April – 2nd May 1997, White Plains, New York, USA*. Ed. by Ahmed Tantawy. Boston, MA: Springer US, 1997, pp. 265–279. DOI: 10.1007/978-0-387-35279-4_17.

[Bia+94] Edoardo Biagioni et al. "Signatures for a Network Protocol Stack: A Systems Application of Standard Ml". In: *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*. LFP '94. Orlando, Florida, USA: Association for Computing Machinery, 1994, pp. 55–64. DOI: 10.1145/182409.182431.

[BMG99] Andrew Begel, Steven McCanne, and Susan L. Graham. "BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture". In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '99. Cambridge, Massachusetts, USA: ACM, 1999, pp. 123–134. DOI: 10.1145/316188.316214.

[Bos+13] Pat Bosshart et al. "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN". In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 99–110.

[Bos+14] Pat Bosshart et al. "P4: Programming Protocol-independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 87–95. DOI: 10.1145/2656877.2656890.

[Bos18] Patrick Bosshart. "Programmable Forwarding Planes at Terabit/s Speeds". In: *2018 IEEE Hot Chips 30 Symposium (HCS). IEEE*. 2018.

[Bro19] Broadcom Inc. *Network Programming Language*. https://nplang.org/. Accessed: 11.03.2022. 2019.

[Brz64] Janusz A. Brzozowski. "Derivatives of Regular Expressions". In: *Journal of the ACM* 11.4 (Oct. 1964), pp. 481–494. DOI: 10.1145/321239.321249.

[Cal06] Ken Calvert. "Reflections on Network Architecture: An Active Networking Perspective". In: *SIGCOMM Comput. Commun. Rev.* 36.2 (2006), pp. 27–30. DOI: 10.1145/1129582.1129590.

[Cam+17] John Peter Campora et al. "Migrating Gradual Types". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), p. 15.

[Cas+07] Martin Casado et al. "Ethane: Taking Control of the Enterprise". In: *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '07. Kyoto, Japan: Association for Computing Machinery, 2007, pp. 1–12. DOI: 10.1145/1282380.1282382.

[Cis21] Cisco. *Cisco Nexus 3000 Series Switches*. https://www.cisco.com/c/en/us/products/switches/nexus-3000-series-switches/index.html. Jan. 2021.

[Con+07] Jeremy Condit et al. "Dependent Types for Low-level Programming". In: *Proceedings of the 16th European Symposium on Programming*. ESOP'07. Braga, Portugal: Springer-Verlag, 2007, pp. 520–535.

[Cor14] Jonathan Corbet. *BPF: The Universal In-kernel Virtual Machine*. Available at https://lwn.net/Articles/599755/, May 2014.

[Del+19] Benjamin Delaware et al. "Narcissus: Correct-by-construction Derivation of Decoders and Encoders from Binary Formats". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341686.

[DH17] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. Tech. rep. 8200. RFC Editor, July 2017. URL: https://www.rfc-editor.org/rfc/rfc8200.txt.

[DH98] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. RFC Editor, Dec. 1998. URL: https://www.rfc-editor.org/rfc/rfc2460.txt.

[Doe+21] Ryan Doenges et al. "Petr4: Formal Foundations for P4 Data Planes". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434322.

[Dum+20] Dragos Dumitrescu et al. "Bf4: Towards Bug-free P4 Programs". In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 571–585.

[Eic+19] Matthias Eichholz et al. "How to Avoid Making a Billion-Dollar Mistake: Type-Safe Data Plane Programming with SafeP4". In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 12:1–12:28. DOI: 10.4230/LIPIcs.ECOOP.2019.12.

[Eic+22] Matthias Eichholz et al. "Dependently-Typed Data Plane Programming". In: *Proceedings of the ACM on Programming Languages* 6.POPL (2022). DOI: 10.1145/3498701.

[ESM04] Robert Ennals, Richard Sharp, and Alan Mycroft. "Linear Types for Packet Processing". In: *Programming Languages and Systems*. Ed. by David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 204–218.

[FF08] Michael Furr and Jeffrey S. Foster. "Checking Type Safety of Foreign Function Calls". In: *ACM Trans. Program. Lang. Syst.* 30.4 (Aug. 2008). DOI: 10.1145/1377492.1377493.

[FG05] Kathleen Fisher and Robert Gruber. "Pads: A Domain-specific Language for Processing Ad Hoc Data". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 295–304. DOI: 10.1145/1065010.1065046.

[Fre+18] Lucas Freire et al. "Uncovering Bugs in P4 Programs with Assertion-based Verification". In: *Proceedings of the Symposium on SDN Research*. SOSR '18. Los Angeles, CA, USA: Association for Computing Machinery, 2018. DOI: 10.1145/3185467.3185499.

[FRZ13] Nick Feamster, Jennifer Rexford, and Ellen Zegura. "The Road to SDN: An Intellectual History of Programmable Networks". In: *Queue* 11.12 (2013), pp. 20–40. DOI: 10.1145/2559899.2560327.

[Gao+20] Jiaqi Gao et al. "Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs". In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 435–450. DOI: 10.1145/3387514.3405879.

[GGW15] Marco Gaboardi, Michael Greenberg, and David Walker. *Type Systems for SDN Controllers*. https://www.cs.princeton.edu/~dpw/papers/typed-controllers-plvnet-2015.pdf. 2015.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *Conference on Programming Language Design and Implementation (PLDI)*. 2005, pp. 213–223.

[Gud+08] Natasha Gude et al. "NOX: Towards an Operating System for Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.3 (July 2008), pp. 105–110. DOI: 10.1145/1384609.1384625.

[HDM14] Wei Huang, Yao Dong, and Ana Milanova. "Type-Based Taint Analysis for Java Web Applications". In: *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 140–154. DOI: 10.1007/978-3-642-54804-8_10.

[Hic+98] Michael Hicks et al. "PLAN: A Packet Language for Active Networks". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. Baltimore, Maryland, USA: Association for Computing Machinery, 1998, pp. 86–93. DOI: 10.1145/289423.289431.

[Hin+09] Timothy L. Hinrichs et al. "Practical Declarative Network Management". In: *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*. WREN '09. Barcelona, Spain: Association for Computing Machinery, 2009, pp. 1–10. DOI: 10.1145/1592681.1592683.

[Hoa09] Tony Hoare. *Null References: The Billion Dollar Mistake*. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/. Aug. 2009.

[Høi+18] Toke Høiland-Jørgensen et al. "The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel". In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. Heraklion, Greece: Association for Computing Machinery, 2018, pp. 54–66. DOI: 10.1145/3281411.3281443.

[HOM06] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. "Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks". In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '06/FSE-14. Portland, Oregon, USA: ACM, 2006, pp. 175–185. DOI: 10.1145/1181775.1181797.

[Iba+19] Stephen Ibanez et al. "The P4->NetFPGA Workflow for Line-Rate Packet Processing". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 1–9. DOI: 10.1145/3289602.3293924.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: A Minimal Core Calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3 (May 2001), pp. 396–450. DOI: 10.1145/503502.503505.

[Jin+17] Xin Jin et al. "NetCache: Balancing Key-Value Stores with Fast In-Network Caching". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 121–136. DOI: 10.1145/3132747.3132764.

[Jin+18] Xin Jin et al. "NetChain: Scale-Free Sub-RTT Coordination". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 35–49.

[Jin18] Xin Jin. *BMV2-based implementation of NetCache*. `https://github.com/netx-repo/netcache-p4`. Mar. 2018.

[JMW10] Trevor Jim, Yitzhak Mandelbaum, and David Walker. "Semantics and Algorithms for Data-dependent Grammars". In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain: Association for Computing Machinery, 2010, pp. 417–430. DOI: `10.1145/1706299.1706347`.

[KG16] Rahul Kumar and B. B. Gupta. "Stepping Stone Detection Techniques: Classification and State-of-the-art". In: *Proceedings of the international conference on recent cognizance in wireless communication & image processing*. Springer. 2016, pp. 523–533.

[Kim+15] Hyojoon Kim et al. "Kinetic: Verifiable Dynamic Network Control". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 59–72.

[Kle+18] George T. Klees et al. "Evaluating Fuzz Testing". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. Oct. 2018.

[Kod15] Chaitanya Kodeboyina. *An open-source P4 switch with SAI support*. `https://p4.org/p4/an-open-source-p4-switch-with-sai-support.html`. June 2015.

[KR18] Ali Kheradmand and Grigore Roșu. *P4K: A Formal Semantics of P4 and Applications*. Tech. rep. https://arxiv.org/abs/1804.01468. University of Illinois at Urbana-Champaign, Apr. 2018.

[Lak+04] T. V. Lakshman et al. "The SoftRouter Architecture". In: *ACM HOTNETS*. ACM, 2004. URL: `https://www.microsoft.com/en-us/research/publication/the-softrouter-architecture/`.

[Li+19] Guanyu Li et al. "NETHCF: Enabling Line-rate and Adaptive Spoofed IP Traffic Filtering". In: *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. 2019, pp. 1–12. DOI: `10.1109/ICNP.2019.8888057`.

[Liu+18] Jed Liu et al. "P4V: Practical Verification for Programmable Data Planes". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: ACM, 2018, pp. 490–503. DOI: `10.1145/3230543.3230582`.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[MC00] Peter J. McCann and Satish Chandra. "Packet Types: Abstract Specification of Network Protocol Messages". In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '00. Stockholm, Sweden: Association for Computing Machinery, 2000, pp. 321–333.

[McC+16] Jedidiah McClurg et al. "Event-Driven Network Programming". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 369–385. DOI: 10.1145/2908080.2908097.

[McK+08] Nick McKeown et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. DOI: 10.1145/1355734.1355746.

[McK+16] Nick McKeown et al. *Automatically Verifying Reachability and Well-formedness in P4 Networks*. Tech. rep. Sept. 2016. URL: https://www.microsoft.com/en-us/research/publication/automatically-verifying-reachability-well-formedness-p4-networks/.

[Mil78] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *Journal of Computer and System Sciences* 17.3 (Dec. 1978), pp. 348–375.

[Mon+13] Christopher Monsanto et al. "Composing Software Defined Networks". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 1–13.

[Mut+10] Chitra Muthukrishnan et al. "Using Strongly Typed Networking to Architect for Tussle". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Monterey, California: Association for Computing Machinery, 2010. DOI: 10.1145/1868447.1868456.

[Nan+08] Aleksandar Nanevski et al. "Ynot: Dependent Types for Imperative Programs". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 229–240. DOI: 10.1145/1411204.1411237.

[Nan+14] Aleksandar Nanevski et al. "Communicating State Transition Systems for Fine-Grained Concurrent Resources". In: *Programming Languages and Systems*. Ed. by Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 290–310.

[Nel+14] Tim Nelson et al. "Tierless Programming and Reasoning for Software-Defined Networks". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle: USENIX Association, 2014, pp. 519–531.

[Net18] Barefoot Networks. *Behavioral Model*. Dec. 2018. URL: https://github.com/p4lang/behavioral-model.

[Nev+18] Miguel Neves et al. "Verification of P4 Programs in Feasible Time Using Assertions". In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. Heraklion, Greece: Association for Computing Machinery, 2018, pp. 73–85. DOI: 10.1145/3281411.3281421.

[NMB06] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. "Polymorphism and Separation in Hoare Type Theory". In: *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 62–73. DOI: 10.1145/1159803.1159812.

[Nöt+18] Andres Nötzli et al. "P4pktgen: Automated Test Case Generation for P4 Programs". In: *Proceedings of the Symposium on SDN Research*. SOSR '18. Los Angeles, CA, USA: Association for Computing Machinery, 2018. DOI: 10.1145/3185467.3185497.

[Nou+19] Mohammad A. Noureddine et al. "P4AIG: Circuit-Level Verification of P4 Programs". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks – Supplemental Volume (DSN-S)*. 2019, pp. 21–22. DOI: 10.1109/DSN-S.2019.00016.

[OCo+18] T. J. OConnor et al. "Pivotwall: SDN-based Information Flow Control". In: *Proceedings of the Symposium on SDN Research*. ACM. 2018, p. 3.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. "PVS: A prototype verification system". In: *Automated Deduction—CADE-11*. Ed. by Deepak Kapur. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 748–752.

[Owr+95] S. Owre et al. "Formal verification for fault-tolerant architectures: prolegomena to the design of PVS". In: *IEEE Transactions on Software Engineering* 21.2 (1995), pp. 107–125. DOI: 10.1109/32.345827.

[P416] The P4 Language Consortium. *$P4_{16}$ Language Specification, Version 1.2.2*. Tech. rep. Available at https://p4.org/specs/, 2021. URL: https://p4.org/p4-spec/docs/P4-16-v1.2.2.pdf.

[PSA16] The P4.org Architecture Working Group. *$P4_{16}$ Portable Switch Architecture (PSA)*. Tech. rep. 2018.

[Ram+19] Tahina Ramananandro et al. "EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1465–1482.

[RKJ08] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. "Liquid Types". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 159–169. DOI: 10.1145/1375581.1375602.

[RŞ10] Grigore Roşu and Traian Florin Şerbănuţă. "An Overview of the K Semantic Framework". In: *Journal of Logic and Algebraic Programming* 79.6 (2010), pp. 397–434. DOI: 10.1016/j.jlap.2010.03.012.

[RV18] Gabriel Radanne and Jérôme Vouillon. "Tierless Web Programming in the Large". In: *Companion Proceedings of the The Web Conference 2018*. WWW '18. Lyon, France, 2018, pp. 681–689. DOI: 10.1145/3184558.3185953.

[Shu+19] Apoorv Shukla et al. "Runtime Verification of P4 Switches with Reinforcement Learning". In: *Proceedings of the 2019 Workshop on Network Meets AI & ML*. NetAI'19. Beijing, China: Association for Computing Machinery, 2019, pp. 1–7. DOI: 10.1145/3341216.3342206.

[Shu+20] Apoorv Shukla et al. "P4Consist: Toward Consistent P4 SDNs". In: *IEEE Journal on Selected Areas in Communications* 38.7 (2020), pp. 1293–1307. DOI: 10.1109/JSAC.2020.2999653.

[Siv+16] Anirudh Sivaraman et al. "Packet Transactions: High-Level Programming for Line-Rate Switches". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016, pp. 15–28. DOI: 10.1145/2934872.2934900.

[Smi+96] Jonathan M. Smith et al. "SwitchWare: Accelerating Network Evolution (White Paper)". In: *University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-96-38* (1996).

[Son+20] Hardik Soni et al. "Composing Dataplane Programs with μP4". In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 329–343. DOI: 10.1145/3387514.3405872.

[Sri18] Manu Sridharan. *Engineering NullAway, Uber's Open Source Tool for Detecting NullPointerExceptions on Android*. Dec. 2018. URL: https://eng.uber.com/nullaway/.

[ST06] Jeremy G. Siek and Walid Taha. "Gradual Typing for Functional Languages". In: *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*. 2006, pp. 81–92.

[Sto+16] Radu Stoenescu et al. "SymNet: Scalable Symbolic Execution for Modern Networks". In: *ACM SIGCOMM*. Florianopolis, Brazil: ACM, 2016, pp. 314–327. DOI: 10.1145/2934872.2934881.

[Sto+18] Radu Stoenescu et al. "Debugging P4 Programs with Vera". In: *ACM SIGCOMM*. Budapest, Hungary: ACM, 2018, pp. 518–532. DOI: 10.1145/3230543.3230548.

[SV08] Jeremy G. Siek and Manish Vachharajani. "Gradual Typing with Unification-Based Inference". In: *Proceedings of the 2008 Symposium on Dynamic Languages*. DLS '08. Paphos, Cyprus: Association for Computing Machinery, 2008. DOI: 10.1145/1408681.1408688.

[Swa+16] Nikhil Swamy et al. "Dependent Types and Multi-Monadic Effects in F*". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 256–270. DOI: 10.1145/2837614.2837655.

[Ten+97] D.L. Tennenhouse et al. "A survey of active network research". In: *IEEE Communications Magazine* 35.1 (1997), pp. 80–86. DOI: 10.1109/35.568214.

[TF06] Sam Tobin-Hochstadt and Matthias Felleisen. "Interlanguage Migration: From Scripts to Programs". In: *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 964–974. DOI: 10.1145/1176617.1176755.

[TF10] Sam Tobin-Hochstadt and Matthias Felleisen. "Logical Types for Untyped Languages". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Baltimore, Maryland, USA: ACM, 2010, pp. 117–128. DOI: 10.1145/1863543.1863561.

[Tia+21] Bingchuan Tian et al. "Aquila: A Practically Usable Verification System for Production-Scale Programmable Data Planes". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 17–32. DOI: 10.1145/3452296.3472937.

[TW07] David L. Tennenhouse and David J. Wetherall. "Towards an Active Network Architecture". In: *SIGCOMM Comput. Commun. Rev.* 37.5 (Oct. 2007), pp. 81–94. DOI: 10.1145/1290168.1290180.

[Vaz+14] Niki Vazou et al. "Refinement Types for Haskell". In: *ICFP*. 2014, pp. 269–282.

[VH11] Andreas Voellmy and Paul Hudak. "Nettle: Taking the Sting Out of Programming Network Routers". In: *Practical Aspects of Declarative Languages*. Ed. by Ricardo Rocha and John Launchbury. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 235–249.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. "A Sound Type System for Secure Flow Analysis". In: *J. Comput. Secur.* 4.2-3 (Jan. 1996), pp. 167–187.

[VKF12] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. "Procera: A Language for High-Level Reactive Network Control". In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 43–48. DOI: 10.1145/2342441.2342451.

[VTV18] Niki Vazou, Éric Tanter, and David Van Horn. "Gradual Liquid Type Inference". In: *Proceedings of the ACM on Programming Languages (PACMPL)* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276502.

[Wan+17] Han Wang et al. "P4FPGA: A Rapid Prototyping Framework for P4". In: *Proceedings of the Symposium on SDN Research*. SOSR '17. Santa Clara, CA, USA: Association for Computing Machinery, 2017, pp. 122–135. DOI: 10.1145/3050220.3050234.

[WF94] A.K. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness". In: *Inf. Comput.* 115.1 (Nov. 1994), pp. 38–94. DOI: 10.1006/inco.1994.1093.

[WGT98] D.J. Wetherall, J.V. Guttag, and D.L. Tennenhouse. "ANTS: a toolkit for building and dynamically deploying network protocols". In: *1998 IEEE Open Architectures and Network Programming*. 1998, pp. 117–129. DOI: 10.1109/OPNARC.1998.662048.

[WHM13] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. "Efficiently solving quantified bit-vector formulas". In: *Formal Methods in System Design* 42.1 (2013), pp. 3–23. DOI: 10.1007/s10703-012-0156-2.

[XH01] Hongwei Xi and Robert Harper. "A Dependently Typed Assembly Language". In: *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*. ICFP '01. Florence, Italy: Association for Computing Machinery, 2001, pp. 169–180. DOI: 10.1145/507635.507657.

[Xi00] Hongwei Xi. "Imperative programming with dependent types". In: *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*. 2000, pp. 375–387. DOI: 10.1109/LICS.2000.855785.

[XP98] Hongwei Xi and Frank Pfenning. "Eliminating Array Bound Checking through Dependent Types". In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: Association for Computing Machinery, 1998, pp. 249–257. DOI: 10.1145/277650.277732.

[XP99] Hongwei Xi and Frank Pfenning. "Dependent Types in Practical Programming". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, pp. 214–227.

[Yan+04] L. Yang et al. *Forwarding and Control Element Separation (ForCES) Famework*. RFC 3746. RFC Editor, Apr. 2004. URL: https://www.rfc-editor.org/rfc/rfc3746.txt.

APPENDIX A

Proofs

A.1 SafeP4

A.1.1 Operations on Header Types

Lemma A.1. $[\![\Theta]\!]|_h = [\![\texttt{Restrict}\ \Theta\ h]\!]$.

Proof. By induction on Θ.

Case $\Theta = 0$**:**

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

Case $\Theta = 1$**:**

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

Case $\Theta = g$**:**

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

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

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

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

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

□

Lemma A.2. $[\![\Theta]\!]|_{\neg h} = [\![\texttt{NegRestrict}\ \Theta\ h]\!]$.

Proof. By induction on Θ.

Case $\Theta = 0$**:**

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

Case $\Theta = 1$**:**

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

Case $\Theta = g$**:**

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

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

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

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

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

□

Lemma A.3. $\forall hs \in S. h \in hs == \texttt{Includes}\ \Theta\ h$.

Proof. By induction on Θ.

Case $\Theta = 0$**:**

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

Case $\Theta = 1$**:**

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

Case $\Theta = g$**:**

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

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

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

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

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

□

Lemma A.4. $[\![\Theta]\!] \setminus h == [\![\texttt{Remove}\ \Theta\ h]\!]$.

Proof. By induction on Θ.

Case $\Theta = 0$**:**

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

Case $\Theta = 1$**:**

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

Case $\Theta = g$**:**

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

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

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

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

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

□

Lemma A.5. $[\![\Theta]\!] == \{\}$ *if and only if* $\texttt{Empty}\ \Theta$.

Proof. By induction on Θ.

Case $\Theta = 0$**:** We have $[\![0]\!] = \{\}$ and *Empty* $0 = \mathit{true}$.

Case $\Theta = 1$**:** We have $[\![1]\!] \neq \{\}$ and *Empty* $1 = \mathit{false}$.

Case $\Theta = h$**:** We have $[\![h]\!] \neq \{\}$ and *Empty* $h = \mathit{false}$.

Case $\Theta = \Theta_1 \cdot \Theta_2$**:** By definition we have $[\![\Theta_1 \cdot \Theta_2]\!] = [\![\Theta_1]\!] \bullet [\![\Theta_2]\!]$ which is equal to $\{S_1 \cup S_2 \mid S_1 \in [\![\Theta_1]\!] \wedge S_2 \in [\![\Theta_2]\!]\}$. It follows that $[\![\Theta_1 \cdot \Theta_2]\!] = \{\}$ iff $[\![\Theta_1]\!] = \{\}$ or $[\![\Theta_2]\!] = \{\}$. By induction hypothesis, we have $[\![\Theta_1]\!] = \{\}$ if and only if *Empty* $\Theta_1 = \mathit{true}$, and $[\![\Theta_2]\!] = \{\}$ if and only if *Empty* $\Theta_2 = \mathit{true}$. The result follows as *Empty* $(\Theta_1 \cdot \Theta_2) = \texttt{Empty}\ \Theta_1 \vee \texttt{Empty}\ \Theta_2$.

Case $\Theta = \Theta_1 + \Theta_2$**:** By definition we have $[\![\Theta_1 + \Theta_2]\!] = [\![\Theta_1]\!] \cup [\![\Theta_2]\!]$. It follows that $[\![\Theta_1 \cdot \Theta_2]\!] \neq \{\}$ iff $[\![\Theta_1]\!] \neq \{\}$ and $[\![\Theta_2[\!] \neq \{\}$. By induction hypothesis, we have $[\![\Theta_1]\!] \neq \{\}$ if and only if *Empty* $\Theta_1 = \mathit{true}$, and $[\![\Theta_2]\!] \neq \{\}$ if and only if *Empty* $\Theta_2 = \mathit{true}$. The result follows as *Empty* $(\Theta_1 + \Theta_2) = \texttt{Empty}\ \Theta_1 \wedge \texttt{Empty}\ \Theta_2$.

□

A.1.2 Safety

Lemma A.6 (Expression Substitution)**.** *If* $\Gamma, x : \tau; \Theta \vdash e : \tau'$ *and* $\cdot; \cdot \vdash \bar{v} : \bar{\tau}$ *then* $\Gamma; \Theta \vdash e[\bar{v}/\bar{x}] : \tau'$

Lemma A.7 (Entailment is Type Alternative)**.** *If* $H \vDash \Theta$ *then* $\mathit{dom}(H) \in [\![\Theta]\!]$.

Proof. By induction on Θ.

Case $\Theta = 0$**:** The case immediately holds as $H \vDash 0$ is a contradiction.

Case $\Theta = 1$**:** By inversion of *Entailment*, $H = \cdot$, and so $\mathit{dom}(H) = \{\} \in [\![1]\!] = \{\{\}\}$.

Case $\Theta = h$**:** By inversion of *Entailment*, $dom(H) = \{h\} \in [\![h]\!] = \{\{h\}\}$.

Case $\Theta = \Theta_1 \cdot \Theta_2$**:** By inversion of *Entailment*, $H = H_1 \cup H_2, H_1 \vDash \Theta_1, H_2 \vDash \Theta_2$. By induction hypothesis, $dom(H_1) \in [\![\Theta_1]\!]$ and $dom(H_2) \in [\![\Theta_2]\!]$. By set theory, $dom(H) = dom(H_1) \cup dom(H_2)$
By definition of $[\![.]\!]$ and $(\bullet)$, $[\![\Theta]\!] = [\![\Theta_1]\!] \bullet [\![\Theta_2]\!] = \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!]\}$ and therefore $dom(H_1) \cup dom(H_2) \in \{hs_1 \cup hs_2 | hs_1 \in [\![\Theta_1]\!] \wedge hs_2 \in [\![\Theta_2]\!]\}$, i.e., $dom(H) \in [\![\Theta]\!]$.

Case $\Theta = \Theta_1 + \Theta_2$**:** By inversion of *Entailment*, either $H \vDash \Theta_1$ or $H \vDash \Theta_2$.

Subcase $H \vDash \Theta_1$**:** By the induction hypothesis, $dom(H) \in [\![\Theta_1]\!]$ and by set theory $dom(H) \in [\![\Theta_1]\!] \cup [\![\Theta_2]\!]$

Subcase $H \vDash \Theta_2$**:** Symmetric to the previous subcase.

□

Lemma A.8 (Included Instances in Domain)**.** *If* $H \vDash \Theta$ *and* `Includes` Θ h*, then* $h \in dom(H)$.

Proof. By induction on Θ.

Case $\Theta = 0$**:** The case immediately holds as $H \vDash 0$ is a contradiction.

Case $\Theta = 1$**:**

By inversion of *Entailment*, $H = \cdot$. The case immediately holds, as `Includes` Θ h is a contradiction.

Case $\Theta = g$**:**

By inversion of *Entailment*, $dom(H) = \{g\}$. By assumption `Includes` Θ h, $h = g$ and thus $h \in dom(H)$.

Case $\Theta = \Theta_1 \cdot \Theta_2$**:** By inversion of *Entailment*, $H = H_1 \cup H_2, H_1 \vDash \Theta_1, H_2 \vDash \Theta_2$. By set theory $dom(H) = dom(H_1) \cup dom(H_2)$. By definition of *Inclusion* and by assumption `Includes` Θ h, `Includes` Θ_1 $h \vee$ `Includes` Θ_2 h.

Subcase `Includes` Θ_1 h**:** By induction hypothesis, $h \in dom(H_1)$ and by assumption $dom(H_1) \subseteq dom(H)$, we can conclude $h \in dom(H)$.

Subcase `Includes` Θ_2 h**:** Symmetric to the previous subcase.

Case $\Theta = \Theta_1 + \Theta_2$**:** By inversion of *Entailment*, either $H \vDash \Theta_1$ or $H \vDash \Theta_2$. By definition of *Inclusion* and by assumption `Includes` Θ h, `Includes` Θ_1 h and `Includes` Θ_2 h.

Subcase $H \vDash \Theta_1$**:** By induction hypothesis, we can conclude $h \in dom(H)$.

Subcase $H \vDash \Theta_2$**:** Symmetric to the previous subcase.

□

Progress

Theorem A.9 (Progress). *If* $\cdot \vdash c : \Theta \Rightarrow \Theta'$ *and* $H \vDash \Theta$*, then either* $c = skip$ *or* $\exists \langle I', O', H', c' \rangle.\ \langle I, O, H, c \rangle \rightarrow \langle I', O', H', c' \rangle$

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

***Case* T-Skip:** $c = skip$
Immediate.

***Case* T-Extr:** $c = extract(h)$
Let $(I', v) = deserialize_\eta(I)$ and $O' = O$ and $H' = H[h \mapsto v]$ and $c' = skip$. The result follows by E-Extract.

***Case* T-Emit:** $c = emit(h)$
If $h \notin dom(H)$, let $I' = I$ and $O' = O$ and $H' = H$, and $c' = skip$. The result follows by E-EmitInvalid. Otherwise, $h \in dom(H)$. Let $H(h) = v$ and $\bar{B} = serialize_\eta(v)$ and $I' = I$ and $O' = O.\bar{B}$ and $H' = H$ and $c' = skip$. The result follows by E-Emit.

***Case* T-Seq:** $c = c_1; c_2$ and $\cdot \vdash c_1 : \Theta \Rightarrow \Theta_1$ and $\cdot \vdash c_2 : \Theta_1 \Rightarrow \Theta_2$
By induction hypothesis, c_1 is either *skip* or there is some $\langle I', O', H', c_1' \rangle$, such that $\langle I, O, H, c_1 \rangle \rightarrow \langle I', O', H', c_1' \rangle$.
If $c_1 = skip$, let $I' = I$ and $O' = O$ and $H' = H$ and $c' = c_2$. The result follows by E-Seq. Otherwise, the result follows by E-Seq1.

***Case* T-If:** $c = if\ (e)\ then\ c_1\ else\ c_2$ and $\cdot; \Theta \vdash e : Bool$ and $\cdot \vdash c_1 : \Theta \Rightarrow \Theta_1$ and $\cdot \vdash c_2 : \Theta \Rightarrow \Theta_2$
By the progress theorem for expressions, we have that e is either *true*, *false*, or there is some e' such that $\langle H, e \rangle \rightarrow e'$.

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

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

Subcase $\langle H, e \rangle \rightarrow e'$**:** Let $I' = I$ and $O' = O$ and $H' = H$ and $c' = if\ (e')\ c_1\ c_2$. The result follows by E-If.

***Case* T-IfValid:** $c = valid(h)\ c_1\ else\ c_2$
If $h \in dom(H)$, let $I' = I$ and $O' = O$ and $H' = H$ and $c' = c_1$. The result follows by E-IfValidTrue Otherwise, $h \notin dom(H)$. Let $I' = I$ and $O' = O$ and $H' = H$ and $c' = c_2$. The result follows by E-IfValidFalse

***Case* T-Apply:** $c = t.apply()$
By Proposition 1, we have $\mathcal{CA}(t, H) = (a_i, \bar{v})$. Let $\mathcal{A}(a_i) = \lambda \bar{x} : \bar{\tau}.\ c_i$. Let $I' = I$ and $O' = O$ and $H' = H$ and $c' = c_i[\bar{v}/\bar{x}]$. The result follows by E-Apply.

***Case* T-Add:** $c = add(h)$
If $h \in dom(H)$, let $I' = I$ and $O' = O$ and $H' = H$ and $c' = skip$. The result follows by E-AddValid. Otherwise, $h \notin dom(H)$. Let $v = init_\eta$ and $I' = I$ and $O' = O$ and $H' = H[h \mapsto v]$ and $c' = skip$. The result follows by E-Add

***Case* T-Remove:** $c = remove(h)$
Let $I' = I$ and $O' = O$ and $H' = H \setminus h$ and $c' = skip$. The result follows by E-Remove.

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

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

Subcase $e = v$**:** By Lemma A.8: $h \in dom(H)$. Let $r = H(h)$ and $r' = \{r\ with\ f = v\}$. Also let $I' = I$ and $O' = O$ and $H' = H[h \mapsto r']$ and $c' = skip$. The result follows by E-Mod.

Subcase $\langle H, e\rangle \to e'$**:** Let $I' = I$ and $O' = O$ and $H' = H$ and $c' = h.f = e'$. The result follows by E-Mod1.

***Case* T-Zero:** $\texttt{Empty}\ \Theta_1$

By Lemma A.7, we have $dom(H) \in [\![\Theta_1]\!]$. By Lemma A.5, we have $[\![\Theta_1]\!] = \{\}$, which is a contradiction. □

Preservation

Lemma A.10 (Restriction Entailed)**.** *If $H \vDash \Theta$ and $h \in dom(H)$, then $H \vDash \texttt{Restrict}\ \Theta\ h$.*

Proof. By induction on Θ.

Case $\Theta = 0$**:** The case immediately holds as $H \vDash 0$ is a contradiction.

Case $\Theta = 1$**:** By inversion of *Entailment*, $H = \cdot$. The case immediately holds as $h \in dom(\cdot)$ is a contradiction.

Case $\Theta = g$**:** By inversion of *Entailment*, $dom(H) = \{g\}$, and so $h = g$. By definition $\texttt{Restrict}\ \Theta\ h = \texttt{Restrict}\ g\ g = g$. By Ent-Inst $H \vDash g$, i.e., $H \vDash \Theta$.

Case $\Theta = \Theta_1 \cdot \Theta_2$**:** By inversion of *Entailment* $H = H_1 \cup H_2, H_1 \vDash \Theta_1, H_2 \vDash \Theta_2$. By $h \in dom(H)$, either $h \in dom(H_1)$ or $h \in dom(H_2)$.

Subcase $h \in dom(H_1)$**:**

By the induction hypothesis, we have $H_1 \vDash \texttt{Restrict}\ \Theta_1\ h$. By Ent-Seq, we have $H_1 \cup H_2 \vDash \texttt{Restrict}\ \Theta_1\ h \cdot \Theta_2$. By Ent-ChoiceL, we have $H_1 \cup H_2 \vDash (\texttt{Restrict}\ \Theta_1\ h \cdot \Theta_2) + (\Theta_1 \cdot \texttt{Restrict}\ \Theta_2\ h)$ which concludes the case.

Subcase $h \in dom(H_2)$**:** Symmetric to the previous subcase.

Case $\Theta = \Theta_1 + \Theta_2$**:** By inversion of *Entailment*, either $H \vDash \Theta_1$ or $H \vDash \Theta_2$.

Subcase $H \vDash \Theta_1$**:** By the induction hypothesis, we have $H \vDash \texttt{Restrict}\ \Theta_1\ h$. By Ent-ChoiceL, $H \vDash \texttt{Restrict}\ \Theta_1\ h + \texttt{Restrict}\ \Theta_2\ h$.

Subcase $H \vDash \Theta_2$**:** Symmetric to the previous subcase.

□

Lemma A.11 (NegRestriction Entailed)**.** *If $H \vDash \Theta$ and $h \notin dom(H)$, then $H \vDash \texttt{NegRestrict}\ \Theta\ h$.*

Proof. By induction on Θ.

Case $\Theta = 0$**:** The case immediately holds as $H \vDash 0$ is a contradiction.

Case $\Theta = 1$**:** By inversion of *Entailment*, $H = \cdot$. By definition of *Negated Restriction*, $\texttt{NegRestrict}\ \Theta\ h = \texttt{NegRestrict}\ 1\ h = 1$. By Ent-Empty $\cdot \vDash 1$, i.e., $H \vDash \texttt{NegRestrict}\ \Theta\ h$.

Case $\Theta = g$**:** By inversion of *Entailment*, $dom(H) = \{g\}$. By assumption $h \neq g$. By definition of *Restriction* $\texttt{NegRestrict}\ \Theta\ h = \texttt{NegRestrict}\ g\ h = g$. By Ent-Inst $H \vDash g$, i.e., $H \vDash \texttt{NegRestrict}\ \Theta\ h$.

Case $\Theta = \Theta_1 \cdot \Theta_2$**:** By inversion of *Entailment*, $H = H_1 \cup H_2, H_1 \vDash \Theta_1, H_2 \vDash \Theta_2$. By $h \notin dom(H)$, $h \notin dom(H_1)$ and $h \notin dom(H_2)$. By the induction hypothesis, $H_1 \vDash \texttt{NegRestrict}\ \Theta_1\ h$ and $H_2 \vDash \texttt{NegRestrict}\ \Theta_2\ h$. By Ent-Seq, $H_1 \cup H_2 \vDash \texttt{NegRestrict}\ \Theta_1\ h \cdot \texttt{NegRestrict}\ \Theta_2\ h$ which finishes the case.

Case $\Theta = \Theta_1 + \Theta_2$**:** By inversion of *Entailment*, either $H \vDash \Theta_1$ or $H \vDash \Theta_2$.

Subcase $H \vDash \Theta_1$**:**

By the induction hypothesis, we have $H \vDash \texttt{NegRestrict}\ \Theta_1\ h$.

By Ent-ChoiceL, $H \vDash \texttt{NegRestrict}\ \Theta_1\ h + \texttt{NegRestrict}\ \Theta_2\ h$.

Subcase $H \vDash \Theta_2$**:** Symmetric to the previous subcase.

□

Lemma A.12 (Entailment Congruence)**.** *If* $H \vDash \Theta$ *and* $dom(H) = dom(H')$ *then* $H' \vDash \Theta$.

Proof. By induction on Θ.

Case $\Theta = 0$**:** The case immediately holds as $H \vDash 0$ is a contradiction.

Case $\Theta = 1$**:** By inversion of *Entailment* $H = \cdot$. By assumption $dom(H) = dom(H')$, $H' = \cdot$ and by Ent-Empty, $H' \vDash \Theta$.

Case $\Theta = g$**:** By inversion of *Entailment*, $dom(H) = \{g\}$. By assumption $dom(H) = dom(H')$ and by Ent-Inst, $H' \vDash \Theta$.

Case $\Theta = \Theta_1 \cdot \Theta_2$**:** By inversion of *Entailment*, $H = H_1 \cup H_2, H_1 \vDash \Theta_1, H_2 \vDash \Theta_2$. By set theory, $dom(H) = dom(H_1) \cup dom(H_2)$. By induction hypothesis if $dom(H_1') = dom(H_1)$ and $dom(H_2') = dom(H_2)$, then $H_1' \vDash \Theta_1$, and $H_2' \vDash \Theta_2$. By Ent-Seq, $H' = H_1' \cup H_2' \vDash \Theta_1 \cdot \Theta_2$.

Case $\Theta = \Theta_1 + \Theta_2$**:** By inversion of *Entailment*, either $H \vDash \Theta_1$ or $H \vDash \Theta_2$.

Subcase $H \vDash \Theta_1$**:** By induction hypothesis, we have $H' \vDash \Theta_1$. By Ent-ChoiceL, $H' \vDash \Theta_1 + \Theta_2$.

Subcase $H \vDash \Theta_2$**:** Symmetric to the previous subcase.

□

Lemma A.13 (Substitution)**.** *If* $\Gamma, x : \tau \vdash c : \Theta \Rrightarrow \Theta'$ *and* $\cdot; \cdot \vdash v : \tau$ *then* $\Gamma \vdash c[v/x] : \Theta \Rrightarrow \Theta'$.

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

Lemma A.14 (Entails Subsumption)**.** *If* $H \vDash \Theta$ *then* $H[h \mapsto v] \vDash \Theta \cdot h$

Proof. We analyze two cases.

Case $h \in dom(H)$**:** By the assumption of the case, we have $dom(H) = dom(H[h \mapsto v])$. Let $H_1 = H[h \mapsto v]$ and $H_2 = \{h \mapsto v\}$. Observe that $H[h \mapsto v] = H_1 \cup H_2$. By Lemma A.12, we have that $H_1 \vDash \Theta$. By Ent-Inst we have $H_2 \vDash h$. By Ent-Seq we have $H[h \mapsto v] \vDash \Theta \cdot h$.

Case $h \notin dom(H)$**:** Let $H_1 = H$ and $H_2 = \{h \mapsto v\}$. Observe that $H[h \mapsto v] = H_1 \cup H_2$. By assumption we have $H_1 \vDash \Theta$. By Ent-Inst we have $H_2 \vDash h$. By Ent-Seq we have $H[h \mapsto v] \vDash \Theta \cdot h$.

□

Lemma A.15 (Entails Removal)**.** *If* $H \vDash \Theta$ *then* $H \setminus h \vDash \texttt{Remove}\ \Theta\ h$.

Proof. By induction on Θ.

Case $\Theta = 0$**:** The case immediately holds, as $H \vDash 0$ is a contradiction.

Case $\Theta = 1$**:** By inversion of *Entailment*, $H = \cdot$. By set theory, $\cdot \setminus h = \cdot$ and $\texttt{Remove}\ 1\ h = 1$. By Ent-Empty, $\cdot \vDash 1$.

Case $\Theta = g$**:** By inversion of *Entailment*, $dom(H) = \{g\}$.

Subcase $g = h$**:** By set theory $H \setminus h = \cdot$. By definition of *Remove*, $\texttt{Remove}\ \Theta\ h = 1$. By Ent-Empty, $\cdot \vDash 1$, which concludes the case.

Subcase $g \neq h$**:** By set theory $H \setminus h = H$. By definition of *Remove*, $\texttt{Remove}\ \Theta\ h = g$. By assumption, $H \vDash \Theta$, which concludes the case.

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

Case $\Theta = \Theta_1 + \Theta_2$**:** By inversion of *Entailment*, either $H \vDash \Theta_1$ or $H \vDash \Theta_2$.

Subcase $H \vDash \Theta_1$**:** By induction hypothesis, $H \setminus h \vDash \texttt{Remove}\ \Theta_1\ h$. By Ent-ChoiceL, applied to $H \setminus h \vDash \texttt{Remove}\ \Theta_1\ h$, and $\texttt{Remove}\ \Theta_2\ h$, we can conclude $H \setminus h \vDash \texttt{Remove}\ \Theta_1\ h + \texttt{Remove}\ \Theta_2\ h$. By definition of *Removal*, $\texttt{Remove}\ \Theta_1\ h + \texttt{Remove}\ \Theta_2\ h = \texttt{Remove}\ (\Theta_1 + \Theta_2)\ h$.

Subcase $H \vDash \Theta_2$**:** Symmetric to previous subcase.

□

Lemma A.16 (Order Extend)**.** *If* $\Theta_1' < \Theta_1$ *then* $\Theta_1' \cdot h < \Theta_1 \cdot h$.

Proof. By assumption $\Theta_1' < \Theta_1$ and the definition of $<$ follows $[\![\Theta_1']\!] \subseteq [\![\Theta_1]\!]$ By $[\![.]\!]$ follows $[\![\Theta_1' \cdot h]\!] == [\![\Theta_1']\!] \bullet \{\{h\}\}$ and $[\![\Theta_1 \cdot h]\!] == [\![\Theta_1]\!] \bullet \{\{h\}\}$.

Let $S \in [\![\Theta_1']\!] \bullet \{\{h\}\}$. By definition of $\bullet$, $S = S' \cup \{h\}$, where $S' \in [\![\Theta_1']\!]$. By $[\![\Theta_1']\!] \subseteq [\![\Theta_1]\!]$, follows that $S' \in [\![\Theta_1]\!]$. By set theory, $S' \cup \{h\} \in [\![\Theta_1]\!] \bullet \{\{h\}\}$. Then $[\![\Theta_1']\!] \bullet \{\{h\}\} \subseteq [\![\Theta_1]\!] \bullet \{\{h\}\}$. □

Lemma A.17 (Order Remove). *If* $\Theta_1' < \Theta_1$ *then* $[\![\texttt{Remove}\ \Theta_1'\ h]\!] \subseteq [\![\texttt{Remove}\ \Theta_1\ h]\!]$.

Proof. Since $[\![\texttt{Remove}\ \Theta_1'\ h]\!] == [\![\Theta_1']\!] \setminus h$ and $[\![\texttt{Remove}\ \Theta_1\ h]\!] == [\![\Theta_1]\!] \setminus h$ by Lemma A.4, we can equivalently show that $[\![\Theta_1']\!] \setminus h \subseteq [\![\Theta_1]\!] \setminus h$, which follows from set theory. □

Lemma A.18 (Order Restrict). *If* $\Theta_1' < \Theta_1$ *then* $[\![\texttt{Restrict}\ \Theta_1'\ h]\!] \subseteq [\![\texttt{Restrict}\ \Theta_1\ h]\!]$

Proof. By Lemma 4.1, $[\![\texttt{Restrict}\ \Theta_1'\ h]\!] == [\![\Theta_1']\!]|h$ and $[\![\texttt{Restrict}\ \Theta_1\ h]\!] == [\![\Theta_1]\!]|h$. By set theory, $[\![\Theta_1']\!]|h \subseteq [\![\Theta_1]\!]|h$ when $[\![\Theta_1']\!] \subseteq [\![\Theta_1]\!]$, so we are done. □

Lemma A.19 (Order NegRestrict). *If* $\Theta_1' < \Theta_1$ *then*
$[\![\texttt{NegRestrict}\ \Theta_1'\ h]\!] \subseteq [\![\texttt{NegRestrict}\ \Theta_1\ h]\!]$

Proof. By Lemma 4.2, $[\![\texttt{NegRestrict}\ \Theta_1'\ h]\!] == [\![\Theta_1']\!]|\neg h$ and $[\![\texttt{NegRestrict}\ \Theta_1\ h]\!] == [\![\Theta_1]\!]|\neg h$. By set theory, $[\![\Theta_1']\!]|\neg h \subseteq [\![\Theta_1]\!]|\neg h$ when $[\![\Theta_1']\!] \subseteq [\![\Theta_1]\!]$, so we are done. □

Lemma A.20 (Order Include). *If* $\Theta_1' < \Theta_1$ *and* $\texttt{Includes}\ \Theta_1\ h$ *then* $\texttt{Includes}\ \Theta_1'\ h$.

Proof. By Lemma A.3, $\texttt{Includes}\ \Theta_1'\ h = h \sqsubset \Theta_1'$. By the same lemma, $\texttt{Includes}\ \Theta_1\ h = h \sqsubset \Theta_1$. Let $S \in [\![\Theta_1']\!]$ to show $h \in S$ and hence $h \sqsubset \Theta_1'$. Since $[\![\Theta_1']\!] \subseteq [\![\Theta_1]\!]$, by assumption and definition of $<$, then $S \in [\![\Theta_1]\!]$. Since $h \sqsubset \Theta_1$, conclude $h \in S$ and we are done. □

Lemma A.21 (Order Empty). *If* $\Theta_1' < \Theta_1$ *and* $\texttt{Empty}\ \Theta_1$ *then* $\texttt{Empty}\ \Theta_1'$.

Proof. By definition of $<$, $[\![\Theta_1']\!] \subseteq [\![\Theta_1]\!]$. By Lemma A.5 and assumption $\texttt{Empty}\ \Theta_1$ follows $[\![\Theta_1]\!] = \{\}$. By set theory $[\![\Theta_1']\!] = \{\}$. The result follows by Lemma A.5. □

Lemma A.22 (Order Choice). *If* $\Theta_a' < \Theta_a$ *and* $\Theta_b' < \Theta_b$ *then* $\Theta_a' + \Theta_b' < \Theta_a + \Theta_b$.

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

Lemma A.23 (Expression Type Bounds). *If* $\Gamma; \Theta \vdash e : \tau$ *and* $\Theta' < \Theta$, *then* $\Gamma; \Theta' \vdash e : \tau$.

Proof. By induction on the typing derivation.

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

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

***Case* T-Field:** We know $e = h.f$, and $\texttt{Includes}\ \Theta\ h$ and $\Theta' < \Theta$. By Lemma A.20, we know $\texttt{Includes}\ \Theta'\ \texttt{h}$ and the result follows by T-Field.

□

Lemma A.24 (Action Type Bounds). *If* $\Gamma; \Theta_1 \vdash a : \bar{\tau} \to \Theta_2$ *and* $\Theta_1' < \Theta_1$, *then* $\exists \Theta_2'. \Gamma; \Theta_1' \vdash a : \bar{\tau} \to \Theta_2'$ *and* $\Theta_2' < \Theta_2$.

Proof. Rule T-Action is the only rule by which we have concluded that $\Gamma; \Theta_1 \vdash a : \bar{\tau} \to \Theta_2$. This rule gives us two facts: we know $a = \lambda \bar{x} : \bar{\tau}.c$, and $\Gamma, \bar{x} : \bar{\tau} \vdash c : \Theta_1 \Rrightarrow \Theta_2$.

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

Case $c = add(h)$**:** The only typing rule that applies is T-Add, so we know $\Theta_2 = \Theta_1 \cdot h$. Now let $\Theta'_2 = \Theta'_1 \cdot h$. Then T-Add shows $\Gamma, \bar{x} : \bar{\tau} \vdash add(h) : \Theta'_1 \Rightarrow \Theta'_1 \cdot h$ and $\Theta'_1 \cdot h < \Theta_1 \cdot h$ follows by Lemma A.16, and we are done.

Case $c = remove(h)$**:** The only typing rule that could have applied is T-Remove, so we know that $\Theta_2 = \texttt{Remove}\ \Theta_1\ h$. Let $\Theta'_2 = \texttt{Remove}\ \Theta'_1\ h$. Then T-Remove shows $\Gamma, \bar{x} : \bar{\tau} \vdash remove(h) : \Theta'_1 \Rightarrow Remove\ \Theta_1\ h$ and $Remove\ \Theta'_1\ h < Remove\ \Theta_1\ h$ follows by Lemma A.17 and by definition of $<$.

Case $c = h.f = v$**:** The only typing rule that could have applied is T-Mod, so we know that $\Theta_2 = \Theta_1$. Let $\Theta'_2 = \Theta'_1$, which proves $\Theta'_2 < \Theta_2$ by assumption. We know by our case assumption that $\Gamma, \bar{x} : \bar{\tau}; \Theta_1 \vdash e : \mathcal{F}(h, f)$ and $\texttt{Includes}\ \Theta_1\ h$.

By T-Mod, we only need to show that

(1) $\Gamma, \bar{x} : \bar{\tau}; \Theta'_1 \vdash e : \mathcal{F}(h, f)$, which follows by Lemma A.23 and

(2) $\texttt{Includes}\ \Theta'_1\ h$, which follows by Lemma A.20.

Case $c = c_1; c_2$**:** The only rule that could have applied is T-Seq, so we know that $\Gamma, \bar{x} : \bar{\tau} \vdash c_1 : \Theta_1 \Rightarrow \Theta_{11}$, and $\Gamma, \bar{x} : \bar{\tau} \vdash c_2 : \Theta_{11} \Rightarrow \Theta_2$.

The inductive hypothesis on c_1 gives us a $\Theta'_{11} < \Theta_{11}$ such that $\Gamma, \bar{x} : \bar{\tau} \vdash c_1 : \Theta'_1 \Rightarrow \Theta'_{11}$.

The inductive hypothesis on c_2 gives us a $\Theta'_2 < \Theta_2$ such that $\Gamma, \bar{x} : \bar{\tau} \vdash c_2 : \Theta'_{11} \Rightarrow \Theta'_2$. The result follows by T-Seq.

□

Lemma A.25 (Control Type Bounds)**.** *If* $\Gamma \vdash c : \Theta_1 \Rightarrow \Theta_2$ *and* $\Theta'_1 < \Theta_1$, *then* $\exists \Theta'_2 . \Gamma \vdash c : \Theta'_1 \Rightarrow \Theta'_2$ *and* $\Theta'_2 < \Theta_2$.

Proof. By induction on a derivation of $\Gamma \vdash c : \Theta_1 \Rightarrow \Theta_2$. We refer to the general assumptions as follows:

(A) $\Gamma \vdash c : \Theta_1 \Rightarrow \Theta_2$ and

(B) $\Theta'_1 < \Theta_1$

Similarly, we refer to the proof goals as

(1) $\exists \Theta'_2 . \Gamma \vdash c : \Theta'_1 \Rightarrow \Theta'_2$

(2) $\Theta'_2 < \Theta_2$

Case **T-Zero:** We know $\texttt{Empty}\ \Theta_1$. By assumption (B) and Lemma A.21 we have $\texttt{Empty}\ \Theta'_1$. Let $\Theta'_2 = \Theta_2$. We have $\Gamma \vdash c : \Theta'_1 \Rightarrow \Theta_2$ by T-Zero, proving (1), and and $\Theta'_2 < \Theta_2$ by reflexivity, proving (2).

Case **T-Skip:** We know $c = skip$ and $\Theta_2 = \Theta_1$. Let $\Theta'_2 = \Theta'_1$. Then by assumption (B) $\Theta'_2 = \Theta'_1 < \Theta_1 = \Theta_2$, proving (2) and $\Gamma \vdash skip : \Theta'_1 \Rightarrow \Theta'_1$ by T-Skip, proving (1).

Case **T-Emit:** We know $c = emit(h)$ and $\Theta_2 = \Theta_1$. Let $\Theta'_2 = \Theta'_1$. Then by assumption (B), $\Theta'_2 = \Theta'_1 < \Theta_1 = \Theta_2$, proving (2) and $\Gamma \vdash emit(h) : \Theta'_1 \Rightarrow \Theta'_1$ by T-Emit, proving (1).

***Case* T-ADD:** We know $c = add(h)$ and $\Theta_2 = \Theta_1 \cdot h$. Let $\Theta'_2 = \Theta'_1 \cdot h$. (1) follows since we can prove $\Gamma \vdash add(h) : \Theta'_1 \Rightarrow \Theta'_1 \cdot h$ by T-ADD. (2), i.e., $\Theta'_1 \cdot h < \Theta_1 \cdot h$, follows from Lemma A.16.

***Case* T-EXTR:** We know $c = extract(h)$ and $\Theta_2 = \Theta_1 \cdot h$. Let $\Theta'_2 = \Theta'_1 \cdot h$. (1) follows since we can prove $\Gamma \vdash extract(h) : \Theta'_1 \Rightarrow \Theta'_1 \cdot h$ by T-EXTRACT. (2), i.e., $\Theta'_1 \cdot h < \Theta_1 \cdot h$, follows from Lemma A.16.

***Case* T-REM:** We know $c = remove(h)$ and $\Theta_2 = \texttt{Remove}\ \Theta_1\ h$. Let $\Theta'_2 = \texttt{Remove}\ \Theta'_1\ h$. (1) follows by T-REM. For (2) we have to show that $\texttt{Remove}\ \Theta'_1\ h < \texttt{Remove}\ \Theta_1\ h$, which follows from Lemma A.17.

***Case* T-MOD:** We know $c = h.f = e$ and $\Theta_2 = \Theta_1$ and $\texttt{Includes}\ \Theta_1\ h$, and $\mathcal{F}(h, f) = \tau$ and $\Gamma; \Theta_1 \vdash e : \tau$. Let $\Theta'_2 = \Theta'_1$. (1) follows by T-MOD, if we can show $\texttt{Includes}\ \Theta'_1\ h$, which follows by assumption (B) and Lemma A.20. (2) follows by assumption (B).

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

***Case* T-IF VALID:** We know $c = valid(h)\ c_1\ else\ c_2$ and $\Gamma \vdash c_1 : \texttt{Restrict}\ \Theta_1\ h \Rightarrow \Theta_t, \Gamma \vdash c_2 : \texttt{NegRestrict}\ \Theta_1\ h \Rightarrow \Theta_f, \Theta_2 = \Theta_t + \Theta_f$. Let $\Theta'_2 = \texttt{Restrict}\ \Theta'_1\ h + \texttt{NegRestrict}\ \Theta'_1\ h$. (1) is immediate from T-IF VALID. (2) follows from Lemmas A.18, A.19 and A.22.

***Case* T-IF:** We know $c = if\ (e)\ c_1\ else\ c_2$, and $\Gamma \vdash c_1 : \Theta_1 \Rightarrow \Theta_{11}$, and $\Gamma \vdash c_2 : \Theta_1 \Rightarrow \Theta_{12}$, and $\Gamma; \Theta_1 \vdash e : Bool$.

By induction hypothesis, there exists Θ'_{11} such that (1a) $\Gamma \vdash c_1 : \Theta'_1 \Rightarrow \Theta'_{11}$ and (2a) $\Theta'_{11} < \Theta_{11}$. Also by induction hypothesis, there exists Θ'_{12} such that (1b) $\Gamma \vdash c_2 : \Theta'_1 \Rightarrow \Theta'_{12}$ and (2b) $\Theta'_{12} < \Theta_{12}$. Let $\Theta'_2 = \Theta'_{11} + \Theta'_{12}$. By Lemma A.23 follows $\Gamma; \Theta'_1 \vdash e : Bool$. (1) then follows by T-IF with (1a) and (1b). (2) follows by Lemma A.22 with (2a) and (2b).

***Case* T-APPLY:** We know

- $c = t.apply()$
- $\Theta_2 = \Theta_{11} + \Theta_{12} + ... + \Theta_{1n}$
- $t.actions = a_1 + a_2 + ... + a_n$
- $\cdot; \Theta_1 \vdash e_j : \tau_j$ for $j = 1, ..., m$
- $\mathcal{CV}(t) = (S_1 ... S_n)$
- $(e_1 ... e_m) = \{e_i | (e_i, m_i) \in t.reads() \wedge \neg \mathsf{maskable}(t, e_i, m_i)\}$
- $\cdot; \texttt{Restrict}\ \Theta_1\ S_i \vdash a_i : \bar{\tau}_i \rightarrow \Theta_{1i}$

We want to construct $\Theta'_2 < \Theta_2$ such that $\Gamma \vdash t.apply() : \Theta'_1 \Rightarrow \Theta'_2$. By repeated application of Lemma A.18, $\texttt{Restrict}\ \Theta'_1\ S_i < \texttt{Restrict}\ \Theta_1\ S_i$. For every i apply Lemma A.24, which gives us $\Gamma; \texttt{Restrict}\ \Theta'_1\ S_i \vdash a : \bar{\tau} \rightarrow \Theta_{1_i}$ and $\Theta'_{1i} < \Theta_{1i}$. Also for every j apply Lemma A.23, which gives us $\cdot; \Theta'_1 \vdash e_j : \tau_j$ for $j = 1, ..., m$

Let $\Theta'_2 = \sum_i \Theta'_{1i}$. (2) follows by T-APPLY. To show (1), i.e., $\Theta'_2 = \sum_i \Theta'_{1i} < \sum_i \Theta_{1i} = \Theta_2$. We know $\Theta'_{1i} < \Theta_{1i}$ for all i. The result follows by repeated application of Lemma A.22.

□

Theorem A.26 (Preservation). *If* $\Gamma \vdash c : \Theta_1 \Rrightarrow \Theta_2$ *and* $\langle I, O, H, c\rangle \to \langle I', O', H', c'\rangle$, *where* $H \vDash \Theta_1$, *then* $\exists \Theta'_1, \Theta'_2.\ \Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ *where* $H' \vDash \Theta'_1$ *and* $\Theta'_2 < \Theta_2$.

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

***Case* T-Skip:** $c = skip$ and $\Theta_2 = \Theta_1$
Vacuously holds as there is no c' such that $\langle I, O, H, c\rangle \to \langle I', O', H', c'\rangle$.

***Case* T-Extr:** $c = extract(h)$ and $\Theta_2 = \Theta_1 \cdot h$
The only evaluation rule that applies to c is E-Extr, so we also have $c' = skip$ and $\mathcal{HT}(h) = \eta$ and $H' = H[h \mapsto v]$ where $deserialize_\eta(I) = (v, I')$.

Let $\Theta'_1 = \Theta'_2 = \Theta_2$. We have $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ by T-Skip, we have $H' \vDash \Theta'_2$ by Lemma A.14, and we have $\Theta'_2 < \Theta_2$ by reflexivity.

***Case* T-Emit:** $c = emit(h)$ and $\Theta_2 = \Theta_1$.
There are two evaluation rules that apply to c, E-Emit and E-EmitInvalid. In either case, $c' = skip$ and $H' = H$.

Let $\Theta'_1 = \Theta'_2 = \Theta_1$. We have $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ by T-Skip, we have $H' \vDash \Theta'_1$ by assumption, and we have $\Theta'_2 < \Theta_2$ by reflexivity.

***Case* T-Seq:** $c = c_1; c_2$ and $\Gamma \vdash c_1 : \Theta_1 \Rrightarrow \Theta_{12}$ and $\Gamma \vdash c_2 : \Theta_{12} \Rrightarrow \Theta_2$
There are two evaluation rules that apply to c, E-Seq1 and E-Seq.

***Subcase* E-Seq:** $c' = c_2$ and $H' = H$
By inversion of $\Gamma \vdash c_1 : \Theta_1 \Rrightarrow \Theta_{12}$ we have $\Theta_{12} = \Theta_1$. Let $\Theta'_1 = \Theta_1$ and $\Theta'_2 = \Theta_2$. We have $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ by assumption, we have $H \vDash \Theta'_1$ also by assumption, and $\Theta'_2 < \Theta_2$ by reflexivity.

***Subcase* E-Seq1:** $c' = c'_1; c_2$ and $\langle I, O, H, c_1\rangle \to \langle I', O', H', c'_1\rangle$.
By IH we have $\Gamma \vdash c_1 : \Theta'_1 \Rrightarrow \Theta'_{12}$ such that $H' \vDash \Theta'_1$ and $\Theta'_{12} < \Theta_{12}$. By Lemma A.25 we have $\Gamma \vdash c_2 : \Theta'_{12} \Rrightarrow \Theta'_2$ for some $\Theta'_2 < \Theta_2$. We have $\Gamma \vdash c_1; c_2 : \Theta'_1 \Rrightarrow \Theta'_2$ by T-Seq, which finishes the case.

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

There are three evaluation rules that apply to c, E-If, E-IfTrue, and E-IfFalse.

***Subcase* E-If:** $c' = if\ (e')\ c_1\ else\ c_2$ and $H' = H$
Let $\Theta'_1 = \Theta_1$ and $\Theta'_2 = \Theta_2$. We have $\Gamma \vdash if\ (e)\ c_1\ else\ c_2 : \Theta'_1 \Rrightarrow \Theta'_2$ by T-If, we have $H \vDash \Theta_1$ by assumption, and we have $\Theta_2 < \Theta'_2$ by reflexivity.

***Subcase* E-IfTrue:** $c' = c_1$ and $H' = H$.
Let $\Theta'_1 = \Theta_1$ and $\Theta'_2 = \Theta_{12}$. We have $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ by assumption, we have $H \vDash \Theta'_1$ also by assumption, and we have $\Theta'_2 < \Theta_2$ by the definition of $<$ and the semantics of types.

***Subcase* E-IfFalse:** $c' = c_2$ and $H' = H$.
Symmetric to the previous case.

***Case* T-IfValid:** $c = valid(h)\ c_1\ else\ c_2$ and $\Gamma \vdash c_1 : \texttt{Restrict}\ \Theta_1\ h \Rrightarrow \Theta_{12}$ and $\Gamma \vdash c_2 : \texttt{NegRestrict}\ \Theta_1\ h \Rrightarrow \Theta_{22}$ and $\Theta_2 = \Theta_{12} + \Theta_{22}$. There are two evaluation rules that apply to c, E-IfValidTrue and E-IfValidFalse

***Subcase* E-IF VALIDTRUE:** $c' = c_1$ and $h \in dom(H)$ and $H' = H$.
Let $\Theta'_1 = \texttt{Restrict}\ \Theta_1\ h$ and $\Theta'_2 = \Theta_{12}$. We have $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ by assumption, we have $H \vDash \Theta'_1$ by Lemma A.10, and we have $\Theta'_2 < \Theta_2$ by the definition of $<$ and semantics of types.

***Subcase* E-IF VALIDFALSE:** $c' = c_2$ and $h \notin dom(H)$ and $H' = H$.
Symmetric to the previous case.

***Case* T-APPLY:** $c = t.apply()$ and $\mathcal{CV}(t) = \bar{S}$ and $t.actions = \bar{a}$ and $\bar{e} = \{e_j \mid (e_j, m_j) \in t.reads() \wedge \neg\mathsf{maskable}(t, e_j, m_j)\}$ and $\cdot; \Theta \vdash e_i : \tau_i$ for $e_i \in \bar{e}$ and $\Theta_2 = \sum(\Theta'_i)$ and $\cdot; \texttt{Restrict}\ \Theta_1\ S_i \vdash a_i : \bar{\tau}_i \to \Theta'_i$ for $a_i \in a$

There is only one evaluation rule that applies to c, E-APPLY.

It follows that $\mathcal{CA}(t, H) = (a_i, \bar{v})$, and $c' = c_i[\bar{v}/\bar{x}]$ where $\mathcal{A}(a_i) = \lambda\bar{x}.\ c_i$. Next, inverting T-ACTION, we have $\cdot, \bar{x} : \bar{\tau}_i \vdash c_i : \texttt{Restrict}\ \Theta\ S_i \Rrightarrow \Theta'_i$. By Proposition 2, we have $\cdot; \cdot \vdash \bar{v} : \bar{\tau}_i$. Hence, by Lemma A.13, we have $\cdot \vdash c_i[\bar{v}/\bar{x}] : \texttt{Restrict}\ \Theta\ S_i \Rrightarrow \Theta'_i$.

Let $\Theta'_1 = \texttt{Restrict}\ \Theta\ S_i$ and $\Theta'_2 = \Theta'_i$. We have already shown that $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$. We have that $H' \vDash \Theta'_1$ by Proposition 3, and we have $\Theta'_2 < \Theta_2$ by the definition of $<$ and the semantics of union types.

***Case* T-ADD:** $c = add(h)$ and $\Theta_2 = \Theta_1 \cdot h$
There are two evaluation rules that apply to c, E-ADD and E-ADDVALID.

***Subcase* E-ADD:** $c' = skip$ and $\mathcal{HT}(h) = \eta$ and $init_\eta = v$ and $H' = H[h \mapsto v]$
Let $\Theta'_1 = \Theta'_2 = \Theta_2$. We have $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ by T-SKIP, we have $H' \vDash \Theta'_1$ by Lemma A.14, and we have $\Theta'_2 < \Theta_2$ by reflexivity.

***Subcase* E-ADDVALID:** $c' = skip$ and $H' = H$
Let $\Theta'_1 = \Theta'_2 = \Theta_2$. We have $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ by T-SKIP, We have $H' \vDash \Theta'_1$ by Lemma A.12 and Lemma A.14 since $dom(H') = dom(H[h \mapsto v])$ for any v, and we have $\Theta'_2 < \Theta_2$ by reflexivity.

***Case* T-REM:** $c = remove(h)$ and $\Theta_2 = \texttt{Remove}\ \Theta_1\ h$
There is only one evaluation rule that applies to c, E-REM, so we have $c' = skip$ and $H' = H \setminus h$. Let $\Theta'_1 = \Theta'_2 = \texttt{Remove}\ \Theta_1\ h$. We have $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ by T-SKIP, we have $H' \vDash \Theta'_1$ by Lemma A.15, and we have $\Theta'_2 < \Theta_2$ by reflexivity.

***Case* T-MOD:** $c = h.f = e$ and $\texttt{Includes}\ \Theta_1\ h$ and $\mathcal{HT}(h, f) = \tau_i$ and $\cdot; \Theta_1 \vdash e : \tau_i$ and $\Theta_2 = \Theta_1$
There are two evaluation rules that applies to c, E-MOD1 and E-MOD.

***Subcase* E-MOD1:** $c' = h.f = e'$ and $e \to e'$ and $H' = H$
By preservation for expressions we have $\cdot; \Theta_1 \vdash e' : \tau_i$. Let $\Theta'_1 = \Theta'_2 = \Theta_1$. We have $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ by T-MOD, we have $H' \vDash \Theta'_1$ by assumption, and we have $\Theta'_2 < \Theta_2$ by reflexivity.

***Subcase* E-MOD:** $c' = skip$ and $dom(H') = dom(H)$
Let $\Theta'_1 = \Theta'_2 = \Theta_1$. We have $\Gamma \vdash c' : \Theta'_1 \Rrightarrow \Theta'_2$ by T-SKIP, we have $H' \vDash \Theta'_1$ by Lemma A.12, and we have $\Theta'_2 < \Theta_2$ by reflexivity.

***Case* T-ZERO:** $\texttt{Empty}\ \Theta_1$
By Lemma A.7, we have $dom(H) \in [\![\Theta_1]\!]$. By Lemma A.5, we have $[\![\Theta_1]\!] = \{\}$, which is a contradiction. □

A.2 Π4

A.2.1 Safety

Lemma A.27 (Semantic Entailment). *If* $(I, O, H) \vDash_{\mathcal{E}} \tau$*, then* $(I, O, H) \in [\![\tau]\!]_{\mathcal{E}}$

Proof. By induction on τ.

Case $\tau = \varnothing$**:** Immediate, since $(I, O, H) \vDash_{\mathcal{E}} \varnothing$ is a contradiction.

Case $\tau = \top$**:** Immediate, since $[\![\top]\!]_{\mathcal{E}} = \mathcal{H}$.

Case $\tau = \Sigma x\colon \tau_1.\tau_2$**:** By inversion of entailment, we get

(A1) $(I, O, H) = (I_1@I_2, O_1@O_2, H_1 \cup H_2)$ and

(A2) $(I_1, O_1, H_1) \vDash_{\mathcal{E}} \tau_1$ and

(A3) $(I_2, O_2, H_2) \vDash_{\mathcal{E}[x \mapsto (I_1, O_1, H_1)]} \tau_2$

By (A2) respectively (A3) and the induction hypothesis, we get

(A4) $(I_1, O_1, H_1) \in [\![\tau_1]\!]_{\mathcal{E}}$ and

(A5) $(I_2, O_2, H_2) \in [\![\tau_2]\!]_{\mathcal{E}[x \mapsto (I_1, O_1, H_1)]}$.

To show that $(I, O, H) \in [\![\Sigma x\colon \tau_1.\tau_2]\!]_{\mathcal{E}} = \{h_1 +\!\!+ h_2 \mid h_1 \in [\![\tau_1]\!]_{\mathcal{E}} \wedge h_2 \in [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h_1]}\}$, we have to show that (I, O, H) is the concatenation of two heaps h_1 and h_2, where $h_1 \in [\![\tau_1]\!]_{\mathcal{E}}$ and $h_2 \in [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h_1]}$, which follows from (A1), (A4) and (A5).

Case $\tau = \tau_1 + \tau_2$**:** By inversion of entailment, either $(I, O, H) \vDash_{\mathcal{E}} \tau_1$ or $(I, O, H) \vDash_{\mathcal{E}} \tau_2$. To show that $(I, O, H) \in [\![\tau_1 + \tau_2]\!]_{\mathcal{E}} = [\![\tau_1]\!]_{\mathcal{E}} \cup [\![\tau_2]\!]_{\mathcal{E}}$, we have to show that $(I, O, H) \in [\![\tau_1]\!]_{\mathcal{E}}$ or $(I, O, H) \in [\![\tau_2]\!]_{\mathcal{E}}$.

Subcase $(I, O, H) \vDash_{\mathcal{E}} \tau_1$**:** By induction hypothesis, $(I, O, H) \in [\![\tau_1]\!]_{\mathcal{E}}$. We can conclude $(I, O, H) \in [\![\tau_1 + \tau_2]\!]_{\mathcal{E}}$.

Subcase $(I, O, H) \vDash_{\mathcal{E}} \tau_2$**:** Symmetric to previous subcase.

Case $\tau = \{x : \tau_1 \mid \varphi\}$**:** By inversion of entailment, we get

(A1) $(I, O, H) \vDash_{\mathcal{E}} \tau$ and

(A2) $[\![\varphi]\!]_{\mathcal{E}[x \mapsto (I, O, H)]} = \mathit{true}$

To show that $(I, O, H) \in [\![\{x : \tau \mid \varphi\}]\!]_{\mathcal{E}} = \{h \mid h \in [\![\tau]\!]_{\mathcal{E}} \wedge [\![\varphi]\!]_{\mathcal{E}[x \mapsto h]}\}$, we have to show that $(I, O, H) \in [\![\tau]\!]_{\mathcal{E}}$ and that $[\![\varphi]\!]_{\mathcal{E}[x \mapsto (I, O, H)]} = \mathit{true}$. The first follows by induction hypothesis and (A1) and the latter by (A2).

Case $\tau = \tau_1[x \mapsto \tau_2]$**:** By inversion of entailment, we get

(A1) $(I_2, O_2, H_2) \vDash_{\mathcal{E}} \tau_2$ for some I_2, O_2, H_2 and

(A2) $(I, O, H) \vDash_{\mathcal{E}[x \mapsto (I_2, O_2, H_2)]} \tau_1$

To show that $(I, O, H) \in [\![\tau_1[x \mapsto \tau_2]]\!]_{\mathcal{E}} = \{h \mid h_2 \in [\![\tau_2]\!]_{\mathcal{E}} \wedge h \in [\![\tau_1]\!]_{\mathcal{E}[x \mapsto h_2]}\}$, we have to show that $(I, O, H) \in [\![\tau_1]\!]_{\mathcal{E}[x \mapsto h_2]}$ where $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$. By induction hypothesis and (A2) follows that $(I, O, H) \in [\![\tau_1]\!]_{\mathcal{E}[x \mapsto (I_2, O_2, H_2)]}$. $(I_2, O_2, H_2) \in [\![\tau_2]\!]_{\mathcal{E}}$ follows by induction hypothesis and (A1), which concludes this case.

□

Lemma A.28 (Semantic Containment Entails). *If* $(I, O, H) \in [\![\tau]\!]_{\mathcal{E}}$*, then* $(I, O, H) \vDash_{\mathcal{E}} \tau$.

Proof. By induction on τ.

Case $\tau = \varnothing$**:** Immediate, since there is no heap in $[\![\varnothing]\!]_{\mathcal{E}}$.

Case $\tau = \top$**:** Result directly follows by Ent-Top.

Case $\tau = \Sigma x : \tau_1.\tau_2$**:** By the semantics of heap types, all heaps $h \in [\![\Sigma x : \tau_1.\tau_2]\!]$ have the form $h = h_1 \mathbin{++} h_2$, where $h_1 = (I_1, O_1, H_1) \in [\![\tau_1]\!]_{\mathcal{E}}$ and $h_2 = (I_2, O_2, H_2) \in [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h_1]}$. By applying the induction hypothesis, we get $(I_1, O_1, H_1) \vDash_{\mathcal{E}} \tau_1$ and $(I_2, O_2, H_2) \vDash_{\mathcal{E}[x \mapsto h_1]} \tau_2$. The result directly follows by Ent-Sigma.

Case $\tau = \tau_1 + \tau_2$**:** By the semantics of heap types, for any $h \in [\![\tau_1 + \tau_2]\!]_{\mathcal{E}}$ holds that either $h \in [\![\tau_1]\!]_{\mathcal{E}}$ or $h \in [\![\tau_2]\!]_{\mathcal{E}}$.

Subcase $h \in [\![\tau_1]\!]_{\mathcal{E}}$**:** By induction hypothesis, $h \vDash_{\mathcal{E}} \tau_1$. The result directly follows by Ent-ChoiceL.

Subcase $h \in [\![\tau_1]\!]_{\mathcal{E}}$**:** Symmetric to previous subcase.

Case $\tau = \{y : \tau_1 \mid \varphi\}$**:** By the semantics of heap types, $h \in [\![\tau_1]\!]_{\mathcal{E}}$ and $[\![\varphi]\!]_{\mathcal{E}[x \mapsto h]} = \mathit{true}$. By induction hypothesis, $h \vDash_{\mathcal{E}} \tau$. The result directly follows by Ent-Refine.

Case $\tau = \tau_1[x \mapsto \tau_2]$**:** By the semantics of heap types, $h \in [\![\tau_1]\!]_{\mathcal{E}[x \mapsto h_2]}$ where $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$. By induction hypothesis, $h_2 \vDash_{\mathcal{E}} \tau_2$ and $h \vDash_{\mathcal{E}[x \mapsto h_2]} \tau_1$. The result directly follows by Ent-Subst.

□

Lemma A.29 (Subtype Entailment). *If* $(I, O, H) \vDash_{\mathcal{E}} \tau_1$ *and* $\mathcal{E} \vDash \Gamma$ *and* $\Gamma \vdash \tau_1 <: \tau_2$*, then* $(I, O, H) \vDash_{\mathcal{E}} \tau_2$.

Proof. By Lemma A.27, $(I, O, H) \in [\![\tau_1]\!]_{\mathcal{E}}$. With $\mathcal{E} \vDash \Gamma$ and by definition of subtyping, $(I, O, H) \in [\![\tau_2]\!]_{\mathcal{E}}$. The result follows by Lemma A.28. □

Lemma A.30 (Extended Environment Entails). *If* $\mathcal{E} \vDash \Gamma$ *and* $(I, O, H) \vDash_{\mathcal{E}} \tau$ *and* $x \notin \mathit{dom}(\mathcal{E})$*, then* $\mathcal{E}[x \mapsto (I, O, H)] \vDash \Gamma, x : \tau$.

Proof. By definition of entailment between environments and typing contexts and by assumptions. □

Progress

Lemma A.31 (Included Instances in Domain). *If* $(I, O, H) \vDash_{\mathcal{E}} \tau$ *and* $\mathsf{Includes}\ \Gamma\ \tau\ \iota$*, then* $\iota \in \mathit{dom}(H)$.

Proof. By Lemma A.27, $(I, O, H) \in [\![\tau]\!]_{\mathcal{E}}$. By assumption $\mathsf{Includes}\ \Gamma\ \tau\ \iota$ and by definition of inclusion, $\forall h \in [\![\tau]\!]_{\mathcal{E}}.\iota \in \mathit{dom}(h)$, we can conclude that $\iota \in \mathit{dom}(H)$. □

Lemma A.32 (Excluded Instances not in Domain). *If* $(I, O, H) \vDash_{\mathcal{E}} \tau$ *and* $\mathsf{Excludes}\ \Gamma\ \tau\ \iota$*, then* $\iota \notin \mathit{dom}(H)$.

Proof. By Lemma A.27, $(I, O, H) \in [\![\tau]\!]_{\mathcal{E}}$. By assumption $\mathsf{Excludes}\ \Gamma\ \tau\ \iota$ and by definition of exclusion, $\forall h \in [\![\tau]\!]_E.\iota \notin dom(h)$, we can conclude that $\iota \notin dom(H)$. □

Lemma A.33 (Expression Progress). *If $\Gamma; \tau \vdash e : \sigma$ and $\mathcal{E} \vDash \Gamma$ and $(I, O, H) \vDash_{\mathcal{E}} \tau$, then either e is a value or $\exists e'.\langle I, O, H, e\rangle \to e'$.*

Lemma A.34 (Formulae Progress). *If $\Gamma; \tau \vdash \varphi : \mathbb{B}$ and $\mathcal{E} \vDash \Gamma$ and $(I, O, H) \vDash_{\mathcal{E}} \tau$, then either φ is a value or $\exists \varphi'.\langle I, O, H, \varphi\rangle \to \varphi'$*

Theorem A.35 (Progress). *If $\Gamma \vdash c : (x : \tau_1) \to \tau_2$ and $\mathcal{E} \vDash \Gamma$ and $(I, O, H) \vDash_{\mathcal{E}} \tau_1$, then either $c = skip$ or there exists $\langle I', O', H', c'\rangle$ such that $\langle I, O, H, c\rangle \to \langle I', O', H', c'\rangle$.*

Proof. By induction on typing derivations of $\Gamma \vdash c : (x : \tau_1) \to \tau_2$.

***Case* T-Skip:** $c = skip$
The result is immediate.

***Case* T-Extract:** $c = extract(\iota)$ and $\Gamma \vdash \mathsf{sizeof}_{pkt_{in}}(\tau_1) \geq \mathsf{sizeof}(\iota)$
By inversion of $(I, O, H) \vDash_{\mathcal{E}} \tau_1$, we know that I contains enough bits such that $deserialize_\eta(I)$ does not fail. Let $(v, I') = deserialize_\eta(I)$ and $O' = O$ and $H' = H[\iota \mapsto v]$ and $c' = skip$. The result follows by E-Extract.

***Case* T-Reset:** $c = reset$
Let $I' = O@I, O' = \langle\rangle, H' = [\,]$ and $c' = skip$. The result follows by E-Reset.

***Case* T-Remit:** $c = remit(\iota)$ and $\mathsf{Includes}\ \Gamma\ \tau_1\ \iota$
By Lemma A.31 we know $\iota \in dom(H)$. Let $I' = I, O' = O@serialize_\eta(H(\iota)), H' = H$ and $c' = skip$. The result follows by E-Remit

***Case* T-Remove:** $c = remove(\iota)$ and $\mathsf{Includes}\ \Gamma\ \tau_1\ \iota$

By Lemma A.31, we can conclude that $\iota \in dom(H)$. Let $I' = I$ and $O' = O$ and $H' = H \setminus \iota$. The result follows by E-Remove.

***Case* T-Mod:** $c = \iota.f := e$ and $\mathsf{Includes}\ \Gamma\ \tau_1\ \iota$ and $\mathcal{F}(\iota, f) = \mathsf{BV}$ and $\Gamma; \tau_1 \vdash e : \mathsf{BV}$ and $\tau_2 = \{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = e[x/\mathsf{heap}]\}$
By Lemma A.33, either e is a value or there is some e' such that $\langle I, O, H, e\rangle \to e'$.

***Subcase* $e = v$:** By Lemma A.31, $\iota \in dom(H)$. Let $r = H(\iota)$ and $r' = \{r\ with\ f = v\}$. Let $I' = I, O' = O, H' = H[\iota \mapsto r]$ and $c' = skip$. The result follows by E-Mod.

***Subcase* $\langle I, O, H, e\rangle \to e'$:** Let $I' = I, O' = O, H' = H'$ and $c' = \iota.f := t'$. The result follows by E-Mod1.

***Case* T-Seq:** $c = c_1; c_2$ and $\Gamma \vdash c_1 : (x : \tau_1) \to \tau_1'$ and $\Gamma \vdash c_2 : (x : \tau_1') \to \tau_2$
By induction hypothesis, c_1 is either *skip* or there is some $\langle I', O', H', c_1'\rangle$, such that $\langle I, O, H, c_1\rangle \to \langle I', O', H', c_1'\rangle$. If $c_1 = skip$, let $I' = I, O' = O, H' = H$ and $c' = c_2$. The result follows by E-Seq. Otherwise, the result follows by E-Seq1.

***Case* T-If:** $c = if(\varphi)\ c_1\ else\ c_2$ and $\Gamma; \tau_1 \vdash e : \mathbb{B}$
By Lemma A.34, we have that φ is either true, false or there is some φ' such that $\langle I, O, H, \varphi\rangle \to \varphi'$.

***Subcase* $\varphi = \mathsf{true}$:** Let $I' = I, O' = O, H' = H$ and $c' = c_1$. The result follows by E-IfTrue.

Subcase $\varphi = \mathsf{false}$**:** Symmetric to previous subcase.

Subcase $\langle I, O, H, \varphi\rangle \to \varphi'$**:** Let $I' = I$ and $O' = O$ and $H' = H$. Further, let $c' = \mathit{if}\,(\varphi')\ c_1\ \mathit{else}\ c_2$. The result follows by E-If.

Case **T-Add:** $c = \mathit{add}(\iota)$ and Excludes $\Gamma\ \tau\ \iota$.
By Lemma A.32, $\iota \notin \mathit{dom}(H)$. The result follows by E-Add.

Case **T-Ascribe:** $c = c_a\ \mathit{as}\ (x : \tau_{a_1}) \to \tau_{a_2}$. Let $I' = I$, $O' = O$, $H' = H$ and $c' = c_a$. The result follows by E-Ascribe.

Case **T-Sub:** $\Gamma \vdash \tau_1 <: \tau_3$ and $\Gamma, x : \tau_1 \vdash \tau_4 <: \tau_2$ and $\Gamma \vdash c : (x : \tau_3) \to \tau_4$. By Lemma A.29, $(I, O, H) \vDash_{\mathcal{E}} \tau_3$. By IH, $c = \mathit{skip}$ or there exists I', O', H', c' s.t. $\langle I, O, H, c\rangle \to \langle I', O', H', c'\rangle$. The result follows directly.

□

Preservation

Lemma A.36 (Semantic Chomp Expression)**.** *For all expressions e, heaps h and h′, environments $\mathcal{E}$ and $\mathcal{E}'$ and variables x, if* $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$*, and* $\mathcal{E}' = \mathcal{E}[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v])]$ *and, if* $x \in \mathit{dom}(\mathcal{E})$*,* $v = \mathcal{E}(x)(\iota)@h(pkt_{in})[0:1]$ *and* $\mathcal{E}(x)(pkt_{in}) = \langle\rangle$ *and* $\mathcal{E}(x)(pkt_{out}) = \langle\rangle$*, and otherwise* $v = h(pkt_{in})[0{:}1]$ *and x not free in e, then*

$$[\![e]\!]_{\mathcal{E}[y\mapsto h]} = [\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(e, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}$$

Proof. Proof by induction on e. We only consider expressions referencing pkt_{in}. All other expressions are not affected by chomping, and therefore the semantic is unchanged.

Case $e = z.pkt_{in}[l{:}r]$**:** Case distinction on $z = y$:

Subcase $z \neq y$**:**

$$\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(z.pkt_{in}[l{:}r], y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &[\![\mathsf{heapRef}_1(z.pkt_{in}[l{:}r], \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &[\![z.pkt_{in}[l{:}r]]\!]_{\mathcal{E}'[y\mapsto h']}
\end{aligned}$$

If $z \neq x$, z binds to some heap in $\mathcal{E}$, which must also be contained in $\mathcal{E}'$ unchanged. If $z = x$, by assumption, $x.pkt_{in}$ maps to the empty bit vector, both in $\mathcal{E}$ and $\mathcal{E}'$.

$$= [\![z.pkt_{in}[l{:}r]]\!]_{\mathcal{E}[y\mapsto h]}$$

Subcase $z = y, r \leq 1$**:**

$$\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(y.pkt_{in}[0{:}1], y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(y.pkt_{in}[0{:}1], y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &[\![\mathsf{heapRef}_1(\mathsf{b}_0 :: \langle\rangle, \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &[\![x.\iota[\mathsf{sizeof}(\iota) - 1{:}\mathsf{sizeof}(\iota) - 1 + 1]@\langle\rangle]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &[\![x.\iota[|v| - 1{:}|v|]]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &\mathcal{E}'(x)(\iota)[|v| - 1]\\
=\ &h(pkt_{in})[0{:}1]\\
=\ &[\![y.pkt_{in}[0{:}1]]\!]_{\mathcal{E}[y\mapsto h]}
\end{aligned}$$

Subcase $z = y, l = 0$**:**

$$
\begin{aligned}
& [\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(y.pkt_{in}[0{:}r], y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']} \\
= & [\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(y.pkt_{in}[0{:}r], y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']} \\
= & [\![\mathsf{heapRef}_1(\mathsf{b}_0 :: y.pkt_{in}[0{:}r-1], \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']} \\
= & [\![x.\iota[\mathsf{sizeof}(\iota) - 1{:}\mathsf{sizeof}(\iota) - 1 + 1]@y.pkt_{in}[0{:}r-1]]\!]_{\mathcal{E}'[y \mapsto h']} \\
= & [\![x.\iota[|v| - 1{:}|v|]@y.pkt_{in}[0{:}r-1]]\!]_{\mathcal{E}'[y \mapsto h']} \\
= & \mathcal{E}'(x)(\iota)[|v| - 1{:}|v|]@h'(pkt_{in})[0{:}r-1]
\end{aligned}
$$

with $v = \mathcal{E}(x)(\iota)@h(pkt_{in})[0]$ follows

$$= h(pkt_{in})[0{:}1]@h'(pkt_{in})[0{:}r-1]$$

with $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$ follows

$$
\begin{aligned}
= & h(pkt_{in})[0{:}1]@h(pkt_{in})[1{:}r] \\
= & h(pkt_{in})[0{:}r] \\
= & [\![y.pkt_{in}[0{:}r]]\!]_{\mathcal{E}[y \mapsto h]}
\end{aligned}
$$

Subcase $z = y, l \neq 0$**:**

$$
\begin{aligned}
& [\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(y.pkt_{in}[l{:}r], y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']} \\
= & [\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(y.pkt_{in}[l{:}r], y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']} \\
= & [\![\mathsf{heapRef}_1(y.pkt_{in}[l-1{:}r-1], \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']} \\
= & [\![y.pkt_{in}[l-1{:}r-1]]\!]_{\mathcal{E}'[y \mapsto h']} \\
= & h'(pkt_{in})[l-1{:}r-1]
\end{aligned}
$$

with $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$ follows

$$
\begin{aligned}
= & h(pkt_{in})[l{:}r] \\
= & [\![y.pkt_{in}[l{:}r]]\!]_{\mathcal{E}[y \mapsto h]}
\end{aligned}
$$

Case $e = z.pkt_{in}$**:** Case distinction on $z = y$:

Subcase $z \neq y$**:** Symmetric to first subcase of previous case.

Subcase $z = y$**:**

$$
\begin{aligned}
& [\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(y.pkt_{in}, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']} \\
= & [\![\mathsf{heapRef}_1(\mathsf{b}_0 :: y.pkt_{in}, \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']} \\
= & [\![x.\iota[\mathsf{sizeof}(\iota) - 1{:}\mathsf{sizeof}(\iota) - 1 + 1]@y.pkt_{in}]\!]_{\mathcal{E}'[y \mapsto h']} \\
= & [\![x.\iota[|v| - 1{:}|v|]@y.pkt_{in}]\!]_{\mathcal{E}'[y \mapsto h']} \\
= & h(pkt_{in})[0{:}1]@h'(pkt_{in})
\end{aligned}
$$

with $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$ follows

$$
\begin{aligned}
= & h(pkt_{in}) \\
= & [\![y.pkt_{in}]\!]_{\mathcal{E}[y \mapsto h]}
\end{aligned}
$$

Case $e = |z.pkt_{in}|$**:** Case distinction on $z = y$:

Subcase $z \neq y$**:**

$$\begin{aligned}&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(|z.pkt_{in}|, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']}\\ =\ &[\![|z.pkt_{in}|]\!]_{\mathcal{E}'[y \mapsto h']}\end{aligned}$$

If $z = x$, the length of $x.pkt_{in} = 0$ in both environments and otherwise, $z.pkt_{in}$ refers to the same heap in both $\mathcal{E}$ and $\mathcal{E}'$.

$$= [\![|z.pkt_{in}|]\!]_{\mathcal{E}[y \mapsto h]}$$

Subcase $z = y$**:**

$$\begin{aligned}&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(|y.pkt_{in}|, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']}\\ =\ &[\![|y.pkt_{in}| + 1]\!]_{\mathcal{E}'[y \mapsto h']}\end{aligned}$$

with $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$ follows

$$= [\![|y.pkt_{in}|]\!]_{\mathcal{E}[y \mapsto h]}$$

Case $e = b :: bv$**:**

$$\begin{aligned}&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(b :: bv, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']}\\ =\ &[\![\mathsf{heapRef}_1(b :: \mathsf{chomp}_1^e(bv, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']}\\ =\ &[\![b :: \mathsf{heapRef}_1(\mathsf{chomp}_1^e(bv, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']}\\ =\ &[\![b]\!]_{\mathcal{E}'[y \mapsto h']} :: [\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(bv, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']}\\ =\ &[\![b]\!]_{\mathcal{E}'[y \mapsto h']} :: [\![bv]\!]_{\mathcal{E}[y \mapsto h]}\end{aligned}$$

by IH follows

$$= [\![b]\!]_{\mathcal{E}[y \mapsto h]} :: [\![bv]\!]_{\mathcal{E}[y \mapsto h]}$$

since b is either 0 or 1

$$= [\![b :: bv]\!]_{\mathcal{E}[y \mapsto h]}$$

Case $e = bv_1 @ bv_2$**:**

$$\begin{aligned}&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(bv_1 @ bv_2, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']}\\ =\ &[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(bv_1, y, \mathsf{b}_0) @ \mathsf{chomp}_1^e(bv_2, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']}\\ =\ &[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(bv_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']} @\\ &[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^e(bv_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']}\\ =\ &[\![bv_1]\!]_{\mathcal{E}[y \mapsto h]} @ [\![bv_2]\!]_{\mathcal{E}[y \mapsto h]}\end{aligned}$$

by IH

$$= [\![bv_1 @ bv_2]\!]_{\mathcal{E}[y \mapsto h]}$$

Case $e = n + m$**:** Symmetric to previous case.

□

Lemma A.37 (Semantic Chomp Formulae)**.** *For all formulae φ, heaps h and h', environments $\mathcal{E}$ and $\mathcal{E}'$ and variables x, if $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$, and $\mathcal{E}' = \mathcal{E}[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v])]$, and if $x \in dom$, $v = \mathcal{E}(x)(\iota)@h(pkt_{in})[0{:}1]$ and $\mathcal{E}(x)(pkt_{in}) = \langle\rangle$ and $\mathcal{E}(x)(pkt_{out}) = \langle\rangle$ and otherwise $v = h(pkt_{in})[0{:}1]$ and x not free in φ, then*

$$[\![\varphi]\!]_{\mathcal{E}[y\mapsto h]} = [\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}$$

Proof. By induction on φ.

Case $\varphi = e_1 = e_2$**:**

$$\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(e_1 = e_2, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota)]\!]_{\mathcal{E}'[y\mapsto h']} \\
=\ &[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(e_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota) = \\
&\quad \mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(e_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota)]\!]_{\mathcal{E}'[y\mapsto h']} \\
=\ &[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(e_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota)]\!]_{\mathcal{E}'[y\mapsto h']} = \\
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(e_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota)]\!]_{\mathcal{E}'[y\mapsto h']}
\end{aligned}$$

by Lemma A.36 follows

$$\begin{aligned}
&= [\![e_1]\!]_{\mathcal{E}[y\mapsto h]} = [\![e_2]\!]_{\mathcal{E}[y\mapsto h]} \\
&= [\![e_1 = e_2]\!]_{\mathcal{E}[y\mapsto h]}
\end{aligned}$$

Case $\varphi = \varphi_1 \wedge \varphi_2$**:**

$$\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi_1 \wedge \varphi_2, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']} \\
=\ &[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1) \wedge \\
&\quad \mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi_2, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']} \\
=\ &[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']} \wedge \\
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi_2, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}
\end{aligned}$$

by IH follows

$$\begin{aligned}
&= [\![\varphi_1]\!]_{\mathcal{E}[y\mapsto h]} \wedge [\![\varphi_2]\!]_{\mathcal{E}[y\mapsto h]} \\
&= [\![\varphi_1 \wedge \varphi_2]\!]_{\mathcal{E}[y\mapsto h]}
\end{aligned}$$

Case $\varphi = \neg\varphi_1$**:**

$$\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\neg\varphi_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h]} \\
=\ &[\![\neg\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h]} \\
=\ &\neg[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h]}
\end{aligned}$$

by IH follows

$$\begin{aligned}
&= \neg[\![\varphi]\!]_{\mathcal{E}[y\mapsto h]} \\
&= [\![\neg\varphi]\!]_{\mathcal{E}[y\mapsto h]}
\end{aligned}$$

Case $\varphi = z.\iota'.valid$**:**

$$
\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(z.\iota'.valid, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']} \\
=\ &[\![z.\iota'.valid]\!]_{\mathcal{E}'[y\mapsto h']} \\
=\ &\iota' \in dom(\mathcal{E}'[y \mapsto h'](z)) \\
=\ &\iota' \in dom(\mathcal{E}[y \mapsto h](z))
\end{aligned}
$$

by definition of $\mathcal{E}'$ and h' follows

$$
= [\![z.\iota'.valid]\!]_{\mathcal{E}[y\mapsto h]}
$$

Case $\varphi = \mathsf{true}$**:**

$$
\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\mathsf{true}, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h]} \\
=\ &[\![\mathsf{true}]\!]_{\mathcal{E}'[y\mapsto h]} \\
=\ &\mathsf{true} \\
=\ &[\![\mathsf{true}]\!]_{\mathcal{E}[y\mapsto h]}
\end{aligned}
$$

Case $\varphi = \mathsf{false}$**:** Symmetric to previous case.

□

Lemma A.38 (Semantic Chomp Refinement)**.** *For all heap types τ, heaps h and h', environments $\mathcal{E}$ and $\mathcal{E}'$ and variables x, if $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$, and $\mathcal{E}' = \mathcal{E}[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v])]$, and, if $x \in dom$, $v = \mathcal{E}(x)(\iota)@h(pkt_{in})[0:1]$ and $\mathcal{E}(x)(pkt_{in}) = \langle\rangle$ and $\mathcal{E}(x)(pkt_{out}) = \langle\rangle$, and otherwise $v = h(pkt_{in})[0{:}1]$ and x not free in τ, then*

$$
[\![\tau]\!]_{\mathcal{E}[y\mapsto h]} = [\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}
$$

Proof. Proof by induction on τ.

Case $\tau = \varnothing$**:**

$$
\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\varnothing, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']} \\
=\ &[\![\varnothing]\!]_{\mathcal{E}'[y\mapsto h']} \\
=\ &\{\} \\
=\ &[\![\tau]\!]_{\mathcal{E}[y\mapsto h]}
\end{aligned}
$$

Case $\tau = \top$**:**

$$
\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\top, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']} \\
=\ &[\![\top]\!]_{\mathcal{E}'[y\mapsto h']} \\
=\ &H \\
=\ &[\![\top]\!]_{\mathcal{E}}
\end{aligned}
$$

Case $\tau = \Sigma z\colon \tau_1.\tau_2$**:**

$$
\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\Sigma z\colon \tau_1.\tau_2, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &[\![\Sigma z\colon \mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_1, x, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1).\\
&\qquad \mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_2, x, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &\{h_1' \mathbin{++} h_2' \mid h_1' \in [\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_1, x, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']} \wedge\\
&\qquad h_2' \in [\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_2, x, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h', z\mapsto h_1']}\}\\
=\ &\{h_1' \mathbin{++} h_2' \mid h_1' \in [\![\tau_1]\!]_{\mathcal{E}[y\mapsto h]} \wedge h_2' \in [\![\tau_2]\!]_{\mathcal{E}[y\mapsto h, z\mapsto h_1']}\}\\
=\ &[\![\Sigma z\colon \tau_1.\tau_2]\!]_{\mathcal{E}[y\mapsto h]}
\end{aligned}
$$

Case $\tau = \tau_1 + \tau_2$**:**

$$
\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_1 + \tau_2, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &[\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_1, x, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1) +\\
&\quad \mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_2, x, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &[\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_1, x, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']} \cup\\
&\quad [\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_2, x, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &[\![\tau_1]\!]_{\mathcal{E}[y\mapsto h]} \cup [\![\tau_1]\!]_{\mathcal{E}[y\mapsto h]}\\
=\ &[\![\tau_1 + \tau_2]\!]_{\mathcal{E}[y\mapsto h]}
\end{aligned}
$$

Case $\tau = \{z : \tau_1 \mid \varphi\}$**:**

$$
\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\{z : \tau_1 \mid \varphi\}, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &[\![\{z : \mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1) \mid\\
&\qquad \mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)\}]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &\{h \mid h \in [\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']} \wedge\\
&\qquad [\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']} = \mathsf{true}\}\\
=\ &\{h \mid h \in [\![\tau_1]\!]_{\mathcal{E}[y\mapsto h]} \wedge [\![\varphi]\!]_{\mathcal{E}[y\mapsto h]} = \mathsf{true}\}\\
=\ &[\![\{z : \tau_1 \mid \varphi\}]\!]_{\mathcal{E}[y\mapsto h]}
\end{aligned}
$$

Case $\tau = \tau_1[z \mapsto \tau_2]$**:**

$$
\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_1[z \mapsto \tau_2], y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &[\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)[z \mapsto\\
&\qquad \mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_2, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]]\!]_{\mathcal{E}'[y\mapsto h']}\\
=\ &\{h \mid h_2 \in [\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_2, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y\mapsto h']} \wedge\\
&\qquad h \in [\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_1, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1]\!]_{\mathcal{E}'[y\mapsto h', z\mapsto h_2]}\}\\
=\ &\{h \mid h_2 \in [\![\tau_2]\!]_{\mathcal{E}[y\mapsto h]} \wedge h \in [\![\tau_1]\!]_{\mathcal{E}[y\mapsto h, z\mapsto h_2]}\}\\
=\ &[\![\tau_1[z \mapsto \tau_2]]\!]_{\mathcal{E}[y\mapsto h]}
\end{aligned}
$$

□

Lemma A.39 (Semantic Chomp$_1$). *For all heap types τ, environments $\mathcal{E}$ and $\mathcal{E}'$, and variables x, if $\mathcal{E}' = \mathcal{E}[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v])]$, and if $x \in dom(\mathcal{E})$, $v = \mathcal{E}(x)(\iota)@h(pkt_{in})[0:1]$ and $\mathcal{E}(x)(pkt_{in}) = \mathcal{E}(x)(pkt_{out}) = \langle\rangle$, otherwise $v = h(pkt_{in})[0:1]$, then $\forall h \in [\![\tau]\!]_{\mathcal{E}}.|h(pkt_{in})| \geq 1 \implies \exists h' \in [\![\mathsf{chompRec}(\tau, 1, \iota, x)]\!]_{\mathcal{E}'}.h' = \mathsf{chomp}^{\Downarrow}(h, 1)$*

Proof.

$$\begin{aligned}
&\forall h \in [\![\tau]\!]_{\mathcal{E}}.|h(pkt_{in})| \geq 1 \implies \exists h' \in [\![\mathsf{chompRec}(\tau, 1, \iota, x)]\!]_{\mathcal{E}'}.h' = \mathsf{chomp}^{\Downarrow}(h, 1)\\
\Leftrightarrow\ &\text{(By definition of chompRec)}\\
&\forall h \in [\![\tau]\!]_{\mathcal{E}}.|h(pkt_{in})| \geq 1 \implies\\
&\qquad \exists h' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}.h' = \mathsf{chomp}^{\Downarrow}(h, 1)
\end{aligned}$$

Proof by induction on τ.

Case $\tau = \varnothing$**:** $[\![\varnothing]\!]_{\mathcal{E}} = \{\}$. As there are no heaps in the semantics, the case holds.

Case $\tau = \top$**:** Let h be some heap from $[\![\top]\!]_{\mathcal{E}} = H$. Let $h' = h$ except that $h'(pkt_{in}) = h(pkt_{in})[1:]$.

By definition of chomp_1 and $\mathsf{heapRef}_1$, $\mathsf{heapRef}_1(\mathsf{chomp}_1(\top, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1) = \top$ and $[\![\top]\!]_{\mathcal{E}'} = H$.

We can conclude that $h' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\top, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}$ and $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$ follows by construction of $h'(pkt_{in}) = h(pkt_{in})[1:]$.

Case $\tau = \Sigma y\colon \tau_1.\tau_2$**:** Let h be some heap from $[\![\Sigma y\colon \tau_1.\tau_2]\!]_{\mathcal{E}}$. We know $h = h1 \mathbin{++} h2$, for some $h_1 \in [\![\tau_1]\!]_{\mathcal{E}}$ and some $h_2 \in [\![\tau_2]\!]_{\mathcal{E}[y \mapsto h1]}$.

We have to show that there exists some

$$h' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\Sigma y\colon \tau_1.\tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}$$

such that $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$.

We deconstruct $[\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}$:

$$\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\Sigma y\colon \tau_1.\tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}\\
=\ &[\![\mathsf{heapRef}_1(\Sigma y\colon \mathsf{chomp}_1(\tau_1, \mathsf{b}_0).\mathsf{chompRef}_1(\tau_2, y, \mathsf{b}_0) +\\
&\quad \Sigma y\colon \{z : \tau_1 \mid |z.pkt_{in}| = 0\}.\mathsf{chomp}_1(\tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}\\
=\ &[\![\Sigma y\colon \mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1).\\
&\qquad \mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_2, x, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1) +\\
&\quad \Sigma y\colon \mathsf{heapRef}_1(\{z : \tau_1 \mid |z.pkt_{in}| = 0\}, \mathsf{b}_0, x, \iota, 1).\\
&\qquad \mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}\\
=\ &[\![\Sigma y\colon \mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1).\\
&\qquad \mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_2, x, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \cup\\
&[\![\Sigma y\colon \{z : \tau_1 \mid |z.pkt_{in}| = 0\}.\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}\\
=\ &\{h'_1 \mathbin{++} h'_2 | h'_1 \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \wedge\\
&\qquad h'_2 \in [\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_2, x, \mathsf{b}_0))]\!]_{\mathcal{E}'[y \mapsto h'_1]}\} \cup\\
&\{h'_1 \mathbin{++} h'_2 | h'_1 \in [\![\{z : \tau_1 \mid |z.pkt_{in}| = 0\}]\!]_{\mathcal{E}'} \wedge\\
&\qquad h'_2 \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h'_1]}\}
\end{aligned}$$

By case distinction on the length of pkt_{in} in h_1.

Subcase $|h_1(pkt_{in})| = 0$**:**

By definition of $\mathcal{E}'$, $[\![\tau_1]\!]_{\mathcal{E}} = [\![\tau_1]\!]_{\mathcal{E}'}$, because, τ_1 can't contain a reference to the newly added bit in $\mathcal{E}'(x)(\iota)$, from which follows that $h_1 \in [\![\tau_1]\!]_{\mathcal{E}'}$.
By semantics of heap types, $h_1 \in [\![\{z : \tau_1 \mid |z.pkt_{in}| = 0\}]\!]_{\mathcal{E}'}$.
By IH there exists

$$h_2' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h_1]}$$

such that $h_2' = \mathsf{chomp}^{\Downarrow}(h_2, 1)$.
Let $h' = h_1 \mathbin{++} h_2'$.
We conclude that

$$\begin{aligned} h' \in \{h_1' \mathbin{++} h_2' | h_1' \in [\![\{z : \tau_1 \mid |z.pkt_{in}| = 0\}]\!]_{\mathcal{E}'} \wedge \\ h_2' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h_1']}\} \end{aligned}$$

and thus $h' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\Sigma y\colon \tau_1.\tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}$.
By assumption $|h_1(pkt_{in})| = 0$, we can conclude that

$$h_1 \mathbin{++} \mathsf{chomp}^{\Downarrow}(h_2, 1) = \mathsf{chomp}^{\Downarrow}(h_1 \mathbin{++} h_2, 1)$$

thus

$$\begin{aligned} h' &= h_1 \mathbin{++} h_2' \\ &= h_1 \mathbin{++} \mathsf{chomp}^{\Downarrow}(h_2, 1) \\ &= \mathsf{chomp}^{\Downarrow}(h_1 \mathbin{++} h_2, 1) \\ &= \mathsf{chomp}^{\Downarrow}(h, 1) \end{aligned}$$

Subcase $|h_1(pkt_{in})| \neq 0$**:**

By IH there exists $h_1' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}$, such that $h_1' = \mathsf{chomp}^{\Downarrow}(h_1, 1)$.
By Lemma A.38 follows that

$$h_2 \in [\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_2, x, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h_1']}$$

Let $h' = h_1' \mathbin{++} h_2$.
We conclude that

$$\begin{aligned} h' \in \{h_1' \mathbin{++} h_2 | h_1' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \wedge \\ h_2 \in [\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_2, x, \mathsf{b}_0))]\!]_{\mathcal{E}'[y \mapsto h_1']}\} \end{aligned}$$

and thus $h' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\Sigma y\colon \tau_1.\tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}$.
With

$$\begin{aligned} h' &= h_1' \mathbin{++} h_2 \\ &= \mathsf{chomp}^{\Downarrow}(h_1, 1) \mathbin{++} h_2 \\ &= \mathsf{chomp}^{\Downarrow}(h_1 \mathbin{++} h_2, 1) \\ &= \mathsf{chomp}^{\Downarrow}(h, 1) \end{aligned}$$

we can conclude this case.

Case $\tau = \tau_1 + \tau_2$**:**

Let h be some heap from $[\![\tau_1 + \tau_2]\!]_{\mathcal{E}}$. By the semantics of heap types, we know $h \in [\![\tau_1]\!]_{\mathcal{E}}$ or $h \in [\![\tau_2]\!]_{\mathcal{E}}$.

Subcase $h \in [\![\tau_1]\!]_{\mathcal{E}}$**:**

By IH we know that there exists

$$h' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}$$

such that $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$.

By set theory and

$$\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1 + \tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
=\ &[\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0) + \mathsf{chomp}_1(\tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
=\ &[\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1) + \\
&\quad \mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
=\ &[\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \cup \\
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}
\end{aligned}$$

we conclude $h' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1 + \tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}$.

Subcase $h \in [\![\tau_2]\!]_{\mathcal{E}}$**:** Symmetric to previous subcase.

Case $\tau = \{y : \tau_1 \mid \varphi\}$**:**

Let h be some heap from $[\![\{y : \tau_1 \mid \varphi\}]\!]_{\mathcal{E}}$.

By the semantics of heap types, we know that $h \in [\![\tau_1]\!]_{\mathcal{E}}$ and $[\![\varphi]\!]_{\mathcal{E}[y \mapsto h]} = \mathsf{true}$. By induction hypothesis there exists $h' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}$ such that $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$.

By Lemma A.37 we know that

$$[\![\varphi]\!]_{\mathcal{E}[y \mapsto h]} = [\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']}$$

To apply Lemma A.37, we must show that x is not free in φ, if $x \notin dom(\mathcal{E})$. If this does not hold, there is no $h \in [\![\tau]\!]_{\mathcal{E}}$, which violates our initial assumption. With

$$\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\{y : \tau_1 \mid \varphi\}, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
=\ &[\![\mathsf{heapRef}_1(\{y : \mathsf{chomp}_1(\tau_1, \mathsf{b}_0) \mid \mathsf{chomp}_1^{\varphi}(\varphi, y, \mathsf{b}_0)\}, \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
=\ &[\![\{y : \mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1) \mid \\
&\qquad \mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)\}]\!]_{\mathcal{E}'} \\
=\ &\{h \mid h \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \wedge \\
&\qquad [\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h]}\}
\end{aligned}$$

and with our assumptions from Lemma A.37 and the induction hypothesis, we conclude that $h' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\{y : \tau_1 \mid \varphi\}, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}$ such that $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$.

Case $\tau = \tau_1[y \mapsto \tau_2]$**:**

Let h be some heap from $[\![\tau_1[y \mapsto \tau_2]]\!]_{\mathcal{E}}$. We know that $h \in [\![\tau_1]\!]_{\mathcal{E}[y \mapsto h_2]}$ for some $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$.

By IH there exists some $h' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h_2]}$ such that $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$.

To conclude this case, we must show that

$$h' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1[y \mapsto \tau_2], \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}$$

From $\mathsf{heapRef}_1$, chomp_1 and the semantics of heap types, we get:

$$\begin{aligned}
&[\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1[y \mapsto \tau_2], \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
=\ &[\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0)[y \mapsto \tau_2], \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
=\ &[\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)[y \mapsto \tau_2]]\!]_{\mathcal{E}'} \\
=\ &\{h_{11} | h_{22} \in [\![\tau_2]\!]_{\mathcal{E}'} \wedge h_{11} \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h_{22}]}\}
\end{aligned}$$

With $h_{11} = h'$ and $h_{22} = h_2$, we can conclude that

$$h' \in [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1[y \mapsto \tau_2], \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'}$$

To argue that $h_2 \in [\![\tau_2]\!]_{\mathcal{E}'}$, we make a case distinction on $x \in dom(\mathcal{E})$.

Subcase $x \notin dom(\mathcal{E})$**:** x cannot appear free in τ and thereby also not in τ_2, otherwise there would be no $h \in [\![\tau]\!]_{\mathcal{E}}$, thus for all $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$, $h_2 \in [\![\tau_2]\!]_{\mathcal{E}'}$.

Subcase $x \in dom(\mathcal{E})$**:** By assumption, there is some $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$. By semantics of heap types, all formulae in τ_2 referencing x, evaluate to true, i.e., they only refer to information contained in $\mathcal{E}(x)$. By assumption, $\mathcal{E}(x)(pkt_{in}) = \mathcal{E}'(x)(pkt_{in})$, $\mathcal{E}(x)(pkt_{out}) = \mathcal{E}'(x)(pkt_{out})$, and $\mathcal{E}(x)(\iota)@h(pkt_{in})[0 : 1] = \mathcal{E}'(x)(\iota)$. Since all information of $\mathcal{E}(x)$ is preserved in $\mathcal{E}'$, $h_2 \in [\![\tau_2]\!]_{\mathcal{E}'}$.

□

Lemma A.40 (ChompRec Unroll)**.** *For all instances ι, if $m + 1 \leq \mathsf{sizeof}(\iota)$, then*

$$\mathsf{chompRec}(\mathsf{chompRec}(\tau, m, x, \iota), 1, x, \iota) = \mathsf{chompRec}(\tau, m + 1, x, \iota)$$

Proof. By induction on n.

Case $n = 0$**:**

$$\begin{aligned}
&\mathsf{chompRec}(\mathsf{chompRec}(\tau, 0, x, \iota), 1, x, \iota) \\
=\ &\mathsf{chompRec}(\tau, 1, x, \iota)
\end{aligned}$$

Case $n = 1$**:**

$$\begin{aligned}
&\mathsf{chompRec}(\mathsf{chompRec}(\tau, 1, x, \iota), 1, x, \iota) \\
=\ &\mathsf{chompRec}(\mathsf{chompRec}(\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau, \mathsf{b}_0), \mathsf{b}_0, \iota, 1), 0, x, \iota), 1, x, \iota) \\
=\ &\mathsf{chompRec}(\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau, \mathsf{b}_0), \mathsf{b}_0, \iota, 1), 1, x, \iota) \\
=\ &\mathsf{chompRec}(\tau, 2, x, \iota)
\end{aligned}$$

Case $n = m$**:**

We assume that the lemma holds for $n = m$. We now have to show that the lemma also holds for $n = m + 1$.

$$\begin{aligned}
&\mathsf{chompRec}(\mathsf{chompRec}(\tau, m+1, x, \iota), 1, x, \iota) \\
=\ &\mathsf{chompRec}(\mathsf{chompRec}(\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau, \mathsf{b}_0), \mathsf{b}_0, \iota, m+1), m, x, \iota), \\
&\qquad 1, x, \iota) \\
=\ &\mathsf{chompRec}(\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau, \mathsf{b}_0), \mathsf{b}_0, \iota, m+1), m+1, x, \iota) \\
=\ &\mathsf{chompRec}(\tau, m+2, x, \iota)
\end{aligned}$$

□

Lemma A.41 (Semantic Chomp Unroll)**.** *For all heaps h and all $n \in \mathbb{N}$, if $|h(pkt_{in})| \geq n + 1$, then* $\mathsf{chomp}^{\Downarrow}(\mathsf{chomp}^{\Downarrow}(h, n), 1) = \mathsf{chomp}^{\Downarrow}(h, n + 1)$

Proof. By definition of $\mathsf{chomp}^{\Downarrow}$,

$$\begin{aligned}
&\mathsf{chomp}^{\Downarrow}(\mathsf{chomp}^{\Downarrow}(h, n), 1) \\
=\ &h[pkt_{in} \mapsto h(pkt_{in})[n:]][pkt_{in} \mapsto h[pkt_{in} \mapsto h(pkt_{in})[n:]](pkt_{in})[1:]] \\
=\ &h[pkt_{in} \mapsto h[pkt_{in} \mapsto h(pkt_{in})[n:]](pkt_{in})[1:]] \\
=\ &h[pkt_{in} \mapsto h(pkt_{in})[n+1:]] \\
=\ &\mathsf{chomp}^{\Downarrow}(h, n+1)
\end{aligned}$$

□

Lemma A.42 (Chomp Slice)**.** *For all heaps h and all $n \in \mathbb{N}$, if $|h(pkt_{in})| \geq n + 1$, then*

$$\mathsf{chomp}^{\Downarrow}(h, n)(pkt_{in})[0:1] = h(pkt_{in})[n:n+1]$$

Proof. By definition of $\mathsf{chomp}^{\Downarrow}$,

$$\mathsf{chomp}^{\Downarrow}(h, n)(pkt_{in})[0:1] = h[pkt_{in} \mapsto h(pkt_{in})[n:]](pkt_{in})[0:1]$$

Let $bv = h(pkt_{in}) = \langle b_0, ..., b_n, ..., b_m \rangle$.

Let bv' be the bit vector we obtain after removing the first n bits from bv', $bv' = \langle b_n, ..., b_m \rangle$.

Accessing the first bit of bv' gives us bit b_n, which is also the n-th bit in bv, i.e., $bv[n:n+1]$. □

Lemma A.43 (Semantic ChompRec)**.** *For all heap types τ, environments $\mathcal{E}$ and $\mathcal{E}'$, variables x and $n \in \mathbb{N}$, if x does not appear free in τ, and $\mathcal{E}' = \mathcal{E}[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto h(pkt_{in})[0:n]])]$, then*

$$\forall h \in [\![\tau]\!]_{\mathcal{E}}.|h(pkt_{in})| \geq n \implies \exists h' \in [\![\mathsf{chompRec}(\tau, n, \iota, x)]\!]_{\mathcal{E}'}.h' = \mathsf{chomp}^{\Downarrow}(h, n).$$

Proof. Proof by induction on n.

Case $n = 0$**:**

$$\begin{aligned}
&\forall h \in [\![\tau]\!]_{\mathcal{E}}.|h(pkt_{in})| \geq 0 \implies \\
&\qquad \exists h' \in [\![\mathsf{chompRec}(\tau, 0, \iota, x)]\!]_{\mathcal{E}'}.h' = \mathsf{chomp}^{\Downarrow}(h, 0) \\
\Leftrightarrow\ &\forall h \in [\![\tau]\!]_{\mathcal{E}}.\exists h' \in [\![\tau]\!]_{\mathcal{E}'}.h' = h
\end{aligned}$$

Let $h' = h$, i.e., we have to show that $h \in [\![\tau]\!]_{\mathcal{E}'}$. By assumption, x is not free in τ, i.e., the binding of x in $\mathcal{E}'$ has no effect on the semantics of τ. Since $\mathcal{E}$ and $\mathcal{E}'$ are otherwise identical, τ evaluated in both environments is described by the same set of heaps, from which we can conclude that $h \in [\![\tau]\!]_{\mathcal{E}'}$.

Case $n = 1$**:**

$$\begin{aligned}
&\forall h \in [\![\tau]\!]_{\mathcal{E}}.|h(pkt_{in})| \geq 1 \implies \\
&\qquad \exists h' \in [\![\mathsf{chompRec}(\tau, 1, \iota, x)]\!]_{\mathcal{E}'}.h' = \mathsf{chomp}^{\Downarrow}(h, 1)
\end{aligned}$$

The result directly follows by Lemma A.39.

Case $n = m + 1$**:**

We assume that the lemma holds for $n = m$, i.e.,

$$\begin{aligned}
&\forall h \in [\![\tau]\!]_{\mathcal{E}}.|h(pkt_{in})| \geq m \implies \\
&\qquad \exists h' \in [\![\mathsf{chompRec}(\tau, m, \iota, x)]\!]_{\mathcal{E}'}.h' = \mathsf{chomp}^{\Downarrow}(h, m)
\end{aligned}$$

We have to show that the lemma also holds for $n = m + 1$, i.e.,

$$\begin{aligned}
&\forall h \in [\![\tau]\!]_{\mathcal{E}}.|h(pkt_{in})| \geq m + 1 \implies \\
&\qquad \exists h' \in [\![\mathsf{chompRec}(\tau, m + 1, \iota, x)]\!]_{\mathcal{E}'}.h' = \mathsf{chomp}^{\Downarrow}(h, m + 1)
\end{aligned}$$

where $\mathcal{E}' = \mathcal{E}[x \mapsto [\iota \mapsto h(pkt_{in})[0{:}m + 1], pkt_{in} \mapsto \langle\rangle, pkt_{out} \mapsto \langle\rangle]]$.

Let h be some heap $h \in [\![\tau]\!]_{\mathcal{E}_0}$.

By induction hypothesis, there exists some $h' \in [\![\mathsf{chompRec}(\tau, m, \iota, x)]\!]_{\mathcal{E}'_0}$, such that $h' = \mathsf{chomp}^{\Downarrow}(h, m)$.

Let $\mathcal{E}_1 = \mathcal{E}'_0 = \mathcal{E}_0[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto h(pkt_{in})[0{:}m]])]$. We use (A) to refer to this assumption.

By Lemma A.39, for all $h_1 \in [\![\mathsf{chompRec}(\tau, m, \iota, x)]\!]_{\mathcal{E}_1}$, there exists some heap h'_1 such that $h'_1 \in [\![\mathsf{chompRec}(\mathsf{chompRec}(\tau, m, \iota, x), 1, \iota, x)]\!]_{\mathcal{E}'_1}$ and $\mathcal{E}'_1 = \mathcal{E}_1[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto \mathcal{E}_1(x)(\iota)@h_1(pkt_{in})[0{:}1]])]$, and $h'_1 = \mathsf{chomp}^{\Downarrow}(h_1, 1)$.

Since, by assumption, $h' \in [\![\mathsf{chompRec}(\tau, m, \iota, x)]\!]_{\mathcal{E}'_0}$ and also $\mathcal{E}_1 = \mathcal{E}'_0$, we can define h_1 to be equal to h', i.e., $h'_1 = \mathsf{chomp}^{\Downarrow}(h', 1)$.

From $h' = \mathsf{chomp}^{\Downarrow}(h, m)$ follows $h'_1 = \mathsf{chomp}^{\Downarrow}(\mathsf{chomp}^{\Downarrow}(h, m), 1)$.

From Lemma A.41 also follows that $h'_1 = \mathsf{chomp}^{\Downarrow}(h, m + 1)$.

We must show that $h'_1 \in [\![\mathsf{chompRec}(\tau, m + 1, \iota, x)]\!]_{\mathcal{E}'}$.

We know that

$$h'_1 \in [\![\mathsf{chompRec}(\mathsf{chompRec}(\tau, m, \iota, x), 1, \iota, x)]\!]_{\mathcal{E}'_1}$$

and by Lemma A.40, $h_1' \in [\![\mathsf{chompRec}(\tau, m+1, \iota, x)]\!]_{\mathcal{E}_1'}$, so we must show that $\mathcal{E}' = \mathcal{E}_1'$.

By assumption, $\mathcal{E}_1' = \mathcal{E}_1[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto \mathcal{E}_1(x)(\iota)@h_1(pkt_{in})[0:1]])]$, where $h_1 = \mathsf{chomp}^{\Downarrow}(h, m)$ (by IH).

Also by assumption, $\mathcal{E}(x)(\iota) = h(pkt_{in})[0 : m]$, i.e., $\mathcal{E}_1' = \mathcal{E}[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto h(pkt_{in})[0:m]@h_1(pkt_{in})[0:1]])]$.

Again, substituting h_1 with h', and by $h' = \mathsf{chomp}^{\Downarrow}(h, m)$, we obtain $\mathcal{E}_1' = \mathcal{E}[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto h(pkt_{in})[0:m]@\mathsf{chomp}^{\Downarrow}(h, m)(pkt_{in})[0:1]])]$.

By Lemma A.42 and by definition of bit vector concatenation, $\mathcal{E}_1' = \mathcal{E}[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto h(pkt_{in})[0:m+1]])] = \mathcal{E}'$.

□

Lemma A.44 (Semantic Chomp)**.** *If x does not appear free in τ, then forall heaps $h \in [\![\tau]\!]_{\mathcal{E}}$ where $|h(pkt_{in})| \geq \mathsf{sizeof}(\iota)$, there exists $h' \in [\![\mathsf{chomp}(\tau, \iota, x)]\!]_{\mathcal{E}'}$ such that $h' = \mathsf{chomp}^{\Downarrow}(h, \mathsf{sizeof}(\iota))$ where $\mathcal{E}' = \mathcal{E}[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto h(pkt_{in})[0 : \mathsf{sizeof}(\iota)]])]$.*

Proof. By definition of chomp, we know that

$$\mathsf{chomp}(\tau, \iota, x) = \mathsf{chompRec}(\tau, \mathsf{sizeof}(\iota), x, \iota)$$

The result follows from Lemma A.43. □

Lemma A.45 (Semantic Chomp$_1$ Inverse)**.** *For all x, v, τ, $\mathcal{E}'$ and h' such that $\mathcal{E}'(x) = (\langle\rangle, \langle\rangle, [\iota \mapsto v])$ and $h' \in [\![\mathsf{chompRec}(\tau, 1, x, \iota)]\!]_{\mathcal{E}'}$ and x not free in τ and $\mathsf{sizeof}(v) \geq 1$, there exists h and $\mathcal{E}$ such that,*

(1) $h \in [\![\tau]\!]_{\mathcal{E}}$ and

(2) $h' = \mathsf{chomp}^{\Downarrow}(h, 1)$ and

and

(3) $\mathcal{E} = \mathcal{E}' \setminus x$ and

(4) $v = h(pkt_{in})[0 : 1]$

or

(3) $x \in dom(\mathcal{E})$ and

(4) $v = \mathcal{E}(x)(\iota)@h(pkt_{in})[0:1]$ and

(5) $\mathcal{E} = \mathcal{E}'[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v[0 : \mathsf{sizeof}(v) - 1]])]$

Proof. We refer to the general assumptions as follows:

(A) $\mathcal{E}'(x) = (\langle\rangle, \langle\rangle, [\iota \mapsto v])$

(B) $h' \in [\![\mathsf{chompRec}(\tau, 1, x, \iota)]\!]_{\mathcal{E}'}$ and

(C) x not free in τ and

(D) $\mathsf{sizeof}(v) \geq 1$

Proof by induction on τ. By definition of chompRec follows that

$$[\![\mathsf{chompRec}(\tau, 1, x, \iota)]\!]_{\mathcal{E}} = [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}}$$

Case $\tau = \varnothing$**:**

$$\begin{aligned}
& [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\varnothing, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
= & [\![\mathsf{heapRef}_1(\varnothing, \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
= & [\![\varnothing]\!]_{\mathcal{E}'} \\
= & \{\}
\end{aligned}$$

As there is no $h' \in [\![\mathsf{chompRec}(\tau, 1, \iota, x)]\!]_{\mathcal{E}'}$, this case is immediate.

Case $\tau = \top$**:**

$$\begin{aligned}
& [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\top, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
= & [\![\mathsf{heapRef}_1(\top, \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
= & [\![\top]\!]_{\mathcal{E}'} \\
= & \mathcal{H}
\end{aligned}$$

Let $\mathcal{E}'$ where $\mathcal{E}'(x) = (\langle\rangle, \langle\rangle, [\iota \mapsto v])$ be arbitrary.

Let $h' \in [\![\mathsf{chompRec}(\top, 1, \iota, x)]\!]_{\mathcal{E}'} = \mathcal{H}$ be arbitrary. We have to distinguish two cases.

Subcase $\mathsf{sizeof}(v) = 1$**:**
Let $\mathcal{E} = \mathcal{E}' \setminus x$ and let $h = h'[pkt_{in} \mapsto v@h'(pkt_{in})]$, i.e., $h(pkt_{in})[0{:}1] = v$. (1) follows by the semantics of heap types. (2) follows by the definition of $\mathsf{chomp}^{\Downarrow}$. (3) and (4) immediately follow from the definition of h and $\mathcal{E}$.

Subcase $\mathsf{sizeof}(v) > 1$**:**
Let $\mathcal{E} = \mathcal{E}'[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v[0 : \mathsf{sizeof}(v) - 1]])]$ and let $h = h'[pkt_{in} \mapsto v[\mathsf{sizeof}(v) - 1 : \mathsf{sizeof}(v)]@h'(pkt_{in})]$. (1) follows by the semantics of heap types. (2) follows by the definition of $\mathsf{chomp}^{\Downarrow}$ (3) and (4) immediately follow from the definition of h and $\mathcal{E}$

Case $\tau = \tau_1 + \tau_2$**:** Let $\mathcal{E}'$ where $\mathcal{E}'(x) = (\langle\rangle, \langle\rangle, [\iota \mapsto v])$ be arbitrary.

Let $h' \in [\![\mathsf{chompRec}(\tau_1 + \tau_2, 1, \iota, x)]\!]_{\mathcal{E}'}$ be arbitrary.

$$\begin{aligned}
& [\![\mathsf{chompRec}(\tau_1 + \tau_2, 1, \iota, x)]\!]_{\mathcal{E}'} \\
= & [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1 + \tau_2, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
= & [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \cup \\
& [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
= & [\![\mathsf{chompRec}(\tau_1, 1, x\iota)]\!]_{\mathcal{E}'} \cup [\![\mathsf{chompRec}(\tau_2, 1, x\iota)]\!]_{\mathcal{E}'}
\end{aligned}$$

We have to distinguish two cases,

1. $h' \in [\![\mathsf{chompRec}(\tau_1, 1, x, \iota)]\!]_{\mathcal{E}'}$ and
2. $h' \in [\![\mathsf{chompRec}(\tau_2, 1, x\iota)]\!]_{\mathcal{E}'}$.

Subcase $h' \in [\![\mathsf{chompRec}(\tau_1, 1, x\iota)]\!]_{\mathcal{E}'}$**:** We further distinguish between the size of v.

Subcase $\mathsf{sizeof}(v) = 1$**:**

By IH, there exists h_1, $\mathcal{E}_1$, such that

(A1) $h_1 \in [\![\tau_1]\!]_{\mathcal{E}_1}$

(A2) $h' = \mathsf{chomp}^{\Downarrow}(h_1, 1)$

(A3) $\mathcal{E}_1 = \mathcal{E}' \setminus x$

(A4) $v = h_1(pkt_{in})[0{:}1]$

Let $h = h_1$ and $\mathcal{E} = \mathcal{E}_1$. $h \in [\![\tau_1 + \tau_2]\!]_{\mathcal{E}}$ follows from the semantics of heap types and by (A1). The rest is immediate.

Subcase $\mathsf{sizeof}(v) > 1$**:** By IH, there exists h_1, $\mathcal{E}_1$, such that

(A1) $h_1 \in [\![\tau_1]\!]_{\mathcal{E}_1}$

(A2) $h' = \mathsf{chomp}^{\Downarrow}(h_1, 1)$

(A3) $\mathcal{E}_1 = \mathcal{E}'[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v[0{:}\mathsf{sizeof}(v) - 1]])]$

(A4) $v = \mathcal{E}(x)(\iota)@h_1(pkt_{in})[0{:}1]$

Let $h = h_1$ and $\mathcal{E} = \mathcal{E}_1$. $h \in [\![\tau_1 + \tau_2]\!]_{\mathcal{E}}$ follows from the semantics of heap types and by (A1). The rest is immediate.

Subcase $h' \in [\![\mathsf{chompRec}(\tau_2, 1, x, \iota)]\!]_{\mathcal{E}'}$**:**

Symmetric to previous subcase.

Case $\tau = \Sigma y\colon \tau_1.\tau_2$**:** Let $\mathcal{E}'$ where $\mathcal{E}'(x) = (\langle\rangle, \langle\rangle, [\iota \mapsto v])$ be arbitrary. Let $h' \in [\![\mathsf{chompRec}(\Sigma y\colon \tau_1.\tau_2, 1, \iota, x)]\!]_{\mathcal{E}'}$ be arbitrary.

$$
\begin{aligned}
&[\![\mathsf{chompRec}(\Sigma y\colon \tau_1.\tau_2, 1, \iota, x)]\!]_{\mathcal{E}'} \\
= &\{h'_1 \mathbin{++} h'_2 | h'_1 \in [\![\mathsf{chompRec}(\tau_1, 1, \iota, x)]\!]_{\mathcal{E}'} \wedge \\
&\qquad h'_2 \in [\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_2, x, \mathsf{b}_0))]\!]_{\mathcal{E}'[y \mapsto h'_1]}\} \cup \\
&\{h'_1 \mathbin{++} h'_2 | h'_1 \in [\![\{z : \tau_1 \mid |z.pkt_{in}| = 0\}]\!]_{\mathcal{E}'} \wedge \\
&\qquad h'_2 \in [\![\mathsf{chompRec}(\tau_2, 1, \iota, x)]\!]_{\mathcal{E}'[y \mapsto h'_1]}\}
\end{aligned}
$$

Case distinction on the membership of h'.

Subcase h' *contained in the first subset***:**

(A1) $h' = h'_1 \mathbin{++} h'_2$ and

(A2) $h'_1 \in [\![\mathsf{chompRec}(\tau_1, 1, \iota, x)]\!]_{\mathcal{E}'}$ and

(A3) $h'_2 \in [\![\mathsf{heapRef}_1(\mathsf{chompRef}_1(\tau_2, x, \mathsf{b}_0))]\!]_{\mathcal{E}'[y \mapsto h'_1]}$.

We distinguish two additional cases.

Subcase $\mathsf{sizeof}(v) = 1$**:** By IH, there exists h_1, $\mathcal{E}_1$, such that

(A4) $h_1 \in [\![\tau_1]\!]_{\mathcal{E}_1}$

(A5) $h'_1 = \mathsf{chomp}^{\Downarrow}(h_1, 1)$

(A6) $\mathcal{E}_1 = \mathcal{E}' \setminus x$

(A7) $v = h_1(pkt_{in})[0{:}1]$

By Lemma A.38, $h'_2 \in [\![\tau_2]\!]_{\mathcal{E}'[y \mapsto h_1]}$. Since $h'_1 \mathbin{++} h'_2$ is defined, i.e., they have disjoint sets of headers and $\mathsf{chomp}^{\Downarrow}$ does not affect the validity of headers, $h_1 \mathbin{++} h'_2$ is defined.

Let $h = h_1 \mathbin{++} h_2'$ and $\mathcal{E} = \mathcal{E}_1$.
(1) follows by (A4) and $h_2' \in [\![\tau_2]\!]_{\mathcal{E}_1[y \mapsto h_1]}$. The latter holds, because x is not free in τ_2 by assumption.
To show (2), we must show that $h_1' \mathbin{++} h_2' = \mathsf{chomp}^{\Downarrow}(h_1 \mathbin{++} h_2', 1) \Leftrightarrow \mathsf{chomp}^{\Downarrow}(h_1, 1) \mathbin{++} h_2' = \mathsf{chomp}^{\Downarrow}(h_1 \mathbin{++} h_2', 1)$. This equality holds, because chomping of one bit from the input packet of h_1 and then concatenating h_2' yields the same heap as concatenating both heaps and then removing the first bit of the input packet.
(3) follows by (A6) and (4) follows by (A7).

Subcase $\mathsf{sizeof}(v) > 1$**:** By IH, there exists h_1, $\mathcal{E}_1$, such that

(A8) $h_1 \in [\![\tau_1]\!]_{\mathcal{E}_1}$

(A9) $h_1' = \mathsf{chomp}^{\Downarrow}(h_1, 1)$

(A10) $\mathcal{E}_1 = \mathcal{E}'[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v[0{:}\mathsf{sizeof}(v) - 1]])]$

(A11) $v = \mathcal{E}_1(x)(\iota)@h_1(pkt_{in})[0{:}1]$

By Lemma A.38, $h_2' \in [\![\tau_2]\!]_{\mathcal{E}'[y \mapsto h_1]}$. Since $h_1' \mathbin{++} h_2'$ is defined, i.e., they have disjoint sets of headers and $\mathsf{chomp}^{\Downarrow}$ does not affect the validity of headers, $h_1 \mathbin{++} h_2'$ is defined.
Let $h = h_1 \mathbin{++} h_2'$ and $\mathcal{E} = \mathcal{E}_1$.
(1) follows by (A8) and $h_2' \in [\![\tau_2]\!]_{\mathcal{E}_1[y \mapsto h_1]}$. The latter holds, because x is not free in τ_2 by assumption.
To show (2), we must show that $h_1' \mathbin{++} h_2' = \mathsf{chomp}^{\Downarrow}(h_1 \mathbin{++} h_2', 1) \Leftrightarrow \mathsf{chomp}^{\Downarrow}(h_1, 1) \mathbin{++} h_2' = \mathsf{chomp}^{\Downarrow}(h_1 \mathbin{++} h_2', 1)$. This equality holds, because chomping of one bit from the input packet of h_1 and then concatenating h_2' yields the same heap as concatenating both heaps and then removing the first bit of the input packet.
(3) follows by (A10) and (4) follows by (A11) and (5) follows by (A10).

Subcase *h' contained in the second subset***:**

(A1) $h' = h_1' \mathbin{++} h_2'$ and

(A2) $h_1' \in [\![\{z : \tau_1 \mid |z.pkt_{in}| = 0\}]\!]_{\mathcal{E}'}$ and

(A3) $h_2' \in [\![\mathsf{chompRec}(\tau_2, 1, \iota, x)]\!]_{\mathcal{E}'[y \mapsto h_1']}$.

We distinguish two cases.

Subcase $\mathsf{sizeof}(v) = 1$**:** By IH, for every $h_1' \in [\![\{z : \tau \mid |z.pkt_{in}| = 0\}]\!]_{\mathcal{E}'}$, there exists h_2, $\mathcal{E}_2$, such that

(A4) $h_2 \in [\![\tau_2]\!]_{\mathcal{E}_2}$

(A5) $h_2' = \mathsf{chomp}^{\Downarrow}(h_2, 1)$

(A6) $\mathcal{E}_2 = \mathcal{E}'[y \mapsto h_1'] \setminus x$

(A7) $v = h_2(pkt_{in})[0{:}1]$

Let $h = h_1' \mathbin{++} h_2$ and $\mathcal{E} = \mathcal{E}_2 \setminus y$. We have to show that $h \in [\![\Sigma x{:}\, \tau_1.\tau_2]\!]_{\mathcal{E}}$. By assumption, x is not free in τ_1 and τ_2. By (A2) and by the fact that $\mathcal{E} = \mathcal{E}' \setminus x$, $h_1' \in [\![\{z : \tau_1 \mid |z.pkt_{in}| = 0\}]\!]_{\mathcal{E}}$ and by subtyping, $h_1' \in [\![\tau_1]\!]_{\mathcal{E}}$.
(1) follows together with (A4).
To show (2), we must show that $h_1' \mathbin{++} h_2' = \mathsf{chomp}^{\Downarrow}(h_1' \mathbin{++} h_2, 1) \Leftrightarrow h_1' \mathbin{++} \mathsf{chomp}^{\Downarrow}(h_2, 1) = \mathsf{chomp}^{\Downarrow}(h_1' \mathbin{++} h_2, 1)$. Since by (A2), the input packet of h_1' is empty, the input packet of both heaps are equal.
(3) follows by (A6) and (4) follows by (A7).

Subcase $\mathsf{sizeof}(v) > 1$**:** By IH, for every $h_1' \in [\![\{z : \tau \mid |z.pkt_{in}| = 0\}]\!]_{\mathcal{E}'}$, there exists h_2, $\mathcal{E}_2$, such that

(A8) $h_2 \in [\![\tau_2]\!]_{\mathcal{E}_2}$

(A9) $h_2' = \mathsf{chomp}^{\Downarrow}(h_2, 1)$

(A10) $\mathcal{E}_2 = \mathcal{E}'[y \mapsto h_1', x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v[0{:}\mathsf{sizeof}(v) - 1]])]$

(A11) $v = \mathcal{E}_2(x)(\iota)@h_2(pkt_{in})[0{:}1]$

Let $h = h_1' \mathbin{++} h_2$ and $\mathcal{E} = \mathcal{E}_2 \setminus y$. To show that $h \in [\![\Sigma x : \tau_1.\tau_2]\!]_{\mathcal{E}}$. By assumption that x is not free in τ_1 and by (A2), $h_1' \in [\![\{z : \tau_1 \mid |z.pkt_{in}| = 0\}]\!]_{\mathcal{E}}$ and by subtyping, $h_1' \in [\![\tau_1]\!]_{\mathcal{E}}$.

(1) follows together with (A8).

To show (2), we must show that $h_1' \mathbin{++} h_2' = \mathsf{chomp}^{\Downarrow}(h_1' \mathbin{++} h_2, 1) \Leftrightarrow h_1' \mathbin{++} \mathsf{chomp}^{\Downarrow}(h_2, 1) = \mathsf{chomp}^{\Downarrow}(h_1' \mathbin{++} h_2, 1)$. Since by (A2), the input packet of h_1' is empty, the input packet of both heaps are equal.

(3) follows by (A10) and (4) follows by (A11) and (5) follows by (A10).

Case $\tau = \{y : \tau_1 \mid \varphi\}$**:**

Let $\mathcal{E}'$ where $\mathcal{E}'(x) = (\langle\rangle, \langle\rangle, [\iota \mapsto v])$ be arbitrary.
Let $h' \in [\![\mathsf{chompRec}(\{y : \tau_1 \mid \varphi\}, 1, \iota, x)]\!]_{\mathcal{E}'}$ be arbitrary.

$$
\begin{aligned}
& [\![\mathsf{chompRec}(\{y : \tau_1 \mid \varphi\}, 1, \iota, x)]\!]_{\mathcal{E}'} \\
= {} & [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\{y : \tau_1 \mid \varphi\}, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
= {} & \{h' | h' \in [\![\mathsf{chompRec}(\tau_1, 1, \iota, x)]\!]_{\mathcal{E}'} \wedge \\
& \qquad [\![\mathsf{heapRef}_1(\mathsf{chomp}_1^{\varphi}(\varphi, y, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'[y \mapsto h']}\}
\end{aligned}
$$

We distinguish two cases.

Subcase $\mathsf{sizeof}(v) = 1$**:**

By IH, there exists h_1, $\mathcal{E}_1$, such that

(A1) $h_1 \in [\![\tau_1]\!]_{\mathcal{E}_1}$

(A2) $h' = \mathsf{chomp}^{\Downarrow}(h_1, 1)$

(A3) $\mathcal{E}_1 = \mathcal{E}' \setminus x$

(A4) $v = h_1(pkt_{in})[0{:}1]$

Let $\mathcal{E} = \mathcal{E}_1$ and $h = h_1$. To show (1), we must show that $h_1 \in [\![\{y : \tau_1 \mid \varphi\}]\!]_{\mathcal{E}_1}$, which follows by (A1) and Lemma A.37.

(2) follows by (A2), (3) follows by (A3) and (4) follows by (A4).

Subcase $\mathsf{sizeof}(v) > 1$**:**

By IH, there exists h_1, $\mathcal{E}_1$, such that

(A1) $h_1 \in [\![\tau_1]\!]_{\mathcal{E}_1}$

(A2) $h' = \mathsf{chomp}^{\Downarrow}(h_1, 1)$

(A3) $\mathcal{E}_1 = \mathcal{E}'[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v[0{:}\mathsf{sizeof}(v) - 1]])]$

(A4) $v = \mathcal{E}_1(x)(\iota)@h_1(pkt_{in})[0{:}1]$

Let $\mathcal{E} = \mathcal{E}_1$ and $h = h_1$. To show (1), we must show $h_1 \in [\![\{y : \tau_1 \mid \varphi\}]\!]_{\mathcal{E}_1}$, which follows by (A1) and Lemma A.37.

(2) follows by (A2), (3) follows by (A3), (4) follows by (A4) and (5) follows by (A3).

Case $\tau = \tau_1[y \mapsto \tau_2]$**:**

Let $\mathcal{E}'$ where $\mathcal{E}'(x) = (\langle\rangle, \langle\rangle, [\iota \mapsto v])$ be arbitrary.
Let $h' \in [\![\mathsf{chompRec}(\tau_1[y \mapsto \tau_2], 1, \iota, x)]\!]_{\mathcal{E}'}$ be arbitrary.

$$
\begin{aligned}
& [\![\mathsf{chompRec}(\tau_1[y \mapsto \tau_2], 1, \iota, x)]\!]_{\mathcal{E}'} \\
= & [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1[y \mapsto \tau_2], \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
= & [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0)[y \mapsto \tau_2], \mathsf{b}_0, x, \iota, 1)]\!]_{\mathcal{E}'} \\
= & [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)[y \mapsto \mathsf{heapRef}_1(\tau_2, \mathsf{b}_0, x, \iota, 1)]]\!]_{\mathcal{E}'} \\
= & [\![\mathsf{heapRef}_1(\mathsf{chomp}_1(\tau_1, \mathsf{b}_0), \mathsf{b}_0, x, \iota, 1)[y \mapsto \tau_2]]\!]_{\mathcal{E}'} \\
= & \{h_1' | h_2 \in [\![\tau_2]\!]_{\mathcal{E}'} \wedge h_1' \in [\![\mathsf{chompRec}(\tau_1, 1, x, \iota)]\!]_{\mathcal{E}'[y \mapsto h_2]}\}
\end{aligned}
$$

We distinguish two cases.

Subcase $\mathsf{sizeof}(v) = 1$**:**

By IH, for every $h_2 \in [\![\tau_2]\!]_{\mathcal{E}'}$, there exists h_1, $\mathcal{E}_1$, such that

(A1) $h_1 \in [\![\tau_1]\!]_{\mathcal{E}_1[y \mapsto h_2]}$

(A2) $h_1' = \mathsf{chomp}^{\Downarrow}(h_1, 1)$

(A3) $\mathcal{E}_1 = \mathcal{E}'[y \mapsto h_2] \setminus x$

(A4) $v = h_1(pkt_{in})[0{:}1]$

Let $\mathcal{E} = \mathcal{E}_1$ and $h = h_1$. Since x and y not free in τ_2, for every heap $h_2 \in [\![\tau_2]\!]_{\mathcal{E}'}$ also holds that $h_2 \in [\![\tau_2]\!]_{\mathcal{E}_1}$, from which we can conclude (1), i.e., $h \in [\![\tau_1[y \mapsto \tau_2]]\!]_{\mathcal{E}}$.

To show (2), we must show that $h' = \mathsf{chomp}^{\Downarrow}(h, 1) = \mathsf{chomp}^{\Downarrow}(h_1, 1)$, which follows from (A2) and the fact that $h' = h_1'$.

(3) follows by choice of $\mathcal{E}$ and (A3). (4) follows by (A4).

Subcase $\mathsf{sizeof}(v) > 1$**:**

By IH, for every $h_2 \in [\![\tau_2]\!]_{\mathcal{E}'}$, there exists h_1, $\mathcal{E}_1$, such that

(A1) $h_1 \in [\![\tau_1]\!]_{\mathcal{E}_1[y \mapsto h_2]}$

(A2) $h_1' = \mathsf{chomp}^{\Downarrow}(h_1, 1)$

(A3) $\mathcal{E}_1 = \mathcal{E}'[y \mapsto h_2, x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v[0{:}\mathsf{sizeof}(v) - 1]])]$

(A4) $v = \mathcal{E}_1(x)(\iota)@h_1(pkt_{in})[0{:}1]$

Let $\mathcal{E} = \mathcal{E}_1$ and $h = h_1$.

To show (1), we must show that $h \in [\![\tau_1[y \mapsto \tau_2]]\!]_{\mathcal{E}_1}$. Since x and y not free in τ_2, for every heap $h_2 \in [\![\tau_2]\!]_{\mathcal{E}'}$ also holds that $h_2 \in [\![\tau_2]\!]_{\mathcal{E}_1}$. The result follows by the semantics of heap types.

To show (2), we must show that $h' = \mathsf{chomp}^{\Downarrow}(h_1, 1)$, which follows from (A2) and $h' = h_1'$.

(3) follows by (A3), (4) follows by (A4) and (5) follows also by (A3).

□

Lemma A.46 (Semantic ChompRec Inverse)**.** *For all variables x, values v, $n \in \mathbb{N}$, heap types τ, environments $\mathcal{E}'$ and heaps h' such that $\mathcal{E}'(x) = (\langle\rangle, \langle\rangle, [\iota \mapsto v])$ and $h' \in [\![\mathsf{chompRec}(\tau, n, \iota, x)]\!]_{\mathcal{E}'}$ and x not free in τ and $\mathsf{sizeof}(v) = n$, there exists h and $\mathcal{E}$ such that $h \in [\![\tau]\!]_{\mathcal{E}}$ and $\mathcal{E} = \mathcal{E}' \setminus x$ and $\mathsf{chomp}^{\Downarrow}(h, n) = h'$.*

Proof. Proof by induction on n.

Case $n = 0$**:**

By definition of chompRec, $\mathsf{chompRec}(\tau, 0, \iota, x) = \tau$. Together with assumption $h' \in [\![\mathsf{chompRec}(\tau, n, \iota, x)]\!]_{\mathcal{E}'}$, we know that $h' \in [\![\tau]\!]_{\mathcal{E}'}$. Let $h = h'$. By assumption, x is not free in τ, thus the binding of x in $\mathcal{E}'$ does not affect the semantics of τ. We can therefore remove the binding altogether, so τ describes the same set of heaps both in $\mathcal{E}'$ and in $\mathcal{E}$. We can conclude that $h \in [\![\tau]\!]_{\mathcal{E}}$. $\mathsf{chomp}^{\Downarrow}(h, 0) = h'$ follows from the definition of semantic chomp.

Case $n = 1$**:**

The result directly follows by Lemma A.45.

Case $n = m + 1$**:**

We assume that the lemma holds for $n = m$. We have to show that the lemma also holds for $n = m + 1$. Let h'_0 be some heap such that $h'_0 \in [\![\mathsf{chompRec}(\tau, m + 1, \iota, x)]\!]_{\mathcal{E}'_0}$. Together with Lemma A.40, we can conclude that

$$h'_0 \in [\![\mathsf{chompRec}(\mathsf{chompRec}(\tau, m, \iota, x), 1, \iota, x)]\!]_{\mathcal{E}'_0}$$

By Lemma A.45, there is some h'_1, $\mathcal{E}'_1$ such that $h'_1 \in [\![\mathsf{chompRec}(\tau, m, \iota, x)]\!]_{\mathcal{E}'_1}$ and $h'_0 = \mathsf{chomp}^{\Downarrow}(h'_1, 1)$ where $\mathcal{E}'_1 = \mathcal{E}'_0[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v[0 : m]])]$. By IH, there exists a h_1 and $\mathcal{E}_1$, such that $h_1 \in [\![\tau]\!]_{\mathcal{E}_1}$ where $\mathcal{E}_1 = \mathcal{E}'_1 \setminus x$ and $h'_1 = \mathsf{chomp}^{\Downarrow}(h_1, m)$. From $h'_0 = \mathsf{chomp}^{\Downarrow}(\mathsf{chomp}^{\Downarrow}(h_1, m), 1)$ and Lemma A.41, follows $h'_0 = \mathsf{chomp}^{\Downarrow}(h_1, m + 1)$.

□

Lemma A.47 (Semantic Chomp Inverse)**.** *For all variables x, values v, instances ι, heap types τ, environments $\mathcal{E}'$ and heaps h' such that $\mathcal{E}'(x) = (\langle\rangle, \langle\rangle, [\iota \mapsto v])$ and $h' \in [\![\mathsf{chomp}(\tau, \iota, x)]\!]_{\mathcal{E}'}$ and x not free in τ, there exists h and $\mathcal{E}$ such that $h \in [\![\tau]\!]_{\mathcal{E}}$ and $\mathcal{E} = \mathcal{E}' \setminus x$ and $\mathsf{chomp}^{\Downarrow}(h, \mathsf{sizeof}(\iota)) = h'$.*

Proof. By definition of chomp, we know that

$$\mathsf{chomp}(\tau, \iota, x) = \mathsf{chompRec}(\tau, \mathsf{sizeof}(\iota), x, \iota)$$

The result follows by Lemma A.46. □

Lemma A.48 (Weakening)**.** *If $\Gamma \vdash c : (x : \tau_1) \to \tau_2$ and variable z does not appear free in τ_1 or τ_2, then $\Gamma, z : \tau \vdash c : (x : \tau_1) \to \tau_2$ for any heap type τ.*

Proof. By induction on the typing derivation. □

Lemma A.49 (Input Type Strengthening)**.** *If $\Gamma \vdash c : (x : \tau_1) \to \tau_2$ and $[\![\tau'_1]\!]_{\mathcal{E}'} \subseteq [\![\tau_1]\!]_{\mathcal{E}}$ and $\mathcal{E} \vDash \Gamma$ and $\mathcal{E}' \vDash \Gamma'$ and $\Gamma \subseteq \Gamma'$ and $\mathcal{E} \subseteq \mathcal{E}'$, then $\exists \tau'_2.\Gamma' \vdash c : (x : \tau'_1) \to \tau'_2$ and $\forall h' \in [\![\tau'_1]\!]_{\mathcal{E}'}.[\![\tau'_2]\!]_{\mathcal{E}'[x \mapsto h']} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h']}$*

Proof. By induction on a derivation of $\Gamma \vdash c : (x : \tau_1) \to \tau_2$ with case analysis on the last rule used. We refer to the proof goals as follows:

(1) $\exists \tau'_2.\Gamma' \vdash c : (x : \tau'_1) \to \tau'_2$

(2) $\forall h' \in [\![\tau_1']\!]_{\mathcal{E}'}.[\![\tau_2']\!]_{\mathcal{E}'[x \mapsto h']} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h']}$

We refer to the assumptions as follows:

(A) $[\![\tau_1']\!]_{\mathcal{E}'} \subseteq [\![\tau_1]\!]_{\mathcal{E}}$

(B) $\mathcal{E} \vDash \Gamma$

(C) $\mathcal{E}' \vDash \Gamma'$

(D) $\Gamma \subseteq \Gamma'$

(E) $\mathcal{E} \subseteq \mathcal{E}'$

***Case* T-Add:**

By inversion of rule T-Add, we get

(A1) Excludes $\Gamma\ \tau\ \iota$ and

(A2) $init_{\mathcal{HT}(\iota)} = v$

(A3) $\tau_2 = \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \iota \mid z.pkt_{out} = z.pkt_{in} = \langle\rangle \wedge z.\iota = v\}$.

Let $\tau_2' = \Sigma y : \{z : \tau_1' \mid z \equiv x\}.\{z : \iota \mid z.pkt_{out} = z.pkt_{in} = \langle\rangle \wedge z.\iota = v\}$. By assumptions (A1), (A) and (C) we can conclude that Excludes $\tau_1'\ \iota\ \Gamma'$ must also hold. (1) follows by T-Add.

Let $h' \in [\![\tau_1']\!]_{\mathcal{E}'}$ be arbitrary.

$$
\begin{aligned}
& [\![\Sigma y\colon \{z : \tau_1' \mid z \equiv x\}.\{z : \iota \mid z.pkt_{out} = z.pkt_{in} = \langle\rangle \wedge z.\iota = v\}]\!]_{\mathcal{E}'[x \mapsto h']} \subseteq \\
& [\![\Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \iota \mid z.pkt_{out} = z.pkt_{in} = \langle\rangle \wedge z.\iota = v\}]\!]_{\mathcal{E}[x \mapsto h']} \\
\Leftrightarrow & \{h_1 +\!\!+\ h_2 \mid h_1 \in [\![\{z : \tau_1' \mid z \equiv x\}]\!]_{\mathcal{E}'[x \mapsto h']} \wedge \\
& \qquad h_2 \in [\![\{z : \iota \mid z.pkt_{out} = z.pkt_{in} = \langle\rangle \wedge z.\iota = v\}]\!]_{\mathcal{E}'[x \mapsto h', y \mapsto h_1]}\} \subseteq \\
& \{h_1 +\!\!+\ h_2 \mid h_1 \in [\![\{z : \tau_1 \mid z \equiv x\}]\!]_{\mathcal{E}[x \mapsto h']} \wedge \\
& \qquad h_2 \in [\![\{z : \iota \mid z.pkt_{out} = z.pkt_{in} = \langle\rangle \wedge z.\iota = v\}]\!]_{\mathcal{E}[x \mapsto h', y \mapsto h_1]}\} \\
\Leftrightarrow & \{h' +\!\!+\ h_2 \mid h_2 \in [\![\{z : \iota \mid z.pkt_{out} = z.pkt_{in} = \langle\rangle \wedge z.\iota = v\}]\!]_{\mathcal{E}'[x \mapsto h', y \mapsto h']}\} \subseteq \\
& \{h' +\!\!+\ h_2 \mid h_2 \in [\![\{z : \iota \mid z.pkt_{out} = z.pkt_{in} = \langle\rangle \wedge z.\iota = v\}]\!]_{\mathcal{E}[x \mapsto h', y \mapsto h']}\}
\end{aligned}
$$

The type $\{z : \iota \mid z.pkt_{out} = z.pkt_{in} = \langle\rangle \wedge z.\iota = v\}$ does not contain any free variables, so the semantics does not depend on the environment. In fact, the sets of heaps described by τ_2' and τ_2 is actually equivalent, which shows (2).

***Case* T-Ascribe:**

By inversion of rule T-Ascribe, we get

(A1) $c = c_a\ as\ (x : \tau_1) \to \tau_2$ and

(A2) $\Gamma \vdash c_a : (x : \tau_1) \to \tau_2$

From assumptions (A2) and (D) together with Lemma A.48 follows that

(A3) $\Gamma' \vdash c_a : (x : \tau_1) \to \tau_2$

Since $\mathcal{E}'$ differs from $\mathcal{E}$ only in that it potentially contains additional bindings, we can conclude that

(A4) $[\![\tau_1]\!]_{\mathcal{E}} = [\![\tau_1]\!]_{\mathcal{E}'}$ and together with assumption (B)

(A5) $\Gamma' \vdash \tau_1' <: \tau_1$.

By assumption (A3) and T-Ascribe we get

(A6) $\Gamma' \vdash c_a\ as\ (x : \tau_1) \to \tau_2 : (x : \tau_1) \to \tau_2$

Let $\tau_2' = \tau_2$. (1) follows by T-Sub.

For (2), we have to show that $\forall h' \in [\![\tau_1']\!]_{\mathcal{E}'}.[\![\tau_2]\!]_{\mathcal{E}'[x \mapsto h']} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h']}$. In fact τ_2 describes the same set of heaps, both in $\mathcal{E}$ and $\mathcal{E}'$. Variable x binds to the same heap and both environments provide the same bindings for any other free variable in τ_2.

***Case* T-Extract:**

By inversion of rule T-Extract, we get

(A1) $\Gamma \vdash \mathsf{sizeof}_{pkt_{in}}(\tau_1) \geq \mathsf{sizeof}(\iota)$

(A2) $\tau_2 = \Sigma y\colon \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}.\{z : \mathsf{chomp}(\tau_1, \iota, y) \mid$
$y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}$

Let $\tau_2' = \Sigma y\colon \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}.\{z : \mathsf{chomp}(\tau_1', \iota, y) \mid$
$y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}$. By assumptions (A) and (A1) follows that $\mathsf{sizeof}_{pkt_{in}}(\tau_1') \geq \mathsf{sizeof}(\iota)$ (1) follows by T-Extract.

Let $h' \in [\![\tau_1']\!]_{\mathcal{E}'}$ be arbitrary.

$$
\begin{aligned}
&[\![\Sigma y\colon \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}.\\
&\qquad \{z : \mathsf{chomp}(\tau_1', \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge\\
&\qquad\qquad z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}]\!]_{\mathcal{E}'[x\mapsto h']} \subseteq\\
&[\![\Sigma y\colon \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}.\\
&\qquad \{z : \mathsf{chomp}(\tau_1, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge\\
&\qquad\qquad z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}]\!]_{\mathcal{E}[x\mapsto h']}\\
\Leftrightarrow &\{h_1 \mathbin{++} h_2 \mid h_1 \in [\![\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}]\!]_{\mathcal{E}[x\mapsto h']} \wedge\\
&\qquad h_2 \in [\![\{z : \mathsf{chomp}(\tau_1', \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge\\
&\qquad\qquad z.pkt_{out} = x.pkt_{out} \wedge\\
&\qquad\qquad z \equiv_\iota x\}]\!]_{\mathcal{E}[x\mapsto h', y\mapsto h_1]}\} \subseteq\\
&\{h_1 \mathbin{++} h_2 \mid h_1 \in [\![\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}]\!]_{\mathcal{E}[x\mapsto h']} \wedge\\
&\qquad h_2 \in [\![\{z : \mathsf{chomp}(\tau_1, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge\\
&\qquad\qquad z.pkt_{out} = x.pkt_{out} \wedge\\
&\qquad\qquad z \equiv_\iota x\}]\!]_{\mathcal{E}[x\mapsto h', y\mapsto h_1]}\}\\
\Leftrightarrow &\{h_1 \mathbin{++} h_2 \mid h_1 \in [\![\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}]\!]_{\mathcal{E}[x\mapsto h']} \wedge\\
&\qquad h_2 \in \{h_{22} | h_{22} \in [\![\mathsf{chomp}(\tau_1', \iota, y)]\!]_{\mathcal{E}[x\mapsto h', y\mapsto h_1]} \wedge\\
&\qquad\qquad [\![y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge\\
&\qquad\qquad z \equiv_\iota x]\!]_{\mathcal{E}[x\mapsto h', y\mapsto h_1, z\mapsto h_{22}]}\}\} \subseteq\\
&\{h_1 \mathbin{++} h_2 \mid h_1 \in [\![\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}]\!]_{\mathcal{E}[x\mapsto h']} \wedge\\
&\qquad h_2 \in \{h_{22} | h_{22} \in [\![\mathsf{chomp}(\tau_1, \iota, y)]\!]_{\mathcal{E}[x\mapsto h', y\mapsto h_1]} \wedge\\
&\qquad\qquad [\![y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge\\
&\qquad\qquad z \equiv_\iota x]\!]_{\mathcal{E}[x\mapsto h', y\mapsto h_1, z\mapsto h_{22}]}\}\}
\end{aligned}
$$

By Lemma A.44, we obtain all heaps contained in $[\![\mathsf{chomp}(\tau_1', \iota, y)]\!]_{\mathcal{E}[x\mapsto h', y\mapsto h_1]}$ by taking all heaps from $[\![\tau_1']\!]_{\mathcal{E}'}$ and removing the first $\mathsf{sizeof}(\iota)$ bits from the input packet. From assumption (A) we know that all heaps described by τ_1' are also contained in the set of heaps described by τ_1 and when we remove the first $\mathsf{sizeof}(\iota)$ bits from the input packet, the relation still holds. Since the rest of the types are identical this also holds for the concatenated heaps. This shows (2) and concludes the case.

***Case* T-If:**

By inversion of rule T-If, we get

(A1) $\Gamma; \tau_1 \vdash e : \mathbb{B}$

(A2) $\Gamma \vdash c_1 : (x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}) \to \tau_{12}$

(A3) $\Gamma \vdash c_2 : (x : \{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\}) \to \tau_{22}$

(A4) $\tau_2 = \{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \varphi[x/\mathsf{heap}]\}$

(A5) $c = \mathit{if}(\varphi)\ c_1\ \mathit{else}\ c_2$

To be able to conclude (1) by T-If, we must show that

(1.1) $\Gamma'; \tau'_1 \vdash \varphi : \mathbb{B}$

(1.2) $\Gamma' \vdash c_1 : (x : \{y : \tau'_1 \mid \varphi[y/\mathsf{heap}]\}) \to \tau'_{12}$

(1.3) $\Gamma' \vdash c_2 : (x : \{y : \tau'_1 \mid \neg\varphi[y/\mathsf{heap}]\}) \to \tau'_{22}$

To apply the IH to c_1, we need some τ'_{IH_1} such that

$$[\![\tau'_{IH_1}]\!]_{\mathcal{E}'} \subseteq [\![\{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}]\!]_{\mathcal{E}}$$

Let $\tau'_{IH_1} = \{y : \tau'_1 \mid \varphi[y/\mathsf{heap}]\}$.

By IH, there exists τ'_{12} such that

(A6) $\Gamma' \vdash c_1 : (x : \{y : \tau'_1 \mid \varphi[y/\mathsf{heap}]\}) \to \tau'_{12}$

(A7) $\forall h'_1 \in [\![\{y : \tau'_1 \mid \varphi[y/\mathsf{heap}]\}]\!]_{\mathcal{E}'}.[\![\tau'_{12}]\!]_{\mathcal{E}'[y \mapsto h'_1]} \subseteq [\![\tau_{12}]\!]_{\mathcal{E}[y \mapsto h'_1]}$

With a similar argument as before, also by IH, there exists τ'_{22} such that

(A8) $\Gamma' \vdash c_2 : (x : \{y : \tau'_1 \mid \neg\varphi[y/\mathsf{heap}]\}) \to \tau'_{22}$

(A9) $\forall h'_2 \in [\![\{y : \tau'_1 \mid \neg\varphi[y/\mathsf{heap}]\}]\!]_{\mathcal{E}'}.[\![\tau'_{22}]\!]_{\mathcal{E}'[y \mapsto h'_2]} = [\![\tau_{22}]\!]_{\mathcal{E}[y \mapsto h'_2]}$

$\Gamma'; \tau'_1 \vdash \varphi : \mathbb{B}$ also holds, because the subtyping relation between τ'_1 and τ_1 ensures that heaps described by τ'_1 have the same shape (i.e., the same instances are valid) and thus we can typecheck formula e in the context of type τ'_1.

(1) follows by T-If.

Let $\tau'_2 = \{y : \tau'_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau'_{22} \mid \neg\varphi[x/\mathsf{heap}]\}$

To show $\forall h'_1 \in [\![\tau'_1]\!]_{\mathcal{E}'}.[\![\{y : \tau'_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\}]\!]_{\mathcal{E}'[x \mapsto h'_1]} \subseteq [\![\{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\}]\!]_{\mathcal{E}[x \mapsto h'_1]}$

Let $h'_1 \in [\![\tau'_1]\!]_{\mathcal{E}'}$ be arbitrary. Case distinction on wheter the formula φ in h'_1 evaluates to true or false.

***Subcase** e* **evaluates to true:**

$$\begin{aligned}
& [\![\{y : \tau'_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\}]\!]_{\mathcal{E}'[x \mapsto h'_1]} \subseteq \\
& [\![\{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\}]\!]_{\mathcal{E}[x \mapsto h'_1]} \\
\Leftrightarrow & [\![\{y : \tau'_{12} \mid \mathsf{true}\} + \{y : \tau_{22} \mid \mathsf{false}\}]\!]_{\mathcal{E}'[x \mapsto h'_1]} \subseteq \\
& [\![\{y : \tau_{12} \mid \mathsf{true}\} + \{y : \tau_{22} \mid \mathsf{false}\}]\!]_{\mathcal{E}[x \mapsto h'_1]} \\
\Leftrightarrow & [\![\tau'_{12}]\!]_{\mathcal{E}'[x \mapsto h'_1]} \subseteq [\![\tau_{12}]\!]_{\mathcal{E}[x \mapsto h'_1]}
\end{aligned}$$

The result follows by (A7).

***Subcase** e* **evaluates to false:** Symmetric to previous subcase. The result follows by (A9).

***Case* T-Mod:**

(A1) $\tau_2 = \{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = e[x/\mathsf{heap}]\}$

(A2) Includes $\Gamma\ \tau_1\ \iota$

(A3) $\Gamma; \tau \vdash t : \mathsf{BV}$

(A4) $\mathcal{F}(\iota, f) = \mathsf{BV}$

(A5) $c = \iota.f := e$

To show: There exists τ_2' such that

(1) $\Gamma' \vdash \iota.f := e : (x : \tau_1') \to \tau_2'$ and

(2) $\forall h_1' \in [\![\tau_1']\!]_{\mathcal{E}'}.[\![\tau_2']\!]_{\mathcal{E}[x \mapsto h_1']} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h_1']}$

Let $\tau_2' = \tau_2$.

Includes $\Gamma\ \tau_1'\ \iota$ follows by assumptions (A2) and (A) and set theory. By assumption (A), we know that τ_1' has the same shape (contains the same instances) as τ_1, so we can typecheck expression e in context τ_1' with a bit vector type, from which follows that $\Gamma'; \tau_1' \vdash e : \mathsf{BV}$. (1) follows by T-Mod.

Let $h_1' \in [\![\tau_1']\!]_{\mathcal{E}'}$ be arbitrary. To show (2), we must show that

$$\begin{aligned}
&[\![\tau_2]\!]_{\mathcal{E}'[x \mapsto h_1']} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h_1']} \\
\Leftrightarrow &[\![\{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = t[x/\mathsf{heap}]\}]\!]_{\mathcal{E}'[x \mapsto h_1']} \subseteq \\
&[\![\{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = t[x/\mathsf{heap}]\}]\!]_{\mathcal{E}[x \mapsto h_1']}
\end{aligned}$$

Since the only free variable is x

$$\begin{aligned}
\Leftrightarrow &[\![\{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = t[x/\mathsf{heap}]\}]\!]_{[x \mapsto h_1']} \subseteq \\
&[\![\{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = t[x/\mathsf{heap}]\}]\!]_{[x \mapsto h_1']}
\end{aligned}$$

The result is immediate.

***Case* T-Remit:**

By inversion of rule T-Remit, we get

(A1) Includes $\Gamma\ \tau\ \iota$

(A2) $\tau_2 = \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}$

Let $\tau_2' = \Sigma y\colon \{z : \tau_1' \mid z \equiv x\}.\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}$.

Includes $\Gamma'\ \tau_1'\ \iota$ follows by assumptions (A1) and (A) and set theory. (1) follows by T-Remit. Let $h \in [\![\tau_1']\!]_{\mathcal{E}}$ be arbitrary.

$$\begin{aligned}
&[\![\Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}]\!]_{\mathcal{E}[x \mapsto h]} \\
=&\{h_1 \mathbin{++} h_2 \mid h_1 \in [\![\{z : \tau_1 \mid z \equiv x\}]\!]_{\mathcal{E}[x \mapsto h]} \wedge \\
&\qquad h_2 \in [\![\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}]\!]_{\mathcal{E}[x \mapsto h, y \mapsto h_1]}\} \\
=&\{h \mathbin{++} h_2 \mid h_2 \in [\![\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}]\!]_{\mathcal{E}[x \mapsto h, y \mapsto h]}\}
\end{aligned}$$

x is the only free variable in $\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}$, which maps to the same heap h in both environments $\mathcal{E}[x \mapsto h, y \mapsto h]$ and $\mathcal{E}'[x \mapsto h, y \mapsto h]$.

$$\begin{aligned}
=&\{h \mathbin{++} h_2 \mid h_2 \in [\![\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}]\!]_{\mathcal{E}'[x \mapsto h, y \mapsto h]}\} \\
=&\{h_1 \mathbin{++} h_2 \mid h_1 \in [\![\{z : \tau_1' \mid z \equiv x\}]\!]_{\mathcal{E}'[x \mapsto h]} \wedge \\
&\qquad h_2 \in [\![\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}]\!]_{\mathcal{E}'[x \mapsto h, y \mapsto h_1]}\} \\
=&[\![\Sigma y\colon \{z : \tau_1' \mid z \equiv x\}.\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}]\!]_{\mathcal{E}'[x \mapsto h]}
\end{aligned}$$

This concludes the case by showing (2).

***Case* T-Remove:**

By inversion of rule T-Remove, we get

(A1) Includes $\Gamma\ \tau_1\ \iota$

(A2) $\tau_2 = \{y : \top \mid \varphi_\iota \wedge \varphi_{pkt} \wedge \neg y.\iota.valid\}$

Let $\tau_2' = \tau_2$.

Includes $\Gamma\ \tau_1'\ \iota$ follows by assumptions (A1) and (A) and set theory. (1) then follows by T-Remove.

$[\![\{y : \top \mid \varphi_\iota \wedge \varphi_{pkt} \wedge \neg y.\iota.valid\}]\!]_{\mathcal{E}'[x \mapsto h']} \subseteq [\![\{y : \top \mid \varphi_\iota \wedge \varphi_{pkt} \wedge \neg y.\iota.valid\}]\!]_{\mathcal{E}[x \mapsto h']}$ also holds, because the only free variable is x, which maps to the same heap in both environments. This shows (2) and concludes the case.

***Case* T-Reset:**

By inversion of rule T-Reset, we get

(A1) $c = reset$

(A2) $\tau_2 = \Sigma y : \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}.\{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{in}\}$

Let $\tau_2' = \tau_2$. (1) follows by T-Reset. Let $h_1' \in [\![\tau_1']\!]_{\mathcal{E}'}$ be arbitrary.

$$\begin{aligned}
&[\![\Sigma y : \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}. \\
&\qquad \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{in}\}]\!]_{\mathcal{E}'[x \mapsto h_1']} \subseteq \\
&[\![\Sigma y : \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}. \\
&\qquad \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{in}\}]\!]_{\mathcal{E}[x \mapsto h_1']}
\end{aligned}$$

Both sets are actually equal, because x is the only free variable in τ_2 and τ_2' respectively. Thus, all other bindings in the environments $\mathcal{E}$ and $\mathcal{E}'$ have no effect on the semantics of τ_2 and τ_2' respectively. This shows (2) and concludes the case.

***Case* T-Seq:**

By inversion of rule T-Seq, we get

(A1) $c = c_1; c_2$

(A2) $\Gamma \vdash c_1 : (x : \tau_1) \to \tau_{12}$

(A3) $\Gamma, x : \tau_1 \vdash c_2 : (y : \tau_{12}) \to \tau_{22}$

(A4) $\tau_2 = \tau_{22}[y \mapsto \tau_{12}]$

By IH with (A2), (A), (B) and (C), there exists some τ_{12}' such that

(A5) $\Gamma' \vdash c_1 : (x : \tau_1') \to \tau_{12}'$

(A6) $\forall h_1' \in [\![\tau_1']\!]_{\mathcal{E}'}.[\![\tau_{12}']\!]_{\mathcal{E}'[x \mapsto h_1']} \subseteq [\![\tau_{12}]\!]_{\mathcal{E}[x \mapsto h_1']}$

Apply the IH again to c_2:

Let $h_1' \in [\![\tau_1']\!]_{\mathcal{E}'}$ be arbitrary. By (A3) $\Gamma, x : \tau_1 \vdash c_2 : (y : \tau_{12}) \to \tau_{22}$. By (A6), $[\![\tau_{12}']\!]_{\mathcal{E}'[x \mapsto h_1']} \subseteq [\![\tau_{12}]\!]_{\mathcal{E}[x \mapsto h_1']}$. $\mathcal{E}[x \mapsto h_1'] \vDash \Gamma, x : \tau_1$ because by assumption $\mathcal{E} \vDash \Gamma$

the entailment holds for all $x_i \neq x$. For x there exists a binding to heap $h'_1 \in [\![\tau_1]\!]_{\mathcal{E}}$ (with assumption (A)) and the entailment between h and τ_1 trivially holds.

To show $\mathcal{E}'[x \mapsto h'_1] \vDash \Gamma', x : \tau'_1$, we must show that $\forall x_i, \tau_i . \Gamma'(x_i) = \tau_i \Rightarrow \mathcal{E}'[x \mapsto h'_1](x_i) = h_i \wedge h_i \vDash_{\mathcal{E}[x \mapsto h'_1]} \tau_i$. Case $x_i \neq x$: this holds by assumption (C). Case $x_i = x$. $\mathcal{E}'[x \mapsto h'_1](x) = h'_1$. To show that $h'_1 \vDash_{\mathcal{E}'[x \mapsto h'_1]} \tau'_1 \Leftrightarrow h'_1 \in [\![\tau'_1]\!]_{\mathcal{E}'[x \mapsto h'_1]}$. By assumption, x is not free in τ'_1, so we can equivalently show that $h'_1 \in [\![\tau'_1]\!]_{\mathcal{E}'}$, which holds by assumption.

Again by IH, there exists some τ'_{22} such that

(A7) $\Gamma', x : \tau'_1 \vdash c_2(y : \tau'_{12}) \to \tau'_{22}$

(A8) $\forall h'_{12} \in [\![\tau'_{12}]\!]_{\mathcal{E}'[x \mapsto h'_1]} . [\![\tau'_{22}]\!]_{\mathcal{E}'[x \mapsto h_1, y \mapsto h'_{12}]} \subseteq [\![\tau_{22}]\!]_{\mathcal{E}[x \mapsto h_1, y \mapsto h'_{12}]}$

Let $\tau'_2 = \tau'_{22}[y \mapsto \tau'_{12}]$. (1) follows by T-Seq.

For (2), we must show that $\forall h'_1 \in [\![\tau'_1]\!]_{\mathcal{E}'} . [\![\tau'_2]\!]_{\mathcal{E}'[x \mapsto h'_1]} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h'_1]}$

Let $h'_1 \in [\![\tau'_1]\!]_{\mathcal{E}'}$ be arbitrary.

$$
\begin{aligned}
& [\![\tau'_{22}[y \mapsto \tau'_{12}]]\!]_{\mathcal{E}'[x \mapsto h'_1]} \subseteq [\![\tau_{22}[y \mapsto \tau_{12}]]\!]_{\mathcal{E}[x \mapsto h'_1]} \\
\Leftrightarrow & \bigcup_{h'_{12} \in [\![\tau'_{12}]\!]_{\mathcal{E}'[x \mapsto h'_1]}} [\![\tau'_{22}]\!]_{\mathcal{E}'[x \mapsto h'_1, y \mapsto h'_{12}]} \subseteq \bigcup_{h_{12} \in [\![\tau_{12}]\!]_{\mathcal{E}[x \mapsto h'_1]}} [\![\tau_{22}]\!]_{\mathcal{E}[x \mapsto h'_1, y \mapsto h_{12}]}
\end{aligned}
$$

The result follows by (A6), (A8) and set theory.

***Case* T-Skip:**

By inversion of rule T-Skip, we get

(A1) $c = skip$

(A2) $\tau_2 = \{y : \tau_1 \mid y \equiv x\}$

Let $\tau'_2 = \{y : \tau'_1 \mid y \equiv x\}$. (1) follows by T-Skip.

To show (2), let $h' \in [\![\tau'_1]\!]_{\mathcal{E}'}$ be an arbitrary heap.

$$
\begin{aligned}
& [\![\{y : \tau'_1 \mid y \equiv x\}]\!]_{\mathcal{E}'[x \mapsto h']} \subseteq [\![\{y : \tau_1 \mid y \equiv x\}]\!]_{\mathcal{E}[x \mapsto h']} \\
\Leftrightarrow & \{h'\} \subseteq [\![\{y : \tau_1 \mid y \equiv x\}]\!]_{\mathcal{E}[x \mapsto h']} \\
\Leftrightarrow & \{h'\} \subseteq \{h'\} && \text{by (A)}
\end{aligned}
$$

***Case* T-Sub:**

By inversion of rule T-Sub, we get

(A1) $\Gamma \vdash c : (x : \tau_3) \to \tau_4$

(A2) $\Gamma \vdash \tau_1 <: \tau_3$

(A3) $\Gamma, x : \tau_1 \vdash \tau_4 <: \tau_2$

By assumption $[\![\tau'_1]\!]_{\mathcal{E}'} \subseteq [\![\tau_1]\!]_{\mathcal{E}}$ and from (A2) follows that $[\![\tau_1]\!]_{\mathcal{E}} \subseteq [\![\tau_3]\!]_{\mathcal{E}}$ and thus $[\![\tau'_1]\!]_{\mathcal{E}'} \subseteq [\![\tau_3]\!]_{\mathcal{E}}$. By IH, there exists τ'_4 such that

(A4) $\Gamma' \vdash c : (x : \tau'_1) \to \tau'_4$

(A5) $\forall h' \in [\![\tau'_1]\!]_{\mathcal{E}'} . [\![\tau'_4]\!]_{\mathcal{E}'[x \mapsto h']} \subseteq [\![\tau_4]\!]_{\mathcal{E}[x \mapsto h']}$

Let $\tau_2' = \tau_4'$. (1) follows by (A4).

For (1), we have to show that $\forall h' \in [\![\tau_1']\!]_{\mathcal{E}'}.[\![\tau_4']\!]_{\mathcal{E}'[x \mapsto h']} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h']}$, which follows by (A3) and (A5) and by set theory.

□

Lemma A.50 (Formulae Preservation). *If* $\Gamma; \tau \vdash \varphi : \mathbb{B}$ *and* $\mathcal{E} \vDash \Gamma$ *and* $(I, O, H) \vDash_{\mathcal{E}} \tau$ *and* $\langle I, O, H, \varphi \rangle \to \varphi'$ *then* $\Gamma; \tau \vdash \varphi' : \mathbb{B}$.

Proof. By induction on a derivation of $\Gamma; \tau \vdash \varphi : \mathbb{B}$. □

Lemma A.51 (Semantic Formulae Preservation). *If* $\Gamma; \tau \vdash \varphi : \mathbb{B}$ *and* $\mathcal{E} \vDash \Gamma$ *and* $(I, O, H) \vDash_{\mathcal{E}} \tau$ *and* $\langle I, O, H, \varphi \rangle \to \varphi'$ *then*

$$[\![\varphi[x/\mathsf{heap}]]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} = [\![\varphi'[x/\mathsf{heap}]]\!]_{\mathcal{E}[x \mapsto (I,O,H)]}$$

Proof. By induction on a derivation of $\Gamma; \tau \vdash \varphi : \mathbb{B}$. □

Lemma A.52 (Expression Preservation). *If* $\Gamma; \tau \vdash e : \sigma$ *and* $\mathcal{E} \vDash \Gamma$ *and* $(I, O, H) \vDash_{\mathcal{E}} \tau$ *and* $\langle I, O, H, e \rangle \to e'$ *then* $\Gamma; \tau \vdash e' : \sigma$.

Proof. By induction on a typing derivation of $\Gamma; \tau \vdash e : \sigma$. □

Lemma A.53 (Semantic Expression Preservation). *If* $\Gamma; \tau \vdash e : \sigma$ *and* $\mathcal{E} \vDash \Gamma$ *and* $(I, O, H) \vDash_{\mathcal{E}} \tau$ *and* $\langle I, O, H, e \rangle \to e'$ *then*

$$[\![e[x/\ cmdVar]]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} = [\![e'[x/\mathsf{heap}]]\!]_{\mathcal{E}[x \mapsto (I,O,H)]}$$

Proof. By induction on a typing derivation of $\Gamma; \tau \vdash e : \sigma$. □

Theorem A.54 (Preservation). *If* $\Gamma \vdash c : (x : \tau_1) \to \tau_2$, $\langle I, O, H, c \rangle \to \langle I', O', H', c' \rangle$, $\mathcal{E} \vDash \Gamma$ *and* $(I, O, H) \vDash_{\mathcal{E}} \tau_1$, *then* $\exists \Gamma', \mathcal{E}', x', \tau_1', \tau_2'.\Gamma' \vdash c' : (x' : \tau_1') \to \tau_2'$ *and* $\mathcal{E}' \vDash \Gamma'$ *and* $\Gamma \subseteq \Gamma'$ *and* $\mathcal{E} \subseteq \mathcal{E}'$ *and* $(I', O', H') \vDash_{\mathcal{E}'} \tau_1'$ *and* $[\![\tau_2']\!]_{\mathcal{E}'[x' \mapsto (I',O',H')]} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto (I,O,H)]}$

Proof. By induction on a derivation of $\Gamma \vdash c : (x : \tau_1) \to \tau_2$ with case analysis on the last rule used. We refer to the proof goals as follows:

(1) $\Gamma' \vdash c' : (x' : \tau_1') \to \tau_2'$

(2) $\mathcal{E}' \vDash \Gamma'$

(3) $\Gamma \subseteq \Gamma'$

(4) $\mathcal{E} \subseteq \mathcal{E}'$

(5) $(I', O', H') \vDash_{\mathcal{E}'} \tau_1'$

(6) $[\![\tau_2']\!]_{\mathcal{E}'[x' \mapsto (I',O',H')]} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto (I,O,H)]}$

General assumptions:

(A) $\Gamma \vdash c : (x : \tau_1) \to \tau_2$

(B) $\langle I, O, H, c \rangle \to \langle I', O', H', c' \rangle$

(C) $\mathcal{E} \vDash \Gamma$

(D) $(I, O, H) \vDash_{\mathcal{E}} \tau_1$

***Case* T-ADD:**

By inversion of rule T-ADD, we get

(A1) $c = add(\iota)$

(A2) Excludes $\Gamma\ \tau_1\ \iota$

(A3) $\tau_2 = \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle \wedge z.\iota = v\}$

Only evaluation rule E-ADD applies to c:

(A4) $\iota \notin dom(H)$

(A5) $\mathcal{HT}(\iota) = \eta$

(A6) $init = v$.

(A7) $I' = I$ and $O' = O$ and $H' = H[\iota \mapsto v]$ and $c' = skip$

Let $\Gamma' = \Gamma, x : \tau_1$ and $\mathcal{E}' = \mathcal{E}[x \mapsto (I, O, H)]$.

Let $\tau_1' = \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle \wedge z.\iota = v\}$ and

$\tau_2' = \{w : \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle \wedge z.\iota = v\} \mid w \equiv x'\}$.

(1) follows by T-SKIP and (2) follows by assumptions (C) and (D) and Lemma A.30. (3) and (4) are immediate.

To show (5), we must show that $(I, O, H[\iota \mapsto v]) \vDash_{\mathcal{E}[x \mapsto (I,O,H)]} \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle \wedge z.\iota = v\}$. By ENT-SIGMA, we must show that

(5.1) $(I, O, H) \vDash_{\mathcal{E}[x \mapsto (I,O,H)]} \{z : \tau_1 \mid z \equiv x\}$ and

(5.2) $(\langle\rangle, \langle\rangle, [\iota \mapsto v]) \vDash_{\mathcal{E}[x \mapsto (I,O,H)y \mapsto (I,O,H)]} \{z : \iota \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = \langle\rangle \wedge z.\iota = v\}$.

(5.1) follows by ENT-REFINE and (D). To show (5.2), by ENT-REFINE, we must show that

(5.2.1) $(\langle\rangle, \langle\rangle, [\iota \mapsto v]) \vDash_{\mathcal{E}[y \mapsto (I,O,H)]} \{z : \top \mid z.\iota.valid \wedge \bigwedge_{\kappa \in dom(\mathcal{HT})} \neg\kappa.valid\}$ and

(5.2.2) $[\![z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = \langle\rangle \wedge z.\iota = v]\!]_{\mathcal{E}[y \mapsto (I,O,H), z \mapsto (\langle\rangle,\langle\rangle,[\iota \mapsto v])]} = true$

(5.2.1) follows by ENT-REFINE, ENT-TOP and the semantics of formulae. (5.2.2) follows from the semantics of formulae.

(6) follows by

$$
\begin{aligned}
&[\![\Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle \wedge z.\iota = v\}]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} \\
=&\{(I, O, H[\iota \mapsto v])\} \\
=&\{h \mid h \in [\![\Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle \wedge \\
&\qquad z.\iota = v\}]\!]_{\mathcal{E}[x \mapsto (I,O,H), x' \mapsto (I,O,H[\iota \mapsto v])]} \wedge \\
&\qquad [\![w \equiv x']\!]_{\mathcal{E}[x \mapsto (I,O,H), x' \mapsto (I,O,H[\iota \mapsto v]), w \mapsto h]}\} \\
=&[\![\{w : \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle \wedge z.\iota = v\} \mid \\
&\qquad w \equiv x'\}]\!]_{\mathcal{E}[x \mapsto (I,O,H), x' \mapsto (I,O,H[\iota \mapsto v])]}
\end{aligned}
$$

***Case* T-Ascribe:**

By inversion of rule T-Ascribe, we get

(A1) $c = c_a \; as \; (x : \tau_1) \to \tau_2$

(A2) $\Gamma \vdash c_a : (x : \tau_1) \to \tau_2$

There is one evaluation rule that applies to c, E-Ascribe, so $I' = I$ and $O' = O$ and $H' = H$ and $c' = c_a$. Let $\Gamma' = \Gamma$, $\mathcal{E}' = \mathcal{E}$, $\tau_1' = \tau_1$ and $\tau_2' = \tau_2$. (1) follows by (A2), (2) follows by assumption (C). (3) and (4) are immediate. (5) follows by assumption (D) and (6) follows from the equality of τ_2 and τ_2', which itself follows by reflexivity.

***Case* T-Extract:**

By inversion of rule T-Extract, we get

(A1) $c = extract(\iota)$

(A2) $\Gamma \vdash \mathsf{sizeof}_{pkt_{in}}(\tau_1) = \mathsf{sizeof}(\iota)$

(A3) $\tau_2 = \Sigma y\colon \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}.\{z : \mathsf{chomp}(\tau_1, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}$

Only evaluation rule E-Extract applies to c:

(A4) $deserialize_{\mathcal{HT}(\iota)}(I) = (v, I')$

(A5) $O' = O$

(A6) $H' = H[\iota \mapsto v]$

(A7) $c' = skip$

Let $\Gamma' = \Gamma, x : \tau_1, \mathcal{E}' = \mathcal{E}[x \mapsto (I, O, H)]$.

Let $\tau_1' = \Sigma y : \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}.\{z : \mathsf{chomp}(\tau_1, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}$.

Let $\tau_2' = \{v : \Sigma y : \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}.\{z : \mathsf{chomp}(\tau_1, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\} \mid v \equiv x\}$.

(1) follows by T-Skip and (2) follows by assumptions (C) and (D) and Lemma A.30. (3) and (4) are immediate.

To show (5), we must show that $(I', O, H[\iota \mapsto v]) \vDash_{\mathcal{E}[x \mapsto (I,O,H)]} \Sigma y : \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}.\{z : \mathsf{chomp}(\tau_1, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}$.

By Ent-Sigma, we must show that

(5.1) $(\langle\rangle, \langle\rangle, [\iota \mapsto v]) \vDash_{\mathcal{E}[x \mapsto (I,O,H)]} \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}$, which follows by Ent-Refine and the semantics of types and

(5.2) $(I', O, H) \vDash_{\mathcal{E}[x \mapsto (I,O,H), y \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v])]} \{z : \mathsf{chomp}(\tau_1, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}$

By Ent-Refine, we must show that

(5.2.1) $(I', O, H) \vDash_{\mathcal{E}[x \mapsto (I,O,H), y \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v])]} \mathsf{chomp}(\tau_1, \iota, y)$
By Lemma A.28, it is sufficient to show that

$$(I', O, H) \in [\![\mathsf{chomp}(\tau_1, \iota, y)]\!]_{\mathcal{E}[x \mapsto (I,O,H), y \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v])]}$$

By assumption (D) and by Lemma A.28 follows $(I, O, H) \in [\![\tau_1]\!]_{\mathcal{E}}$.
By Lemma A.44, there exists some heap

$$h \in [\![\mathsf{chomp}(\tau_1, \iota, y)]\!]_{\mathcal{E}[x \mapsto (I,O,H), y \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v])]}$$

such that $h = \mathsf{chomp}^{\Downarrow}((I, O, H), \mathsf{sizeof}(\iota))$.
From the definition of $\mathsf{chomp}^{\Downarrow}$ follows that

$$\mathsf{chomp}^{\Downarrow}((I, O, H), \mathsf{sizeof}(\iota)) = (I'', O, H)$$

where $I'' = I[\mathsf{sizeof}(\iota):] = I'$.

(5.2.2) $[\![y.\iota@z.pkt_{in} = x.pkt_{in} \ \wedge \ z.pkt_{out} = x.pkt_{out} \ \wedge$
$z \equiv_\iota x]\!]_{\mathcal{E}[x \mapsto (I,O,H), y \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto v]), z \mapsto (I', O, H)]}$, which follows by the definition of *deserialize* and the semantics of formulae and expressions.

Finally, we must show that $[\![\tau_2]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} \subseteq [\![\tau_2']\!]_{\mathcal{E}[x \mapsto (I', O, H[\iota \mapsto v])]}$.

$$\begin{aligned}
&[\![\tau_2]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} \\
=&[\![\Sigma y\colon \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}. \\
&\qquad \{z : \mathsf{chomp}(\tau_1, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge \\
&\qquad\qquad z \equiv_\iota x\}]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} \\
=&\{h_1 \mathbin{++} h_2 \mid h_1 \in [\![\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} \wedge \\
&\qquad h_2 \in [\![\{z : \mathsf{chomp}(\tau_1, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge \\
&\qquad\qquad z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}]\!]_{\mathcal{E}[x \mapsto (I,O,H), y \mapsto h_1]}\} \\
=&\{h_1 \mathbin{++} h_2 \mid h_1 \in [\![\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} \wedge \\
&\qquad h_2 \in \{h_2' \in [\![\mathsf{chomp}(\tau_1, \iota, y)]\!]_{\mathcal{E}[x \mapsto (I,O,H), y \mapsto h_1]} \wedge \\
&\qquad\qquad [\![y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge \\
&\qquad\qquad z \equiv_\iota x]\!]_{\mathcal{E}[x \mapsto (I,O,H), y \mapsto h_1, z \mapsto h_2']}\}\}
\end{aligned}$$

By Lemma A.44

$$\begin{aligned}
=&\{h_1 \mathbin{++} h_2 \mid h_1 \in [\![\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} \wedge \\
&\qquad h_2 \in \{h \in [\![\tau_1]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} \wedge h_2' = \mathsf{chomp}^{\Downarrow}(h, \mathsf{sizeof}(\iota)) \wedge \\
&\qquad\qquad [\![y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge \\
&\qquad\qquad z \equiv_\iota x]\!]_{\mathcal{E}[x \mapsto (I,O,H), y \mapsto h_1, z \mapsto h_2']}\}\}
\end{aligned}$$

By definition of $\mathsf{chomp}^{\Downarrow}$ and semantics of types

$$
\begin{aligned}
&=\{(I', O, H[\iota \mapsto v])\}\\
&=\{h \mid h \in [\![\Sigma y\colon \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}.\\
&\qquad\qquad \{z : \mathsf{chomp}(\tau_1, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge\\
&\qquad\qquad\quad z.pkt_{out} = x.pkt_{out} \ \wedge z \equiv_\iota x\}]\!]_{\mathcal{E}[x\mapsto(I,O,H),x'\mapsto(I',O,H[\iota\mapsto v])]} \wedge\\
&\qquad [\![v \equiv x']\!]_{\mathcal{E}[x\mapsto(I,O,H),x'\mapsto(I',O,H[\iota\mapsto v]),v\mapsto h]}\}\\
&=[\![\{v : \Sigma y\colon \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}.\\
&\qquad\qquad \{z : \mathsf{chomp}(\tau_1, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge\\
&\qquad\qquad\qquad\qquad z.pkt_{out} = x.pkt_{out} \ \wedge z \equiv_\iota x\} \mid\\
&\qquad v \equiv x\}]\!]_{\mathcal{E}[x\mapsto(I,O,H),x'\mapsto(I',O,H[\iota\mapsto v])]}
\end{aligned}
$$

This shows (6) and concludes the case.

***Case* T-If:**

By inversion of rule T-If, we get

(A1) $\Gamma; \tau_1 \vdash \varphi : \mathbb{B}$

(A2) $\Gamma \vdash c_1 : (x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}) \to \tau_{12}$

(A3) $\Gamma \vdash c_2 : (x : \{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\}) \to \tau_{22}$

(A4) $\tau_2 = \{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\}$

T-If There are three evaluation rules that apply to c.

***Subcase* E-If:**

(A5) $c' = \mathit{if}\,(\varphi')\ c_1\ \mathit{else}\ c_2$

(A6) $I' = I, O' = O, H' = H$

Let $\Gamma' = \Gamma$, $\mathcal{E}' = \mathcal{E}$, $x' = x$,
$\tau_1' = \tau_1$ and
$\tau_2' = \{y : \tau_{12} \mid \varphi'[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi'[x/\mathsf{heap}]\}$.
By Lemma A.50, $\Gamma; \tau_1 \vdash \varphi' : \mathbb{B}$.
By Lemma A.51, $[\![\varphi[x/\mathsf{heap}]]\!]_{\mathcal{E}[x\mapsto(I,O,H)]} = [\![\varphi'[x/\mathsf{heap}]]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}$.
From (A2) follows that $\Gamma \vdash c_1 : (x : \{y : \tau_1 \mid \varphi'[y/\mathsf{heap}]\}) \to \tau_{12}$ and from (A3) follows that $\Gamma \vdash c_2 : (x : \{y : \tau_1 \mid \neg\varphi'[y/\mathsf{heap}]\}) \to \tau_{22}$ (1) follows by T-If.
(2) follows by assumption (C), (3) and (4) are immediate.
(5) follows by assumption (D).
(6) follows together with the assumption

$$[\![\varphi[x/\mathsf{heap}]]\!]_{\mathcal{E}[x\mapsto(I,O,H)]} = [\![\varphi'[x/\mathsf{heap}]]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}$$

from the equality $[\![\tau_2']\!]_{\mathcal{E}'[x\mapsto(I',O',H')]} = [\![\tau_2]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}$.

$$
\begin{aligned}
&[\![\tau_2]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}\\
&=[\![\{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\}]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}\\
&=[\![\{y : \tau_{12} \mid \varphi'[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi'[x/\mathsf{heap}]\}]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}\\
&=[\![\tau_2']\!]_{\mathcal{E}'[x\mapsto(I',O',H')]}
\end{aligned}
$$

***Subcase* E-IfTrue:**

(A7) $c' = c_1$

(A8) $I' = I, O' = O, H' = H$

(A9) $\varphi = \mathsf{true}$

Let $\Gamma' = \Gamma, \mathcal{E}' = \mathcal{E}, x' = x$,
$\tau_1' = \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\} = \{y : \tau_1 \mid \mathsf{true}\} = \tau_1$ and
$\tau_2' = \tau_{12}$.
(1) holds by assumption (A2), (2) holds by assumption (C). (3) and (4) are immediate and (5) $(I', O', H') \vDash_{\mathcal{E}'} \tau_1' \Leftrightarrow (I, O, H) \vDash_{\mathcal{E}} \tau_1$ holds by assumption (D).
(6) follows from

$$
\begin{aligned}
&[\![\{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\}]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}\\
=&[\![\{y : \tau_{12} \mid \mathsf{true}\} + \{y : \tau_{22} \mid \mathsf{false}\}]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}\\
=&[\![\tau_{12}]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}\\
=&[\![\tau_2']\!]_{\mathcal{E}'[x\mapsto(I',O',H')]}
\end{aligned}
$$

***Subcase* E-IfFalse:**

Symmetric to previous subcase.

***Case* T-Mod:**

By inversion of rule T-Mod, we get

(A1) $c = \iota.f := e$

(A2) $\mathsf{Includes}\ \Gamma\ \tau_1\ \iota$

(A3) $\mathcal{F}(\iota, f) = \mathsf{BV}$

(A4) $\Gamma; \tau_1 \vdash e : \mathsf{BV}$

(A5) $\tau_2 = \{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = e[x/\mathsf{heap}]\}$

(A6) $\varphi_{pkt} \triangleq y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out}$

(A7) $\varphi_\iota \triangleq \forall \kappa \in dom(\mathcal{HT}).\iota \neq \kappa \rightarrow y.\kappa = x.\kappa$

(A8) $\varphi_f \triangleq \forall g \in dom(\mathcal{HT}(\iota)).f \neq g \rightarrow y.\iota.g = x.\iota.g$

There are two evaluation rule that apply to c.

***Subcase* E-Mod:**

(A9) $H(\iota) = r$

(A10) $r' \triangleq \{\mathrm{r\ with}\ f = v\}$

(A11) $(I', O', H') = (I, O, H[\iota \mapsto r'])$

(A12) $c' = skip$

(A13) $e = v$

(A14) $\tau_2 = \{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = v[x/\mathsf{heap}]\}$

Let $\Gamma' = \Gamma, x : \tau_1$ and $\mathcal{E}' = \mathcal{E}[x \mapsto (I, O, H)]$ and $\tau_1' = \tau_2$ and $\tau_2' = \{z : \tau_2 \mid z \equiv x\}$.

(1) follows by T-Skip and T-Sub. (2) follows by assumptions (C) and (D) and Lemma A.30. (3) and (4) are immediate.

To show (5), we must show that $(I, O, H[\iota \mapsto r']) \vDash_{\mathcal{E}[x \mapsto (I,O,H)]} \top$ and $[\![\varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = v[x/\mathsf{heap}]]\!]_{\mathcal{E}[x \mapsto (I,O,H), y \mapsto (I,O,H[\iota \mapsto r'])]} = \mathsf{true}$. Since $\mathcal{E}(x)(I) = \mathcal{E}(y)(I)$ and $\mathcal{E}(x)(O) = \mathcal{E}(y)(O)$, φ_{pkt} holds. Similarly, $\mathcal{E}(x)(H) = \mathcal{E}(y)(H)$ in every aspect, except for field f of instance ι, so φ_f and φ_{inst} also hold. $y.\iota.f = v[x/\mathsf{heap}]$ also holds, because

$$\begin{aligned}
& [\![y.\iota.f]\!]_{\mathcal{E}[x \mapsto (I,O,H), y \mapsto (I,O,H[\iota \mapsto r'])]} = \\
& [\![v[x/\mathsf{heap}]]\!]_{\mathcal{E}[x \mapsto (I,O,H), y \mapsto (I,O,H[\iota \mapsto r'])]} \\
\Leftrightarrow\ & v = [\![v[x/\mathsf{heap}]]\!]_{[x \mapsto (I,O,H)]} \\
\Leftrightarrow\ & v = v
\end{aligned}$$

To show

$$\begin{aligned}
& [\![\tau_2']\!]_{\mathcal{E}'[x' \mapsto (I',O',H')]} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} \\
\Leftrightarrow\ & [\![\{z : \tau_2 \mid z \equiv x\}]\!]_{\mathcal{E}[x \mapsto (I,O,H), x' \mapsto (I',O',H')]} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} \\
\Leftrightarrow\ & \{(I', O', H')\} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} \\
\Leftrightarrow\ & \{(I, O, H[\iota \mapsto r'])\} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto (I,O,H)]}
\end{aligned}$$

To show (6), let $h = (I, O, H[\iota \mapsto r'])$. Therefore,

(A15) $h(pkt_{in}) = I$ and

(A16) $h(pkt_{out}) = O$ and

(A17) for all $\kappa \in dom(\mathcal{HT})$ such that $\kappa \neq \iota$, $h(\kappa) = H(\kappa)$

(A18) for all $g \in dom(\mathcal{HT}(\iota))$ such that $g \neq f$, $h(\iota)(g) = H(\iota)(g)$

(A19) $h(\iota)(f) = v[x/\mathsf{heap}] = v$

From the semantics of types follows that $h \in [\![\tau_2]\!]_{\mathcal{E}[x \mapsto (I,O,H)]}$.

***Subcase* E-Mod1:**

(A20) $\langle I, O, H, t\rangle \to e'$

(A21) $c' = \iota.f := e'$

(A22) $I' = I, O' = O$ and $H' = H$

Let $\Gamma' = \Gamma$ and $\mathcal{E}' = \mathcal{E}$ and $x' = x$.

Let $\tau_1' = \tau_1$ and

$\tau_2' = \{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = e'[x/\mathsf{heap}]\}$.

By semantic expression preservation (Lemma A.53), we know that $[\![e[x/\mathsf{heap}]]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} = [\![e'[x/\mathsf{heap}]]\!]_{\mathcal{E}[x \mapsto (I,O,H)]}$.

If $\Gamma; \tau_1 \vdash e : \mathsf{BV}$ and $[\![e[x/\mathsf{heap}]]\!]_{\mathcal{E}[x \mapsto (I,O,H)]} = [\![e'[x/\mathsf{heap}]]\!]_{\mathcal{E}[x \mapsto (I,O,H)]}$ holds, then it must hold that $\Gamma; \tau_1 \vdash e' : \mathsf{BV}$.

(1) follows by T-Mod, (2) follows from assumption (C). (3) and (4) are immediate. (5) follows from assumption (D).

(6) follows from

$$
\begin{aligned}
&[\![\tau_2']\!]_{\mathcal{E}'[x'\mapsto(I',O',H')]} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x\mapsto(I,O,H)]} \\
\Leftrightarrow &[\![\tau_2']\!]_{\mathcal{E}[x\mapsto(I,O,H)]} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x\mapsto(I,O,H)]} \\
\Leftrightarrow &[\![\{y:\top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = e'[x/\mathsf{heap}]\}]\!]_{\mathcal{E}[x\mapsto(I,O,H)]} \subseteq \\
&[\![\{y:\top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = e[x/\mathsf{heap}]\}]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}
\end{aligned}
$$

together with assumption

$$[\![e[x/\mathsf{heap}]]\!]_{\mathcal{E}[x\mapsto(I,O,H)]} = [\![e'[x/\mathsf{heap}]]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}$$

and the semantics of types.

***Case* T-Remit:**

By inversion of rule T-Remit, we get

(A1) $c = remit$

(A2) $\tau_2 = \Sigma y\colon \{z:\tau_1 \mid z \equiv x\}.\{z:\epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}$

(A3) $\mathsf{Includes}\ \Gamma\ \tau_1\ \iota$

There is only evaluation rule E-Remit that applies to c.

(A4) $\iota \in dom(H)$

(A5) $\mathcal{HT}(\iota) = \eta$

(A6) $serialize_\eta(H(\iota)) = bv$

(A7) $I' = I,\ O' = O@bv,\ H' = H,\ c' = skip$

Let $\Gamma' = \Gamma, x:\tau_1$ and $\mathcal{E}' = \mathcal{E}[x \mapsto (I,O,H)]$.

Let $\tau_1' = \Sigma y\colon \{z:\tau_1 \mid z \equiv x\}.\{z:\epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}$ and

$\tau_2' = \{v : \Sigma y\colon \{z:\tau_1 \mid z \equiv x\}.\{z:\epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\} \mid v \equiv x'\}$.

(1) follows by T-Skip and (2) follows by assumptions (C) and (D) and Lemma A.30.

(3) and (4) are immediate.

For (5) we have to show that $(I, O@bv, H) \vDash_{\mathcal{E}[x\mapsto(I,O,H)]} \Sigma y\colon \{z:\tau_1 \mid z \equiv x\}.\{z:\epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}$. By Ent-Sigma, we must show that

(5.1) $(I,O,H) \vDash_{\mathcal{E}[x\mapsto(I,O,H)]} \{z:\tau_1 \mid z \equiv x\}$ and

(5.2) $(\langle\rangle, bv, [\,]) \vDash_{\mathcal{E}[x\mapsto(I,O,H),y\mapsto(I,O,H)]} \{z:\epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}$

For (5.1) we must show by Ent-Refine that $(I,O,H) \vDash_{\mathcal{E}[x\mapsto(I,O,H)]} \tau_1$, which follows by assumption (D) and the fact that x is not free in τ_1. We must also show that $[\![z \equiv x]\!]_{\mathcal{E}[x\mapsto(I,O,H),z\mapsto(I,O,H)]}$, which follows by the semantics of formulae and by reflexivity.

To show (5.2), by Ent-Refine, we must show that

$$(\langle\rangle, bv, [\,]) \vDash_{\mathcal{E}[x\mapsto(I,O,H),y\mapsto(I,O,H)]} \epsilon$$

and that

$$[\![z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota]\!]_{\mathcal{E}[x\mapsto(I,O,H),y\mapsto(I,O,H),z\mapsto(\langle\rangle,bv,[])]} = \mathsf{true}$$

The first follows after unfolding the definition of ϵ by Ent-Top, Ent-Refine and the semantics of formulae. The second follows by the semantics of formulae and (A6).

(6) follows by

$$\begin{aligned}
&[\![\tau_2]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}\\
=&[\![\Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}\\
=&\{h_1 \mathbin{++} h_2 \mid h_1 \in [\![\{z : \tau_1 \mid z \equiv x\}]\!]_{\mathcal{E}[x\mapsto(I,O,H)]} \wedge\\
&\qquad h_2 \in [\![\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}]\!]_{\mathcal{E}[x\mapsto(I,O,H),y\mapsto h_1]}\}\\
=&\{(I,O,H) \mathbin{++} (\langle\rangle, bv, [])\}\\
=&\{(I, O@bv, H)\}\\
=&\{h \mid h \in [\![\Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\\
&\qquad \{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge\\
&\qquad\qquad z.pkt_{out} = x.\iota\}]\!]_{\mathcal{E}[x\mapsto(I,O,H),x'\mapsto(I,O@bv,H)]} \wedge\\
&\qquad [\![v \equiv x']\!]_{\mathcal{E}[x\mapsto(I,O,H),x'\mapsto(I,O@bv,H),v\mapsto h]}\}\\
=&[\![\{v : \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\\
&\qquad \{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\} \mid\\
&\quad v \equiv x'\}]\!]_{\mathcal{E}[x\mapsto(I,O,H),x'\mapsto(I,O@bv,H)]}\\
=&[\![\tau_2']\!]_{\mathcal{E}'[x'\mapsto(I',O',H')]}
\end{aligned}$$

***Case* T-Remove:**

By inversion of rule T-Remove, we get

(A1) $c = remove(\iota)$

(A2) $\tau_2 = \{y : \top \mid \varphi_\iota \wedge \varphi_{pkt} \wedge \neg y.\iota.valid\}$

Only evaluation rule E-Remove applies to c.

(A3) $c' = skip$

(A4) $I' = I$ and $O' = O$ and $H' = H \setminus \iota$

Let $\Gamma' = \Gamma, x : \tau_1$ and $\mathcal{E}' = \mathcal{E}[x \mapsto (I, O, H)]$.

Let $\tau_1' = \tau_2$ and let $\tau_2' = \{z : \tau_2 \mid z \equiv x'\}$.

(1) follows by T-Skip. (2) follows by assumptions (C) and (D) and Lemma A.30. (3) and (4) are immediate.

To show (5), we must show that $(I, O, H \setminus \iota) \vDash_{\mathcal{E}[x\mapsto(I,O,H)]} \{y : \top \mid \varphi_\iota \wedge \varphi_{pkt} \wedge \neg y.\iota.valid\}$, which follows by Ent-Refine, Ent-Top and the semantics of formulae and expressions.

(6) follows by

$$
\begin{aligned}
&[\![\tau_2']\!]_{\mathcal{E}'[x' \mapsto (I', O', H')]} \\
=&[\![\{z : \{y : \top \mid \varphi_\iota \wedge \varphi_{pkt} \wedge \neg y.\iota.valid\} \mid z \equiv x'\}]\!]_{\mathcal{E}[x \mapsto (I,O,H), x' \mapsto (I,O,H \setminus \iota)]} \\
=&\{h \mid h \in [\![\{y : \top \mid \varphi_\iota \wedge \varphi_{pkt} \wedge \neg y.\iota.valid\}]\!]_{\mathcal{E}[x \mapsto (I,O,H), x' \mapsto (I,O,H \setminus \iota)]} \wedge \\
&\quad [\![z \equiv x']\!]\mathcal{E}[x \mapsto (I,O,H), x' \mapsto (I,O,H \setminus \iota), z \mapsto h]\} \\
=&\{(I, O, H \setminus \iota)\} \\
=&[\![\{y : \top \mid \varphi_\iota \wedge \varphi_{pkt} \wedge \neg y.\iota.valid\}]\!]_{\mathcal{E}[x \mapsto (I,O,H)]}
\end{aligned}
$$

***Case* T-Reset:**

By inversion of rule T-Reset, we get

(A1) $c = reset$

(A2) $\tau_2 = \Sigma y : \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}.\{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{in}\}$

There is only one evaluation rule that applies to c, E-Reset.

(A3) $c' = skip$

(A4) $I' = O@I$, $O' = \langle\rangle$ and $H' = []$

Let $\mathcal{E}' = \mathcal{E}[x \mapsto (I, O, H)]$ and $\Gamma' = \Gamma, x : \tau_1$.

Let $\tau_1' = \Sigma y : \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}.\{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{in}\}$

Let $\tau_2' = \{v : \Sigma y : \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}.\{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{in}\} \mid v \equiv x'\}$

(1) follows by T-Skip and (2) follows by assumptions (C) and (D) and Lemma A.30.

(3) and (4) are immediate.

To show (5), we must show that

$$
(O@I, \langle\rangle, []) \vDash_{\mathcal{E}[x \mapsto (I,O,H)]} \Sigma y : \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}. \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{in}\}
$$

By Ent-Sigma, we must show that

$$
(O, \langle\rangle, []) \vDash_{\mathcal{E}[x \mapsto (I,O,H)]} \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}
$$

and

$$
(I, \langle\rangle, []) \vDash_{\mathcal{E}[x \mapsto (I,O,H), y \mapsto (O, \langle\rangle, [])]} \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{in}\}
$$

Both follow after unfolding the definition of ϵ by Ent-Refine, Ent-Top and the semantics of formulae.

(6) follows by

$$
\begin{aligned}
&[\![\Sigma y\colon \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}.\\
&\qquad \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}\\
=&\{h_1 \mathbin{++} h_2 \mid h_1 \in [\![\{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}_{\mathcal{E}[x\mapsto(I,O,H)]}]\!] \wedge\\
&\qquad h_2 \in [\![\{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}]\!]_{\mathcal{E}[x\mapsto(I,O,H),y\mapsto h_1]}\}\\
=&\{(O, \langle\rangle, [\,]) \mathbin{++} (I, \langle\rangle, [\,])\}\\
=&\{(O@I, \langle\rangle, [\,])\}\\
=&[\![\{v : \Sigma y\colon \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}.\\
&\qquad \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{in}\} \mid\\
&\quad v \equiv x'\}]\!]_{\mathcal{E}[x\mapsto(O@I,\langle\rangle,[\,]),x'\mapsto(O@I,\langle\rangle,[\,])]}
\end{aligned}
$$

***Case* T-Seq:**

By inversion of rule T-Seq, we get

(A1) $c = c1; c2$

(A2) $\Gamma \vdash c_1 : (x : \tau_1) \to \tau_{12}$

(A3) $\Gamma, x : \tau_1 \vdash c_2 : (y : \tau_{12}) \to \tau_{22}$

(A4) $\tau_2 = \tau_{22}[y \mapsto \tau_{12}]$

***Subcase* E-Seq:**

(A5) $c_1 = skip$

(A6) $c' = c_2$

By E-Seq, $I' = I$, $O' = O$, $H' = H$.

Let $\Gamma' = \Gamma, x : \tau_1$, $\mathcal{E}' = \mathcal{E}[x \mapsto (I, O, H)]$ and $x' = y$.

Let $\tau_1' = \tau_{12}$ and $\tau_2' = \tau_{22}$.

(1) follows by (A3), (2) follows by assumptions (C) and (D) and Lemma A.30. (3) and (4) are immediate.

To show (5), we must show that $(I, O, H) \vDash_{\mathcal{E}[x\mapsto(I,O,H)]} \{z : \tau_1 \mid z \equiv x\}$ holds. By Ent-Refine, we must show that $(I, O, H) \vDash_{\mathcal{E}[x\mapsto(I,O,H)]} \tau_1$ and $[\![z \equiv x]\!]_{\mathcal{E}[x\mapsto(I,O,H),z\mapsto(I,O,H)]}$, which follows by assumption (D) and the semantics of formulae.

For (6), we must show that

$$
\begin{aligned}
&[\![\tau_2']\!]_{\mathcal{E}'[x\mapsto(I',O',H')]} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}\\
\Leftrightarrow&[\![\tau_{22}]\!]_{\mathcal{E}[x\mapsto(I,O,H),y\mapsto(I,O,H)]} \subseteq [\![\tau_{22}[y \mapsto \{z : \tau_1 \mid z \equiv x\}]]\!]_{\mathcal{E}[x\mapsto(I,O,H)]}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\tau_{22}[y \mapsto \tau_{12}]]\!]_{\mathcal{E}[x\mapsto(I,O.H)]}\\
=&\{h_{22} | h_{12} \in [\![\{z : \tau_1 \mid z \equiv x\}]\!]_{\mathcal{E}[x\mapsto(I,O,H)]} \wedge h_{22} \in [\![\tau_{22}]\!]_{\mathcal{E}[x\mapsto(I,O,H),y\mapsto h_{12}]}\}\\
=&\{h_{22} | h_{12} = (I, O, H) \wedge h_{22} \in [\![\tau_{22}]\!]_{\mathcal{E}[x\mapsto(I,O,H),y\mapsto h_{12}]}\}\\
=&\{h_{22} | h_{22} \in [\![\tau_{22}]\!]_{\mathcal{E}[x\mapsto(I,O,H),y\mapsto(I,O,H)]}\}\\
=&[\![\tau_{22}]\!]_{\mathcal{E}[x\mapsto(I,O,H),y\mapsto(I,O,H)]}
\end{aligned}
$$

This shows (6) and concludes this subcase.

***Subcase* E-Seq1:**

(A7) $c' = c_1'; c_2$

(A8) $\langle I, O, H, c_1 \rangle \to \langle I', O', H', c_1' \rangle$

By IH with (A2), (A7), (C) and (D), there exists $\Gamma', \mathcal{E}', \tau_1', \tau_{12}', x'$, such that,

(A9) $\Gamma' \vdash c_1' : (x' : \tau_1') \to \tau_{12}'$ where

(A10) $\mathcal{E}' \vDash \Gamma'$

(A11) $\Gamma \subseteq \Gamma'$

(A12) $\mathcal{E} \subseteq \mathcal{E}'$

(A13) $(I', O', H') \vDash_{\mathcal{E}'} \tau_1'$

(A14) $[\![\tau_{12}']\!]_{\mathcal{E}'[x' \mapsto (I', O', H')]} \subseteq [\![\tau_{12}]\!]_{\mathcal{E}[x \mapsto (I, O, H)]}$

(1) follows by T-Seq, if we can show that there exists some τ_{22}', such that $\Gamma', x' : \tau_1' \vdash c_2 : (y : \tau_{12}') \to \tau_{22}'$ where $\tau_2' = \tau_{22}'[y \mapsto \tau_{12}']$:

$$\text{T-Seq}\quad \frac{\Gamma' \vdash c_1' : (x : \tau_1') \to \tau_{12}' \qquad \Gamma', x : \tau_1' \vdash c_2 : (y : \tau_{12}') \to \tau_{22}'}{\Gamma' \vdash c_1'; c_2 : (x : \tau_1') \to \tau_{22}'[y \mapsto \tau_{12}']}$$

By Lemma A.49 with (A3) and (A13), there exists some τ_{22}' such that

(A15) $\Gamma', x' : \tau_1' \vdash c_2 : (y : \tau_1') \to \tau_{22}'$

(A16) $\forall h' \in [\![\tau_{12}']\!]_{\mathcal{E}'[x' \mapsto (I', O', H')]} . [\![\tau_{22}']\!]_{\mathcal{E}'[x' \mapsto (I', O', H'), y \mapsto h']} \subseteq$
$[\![\tau_{22}]\!]_{\mathcal{E}[x \mapsto (I, O, H), y \mapsto h']}$

(2) follows by (A10) and (3) follows by (A11), (4) follows by (A12) and (5) follows by (A13).

(6) follows by

$$\begin{aligned}
& [\![\tau_2']\!]_{\mathcal{E}'[x' \mapsto (I', O', H')]} \subseteq [\![\tau_2]\!]_{\mathcal{E}[x \mapsto (I, O, H)]} \\
\Leftrightarrow & [\![\tau_{22}'[y \mapsto \tau_{12}']]\!]_{\mathcal{E}'[x \mapsto (I', O', H')]} \subseteq [\![\tau_{22}[y \mapsto \tau_{12}]]\!]_{\mathcal{E}[x \mapsto (I, O, H)]} \\
\Leftrightarrow & \bigcup_{h' \in [\![\tau_{12}']\!]_{\mathcal{E}'[x \mapsto (I', O', H')]}} [\![\tau_{22}']\!]_{\mathcal{E}'[x \mapsto (I', O', H'), y \mapsto h']} \subseteq \\
& \bigcup_{h \in [\![\tau_{12}]\!]_{\mathcal{E}[x \mapsto (I, O, H)]}} [\![\tau_{22}]\!]_{\mathcal{E}[x \mapsto (I, O, H), y \mapsto h]}
\end{aligned}$$

and by (A14), (A16) and the semantics of heap types.

***Case* T-Skip:**

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

***Case* T-Sub:**

(A1) $\Gamma \vdash c : (x : \tau_3) \to \tau_4$

(A2) $\Gamma \vdash \tau_1 <: \tau_3$

(A3) $\Gamma, x : \tau_1 \vdash \tau_4 <: \tau_2$

By Lemma A.29 with assumptions (C), (D) and (A2),

(A4) $(I, O, H) \vDash_{\mathcal{E}} \tau_3$.

By IH with (A1), (B), (C) and (A4), there exists $\Gamma', \mathcal{E}', \tau_3', \tau_4', x'$ such that

(A5) $\Gamma' \vdash c : (x' : \tau_3') \to \tau_4'$

(A6) $\mathcal{E}' \vDash \Gamma'$

(A7) $\Gamma \subseteq \Gamma'$

(A8) $\mathcal{E} \subseteq \mathcal{E}'$

(A9) $(I', O', H') \vDash_{\mathcal{E}'} \tau_3'$

(A10) $[\![\tau_4']\!]_{\mathcal{E}'[x' \mapsto (I', O', H')]} \subseteq [\![\tau_4]\!]_{\mathcal{E}[x \mapsto (I, O, H)]}$

Let $\tau_1' = \tau_3'$ and $\tau_2' = \tau_4'$. (1) follows by (A5), (2) follows by assumption (A6), (3) follows by assumption (A7), (4) follows by assumption (A8), (5) follows by assumption (A9) and (6) follows by (A3) and (A10).

□

A.2.2 Algorithmic Typing Correctness

Lemma A.55 (Subtype Reflexivity). *For all subtyping contexts* Γ *and heap types* τ, $\Gamma \vdash \tau <: \tau$.

Proof. Immediate. □

Lemma A.56 (Subtype Transitivity). *If* $\Gamma \vdash \tau_1 <: \tau_2$, *and* $\Gamma \vdash \tau_2 <: \tau_3$, *then* $\Gamma \vdash \tau_1 <: \tau_3$.

Proof. Assume $\Gamma \vdash \tau_1 <: \tau_2$ and also assume $\Gamma \vdash \tau_2 <: \tau_3$. Let $\mathcal{E} \vDash \Gamma$ and $h \in [\![\tau_1]\!]_{\mathcal{E}}$ be arbitrary. By the first assumption $h \in [\![\tau_2]\!]_{\mathcal{E}}$. By the second assumption $h \in [\![\tau_3]\!]_{\mathcal{E}}$. □

Lemma A.57 (Environment Entails Subtype). *If* $\Gamma \vdash \tau_1 <: \tau_3$, *and* $\mathcal{E} \vDash \Gamma, x : \tau_3$, *then* $\mathcal{E} \vDash \Gamma, x : \tau_1$.

Proof. Let $\Gamma \vdash \tau_1 <: \tau_3$ and $\mathcal{E} \vDash \Gamma, (x : \tau_3)$. Let $\mathcal{E} \vDash \Gamma, (x : \tau_1)$. We can write $\mathcal{E} = \mathcal{E}'[x \mapsto h_1]$, such that $\mathcal{E}' \vDash \Gamma$, and $h_1 \in [\![\tau_1]\!]_{\mathcal{E}'}$. The definition of subtyping gives $h_1 \in [\![\tau_3]\!]_{\mathcal{E}'}$. The result follows by definition of entailment. □

Lemma A.58 (Context Strengthening). *If* $\Gamma \vdash \tau_1 <: \tau_3$ *and* $\Gamma, x : \tau_3 \vdash \tau_2 <: \tau_4$ *then* $\Gamma, x : \tau_1 \vdash \tau_2 <: \tau_4$.

Proof. Assume $\Gamma \vdash \tau_1 <: \tau_3$ and further assume $\Gamma, x : \tau_3 \vdash \tau_2 <: \tau_4$. Let $\mathcal{E} \vDash \Gamma, x : \tau_1$. By Lemma A.57 and the first assumption, $\mathcal{E} \vDash \Gamma, (x : \tau_3)$. Let $h \in [\![\tau_2]\!]_{\mathcal{E}}$, the second assumption gives that $h \in [\![\tau_2]\!]_{\mathcal{E}}$, and we're done. □

Lemma A.59 (Packet Bound Subtype). $\Gamma \vdash \mathsf{sizeof}_{pkt_{in}}(\tau) \geq N$ *iff* $\Gamma \vdash \tau <: \{x : \top \mid |x.pkt_{in}| \geq \mathsf{sizeof}(\iota)\}$.

Proof. We show each direction separately.

(⇒) Assume $\Gamma \vdash \mathsf{sizeof}_{pkt_{in}}(\tau) \geq N$. Let $\mathcal{E} \vDash \Gamma$ and $h \in [\![\tau]\!]_{\mathcal{E}}$ be arbitrary. By definition, $|h(pkt_{in})| \geq N$. By the definition of subtyping, it suffices to show $h \in [\![\{x : \top \mid |x.pkt_{in}| \geq N\}]\!]_{\mathcal{E}}$. By definition, $[\![\{x : \top \mid |x.pkt_{in}| \geq N\}]\!]_{\mathcal{E}} = \{h \mid h \in \mathcal{H} \wedge h(pkt_{in}) \geq N\}$, which concludes this case.

($\Leftarrow$) Assume $\Gamma \vdash \tau <: \{x : \top \mid |x.pkt_{in}| \geq N\}$. We have to show that $\Gamma \vdash \mathsf{sizeof}_{pkt_{in}}(\tau) \geq N$. Let $\mathcal{E} \vDash \Gamma$ and $h \in [\![\tau]\!]_{\mathcal{E}}$ be arbitrary. By the definition of subtyping, $h \in [\![\{x : \top \mid |x.pkt_{in}| \geq N\}]\!]_{\mathcal{E}}$. By definition of the semantics, we can conclude $h(pkt_{in}) \geq N$.

□

Lemma A.60 (Chomp Subtype). *If x not free in τ and τ', and $\Gamma \vdash \mathsf{sizeof}_{pkt_{in}}(\tau) \geq \mathsf{sizeof}(\iota)$ and $\Gamma \vdash \tau <: \tau'$, then $\Gamma, x : \{y : \iota \mid y.pkt_{in} = y.pkt_{out} = \langle\rangle\} \vdash \mathsf{chomp}(\tau, \iota, x) <: \mathsf{chomp}(\tau', \iota, x)$.*

Proof. Given some heap $h' \in [\![\mathsf{chomp}(\tau', \iota, x)]\!]_{\mathcal{E}'}$. By Lemma A.47 there exists some $\mathcal{E}$ and $h'' \in [\![\tau']\!]_{\mathcal{E}}$, such that $\mathcal{E}' = \mathcal{E}[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto h''(pkt_{in})[\mathsf{o}{:}\mathsf{sizeof}(\iota)]])]$ and $h' = \mathsf{chomp}^{\Downarrow}(h'', \mathsf{sizeof}(\iota))$. By assumption $\Gamma \vdash \tau <: \tau'$, we also know that $h'' \in [\![\tau]\!]_{\mathcal{E}}$.

By Lemma A.44, we know that there exists $h \in [\![\mathsf{chomp}(\tau, \iota, x)]\!]_{\mathcal{E}''}$ such that $h = \mathsf{chomp}^{\Downarrow}(h'', \mathsf{sizeof}(\iota))$ and $\mathcal{E}'' = \mathcal{E}[x \mapsto (\langle\rangle, \langle\rangle, [\iota \mapsto h''(pkt_{in})[\mathsf{o} : \mathsf{sizeof}(\iota)]])]$. From $h' = \mathsf{chomp}^{\Downarrow}(h'', \mathsf{sizeof}(\iota))$ and $h = \mathsf{chomp}^{\Downarrow}(h'', \mathsf{sizeof}(\iota))$ follows by the transitivity of equality that $h' = h$. By the fact that $\mathcal{E}' = \mathcal{E}''$ follows that for every heap $h' \in [\![\mathsf{chomp}(\tau', \iota, x)]\!]_{\mathcal{E}'}$ also holds that $h' \in [\![\mathsf{chomp}(\tau, \iota, y)]\!]_{\mathcal{E}''}$. □

Lemma A.61 (Refinement Subtype). *If $\Gamma \vdash \tau' <: \tau$, then $\Gamma \vdash \{x : \tau' \mid \varphi\} <: \{x : \tau \mid \varphi\}$.*

Proof. Let $\mathcal{E} \vdash \Gamma$ and $h \in [\![\{x : \tau' \mid \varphi\}]\!]_{\mathcal{E}}$. Then $h \in [\![\tau']\!]_{\mathcal{E}}$, and $[\![\varphi]\!]_{\mathcal{E}[x \mapsto h]} = \mathit{true}$. By assumption $\Gamma \vdash \tau' <: \tau$, $h \in [\![\tau]\!]_{\mathcal{E}}$. Conclude $h \in [\![\{x : \tau \mid \varphi\}]\!]_{\mathcal{E}}$ by definition. The result follows. □

Lemma A.62 (Sigma Left-Subtype). *If $\Gamma \vdash \tau_1' <: \tau_1$ then $\Gamma \vdash \Sigma x{:}\, \tau_1'.\tau_2 <: \Sigma x{:}\, \tau_1.\tau_2$.*

Proof. Let $\mathcal{E} \vDash \Gamma$, and $h \in [\![\Sigma x : \tau_1'.\tau_2]\!]_{\mathcal{E}}$. By definition of the semantics, $h = h_1 + + h_2$,where $h_1 \in [\![\tau_1']\!]_{\mathcal{E}}$ and $h_2 \in [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h]}$. By assumption $\Gamma \vdash \tau_1' <: \tau_1$ follows, $h_1 \in [\![\tau_1]\!]_{\mathcal{E}}$. By the definition of the semantics, $h \in [\![\Sigma x{:}\, \tau_1.\tau_2]\!]_{\mathcal{E}}$. The result follows. □

Lemma A.63 (Sigma Right-Subtype). *If $\Gamma, x : \tau_1 \vdash \tau_2' <: \tau_2$ then $\Gamma \vdash \Sigma x{:}\, \tau_1.\tau_2' <: \Sigma x{:}\, \tau_1.\tau_2$.*

Proof. Let $\mathcal{E} \vDash \Gamma$ and $h \in [\![\Sigma x{:}\, \tau_1.\tau_2']\!]_{\mathcal{E}}$. By definition, $h = h_1 ++ h_2$ such that $h_1 \in [\![\tau_1]\!]_{\mathcal{E}}$, and $h_2 \in [\![\tau_2']\!]_{\mathcal{E}[x \mapsto h_1]}$. Notice that $\mathcal{E}[x \mapsto h_1] \vDash \Gamma, (x : \tau_1)$, so by assumption $\Gamma, x : \tau_1 \vdash \tau_2' <: \tau_2$ follows $h_2 \in [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h_1]}$. By the definition of the semantics $h \in [\![\Sigma x{:}\, \tau_1.\tau_2]\!]_{\mathcal{E}}$. The result follows. □

Lemma A.64 (Substitution Subtype). *If $\Gamma \vdash \tau_1' <: \tau_1$ and $\Gamma, (x : \tau_1') \vdash \tau_2' <: \tau_2$, then $\Gamma \vdash \tau_2'[x \mapsto \tau_1'] <: \tau_2[x \mapsto \tau_1]$.*

Proof. Let $\mathcal{E} \vDash \Gamma$ and $h_2 \in [\![\tau_2'[x \mapsto \tau_1']]\!]_{\mathcal{E}}$. Then we know $h_1 \in [\![\tau_1']\!]_{\mathcal{E}}$ and $h_2 \in [\![\tau_2']\!]_{\mathcal{E}[x \mapsto h_1]}$. Assumption $\Gamma \vdash \tau_1' <: \tau_1$ tells us that $h_1 \in [\![\tau_1]\!]_{\mathcal{E}}$. Notice that $\mathcal{E}[x \mapsto h_1] \vDash \Gamma, (x : \tau_1')$. Assumption $\Gamma, (x : \tau_1) \vdash \tau_2' <: \tau_2$ gives $h_2 \in [\![\tau_2]\!]_{\mathcal{E}[x \mapsto h_1]}$. By the definition of the semantics of heap types, $h_2 \in [\![\tau_2[x \mapsto \tau_1]]\!]_{\mathcal{E}}$. □

Lemma A.65 (Choice Subtype). *If $\Gamma \vdash \tau_1' <: \tau_1$, and $\Gamma \vdash \tau_2' <: \tau_2$, then $\Gamma \vdash \tau_1' + \tau_2' <: \tau_1 + \tau_2$*

Proof. Let $\mathcal{E} \vDash \Gamma$. Let $h \in [\![\tau_1' + \tau_2']\!]_{\mathcal{E}}$. By semantics of heap types, either $h \in [\![\tau_1']\!]_{\mathcal{E}}$ or $h \in [\![\tau_2']\!]_{\mathcal{E}}$.

Subcase $h \in [\![\tau_1']\!]_{\mathcal{E}}$**:** By assumption $\Gamma \vdash \tau_1' <: \tau_1$ it also holds that $h \in [\![\tau_1]\!]_{\mathcal{E}}$ and we can conclude that $h \in [\![\tau_1 + \tau_2]\!]_{\mathcal{E}}$.

Subcase $h \in [\![\tau_2']\!]_{\mathcal{E}}$**:** By assumption $\Gamma \vdash \tau_2' <: \tau_2$ it also holds that $h \in [\![\tau_2]\!]_{\mathcal{E}}$ and we can conclude that $h \in [\![\tau_1 + \tau_2]\!]_{\mathcal{E}}$.

□

Lemma A.66 (Context-Bound Refinement Subtype)**.** *If* heap *is the only free binder in* φ*, and* $\Gamma, x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\} \vdash \tau_2' <: \tau_2$ *then* $\Gamma, x : \tau_1 \vdash \{y : \tau_2' \mid \varphi[x/\mathsf{heap}]\} <: \{y : \tau_2 \mid \varphi[x/\mathsf{heap}]\}$.

Proof. Let $\mathcal{E} \vDash \Gamma, x : \tau_1$. We can write this as $\mathcal{E} = \mathcal{E}'[x \mapsto h_1]$, where $h_1 \in [\![\tau_1]\!]_{\mathcal{E}}$. Let $h_2 \in [\![\{y : \tau_2' \mid \varphi[x/\mathsf{heap}]\}]\!]_{\mathcal{E}}$. Then $h_2 \in [\![\tau_2']\!]_{\mathcal{E}}$ and $[\![\varphi[x/\mathsf{heap}]]\!]_{\mathcal{E}[y \mapsto h_2]} = \mathit{true}$. Compute as follows, recalling that heap is the only free binder in φ:

$$
\begin{aligned}
&[\![\varphi[x/\mathsf{heap}]]\!]_{\mathcal{E}[y \mapsto h_2]} \\
=&[\![\varphi[x/\mathsf{heap}]]\!]_{\mathcal{E}} \\
=&[\![\varphi[x/\mathsf{heap}]]\!]_{\mathcal{E}'[x \mapsto h_1]} \\
=&[\![\varphi[y/\mathsf{heap}]]\!]_{\mathcal{E}'[y \mapsto h_1]}
\end{aligned}
$$

Together with assumption $h_1 \in [\![\tau_1]\!]_{\mathcal{E}}$, we get $h_1 \in [\![\{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}]\!]_{\mathcal{E}'}$, and thus $\mathcal{E}'[x \mapsto h_1] \vDash \Gamma, x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}$.

With assumption $\Gamma, x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\} \vdash \tau_2' <: \tau_2$, we can conclude that $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$. Since we already have that $[\![\varphi[x/\mathsf{heap}]]\!]_{\mathcal{E}[y \mapsto h_2]} = \mathit{true}$, it follows that $h_2 \in [\![\{y : \tau_2 \mid \varphi[x/\mathsf{heap}]\}]\!]_{\mathcal{E}}$, which is what we wanted to show. □

Lemma A.67 (If Choice Subtype)**.** *If* heap *is the only free binder in* φ *and* $\Gamma, x : \{y : \tau_1' \mid \varphi[y/\mathsf{heap}]\} \vdash \tau_{12}' <: \tau_{12}$*, and* $\Gamma, x : \{y : \tau_1' \mid \neg\varphi[y/\mathsf{heap}]\} \vdash \tau_{22}' <: \tau_{22}$*, then* $\Gamma, x : \tau_1' \vdash \{y : \tau_{12}' \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22}' \mid \varphi[x/\mathsf{heap}]\} <: \{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\}$

Proof. By Lemmas A.66 and A.65. □

Lemma A.68 (Algorithmic Weakening)**.** *If* $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2$ *and variable* y *does not appear free in* τ_1 *or* τ_2*, then* $\Gamma, y : \tau \vdash c : (x : \tau_1) \rightsquigarrow \tau_2$ *for any heap type* τ*.*

Proof. By induction on the typing derivation. □

Lemma A.69 (Typing Context Subtype)**.** *If* $\Gamma, x : \tau_1 \vdash c : (y : \tau_{12}) \rightsquigarrow \tau_{22}$ *and* $\Gamma \vdash \tau_1' <: \tau_1$*, then* $\Gamma, x : \tau_1' \vdash c : (y : \tau_{12}) \rightsquigarrow \tau_{22}$.

Proof. If x is not free in τ_{12} or τ_{22}, the result follows from Lemma A.68. Otherwise we proceed by induction on the typing derivation. We refer to the general assumptions as follows:

(A) $\Gamma, x : \tau_1 \vdash c : (y : \tau_{12}) \rightsquigarrow \tau_{22}$

(B) $\Gamma \vdash \tau_1' <: \tau_1$

Case **T-Extract-Algo:** By inversion of T-Extract-Algo, we know

(A1) $\Gamma, x : \tau_1 \vdash \mathit{extract}(\iota) : (y : \tau_{12}) \rightsquigarrow \tau_{22}$

(A2) $\tau_{22} = \Sigma z\colon \{v : \iota \mid \varphi_1\}.\{v : \mathsf{chomp}(\tau_{12}, \iota, z) \mid \varphi_2\}$

(A3) $\Gamma, x : \tau_1 \vdash \tau_{12} <: \{z : \top \mid |z.pkt_{in}| \geq \mathsf{sizeof}(\iota)\}$

By Lemma A.58 applied to (A3) and (B) follows

(A4) $\Gamma, x : \tau_1' \vdash \tau_{12} <: \{z : \top \mid |z.pkt_{in}| \geq \mathsf{sizeof}(\iota)\}$

The result follows by T-Extract-Algo.

***Case* T-Seq-Algo:** By inversion of T-Seq-Algo, we know

(A1) $\Gamma, x : \tau_1 \vdash c_1; c_2 : (y : \tau_{12}) \rightsquigarrow \tau_{22}$

(A2) $\Gamma, x : \tau_1 \vdash c_1 : (y : \tau_{12}) \rightsquigarrow \tau_{12}'$

(A3) $\Gamma, x : \tau_1, y : \tau_{12} \vdash c_2 : (z : \tau_{12}') \rightsquigarrow \tau_{22}'$

(A4) $\tau_{22} = \tau_{22}'[z \mapsto \tau_{12}']$

By IH applied to (A2) and (B) follows

(A5) $\Gamma, x : \tau_1' \vdash c_1 : (y : \tau_{12}) \rightsquigarrow \tau_{12}'$

By IH applied to (A3) and (B) follows

(A6) $\Gamma, x : \tau_1', y : \tau_{12} \vdash c_2 : (z : \tau_{12}') \rightsquigarrow \tau_{22}'$

The result follows by T-Seq-Algo with (A5) and (A6).

***Case* T-Skip-Algo:** The result immediately follows by T-Skip-Algo.

***Case* T-Remit-Algo:** By inversion of T-Remit-Algo, we know

(A1) $\Gamma, x : \tau_1 \vdash \tau_{12} <: \iota_{\approx}$

By Lemma A.58 applied to (A1) and (B) follows

(A2) $\Gamma, x : \tau_1' \vdash \tau_{12} <: \iota_{\approx}$

The result follows by T-Remit-Algo.

***Case* T-Reset-Algo:** The result immediately follows by T-Reset-Algo.

***Case* T-Ascribe-Algo:** By inversion of T-Ascribe-Algo, we know

(A1) $\Gamma, x : \tau_1 \vdash c_0 \; as \; (y : \hat{\tau}_{12}) \to \tau_{22} : (y : \tau_{12}) \rightsquigarrow \tau_{22}$

(A2) $\Gamma, x : \tau_1 \vdash c_0 : (y : \hat{\tau}_{12}) \rightsquigarrow \tau_{22}'$

(A3) $\Gamma, x : \tau_1 \vdash \tau_{12} <: \hat{\tau}_{12}$

(A4) $\Gamma, x : \tau_1, y : \hat{\tau}_{12} \vdash \tau_{22}' <: \tau_{22}$

By IH applied to (A2) and (B) follows

(A5) $\Gamma, x : \tau_1' \vdash c_0 : (y : \hat{\tau}_{12}) \rightsquigarrow \tau_{22}'$

By Lemma A.58 applied to (A3) and (B) follows

(A6) $\Gamma, x : \tau_1' \vdash \tau_{12} <: \hat{\tau}_{12}$

By Lemma A.58 applied to (A4) and (B) follows

(A7) $\Gamma, x : \tau_1', y : \hat{\tau}_{12} \vdash \tau_{22}' <: \tau_{22}$

The result follows by T-Ascribe-Algo with (A5), (A6) and (A7).

***Case* T-If-Algo:**

By inversion of T-If-Algo, we know

(A1) $\Gamma, x : \tau_1 \vdash c_1 : (y : \{z : \tau_{12} \mid \varphi[z/\mathsf{heap}]\}) \rightsquigarrow \tau_{12}'$

(A2) $\Gamma, x : \tau_1 \vdash c_1 : (y : \{z : \tau_{12} \mid \neg\varphi[z/\mathsf{heap}]\}) \rightsquigarrow \tau_{12}''$

(A3) $\tau_{22} = \{z : \tau_{12}' \mid \varphi[x/\mathsf{heap}]\} + \{z : \tau_{12}'' \mid \neg\varphi[x/\mathsf{heap}]\}$

(A4) $\Gamma, x : \tau_1; \tau_{12} \vdash \varphi : \mathbb{B}$

By IH applied to (A1) and (B) follows

(A5) $\Gamma, x : \tau_1' \vdash c_1 : (y : \{z : \tau_{12} \mid \varphi[z/\mathsf{heap}]\}) \rightsquigarrow \tau_{12}'$

By IH applied to (A2) and (B) follows

(A6) $\Gamma, x : \tau_1' \vdash c_1 : (y : \{z : \tau_{12} \mid \neg\varphi[z/\mathsf{heap}]\}) \rightsquigarrow \tau_{12}''$

Since $\Gamma, x : \tau_1; \tau_{12} \vdash \varphi : \mathbb{B}$, it also holds that $\Gamma, x : \tau_1'; \tau_{12} \vdash \varphi : \mathbb{B}$. The result then follows by T-If-Algo.

***Case* T-Mod-Algo:** By inversion of T-Mod-Algo, we know

(A1) $\Gamma, x : \tau_1 \vdash \tau_{12} <: \iota_{\approx}$

(A2) $\Gamma, x : \tau_1; \tau_{12} \vdash e : \mathsf{BV}$

By Lemma A.58 applied to (A1) and (B) follows

(A3) $\Gamma, x : \tau_1' \vdash \tau_{12} <: \iota_{\approx}$

Since $\Gamma, x : \tau_1; \tau_{12} \vdash e : \mathsf{BV}$, it also holds that $\Gamma, x : \tau_1'; \tau_{12} \vdash e : \mathsf{BV}$. The result follows by T-Mod-Algo.

***Case* T-Add-Algo:** By inversion of T-Add-Algo, we know

(A1) $\Gamma, x : \tau_1 \vdash \tau_{12} <: \{x : \top \mid \neg x.\iota.\mathit{valid}\}$

By Lemma A.58 applied to (A1) and (B) follows

(A2) $\Gamma, x : \tau_1' \vdash \tau_{12} <: \{x : \top \mid \neg x.\iota.\mathit{valid}\}$

The result follows by T-Add-Algo.

***Case* T-Remove-Algo:** By inversion of T-Remove-Algo, we know

(A1) $\Gamma, x : \tau_1 \vdash \tau_{12} <: \iota_{\approx}$

By Lemma A.58 applied to (A1) and (B) follows

(A2) $\Gamma, x : \tau_1' \vdash \tau_{12} <: \iota_{\approx}$

The result follows by T-Remove-Algo.

□

Lemma A.70 (Algorithmic Input Subtype). *If* $\Gamma \vdash \tau_1' <: \tau_1$ *and* $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2$ *such that* x *is not free in* τ_1 *or* τ_1'*, then there exists* 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. We refer to the general assumptions as follows:

(A) $\Gamma \vdash \tau_1' <: \tau_1$

(B) $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2$

(C) x not free in τ_1 or τ_1'

We refer to the proof goals as follows:

(1) $\Gamma \vdash c : (x : \tau_1') \rightsquigarrow \tau_2'$

(2) $\Gamma, x : \tau_1' \vdash \tau_2' <: \tau_2$

***Case* T-Extract-Algo:** By inversion of T-Extract-Algo, we know

(A1) $c = extract(\iota)$

(A2) $\Gamma \vdash \tau_1 <: \{x : \top \mid |x.pkt_{in}| \geq \mathsf{sizeof}(\iota)\}$

(A3) $\varphi_1 \triangleq z.pkt_{in} = z.pkt_{out} = \langle\rangle$

(A4) $\varphi_2 \triangleq y.\iota@z.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x$

(A5) $\tau_{21} \triangleq \{z : \iota \mid \varphi_1\}$

(A6) $\tau_{22} \triangleq \{z : \mathsf{chomp}(\tau_1, \iota, y) \mid \varphi_2\}$

(A7) $\tau_2 = \Sigma y : \tau_{21}.\tau_{22}$

By Lemma A.56 with (A) and (A2),

(A8) $\Gamma \vdash \tau_1' <: \{x : \top \mid |x.pkt_{in}| \geq \mathsf{sizeof}(\iota)\}$

By Lemma A.60 with (A2) and (A),

(A9) $\Gamma, y : \tau_{21} \vdash \mathsf{chomp}(\tau_1', \iota, y) <: \mathsf{chomp}(\tau_1, \iota, y)$

(1) follows by T-Extract-Algo. (2) follows by Lemmas A.61 and A.63 and (A9).

***Case* T-Seq-Algo:** By inversion of T-Seq-Algo, we know

(A1) $c = c_1; c_2$

(A2) $\Gamma \vdash c_1 : (x : \tau_1) \rightsquigarrow \tau_{12}$

(A3) $\Gamma, x : \tau_1 \vdash c_2 : (y : \tau_{12}) \rightsquigarrow \tau_{22}$

(A4) $\tau_2 = \tau_{22}[y \mapsto \tau_{12}]$

By IH applied to (A2) and (A), there is some τ_{12}' such that

(A5) $\Gamma \vdash c_1 : (x : \tau_1') \rightsquigarrow \tau_{12}'$, and

(A6) $\Gamma, x : \tau_1' \vdash \tau_{12}' <: \tau_{12}$

By Lemma A.69 with (A) and (A3),

(A7) $\Gamma, x : \tau_1' \vdash c_2 : (y : \tau_{12}) \rightsquigarrow \tau_{22}$

By IH applied to (A7) and (A6), there is some τ_{22}' such that

(A8) $\Gamma, x : \tau_1' \vdash c_2 : (y : \tau_{12}') \rightsquigarrow \tau_{22}'$, and

(A9) $\Gamma, x : \tau_1', y : \tau_{12}' \vdash \tau_{22}' <: \tau_{22}$

(1) follows by T-Seq-Algo with (A5) and (A8). To show (2), we just need to show that $\Gamma, x : \tau_1' \vdash \tau_{22}'[y \mapsto \tau_{12}'] <: \tau_{22}[y \mapsto \tau_{12}]$. This follows by Lemma A.64 applied to (A6) and (A9).

***Case* T-Skip-Algo:** Immediate by T-Skip-Algo.

***Case* T-Remit-Algo:** By inversion of T-Remit-Algo, we know

(A1) $\Gamma \vdash \tau_1 <: \iota_{\approx}$

(A2) $c = remit(\iota)$

(A3) $\tau_2 = \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}$

By Lemma A.56 with (A1) and (A) follows

(A4) $\Gamma \vdash \tau_1' <: \iota_{\approx}$

Let $\tau_2' = \Sigma y\colon \{z : \tau_1' \mid z \equiv x\}.\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}$. (1) follows by T-Remit-Algo. By Lemma A.58, and since x does not occur free in τ_1 or τ_1', we know

(A5) $\Gamma, x : \tau_1' \vdash \tau_1' <: \tau_1$

By Lemma A.61 we know

(A6) $\Gamma, x : \tau_1' \vdash \{z : \tau_1' \mid z \equiv x\} <: \{z : \tau_1 \mid z \equiv x\}$

By Lemma A.62 with $\tau_{22} = \{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota\}$ follows

(A7) $\Gamma, x : \tau_1' \vdash \Sigma y\colon \{z : \tau_1' \mid z \equiv x\}.\tau_{22} <: \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\tau_{22}$

This shows (2) and concludes this case.

***Case* T-Reset-Algo:** By inversion of T-Reset-Algo, we know

(A1) $\tau_2 = \Sigma y\colon \{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}\}.\{z : \epsilon \mid z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = z.pkt_{in}\}$

Let $\tau_2' = \tau_2$. (1) follows by T-Reset-Algo and (2) follows by Lemma A.55.

***Case* T-Ascribe-Algo:** By inversion of T-Ascribe-Algo, we know

(A1) $c = c_0\ as\ (x : \hat{\tau}_1) \to \tau_2$

(A2) $\Gamma \vdash c_0 : (x : \hat{\tau}_1) \rightsquigarrow \tau_c$

(A3) $\Gamma \vdash \tau_1 <: \hat{\tau}_1$

(A4) $\Gamma, x : \hat{\tau}_1 \vdash \tau_c <: \tau_2$

By Lemma A.56 applied to (A) and (A3) follows that

(A5) $\Gamma \vdash \tau_1' <: \hat{\tau}_1$

Let $\tau_2' = \tau_2$. (1) follows by T-Ascribe-Algo with (A2), (A4) and (A5). (2) follows by Lemma A.55.

***Case* T-If-Algo:** By inversion of T-If-Algo, we know

(A1) $c = \mathit{if}(\varphi)\ c_1\ \mathit{else}\ c_2$

(A2) $\Gamma; \tau_1 \vdash \varphi : \mathbb{B}$

(A3) $\Gamma \vdash c_1 : (x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}) \rightsquigarrow \tau_{12}$

(A4) $\Gamma \vdash c_2 : (x : \{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\}) \rightsquigarrow \tau_{22}$

(A5) $\tau_2 = \{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\}$

By (A) and Lemma A.61, we know

(A6) $\Gamma \vdash \{y : \tau_1' \mid \varphi[y/\mathsf{heap}]\} <: \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}$, and

(A7) $\Gamma \vdash \{y : \tau_1' \mid \neg\varphi[y/\mathsf{heap}]\} <: \{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\}$

By applying the IH to (A6) and (A3) we get τ_{12}' such that

(A8) $\Gamma \vdash c_1 : (x : \{y : \tau_1' \mid \varphi[y/\mathsf{heap}]\}) \rightsquigarrow \tau_{12}'$, and

(A9) $\Gamma, x : \{y : \tau_1' \mid \varphi[y/\mathsf{heap}]\} \vdash \tau_{12}' <: \tau_{12}$

By applying the IH to (A7) and (A4) we get τ_{22}' such that

(A10) $\Gamma \vdash c_2 : (x : \{y : \tau_1' \mid \neg\varphi[y/\mathsf{heap}]\}) \rightsquigarrow \tau_{22}'$, and

(A11) $\Gamma, x : \{y : \tau_1' \mid \neg\varphi[y/\mathsf{heap}]\} \vdash \tau_{22}' <: \tau_{22}$

From (A2) and (A), we can conclude that

(A12) $\Gamma; \tau_1' \vdash \varphi : \mathbb{B}$

Let $\tau_2' = \{y : \tau_{12}' \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22}' \mid \neg\varphi[x/\mathsf{heap}]\}$. (2) follows by Lemma A.67. (1) follows by T-If-Algo with (A8), (A10) and (A12).

***Case* T-Mod-Algo:** By inversion of T-Mod-Algo, we know

(A1) $\Gamma \vdash \tau_1 <: \iota_{\approx}$

(A2) $\Gamma; \tau_1 \vdash e : \mathsf{BV}$

(A3) $\tau_2 = \{y : \top \mid \varphi \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = e[x/\mathsf{heap}]\}$

By Lemma A.56 with (A) and (A1) follows

(A4) $\Gamma \vdash \tau_1' <: \iota_{\approx}$

From (A2) and (A), we can conclude

(A5) $\Gamma; \tau_1' \vdash e : \mathsf{BV}$

Let $\tau_2' = \tau_2$. (1) follows by T-Mod-Algo with (A4) and (A5). (2) follows by Lemma A.55.

***Case* T-Add-Algo:** By inversion of T-Add-Algo, we know

(A1) $\Gamma \vdash \tau_1 <: \{x : \top \mid \neg x.\iota.\mathit{valid}\}$

(A2) $\tau_2 = \Sigma y{:}\ \{z : \tau_1 \mid z \equiv x\}.\{z : \iota \mid z.\mathit{pkt}_{in} = z.\mathit{pkt}_{out} = \langle\rangle \wedge z.\iota = v\}$

By Lemma A.56 with (A1) and (A) follows

(A3) $\Gamma \vdash \tau_1' <: \{x : \top \mid \neg x.\iota.valid\}$

Let $\tau_2' = \Sigma y\colon \{z : \tau_1' \mid z \equiv x\}.\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle \wedge z.\iota = v\}$. (1) follows by T-Add-Algo. By Lemma A.58 with (A) and (C),

(A4) $\Gamma, x : \tau_1' \vdash \tau_1' <: \tau_1$

By Lemma A.61 follows

(A5) $\Gamma, x : \tau_1' \vdash \{z : \tau_1' \mid z \equiv x\} <: \{z : \tau_1 \mid z \equiv x\}$

(2) follows by Lemma A.62 with (A5).

***Case* T-Remove-Algo:** By inversion of T-Remove-Algo, we know

(A1) $\Gamma \vdash \tau_1 <: \iota_\approx$

By Lemma A.56 with (A1) and (A),

(A2) $\Gamma \vdash \tau_1' <: \iota_\approx$

Let $\tau_2' = \tau_2$. (1) follows by T-Remove-Algo with (A2). (2) follows by Lemma A.55.

□

Lemma A.71 (Includes Subtype)**.** $\mathsf{Includes}\ \Gamma\ \tau\ \iota \iff \Gamma \vdash \tau <: \iota_\approx$

Proof. Prove each direction separately

(⇒) Assume $\mathsf{Includes}\ \Gamma\ \tau\ \iota$. Let $\mathcal{E} \vDash \Gamma$ and $h \in [\![\tau]\!]_{\mathcal{E}}$. By definition of the inclusion relation, $\iota \in dom(h)$. By the definition of subtyping, it suffices to show $h \in [\![\iota_\approx]\!]_{\mathcal{E}}$. By definition, $[\![\iota_\approx]\!]_{\mathcal{E}} = \{h \mid \iota \in dom(h)\}$, and we're done.

(⇐) Assume $\Gamma \vdash \tau <: \iota_\approx$. Show $\mathsf{Includes}\ \Gamma\ \tau\ \iota$. To that end, let $\mathcal{E} \vDash \Gamma$ and $h \in [\![\tau]\!]_{\mathcal{E}}$ be arbitrary. By the definition of subtyping, $h \in [\![\iota_\approx]\!]_{\mathcal{E}}$. By definition of the semantics, conclude $\iota \in dom(h)$.

□

Lemma A.72 (Excludes Subtype)**.** $\mathsf{Excludes}\ \Gamma\ \tau\ \iota$ *iff* $\Gamma \vdash \tau <: \{x : \top \mid \neg x.\iota.valid\}$

Proof. Prove each direction separately

(⇒) Assume $\mathsf{Excludes}\ \Gamma\ \tau\ \iota$. Let $\mathcal{E} \vDash \Gamma$ and $h \in [\![\tau]\!]_{\mathcal{E}}$. By definition of the exclusion relation, $\iota \notin dom(h)$. By the definition of subtyping, it suffices to show $h \in [\![\{x : \top \mid \neg x.\iota.valid\}]\!]_{\mathcal{E}}$. By definition, $[\![\{x : \top \mid \neg x.\iota.valid\}]\!]_{\mathcal{E}} = \{h \mid \iota \notin dom(h)\}$, and we're done.

(⇐) Assume $\Gamma \vdash \tau <: \{x : \top \mid \neg x.\iota.valid\}$. Show $\mathsf{Excludes}\ \Gamma\ \tau\ \iota$. To that end, let $\mathcal{E} \vDash \Gamma$ and $h \in [\![\tau]\!]_{\mathcal{E}}$ be arbitrary. By the definition of subtyping, $h \in [\![\{x : \top \mid \neg x.\iota.valid\}]\!]_{\mathcal{E}}$. By definition of the semantics, conclude $\iota \notin dom(h)$.

□

Theorem A.73 (Algorithmic Typing Correctness)**.** *For all subtyping contexts* Γ*, commands* c*, variables* x*, heap types* 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. ($\Rightarrow$) Assume $\Gamma \vdash c : (x : \tau_1) \to \tau_2$. Proceed by induction on the typing derivation, leaving Γ general. We refer to the proof goals as follows:

(1) $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2'$

(2) $\Gamma, (x : \tau_1) \vdash \tau_2' <: \tau_2$

***Case* T-Extract:**

(A1) $c = extract(\iota)$

(A2) $\Gamma \vdash \mathsf{sizeof}_{pkt_{in}}(\tau_1) \geq \mathsf{sizeof}(i)$

(A3) $\varphi_1 \triangleq z.pkt_{in} = z.pkt_{out} = \langle\rangle$

(A4) $\varphi_2 \triangleq y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x$

(A5) $\tau_2 = \Sigma y\colon \{z : \iota \mid \varphi_1\}.\{z : \mathsf{chomp}(\tau_1, \iota, y) \mid \varphi_2\}$

The only algorithmic rule that applies to $extract(\iota)$ is T-Extract-Algo. Since $\Gamma \vdash \tau_1 <: \{x : \top \mid |x.pkt_{in}| \geq \mathsf{sizeof}(\iota)\}$ by (A2) and Lemma A.59, T-Extract-Algo produces τ_2' such that

(A6) $\varphi_1' \triangleq z.pkt_{in} = z.pkt_{out} = \langle\rangle$

(A7) $\varphi_2' \triangleq y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x$

(A8) $\tau_2' = \Sigma y\colon \{z : \iota \mid \varphi_1'\}.\{z : \mathsf{chomp}(\tau_1, \iota, y) \mid \varphi_2'\}$

which shows (1). (2) follows by Lemma A.55.

***Case* T-Seq:**

(A1) $c = c_1; c_2$

(A2) $\Gamma \vdash c_1 : (x : \tau_1) \to \tau_{12}$

(A3) $\Gamma, (x : \tau_1) \vdash c_2 : (y : \tau_{12}) \to \tau_{22}$

(A4) $\tau_2 = \tau_{22}[x \mapsto \tau_{12}]$

By applying the IH to (A2), we get τ_{12}' such that

(A5) $\Gamma \vdash c_1 : (x : \tau_1) \rightsquigarrow \tau_{12}'$, and

(A6) $\Gamma, x : \tau_1 \vdash \tau_{12}' <: \tau_{12}$

By applying the IH to (A3), we get τ_{22}' such that

(A7) $\Gamma, x : \tau_1 \vdash c_2 : (y : \tau_{12}) \rightsquigarrow \tau_{22}'$, and

(A8) $\Gamma, x : \tau_1, y : \tau_{12} \vdash \tau_{22}' <: \tau_{22}$

By Lemma A.70 with (A6) and (A7) there exists τ_{22}'' such that

(A9) $\Gamma, x : \tau_1 \vdash c_2 : (y : \tau_{12}') \rightsquigarrow \tau_{22}''$, and

(A10) $\Gamma, x : \tau_1, y : \tau_{12}' \vdash \tau_{22}'' <: \tau_{22}'$

By T-Seq-Algo with (A5) and (A9) follows

(A11) $\Gamma \vdash c_1; c_2 : (x : \tau_1) \rightsquigarrow \tau_{22}''[y \mapsto \tau_{12}']$

which shows (1). By Lemma A.58 with (A6) and (A8) follows

(A12) $\Gamma, x : \tau_1, y : \tau_{12}' \vdash \tau_{22}' <: \tau_{22}$

By Lemma A.56 with (A10) and (A12) follows

(A13) $\Gamma, x : \tau_1, y : \tau_{12}' \vdash \tau_{22}'' <: \tau_{22}$

By Lemma A.64 follows

(A14) $\Gamma, x : \tau_1 \vdash \tau''_{22}[y \mapsto \tau'_{12}] <: \tau_{22}[y \mapsto \tau_{12}]$

which shows (2) and concludes this case.

***Case* T-Skip:** (1) follows by T-Skip-Algo and (2) follows by Lemma A.55.

***Case* T-Remit:** By inversion of T-Remit, we know

(A1) $c = remit(\iota)$

(A2) Includes $\Gamma\ \tau_1\ \iota$

(A3) $\varphi \triangleq z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota$

(A4) $\tau_2 = \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \epsilon \mid \varphi\}$

By Lemma A.71 and (A2), T-Remit-Algo computes τ'_2 such that

(A5) $\Gamma \vdash remit(\iota_i) : ((x : \tau_1)) \rightsquigarrow \tau'_2$, and

(A6) $\tau'_2 = \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \epsilon \mid z.pkt_{in} = \langle\rangle \wedge z.pkt_{in} = x.\iota\}$

which shows (1). Since $\tau'_2 = \tau_2$, (2) follows by Lemma A.55.

***Case* T-Reset:** (1) follows by T-Reset-Algo and (2) follows by Lemma A.55.

***Case* T-Ascribe:** By inversion of T-Ascribe, we know

(A1) $c = c_0\ as\ (x : \tau_1) \to \tau_2$

(A2) $\Gamma \vdash c_0 : (x : \tau_1) \to \tau_2$

By IH applied to (A2), there exists $\hat{\tau}_2$ such that

(A3) $\Gamma \vdash c_0 : (x : \tau_1) \rightsquigarrow \hat{\tau}_2$

(A4) $\Gamma, x : \tau_1 \vdash \hat{\tau}_2 <: \tau_2$

By T-Ascribe-Algo with (A3), (A4) and Lemma A.55,

(A5) $\Gamma \vdash c_0\ as\ (x : \tau_1) \to \tau_2 : (x : \tau_1) \rightsquigarrow \tau_2$

showing (1). (2) follows by Lemma A.55.

***Case* T-If:** By inversion of T-If, we know

(A1) $c = if(\varphi)\ c_1\ else\ c_2$

(A2) $\Gamma; \tau_1 \vdash \varphi : \mathbb{B}$

(A3) $\Gamma \vdash c_1 : (x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}) \to \tau_{12}$

(A4) $\Gamma \vdash c_2 : (x : \{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\}) \to \tau_{22}$

(A5) $\tau_2 = \{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\}$

By the IH applied to (A3) there exists τ'_{12} such that

(A6) $\Gamma \vdash c_1 : (x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}) \rightsquigarrow \tau'_{12}$

(A7) $\Gamma, x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\} \vdash \tau'_{12} <: \tau_{12}$

By the IH applied to (A4) there exists τ'_{22} such that

(A8) $\Gamma \vdash c_2 : (x : \{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\}) \rightsquigarrow \tau'_{22}$

(A9) $\Gamma, x : \{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\} \vdash \tau'_{22} <: \tau_{22}$

(1) follows by T-If-Algo with (A2), (A6) and (A8).

(2) follows by Lemma A.67 with (A7) and (A9).

***Case* T-Mod:** By inversion of T-Mod, we know

(A1) $c = \iota.f := e$

(A2) Includes $\Gamma\ \tau_1\ \iota$

(A3) $\mathcal{F}(\iota, f) = \mathsf{BV}$

(A4) $\Gamma; \tau_1 \vdash e : \mathsf{BV}$

(A5) $\tau_2 = \{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = e[x/\mathsf{heap}]\}$

By Lemma A.71 and (A2),

(A6) $\Gamma \vdash \tau_1 <: \iota_i$

(1) follows by T-Mod-Algo with (A3),(A4), (A5), and (A6). (2) follows by Lemma A.55.

***Case* T-Add:** By inversion of T-Add, we know

(A1) $\mathsf{Excludes}\ \Gamma\ \tau_1\ \iota$

(A2) $\mathsf{init}_{\mathcal{HT}(\iota)} = v$

(A3) $\tau_2 = \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle \wedge z.\iota = v\}$

By Lemma A.72 and (A1),

(A4) $\Gamma \vdash \tau <: \{x : \top \mid \neg x.\iota.valid\}$

(1) follows by T-Add-Algo with (A1) and (A2). (2) follows by Lemma A.55.

***Case* T-Sub:** By inversion of T-Sub, there exists some τ_3 and τ_4 such that

(A1) $\Gamma \vdash \tau_1 <: \tau_3$

(A2) $\Gamma, x : \tau_1 \vdash \tau_4 <: \tau_2$

(A3) $\Gamma \vdash c\colon (x : \tau_3) \to \tau_4$

By applying the IH to (A3) there is some τ_4' such that

(A4) $\Gamma \vdash c : (x : \tau_3) \rightsquigarrow \tau_4'$, and

(A5) $\Gamma, x : \tau_3 \vdash \tau_4' <: \tau_4$

By Lemma A.58 together with (A1) and (A5), follows

(A6) $\Gamma, x : \tau_1 \vdash \tau_4' <: \tau_4$

By applying Lemma A.70 to (A1) and (A4) we get τ_4'' such that

(A7) $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_4''$, and

(A8) $\Gamma, x : \tau_1 \vdash \tau_4'' <: \tau_4'$

(1) follows by (A7). (2) follows by repeated application of Lemma A.56 with (A2), (A6) and (A8).

***Case* T-Remove:** By inversion of T-Remove, we know

(A1) $\mathsf{Includes}\ \Gamma\ \tau_1\ \iota$

(A2) $\tau_2 = \{y : \top \mid \varphi_\iota \wedge \varphi_{pkt} \wedge \neg y.\iota.valid\}$

By Lemma A.71 with (A1), we get

(A3) $\Gamma \vdash \tau_1 <: \iota_\approx$

(1) follows by T-Remove-Algo. (2) follows by Lemma A.55.

($\Leftarrow$) Proceed by induction on the typing derivation. We refer to the general assumptions as follows:

(A) $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2'$

(B) $\Gamma, (x : \tau_1) \vdash \tau_2' <: \tau_2$

***Case* T-Extract-Algo:** By inversion of T-Extract-Algo, we know

(A1) $\Gamma \vdash \tau_1 <: \{x : \top \mid |x.pkt_{in}| \geq \mathsf{sizeof}(\iota)\}$

(A2) $\tau_2' = \Sigma y\colon \{z : \iota \mid \varphi_1\}.\{z : \mathsf{chomp}(\tau_1, \iota, y) \mid \varphi_2\}$

By Lemma A.59 follows

(A3) $\Gamma \vdash \mathsf{sizeof}_{pkt_{in}}(\tau_1) \geq \mathsf{sizeof}(\iota)$

The result follows by T-Extract with (A3) and Lemma A.55.

***Case* T-Seq-Algo:** By inversion of T-Seq-Algo, we know

(A1) $c = c_1; c_2$

(A2) $\Gamma \vdash c_1 : ((x : \tau_1)) \rightsquigarrow \tau_{12}$

(A3) $\Gamma, (x : \tau_1) \vdash c_2 : ((y : \tau_{12})) \rightsquigarrow \tau_{22}$

(A4) $\tau_2' = \tau_{22}[y \mapsto \tau_{12}]$

(A5) $\Gamma, x : \tau_1 \vdash \tau_2' <: \tau_2$

With Lemma A.55 follows

(A6) $\Gamma, x : \tau_1 \vdash \tau_{12} <: \tau_{12}$

By IH with (A2) and (A6) follows

(A7) $\Gamma \vdash c_1 : (x : \tau_1) \to \tau_{12}$

Similarly, applying the IH to (A3) gives

(A8) $\Gamma, (x : \tau_1) \vdash c_2 : (x : \tau_{12}) \to \tau_{22}$

By T-Seq with (A7) and (A8) follows

(A9) $\Gamma \vdash c_1; c_2 : (x : \tau_1) \to \tau_{22}[y \mapsto \tau_{12}]$

By (A4), (A5), Lemma A.55 and T-Sub follows

(A10) $\Gamma \vdash c_1; c_2 : (x : \tau_1) \to \tau_2$

which concludes this case.

***Case* T-Skip-Algo:**

The result follows by T-Skip, Lemma A.55, and T-Sub.

***Case* T-Remit-Algo:** The result follows by T-Remit, Lemma A.71, Lemma A.55, and T-Sub.

***Case* T-Reset-Algo:** The result follows by T-Reset, Lemma A.55, and T-Sub.

***Case* T-Ascribe-Algo:** By inversion of T-Ascribe-Algo, we know

(A1) $c = c_0 \; as \; (x : \hat{\tau}_1) \to \tau_2'$

(A2) $\Gamma \vdash c_0 : (x : \hat{\tau}_1) \rightsquigarrow \tau_2''$

(A3) $\Gamma \vdash \tau_1 <: \hat{\tau}_1$

(A4) $\Gamma, x : \hat{\tau}_1 \vdash \tau_2'' <: \tau_2'$

By IH applied to (A2) and (A4), we get

(A5) $\Gamma \vdash c_0 : (x : \hat{\tau}_1) \to \tau_2'$

By T-Ascribe follows from (A5) that

(A6) $\Gamma \vdash c_0 \; as \; (x : \hat{\tau}_1) \to \tau_2' : (x : \hat{\tau}_1) \to \tau_2'$

The result follows by T-Sub with assumptions (B), (A3) and (A6).

***Case* T-If-Algo:** By inversion of T-If-Algo, we know

(A1) $c = if(\varphi) \; c_1 \; else \; c_2$

(A2) $\Gamma; \tau_1 \vdash \varphi : \mathbb{B}$

(A3) $\Gamma \vdash c_1 : (x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}) \rightsquigarrow \tau_{12}$

(A4) $\Gamma \vdash c_2 : (x : \{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\}) \rightsquigarrow \tau_{22}$

(A5) $\tau_2' = \{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\}$

By Lemma A.55, we can conclude

(A6) $\Gamma, x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\} \vdash \tau_{12} <: \tau_{12}$

By IH with (A3) and (A6) follows

(A7) $\Gamma \vdash c_1 : (x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}) \to \tau_{12}$

We can reason similarly as before to conclude

(A8) $\Gamma \vdash c_2 : (x : \{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\}) \to \tau_{22}$

The result follows by T-If with (A2), (A7), (A8) and (A5) and by T-Sub with assumption (B).

***Case* T-Mod-Algo:** The result follows by T-Mod with Lemmas A.55 and A.71 and T-Sub.

***Case* T-Add-Algo:** The result follows by T-Add with Lemmas A.55 and A.72 and T-Sub.

***Case* T-Remove-Algo:** The result follows by T-Remove with Lemmas A.55 and A.71 and T-Sub.

□

A.2.3 Decidability of Typechecking

Lemma A.74 (Refinement Bound)**.** *For every* Γ, x, τ, φ, N, *such that* $\Gamma \vdash \tau \leq N$, $\Gamma \vdash \{x : \tau \mid \varphi\} \leq N$.

Proof. Let Γ, x, τ, φ, N, be given such that $\Gamma \vdash \tau \leq N$. Let $\mathcal{E} \vDash \Gamma$. Further, let $h \in [\![\{x : \tau \mid \varphi\}]\!]_{\mathcal{E}}$. By the semantics of heap types, we also know that $h \in [\![\tau]\!]_{\mathcal{E}}$. Assumption $\Gamma \vdash \tau \leq N$ gives us that $|h(pkt_{in})| + |h(pkt_{out})| \leq N$, which is what we want to show. □

Lemma A.75 (Bound Constraints)**.** *For every* Γ, x, y, τ_1, τ_2, *and* φ, *such that* heap *is the only free variable in* φ, $\Gamma, x : \tau_1 \vdash \{y : \tau_2 \mid \varphi[x/\mathsf{heap}]\} \leq N$, *if and only if* $\Gamma, x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\} \vdash \tau_2 \leq N$.

Proof. Let Γ, x, y, τ_1, τ_2, and φ be given. Prove each direction separately.

($\Rightarrow$) Assume $\Gamma, x : \tau_1 \vdash \{y : \tau_2 \mid \varphi[x/\mathsf{heap}]\} \leq N$.

Let $\mathcal{E} \vDash \Gamma, x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}$. We can write $\mathcal{E} = \mathcal{E}'[x \mapsto h]$ for some $\mathcal{E}' \vDash \Gamma$, and some $h_1 \in [\![\tau_1]\!]_{\mathcal{E}'}$, such that $[\![\varphi[y/\mathsf{heap}]]\!]_{\mathcal{E}'[y \mapsto h_1]} = [\![\varphi]\!]_{\mathcal{E}'[\mathsf{heap} \mapsto h_1]} = \mathit{true}$. Since y does not occur in φ, then we also have $[\![\varphi]\!]_{\mathcal{E}[\mathsf{heap} \mapsto h, y \mapsto h_2]} = \mathit{true}$.

Now, consider $h_2 \in [\![\tau_2]\!]_{\mathcal{E}'[x \mapsto h_1]}$. To show that $|h_2(pkt_{in})| + |h_2(pkt_{out})| \leq N$. Since $[\![\varphi]\!]_{\mathcal{E}[\mathsf{heap} \mapsto h, y \mapsto h_2]} = \mathit{true} = [\![\varphi[x/\mathsf{heap}]]\!]_{\mathcal{E}[x \mapsto h, y \mapsto h_2]}$, we can conclude that $h_2 \in [\![\{y : \tau_2 \mid \varphi[x/\mathsf{heap}]\}]\!]_{\mathcal{E}'[x \mapsto h]}$. Now, since $\mathcal{E}'[x \mapsto h] \vDash \Gamma, (x : \tau_1)$, the result follows by our initial assumption assumption $\Gamma, (x : \tau_1) \vdash \{y : \tau_2 \mid \varphi[x/\mathsf{heap}]\} \leq N$.

(⇐) Assume $\Gamma, (x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}) \vdash \tau_2 \leq N$.

Let $\mathcal{E} \vDash \Gamma, (x : \tau_1)$. We can write $\mathcal{E} = \mathcal{E}'[x \mapsto h_1]$ where $h_1 \in [\![\tau_1]\!]_{\mathcal{E}'}$ and $\mathcal{E}' \vDash \Gamma$.

Now consider $h_2 \in [\![\{y : \tau_2 \mid \varphi[x/\mathsf{heap}]\}]\!]_{\mathcal{E}'[x \mapsto h_1]}$. To show $|h_2(pkt_{in})| + |h_2(pkt_{out})| < N$.

By the semantics of heap types, we have $[\![\varphi]\!]_{\mathcal{E}'[x \mapsto h_1, y \mapsto h_2]} = [\![\varphi]\!]_{\mathcal{E}'[\mathsf{heap} \mapsto h_1, y \mapsto h_2]} =$ *true*. Since y is not free in φ, we also have $[\![\varphi]\!]_{\mathcal{E}'[\mathsf{heap} \mapsto h_1]} = [\![\varphi[y/\mathsf{heap}]]\!]_{\mathcal{E}'[y \mapsto h_1]} =$ *true*, so we can conclude that $h_1 \in [\![\{y : \tau_1 \mid e[y/\mathsf{heap}]\}]\!]_{\mathcal{E}'}$. By our initial assumption, every heap in τ_2 is bounded and as such also heap $h_2 \in [\![\{y : \tau_2 \mid \varphi[x/\mathsf{heap}]\}]\!]_{\mathcal{E}'[x \mapsto h_1]}$.

□

Lemma A.76 (Bound Choice). *If* $\Gamma \vdash \tau_1 \leq N$ *and* $\Gamma \vdash \tau_2 \leq M$*, then* $\Gamma \vdash \tau_1 + \tau_2 \leq \max(M, N)$.

Proof. Let $\mathcal{E} \vDash \Gamma$ and $h \in [\![\tau_1 + \tau_2]\!]_{\mathcal{E}}$. We have to show that $|h(pkt_{in})| + |h(pkt_{out})| \leq \max(M, N)$.

Case $M = N$**:**

Assume $M = N$, so $\max(M, N) = M = N$. By the semantics of heap types $h \in [\![\tau_1]\!]_{\mathcal{E}}$ or $h \in [\![\tau_2]\!]_{\mathcal{E}}$.

Subcase $h \in [\![\tau_1]\!]_{\mathcal{E}}$**:** The result immediately follows by assumption $\Gamma \vdash \tau_1 \leq N$.

Subcase $h \in [\![\tau_2]\!]_{\mathcal{E}}$**:** The result immediately follows by assumption $\Gamma \vdash \tau_2 \leq M$.

Case $M > N$**:** Without loss of generality, we assume that $M > N$, so $\max(M, N) = M$. By the semantics of heap types $h \in [\![\tau_1]\!]_{\mathcal{E}}$ or $h \in [\![\tau_2]\!]_{\mathcal{E}}$.

Subcase $h \in [\![\tau_1]\!]_{\mathcal{E}}$**:** By assumption $\Gamma \vdash \tau_1 \leq N$ and since by assumption $N < M$, it follows $\Gamma \vdash \tau_1 \leq M$.

Subcase $h \in [\![\tau_2]\!]_{\mathcal{E}}$**:** The result immediately follows by assumption $\Gamma \vdash \tau_2 \leq M$.

□

Lemma A.77 (Bound Substitution). *For all* Γ, y, τ_1, τ_2, N, $\Gamma \vdash \tau_2[y \mapsto \tau_1] \leq N$ *if and only if* $\Gamma, (y : \tau_1) \vdash \tau_2 \leq N$.

Proof. Let Γ, y, τ_1, τ_2, and N be given. Prove each direction separately:

(⇒) Assume $\Gamma \vdash \tau_2[y \mapsto \tau_2] \leq N$. Let $\mathcal{E} \vDash \Gamma, y : \tau_1$ such that $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$. This means there is some $\mathcal{E}' \vDash \Gamma$ and $h_1 \in [\![\tau_1]\!]_{\mathcal{E}'}$ such that $h_2 \in [\![\tau_2]\!]_{\mathcal{E}'[y \mapsto h_1]}$. By the semantics of heap types follows that $h_2 \in [\![\tau_2[y \mapsto \tau_1]]\!]_{\mathcal{E}'}$. With the initial assumption $\Gamma \vdash \tau_2[y \mapsto \tau_1] \leq N$, we can conclude that $|h_2(pkt_{in})| + |h_2(pkt_{out})| \leq N$.

(⇐) Assume $\Gamma, y : \tau_1 \vDash \tau_2 \leq N$. Let $\mathcal{E} \vDash \Gamma$ and let $h_2 \in [\![\tau_2[y \mapsto \tau_1]]\!]_{\mathcal{E}}$. By the semantics of heap types, there is some $h_1 \in [\![\tau_1]\!]_{\mathcal{E}}$ such that $h_2 \in [\![\tau_2]\!]_{\mathcal{E}[y \mapsto h_1]}$. Notice that $\mathcal{E}[y \mapsto h_1] \vDash \Gamma, y : \tau_1$. The initial assumption proves that $|h(pkt_{in})| + |h(pkt_{out})| \leq N$.

□

Theorem A.78 (Forwards MTU Bound). *For every Γ, c, x, τ_1, τ_2, and $N \in \mathbb{N}$, 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. Proceed by induction on c, leaving Γ and N general. We refer to the general assumptions as follows:

(A) $\Gamma \vdash \tau_1 \leq N$

(B) $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2$

Case *extract*(ι)**:**

The only algorithmic typing rule that applies to *extract*(ι) is T-Extract-Algo. By inversion, we know

(A1) $\tau_2 = \Sigma y\colon \{z : \iota \mid \varphi_1\}.\{z : \mathsf{chomp}(\tau_1, \iota, y) \mid \varphi_2\}$

(A2) $\varphi_1 = z.pkt_{in} = z.pkt_{out} = \langle\rangle$

(A3) $\varphi_2 = y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x$.

Since $\mathsf{emit}(\mathit{extract}(\iota)) = 0$, it suffices to show $\Gamma, x : \tau_1 \vdash \tau_2 \leq N$.

Let $\mathcal{E} \vDash \Gamma, x : \tau_1$ and let $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$. We can write $\mathcal{E}$ as $\mathcal{E}'[x \mapsto h_1]$ where $h_1 \in [\![\tau_1]\!]_{\mathcal{E}'}$.

By definition of the semantics of heap types we know there are some h_{21} and h_{22} such that

(A4) $h_2 = h_{21} +\!\!+ h_{22}$,

(A5) $h_{21}(pkt_{in}) = h_{21}(pkt_{out}) = \langle\rangle$

(A6) $h_1(pkt_{in}) = h_{21}(\iota)@h_{22}(pkt_{in})$

(A7) $h_1(pkt_{out}) = h_{22}(pkt_{out})$

We can further conclude that

(A8) $h_2(pkt_{out}) = h_1(pkt_{out})$,

(A9) $h_2(pkt_{in}) = h_{22}(pkt_{in}) = h_1(pkt_{in})[|\iota| :]$

From assumption (A) follows

(A10) $|h_1(pkt_{in})| + |h_1(pkt_{out})| \leq N$

Together with (A8) and (A9), we can conclude that $|h_2(pkt_{in})| + |h_2(pkt_{out})| < |h_1(pkt_{in})| + |h_1(pkt_{out})| \leq N$.

Case *add*(ι)**:**

The only algorithmic typing rule that applies to *add*(ι) is T-Add-Algo. By inversion, we know

(A1) $\tau_2 = \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle \wedge z.\iota = v\}$.

Since $\mathsf{emit}(\mathit{add}(\iota)) = 0$, it suffices to show that $\Gamma, x : \tau_1 \vdash \tau_2 \leq N$.

Let $\mathcal{E} \vDash \Gamma, x : \tau_1$, and $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$. We can write $\mathcal{E}$ as $\mathcal{E}'[x \mapsto h_1]$ where $h_1 \in [\![\tau_1]\!]_{\mathcal{E}'}$.

By definition of the semantics of heap types we know there are some h_{21} and h_{22} such that

(A2) $h_2 = h_{21} +\!\!+ h_{22}$

(A3) $h_{21} = h_1$

(A4) $h_{22}(pkt_{in}) = h_{22}(pkt_{out}) = \langle\rangle$

From these three equations we can conclude that

(A5) $h_2(pkt_{in}) = h_1(pkt_{in})$

(A6) $h_2(pkt_{out}) = h_1(pkt_{out})$

The result follows by assumption (A).

Case *remove*(ι)**:**

The only algorithmic typing rule that applies to *remove*(ι) is T-Remove-Algo. By inversion, we know

(A1) $\tau_2 = \{y : \top \mid \varphi_\iota \wedge \varphi_{pkt} \wedge \neg y.\iota.valid\}$

(A2) $\varphi_{pkt} = y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out}$

Since $\mathsf{emit}(remove(\iota)) = 0$, it suffices to show that $\Gamma, x : \tau_1 \vdash \tau_2 \leq N$.

Let $\mathcal{E} \vDash \Gamma, x : \tau_1$ and $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$. We can write $\mathcal{E}$ as $\mathcal{E}'[x \mapsto h_1]$ where $h_1 \in [\![\tau_1]\!]_{\mathcal{E}'}$. From assumption (A1) and by the semantics of heap types follows

(A3) $h_2(pkt_{in}) = h_1(pkt_{in})$

(A4) $h_2(pkt_{out}) = h_1(pkt_{out})$

The result follows by assumption (A).

Case $\iota.f := e$**:**

The only algorithmic typing rule that applies to $\iota.f := e$ is T-Mod-Algo. By inversion, we know

(A1) $\tau_2 = \{y : \top \mid \varphi_{pkt} \wedge \varphi_\iota \wedge \varphi_f \wedge y.\iota.f = e[x/\mathsf{heap}]\}$

(A2) $\varphi_{pkt} = y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out}$

Since $\mathsf{emit}(\iota.f := e) = 0$, it suffices to show that $\Gamma, x : \tau_1 \vdash \tau_2 \leq N$.

Let $\mathcal{E} \vDash \Gamma, x : \tau_1$ and $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$. We can write $\mathcal{E}$ as $\mathcal{E}'[x \mapsto h_1]$ where $h_1 \in [\![\tau_1]\!]_{\mathcal{E}'}$. From assumption (A1) and by the semantics of heap types follows

(A3) $h_2(pkt_{in}) = h_1(pkt_{in})$

(A4) $h_2(pkt_{out}) = h_1(pkt_{out})$

The result follows by assumption (A).

Case *remit*(ι)**:**

The only algorithmic typing rule that applies to *remit*(ι) is T-Remit-Algo. By inversion, we know

(A1) $\tau_2 = \Sigma y\colon \{z : \tau_1 \mid z \equiv x\}.\{z : \epsilon \mid \varphi\}$

(A2) $\varphi = z.pkt_{in} = \langle\rangle \wedge z.pkt_{out} = x.\iota$

Since $\mathsf{emit}(\mathit{remit}(\iota)) = \mathsf{sizeof}(\iota)$, we have to show that $\Gamma, x : \tau_1 \vdash \tau_2 \leq N + \mathsf{sizeof}(\iota)$.

Let $\mathcal{E} \vDash \Gamma, x : \tau_1$ and $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$. We can write $\mathcal{E}$ as $\mathcal{E}'[x \mapsto h_1]$ where $h_1 \in [\![\tau_1]\!]_{\mathcal{E}'}$. By the semantics of heap types, there exists h_{21} and h_{22} such that

(A3) $h_2 = h_{21} +\!\!+ h_{22}$

(A4) $h_{21} = h_1$

(A5) $h_{22}(pkt_{in}) = \langle\rangle$

(A6) $h_{22}(pkt_{out}) = h_1(\iota)$

From (A4) and (A5), we can conclude that

(A7) $h_2(pkt_{in}) = h_1(pkt_{in})$

From (A4) and (A6), we can further conclude that

(A8) $h_2(pkt_{out}) = h_1(pkt_{out}) @ h_1(\iota)$

From (A7) and (A8) then follows

(A9) $|h_2(pkt_{in})| + |h_2(pkt_{out})| = |h_1(pkt_{in})| + |h_1(pkt_{out})| + \mathsf{sizeof}(\iota)$

Together with assumption (A), we can conclude that $|h_2(pkt_{in})| + |h_2(pkt_{out})| \leq N + \mathsf{sizeof}(\iota)$.

***Case** reset*:

The only algorithmic typing rule that applies to *reset* is T-Reset-Algo. By inversion, we know

(A1) $\tau_2 = \Sigma y\colon \{z : \epsilon \mid \varphi_1\}.\{z : \epsilon \mid \varphi_2\}$

(A2) $\varphi_1 = z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{out}$

(A3) $\varphi_2 = z.pkt_{out} = \langle\rangle \wedge z.pkt_{in} = x.pkt_{in}$

Since $\mathsf{emit}(\mathit{reset}) = 0$, we have to show that $\Gamma, x : \tau_1 \vdash \tau_2 \leq N$.

Let $\mathcal{E} \vDash \Gamma, x : \tau_1$ and $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$. We can write $\mathcal{E}$ as $\mathcal{E}'[x \mapsto h_1]$ where $h_1 \in [\![\tau_1]\!]_{\mathcal{E}'}$. By the semantics of heap types, there exists h_{21} and h_{22} such that

(A4) $h_2 = h_{21} +\!\!+ h_{22}$

(A5) $h_{21}(pkt_{out}) = \langle\rangle$

(A6) $h_{21}(pkt_{in}) = h_1(pkt_{out})$

(A7) $h_{22}(pkt_{out}) = \langle\rangle$

(A8) $h_{22}(pkt_{in}) = h_1(pkt_{in})$

By (A4), (A5) and (A7) follows

(A9) $h_2(pkt_{out}) = \langle\rangle$

and by (A4), (A6) and (A8) follows

(A10) $h_2(pkt_{in}) = h_1(pkt_{out}) @ h_1(pkt_{in})$

Since by assumption (A), $|h_1(pkt_{in})| + |h_1(pkt_{out})| \leq N$, by (A9) and (A10), $|h_2(pkt_{in})| + |h_2(pkt_{out})| \leq N$.

Case *if*(φ) c_1 *else* c_2**:** The only algorithmic typing rule that applies is T-If-Algo. By inversion, we know

(A1) $\tau_2 = \{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\}$

(A2) $\Gamma \vdash c_1 : (x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}) \rightsquigarrow \tau_{12}$

(A3) $\Gamma \vdash c_1 : (x : \{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\}) \rightsquigarrow \tau_{22}$

Since $\mathsf{emit}(\mathit{if}(\varphi)\ c_1\ \mathit{else}\ c_2) = \max(\mathsf{emit}(c_1), \mathsf{emit}(c_2))$, we have to show that $\Gamma, x : \tau_1 \vdash \tau_2 \leq N + \max(\mathsf{emit}(c_1), \mathsf{emit}(c_2))$.

By Lemma A.74 and assumption (A) follows

(A4) $\Gamma \vdash \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\} \leq N$

(A5) $\Gamma \vdash \{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\} \leq N$

Applying the IH to (A2) and (A2) with (A4) and (A5) respectively, gives

(A6) $\Gamma, x : \{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\} \vdash \tau_{12} \leq N + \mathsf{emit}(c_1)$

(A7) $\Gamma, x : \{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\} \vdash \tau_{22} \leq N + \mathsf{emit}(c_2)$

By Lemma A.75 with (A6) and (A7) respectively follows

(A8) $\Gamma, x : \tau_1 \vdash \{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} \leq N + \mathsf{emit}(c_1)$

(A9) $\Gamma, x : \tau_1 \vdash \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\} \leq N + \mathsf{emit}(c_2)$

By Lemma A.76 with (A8) and (A9) follows

(A10) $\Gamma, x : \tau_1 \vdash \{y : \tau_{12} \mid \varphi[x/\mathsf{heap}]\} + \{y : \tau_{22} \mid \neg\varphi[x/\mathsf{heap}]\} \leq \max(N + \mathsf{emit}(c_1), N + \mathsf{emit}(c_2))$

The result follows together with the fact that $\max(A + B, A + C) = A + \max(B, C)$.

Case $c_1; c_2$**:**

The only algorithmic typing rule that applies to $c_1; c_2$ is T-Seq-Algo. By inversion, we know

(A1) $\Gamma \vdash c_1 : (x : \tau_1) \rightsquigarrow \tau_{12}$

(A2) $\Gamma, x : \tau_1 \vdash c_2 : (y : \tau_{12}) \rightsquigarrow \tau_{22}$

(A3) $\tau_2 = \tau_{22}[y \mapsto \tau_{12}]$

Since $\mathsf{emit}(c_1; c_2) = \mathsf{emit}(c_1) + \mathsf{emit}(c_2)$, we have to show that $\Gamma, x : \tau_1 \vdash \tau_2 \leq N + \mathsf{emit}(c_1) + \mathsf{emit}(c_2)$. Let $\mathcal{E} \vDash \Gamma, x : \tau_1$ and let $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$.

By applying the IH to (A1), we get

(A4) $\Gamma, x : \tau_1 \vdash \tau_{12} \leq N + \mathsf{emit}(c_1)$

Since we left Γ and N general, we can apply the IH again to (A2) and get

(A5) $\Gamma, x : \tau_1, y : \tau_{12} \vdash \tau_{22} \leq N + \mathsf{emit}(c_1) + \mathsf{emit}(c_2)$

The result follows by Lemma A.77 with (A5).

***Case** skip*:

The only algorithmic typing rule that applies to *skip* is T-Skip-Algo. By inversion, we know

(A1) $\tau_2 = \{y : \tau_1 \mid y \equiv x\}$

Since $\mathsf{emit}(\mathit{skip}) = 0$, we have to show that $\Gamma, x : \tau_1 \vdash \tau_2 \leq N$.

Let $\mathcal{E} \vDash \Gamma, x : \tau_1$ and $h_2 \in [\![\tau_2]\!]_{\mathcal{E}}$. We can write $\mathcal{E}$ as $\mathcal{E}'[x \mapsto h_1]$ where $h_1 \in [\![\tau_1]\!]_{\mathcal{E}'}$.

By the semantics of heap types, follows

(A2) $h_2(pkt_{in}) = h_1(pkt_{in})$

(A3) $h_2(pkt_{out}) = h_1(pkt_{out})$

The result follows by assumption (A).

***Case** c_0 as $(x : \hat{\tau}_1) \to \tau_2$*:

The only algorithmic typing rule that applies is T-Ascribe-Algo. By inversion, we know

(A1) $\Gamma \vdash c_0 : (x : \hat{\tau}_1) \rightsquigarrow \tau_c$

(A2) $\Gamma \vdash \tau_1 <: \hat{\tau}_1$

(A3) $\Gamma, x : \hat{\tau}_1 \vdash \tau_c <: \tau_2$

Since $\mathsf{emit}(c_0 \text{ as } \sigma) = \mathsf{emit}(c_0)$, we have to show that $\Gamma, x : \tau_1 \vdash \tau_2 \leq N + \mathsf{emit}(c_0)$. By our initial assumption, every ascribed type is also bounded by N. We therefore have $\Gamma, x : \tau_1 \vdash \tau_2 \leq N$ from which the result immediately follows.

□

Theorem A.79 (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. By Theorem A.73 (Algorithmic Typing Correctness), we can equivalently show that $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2'$ and $\Gamma, x : \tau_1 \vdash \tau_2' <: \tau_2$ are decidable. By Theorem A.78, τ_2' is bounded. $\Gamma, x : \tau_1 \vdash \tau_2' <: \tau_2$ is therefore decidable by finite enumeration.

To show that $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2'$ is decidable, we proceed by induction on the algorithmic typing derivation.

***Case** T-Skip-Algo*:

Immediate, because T-Skip-Algo does not perform any subtyping checks.

***Case** T-Reset-Algo*:

Also immediate, because T-Reset-Algo does not perform any subtyping checks.

***Case** T-Seq-Algo*:

By inversion of T-Seq-Algo,

(A1) $\Gamma \vdash c_1 : (x : \tau_1) \rightsquigarrow \tau_{12}$

(A2) $\Gamma, x : \tau_1 \vdash c_2 : (y : \tau_{12}) \rightsquigarrow \tau_{22}$

By (A1) and Theorem A.78,

(A3) $\Gamma, x : \tau_1 \vdash \tau_{12} \leq N + \mathsf{emit}(c_1)$

Applying the IH to (A1) with assumption $\Gamma \vdash \tau_1 \leq N$ and (A3) gives us that $\Gamma \vdash c_1 : (x : \tau_1) \rightsquigarrow \tau_{12}$ is decidable.

Again, by Theorem A.78 with (A2) and (A3), follows

(A4) $\Gamma, x : \tau_1, y : \tau_{12} \vdash \tau_{22} \leq N + \mathsf{emit}(c_1) + \mathsf{emit}(c_2)$

By IH follows that $\Gamma, x : \tau_1 \vdash c_2 : (y : \tau_{12}) \rightsquigarrow \tau_{22}$ is decidable and thus type checking the sequence of both commands is decidable.

***Case* T-Add-Algo:**

By inversion, we know that T-Add-Algo performs the subtyping check $\Gamma \vdash \tau_1 <: \{x : \top \mid \neg x.\iota.valid\}$. To show that type checking is decidable in this case, we must show that $\Gamma \vdash \tau_1 <: \{x : \top \mid \neg x.\iota.valid\}$ is decidable. This is the case because we can finitely enumerate the heaps h described by τ_1 and check wether every h is a member of $\{x : \top \mid \neg y.\iota.valid\}$.

***Case* T-Extract-Algo:**

By inversion, we know that T-Extract-Algo performs the subtyping check $\Gamma \vdash \tau_1 <: \{x : \top \mid |x.pkt_{in}| \geq \mathsf{sizeof}(\iota)\}$. To show that type checking is decidable in this case, we must show that $\Gamma \vdash \tau_1 <: \{x : \top \mid |x.pkt_{in}| \geq \mathsf{sizeof}(\iota)\}$ is decidable. This is the case because we can finitely enumerate the heaps h described by τ_1 and check wether every h is a member of $\{x : \top \mid |x.pkt_{in}| \geq \mathsf{sizeof}(\iota)\}$.

***Case* T-Remove-Algo:**

By inversion, we know that T-Remove-Algo performs the subtyping check $\Gamma \vdash \tau_1 <: \{x : \top \mid x.\iota.valid\}$. To show that type checking is decidable in this case, we must show that $\Gamma \vdash \tau_1 <: \{x : \top \mid x.\iota.valid\}$ is decidable. This is the case because we can finitely enumerate the heaps h described by τ_1 and check wether every h is a member of $\{x : \top \mid x.\iota.valid\}$.

***Case* T-Remit-Algo:**

Identical to the previous subcase.

***Case* T-Mod-Algo:**

Identical to the previous subcase.

***Case* T-If-Algo:**

Since τ_1 is bounded by assumption and refining the input type does not increase the size, $\{y : \tau_1 \mid \varphi[y/\mathsf{heap}]\}$ and $\{y : \tau_1 \mid \neg\varphi[y/\mathsf{heap}]\}$ are still bounded. By Theorem A.78 then follows that the output types of c_1 and c_2 are also bounded. By IH applied to c_1 and c_2, we get that the algorithmic type checking applied to c_1 and c_2 respectively is decidable and thus checking the conditional is decidable.

***Case* T-Ascribe-Algo:**

By assumption, Γ, τ_1 and $\hat{\tau}_1$ are bounded, so $\Gamma \vdash \tau_1 <: \hat{\tau}_1$ is decidable by finite enumeration. Since by assumption $\Gamma \vdash \tau_1 \leq N$, by Theorem A.78 follows that $\Gamma, x : \hat{\tau}_1 \vdash \tau_c \leq N + \mathsf{emit}(c_0)$. By IH then follows that $\Gamma \vdash c : (x : \tau_1) \rightsquigarrow \tau_2'$ is decidable. Since τ_c is bounded and by assumption also τ_2' is bounded, we can finitely enumerate, so $\Gamma, x : \hat{\tau}_1 \vdash \tau_c <: \tau_2'$ is also decidable and thus type checking an ascribed command is decidable.

□

A.2.4 Type Equivalences

Lemma A.80 (Rewrite Sigma Extract). *For all* Γ, x, τ *and* ι, *if* $\Gamma \vdash \mathsf{sizeof}_{pkt_{in}}(\tau) \geq \mathsf{sizeof}(\iota)$ *and* x *does not occur free in* τ, *then*

$$\Gamma, x : \tau \vdash$$

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

$$\doteq$$

$$\left\{ y : \top \;\middle|\; \begin{array}{l} y.\iota.valid \wedge \bigwedge_{\kappa \in dom(\mathcal{HT}) \wedge \kappa \neq \iota} y.\kappa = x.\kappa \wedge \\ y.\iota@y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out} \end{array} \right\}$$

Proof. Proof each direction separately.

(⇒) Let $\mathcal{E} \vDash \Gamma, x : \tau$. We know $\mathcal{E} = \mathcal{E}'[x \mapsto h]$ such that $h \in [\![\tau]\!]_{\mathcal{E}'}$. Let $h_\Sigma \in [\![\Sigma y : \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}.\{z : \mathsf{chomp}(\tau, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}]\!]_{\mathcal{E}'[x \mapsto h]}$ be arbitrary. By the semantics of heap types follows

(A1) $h_\Sigma = h_1 +\!\!+ h_2$

(A2) $h_1 \in [\![\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}]\!]_{\mathcal{E}'[x \mapsto h]}$

(A3) $h_2 \in [\![\{z : \mathsf{chomp}(\tau, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}]\!]_{\mathcal{E}'[x \mapsto h, y \mapsto h_1]}$

(A4) $h_2 \in [\![\mathsf{chomp}(\tau, \iota, y)]\!]_{\mathcal{E}'[x \mapsto h, y \mapsto h_1]}$

(A5) $[\![y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x]\!]_{\mathcal{E}'[x \mapsto h, y \mapsto h_1, z \mapsto h_2]} = true$

By Lemma A.47, there exists $\hat{h}_2 \in [\![\tau]\!]_{\mathcal{E}'[x \mapsto h]}$ such that

(A6) $h_2 = \mathsf{chomp}^{\Downarrow}(\hat{h}_2, \mathsf{sizeof}(\iota))$

Together with (A5), we can conclude that

(A7) $\hat{h}_2(pkt_{out}) = h(pkt_{out})$

(A8) $\hat{h}_2(pkt_{in}) = h(pkt_{in})[\mathsf{sizeof}(\iota):]$

(A9) $\forall \kappa \neq \iota . \hat{h}_2(\kappa) = h(\kappa)$

(A10) $\iota \notin dom(\hat{h}_2)$

$h_\Sigma \in [\![\{y : \top \mid y.\iota.valid \wedge \bigwedge_{\kappa \in dom(\mathcal{HT}) \wedge \kappa \neq \iota} y.\kappa = x.\kappa \wedge y.\iota@y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out}\}]\!]_{\mathcal{E}}$ follows by the semantics of heap types with (A1), (A6), (A7), (A8), (A9) and (A10).

(⇐) Let $\mathcal{E} \vDash \Gamma, x : \tau$. We know $\mathcal{E} = \mathcal{E}'[x \mapsto h]$ such that $h \in [\![\tau]\!]_{\mathcal{E}'}$. Let $\hat{h} \in [\![\{y : \top \mid y.\iota.valid \wedge \bigwedge_{\kappa \in dom(\mathcal{HT}) \wedge \kappa \neq \iota} y.\kappa = x.\kappa \wedge y.\iota@y.pkt_{in} = x.pkt_{in} \wedge y.pkt_{out} = x.pkt_{out}\}]\!]_{\mathcal{E}'[x \mapsto h]}$

By the semantics of heap types,

(A1) $\hat{h}(\iota) = h(pkt_{in})[0{:}\mathsf{sizeof}(\iota)]$

(A2) $\forall \kappa \neq \iota.\hat{h}(\kappa) = h(\kappa)$

(A3) $\hat{h}(pkt_{out}) = h(pkt_{out})$

(A4) $\hat{h}(\iota)@\hat{h}(pkt_{in}) = h(pkt_{in}) \Leftrightarrow \hat{h}(pkt_{in}) = h(pkt_{in})[\mathsf{sizeof}(\iota):]$

To show that $\hat{h} \in [\![\Sigma y : \{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}.\{z : \mathsf{chomp}(\tau, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}]\!]_{\mathcal{E}'[x \mapsto h]}$, we have to show that there exists h_1 and h_2 such that $\hat{h} = h_1 \mathbin{++} h_2$ and $h_1 \in [\![\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}]\!]_{\mathcal{E}'[x \mapsto h]}$ and $h_2 \in [\![\{z : \mathsf{chomp}(\tau, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}]\!]_{\mathcal{E}'[x \mapsto h, y \mapsto h_1]}$.

Let $h_1(pkt_{in}) = h_1(pkt_{out}) = \langle\rangle$ and $h_1(\iota) = h(pkt_{in})[\mathsf{o} : \mathsf{sizeof}(\iota)]$ and no other instances be valid in heap h_1. $h_1 \in [\![\{z : \iota \mid z.pkt_{in} = z.pkt_{out} = \langle\rangle\}]\!]_{\mathcal{E}'[x \mapsto h]}$ then follows by the semantics of heap types. By Lemma A.44, there exists $h_2 \in [\![\mathsf{chomp}(\tau, \iota, y)]\!]_{\mathcal{E}'[y \mapsto h_1]}$ such that $h_2 = \mathsf{chomp}^{\Downarrow}(h, \mathsf{sizeof}(\iota))$. Since x not free in τ, it also holds that $h_2 \in [\![\mathsf{chomp}(\tau, \iota, y)]\!]_{\mathcal{E}'[x \mapsto h, y \mapsto h_1]}$.

Since $h_1(\iota)@h_2(pkt_{in}) = h(pkt_{in})$, $h_2(pkt_{out}) = h(pkt_{out})$ and since chomp does not change already valid header instances also for all $\kappa \neq \iota$, $h_2(\kappa) = h(\kappa)$, we can conclude that $[\![y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x]\!]_{\mathcal{E}'[x \mapsto h, y \mapsto h_1, z \mapsto h_2]} = \mathit{true}$ and thus $h_2 \in [\![\{z : \mathsf{chomp}(\tau, \iota, y) \mid y.\iota@z.pkt_{in} = x.pkt_{in} \wedge z.pkt_{out} = x.pkt_{out} \wedge z \equiv_\iota x\}]\!]_{\mathcal{E}'[x \mapsto h, y \mapsto h_1]}$.

By the semantics of heap types, we can further conclude that

(A5) $(h_1 \mathbin{++} h_2)(pkt_{in}) = h_2(pkt_{in}) = h'(pkt_{in})$

(A6) $(h_1 \mathbin{++} h_2)(pkt_{out}) = h(pkt_{out}) = h'(pkt_{out})$

(A7) $(h_1 \mathbin{++} h_2)(\iota) = h(pkt_{in})[\mathsf{o}{:}\mathsf{sizeof}(\iota)] = h'(\iota)$

(A8) $\forall \kappa \neq \iota.(h_1 \mathbin{++} h_2)(\kappa) = h_2(\kappa) = h(\kappa) = h'(\kappa)$

This shows that actually $h' = h_1 \mathbin{++} h_2$ and concludes this case.

□