Vrije Universiteit Amsterdam

University of Amsterdam

Master Thesis

Nano P4: Towards Formal Verification of P4 and P4 Applications using Isabelle/HOL

Author: Johannes Blaser BSc

Student Number: **VU:** 2655674
UvA: 11044527

Main Supervisors: Dr. Jasmin Blanchette
Prof. Herbert Bos
Dr. Cristiano Giuffrida
Dr. Erik van der Kouwe

Daily Supervisor: Manuel Wiesinger MSc

A thesis submitted in fulfilment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

6th January 2021

"I don't want to turn to dust, having known nothing at all."
from The Ants*, by Exurb1a*

Abstract

P4 is a novel programming language that makes the dataplane programmable at line-rate speeds; bringing unparalleled customisability. This flexibility also leaves considerable room for error. Bugs and mistakes can fundamentally change the meaning of a program and lead to potentially exploitable vulnerabilities.

We present Nano P4: A novel approach for the formalisation and verification of P4 and P4 applications leveraging the Isabelle/HOL proof assistant. We provide a small-step semantics of P4 actions, including numerous correctness proofs and a strict typing system. We show that our semantics can be used to formally reason about properties of P4 and P4 applications, including termination or the prevention of the usage of uninitialised variables. Moreover, we show that our semantics can be used to reason about the correctness of program transformations, such as verified optimisation routines. Additionally, we provide a formalisation of P4's parser state machines and show that Nano P4 can be used to reason about reachability properties and parser optimisations.

With Nano P4 we show that it is possible to use the Isabelle/HOL proof assistant to formalise and verify P4 and P4 applications. Moreover, we show that Nano P4 is a versatile and powerful tool to reason about P4 and P4 applications.

Acknowledgements

During the developing and writing of this thesis I received a lot of help and support. You have all contributed to my thesis and keeping me motivated, even during the COVID-19 pandemic.

Firstly I would like to thank my direct supervisors. I've had the privilege of having more than just one supervisor, all with different areas of expertise, each bringing an invaluable insight with their feedback and support.

I would like to thank Jasmin Blanchette, whose expertise about all things formal verification and Isabelle, enabled me to write this thesis. I greatly appreciate this and thank you for your continuous assistance and readiness to answer even my sometimes simplistic questions.

I would also like to thank Herbert Bos, Cristiano Giuffrida, and Erik van der Kouwe from the VUSec research group. Your courses and your guidance kept me on my toes and taught me an incredible amount throughout my entire master's degree. Thank you.

From the VUSec team I would also like to thank my daily supervisor: Manuel Wiesinger. Your valuable continuous advice and discussions allowed me to successfully complete my thesis; thank you.

In addition, I would like to thank both my parents and my wonderful partner. You are all always there for me. Your unconditional love and support allowed me to get this far, and continues to motivate me to get further every day.

Lastly I would like to thank all of my friends, colleagues, and all of the other wonderful people that provide me with support, engaging discussions, ideas, and motivation to keep going every day. Thank you all.

Contents

1

Introduction

1.1 Introduction

1.1.1 Traditional Hardware

Traditional networking devices are often all designed, created, programmed, and maintained by the vendor of the device. The hardware is often specific to the intended behaviour of the device and the software commonly closed-source and proprietary. The way the device behaves, the protocols it supports and runs, and the ways in which it can be used are all hard-coded into the device by the vendor. Each instance of a device runs its own routing algorithms that populate its forwarding tables, and together these devices behave according to the protocol running in their control plane (figure 1.1). Changing these protocols requires changing the firmware on all devices, if the hardware even supports the new protocol. Without such support, changing a network's behaviour could require changing all of the hardware in it.

This means that a consumer is completely dependent on the vendor, both for what the devices can do and for continued updates and support. When a new protocol is created, or the intended use-case of a device changes, customers might be forced to purchase completely new devices, even if the older hardware is still completely operational. For more complicated and unique use-cases a device that supports the desired behaviour might not even exist or be prohibitively complicated to implement, because there is no economic incentive for vendors to create a convenient device that only a limited number of consumers would purchase.

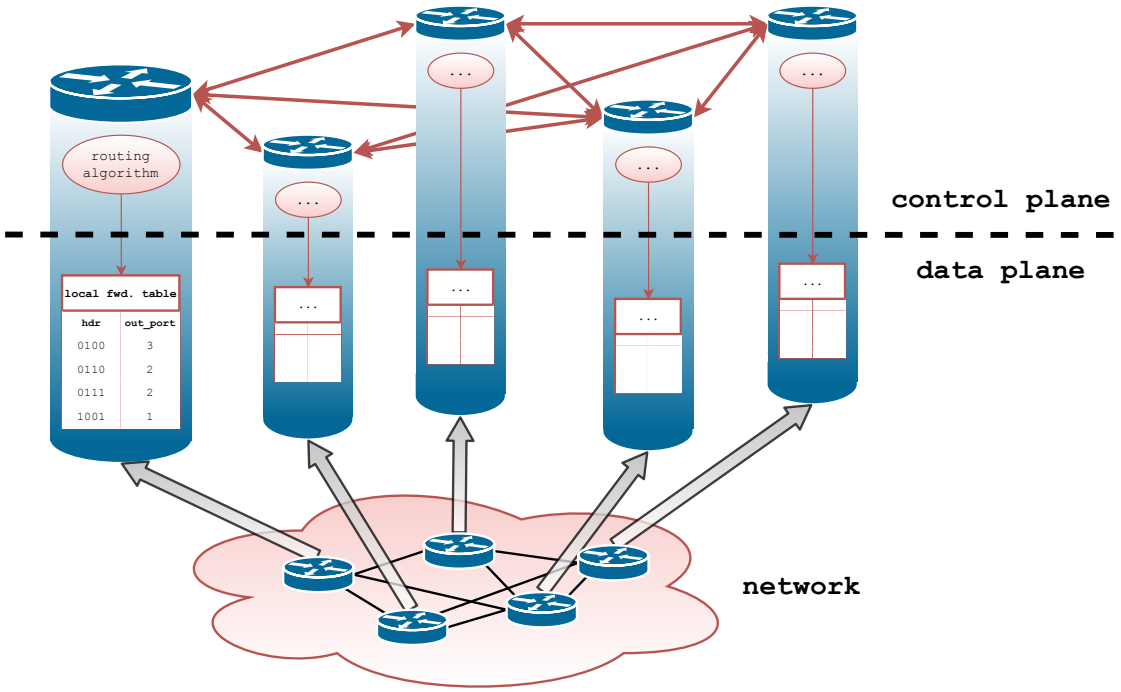

Figure 1.1: Traditional network devices have their control plane hard-coded into them. The routing algorithm they are running populates their forwarding tables without outside input, and cannot easily be changed. Changing the functionality requires changing the firmware or potentially the device itself. This diagram has been modified from *Computer Networking: A Top-Down Approach* [1].

1.1.2 Separating the Control Plane and Data Plane

In the 1990s researchers actively researched ways in which network devices could be made more dynamic and customisable. One direction of research focused on the injection of executable code into the network. The Active Internet Protocol [2] allowed for such injections, where code contained in packets would be executed at networking devices along a packet's path in a network. The Tempest framework [3] extends this and provides a programmable network environment at different levels of granularity, by also allowing a user to inject code into the network, or even allowing third parties to program entire virtual networks. Other approaches in the 1990s focused on the physical separation of the control plane and the data plane [4]. If these planes no longer exist together, they can also be altered independently (figure 1.2).

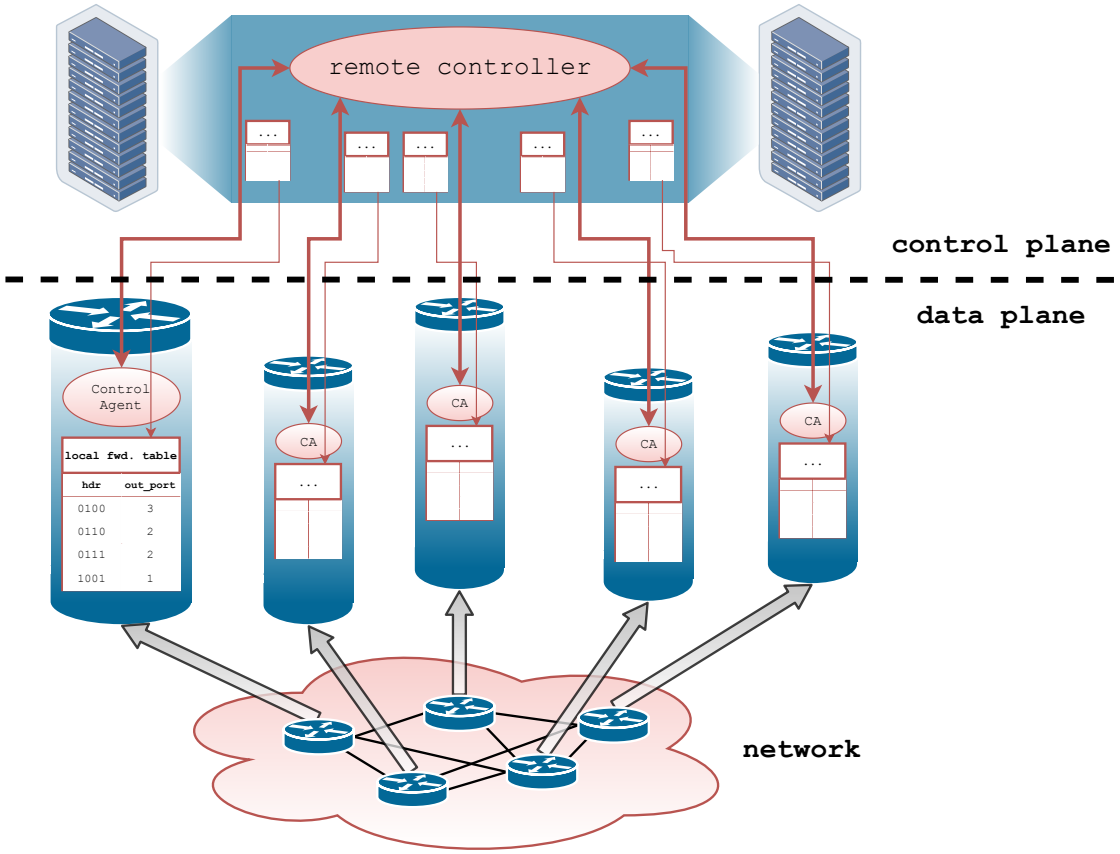

Figure 1.2: The control plane and the data plane can be physically separated by placing the control plane into a remote controller. This controller sends control instructions to the Control Agents in the network. The forwarding tables of a device are filled according to the instructions from the controller. The OpenFlow [5] protocol implements this separation. This diagram has been modified from *Computer Networking: A Top-Down Approach* [1].

1.1.3 OpenFlow

The *OpenFlow protocol* [5] is one example that puts this separation into practice. It removes the routing algorithms from the network devices and places them into a separate *controller* responsible for administrating the layer three forwarding tables. The controller exists outside of the devices themselves, and can send control instructions to each *Control Agent (CA)* in the network via the strictly defined OpenFlow protocol (figure 1.3).

It physically separates the control plane and the data plane from each other. Changing the controller then entirely changes the behaviour of the network. Moreover, changing the behaviour of a network is now possible by changing only the controller, while the other devices in the network can remain the same. The behaviour of the controller itself can

be customised through a *Northbound API* supplied by the vendor of the hardware. This allows a consumer to specify how the controller should control the forwarding tables and thus specifies the behaviour of the control plane (figure 1.3).

Though OpenFlow brings far greater customisability, it remains limited to administrating the forwarding tables through the API. Moreover, OpenFlow assumes the hardware is a non-programmable given that limits what the device can do. Combined, this made it difficult to achieve custom behaviour beyond simply modifying the entries in the forwarding tables.

1.1.4 P4

P4 instead approaches network programmability from the bottom up; rather than being limited by fixed-function hardware, P4 assumes the hardware is general and can easily be reprogrammed. This allows P4 to conveniently implement far more complex behaviours directly in the data plane. Rather than having to invent custom and complicated solutions to achieve a desired behaviour, the device itself can directly be reprogrammed to exhibit the desired forwarding and processing behaviour. This eliminates the need for a static protocol like OpenFlow. Similar to a regular C program, a P4 program is compiled against different target devices to run on their specific hardware, and the resulting binary can be pushed to the P4-enabled device. Such a P4 program specifies the structure of the forwarding tables and internal control flow. Those forwarding tables are populated through an automatically generated API by the control plane, which resides in a runtime controller (figure 1.3).

This bottom-up approach makes the data plane fully programmable, which makes P4-controlled devices far more customisable than previously conveniently possible. Running custom protocols or tailoring the control plane to best suit the topology of a particular network requires only programming the devices to behave as desired. Because P4 makes the data plane programmable, it can also be used to efficiently gather telemetry on the network by observing packets flowing through the network. This telemetry can be used to analyse the state of the network, congestion in the network, or even feed intrusion detection systems [7, 8]. Moreover, P4's approach is more economical and sustainable, as a device can simply be reprogrammed, rather than having to purchase a new device to add the desired functionality.

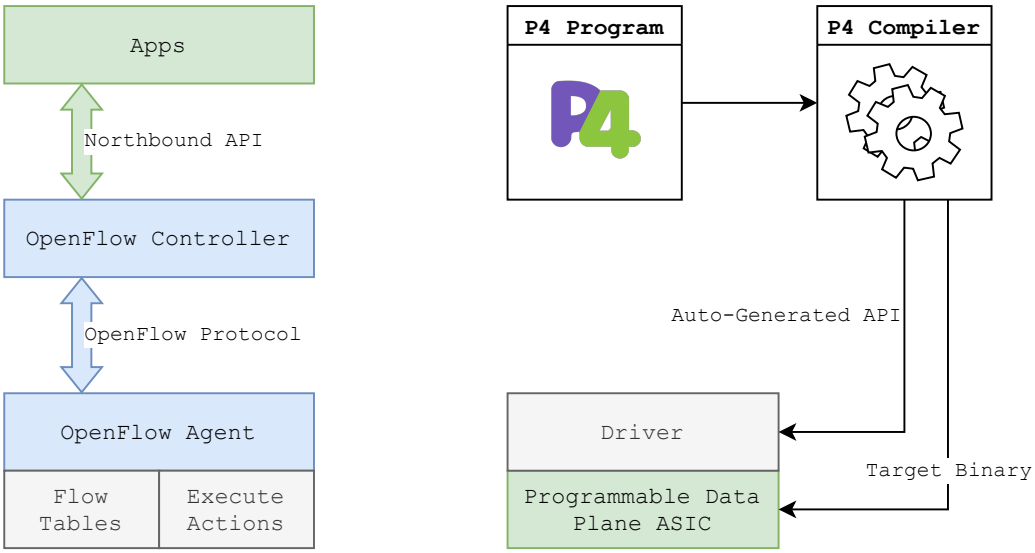

Figure 1.3: OpenFlow controllers communicate with applications through a Northbound API. The controller sends instructions to the control agents through the OpenFlow Protocol (left). P4 expands the customisability by directly programming the network device itself, rather than communicating through an intermediate controller (right). This diagram is modified from a blog post by the P4 language consortium [6].

1.1.5 Risk of Flexibility

This increased flexibility also increases the potential for mistakes. Small programming errors and bugs can lead to vulnerabilities that can be exploited [9]. Such exploits form a risk both for the operation of a network, as well as for the security of that network. This is made worse by P4's common usage in core networks and large data centres. At these places, a device being exploited or disabled can have a significant impact on the network [9]. Hence, being able to show that a P4 program is necessarily correct with regards to its intended meaning is a valuable area of research.

Verifying the operation of an application to be correct can be done in many ways. Symbolic execution, model checking, and fuzzing are examples that have already been applied for the verification of P4 applications [10, 11, 12, 13, 14]. Many of these verification approaches rely on the programmer to add assertions that can be checked at compile-time or at runtime. These assertions can be used to show that certain security properties are never violated.

However, we believe verification should rely on a strictly defined semantics of the language and should not require the programmer to manually annotate their code with assertions — which, being manual, is error-prone. Moreover, such a semantics can be used to reason about more than just specific programs. They can also be used to reason about properties of the entire language, such as verifying the correctness of optimisation routines, or as a point of reference against which to verify compiler or other implementations. An absolute and unambiguous semantics of P4 could be invaluable for the future of P4 and its applications.

1.1.6 Nano P4

In this thesis we present *Nano P4*: A novel approach to verifying P4 and its applications. We use the automated theorem prover Isabelle/HOL [15] to define a small-step semantics of P4 actions and a formalisation of the P4 parser state machines. We use this semantics to verify numerous properties of P4 and show that it can also be used to reason about it and verify transformations like optimisation routines. We show that Isabelle/HOL can be used to formalise the most recent version of P4 — $P4_{16}$ — and its applications.

1.2 Structure of this Thesis

This thesis is structured as follows. In section 1.3 we present a brief graphical overview of Nano P4 and its components. In chapter 2 we provide background information on P4 (section 2.1) and Isabelle/HOL (section 2.2). In chapter 3 we present Nano P4 through four sections: in section 3.1 we present our small-step semantics of P4 actions, in section 3.2 we show how this semantics can be used for the verification of security properties, in section 3.3 we verify the extension of this semantics by presenting program transformations in the form of optimisation routines, and finally in in section 3.4 we formalise the P4 parser state machines. We evaluate our complete formalisation effort in chapter 4. In chapter 5 we discuss Nano P4 and its limitations (section 5.1), and moreover discuss potential directions for future work (section 5.2). We discuss related work in chapter 6, before concluding in chapter 7.

1.3 Overview

The result of this thesis is Nano P4, consisting of around 4000 lines of code and over 200 lemmas and theorems. Nano P4 is open source and we published its source code freely on Github: https://github.com/Johanmyst/Nano-P4.

We formalise multiple parts of P4 in Nano P4. Nano P4 is not a complete verification, but a collection of these different components. Most notably, we formalise a substantial part of P4 actions and the sequential code they capture. This includes an executable small-step semantics, including a rigorous and strict typing system and numerous correctness proofs. We use Nano P4 to reason about properties like conditional termination, progress and preservation, and to perform reachability analyses. We also formalise a substantial part of P4's parser state machines including numerous correctness proofs. We include a separate big-step semantics for this and use this to analyse reachability properties and perform dead-state filtering. We provide a visual overview of Nano P4's components in figure 1.4.

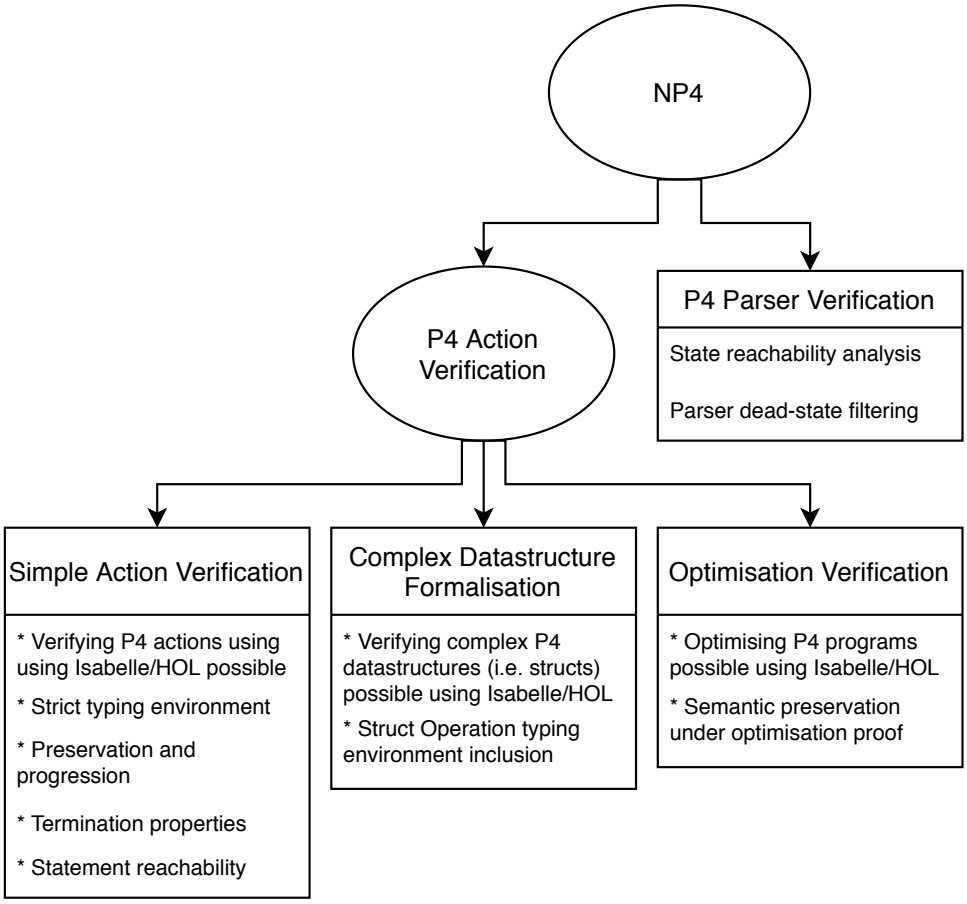

Figure 1.4: A graphical overview of the components of Nano P4.

We based Nano P4 on the most recent release of P4 at the time this project started: $P4_{16}$ version 1.2.0 (released on the 23rd of October, 2019) [16]. Since then a more recent version of $P4_{16}$ has been released: version 1.2.1 [17]. A significant change in this new version is a more detailed description of the semantics regarding invalid and uninitialised header accesses, and some additions were made to the semantics of P4. None of these changes impact the validity or effectiveness of Nano P4.

The work presented took a little over six months to complete, excluding writing this thesis, by a master student with no prior experience with either P4, Isabelle, or formal verification. The majority of the time went into understanding the theory behind formal verification and using Isabelle/HOL to formalise a programming language. The book *Concrete Semantics* by Gerwin Klein and Tobias Nipkow [18] has been invaluable in this process.

1.4 Contributions

More specifically, the contributions we make in this thesis are as follows:

- We provide a formalisation effort for $P4_{16}$.
- We provide a small-step semantics for a subset of P4 actions in the proof assistant Isabelle/HOL, including strict typing environment and numerous correctness proofs.
- We show that this semantics can be used to verify general properties such as type-soundness, progression, or preservation.
- We show that this semantics can be used to verify security properties such as prohibiting access of uninitialised variables, or prohibiting out-of-bounds accesses in P4 header stacks.
- We show that this semantics can be extended with optimisation routines that can be proven to maintain semantic equivalence across the transformation.
- We provide a formalisation of the P4 parser state machine.
- We show that this formalisation can be used to analyse reachability properties within the P4 parser state machine such as reducing the state machine to the minimal semantically equivalent state machine.

2

Background

In this chapter we present background on networking and P4 (section 2.1), as well as formalisation and the Isabelle/HOL proof assistant (section 2.2).

2.1 P4

2.1.1 The P4 Language

Programming Protocol-Independent Packet Processors — or P4 — assumes the hardware it runs on is programmable. A P4 program specifies the functionality of the forwarding device by defining what headers a device can understand, how incoming packets should be parsed, what the internal control flow looks like, and how packets are serialised back onto the wire again. The core logic of a P4 program is based on match-action tables: tables that bind specific keys to particular actions. Values from the header fields of incoming packets can be used as the key in these tables, thereby guiding the internal control flow. This makes the data plane programmable by specifying the structure of the forwarding tables and allowing for the processing of individual packets [19]. Note that a P4 program controls the data plane, not the control plane. A P4 program specifies the structure of the forwarding tables; these tables are populated by the control plane. With P4 the control plane resides in a separate *runtime controller* that communicates with the P4 program through an automatically generated API [19]. Also note that the control plane is distinct from the internal control flow logic of a P4 program.

2.1.2 Target Architectures

Similar to how a C program is compiled against a specific architecture to convert general C-constructs to architecture-specific constructs, a P4 program is compiled against a specific architecture. The *Protocol Independent Switch Architecture (PISA)* is an example of an architecture that P4 can run on (figure 2.1). Its hardware structure has a clear separation between parsing, performing control flow logic, and serialising — or *deparsing* — packets onto the physical medium again.

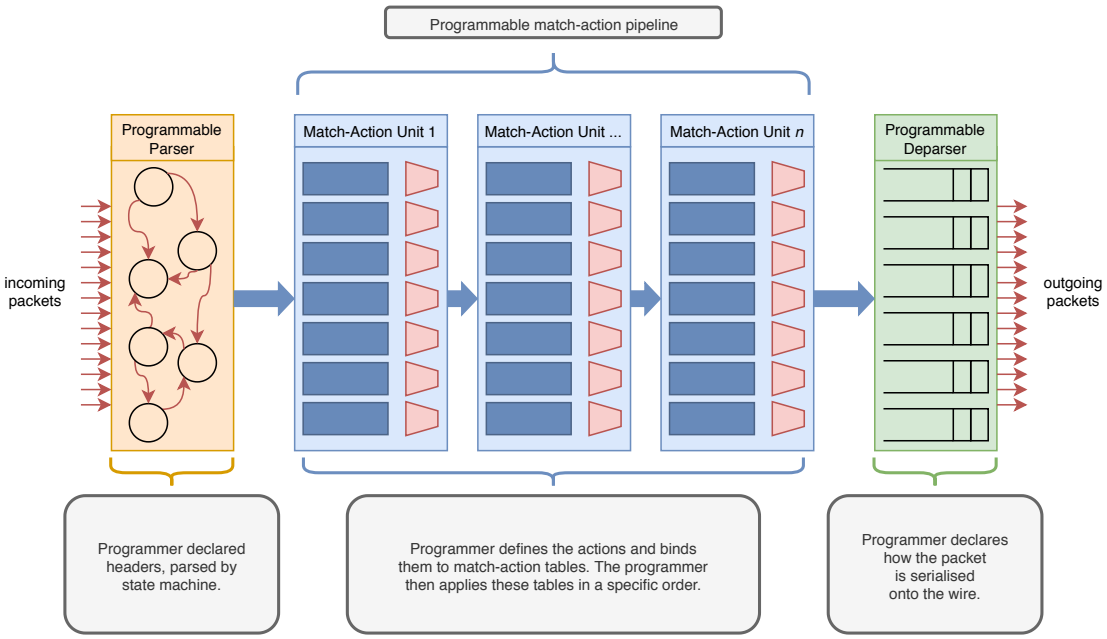

Figure 2.1: Protocol Independent Switch Architecture (PISA) is one target P4 can run on. The internal structure consists of a parser unit, a set of programmable match-action units set up in a pipeline, and a deparser [16]. This diagram is modified from a figure from a presentation by the P4 language consortium [20].

PISA is not the only architecture P4 programs can run on; the vendor of a P4-enabled device supplies a compiler that compiles general P4 code to produce a binary that can run on that specific target device. Alongside the compiler, a vendor supplies an *architecture description*. This describes exactly how the hardware is structured and which components are present. This way, hardware modules that are not necessarily present on all devices can still be used. For example, the architecture description might describe the presence of a hardware checksum module. These modules are defined in *external libraries*, and are accessible from code through *extern objects*. The official P4 specification defines the core set of instructions that all devices must implement; additional functionality is provided

through extern objects specified in the architecture description and external libraries. This description also defines parts and parameters that the official specification leaves open to the devices [16]. For example, the maximum supported bit-width can be different per device; each device is free to set its own maximum. Such values are described in the architecture description (figure 2.2).

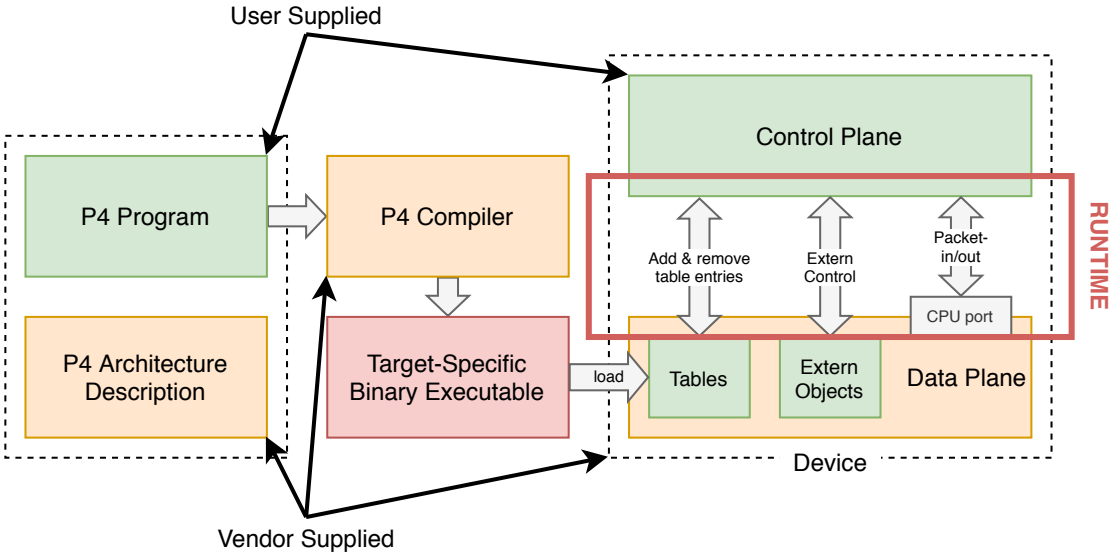

Figure 2.2: P4 allows a user to specify how a device should behave by defining the controller and the P4 program. Combined, these describe how the control plane and data plane are structured and behave. The vendor-supplied architecture description describes what the hardware looks like and what hardware components are present (e.g. hardware checksum units). The vendor-supplied compiler is used to compile the P4 program for the specific target [16]. This diagram is modified from a figure from a presentation by the P4 language consortium [20].

2.1.3 P4 Language Elements

The P4 language is a high-level, domain-specific, declarative programming language with a syntax similar to C. Because P4 is highly domain-specific it omits many general-purpose concepts that general-purpose languages like C have, such as loops or arbitrary arrays. Oppositely, P4 has a number of domain-specific concepts natively built-in. The main components of P4 are the parser logic and the control flow logic. These can make use of P4's expressions and data types. We show an overview of the core P4 language elements in figure 2.3.

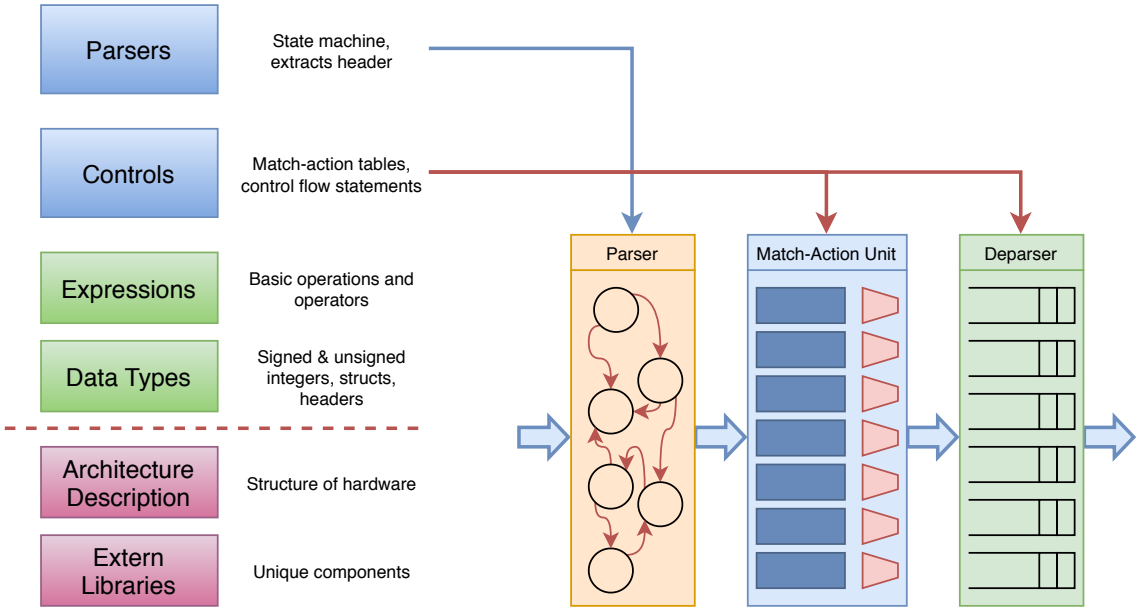

Figure 2.3: The P4 language is built on two components: the P4 parser logic and control flow logic. The parsers are built on state machines, and are used for serialisation and deserialisation. The parsers work with user-defined headers and header fields. The control flow logic is built on match-action tables that, when combined, form a control flow pipeline. Both components can make use of P4 data types and expressions [16]. The architecture description and extern libraries capture hardware-specific information and components. This diagram is modified from a figure from a presentation by the P4 language consortium [20].

P4 Headers and Parsers

P4 Headers P4 does not natively support any particular protocols, nor does it know any header formats out of the box. Instead, a programmer must specify exactly what the used headers look like using *header types*. These define both the structure of the header and its fields, including their sizes and order. These types can capture completely custom headers or headers used by existing protocols. We show an example of an Ethernet header in listing 1. The Ethernet header contains two 48-bit wide addresses and one 16-bit type field. P4 supports the use of custom type definitions through `typedef` or `type` keywords to make code easier to read and maintain. The struct in the example contains the final headers that the parser should fill in.

P4 Parser Packets arrive at a P4 device as a stream of raw bytes. Given the previous header type definitions, the P4 program now knows how to interpret the stream of bytes. An abstract state machine is used to incrementally parse this stream and create a header

```
typedef bit<48> EthernetAddress;

header Ethernet_h {
    EthernetAddress dstAddr;
    EthernetAddress srcAddr;
    bit<16>         etherType;
}

struct Parsed_packet {
    Ethernet_h ethernet;
}
```

Listing 1: A simple example of what the P4 representation of a standard Ethernet header could look like. The header contains two 48-bit wide fields for the source and destination addresses. The header also contains a 16-bit type field. The `typedef` is used to make the code easier to read. This example is modified from the official $P4_{16}$ specification [16].

object that contains the values of the incoming packet. A P4 parser always starts in the `start` state. Within a state headers can be extracted using the `extract()` method and transitions can be defined using the `transition` keyword. The values extracted from the headers can be used to conditionally guide the transitions, typically through a `select()` statement. This allows these parsers to also parse nested and encapsulated packets into custom header objects. In listing 2 we show how the Ethernet header we present in listing 1 can be extracted into a usable `Parsed_packet` struct. Values of the header can now be used to guide the program logic. For example, the value of the `etherType` field is now accessible and a conditional transition can use its value. In the example, we accept a packet if and only if the `etherType` matches `0x0800` (listing 2). This type corresponds to an IPv4 packet inside of the Ethernet frame. If no transition is specified, the packet is automatically rejected. A packet in a P4 parser can either be accepted, processed, and forwarded; or rejected and dropped.

P4 Controls

Actions P4's computational logic is contained in `action` objects. These actions contain the imperative code that performs the actual logic and manipulations on the headers of a packet. Actions can be used to, amongst others, decrement the *Time-To-Live (TTL)* field in an IP header, or set the next hop address based on the destination address of a packet. We show an example of an action that decrements the TTL field and sets the next hop's address in listing 3.

```
parser Parser(packet_in b, out Parsed_packet p) {
    state start {
        b.extract(p.ethernet);
        transition select(p.ethernet.etherType) {
            0x0800: accept; // Valid Ethernet header
            // No default rule: other packets rejected
        }
    }
}
```

Listing 2: The Ethernet header we defined in listing 1 can be extracted into the usable program struct `Parsed_packet` through a parser state machine. This state machine extracts the headers from the incoming byte stream, and makes transitions based on the data contained in them. In this case, an Ethernet frame will only be accepted if it is carrying an IPv4 packet and its `etherType` field thus equals `0x0800`. This example is modified from the official $P4_{16}$ specification [16].

Match-Action Controls Simply processing all packets identically is often insufficient to achieve the desired behaviour. Instead, certain packets should trigger certain actions. To achieve this, P4 combines actions in a *match-action table*. These tables are a generalisation of traditional switch tables. They provide a link between a particular key and a corresponding action. This key can be constructed from different sources such as: entries in the headers, meta-data that the P4 program tracks with the header, or even stateful external objects. For example, we show how such a table can be constructed for IPv4 routing in listing 3. Two actions are defined: one to drop a packet and another to set the next hop address including setting the corresponding outgoing port interface. The key is a *longest-prefix match* on the destination address. The table itself is populated by the runtime controller. This controller inserts entries that link a particular key to a particular action. In this case, the keys will be bound to the `Set_nhop()` action, while undefined keys are implicitly bound to the `Drop_action()` action. P4 refers to the entire table as a singular *match-action unit*.

Multiple of these match-action units can be chained together to form complex behaviours and logics. Such a match-action pipeline is captured in the *control flow objects*. These *controls* define which actions and tables are defined and in what order they are *applied*. Between applications of a particular match-action table, error conditions can be checked, such as whether the output port was set to the `DROP_PORT` (listing 4).

```
action Drop_action() { outCtrl.outputPort = DROP_PORT; }
action Set_nhop(IPv4Address ipv4_dest, PortId port) {
    nextHop = ipv4_dest;
    headers.ip.ttl = headers.ip.ttl - 1;
    outCtrl.outputPort = port;
}

table ipv4_match {
    key = { headers.ip.dstAddr: lpm; }  // Longest-prefix match
    actions = { Drop_action; Set_nhop; }
    size = 1024;
    default_action = Drop_action;
}
```

Listing 3: Actions capture the imperative code that makes up the core packet-processing logic of P4 programs. These actions are used in match-action tables to link certain actions to specific keys. At runtime, often based on extracted headers, a specific key is matched against and the corresponding action is executed [16].

In figure 2.4 we show how such a chaining can be used in a simple IPv4 forwarding device. Within this flow there are multiple points at which the packet can be dropped, such as if a parser error has occurred. Then, based on the IPv4 destination address, the next hop and output port are set and the TTL is decremented. In the next table the TTL is checked. Then, the destination MAC address is set in the Ethernet frame. Lastly, the source MAC address is set as the current switch.

P4 Deparsers

A P4 program also has to re-serialise a packet before this packet can be output onto the physical medium. P4 uses the control objects to achieve this. Similar to how headers are extracted in the parser using the `packet.extract()` method, packets are serialised using the `packet.emit()` method. This process can also involve some logic, such as setting the checksum to a new value if the entries of the packet have been changed and are still valid. We show an example of a deparser that emits IP and Ethernet headers in listing 5.

P4 Type System

$P4_{16}$ is a statically typed language with a strict typing environment. Programs that do not pass the type checker are rejected by the official $P4_{16}$ specification. P4 has a limited number of types built in, with strictly defined interactions between them: "*P4 prefers to*

```
control Pipe(inout Parsed_packet headers,
                in error parseError,
                in InControl inCtrl,
                out OutControl outCtrl) {
    action action_1() { ... }
    ...
    action action_n() { ... }

    table table_1 { ... }
    ...
    table table_n { ... }

    apply {
        table_1.apply();
        ...
        table_n.apply();
        // Optional check for error conditions
        if (outCtrl.outputPort == DROP_PORT) return;
    }
}
```

Listing 4: Control flow objects in P4 contain multiple actions and tables, which are applied in a specific order. This creates a *pipeline* of match-action units, which can create complex behaviours [16].

forbid, rather than to surprise" [16]. This means that operations between types are strictly defined and only a limited number of implicit actions exist. For example, in P4, using an integer in a conditional statement is invalid and this integer must be explicitly cast to a Boolean value. Through such restrictions P4 aims to eliminate as much ambiguity as possible.

There are few data types natively built into P4. Due to P4's domain-specific nature it has no need for strings or string manipulations, arbitrary lists or list comprehensions, pointers or pointer-based objects, or other complex data structures. Instead, P4 includes a number of domain-specific data types such as headers, parsers, and match-action tables. P4 does allow for the specification of custom types through the `typedef` and `type` statements. The `typedef` statement introduces a type synonym and functions purely as a new symbol to address the same type. The `type` keyword does introduce a non-synonymous type, but only comparison and cast operations are supported for these. These custom definitions are aimed at making P4 code easier to read, as well as preventing unnecessary mix-ups through type confusions.

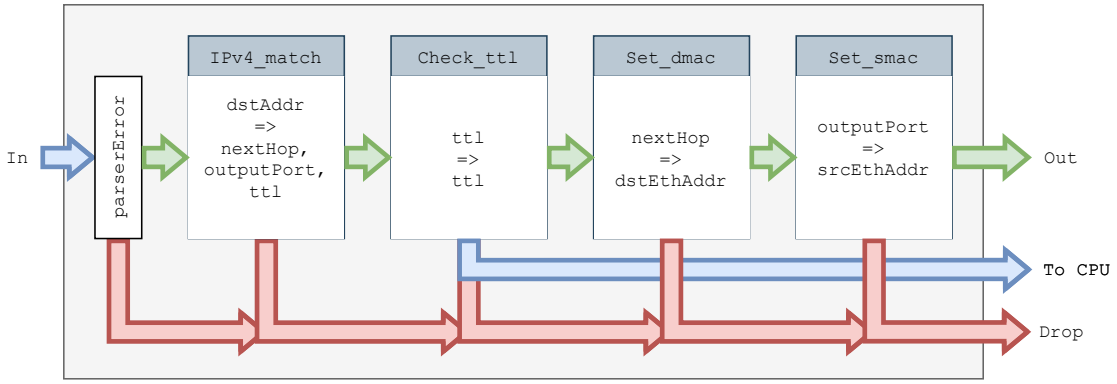

Figure 2.4: Multiple match-action tables can be combined to form complex control flows. For a simple IPv4 switch, first the next hop address is set and the TTL decremented. The TTL is decremented if and only if the destination address matches a key in the table. Then, the TTL field is checked. Then, based on the next hop, the destination Ethernet address is set and the new source address is set to the port through which the packet leaves the switch [16].

2.1.4 $P4_{14}$ and $P4_{16}$

P4 is a young language that still undergoes development and changes. One major recent change was the version change from $P4_{14}$ to $P4_{16}$. This change is significant because it sacrifices backwards compatibility to become a more stable language. The change was made because $P4_{14}$ had many architecture-specific components natively built into the language, making $P4_{14}$ unnecessarily complex. When a vendor introduced new hardware or changes to a device's structure, these changes had an effect on the entire $P4_{14}$ language. $P4_{16}$ separates these architecture specific components explicitly, making $P4_{16}$ far simpler relative to $P4_{14}$. This is done by moving all architecture-specific components into the architecture description [16]. The architecture description and library are supplied by the vendor. This reduces the core of P4 significantly and allows vendors to change their architecture rapidly, without requiring a change to the core of P4, making P4 stabler (figure 2.5).

2.1.5 Formalising P4

Which approaches to formalising a programming language are suitable depends greatly on the design of the language itself. P4 has a number of characteristics that influence the formalisation process significantly.

```
control Deparser(inout Parsed_packet p, packet_out b) {
    Checksum16() ck; // Declare checksum unit
    apply {
        b.emit(p.ethernet); // Emit Ethernet header
        if (p.ip.isValid()) {
            ck.clear();               // Prepare checksum unit
            p.ip.hdrChecksum = 16w0; // Clear checksum
            ck.update(p.ip);          // Compute new checksum
            p.ip.hdrChecksum = ck.get(); // Set checksum
        }
        b.emit(p.ip); // Emit IP header
    }
}
```

Listing 5: P4 deparsers make use of the same control objects that capture the core match-action pipeline in the P4 program. The deparser emits a byte stream using the `packet.emit()` method. Such deparsers can also include logic, for example to update checksum fields and such [16].

No Complicated Logic

P4 is syntactically similar to C but omits concepts that are not necessary for the networking domain. For example, P4 has no notion of loops. This simplifies an application, because each instruction necessarily follows the previous sequentially. Furthermore, P4's memory space is simpler than that of C. Though P4 does have some data types that C does not have, P4 does not have a notion of pointers — neither for functions nor for data types. This means P4 has a flat memory space and thus the program state is straightforward as a result.

P4 Concurrency Model

P4 only supports parallelisation at the packet-level: each packet is processed in its own thread with its own local storage. Because of this, there is no complex interleaving of concurrent executions that can influence each other. This does not apply to the extern objects, which are global across all threads. If a register is accessible through an extern object, operations on it are subject to data races. We discuss some potential ways of still taking them into consideration as future work in subsection 5.2.

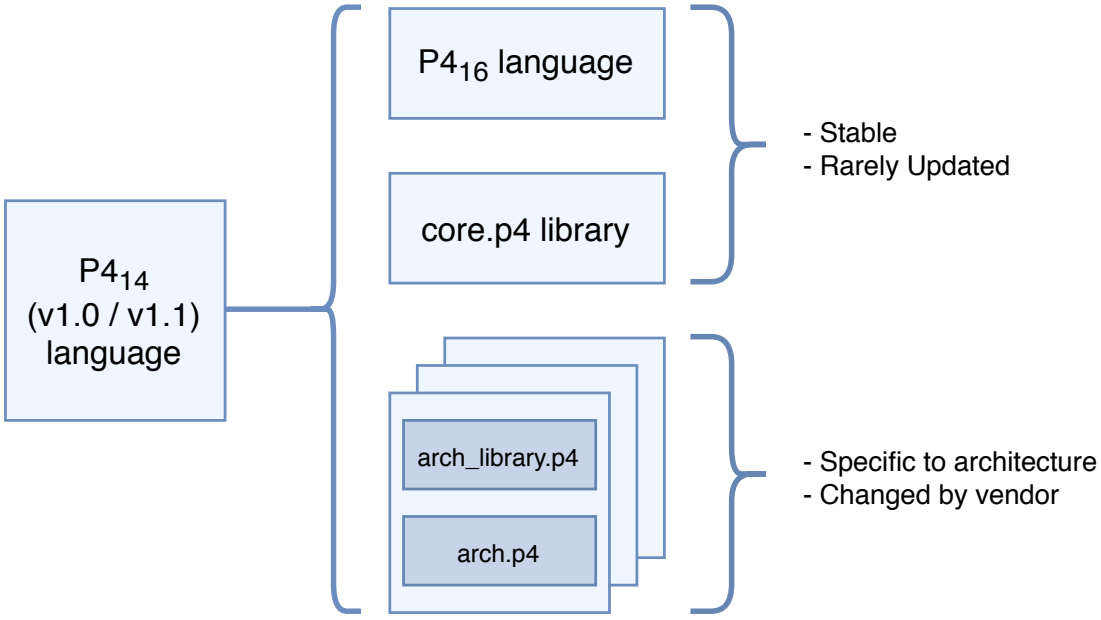

Figure 2.5: $P4_{14}$ is the older version of P4 that is no longer supported. $P4_{14}$ is a far larger language containing nearly 70 keywords, some of which are architecture-specific. With the introduction of $P4_{16}$ many of these architecture-specific components have been moved to architecture-specific libraries and only a small core P4 language remains, which is applicable across all target devices [16]. This diagram is taken and modified from the official P4 specification [16].

Data Plane, not Control Plane

A P4 program specifies what the data plane looks like and the compiler provides the API through which the control plane can communicate with the data plane. However, the controller itself is not defined by the P4 program. Consequently, the full space of entries the forwarding tables will be populated with, is not known at compile time. We discuss some potential future approaches to verifying properties of match-action tables, or the control plane, in subsection 5.2.

2.2 The Isabelle Proof Assistant

2.2.1 Proof Assistants

Proof assistants, or *interactive theorem provers*, are tools that assist with the development of formal proofs. Such tools include various features to aid the development process of a proof, as well as provide mechanisms to check the final proof for correctness. Some proof

assistants provide a rigorous typing system, or provide automatic model checkers to find counter-examples. Some can even automatically prove certain properties. These tools help to make writing a clear and structured formal proof far easier than writing the proof by hand; thus these tools are widely used for a spectrum of applications.

Many proof assistants exist; some well-known examples are Coq [21], Isabelle [15], and Lean [22]. These are all based on different implementations with different foci and tools offered. We are particularly interested in the Isabelle proof assistant due to its flexibility and large degree of proof automation. Moreover, the quantity of available literature for Isabelle is also a significant advantage [18]. In this section we provide background on the Isabelle proof assistant, extended with *Higher Order Logic (HOL)*. We also discuss how Isabelle/HOL can be used for the formalisation and verification of programs and programming languages.

Why Isabelle?

The Isabelle proof assistant has enjoyed common use since its publication in 1986 [23]. It is built on a small logical core, which makes it trustworthy. This small logical core can be extended to include numerous different logics, like *First-Order Logic (FOL)*, *Higher-Order Logic (HOL)*, or *Zermelo–Fraenkel set theory (ZFC)*. The most commonly used extension is HOL: Isabelle/HOL. Essentially, Isabelle/HOL is a marriage of functional programming and formal logic, which allows for reasoning about higher order logic properties and provides tools to define complex data types, functions, and inductive predicates. This has made Isabelle/HOL a powerful tool for the formalisation of numerous mathematical [24, 25] and computer science related [26, 27, 28] problems. Moreover, it has been used for the formalisation of entire programming languages like Java [29], as well as the formalisation of systems that use these languages, such as compilers [30].

Isabelle also offers a number of tools and plugins that simplify the development process of a formal proof. It features many built-in proving methods, such as a term rewriting simplification engine, tableaux prover, or algebraic provers. Furthermore, Isabelle also features a complete proof-automation interface through *Sledgehammer*, which is a tool that invokes additional external proof engines based on any logical system. For instance, Sledgehammer uses *Satisfiability Modulo Theories (SMT)* solvers like CVC4 [31] and Z3 [32] and external automated theorem provers including E [33] and SPASS [31]. Isabelle also has tools to automate finding counter-examples through two model generators: Nitpick [34]

and Nunchaku [35]. These can be called at any point during a proof, helping to quickly find counter-examples to ensure the direction of the proof is correct. Because of these advantages we use Isabelle/HOL to develop our formalisation of P4 and its applications.

2.2.2 Using Isabelle

Data Types

Isabelle/HOL combines functional programming and formal logic. A user can define complex custom (and possibly even recursive) data types. In listing 6 we show how such recursive data types can be used to define a *polymorphic* list type called `list`. The list is polymorphic and can thus hold entries of any type, as denoted by the *type variable* `'a`. Using type variables means definitions can be general and do not need to be repeated for different types.

The list data type is defined for two cases: the empty list, `Nil`, and a consecutive value and further list, `Cons`. The terms `Nil` and `Cons` are **constructors**. The `Nil` case contains no values; the `Cons` case contains two values: a value of type `'a`, and another list. A list containing three entries `x1`, `x2`, and `x3` is then captured by `Cons x1 (Cons x2 (Cons x3 Nil))`. Note that the final entry is necessarily the empty list.

```
0 datatype 'a list = Nil |
1     Cons 'a "'a list"
```

Listing 6: Lists can be captured using Isabelle/HOL data types by defining the empty list (Nil) and an element with a list following it. This list definition is *polymorphic*, meaning it can contain any type. This is denoted by the *type variable* `'a` [18].

Isabelle/HOL also has a notion of optional data types through the polymorphic `option` data type (listing 7). This data type adds a new element to an existing type `'a`: `None`. The `None` constructor captures the empty case. If the optional data type does have a value of type `'a` it is captured by `Some 'a`. This explicitly separates the empty case from the case where a value exists. Such optional definitions can easily capture concepts like lookup tables, which might not return a value if the key is not found in the table.

```
datatype 'a option = None | Some 'a
```

Listing 7: The `option` data type explicitly introduces an emty element to an existing type: `None`. This allows for the explicit separation between a type containing information and the empty type [18].

Functions and Inductive Predicates

A user can also write functions and define *inductive predicates*, which may also be recursive. Such definitions can capture complex behaviours and also use custom data types. For example, we define a data type to capture the natural numbers in listing 8. Our data type definitions have two constructors again: One for the zero value and another for a successor of this value. The natural number 3 is then captured as `Suc (Suc (Suc 0))`. Note that the constructor for the zero value, `0`, is a constructor, not an actual value. Note also this redefines Isabelle/HOL's built-in notion of natural numbers, which support the usage of numerals like 3 instead of having to type `Suc (Suc (Suc 0))`.

```
datatype nat = 0 | Suc nat
```

Listing 8: Isabelle/HOL custom data types can be used to capture natural numbers. These numbers can either be zero, or a successor of zero.

We can then use this data type, for example, in a function to determine whether a number is even or not. In listing 9 we define this function by case-distinction over our natural number data type. If the input matches the zero-constructor `0`, we return `True`. In case the input matches a single successor (`Suc 0`), the function returns `False`. Any subsequent cases are recursively reduced by peeling off two successors. After all, if n is even, $n + 2$ — or `Suc (Suc 0)` — must also be even. These three cases capture all possible values of the `nat` data type that we defined.

```
fun evn :: "nat ⇒ bool" where
    "evn 0 = True" |
    "evn (Suc 0) = False" |
    "evn (Suc (Suc n)) = evn n"
```

Listing 9: A functional function definition can define evenness for natural numbers by case-distinction over the cases of the natural number. Two base cases exist: either the natural number is an even zero, or an uneven successor of zero. All further cases can be captured by subtracting two, and calling the evenness function recursively.

In listing 10 we show an equivalent *inductive predicate*. An inductive predicate essentially defines "rules" for the cases where the predicate is *defined*. In listing 10 we define two named rules in the inductive predicate `ev`: `ev0` and `evSS`. Rule `ev0` defines the case where the given natural number matches the case `0`; evenness is defined and true for the number zero. Rule `evSS` defines the rule for the double successor of a number for which the predicate is defined. In other words: given that n is even, $n + 2$ is also defined as even. This notion of prerequisites — "given that X have Y" — is captured in Isabelle with the long right arrow: $X \Longrightarrow Y$. Multiple prerequisites can be chained together: writing $X \Longrightarrow Y \Longrightarrow Z$ is equivalent to writing $X \wedge Y \Longrightarrow Z$.

```
0 inductive ev :: "nat ⇒ bool" where
1     ev0:    "ev 0" |
2     evSS:   "ev n ⟹ ev (n + 2)"
```

Listing 10: Inductive predicates can be used to capture evenness of natural numbers. If the input natural number equals zero, it is even by definition and the predicate is defined for this case. Any number $n + 2$ is even if and only if the natural number n is also even.

This concept translates intuitively to the deductive notation. We show this notation for the two inductive predicate rules `ev0` and `evSS` in figure 2.6. Instead of using the long right arrow notation, prerequisites are shown above the horizontal line and the conclusion below it.

$$\frac{}{ev0}\ \textbf{ev0} \qquad\qquad \frac{evn}{ev(n+2)}\ \textbf{evSS}$$

Figure 2.6: We can use logical deduction notation to depict Isabelle inductive predicates intuitively. This shows the two rules for the evenness definition given in listing 10.

Proving Equivalence Lemmas

Intuitively, we would expect the functional and inductive predicate definitions of evenness to be equivalent: Having `ev n` means we must also have `evn n` (lemma 2.2.1) and vice versa (lemma 2.2.2).

Lemma 2.2.1 *ev n* $\Longrightarrow$ *evn n*

Lemma 2.2.2 *evn n* $\Longrightarrow$ *ev n*

But rather than intuitively knowing these lemmas hold, we want to undeniably prove that these lemmas hold. This is where the logic side of Isabelle/HOL applies. We can write lemmas and theorems in Isabelle using the function and inductive predicate we defined in listings 9 and 10. For example, we show how we can write the equivalence relation from lemma 2.2.1 in listing 11. We can then start *applying* proof methods to work towards proving the lemma. In listing 11 we apply an induction over the definition of the inductive predicate `ev`. This produces two *sub-goals* in the *proof state*. If the sub-goals can be proven, the complete lemma or theorem is also proven. Here, one sub-goal exists for each rule defined in the inductive predicate:

1. `evn 0`
2. $\forall$ `n. ev n` $\wedge$ `evn n` $\Longrightarrow$ `evn (n + 2)`

The first case is trivially true: `evn 0` returns `True` directly and thus the first sub-goal holds. The second sub-goal states that for any n — given that we have `ev n` and `evn n` — we must also have `evn (n + 2)`. The inductive rule `evSS` from listing 10 defines that if we have `ev n`, we necessarily also have `ev (n + 2)`. Therefore, we can prove this sub-goal through simplification. We apply Isabelle/HOL's simplification engine through `simp_all` (listing 11).

```
0 lemma ev_implies_evn: "ev n ⟹ evn n"
1     apply (induction rule: ev.induct)
2       apply (simp_all)
3     done
```

Listing 11: This lemma states that if the inductive predicate `ev` is defined for `n`, the functional definition of `evn` must return `True`.

This notion of applying proof methods can quickly prove small lemmas and theorems. However, for larger proofs the structure of a proof can be hard to follow. Isabelle features its own formal proving language: *Intelligible Semi-Automated Reasoning (Isar)*. This language is closer to the pen and paper style of formal reasoning, making it easier to follow. We prove the remaining direction captured in lemma 2.2.2 using Isar in listing 12. We start the proof by induction over n, following the recursive definition of `evn`. This produces three sub-goals that have to be proven, one for each case of the definition of `evn`:

1. `evn 0 ⟹ ev 0`
2. `evn (Suc 0) ⟹ ev (Suc 0)`
3. `∀ n. (evn n ⟹ ev n) ⟹ evn (Suc (Suc n)) ⟹ ev (Suc (Suc n))`

The first sub-goal is again trivial and holds by the definition of rule `ev0` from the inductive predicate. We can tell Isabelle to use this rule explicitly using the `using ev0 by auto` keywords, which adds the rule `ev0` to the automated proving engine `auto`.

The second sub-goal also holds arbitrarily: If the prerequisite of a logical consequence is false, the statement necessarily holds. In this case `evn (Suc 0)` returns `False` and thus the second sub-goal holds. No additional rules are required and this can be proven directly by Isabelle's automatic prover.

The third sub-goal has two prerequisites. Firstly, `evn n` must imply `ev n`. Secondly, we must have `evn (Suc (Suc n))`. Given these two prerequisites we must also have `ev (Suc (Suc n))` for the third sub-goal to hold. From these prerequisites we have that `evn (Suc (Suc n))` holds and that `evn n ==> ev n` holds for any n. From the definition of `evn` we know that if we have `evn (Suc (Suc n))` we must also have `evn n`, which implies that we must also have `ev n`. By the definition of the predicate `ev` we therefore also know that we have `ev (n + 2)`, or `ev (Suc (Suc n))`. We can let Isabelle prove this by adding the definitions of the rule `evSS` to its automatic proof engine (listing 12).

```
0  lemma evn_implies_ev: "evn n ⟹ ev n"
1  proof (induction n rule: evn.induct)
2      case 1
3      then show ?case
4          using ev0 by auto
5  next
6      case 2
7      then show ?case
8          by auto
9  next
10     case 3
11     then show ?case
12         using evSS by auto
13 qed
```

Listing 12: This lemma states that if the functional definition of `evn` returns `True`, the inductive predicate `ev` is defined for `n`. This prove uses Isar to structure the proof clearly.

2.2.3 Isabelle and Programming Language Formalisation

These data types, functions, and predicates are not just useful for reasoning about mathematical properties. They can also be used to capture the meaning — the *semantics* — of a programming language. We refer to this process as *formalisation*. A formal semantics is an unambiguous and rigorous definition that captures the entire meaning and operations of a language (or a subset of a language) and checks it using a proof assistant. This can be used to reason about and *formally verify* properties of the language — prove that they necessarily hold and are never violated. Generally speaking, formalising a language using Isabelle/HOL can be roughly separated into four steps:

1. Specifying the syntax of the language.
2. Defining a notion of *program state*.
3. Assigning concrete values to the syntax.
4. Giving meaning to statements: Defining the semantics of the language.

Syntax

Isabelle/HOL data types can capture the syntax of a language. For example, we show how a data type `expr` can capture arithmetic expressions in listing 13. We define three constructors: `N`, `V`, and `Plus`. `N` contains a natural number, `V` contains a variable name, and `Plus` contains two recursive sub-expressions. Their intended meaning is evident: `N n` captures a constant value n, `V v` captures a variable v, and `Plus e1 e2` intends to sum the values of `e1` and `e2`.

We also show how a data type can be used to capture the syntax of a statement in listing 13. We define statements through three constructors: `SKIP`, `Assign`, and `Seq`. Their intended meaning is straightforward: `SKIP` is a no-op, `Assign` assigns the value of an arithmetic expression to a variable, and `Seq` composes two statements sequentially.

```
0 datatype expr = N int | V string | Plus expr expr
1 datatype stmt = SKIP | Assign string expr | Seq stmt stmt
```

Listing 13: Isabelle/HOL data types can be used to capture the syntax of a language. In this case, an `expr` data type captures the syntax of expressions, while `stmt` captures the syntax of statements. We modified this example from *Concrete Semantics* [18].

These data types capture the syntax of another language, but it is not the regular syntax as written in a code file. The syntax one sees written in strings is referred to as the *concrete syntax*: `x = 3 + 2 + 1`. We instead capture the *abstract syntax* with these Isabelle/HOL data types. Capturing the concrete syntax `x = 3 + 2 + 1` using abstract syntax would yield: `Assign (V "x") (Plus (N 3) (Plus (N 2) (N 1)))`. An abstract syntax is often less ambiguous than its concrete counterpart, because nesting rules and such are made visually explicit. This is plain to see in our abstract syntax example, because the second `Plus` expression is captured as a sub-expression of the first. Such an abstract syntax could also be represented using an *Abstract Syntax Tree (AST)*.

Program State

The concrete syntax `x = 3 + 2 + 1` intends to store the value of $3 + 2 + 1$ into the named variable x. After the assignment, the value for x is "remembered". Such *remembered information* is captured in a *program state*. What the state of a program looks like and how it can change is heavily influenced by the language design. In our running example a mapping from names to integer values suffices (listing 14).

```
0 type_synonym state = "string ⇒ int"
```

Listing 14: How state can be modelled is heavily influenced by the language design. For simple languages a mapping from names to values suffices. In this example, matching the syntax we defined in listing 13, a mapping from a name to an integer value suffices. We modified this example from *Concrete Semantics* [18].

Concrete Values

Our abstract syntax represents $x = 3{+}2{+}1$ as `Assign (V "x") (Plus (N 3) (Plus (N 2) (N 1)))`, but these constructors have no actual meaning yet. They are merely labels for the constructor that takes two further expressions; Isabelle/HOL does not know that we intend the result of this expression to be the sum of the values of its sub-expressions. We have to assign this *concrete meaning* to these operators. In listing 15 we define a function that takes an expression and a program state, and computes the result. In this function we map an expression containing a constant value `N n` directly to n. We map a variable `x` to the value that name `x` is mapped to in state `s`. Note that `x` is a variable here, and can contain any variable name. The actual string "x" is denoted as `"x"`. Lastly, we

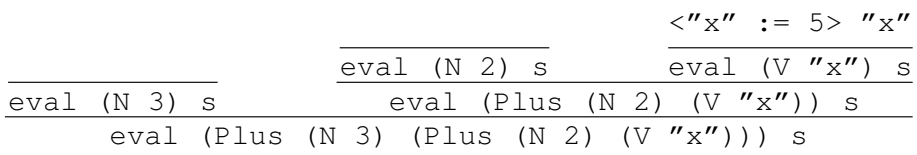

Figure 2.7: We can capture the evaluation of the expression `eval (Plus (N 3) (Plus (N 2) (V "x"))) <"x" := 5>` in a derivation tree. Such a tree shows all steps of the derivation, and exists only for valid expressions. Note that `s` is equal to `s("x" := 5)` in this example.

map the value of constructor `Plus e1 e2` to the actual sum of the two sub-expressions `e1` and `e2`. We use our function recursively to compute the value of both of these sub-expressions.

```
0 fun eval :: "expr ⇒ state ⇒ int" where
1     "eval (N n) s = n" |
2     "eval (V x) s = s x" |
3     "eval (Plus e1 e2) s = eval e1 s + eval e2 s"
```

Listing 15: A function can be used to assign concrete values to an abstract syntax. In this case, the value of the syntax constructor `N` is mapped directly to its value and a variable access with name `x` is taken from the state `s`. The value of the sum constructor `Plus e1 e2` is mapped to the sum of the two sub-expressions. We modified this example from *Concrete Semantics* [18].

Correct State Evaluating the expression `Plus (N 3) (Plus (N 2) (N 1))` yields the value 6. An expression containing a variable requires a state to be passed. For example, `Plus (V "x") (N 2)` requires the value of `"x"` to be known. If we call `eval (Plus (V "x") (N 2)) <>`, where `<>` is the empty state, no evaluation is possible. If we call the function again with a state that maps `"x"` to the integer value 5, `eval (Plus (V "x") (N 2)) <"x" := 5>`, we would see the return value 7. We can also depict this derivation in a *derivation tree*. Such a tree exists only if the derivation is valid. We show an example of such a derivation tree for the expression `eval (Plus (N 3) (Plus (N 2) (V "x"))) <"x" := 5>` in figure 2.7. Such a tree can be used to analyse the evaluation, for example to find where the derivation got stuck.

The Semantics

As statements are executed the program state can change: After executing the assignment statement `Assign "x" (N 3)` from the empty state, the value of x is now mapped to 3. How statements modify the state is their semantics. In listing 16 we show how we define a function that captures how statements influence a program state. This function takes a statement and program state pair and returns the resulting state. For the `SKIP` statement the state remains unchanged. For the assignment statements `Assign vname expr`, the state is updated with the value of `expr` as the new value for named variable `vname`. Note that this uses our previously defined evaluation function to get the value of `expr`. Lastly, for sequential execution `Seq stmt1 stmt2` the first statement is executed with state `s` to get the resulting state of that statement. This is used as the input state for the execution of statement `stmt2`.

```
0 fun step :: "(stmt * state) ⇒ state" where
1     "step (SKIP, s) = s" |
2     "step (Assign vname expr, s) = s (vname := (eval expr s))" |
3     "step (Seq stmt1 stmt2, s) = step stmt2 (step stmt1 s)"
```

Listing 16: A simple example modified from *Concrete Semantics* [18] that shows how Isabelle functions can be used to take the data types defined in listing 13 and assign concrete values to them. This gives a semantics to the syntax.

The definition we provide in listing 16 is a *big-step semantics*: The execution progresses from the initial state to the final state in "one big step". Calling the function `step` with a statement and a state, executes all statements and results in a singular final state. This is also true for recursively nested statements. Alternatively, a *small-step semantics* could be defined. Such a semantics progresses in individual steps. Instead of executing all nested statements, only the lowest nested statement is executed as a single step. Such a small-step definition is more expressive, as it allows us to analyse the state at any point during the execution of the statements. A big-step semantics can either yield a final state, or not yield a final state. Therefore, a big-step semantics cannot differentiate between non-termination, a crash, or an invalid statement (figure 2.8).

Because we defined this semantics as functions, we have an *executable formal semantics*. Isabelle can run these functions to compute an actual value, as well as generating code [36] in languages like Ocaml [37], SML [38], Scala [39], and Haskell [40] that can subsequently be executed.

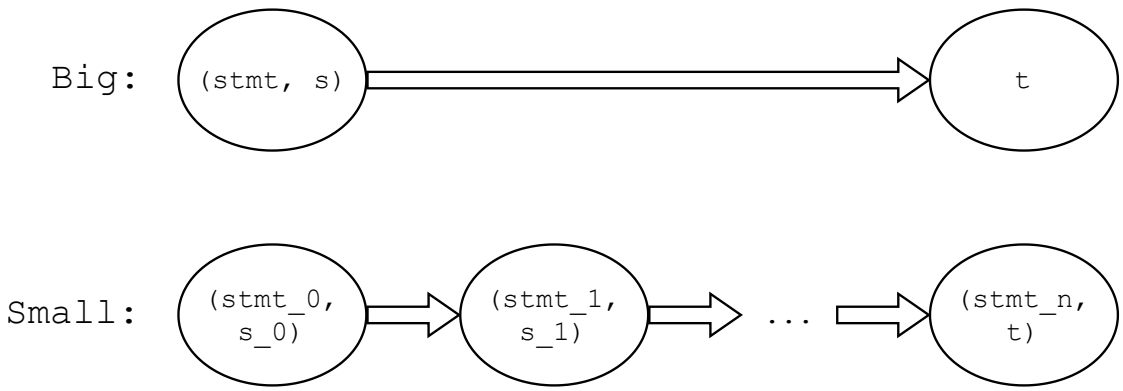

Figure 2.8: A semantics can be defined as a **big-step** or *small-step* semantics. The big-step semantics executes an entire program in one step — it returns the final state given an initial state and a statement. Oppositely, the small-step semantics progresses in a step-by-step manner. The latter allows for the analysis of intermediate states.

3

Nano P4

In this chapter we present our main contribution: Nano P4, a formalisation of a subset of P4. We discuss the distinct components of Nano P4 separately. In section 3.1 we present the main component of Nano P4: the formalisation and verification of P4 actions. In section 3.2 we use this formalisation and semantics to show that Nano P4 can be used to verify security properties of different classes. In section 3.3 we show how the same formalisation and semantics can be used to verify semantic preservation across program transformations through verified optimisation routines. Lastly, in section 3.4 we present and discuss our formalisation of P4 parser state machines.

3.1 Verification of P4 Actions

In this section we present and discuss the main focus of Nano P4: the formalisation of P4 actions. We discuss how Nano P4 captures the syntax of P4 actions, as well as how Nano P4 maps this syntax to concrete values. We also discuss the executable small-step semantics we provide, as well as the strict typing system we define.

3.1.1 P4 Action Syntax

The official P4 language specification [16] includes a rigorous description of P4's syntax, including also the YACC/Bison grammar that is used to parse a P4 program. Combined, these offer a detailed description of the concrete syntax of P4. However, the syntax we are interested in is not the concrete syntax; for the formalisation of P4 it matters little whether lines are ended with a semicolon or not. The syntax we are interested in is the abstract syntax.

```
datatype baseType = BBOOL bool
    | BUINT nat (* Unsigned integer *)
    | BSINT int (* Signed integer *)
    ...
    (* Struct: typename * membername * value (heterogeneous) *)
    | DSTRUCT "(identifier * vname * baseType option) list"
```

Listing 17: Isabelle/HOL's native types can be used to directly capture P4's base types. Isabelle/HOL's list structures can be used to capture the syntax of a P4 struct, and thus other derived types, as well.

Base Types

The most basic elements of a P4 program are the *base types*, like signed or unsigned integer values. These can be captured directly by Isabelle/HOL's native data types (listing 17).

These base types can be used to form more complex *derived types*, such as *headers*, *enums*, *tuples*, or *structs*. Essentially, these are nothing more than specifically structured combinations of base types. The base types form a collection of typed and named entries, that can possibly contain a value. We can use Isabelle/HOL's built-in list structures to capture this syntax (listing 17). Note that this also supports nesting of these derived types. This follows P4's well-defined and strict nesting rules; structs, for example, are allowed to contain further structs.

Of P4's derived types, the struct is the most general. An enum is similar, but cannot contain different types. A tuple is comparable as well, but does not contain names for its sub-fields. Therefore, we consider the struct to be the most general. By showing that we can formalise P4's struct, we show that it is possible to also formalise headers, enums, and tuples. We also formalise headers and header stacks in subsection 3.2.1.

Expressions

These types can be used in *expressions*. For example, two base unsigned integers can be summed together, or a named member of a struct can be accessed. We define these expressions recursively. The *base case* of an expression is a base type itself and more complex expressions can be formed by combining expressions recursively (listing 18).

```
datatype expression = BASE baseType
    | NamedVar vname (* Named variable access *)
    | TernExpr expression expression expression (* expr1 ? expr2 : expr 3 *)
    ...
    | ExprMem expression vname (* Member access: var . name *)
    | BIN_ADD expression expression (* binary addition: expr + expr *)
    ...
```

Listing 18: P4's expressions can be captured recursively in Isabelle as shown here. The base case is P4's base types, and further expressions are built recursively from this base case.

```
datatype val = BOOL bool
    | UINT nat (* Unsigned integer *)
    | SINT int (* Signed integer *)
    ...
    | STRUCT "(identifier * vname * val option) list" (* Heterogeneous *)

type_synonym vname = string
type_synonym state = "vname ⇒ val"
```

Listing 19: When a P4 expression is evaluated it produces a result value. We capture this value using a separate `val` data type. The program state of a P4 program is captured as a mapping from variable names to these values.

Note that we keep these expressions general on purpose. Not all expressions are supported for all base types: one Boolean value cannot be summed with another. However, our data types do allow for these cases to be captured. Instead of filtering such invalid expressions out by constricting our data types, we filter such cases in their evaluation. This keeps the formalisation significantly clearer.

Result Values

When a P4 expression is evaluated it produces a result *value*. We capture the output, or result, of such evaluations using a `val` data type. This data type has a one-to-one correspondence with the base types. We use a separate `val` data type, because it allows us to distinguish between the raw P4 program as input and the result that program produces as output (listing 19).

Because the program state of a P4 program is plain we can capture the state as a direct mapping from variable names to these output values (listing 19).

```
datatype statOrDecl = SKIP
    ...
    | BlockStmt "statOrDecl list" (* List of statements *)
    | ConditionalStmt expression statOrDecl statOrDecl (* Conditional *)
    | AssignmentStmt lvalue expression (* Assign a value to an lvalue *)
```

Listing 20: P4 contains statements that capture the core logic of the P4 program. We capture blocks of sequential statements leveraging Isabelle/HOL's list constructs. Other statements are recursively defined.

Statements

Expressions are also used in P4 *statements*. These statements form the core logic of P4's sequential code. P4 features a minimal set of statements. Hence they can be directly modelled in Isabelle/HOL. Most notable are the conditional statement and the assignment statement (listing 20). We use the empty statement `SKIP` as the base case: This is the statement that cannot be executed any further.

We model sequential execution through a list of statements. Because our definitions are recursive, a block statement can contain further nested block statements. Each block can be envisioned to be encapsulated with curly braces and thus such a list of statements also intuitively captures levels of scoping.

Conditional statements contain an expression that should contain a Boolean value and two sub-statements. The intuitive meaning of the conditional statement is straightforward: if the expression evaluates to `True`, the first statement is executed, otherwise the second statement. This corresponds to the standard notion of `if (b) do_true else do_false`. Note that both sub-statements are mandatory. Modelling an empty `else` branch requires filling the second sub-statement with the empty statement `SKIP`.

We capture assignment statements as simply containing an `lvalue` and an expression. The expression contains the value that should be stored in the `lvalue`.

3.1.2 Concrete Value Mapping

We deliberately kept expression data types general; we restrict our semantics to only accept legal operations in the evaluation function. The official P4 specification [16] states exactly which operations are supported for which underlying types. For example: summing two unsigned integers is valid, but summing an unsigned and a signed integer is invalid.

P4 operates under the "forbid, rather than surprise" principle. This means that when an operation is undefined, the operation is forbidden, rather than yielding an undefined result.

We use an inductive definition to define only those operations that the official P4 specification [16] supports. By using an inductive predicate, we mirror this "forbid, rather than surprise" principle. Illegal operations are not defined and therefore will not produce a valid derivation. We show some of the inductive rules we define to capture evaluation in listing 21. For each operation defined in the P4 specification [16] we define a rule that captures its meaning. For example: a base type maps directly to its value. Similarly, a named variable access is taken directly from the state `s`. More complex operations have more complex rules, some with multiple variations supporting the same operation for different underlying types.

Accessing a struct member is more complex, and its evaluation has prerequisites. Our syntax for an expression access contains an expression and a name. To access a struct, we recursively evaluate the expression argument. The operation for accessing a struct member is only defined if this recursive evaluation yields a `STRUCT` type. Then, we retrieve the element from the struct with the requested name using a helper function. Because the name could not exist in the struct, the helper function `getStructMem` can return an *optional value*: `None` or `Some v`. We only define the operation in those cases where some value `v` is found. In that case, the evaluation of the struct access is defined as this value `v`.

Ternary expressions evaluate either to their second or third sub-expressions, based on the value of the first sub-expression. We again recursively evaluate the first sub-expression and define this operation if and only if the first sub-expression yields a Boolean value. Then, we split this operation into two rules: one for when the Boolean value equals `True`, and another for when it equals `False`. Based on this value we recursively evaluate either the second or third sub-expressions respectively and use that as the ternary expression's result.

We also show how we restrict to correctly typed operations in listing 21, specifically one case of binary addition. The syntax of binary addition contains two sub-expressions, which we again recursively evaluate. If and only if both sub-expressions evaluate to unsigned integer base types `n1` and `n2` respectively, do we define the result to be the sum of `n1` and `n2`. The addition of two unsigned integers is thus defined in one *derivation rule*. We similarly define rules for binary addition of P4's other numerical base types: signed, variable-width, and infinite-precision integers [16].

```
inductive eval :: "expression ⇒ state ⇒ val ⇒ bool" where
      RBASE: "eval (BASE b) s (baseToVal b)"
    | NAMEDVAR: "eval (NamedVar varName) s (s varName)"
    ...
    | STRUCTMEM: "eval expr s (STRUCT entries) ⟹
                  getStructMem (STRUCT entries) varName = (Some v) ⟹
                  eval (ExprMem expr varName) s v"
    ...
    | TERNTRUE:  "eval e1 s (BOOL b) ⟹ b = True ⟹ eval e2 s v ⟹
                  eval (TernExpr e1 e2 e3) s v"
    | TERNFALSE: "eval e1 s (BOOL b) ⟹ b = False ⟹ eval e3 s v ⟹
                  eval (TernExpr e1 e2 e3) s v"
    ...
    | BADDU: "eval e1 s (UINT n1) ⟹
              eval e2 s (UINT n2) ⟹
              eval (BIN_ADD e1 e2) s (UINT (n1 + n2))"
    ...
```

Listing 21: We define concrete value evaluation of expressions using an inductive predicate `eval`. We define a rule per defined operation, such as evaluation of ternary expressions or binary addition. The base case is a direct base type, or a named variable access from the program state `s`.

Determinism

With this definition we can prove properties about P4. For example, we verify our implementation is deterministic: if the predicate `eval` is called twice with the same arguments, the result must be identical. We formalise this in lemma 3.1.1. This proves that our implementation is deterministic, which also means it is free from typos leading to undefined variables. For example, if we accidentally typed `n1 + n3` as the result of binary addition, instead of `n1 + n2`, `n3` would likely be undefined. Therefore, the result of evaluating such an expression would be undefined, and thus be non-deterministic. Proving our implementation to be deterministic rules out this class of mistakes. We prove determinism of our inductive predicate `eval` by induction over its rules. Apart from some effort required to prove determinism of struct accesses, the remaining proof is automatic.

Lemma 3.1.1 *eval e s v ⟹ eval e s v′ ⟹ v′ = v*

```
inductive small_step :: "(statOrDecl * state * nat) ⇒
                          (statOrDecl * state * nat) ⇒ bool"
 (infix "⇝" 55) where
    Exit: "(ExitStat, s, n) ⇝ (SKIP, s, n-1)"
    ...
```

Listing 22: The fingerprint of our small-step inductive predicate is a transition from one triple of a statement, program state, and progress counter, to the next. We use the ⇝ symbol to denote this transition. This example shows how an exit statement `ExitStat` is executed in one step and progresses to the empty statement `SKIP`.

3.1.3 P4 Action Semantics

Evaluating an expression yields a value. Evaluating an entire P4 action yields a resulting program state. This *final state* depends both on the statements in the action code, as well as the initial state. P4 programs do not have other side-effects, such as output through print statements; the only "output" of a P4 action is this final program state. However, we are also interested in the intermediate states that exist during the execution and want to be able to formally reason about these intermediate states. Hence, we implement an *executable small-step semantics* of P4 actions.

We define our small-step semantics as a transition relation between triples: a triple consisting of a statement, program state, and progress counter. The input triple captures the next step to be executed, the current program state, and the current progress counter. The transition models a single step of the input statement and updates the program state and progress counters appropriately. We use the notation `csn` ⇝ `csn'` to denote a single small-step transition between triples `csn` and `csn'`. For example, in listing 22, we show the fingerprint of our inductive predicate `small_step`. We also show how the exit statement `ExitStat` is transformed in a single step from the exit statement to the empty statement `SKIP` in listing 22. We use the empty statement `SKIP` as the "dead statement" with no further progression possible. At the end of a correct execution, only the empty statement should thus remain.

Note that we use the progress counter as a sort of primitive clock function. On each step, we decrease the progress counter by one. As long as there are statements to be executed, there is progress to be made, and the counter will thus decrease. The choice to decrease the counter rather than increase is an arbitrary one for this purpose. We can use this counter to analyse progress and termination properties. For example, we can prove that while there still is progress to be made, the counter will necessarily decrease.

```
inductive small_step :: "(statOrDecl * state * nat) ⇒
                          (statOrDecl * state * nat) ⇒ bool"
 (infix "⇝" 55) where
    ...
    | Assign: "eval expr s v ⟹
                (AssignmentStmt (NameLVal vName) expr, s, n) ⇝
                (SKIP, s (vName := v), n-1)"
    ...
```

Listing 23: We define concrete value evaluation of expressions using an inductive predicate `eval`. We define a rule per defined operation, such as evaluation of ternary expressions or binary addition. The base case is a direct base type, or named variable access from the program state `s`.

Conversely, if the counter does not decrease, no further progress is possible. In this way, we use such counters as a primitive termination proofs for our small-step semantics. We capture these properties more formally in lemmas 3.1.2 and 3.1.3 respectively.

Lemma 3.1.2 $(c,\ s,\ n) \rightsquigarrow (c',\ s',\ n') \Longrightarrow n' \leq n$

Lemma 3.1.3 $n' > n \Longrightarrow \neg((c,\ s,\ n) \rightsquigarrow (c',\ s',\ n'))$

Assignments and Declarations

Assignment statements change the program state upon execution. The syntax of the assignment statement contains an expression and a variable name to which the resultant value of the expression should be assigned. The correctness of the assignment statement thus also depends on the correctness of the expression contained in the assignment statement. If and only if the evaluation of the expression is both valid and exists, do we define the inductive rule for assignment. In such a correct case, the assignment statement is executed and replaced with the empty statement. The program state is updated to now map the variable name to the new value (listing 23).

Conditional Statements

Similar to how we split the evaluation of a ternary expression into separate rules for when the first sub-expression evaluated to `True` or `False`, we also split the small-step semantics for the execution of a conditional statement. The syntax of a conditional statement contains an expression and two sub-statements. We evaluate the expression and the result is defined if and only if it evaluates to a Boolean value. P4 does not support implicit casts to a Boolean

```
inductive small_step :: "(statOrDecl * state * nat) ⇒
                          (statOrDecl * state * nat) ⇒ bool"
 (infix "⇝" 55) where
    ...
    | CondTrue:  "eval e s (BOOL True) ⟹
                  ((ConditionalStmt e stmt1 stmt2), s, n) ⇝
                  (stmt1, s, n-1)"
    | CondFalse: "eval e s (BOOL False) ⟹
                  ((ConditionalStmt e stmt1 stmt2), s, n) ⇝
                  (stmt2, s, n-1)"
    ...
```

Listing 24: The execution of a conditional statement requires two separate rules. One for the case where the expression of the conditional statement evaluates to `True`, and another for when the expression evaluates to `False`. The conditional statement is replaced with either the first or second sub-statement depending on this evaluation.

value. The syntax `if (n) ...`, where `n` is a numeric value, is simply not valid in P4. Therefore, we define the evaluation of the conditional expression if and only if the sub-expression evaluates to a Boolean value. Depending on whether the sub-expression evaluates to `True` or `False`, the conditional statement evaluates to either the first or second sub-statement respectively.

Block Statements

We capture P4 block statements by leveraging Isabelle/HOL's list constructs. To execute a list of statements we split the execution into three cases. The base case is the execution of the empty list. In this case, the block statement containing only an empty list is directly replaced with the empty statement. The next case is if the head of the list of the block statement is the empty statement. This implies that the head of the list was either the empty statement or has already been executed. In this case, we can remove the head of the list. This leaves a block statement containing only the tail of the list of statements. The third case captures a block statement containing a list whose head is not the empty statement. In this case, we execute the head of the statement list as a single step. We then replace the head of the list with the statement that this single-step execution yielded, and repeat the exercise (listing 25).

```
inductive small_step :: "(statOrDecl * state * nat) ⇒
                         (statOrDecl * state * nat) ⇒ bool"
 (infix "⇝" 55) where
    ...
    | EmptyBlock: "(BlockStmt [], s, n) ⇝ (SKIP, s, n-1)"
    | EmptyFirst: "(BlockStmt (SKIP # rest), s, n) ⇝
                   (BlockStmt rest, s, n-1)"
    | FullBlock:  "(stmt, s, n) ⇝ (stmt', s', n') ⟹
                   n' ≤ n ⟹
                   (BlockStmt (stmt # rest), s, n) ⇝
                   (BlockStmt (stmt' # rest), s', n')"
```

Listing 25: We capture block statements using a list of further statements. Executing such a block statement yields three cases. An empty block statement is directly replaced with the empty statement. A block statement containing a list with the empty statement as the head is replaced with the same block statement with the head stripped. If the block statement contains a list with a non-empty head, the head statement is executed using a single step. The head of the list is replaced with the resulting statement.

Execution Example

The small-step semantics we defined is an executable one. To execute a P4 action we first need to convert it into our Isabelle/HOL syntax. For example, the P4 code `if (x == 3) y = 5` would be captured by the Isabelle/HOL code:

```
ConditionalStmt (BIN_EQU (NamedVar ”x”) (BaseType (UINT 3)))
(AssignmentStmt ”y” (BaseType (UINT 5))) (SKIP)
```

If `x` and `y` are defined as numerical values, this action is valid; derivation rules exist for every step of the derivation. We can execute this statement as a single step using our small-step semantics. Given the state `<”x” := (UINT 3)>`, the conditional evaluates to `True` and the conditional statement would progress into the assignment statement. More specifically, we would have the following transition:

```
(ConditionalStmt (BIN_EQU (NamedVar ”x”) (BaseType (UINT 3)))
(AssignmentStmt ”y” (BaseType (UINT 5))) (SKIP), s, n)
⇝ ((AssignmentStmt ”y” (BaseType (UINT 5))), s, n - 1)
```

Reflexive Transitive Closure

Our small-step analysis allows us to analyse single steps of the execution of a P4 action. However, we also want to be able to analyse properties of executions of any number of these individual steps, this includes the full program execution or no executions at all.

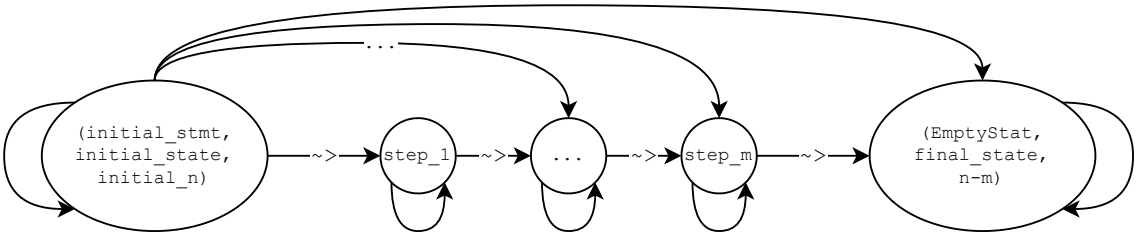

Figure 3.1: We can use our single-step small-step semantic relation to analyse the space any number of these individual steps in a program by observing the *reflexive transitive closure*. This relation yields the set of all reachable states in any number of steps (including zero steps) from the initial state.

To analyse properties like reachability properties we thus want to analyse the full set of reachable states from a given starting state. For this, we use the *reflexive transitive closure* of our small-step transition relation. This will yield the set of all states reachable in any number of steps, including zero steps, from a particular initial state. In our case the initial state is the triple containing the initial statement, program state, and progress counter.

The reflexive transitive closure of a relation R is defined as the smallest relation S such that S is a superset of R, and that S is both reflexive and transitive. Because S is reflexive, there must exists a transition from state x to itself for all states $x \in S$. Because S is also transitive, we know that if there exist states x, y, and z, and a path from x to y and from y to z, there must also exist a direct relation from x to z (figure 3.1).

To implement our reflexive transitive closure relation we take the general polymorphic reflexive transitive closure relation `star` from *Concrete Semantics* [18]. This predicate defines the reflexive transitive closure on relation `r`. It is defined through two rules. The first rule captures reflexivity: there always exists a relation from every state to itself. The second rule captures transitivity: if there exist three states x, y, and z, and there exists a path from x to y, and from y to z, there also exists a relation directly from x to z (listing 26). Note that this definition uses itself recursively to capture this path as any number of steps.

We apply this notion using our small-step semantics as the relation r. The reflexive transitive closure of our small-step semantics means that we can get the set of all reachable states in any number, zero or higher, small-step transitions. From a starting triple `csn` consisting of a starting statement, program state, and progress counter, the relation

```
inductive star :: "('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"
  for r :: "('a ⇒ 'a ⇒ bool)" where
refl:   "star r x x" |
step:   "r x y ⟹ star r y z ⟹ star r x z"

abbreviation small_steps :: "(statOrDecl * state * nat) ⇒
                              (statOrDecl * state * nat) ⇒ bool"
    (infix "⇝*" 55)
  where "x ⇝* y ⟹ star small_step x y"
```

Listing 26: We can use our single-step small-step semantic relation to analyse the full execution of a program by observing the *reflexive transitive closure*. This relation yields the set of all reachable states in any number of steps (including zero steps) from the initial state.

`star` yields the set of all states reachable from `csn` in any number of steps. We use the abbreviation `csn ⇝* csn'` to mean that `csn'` is reachable from `csn` in any number of steps (listing 26).

Statement Reachability Analysis

The reflexive transitive closure allows us to analyse the complete set of reachable states from any initial state. We also define a function to yield whether or not a function is reachable in any particular number of steps: *n-step reachability*. This allows us to not only analyse if a state is reachable at all, but also reason about in how many steps that state is reachable. We denote this relation as `csn ⇝(n) csn'`, where this means that triple `csn'` is reachable from triple `csn` in exactly `n` steps. We define this function by induction over `n`. This yields two cases: The function is called with `n` equal to zero, or the function gets called with a successor of `n`. If called with an argument of zero, the only possible relation should be that `csn` equals `csn'`. If the function is called with a successor of `n` as argument, the relation between `csn` and `csn''` exists if and only if a one-step relation exists between `csn` and `csn'` and an n-step relation exists between `csn'` and `csn''` (listing 27).

3.1.4 P4 Action Typing Environment

We only define operations on *well-typed* expressions: expression whose types are all correct. However, this does not allow us to reason about the well-typedness of statements or the result of expressions, nor can we formally reason about properties that well-typed actions exhibit. We include an extensive strict typing system in Nano P4. Similar to defining

```
fun small_steps_n :: "(statOrDecl * state * nat) ⇒ nat ⇒
                      (statOrDecl * state * nat) ⇒ bool"
    ("_ ⇝'(_') _" [55,0,55] 55)
  where
      "(cs ⇝(0) cs') = (cs' = cs)"
    | "(cs ⇝(Suc n) cs'') = (∃cs'. cs ⇝ cs' ∧ cs' ⇝(n) cs'')"
```

Listing 27: We define a function `csn ⇝(n) csn'` to define the reachability of `csn'` from `csn` in exactly `n` steps. We define this by induction over `n`, yielding two cases. Either the function is called with the argument zero, or the function is called with a successor of `n`. In the first case the only reachable state from `csn` is `csn` itself. Otherwise, the function is only defined if there exists a multi-step relation between `csn` and `csn''`.

```
datatype ty = BOOLty | UINTty | SINTty | ...
    | STRUCTty "(vname * ty option) list"

type_synonym typeEnv = "vname ⇒ ty"
```

Listing 28: We define a one-to-one mapping between base types and our types. Derived types like structs use Isabelle/HOL's list constructs to capture the types of their member fields. We define helper functions to convert between base types and these derived types.

the semantics of P4 actions, we start by defining the possible types. Instead of defining a program state, we define a *typing state* or *typing environment*. This environment contains a mapping of all variables to their types. Then we define an evaluation function that yields the resulting type of the execution of an expression, and finally we define typing of program state and statements. We show that we can use this type system to reason about properties like progression and preservation.

P4 Action Types

We start by defining which types make up our type system. In our case we define a one-to-one correspondence between the base types we define for the P4 actions and their resultant types. Note that these types consist only of constructors and have no values associated with them. For derived types like a struct, we capture the types of its member fields with another list. We define helper functions to convert a struct and its member fields to a corresponding struct type and its type member fields. We define our typing environment as a direct mapping from variable names to their type, and use Γ to denote this typing environment (listing 28).

```
inductive exprTyping :: "typeEnv ⇒ expression ⇒ ty ⇒ bool"
    ("(1_/ ⊢/ (_ ::/ _))" [50,0,50] 50) where
      BASE_ty:     "Γ ⊢ (BASE b) :: getBaseType (b)"
    | EXPRMEM_ty: "Γ ⊢ e1 :: (STRUCTty entries) ⟹
                    getTyMemTy (STRUCTty entries) vName = (Some τ) ⟹
                    Γ ⊢ ExprMem e1 vName :: τ"
    | VAR_ty:      "Γ ⊢ NamedVar vName :: Γ vName"
    ...
    | BEQUU_ty:    "Γ ⊢ e1 :: UINTty ⟹
                    Γ ⊢ e2 :: UINTty ⟹
                    Γ ⊢ BIN_EQU e1 e2 :: BOOLty"
    ...
```

Listing 29: The type of an expression depends on its contents and the typing environment Γ. If an expression is well-typed in the context of Γ, we denote this as $\Gamma \vdash e :: \tau$. Note that the sub-expressions need not necessarily be of the same type as the resulting type of the expression.

Concrete Type Evaluation

To find the resulting type of the evaluation of an expression, we define a *typed evaluation* predicate. With this predicate we can reason about whether or not an expression is well-typed in the context of a given typing environment Γ. We denote a well-typed expression e as having type τ in the context of typing environment Γ as follows: $\Gamma \vdash e :: \tau$.

To define this predicate we create a rule to capture every well-typed operation. For example, an expression containing a singular constant value, directly renders that constant value's type. Similarly, the type of a named variable is taken directly from the typing environment Γ. When accessing a member field of a struct, this yields the type of that member field. For other operations, such as the equality operator, the resulting type need not necessarily be the same as the input types; comparing two unsigned integers yields a Boolean value. In this way we define a rule for every possible valid and well-typed operation in P4 actions (listing 29).

Statement Typing

We apply the same concept to P4 statements. An assignment statement is considered well-typed if its contents are well-typed. For example, if the type of a variable matches the type of the expression whose value is assigned to that variable (and that that variable name was declared with that type prior to the assignment), then that assignment statement is well-typed. Other statements, like block statements, are well-typed if all of their sub-

```
inductive stmtTyping :: "typeEnv ⇒ statOrDecl ⇒ bool"
    (infix "⊨" 50) where
      Empty_ty: "Γ ⊨ SKIP"
    ...
    | Block_Empty_ty: "Γ ⊨ (BlockStmt [])"
    | BlockFull_ty: "Γ ⊨ stmt ⟹
                     Γ ⊨ (BlockStmt rest) ⟹
                     Γ ⊨ (BlockStmt (stmt # rest))"
    | Assign_ty: "Γ ⊢ e :: Γ (vName) ⟹
                  Γ ⊨ (AssignmentStmt (NameLVal vName) e)"
    ...
```

Listing 30: Statements are well-typed if their contents are well-typed. For example, the assignment statement is well-typed if and only if the expression is well-typed and the type of the expression matches the a priori declared type of that variable. A block statement is well-typed if all of its sub-statements are well-typed.

```
definition stateTyping :: "typeEnv ⇒ state ⇒ bool" (infix "⊫" 50)
  where "Γ ⊫ s⟷(∀x. getValType (s x) = Γ x)"
```

Listing 31: The typing environment is considered well-typed if all of the variables in the typing environment match with the corresponding types in the program state.

statements are well-typed. The empty statement depends on no further statements, so it is by definition well-typed. We denote a statement being well-typed in the context of typing environment Γ as $\Gamma \models stmt$ (listing 30).

State Typing

The state is considered well-typed if all of the variables captured in the typing environment match the type of that same variable in the program state. That is, if a variable `"x"` has type `UINTty` in the typing environment Γ, the variable `"x"` must contain a value of type `UINT n` in the program state `s`. We denote the state being well-typed under typing environment Γ as $\Gamma \models s$ (listing 31).

Type Derivation

If a statement and its derivation are well-typed, a corresponding *type derivation tree* can also be constructed. Similar to a regular derivation tree, such a type derivation tree provides insight into the behaviour of the typing environment. Note that such a tree can only be constructed if the statement is well-typed. For example, the P4 statement

$$\dfrac{\dfrac{\texttt{GetTy (U 5) = Uty}}{\Gamma \vdash \texttt{BTy (U 5) : Uty}}}{\Gamma \models \texttt{(VDcl "x" e1)}} \qquad \dfrac{\dfrac{\Gamma\texttt{"y" = Uty}}{\Gamma \vdash \texttt{Var "y" : Uty}} \qquad \dfrac{\texttt{(getTy (U 3) = Uty}}{\Gamma \vdash \texttt{BTy (U 3) : Uty}}}{\Gamma \models \texttt{(VDcl "y" e2)}}$$

$$\Gamma \models \texttt{Blck [(VDcl "x" e1), (VDcl "y" e2)]}$$

Figure 3.2: The P4 statement `bit<32> x = 5; bit<32> y = x + 3;` is captured in Isabelle as follows: `BlockStmt [(VariableDecl "x" (Some (BaseType (UINT 5)))), (VariableDecl "y" (Some (BIN_ADD (NamedVar "x") (BaseType (UINT 3)))))]`. We show the corresponding type derivation tree in this figure. Note that we use the simplifications `expr1 = Some (BaseType (UINT 5))`, `expr2 = Some (BIN_ADD (NamedVar "x") (BaseType (UINT 3)))`, and shorten all names for brevity.

`bit<32> x = 5; bit<32> y = x + 3;` is captured in Isabelle as follows:

```
BlockStmt [(VariableDecl "x" (Some (BaseType (UINT 5)))),
(VariableDecl "y" (Some (BIN_ADD (NamedVar "x")
(BaseType (UINT 3)))))]
```

This statement is well-typed in the context of typing environment Γ, and can be captured by the derivation tree shown in figure 3.2. We say that this statement *follows from the typing environment* Γ, which we denote as:

$\Gamma \vdash$
```
BlockStmt [(VariableDecl "x" (Some (BaseType (UINT 5)))),
(VariableDecl "y" (Some (BIN_ADD (NamedVar "x")
(BaseType (UINT 3)))))].
```

Type Soundness

We can use the Nano P4 typing system to formally reason about the P4 typing system. For example, we can prove type preservation of expressions: If an expression and state are well-typed, the result is necessarily also well-typed. Moreover, we can prove properties such as expression progression: If an expression and state are well-typed, an evaluation for that expression must exist. We can extend this proof to even reason about complete progression: If a statement and state are well-typed and that statement is not the final state, there must be further executions that can be made; the execution has not yet terminated. We capture these three properties more formally in lemmas 3.1.4, 3.1.5, and 3.1.6 respectively.

Lemma 3.1.4

$$\Gamma \vdash \mathit{expr} :: \ t \Longrightarrow \mathit{eval\ expr\ s\ v} \Longrightarrow \Gamma \mid\mid\models s \Longrightarrow \mathit{getValType\ v} = t$$

Lemma 3.1.5

$$\Gamma \vdash e :: t \Longrightarrow \Gamma \mid\mid\models s \Longrightarrow \exists v.\ \ eval\ e\ s\ v$$

Lemma 3.1.6

$$\Gamma \models stmt \Longrightarrow \Gamma \mid\mid\models s \Longrightarrow stmt \neq SKIP \Longrightarrow \exists csn'.\ \ (c,\ s,\ n) \rightsquigarrow csn'$$

Based on these proofs we can prove type-soundness of our semantics. That is to say, as long as the action is type sound, there is progress to be made. For this, we combine our proofs of the reflexive transitive closure with our typing environment. We prove that — given that there exists a relation between triple `csn` and `csn'` in any number of steps including zero; that statement `c` and state `s` are well-typed; and that the statement `c'` is not the empty statement — there must exist another triple `csn''` that state `csn'` can progress to. With this, we show that our semantics must terminate, given that the action is type-sound. We formalise this in theorem 3.1.7.

Theorem 3.1.7

$$\begin{aligned}(c,\ s,\ n) \rightsquigarrow * (c',\ s',\ n') \Longrightarrow\\ \Gamma \models c \Longrightarrow\\ \Gamma \mid\mid\models s \Longrightarrow\\ c' \neq SKIP \Longrightarrow\\ \exists csn''.\ \ (c',\ s',\ n') \rightsquigarrow csn''\end{aligned}$$

3.2 Verifying Security Properties

In section 3.1 we presented Nano P4, its small-step semantics and its strict typing system. Nano P4 only accepts semantically correct programs; no derivations exist for illegal P4 programs, and Nano P4 rejects those already. Ensuring that a specific property is never violated in a P4 program using Nano P4 requires changing Nano P4 such that programs that violate the property become illegal in Nano P4. In this section we show how we use Nano P4 to ensure security properties, and discuss how additional properties can be added to Nano P4. In subsection 3.2.1 we show how we ensure a P4 program to be free from

out-of-bound header stack accesses, and in subsection 3.2.2 we show how we ensure that no uninitialised data can be used in Nano P4. In subsection 3.2.3 we show how Nano P4 can conceptually be extending by proposing an extension that ensures domain-specific parameter-direction properties of P4.

3.2.1 Out-of-Bound Header Stacks

P4 does not have a notion of general lists. However, P4 does have a notion of header stacks: essentially a list of headers which can be indexed. These indices can be greater than the size of the stack, leading to out-of-bound accesses. Similarly, a header stack can be popped more times than it contains entries, similarly leading to out-of-bound accesses. This leads to undefined behaviour which can be exploited [9]. We show that we can easily extend the semantics of Nano P4 such that indices must fall in range of the size of the stack. This restricts the semantics of Nano P4 to only accept programs that are free from out-of-bound accesses.

Modelling P4 Headers

Similar in principle to a struct, a P4 header is essentially a list of type and field name pairs. We use Isabelle/HOL's list constructs again to capture these headers as a list of string pairs. Note that for the purposes of restricting our semantics to disallow out-of-bound accesses, we do not include values in these headers and focus merely on the declaration of such a header. The size of the header stack is necessarily known at compile-time, as per the official P4 specification [16]. We show how we capture the syntax of a P4 header and header stack in listing 32. To support this we add a new operation, index-based access `StckIdx`, containing the name of a variable and the natural number capturing the index. Note that P4 allows signed integers to be used and that indexing a negative index yields an undefined value. We are trying to restrict undefined values. Therefore, we disallow the usage of negative numbers by restricting the syntax to only accept natural numbers. We could also have disallowed the use of negative numbers in the evaluation function (as in subsection 3.1.2), similarly to how we restrict more general syntax in other cases.

```
datatype baseType = BBOOL bool
    ...
    | BHEADER "(identifier * identifier) list"
    | BHSTACK "((identifier * identifier) list) list"

datatype expression = BASE baseType
    ...
    | StckIdx expression nat
    ...
```

Listing 32: We capture the syntax of P4 headers and header stacks by leveraging Isabelle/HOL's list constructs. We also extend Nano P4's expressions with an index-access containing a natural number and the name of the header stack to be accessed.

```
inductive eval :: "expression ⇒ state ⇒ val ⇒ bool" where
    ...
    (* Header access iff n < len stck *)
    | STCKIDX: "eval e1 s (HSTACK stck) ⟹
                n < length stck ⟹
                eval (StckIdx e1 n) s (HEADER (stck ! n))"
```

Listing 33: Accessing an index of a header stack requires that the sub-expression of the `StckIdx` syntax evaluates to a header stack with list `stck`. Moreover, the index `n` should be smaller than the length of the list `stck` to be a valid access.

Evaluating Indexing

Next we define how to evaluate accessing an index of a header stack. The prerequisites we require are that the expression contained in the `StckIdx` syntax, evaluates to a header stack with entries `stck`. The evaluation is defined as long as the index `n` is smaller than the length of the list `stck`. If these prerequisites are met, the evaluation simply yields the header at index `n` from the list `stck` (listing 33).

Typing Environment

Restricting out-of-bound header stack accesses does not affect P4 statements and thus no alterations to the small-step semantics of Nano P4 are necessary. However, adding this restriction does require some additions to Nano P4's typing system. We define the header type to contain a single additional integer value with its type. This integer is set to the number of entries in the header. Using this number we know the upper limit of an access to the header. We then define a header-index expression as well-typed if and only if the

```
datatype UINTty | ... | HSTACKty nat

fun getValType :: "val ⇒ ty" where
      "getValType (UINT n) = UINTty"
    ...
    | "getValType (HSTACK stck) = HSTACKty (length stck)"

inductive exprTyping :: "typeEnv ⇒ expression ⇒ ty ⇒ bool" where
    ...
    (* Header access iff n < len stck *)
    | STCKIDX_ty: "Γ ⊢ e1 :: HSTACKty stck ⟹
        n < stck ⟹
        Γ ⊢ StckIdx e1 n :: HEADERty"
```

Listing 34: The header stack indexed access is only well-typed if and only if the index lies within the bounds of the header stack. To support this check the header stack type contains an additional integer value that captures the length of the header stack. The header stack indexed access is only considered well-typed if the index lies beneath this upper bound.

index is lower than this natural number (listing 34). This also means a header-index access is well-typed if and only if there also exists an evaluation for it. Including the out-of-bound restriction does not change or affect our type system based proofs.

Using the Semantics

With this extension, the semantics of Nano P4 are restricted such that correct derivation rules only exist for P4 programs free from out-of-bound accesses of header stacks. This approach can also be taken to ensure other security properties are met. We show that such extensions are not necessarily complicated to add and need not fundamentally change the structure of our approach.

3.2.2 Uninitialised Data

Another potentially exploitable security issue in P4 programs is the usage of uninitialised objects [9], such as uninitialised variables or header fields. Our semantics already disallow such cases by design. In section 3.1 we note that no derivation rules exist for incorrect P4 code. The same applies to the program state. If a P4 action relies on a program state to contain the value of a specific variable, but it has not been declared, no derivation exists for that variable. Hence, the program is rejected by Nano P4.

```
type_synonym state = "vname ⇒ val option"

inductive eval :: "expression ⇒ state ⇒ val ⇒ bool" where
    ...
    | NAMEDVAR: "s varName = Some v ⟹ eval (NamedVar varName) s v"
    ...
```

Listing 35: The state could be replaced to be a mapping from variable names to optional values to capture more information about declarations, initialisation, and assignments. The evaluation of a variable would only be valid if the variable contains a value and not a `None`.

For example, the P4 code `x = y;` relies on the value of `y` being known. Given the state `<"y" := (UINT 3)>`, a derivation is possible. However, given the empty state `<>`, no derivation exists. In this way Nano P4 ensures security properties like excluding the usage of uninitialised variables.

Declaration, Assignment, or Initialisation

Note that in our semantics we make no difference between a declaration, assignment, or initialisation; we declare and initialise a variable when it is assigned a value. Our state is a mapping from variable names to their values. A variable cannot be declared without containing a value in our case, which is why we do not make this distinction. This can easily be changed by replacing our state with a mapping from variable names to *optional values*. The evaluation of a named variable then only exists if it contains a value (listing 35).

We then change the declaration, initialisation, and assignment statements to be distinct. Assignment would only be defined if the state already has the variable declared. A declaration would merely declare the variable with no value assigned, while initialisation would immediately assign to it a value (listing 36). Due to time constraints we do not incorporate this distinction into Nano P4.

3.2.3 Extending Nano P4 with Directions

This shows how we can use Nano P4 to guarantee specific properties are met. In general, given a specific criterion, we can extend or modify the semantics of Nano P4 such that it only accepts those programs that meet the criterion. This requires defining the property formally, extending Nano P4 such that the semantics have the necessary components to reason about that property, and then ensuring that the property is met. We show how this process can generally be performed using P4's directional parameters.

```
inductive small_step :: "(statOrDecl * state * nat) ⇒
                         (statOrDecl * state * nat) ⇒ bool"
    (infix "⇝" 55) where
    ...
    | AssignNone: "s vName = None ⟹ eval e s v ⟹
                   (AssignmentStmt (NameLVal vName) e, s, n) ⇝
                   (SKIP, s (vName := Some v), n-1)"
    | AssignSome: "s vName = Some v ⟹ eval e s v' ⟹
                    (AssignmentStmt (NameLVal vName) e, s, n) ⇝
                    (SKIP, s (vName := Some v'), n-1)"
    | VarDecl:    "(VariableDecl (NameLVal vName) None, s, n) ⇝
                    (SKIP, s (vName := None), n-1)"
    | VarInit:    "eval e s v ⟹
                    (VariableDecl (NameLVal vName) (Some e), s, n) ⇝
                    (SKIP, s, (vName := v), n-1)"
```

Listing 36: Capturing a difference between initialisation, declaration, and assignment could be done by splitting these statement rules. When a variable is declared without a value it is inserted in the state as a mapping to `None`. Otherwise, it is assigned some value `Some v`. Assignment can only occur if the variable is already declared and initialised in the state.

Directional Parameters

P4 knows different *directions* for function parameters: *in*, *in/out*, and *out*. *In* parameters are inputs for the function and are defined before calling the function. They are not output, so are ignored and discarded after the function returns. *Out* parameters are unset when the function is called and must be set in the function; they are output. A function that does not set such out-direction parameters is flawed. Reading an out-direction parameter before it is set, yields undefined behaviours and is also a fault. *In/out* parameters are a combination of these: They are set before calling the function and are also considered as output of the function.

Reading an undefined out-direction parameter is already disallowed by the semantics of Nano P4. We describe how this is done in this section and in section 3.1. Ensuring that out-direction parameters are set after the execution of a function could be achieved by extending the semantics of Nano P4. Currently, Nano P4 does not include a notion of function calls, so we assume functions are inlined. More formally, this property requires that all variables that carry the *out* direction must be assigned a value when the final state has been reached. We capture the property formally in lemma 3.2.1.

Lemma 3.2.1

$$final\ (c,\ s,\ n) \wedge (\forall v.\quad (v \in m \wedge m\ v\ =\ OUT) \Longrightarrow (\exists v'.\quad v\ =\ Some\ v'))$$

```
datatype direction = IN | OUT | INOUT
type_synonym directionMap = "vname ⇒ direction"
type_synonym state = "vname ⇒ val"

definition "correct_term (c, s, m) ⟷
            final (c, s) ∧ (∀v. (v ∈ m ∧ m v = OUT) ⟹ (∃ v'. v = v'))"
```

Listing 37: Correctness of directional mapping could be defined as in lemma 3.2.1: for every variable in the direction mapping that is specified as direction out, it must at the end of the program execution have a value assigned to it.

To capture the property captured in lemma 3.2.1 we need to extend the semantics of Nano P4 such that it can reason about these directional parameters. For example, we could add a direction data type, with a constructor for each direction. Then, we could define a mapping from variable names to their direction. We could then define correctness of a statement, state, and progress counter triple using Isabelle/HOL's `definition` keyword. This keyword is similar to a predicate, and allows us to define cases that are true if and only if the right hand side of the definition holds. In listing 37 we use this definition to define when a statement, state, and direction mapping triple have correctly terminated. This is the case if and only if the statement and state are the final ones; there is no further progress possible, and all variables in the directional mapping that carry the `OUT` direction have some value assigned in the program state. Fully implementing and proving correctness of this extension lies beyond the scope of this thesis however.

3.2.4 Security Property Verification

The small-step semantics of Nano P4 are not defined for faulty programs: no derivations exist for illegal programs and hence they are rejected by Nano P4. This small-step semantics of Nano P4 can be further extended to restrict or expand the class of programs it accepts by adding, modifying, or removing semantic derivation rules. We show that we can use this to use Nano P4 to guarantee programs never violate security properties. This requires a distinguishable criterion. For example: "a program exhibits no undefined behaviour" is neither specific nor distinguishable. Such properties must be further specified such that the semantics can distinguish between those programs that meet the criterion and those

that do not. For example: "All well-typed programs must terminate" can be expressed as a security property. We show that, given such criteria, these security properties can be verified thoroughly using a system like Nano P4.

Such extensions are particularly useful for P4. The official P4 specification leaves room for vendors to define additional restrictions on top of those presented in the official specification [16]. For example, the official P4 specification gives no hard limit to the maximum width of bit strings. However, hardware is unlikely to support a million-bit wide bit string. Therefore, a vendor can impose restrictions like the maximum allowed width. These limits change the semantics of the vendor-specific implementation of P4 slightly. All of these architecture-specific restrictions are detailed in the architecture description. This description includes all pre-defined global variables. Thus, adding these to the initial state and defining such restrictions a priori would be easy. This allows vendors to use systems like Nano P4 to verify their specific implementation of P4, as well as P4 itself.

3.3 Verification of Program Optimisations

P4 focuses on efficiency in combination with flexibility. P4-enabled devices are aimed at large data centres, core backbone networks, or other high-throughput environments. Therefore, optimisations are a worthwhile endeavour. However, optimisations at the cost of correctness are undesirable, as they could lead to vulnerabilities. We show that we can use Nano P4 to verify program transformations, such as optimisation routines. We present a formalisation of two common compiler optimisation routines: constant folding and constant propagation. We show that we can use our semantics to define such transformations and prove such transformation maintain *semantic equivalence*; that is, the transformations do not change the *meaning* of the program. Note that due to time constraints we could not finish the complete semantic equivalence proofs and therefore limited ourselves to their implementation.

3.3.1 Constant Folding and Propagation

Constant folding computes, if possible, the constant value of an expression at compile-time. The expression is then replaced with the constant value. This removes the need to calculate expressions rendering the same result numerous times. This saves time and thus optimises the program. Similarly, constant propagation takes all constant values in a program and propagates them to other variables wherever possible. In this way, wherever variables are used whose contents are constant, these variables themselves can be replaced by their

```
0 bit<32> x = 16 + 32 + 64;
1 bit<32> y = x * 2;
```

Listing 38: Two P4 statements that can be optimised by using constant folding and propagation. The first expression can be replaced with a constant value 112, while the second expression can be replaced with a constant value of 224.

```
0 type_synonym tab = "vname ⇒ val option"
```

Listing 39: The shadow table is a mapping from variable names to optional values. The table tracks constant values; if a constant is known for a variable it is stored in the shadow table.

constant value, again reducing the number of operations and optimising the program. The only remaining expressions and statements after constant folding and constant propagation are those that depend on values that are not known a priori.

In listing 38 we show two P4 statements. The first statement assigns to `x` the value of `16 + 32 + 64`. Instead of calculating this every time the program is executed, it is calculated once during an optimisation pass. This statement can therefore be replaced with the statement `bit<32> x = 112;`, which would be semantically equivalent. The variable `x` now has a constant value, on which the value of `y` in the second statement depends. Through constant propagation we can also optimise the second statement by replacing it with `bit<32> y = 224;`. These two statements are semantically equivalent to the original statement, but contain four fewer computations than the original. Other optimisation routines exist as well. For example, if `x` is never used after being used to determine the value of `y`, it can be considered *dead* and could be removed from the program entirely. This is referred to as *liveness analysis*. Such more complicated optimisations lie beyond the scope of this thesis however.

3.3.2 Expression Optimisation

Shadow Tables

To optimise expressions we define a function that takes an expression and returns its optimised version. To know which values are constants, this function requires a *shadow table*. The shadow table is similar to our program state; it contains a mapping from variable names to values. However, the shadow table only tracks constant values when they exist. Hence, the mapping is from variable names to optional constant values (listing 39).

```
definition "approx t s ⟷ (∀vName v. t vName = Some v ⟹ s vName = v)"
```

Listing 40: A shadow table `t` approximates program state `s` if and only if all of the variables for which the shadow table has a value have a matching value between the shadow table and the program state.

```
fun efold :: "expression ⇒ tab ⇒ expression" where
      "efold (BASE bVal) _ = (BASE bVal)"
    | "efold (NamedVar vName) t = (case t vName of None ⇒ NamedVar vName
            | Some vVal ⇒ BASE (valToBase vVal))"
    | "efold (TernExpr e1 e2 e3) t = (case efold e1 t of
              BASE (BBOOL True) ⇒ efold e2 t
            | BASE (BBOOL False) ⇒ efold e3 t
            | e1' ⇒ TernExpr e1' e2 e3)"
    ...
    | "efold (BIN_ADD e1 e2) t = (case (efold e1 t, efold e2 t) of
              (BASE (BSINT n1), BASE (BSINT n2)) ⇒ BASE (BSINT (n1 + n2))
            | (BASE (BUINT n1), BASE (BUINT n2)) ⇒ BASE (BUINT (n1 + n2))
            | (BASE (BIINT n1), BASE (BIINT n2)) ⇒ BASE (BIINT (n1 + n2))
            | (e1', e2') ⇒ BIN_ADD e1' e2')"
    ...
```

Listing 41: We perform optimisations of expressions by pattern matching the input expression. The base case that cannot be optimised is the constant value. Variables are looked up in the shadow table and replaced if a constant value is found. We recursively optimise sub-expressions containing further sub-expressions, like binary addition. We replace those expressions and sub-expressions with their optimised versions wherever possible.

This shadow table then provides the source to replace variables with values. If the shadow table were to contain values that have nothing to do with the program state, the resulting optimisation based on this shadow table would be nonsensical. We thus need to define the property of *approximation*. We say a shadow table `t` approximates program state `s` if and only if all of the variables that have a constant value in the shadow table. Those values are necessarily the same in the program state (listing 40).

Replacing Expressions via a Shadow Table

We define the optimisation via a shadow table by pattern matching input expressions. If an input expression is a base value, it cannot be optimised any further. A variable can only be optimised if it has a constant value associated with it in the shadow table, which is looked up and returned if it exists (listing 38).

Some expressions contain recursive expressions within them. We optimise these sub-expressions and based on the optimisation result, we replace the expression itself. For example, in listing 38 we show how we recursively optimise the first sub-expression of a ternary expression. If this first sub-expression optimises to a constant Boolean value of `True`, we can statically replace the entire ternary expression with the optimised version of the second sub-expression. Similarly, if the first sub-expression optimises to `False`, we can replace the ternary expression with the optimised third sub-expression. Because we do want to keep potential optimisations in the first sub-expression, even if it couldn't be optimised to a constant value, we replace the first sub-expression in the ternary expression with its optimised equal.

Similarly, in listing 38 we show how we recursively optimise both sub-expressions of binary operators like binary addition. We pattern match the result of the optimisation of both sub-expressions. If both optimise to constants of the same type, we can replace the optimisation directly with the sum of the two constants. If one or both sub-expressions cannot be optimised to a constant value, we keep the binary addition, but replace its sub-expressions with their optimised versions.

Proving Correctness

To prove that our optimisation routine is correct, we show that it preserves semantic equivalence. For expressions this means that the values they evaluate to are identical to their non-optimised counterparts. More formally: Given that shadow table `t` approximates state `s` and that the non-optimised expression `e` evaluates to value `v` in the context of state `s`, the evaluation of the optimised version of expression `e` must also equal value `v`. We capture this in lemma 3.3.1. Note again that we have not been able to exhaustively complete this proof due to time constraints.

Lemma 3.3.1

$$\mathit{approx}\ t\ s \Longrightarrow \mathit{eval}\ e\ s\ v \Longrightarrow \mathit{eval}\ (\mathit{efold}\ e\ t)\ s\ v' \Longrightarrow v' = v$$

3.3.3 Constant Definitions

Recall that our optimisation function replaces a variable by a constant value if the shadow table contains a mapping from that variable's name to a constant value. The shadow table is initially empty. Whenever we encounter a variable assignment to a constant value, we

```
fun defs :: "statOrDecl ⇒ tab ⇒ tab" where
      "defs SKIP t = t"
    ...
    | "defs (AssignmentStmt vName expr) t = (case efold expr t of
                  (BASE bVal) ⇒ t (vName := Some (baseToVal bVal))
                | _ ⇒ t (vName := None))"
    ...
```

Listing 42: Statements can have an effect on the shadow table. We define the `defs` function to recursively traverse a statement and all of its sub-statements to yield all constant value assignments that occur in those statements.

need to update our shadow table to reflect this constant value. We define a function `defs` that builds the shadow table based on all assignments that occur in a statement and its sub-statements.

The function recursively traverses the statement and its sub-statements. Statements that have no effect on the table, such as the empty statement, directly return the table. If the statement is an assignment statement, the function will use our previously defined optimisation function to optimise the expression contained in the assignment statement. If the optimised expression is a constant value, we update the shadow table to include a mapping from the variable name to this constant value (listing 42).

3.3.4 Statement Optimisation

We use the expression optimisation function and constant declaration function, to optimise entire statements and by extension, entire programs. This means replacing all expressions in all statements and their sub-statements recursively by the optimised versions of those expressions, taking into account new constant declarations as we recursively optimise a statement. For this we define a *mutually dependent* pair of functions `fold` and `folds`. The function `fold` applies the functions we defined in figures 41 and 42 to a singular statement. We need the function `folds`, which applies function `fold` to a list of statements, to optimise the list of statements contained in the block statement. Because these functions depend on each other, they are mutually dependent in their definition (listing 43). Note that we replace the expression in the assignment statement with its optimised counterpart. We use our function to retrieve all constant definitions in a statement in our optimisation of the block statement.

```
fun
  folds :: "statOrDecl list ⇒ tab ⇒ statOrDecl list"
and
  fold :: "statOrDecl ⇒ tab ⇒ statOrDecl"
where
      "folds [] t = []"
    | "folds (stmt # stmts) t =
           ((fold stmt t) # (folds stmts (defs stmt t)))"
    ...
    | "fold (BlockStmt stmts) t = (BlockStmt (folds stmts t))"
    | "fold (AssignmentStmt vName expr) t =
           (AssignmentStmt vName (efold expr t))"
    ...
```

Listing 43: We declare a pair of mutually dependent functions to apply our previously defined optimisation functions to optimise statements. We need both definitions for the optimisation of block statements, which contain a list of further statements.

Proving Correctness

To show that this entire routine is correct, we would need to show that it preserves semantic equivalence: The final program state that can be reached from the initial state is identical for the optimised and non-optimised set of statements. Due to time constraints we were not able to implement this proof completely. To complete this proof, we would need to prove two properties: that all states in the reflexive transitive closure of the optimised statements are reachable in the reflexive transitive closure of the non-optimised statements, as well as that the final state the optimised and non-optimised statements can reach are identical.

3.3.5 Optimisation Verification

Though our proof is not complete, we show that it is possible to use a formalisation like Nano P4 for more than just the verification of P4 applications. The semantics also enables us to reason about program transformations, including optimisation routines. This makes a system like Nano P4 significantly more powerful and versatile than direct program verification engines.

3.4 P4 Parser Formalisation

We have shown that we can formalise the imperative code found in P4's actions and use our formalisation to reason about it. However, P4 does not just contain imperative code. Due to P4's domain-specific nature, P4 also contains entirely different concepts, like the parser state machines. In this section we show that we can extend Nano P4 to include a minimalistical formalisation of P4 parser state machines. We are specifically interested in analysing which transitions are possible and if we can use Isabelle/HOL to optimise such parsers. Moreover, we are interested in analysing the program state that the parser can reach in any parser state. The program state that is reachable in the *accept* parser state is the entry-state of the main control sequence of a P4 application. Hence, being able to reason about this entry-state is useful for formalising an entire P4 application. We show that Isabelle/HOL could therefore be used to also formalise the more unique aspects of P4 and thus used to eventually formalise P4 entirely.

3.4.1 Formalising the State Machine

Parser Syntax

We capture the syntax of the P4 parser state machine using two data types: One to capture a single parser state, and another to capture the parser itself. A state can contain transitions to other states. We are not interested in the exact conditions under which these transitions occur, but only which transitions are possible from a given state. Therefore, we model the possible transitions simply as an ordered list of other parser states. A parser state can also contain imperative code that can change the program state. We capture all of the imperative code in an ordered list of statements. For simplicity, we assume that all imperative code is executed before any transitions are made; capturing these in one ordered list thus suffices. We also include a string to capture the name of the parser state.

The `Parser` data type contains a list of all the states that are declared in that parser. We do not require this list to be ordered; the order becomes apparent from following the transitions. In P4, the entry point is by definition the state named "*start*" [16]. We assume a parser can also execute some sequential code before any of the transitions are called and we capture this code as an ordered list of statements (listing 44).

```
datatype parserState = State name "statement list" "parserState list"
datatype parser = Parser name "statement list" "parserState list"
```

Listing 44: We capture the syntax of P4 parsers and parser states using two data types. Both contain an ordered list of statements. The parser state contains a list of parser states that it can transition to. The parser contains a list of all parser states contained in the parser.

```
fun big_step :: "(statement * varState) ⇒ varState" where
      BEmpty: "big_step (SKIP, s) = s"
    | BDecl:  "big_step ((VarDecl vName expr), s) =
                    s (vName := (eval expr s))"

fun get_varState :: "varState ⇒ statement list ⇒ varState" where
      "get_varState s [] = s"
    | "get_varState s (x # xs) = get_varState (big_step (x, s)) xs"
```

Listing 45: We implement a minimal big-step semantics through a simple functional definition. We also implement a function to yield the final state given an initial state and a list of statements to be executed.

Big-Step Semantics

We are primarily interested in the parser, its parser states, and the possible program states reached in each parser state. Hence, we simplify the semantics we implement for the imperative code of each parser state. In the previous sections discuss Nano P4's small-step semantics. For the purposes of formalising the P4 parser state machine, we instead use a minimal big-step semantics. The big-step semantics will then provide us with the state after a statement is applied. We also implement a function, given a starting state and a list of statements, to compute the final program state after all of the statements have been executed (listing 45).

To prove that our big-step semantics is correct, we prove equality to a subset from our previously defined small-step semantics. This proof is automatic and we formalise it in theorem 3.4.1.

Theorem 3.4.1

$$(big\_step\ cs\ =\ t)\ =\ cs \rightsquigarrow *\ (SKIP,\ t)$$

3.4.2 Reachability Analysis

We define a set of functions that recursively traverse the reachable states, starting from the start state. If an error is encountered during parsing, the state machine automatically transitions to the *reject* state [16]. Therefore, we assume that the reject state is necessarily reachable. Using this we find the set of all reachable states from the start state. We also define a number of functions that use our earlier big-step semantic definition to retrieve the program states that these parser states can be in. Note that if multiple paths exist to the same parser state, their program states might be different. The path leading to a state might change the program state in different ways, leading to this difference. Our analysis sees the same parser state with different program states as different entities.

We define a set of lemmas to prove correctness of our formalisation. For example, we verify that the set of all reachable states is necessarily a subset of the set of all states, including the reject-state. We sanity-check our reachability and unreachability functions by proving lemmas stating that the set of unreachable states has no overlap with the set of reachable states, and vice versa. We formalise this small subset of examples in lemmas 3.4.2, 3.4.3, and 3.4.4 respectively.

Lemma 3.4.2

$$prsr\_reachable\ p \subseteq ((get\_prsr\_states\ p) \cup (State\ "reject"\ []\ []))$$

Lemma 3.4.3

$$pS \in (get\_prsr\_states\ p) \implies (pS \in (prsr\_reachable\ p) \iff pS \notin (prsr\_unreachable\ p))$$

Lemma 3.4.4

$$pS \in (get\_prsr\_states\ p) \implies (pS \notin (prsr\_reachable\ p) \iff pS \in (prsr\_unreachable\ p))$$

3.4.3 Parser Formalisation

Using these definitions we define further functions to yield the *minimal equivalent subset*: the set of reachable states. This filters out *dead states* from the parser. In doing this, we show that it is possible to reason about P4's parser state machines, both in terms of reachable parser states, as well as about reachable program states. We can use such results to reason about the possible entry points of the control sequences captured in a P4 program. We also show that it is possible to use such a formalisation to perform verified transformations on the parser state machines. Combined, this shows that we can also use a system like Nano P4 to verify more domain-specific and unique aspects of P4, such as the parsers.

4

Evaluation

With Nano P4 we provide a formalisation of the imperative code of P4. The main power of Nano P4 is to prove properties about P4 itself. However, we can also provide Nano P4 with specific P4 applications and reason about them. Moreover, because our semantics is an operational one, we can even ask Isabelle/HOL to execute a single execution for us and reason about that single execution. Nano P4 lacks an automated parser, or translator unit, that automatically translates raw P4 code into Nano P4's data types. This means that translating an action requires manual effort. In this section we perform one such translation and we discuss and evaluate this translation process. The translation for this example took under five minutes.

4.1 P4 Actions

4.1.1 Correct P4 Actions

A common and realistic use-case for a P4 action is to decrement the *Time-To-Live (TTL)* counter (listing 46). Here, we take a *parsed header struct* containing the data as produced by the P4 parser. From this, we read the TTL field and copy it into a local variable. We decrement this value by one. Then, we perform a check to see if the TTL counter has reached zero. If so, we set the output port variable `OUT_PORT` to the drop port `DROP_PORT`. This code snippet is valid P4 code and a derivation exists for it if and only if the program state contains a value for the `ttl` field of the header struct.

The P4 action we show in listing 46 contains three sequential statements, with one nested statement. We capture this using a block statement. The first entry in the list assigns to variable `"ttl"` the value extracted from the field `"ttl"` in struct `"header"`.

```
ttl = header.ttl;
ttl = ttl - 1;
if (ttl = 0) {
    OUT_PORT = DROP_PORT;
}
```

Listing 46: Assuming the parsed header struct is valid, this P4 action is correct. The snippet takes the TTL field from the header, decrements it, and checks if it has expired. If so, the output port is set to the drop port.

```
BlockStmt [AssignmentStmt (NameLVal ''ttl'')
                (ExprMem (BASE (DSTRUCT [(''bit<32>'',
                                          ''ttl'',
                                          Some (BUINT 1))])) ''ttl''),
           AssignmentStmt (NameLVal ''ttl'')
                (BIN_MIN (NamedVar ''ttl'') (BASE (BUINT 1))),
           ConditionalStmt (BIN_EQU (NamedVar ''ttl'') (BASE (BUINT 0)))
                (AssignmentStmt (NameLVal ''OUT_PORT'')
                    (NamedVar (''CPU_PORT''))) SKIP
          ]
```

Listing 47: The P4 code shown in listing 46 can be converted into Nano P4 constructs. This yields the above code.

Note that because this concerns a parsed header struct, we use a struct, not a header for this. Here, we do not care about the value of the other header fields, so we omit them for brevity.

The second statement in the list assigns a value to variable `"ttl"`. The new value is a binary subtraction of the constant unsigned integer 1 from the variable `"ttl"`. We captured the TTL field as an unsigned integer, so a non-specified numerical constant matches this type.

The last statement in the list is a conditional statement. The conditional contains an expression of binary equivalence, checking whether the variable `"ttl"` equals the unsigned integer constant 0. The first sub-statement, the `True` case, contains an assignment that assigns to the variable `OUT_PORT` the value of the variable `DROP_PORT`. We assume their types match. The else-clause of the conditional contains only the empty statement, as in the P4 snippet the else-clause is omitted. We show the full Nano P4 conversion in listing 47.

```
<''header'' := (STRUCT [(''bit<32>'',
                         ''ttl'',
                         (UINT 1))]),
                         ''DROP_PORT'' := (UINT 32))>
```

Listing 48: This state contains a `header` struct containing a TTL value of 1 and a `DROP_PORT` of 32. Both of these values are unsigned integers.

```
values "small_step (BlockStmt [
    AssignmentStmt (NameLVal ''ttl'')
        (ExprMem (BASE (DSTRUCT [(''bit<32>'',
                                  ''ttl'',
                                  Some (BUINT 32))])) ''ttl''),
    AssignmentStmt (NameLVal ''ttl'')
        (BIN_MIN (NamedVar ''ttl'') (BASE (BUINT 1))),
    ConditionalStmt (BIN_EQU (NamedVar ''ttl'') (BASE (BUINT 0)))
        (AssignmentStmt (NameLVal ''OUT_PORT'')
            (NamedVar (''CPU_PORT'')))
        SKIP],
    <''header'' := (STRUCT [(''bit<32>'',
                             ''ttl'',
                             (UINT 1))]),
     ''DROP_PORT'' := (UINT 32)>,
    1000)"
```

Listing 49: The Nano P4 data types shown in listing 47 can be executed through our operational small-step semantics. For this we have to provide a program state and progress counter. In this case we pick arbitrary but correct values.

Because our semantics is an operational one, we can ask Isabelle/HOL to execute it. For this, we need to provide the information the execution requires. In our case: a program state and a natural number for the progress counter. The variables that have to be pre-defined in the program state are the `header` struct containing a `TTL` field and the `DROP_PORT`. As arbitrary values, we take an unsigned TTL of 1, and say that the `DROP_PORT` is defined as the unsigned integer 32. The progress counter simply needs to be sufficient, so we arbitrarily take 1000 (listing 48).

We can then use the Isabelle/HOL keyword `values` to have Isabelle/HOL calculate the resulting state. The output is another program state. Because the TTL we picked equals zero on executing the conditional expression, the output state contains our header field with the old TTL of 1, the `DROP_PORT` set to 32, as well as the new `OUT_PORT` set to 32 (listing 49).

```
0 |''header'' := (STRUCTty [(''ttl'', UINTty)]),
1  ''DROP_PORT'' := UINTty,
2  ''OUT_PORT'' := UINTty|
```

Listing 50: This typing environment contains a `header` struct containing a TTL value of 1, and a `DROP_PORT` of 32. Both these values are unsigned integers.

The evaluation shown in listing 49 is correct in the context of the provided state; a derivation exists for it. Porting this simple example from P4 into Nano P4 takes only a few minutes. However, the complexity of such a translation would rise quickly as the complexity of the P4 code increases. Because Isabelle/HOL is essentially a combination of functional programming and logic, it would be possible to write a parser that automatically performs this translation.

4.1.2 Type Verification

Porting the above P4 code into our type system requires us to define the typing environment Γ. This environment has to mirror the variables declared in our program state. In our case, our typing environment should contain a type for the `header` struct and its member field `ttl`, as well as the `DROP_PORT` and `OUT_PORT` variables. The `header` is of type struct. The `ttl` field and both the `DROP_PORT` and `OUT_PORT` variables are unsigned integers of sufficient width. We show how we capture this in the typing environment Γ in listing 50.

The definition of our typing system is also operational. We can again ask Isabelle/HOL to execute and evaluate a statement in the context of a particular typing environment and return whether the statement is correct in this context or not. We can do this in a similar way to how we retrieved an output state by providing Isabelle/HOL with a top-level block statement and the corresponding typing environment Γ (listing 50). Because this P4 code is well-typed, a derivation exists for it and the statement is valid in the context of Γ.

4.1.3 Incorrect P4 Action

In section 3.2 we discuss the usage of uninitialised variables and how Nano P4 disallows their usage. To give an example of this, we look at the code snippet from listing 46 again. The code itself is not incorrect, but its practical correctness depends on the provided program state. If the header field `header.ttl` is not defined in the state when its value is

```
values "|''header'' := (STRUCTty [(''ttl'', UINTty)]),
            ''DROP_PORT'' := UINTty,
            ''OUT_PORT'' := UINTty|
    ⊨ (BlockStmt [AssignmentStmt (NameLVal ''ttl'')
            (ExprMem (BASE (DSTRUCT [(''bit<32>'',
                                      ''ttl'',
                                      Some (BUINT 1))])) ''ttl''),
        AssignmentStmt (NameLVal ''ttl'') (BIN_MIN (NamedVar ''ttl'') (BASE
    (BUINT 1))),
        ConditionalStmt (BIN_EQU (NamedVar ''ttl'') (BASE (BUINT 0)))
            (AssignmentStmt (NameLVal ''OUT_PORT'') (NamedVar (''CPU_PORT'')))
            SKIP])"
```

Listing 51: We can also reason about type-correctness of the Nano P4 statement shown in listing 49. We defined the typing environment Γ in listing 50. Combined, we can ask Isabelle/HOL to return whether the block statement is valid in the context of Γ.

read, no derivation rule exists for the variable `header.ttl`. In such a case Isabelle/HOL reports that no derivation exists and rejects the code. The same applies to incorrectly typed programs and programs that do not adhere to the semantics we define in Nano P4.

4.2 P4 Parsers

We also present a formalisation of P4 parser state machines with Nano P4. Again, Nano P4 lacks a translator to automatically parse raw P4 code into Nano P4 constructs. As an example, observe the simple parser state machine in figure 4.1. This parser extracts an Ethernet header from the packet. If the `ethertype` matches `0x800` for IPv4, it continues with IPv4 extraction. The packet header is accepted if and only if no erroneous data is encountered in the IPv4 header. If any errors are encountered along the way, the packet is rejected and dropped.

We convert this to our formalisation of P4 parsers in listing 52. For brevity we gather all statements at a parser using the `stmts_x` keyword, where x is a unique number for every parser state and the number zero is reserved for the statements of the parser itself. Note that we define these states in reverse order, so that we can define labels using the Isabelle/HOL `let x = y` statements. We can then use this parser formalisation in our functions to retrieve the possible program states at the `accept` state. We omit an example of this, because our semantics for statements in our parser formalisation is minimal. Extending this semantics with the semantics we define in section 3.1 would make this significantly more interesting.

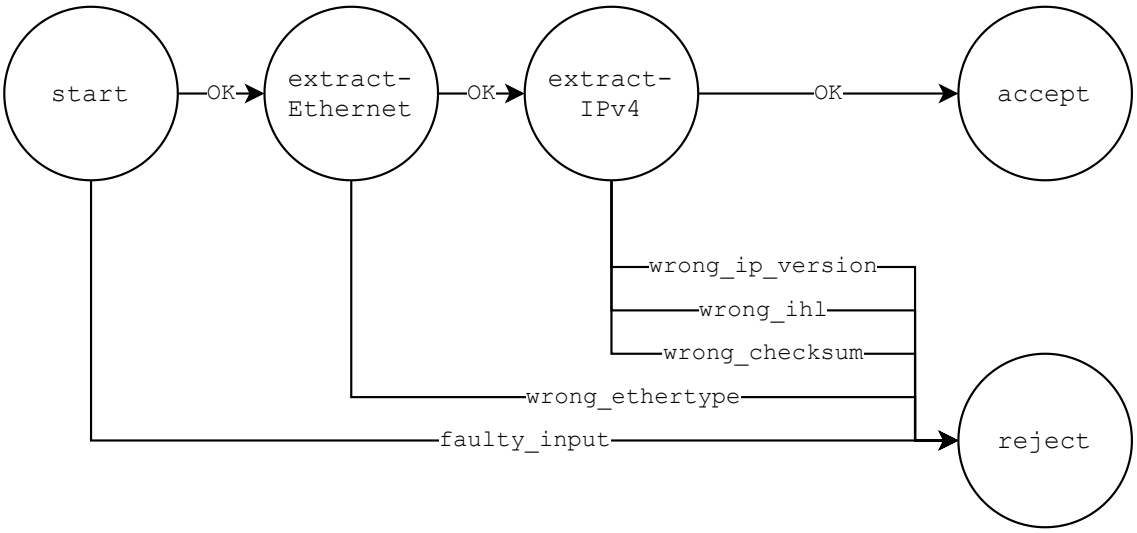

Figure 4.1: A simple diagram illustrating a simple Ethernet and IPv4 parser. The packet is only accepted if the `ethertype` matches `0x800` for IPv4 and all the data in the IPv4 header is valid.

```
0 let p_reject = State ''reject'' [] []
1 let p_accept = State ''accept'' [] []
2
3 let p_e_ipv4 = State ''extract-IPv4'' stmts_3 [p_reject, p_accept]
4 let p_e_enet = State ''extract-Ethernet'' stmts_2 [p_reject, p_e_ipv4]
5 let p_start = State ''start'' stmts_1 [p_reject, p_e_enet]
6
7 let simple_parser = Parser ''simple'' stmts_0
8     [p_reject,p_accept, p_e_ipv4, p_e_enet, p_start]
```

Listing 52

5

Discussion and Future Work

In this chapter we discuss some of the fundamental limitations of Nano P4 (section 5.1) and some potential directions for future work (section 5.2).

5.1 Discussion

5.1.1 P4 Action Verification Limitations

The semantics and typing system we define in Nano P4 provide a powerful tool to analyse and reason about P4 actions. Nano P4 does not feature complete coverage of all of P4's action constructs, but rather formalises a representative subset. For example: Nano P4 lacks the formalisation of function calls. However, function calls in P4 are simple and introduce no new behaviour like recursion; they are merely *syntactic sugar* aimed at easing the programmer's job. Hence we did not focus on implementing such extensions in Nano P4.

Dependent Data Types

A more fundamental limitation is Isabelle/HOL's lack of dependent data types; the creation of one type cannot dynamically depend on another. This makes it more difficult to capture the concept of a bit string whose width is only known at compile-time — let alone dynamically sized bit strings. Isabelle/HOL does support many of the bit-wise operations that P4 includes, such as logical operators like the XOR operator, through the *Machine Words theory* [41]. This library however requires that the width of the bit strings fed to those operators must be known when writing the formalisation and cannot depend on another variable. This makes it difficult to use this library to model arithmetic overflows and underflows dynamically, because semantic rules cannot use the width of a variable in their

definition. A system-wide maximum width could easily be captured using this library, but different widths of individual variables would be difficult to capture using this library — even if the width is known at compile time.

A practical workaround could capture the maximum width of a bit string in a separate parameter, and only define arithmetic operations if their result is above or below the admissible minima and maxima respectively. This is similar to how we captured the maximum size of a header stack in subsection 3.2.1, and used it to limit out-of-bound accesses.

A numerical value could also explicitly be captured as a list of Boolean values instead of using Isabelle's built-in notion of integers and natural numbers. However, applying such a workaround could scale poorly to the entire P4 language and could complicate the P4 semantics in Nano P4.

Alternatively, another proof assistant could be used to formalise P4, such as Coq [21], which does feature support for dependent data types. However, other proof assistants could have other limitations or shortcomings, like lower degrees of proof automation or less literature being available. Hence, we developed our formalisation using Isabelle/HOL.

Type Nesting

A limitation in the way we formalise derived types like structs is that our formalisation only coincidentally adheres to P4's type nesting rules. P4 strictly defines which element types are allowed within each derived type: A struct can contain a tuple, but a header cannot [16]. In section 3.1 we defined the struct type as one of our base types containing further base types recursively. Thus, a struct can contain any further base types, including other structs. Though nested structs are allowed, if we were to define headers and tuples similarly, Nano P4 would allow their nesting arbitrarily. Disallowing specific nestings could be achieved by separating the base type and derived type, but that could make the syntax of our formalisation more complicated to read. Illegal nestings could also be disallowed during the evaluation of expressions and statements, similar to how we disallowed the addition of unsigned and signed integers in their evaluation function.

5.1.2 Compile-Time Known Properties

Nano P4 can be used at compile-time. That is: Nano P4 takes a manually converted P4 program and statically analyses it. Nano P4 cannot perform dynamic verification. All of the properties Nano P4 can reason about and verify must be visible statically. This

is further complicated by P4 having "two sources of input": The incoming data stream and the control plane. Particularly the latter makes it difficult to reason about with which entries the controller will populate the forwarding tables, as the control plane is unknown. A property like "no bad IPv4 address can be inserted into the forwarding table" is an example of an out-of-scope property, because it requires knowledge of the controller. Formalising the controller requires an entirely different — and not necessarily related to P4 — formalisation effort and thus lies beyond the scope for Nano P4. Oppositely, a property like "all actions declared for this forwarding table are valid" is in-scope, because the property relies on definitions within the P4 application.

5.1.3 Separate Verification

We present a formalisation of different aspects of P4 with Nano P4. These exist as separate components in Nano P4. For example, we built our formalisation of P4 actions separately from our formalisation of P4 parser state machines. Moreover, we built a formalisation of P4 actions with complicated data types like structs separately from our formalisation of optimisation routines. This was done due to time constraints, not fundamental constraints. Though it significantly reduces Nano P4's applicability in its current state, we chose this approach because our aim is to show that Isabelle/HOL can be used to verify P4. Building an efficient and applicable version of Nano P4 we leave for future work.

5.2 Future Work

Future research could logically focus on expanding Nano P4 to cover more — and eventually all — of P4. Development could start by expanding Nano P4 to cover the entire range of P4 expressions and statements, rather than an illustrative but minimal subset. The complete version should include the addition of all types of expressions, function calls, method calls, and more. In this section we focus on ways Nano P4 could be extended to include the full spectrum of P4.

5.2.1 Control Constructs

Control constructs are where a user defines actions, tables, and other control flow; they thus make up the brunt of the logic of a P4 program. Formalising these requires formalising P4 match-action tables. In subsection 5.1.1 we mention that formalising exactly which entries can be inserted at run-time requires formalising the controller. The controller runs

a routing algorithm, some of which have already been formalised in Isabelle/HOL [42, 43]. Such a pre-existing formalisation could be combined with Nano P4 to analyse the entries that can be inserted into forwarding tables at runtime.

Instead of formalising each entry that could be inserted into the table, we could restrict our reasoning to the set of possible control-flows within such a table. If we model a match-action table as, given an input state, yielding a set of possible output states, Nano P4 already contains the majority of the components needed. This approach would allow us to reason about the complete set of possible routes a packet could take in a P4 program, while not requiring a complete formalisation of the controller. Formalising the entire control construct then requires chaining these tables together. If one table yields a set of potential output states, they can be seen as input states for the next table. This is similar to how our parser formalisation can yield a state that can be interpreted as the input state of the rest of the P4 program. We could use such a complete formalisation to reason about properties of final program states.

5.2.2 Complete Parser Verification

Furthermore, Nano P4's parser verification could be extended to cover the full complexity of the P4 parser state machine. Currently Nano P4 is able to reason about reachability properties of these parsers and uses a minimal big-step operational semantics to reason about the reachable program and parser states. The big-step semantics of the parser state machine could be replaced by the small-step semantics we defined for P4 actions. This would provide significantly more detailed potential initial states from the parser formalisation: All program states that are possible at the "*accept*" parser state are potential initial states. The initial state for the parser is simply the set of all global variables and architecture-specific variables defined in the architecture description model.

5.2.3 External Objects

Since the release of $P4_{16}$, architecture-specific functionalities available only on some devices have been moved to *external objects*. These can be invoked using the `extern` keyword. The functions to use these external objects are architecture-specific and are implemented by the vendor. Formalising these completely would require full access to the definition of these functions, for all possible architectures. This would likely be a prohibitively large undertaking and we consider such an undertaking as infeasible. External libraries do expose their *function signatures*, meaning the parameters it takes and its return type are known.

These could be used to form a primitive formalisation of the external object and its return type. Even if the exact functionality of the external objects cannot be formalised and analysed in the same way as Nano P4 does, how an external component is used in the P4 application can be formalised.

5.2.4 Convenience

Currently, converting a P4 program into Nano P4 is manual and thus labour-intensive and error-prone. Since Isabelle/HOL is essentially a combination of functional programming and formal logic, a parser or translator to automate this could be written using the functional programming side of Isabelle/HOL. This would make both the conversion process less error-prone — assuming the translation is correct — as well as make this conversion process a lot less time consuming, making the usage of Nano P4 far more practical.

6

Related Work

To the best of our knowledge Nano P4 is the first verification effort for P4 using a proof assistant like Isabelle/HOL. Other verification efforts do exist for P4; in this section we discuss some of them and their core approaches. In section 6.9 we discuss how the other verification efforts relate to Nano P4.

6.1 P4K

The researchers behind P4K [44] note that verifying any language or application starts with defining the unambiguous semantics of the language. They provide a semantics of $P4_{14}$ built using the K framework [45]. The K framework is a rewrite-based executable semantics framework. By defining configurations, computations and rules, a user can define the semantics of programming languages, type systems, and formal analysis tools. K then offers a number of tools this semantics can use, such as a verification engine, symbolic execution engine, and more. The researchers provide an executable semantics of $P4_{14}$ and applications and use the tools built into the K framework to show that this can be used to formally verify P4 applications. Similar to Nano P4, this approach suffers from only being able to verify properties of the language or applications that exist at compile-time. However, the K-framework offers built-in symbolic execution tools that allow them to simulate network packets and analyse the behaviour of a P4 program under these simulated packets. These additional tools make P4K a more practically usable verification tool than Nano P4 in its current state. However, we believe Isabelle/HOL is more prevalent than the K-framework, and thus Nano P4 could be more generally usable than P4K.

6.2 P4-NOD

While Nano P4 provides a small-step semantics, P4-NOD provides a big-step operational semantics, that the researchers behind it use to translate $P4_{14}$ into Datalog [13]. Datalog is a declarative logic language where users define facts, rules, terms, and atoms. This system of declarations can then be solved using a solver. The researchers convert P4 into Datalog and then use a Datalog solver to analyse reachability properties and well-formedness properties of a P4 application. This allows them to quickly make statements about these properties of P4 programs.

6.3 ASSERT-P4

ASSERT-P4 [46] combines assertion-checking with symbolic execution. With ASSERT-P4 a programmer writes annotations into their P4 code; these annotations specify properties they want to verify in their P4 program. ASSERT-P4 then converts the annotated program into a C model and uses symbolic execution to symbolically execute all possible code paths. For each of these ASSERT-P4 checks whether the assertion is violated or not. This allows the researchers behind ASSERT-P4 to uncover bugs in P4 applications. Because their approach relies on user-defined assertions, they rely on the user to define all assertions correctly making the system (human-)error-prone. Because their approach also relies on symbolic execution, they inherit its drawbacks; ASSERT-P4 suffers from slow speeds and has an exponential verification time based on the complexity of the program.

6.4 P4V

P4V similarly asks a user to define safety properties as assertions [10]. Like Nano P4, P4V uses an automated theorem prover, Z3 [32], to determine if a path exists from the initial state to a state violating any of the safety properties. Unlike Nano P4, P4V defines the semantics by translation, rather than specifying the operational semantics itself. It translates a P4 program to Guarded Command Language (GCL) [47], which allows the researchers behind P4V to use already existing systems built atop GCL, such as optimisation routines to speed up the verification process. In doing so P4V can verify programs quickly relative to other P4 verification efforts. One drawback of P4V is that it relies on error-prone user-defined safety properties, rather than a fixed semantics.

6.5 Vera

Similarly built atop symbolic execution is Vera [14]. It defines a new language, NetCTL, where a user can define properties that should be verified in a P4 program. Vera can verify properties including those affected by the control plane, because it analyses P4 snapshots, rather than the compile-time P4 program. This means that in the P4 snapshots, table entries are taken into account and influence the result of the verification. The P4 snapshot is translated into SEFL [48]. SEFL is a language that enables scalable symbolic execution for networks. This gives Vera a program containing virtual ports, which can then use symbolic execution to generate a packet for all possible header layouts. It then observes the behaviour that those packets trigger. By using novel data structures to capture P4 constructs, Vera manages to achieve verification times in the order of seconds/minutes — even while being built using symbolic execution.

6.6 P4-C

P4-C combines symbolic execution, translation, and assertions [49]. Users annotate a P4 program with assertions that express general network correctness properties. The resulting program is transformed into regular C models, similar to ASSERT-P4 [46]. The researchers then use symbolic execution to execute all possible code paths symbolically. If any code path violates any of the assertions, this is marked as a bug. The researchers extend their approach with different optimisation techniques to mitigate the low efficiency of using symbolic execution. In doing so, the researchers are able to verify P4 programs in less than a minute.

6.7 P4AIG

A more novel route is taken with P4AIG [50]. With P4AIG the researchers behind it use sequential circuit analysis to verify P4 programs. P4AIG interprets a P4 program as a hardware pipeline instead of a software program. This is possible because the nature of a P4 program is inherently similar to a hardware pipeline. However, P4AIG is purely conceptual and has not been implemented yet.

6.8 P4RL

Another novel approach is taken by P4RL [12]. Here, the researchers behind it note that due to the dynamic nature of P4 programs, static verification is not very well suited. Rather than verifying the entire P4 program up front, P4RL is a runtime system that checks user-defined assertions at runtime. To check whether an assertion is violated, P4RL uses learning-guided fuzzing. P4RL comes with its own language, P4q, to define security properties as simple conditional statements. These are used to train an agent for triggering runtime bugs automatically. This is then used to verify the P4 program continuously after deployment. The advantage of this approach is the continued guarantee that the program is still correct at any future point in time. However, the nature of fuzzing is such that some test cases could still be missed. Furthermore, such a system needs to run continuously or intermittently, which creates a continuous computational burden, rather than a one-time upfront cost.

6.9 Relation to Nano P4

Many of these approaches are built using symbolic execution to simulate all possible packets and code paths, or verify a P4 program by translation to another language for which a verification engine already exists. This allows many of these approaches to verify specific P4 programs quickly. Unlike Nano P4, however, none of these approaches can reason about the P4 language itself and most do not provide a formal semantics of P4. Another major drawback of most of the approaches is requiring users to annotate their code manually, or specify parameters for testing. The formal semantics in Nano P4 makes for a more robust verification process with a strong and reliable mathematical foundation.

7

Conclusion

In this thesis we show that it is possible to use Isabelle/HOL to formalise parts of the P4 language and use this formalisation to prove properties of P4 and P4 applications. We call our formalisation Nano P4. To the best of our knowledge Nano P4 is the first project to use a proof assistant like Isabelle/HOL to formalise P4 and P4 applications.

We present an executable small-step semantics of P4 actions, including a strict typing system. We use this to prove general properties like termination, as well as security properties like asserting header stacks are not accessed out-of-bounds. Moreover, we show that we can use this semantics to write verified program transformations, such as optimisation routines. Additionally, we present a formalisation of P4 parser state machines. This formalisation can be used to reason about reachability properties, as well as to reason about parser state machine transformations like optimisations.

Nano P4 is not complete; it lacks an automated parser and consists of separate components that each verify a different aspect of P4. Such separate components make it cumbersome to practically use Nano P4 to verify a complete P4 program. Nano P4 also makes assumptions to circumvent some fundamental limitations of Isabelle/HOL, most notably the lack of dependent data types. Due to this limitation we, for example, assume that all numeric values and operations are "wide enough" to contain the intended values. Consequently Nano P4 cannot reason about some arithmetic exceptions.

Even so, we show that using automated theorem provers to verify P4 and P4 applications is feasible. Automated theorem provers offer reliable and trustworthy formal mathematical proofs. Through Nano P4 we can provide absolute guarantees that specific properties hold, rather than testing and hoping that the tests covered all potential bugs. Furthermore, Nano P4 also allows us to reason about properties of the P4 language itself, not just individual applications. This can not only be used to verify correctness of new

additions or changes to the language, but can also be used as a point of reference with which to verify compiler or other implementations. This is unlike most existing verification or formalisation efforts. Because of this, we believe Nano P4 is a useful and powerful tool for the future of P4 and P4 applications.

References

[1] James F Kurose and Keith W Ross. *Computer networking: A top-down approach*. Addison Wesley, 2017. 2, 3

[2] David J Wetherall and David L Tennenhouse. **The active ip option**. In *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, pages 33–40, 1996. 2

[3] J. E. van der Merwe, S. Rooney, L. Leslie, and S. Crosby. **The Tempest-a practical framework for network programmability**. *IEEE Network*, **12**(3):20–28, 1998. 2

[4] Andrew T Campbell, Herman G De Meer, Michael E Kounavis, Kazuho Miki, John B Vicente, and Daniel Villela. **A survey of programmable networks**. *ACM SIGCOMM Computer Communication Review*, **29**(2):7–23, 1999. 2

[5] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. **OpenFlow: enabling innovation in campus networks**. *ACM SIGCOMM Computer Communication Review*, **38**(2):69–74, 2008. 3

[6] Jen Rexford Nick McKeown. **Clarifying the differences between P4 and OpenFlow**, 5 2018. 5

[7] S. Knossen, J. Hill, and P. Grosso. **Hop Recording and Forwarding State Logging: Two Implementations for Path Tracking in P4**. In *2019 IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*, pages 36–47, 2019. 4

[8] Joseph Hill, Mitchel Aloserij, and Paola Grosso. **Tracking network flows with P4**. In *2018 IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*, pages 23–32. IEEE, 2018. 4

[9] Mihai Valentin Dumitru, Dragos Dumitrescu, and Costin Raiciu. **Can we exploit buggy P4 programs?** In *Proceedings of the Symposium on SDN Research*, pages 62–68, 2020. 5, 48, 50

[10] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. **P4v: Practical verification for programmable data planes**. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 490–503, 2018. 5, 78

[11] Andrei Alexandru Agape and Mădălin Claudiu Dănceanu. *P4Fuzz: A Compiler Fuzzer for Securing P4 Programmable Dataplanes*. PhD thesis, Master's thesis, Aalborg University, 2018. 5

[12] Apoorv Shukla, Kevin Nico Hudemann, Artur Hecker, and Stefan Schmid. **Runtime Verification of P4 Switches with Reinforcement Learning**. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, pages 1–7, 2019. 5, 80

[13] Nuno P Lopes, Nikolaj Bjørner, Nick McKeown, Andrey Rybalchenko, Dan Talayco, and George Varghese. **Automatically verifying reachability and well-formedness in P4 Networks**. *Technical Report, Tech. Rep.*, 2016. 5, 78

[14] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. **Debugging P4 programs with Vera**. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 518–532, 2018. 5, 79

[15] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002. 6, 20

[16] Mihai Budiu and Chris Dodd. **The p416 programming language (version 1.2.0)**. *ACM SIGOPS Operating Systems Review*, **51**(1):5–14, 2017. 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 31, 34, 35, 48, 54, 60, 62, 72

[17] Mihai Budiu and Chris Dodd. **The p416 programming language (version 1.2.1)**. *ACM SIGOPS Operating Systems Review*, **51**(1):5–14, 2017. 8

[18] Tobias Nipkow and Gerwin Klein. *Concrete semantics: with Isabelle/HOL*. Springer, 2014. 8, 20, 21, 22, 26, 27, 28, 29, 41

[19] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. **P4: Programming protocol-independent packet processors**. *ACM SIGCOMM Computer Communication Review*, **44**(3):87–95, 2014. 9

[20] P4 Language Consortium. **P4$_{16}$ tutorial slides**, 2017. 10, 11, 12

[21] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. **The Coq Proof Assistant Reference Manual : Version 6.1**. Research Report RT-0203, INRIA, May 1997. Projet COQ. 20, 72

[22] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. **The Lean theorem prover (system description)**. In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015. 20

[23] Lawrence C Paulson. **Natural deduction as higher-order resolution**. *The Journal of Logic Programming*, **3**(3):237–258, 1986. 20

[24] Jeremy Avigad and Kevin Donnelly. **Formalizing O notation in Isabelle/HOL**. In *International Joint Conference on Automated Reasoning*, pages 357–371. Springer, 2004. 20

[25] Christine Röckl, Daniel Hirschkoff, and Stefan Berghofer. **Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the π-calculus and mechanizing the theory of contexts**. In *International Conference on Foundations of Software Science and Computation Structures*, pages 364–378. Springer, 2001. 20

[26] Pierre Chartier. **Formalisation of B in Isabelle/HOL**. In *International Conference of B Users*, pages 66–82. Springer, 1998. 20

[27] Jan Olaf Blech, Lars Gesellensetter, and Sabine Glesner. **Formal verification of dead code elimination in Isabelle/HOL**. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 200–209. IEEE, 2005. 20

[28] Norbert Schirmer. *Verification of sequential imperative programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006. 20

[29] Tobias Nipkow, David Von Oheimb, and Cornelia Pusch. **in a Theorem Prover**. *Foundations of Secure Computation*, **175**:117, 2000. 20

[30] Martin Strecker. **Formal verification of a Java compiler in Isabelle**. In *International Conference on Automated Deduction*, pages 63–77. Springer, 2002. 20

[31] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. **Cvc4**. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011. 20

[32] Leonardo De Moura and Nikolaj Bjørner. **Z3: An efficient SMT solver**. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. 20, 78

[33] Stephan Schulz. **E–a brainiac theorem prover**. *Ai Communications*, **15**(2, 3):111–126, 2002. 20

[34] Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. **A verified SAT solver framework with learn, forget, restart, and incrementality**. *Journal of automated reasoning*, **61**(1-4):333–365, 2018. 20

[35] Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes, and Cesare Tinelli. **Model finding for recursive functions in SMT**. In *International Joint Conference on Automated Reasoning*, pages 133–151. Springer, 2016. 21

[36] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, **2283**. Springer Science & Business Media, 2002. 29

[37] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. **The OCaml system release 4.02**. *Institut National de Recherche en Informatique et en Automatique*, **54**, 2014. 29

[38] Arthur M Geoffrion. **The SML language for structured modeling: Levels 1 and 2**. *Operations Research*, **40**(1):38–57, 1992. 29

[39] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. **The Scala language specification**, 2004. 29

[40] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. 29

[41] Jeremy Dawson. **Isabelle theories for machine words**. *Electronic Notes in Theoretical Computer Science*, **250**(1):55–70, 2009. 71

[42] Timothy Bourke, Rob van Glabbeek, and Peter Höfner. **A mechanized proof of loop freedom of the (untimed) AODV routing protocol**. In *International Symposium on Automated Technology for Verification and Analysis*, pages 47–63. Springer, 2014. 74

[43] Huabing Yang, Xingyuan Zhang, and Yuanyuan Wang. **A correctness proof of the srp protocol**. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 7–pp. IEEE, 2006. 74

[44] Ali Kheradmand and Grigore Rosu. **P4K: A formal semantics of P4 and applications**. *arXiv preprint arXiv:1804.01468*, 2018. 77

[45] Grigore Rosu and Traian Florin Serbănuţă. **An overview of the K semantic framework**. *The Journal of Logic and Algebraic Programming*, **79**(6):397–434, 2010. 77

[46] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. **Uncovering bugs in p4 programs with assertion-based verification**. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018. 78, 79

[47] Edsger W Dijkstra. **Guarded commands, nondeterminacy and formal derivation of programs**. *Communications of the ACM*, **18**(8):453–457, 1975. 78

[48] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. **Symnet: Scalable symbolic execution for modern networks**. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 314–327, 2016. 79

[49] Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, and Marinho Barcellos. **Verification of p4 programs in feasible time using assertions**. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 73–85, 2018. 79

[50] Mohammad A Noureddine, Amanda Hsu, Matthew Caesar, Fadi A Zaraket, and William H Sanders. **P4AIG: Circuit-Level Verification of P4 Programs**. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks–Supplemental Volume (DSN-S)*, pages 21–22. IEEE, 2019. 79