Fall 2018

Pip: An Abstract Dataplane and Virtual Machine

Samuel Goodrick
sdg31@zips.uakron.edu

Pip: An Abstract Dataplane and Virtual Machine

Samuel Goodrick

April 27, 2018

Abstract

We present an abstract machine and S-expression-based programming language to describe OpenFlow-style software-defined networking. The implemented Pip virtual machine and language provide facilities for packet decoding, safely writing and setting bitfields within packets, and switching based on packet contents. We have outlined an abstract syntax and structural operational semantics for Pip, thus allowing Pip programs to have predictable and provable properties. Pip allows for easy and safe access and writing to packet fields, as well as a programmable packet pipeline that will rarely stall.

1 Introduction

Conventional switches are configured using vendor-specific interfaces that lack scalability and flexibility. In addition, manual switch configuration is heavily prone to error and compromise. Software-defined networking (SDN) is a paradigm that helps to mitigate user error in packet switching and increase network scalability by allowing switches to be programmed through a high-level programming language. However, there is no standard language for SDN; some programmers have tried to use C or Python, others have developed domain-specific languages to achieve these purposes. There are problems using both general purpose languages and domain-specific languages. The semantics of general purpose languages make heavy use of program constructs that allow for invocation of undefined behaviors such as buffer overflow, pointer errors, data races, and so on. On the other hand, domain-specific languages, may not be sufficient to express the kinds of computations necessary for higher-level SDN applications [1]. We wanted either to build a programming language for SDN, or to identify existing languages that might extended to safely and efficiently support SDN. To do this, we needed to find a "sweet spot" between the two extremes of fully general versus highly specific languages. To begin the study of safe SDN languages, we attempted to build a virtual system for SDN that enables us to study and reason about SDN programs, assisting us in identifying what properties of a language are most useful for safely and efficiently supporting SDN. This paper describes the design and implementation of that system.

Our approach is to create a minimal abstract machine to represent the kinds of computational facility present in a typical OpenFlow-like data plane. The description of our abstract machine includes basic notions of state and the smallest possible set of operations needed to define SDN programs. In essence, we attempted to construct an assembly language for OpenFlow-style packet processing. We defined the abstract syntax of a programming language that operates the machine as well as the operational semantics for the abstract machine. In order better experiment with it, we also implemented a virtual machine capable of executing SDN programs.

Now that we have a model, we hope to start reasoning about higher level language constructs that support safe and efficient switch configurations. We hope to use our findings to construct minimal abstract machines out of subsets of other dataplane domain-specific languages. Defining an abstract machine through operational semantics is especially useful when attempting to prove properties about a program, as operational semantics

of a language give a good basis for proof through logic and dynamic logic. In the future, we would like to examine techniques for dynamic logical analysis of dataplane programs.

The Pip language has no object model or declaration system, as well as minimal control structure. By removing much of the freedom present in general-purpose programming languages and restricting the programmer to simple packet-switching actions and buffer modification, Pip programs are able to maintain provable properties and provable termination. The operational semantics of Pip allow for a logical description of the state of the program at any time. In order to maintain these provable properties, we allow the programmer to output a modified packet to a "controller", a program written in a general-purpose programming language that has the facilities for mutable state or abstract-syntax tree modification. Separating the controller from the dataplane allows for packet switching to be reasoned about mathematically without introducing undecidable outcomes.

Despite its previously mentioned advantages, the lack of the object model comes with a possible set of undefined behaviors. Our storage-based model does not stop the programmer from performing overlapping writes of a buffers or reading across packet boundaries. In the future, Pip may include a non-intrusive declaration system, allowing the user to declare packet headers and use them like variables, thus limiting the access the programmer has, at any time, to single packet headers. This functionality was previously explored in the Steve language. Our abstract machine provides other opportunities for executing undefined behaviors as well. In particular, it is easy to build programs that fail to validate input, as a consequence of implementation.

This paper is organized as follows: Section 2 demonstrates the use of the Pip programming language to create simple SDN applications. Section 3 gives a detailed explanation of and serves as a reference to the Pip programming language. It provides proper documentation to the various language constructs of Pip.

Section 4 describes the abstract syntax of the Pip language, first presenting the types and expressions of the language before defining the proper formation of actions, or the instructions of the language. Section 5 explains the operational semantics and state of the machine, and shows a transition relation for all actions presented in Section 4.

Section 6 describes the implementation of the virtual machine and language interpreter.

Section 7 provides discussion of some undefined behaviors and issues with the language. Section 8 discusses future work and solutions to the issues of Pip discovered in related work.

2 Sample Programs

In this chapter we describe three Pip programs. First, we will see two simple, static Pip programs: a stateless wire and stateless firewall program. More complicated (stateful) programs can be created by outputting to the controller port and dynamically adjusting the abstract syntax tree. Doing so removes any proveable properties about a Pip program. The third and final program, a stateful wire, will use a controller program.

2.1 Stateless Wire

A wire is a common network function involving two ports: when one of the ports receives a packet, it outputs on the other port. In general, the wire is not aware of either port and must learn of their existence dynamically (cite hoang). Pip does not allow for dynamic state changes, thus since the wire does not have any capability to learn of ports dynamically, will simply send a packet to the other of two known ports, port 1 and port 2.

```
(pip
  (table wire exact
    (actions
```

```
      (copy
        (bits physical_port 0 32)
        (bits key 0 32)
        32)
      (match)
    ) ; actions
    (rules
      (rule (port 1)
        (actions (output (port 2)))
      (rule (port 2)
        (actions (output (port 1)))
      (rule (miss)
        (actions (drop))
      )
    )
  )
)
```

We begin by defining an *exact match* table called **wire**. An exact match table matches on integers. When the table begins execution, 32 bits are copies from the **physical_port** of the packet (a context variable representing the physical ingress port of the packet) into the table's **key** register. When the **match** action is reached, table prep ends and matching can begin. Three *match rules* are defined: port 1, port 2, and **miss**. If the value of the **key** register is equal to one of these rules, the rule's *action list* begins execution. For example, if the **key** register was equal to 1, then the packet would **output** to port 2. If the **key** was not equal to 1 or 2, the **miss** rule would begin execution, dropping the packet.

2.2 Stateless Firewall

Here we define a simple TCP/IPv4 firewall that disables some types of packets from accessing the Internet. If the packet is of type IPv4 with protocol TCP, we will drop any packets to port 80 (HTTP) or port 443 (HTTPS). If the packet is a TCP/IPv4 packet but is not attempting to access ports 80 or 443, we will allow the controller to handle egress processing, as the output port of the packet will still need to be determined.

```
(pip
  (table ipv4_check exact
    (actions
      (copy
        (named_field eth.type)
        (bits key 0 16)
        16)
      (match)
    )
    (rules
      (rule (int i16 0x0800)
        (actions (goto tcp_check)))
      (rule (miss)
        (actions (drop)))
    )
  )

  (table tcp_check exact
    (actions
```

```
      (copy
        (named_field ipv4.type)
        (bits key 0 8)
        8)
      (match)
    )
    (rules
      (rule (int i8 0x06)
        (actions (goto firewall)))
      (rule (miss)
        (actions (drop)))
    )
  )

  (table firewall exact
    (actions
      (copy
        (named_field tcp.src)
        (bits key 0 16)
        16)
      (match)
    )
    (rules
      (rule (int i16 80)
        (actions (drop)))
      (rule (int i16 443)
        (actions (drop)))
      (rule (miss)
        (output (port local)))
    )
  )
)
```

Here we see a program taking advantage of the **goto** action control structure. The ipv4_check table will match on the *Ethernet ethertype* of the packet. If the ethertype is 0x800, IPv4, we will jump to the tcp_check table, dropping the packet in any other case. In *tcp_check*, we match on the *IPv4 protocol* field. If it is equal to 0x06, or TCP, we will jump the final table, *firewall*. Here, we will match the *TCP source port* on ports 80 and 443. If there is a match, we drop the packet, disallowing Internet access. Otherwise, we output to the reserved `local` **port** to continue processing on the local IP stack.

2.3 The Wire

Meaningful computation can rarely be achieved while maintaing an immutable abstract syntax tree. As such, we will examine a Pip program that outputs to the `controller` port, and see how its abstract syntax tree is modified dynamically to create a useful application.

In the beginning of the chapter, we discussed a stateless wire. There is little value in this program, but we can modify it to create a fully functional wire. Recall that a wire involves two ports, when one port receives a packet, it outputs to the other. In the stateless wire, these two ports were predefined and immutable. Here, we will create a wire that learns of its ports' existence dynamically.

```
(pip
  (table wire exact
```

```
    (actions
      (copy
        (bits physical_port 0 32)
        (bits key 0 32)
        32)
      (match)
    )
    (rules
      (rule (miss)
        (actions (output (port controller)))
      )
    )
  )
)
```

We begin with a Pip program that contains only a miss rule. The program examines the physical ingress port of a packet and, upon missing, outputs to the `controller`. Now, we will create an algorithm to modify the AST and insert a rule where needed. We define this algorithm below. Note: we assume `eval` is an instance of the evaluator, containing the full evaluation state. We assume basic data structure accessor functions, such as `front` and `push` are defined for list structures. Even though we only wish to add 2 rules, the table will have 3 total rules at the end of the program, since the **miss** rule must be accounted for. Because we push to the back of the rule list, the miss rule will be at the 0th position in the list. The controller iterates

Algorithm 1 Wire Controller

```
let table = front(program.tables)
let port = eval.ingress_port
foreach rule ∈ table.rules
    if(port ≠ rule and size(table.rules) < 3)
        push(table.rules, new rule(key = port))
    else if(size(table.rules) = 3)
        table.rules_0.action_list = new action_list(new drop_action)
```

through all rules in the only table in the program. If both new port-matching rules have not been added to the table, the current physical ingress port is added as a rule. Once both rules have been added, the miss rule is transformed to have only a **drop** action in its action list. After this controller program has run for several packets with different inputs, a reserialized Pip program will look identical to the Stateless Wire above.

3 The Pip Language

This section serves as a reference for the grammar and syntax of the Pip programming language.

Pip is an S-expression-based language. There are 6 basic S-expressions used in the Pip language:

1. Pip
2. Table
3. Action List
4. Action
5. Rule List
6. Rule

3.1 Pip S-Expression

Pip S-Expressions are the expressions that contain the program itself, comparable to the main function in C. Pip expressions can contain any amount of table expressions as parameters. Example:

```
(pip (table ...) (table...))
```

3.2 Table S-Expression

Table S-Expressions define a match table. They contain, in order, a name, a table type (exact match tables are currently the only supported table type), a preparation action list (an action list that sets the key register of the table) ending with a **match** action, and a rule list that defines the matching rules of the table. For example:

```
(table table_name exact
  (actions
    (set (bits key 0 32) (int i32 0))
    (match)
  )
  (rules
    (rule ...)
  )
)
```

In this table, an exact-match table called *table_name* is declared. Its key register is **set** to the 32-bit integer 0 and matching begins. The rules in the rule list will then be examined and possibly executed. Rules and key registers will be explained later in the chapter.

3.3 Action Lists and Actions

Actions are the basic instructions in Pip. Actions Lists are simply sequences of one or more actions. Pip has 9 actions, described as follows:

1. **write** writes an action the stored action list of the virtual machine.
2. **clear** clears all actions in the stored action list.
3. **drop** drops the packet and discontinues processing.
4. **match** ends the prep list of a table and begins matching.
5. **goto** jumps to another table. Only tables that appear later in the program may be jumped to.
6. **output** sends the packet out a specific port.
7. **advance** increments the context variable **header_offset** by the specified amount. This determines the distance in bits of the **header** address space from the **packet** address space.
8. **copy** copies n bits from one bitfield to another. The programmer may not copy past th eend of an address space.
9. **set** sets a bitfield to some literal value.

An example action list follows:

```
(actions
```

```
  (set (named_field tcp.dst) 443) (drop))
```

An action is marked by the **action** keyword. Individual actions are not marked by a keyword as they only appear within action lists and write action parameters.

3.4 Rule Lists and Rules

Rules are the entries in a table that are matched upon. A rule has a key (not to be confused with a **key** register) that is equality compared to the **key** register. In an exact match table, a rule key is of type **int**(n). Along with a key, a rule contains an action list that will be executed if the rule key matches the key register. A rule list is a sequence of one or more rules. An example rule list follows:

```
(rules
  (rule (int i32)
    (actions ...)))
```

A rule is marked by the **rule** keyword and a rule list by the **rules** keyword.

3.5 Key Register

The **key** register is a 64-bit field that is contained in the table data structure. The **key** register is equality compared to the table's **rule** keys. In other words, matching in a table works similarly to switch statements in C.

3.6 Address Spaces

There are 4 address spaces in the Pip language and virtual machine:

1. **packet** denotes the 0th bit in the packet.
2. **header** denotes the 0th bit in the packet plus the value of the **header_offset** context variable.
3. **meta** is the 64-bit metadata register. It can be used to store values for scratch processing.
4. **key** is the 64-bit key register.

3.7 Reserved Ports

Pip inherits the following reserved ports from the OpenFlow standard:

1. **drop** or port 0. Outputting to this port has the same effect as the **drop** action.
2. **all** outputs to all ports on the device.
3. **controller** sends the packet to some non-Pip program. Our implementation uses C++ programs.
4. **in_port** outputs to the physical ingress port of the packet.
5. **any** outputs to a random port.
6. **local** outputs to the local IP stack.

3.8 Named Fields

Named fields are bitfields; they have type **bits**. They are equivalent in every way to a bitfield in the **packet** address space, and only exist as syntactic sugar, but help to prevent memory access violations. A named field should always be preferred to a raw bitfield if available. Named fields exist in Pip for Ethernet frames, IPv4 fields, and TCP fields. The naming scheme uses common abbreviations, for example **eth.dst** representing the Ethernet destination MAC address, and **ipv4.len** representing the IPv4 header length. IPv6 and UDP will be added in the future.

4 Abstract syntax

In this section, we explain the abstract syntax of the Pip programming language. Pip uses a type system to ensure semi-semantically correct instructions. Instructions, known as actions, can take well-formed expressions or actions as parameters. See Section 4.3 for specification of action parameters. This section defines abstract syntax in Backus-Naur Form, an interested reader can learn more about this notation in [2] and [3].

4.1 Types

$\tau ::=$	**int**(n)	integer types
	bits(a, o, l)	bitfields
	port	ports
	table	tables

1. **int**(n) types are unsigned integers with a bit-width of size n.
2. **bits** types are bitfields within the storage space of the virtual machine. Bitfields are defined as having an *address space* that denote which area of storage they reside: packet, header, key, meta, ingress_port, and physical_port. A bitfield must also specify its length in bits as well as its offset from bit 0 of the address space, again in bits.
3. **port** types represent numbered ports. Some reserved ports are named, and occupy a static number. For example, the **drop** port is assigned to 0.
4. **table** types are data structures that contain all other Pip constructs. A table has a *key register* to be matched against. The programmer defines the value of the key register, which is computed at runtime. Tables have *rules* which, if equal to the key register, will execute a block of actions.

4.2 Expressions

expressions ::=	**int**$(w)n$	where w and n are integer literals.
	| **port** n	where n is an integer literal.
	| **bits** $as\ pos\ len$	where pos and len are integer literals.
		and as is an address space.
	| **miss**	
	| **ref** id	where id is the name of a table.

4.3 Actions

Actions are instructions that either have no parameters, an expression as a parameter, or another action as a parameter.

actions ::=	**write** *action*	*action* must be an **action**.
	| **clear**	
	| **drop**	
	| **match**	
	| **goto** *table*	table must be of type **table**.
	| **output** *port*	port must be of type **port**.
	| **advance** n	n must be an integer literal.
	| **copy**src dst n	src and dst must be of type **bits**
		n must be of type **int**(w).
	| **set**$field$ $value$	$field$ must be of type **bits** and
		$value$ must be of type **int**(w).

5 Operational Semantics

5.1 Notation

The state of the evaluator, σ, is the tuple consisting of
packet, π
action_list, α
stored_action_list, Λ
context, Γ, which is the tuple:
$\langle ingress\_port,\ egress\_port,\ physical\_port,\ key,\ meta,\ header\_offset,\ table \rangle$

Compactly, $\sigma = \langle \pi, \alpha, \Lambda, \Gamma \rangle$

We can examine the status of π or Γ by using the index operator, []. The index itself is context-sensitive: π is indexed by some *field*, Γ is indexed by one of its substructures. We will use the assignment operator, $\leftarrow$, within the index operator to assign a bitfield to a value or the value of another bitfield. A field in π can be set to a literal value or copied from a bitfield or register. For example:

$$\pi[field \leftarrow value] \tag{1}$$

$$\pi[field \leftarrow \mathbb{r}] \tag{2}$$

$$\pi[field \leftarrow bitfield] \tag{3}$$

A variable in Γ is accessed similarly, but uses the dot operator, ., to access its substructures within the index operator. Example:

$$\Gamma[\Gamma.\textbf{key} \leftarrow bitfield] \tag{4}$$

One should not confuse the range operator, [), with the index operator. To avoid ambiguity, the range operator *always* appears inside of the index operator, []. Example:

$$\pi[field_1 \leftarrow field_2[0, 10)] \tag{5}$$

Comments:

1. Field *field* with a packet, π, is set to some literal value, *value*.
2. The value of register r is copied into *field*.
3. The value of some bitfield, *bitfield* is copied into *field*.
4. The value of the **key** register in Γ is set to the value of *bitfield*
5. The bits of $field_1$ in π are set to the bits 0, up to but not including 10, in $field_2$.

5.2 Operational Semantics 5-Tuple

Turbak and Gifford define operational semantics for a language as the 5-Tuple, $\langle CF, \Rightarrow, FC, IF, OF \rangle$. CF is the set of all possible configurations that an abstract machine may be in. $\Rightarrow$ is the transition relation, a function that transitions from one configuration to another. FC is the set of all final configurations, the possible ending configurations of the machine. IF is the input function, a function that translates an input into the beginning state of the machine. Finally, OF is the output function, or the transition of the final state into some useful value or domain [4]. In Pip, we define the 5-tuple as follows:
pip = $\langle IF,\ CF,\ \Rightarrow,\ FC,\ OF \rangle$
IF = program $\times\ \pi \rightarrow CF$
CF = $\alpha \times \Gamma \times \pi \times Lambda$
$\Rightarrow$ = Defined in section "Transition Relation".
FC = $(\Lambda = \varnothing) \times (\alpha = \varnothing) \times \pi \times \Gamma[egress\_port \neq 0]$
OF = $FC \rightarrow$ `serialization`

`serialization` is the output state of Pip: a textual logging of all states that the machine entered.

5.3 Transition Relation

$(\textbf{advance}\ n \cdot \bar{\alpha}), \langle \pi, \Lambda, \Gamma \rangle \Rightarrow (\bar{\alpha}), \langle \pi, \Lambda, \Gamma[\textbf{header} \leftarrow \textbf{header} + n] \rangle$
$(\textbf{clear} \cdot \bar{\alpha}), \langle \pi, \Lambda, \Gamma \rangle \Rightarrow (\bar{\alpha}), \langle \pi, \varnothing, \Gamma \rangle$
$(\textbf{drop} \cdot \bar{\alpha}), \langle \pi, \Lambda, \Gamma \rangle \Rightarrow \varnothing, \langle \pi, \varnothing, \Gamma[\textbf{out} \leftarrow 0] \rangle$
$(\textbf{write}\ action \cdot \bar{\alpha}), \langle \pi, \Lambda, \Gamma \rangle \Rightarrow (\bar{\alpha}), \langle \pi, \Lambda \cdot action, \Gamma \rangle$
$(\textbf{goto}\ T \cdot \bar{\alpha}), \langle \pi, \Lambda, \Gamma \rangle \Rightarrow (\bar{\alpha}), \langle \pi, \Lambda, \Gamma[\textbf{current\_table} \leftarrow T] \rangle$
$(\textbf{set}\ field\ value \cdot \bar{\alpha}), \langle \pi, \Lambda, \Gamma \rangle \Rightarrow (\bar{\alpha}), \langle \pi', \Lambda, \Gamma' \rangle$
$(\textbf{output port}\ n \cdot \bar{\alpha}), \langle \pi, \Lambda, \Gamma \rangle \Rightarrow (\bar{\alpha}), \langle \pi, \Lambda, \Gamma[\textbf{egress\_port} \leftarrow \textbf{port}\ n] \rangle$
$(\textbf{copy}\ as_1\ src\ as_2\ dst \cdot \bar{\alpha}), \langle \pi, \Lambda, \Gamma \rangle \Rightarrow (\bar{\alpha}), \langle \pi', \Lambda, \Gamma' \rangle$

The **advance** command advances the header offset by n bits. This can be used by a programmer to adjust the decoding offset relative to a packet header. The **clear** command clears the stored action list. This can be used by a program reset any accumulated actions if certain conditions are detected during packet analysis (e.g., filtering flows based on UDP packet content). The **drop** command truncates both the action list and stored action list, and sets the output port to 0 (the null port). The effect of this to cause the program to terminate such that the execution environment will not forward the packet. The **write** command appends an action to the stored action list. This is the only way to write to the stored action list, allowing for some action to be taken upon egress, providing the packet is not dropped. The **goto** command jumps from the current match table to a new table. The key register is cleared and rewritten based on the new table's rules. This can be used as a rudimentary control structure. The **set** command sets a bitfield, *field* to some value *value*. This is used to set fields to a literal value. The **output** action outputs to the specified **port**, n, beginning egress processing. The **copy** action copies from one bitfield, *src* to another bitfield, *dst*. A bitfield

can be copied from any address space to any other address space, except for key registers, which cannot be used as a source.

Definition of copy function:

$$
\begin{aligned}
&(\textbf{copy}\ meta\ src\ packet\ dst\ n) \rightarrow \langle \pi[dst \leftarrow \Gamma.\textbf{meta}[dst, dst+n)], \Lambda, \Gamma\rangle \\
&(\textbf{copy}\ meta\ src\ header\ dst\ n) \rightarrow \langle \pi[dst+\textbf{hoff} \leftarrow \Gamma.meta[dst, dst+n)], \Lambda, \Gamma\rangle \\
&(\textbf{copy}\ meta\ src\ key\ dst\ n) \rightarrow \langle \pi, \Lambda, \Gamma[\Gamma.\textbf{key} \leftarrow \Gamma.\textbf{meta}[meta, meta+n)]\rangle \\
&(\textbf{copy}\ packet\ src\ packet\ dst\ n) \rightarrow \langle \pi[dst \leftarrow \pi[src, src+n)], \Lambda, \Gamma\rangle \\
&(\textbf{copy}\ packet\ src\ header\ dst\ n) \rightarrow \langle \pi[dst+\textbf{hoff} \leftarrow \pi[src, src+n)], \Lambda, \Gamma\rangle \\
&(\textbf{copy}\ packet\ src\ key\ dst\ n) \rightarrow \langle \pi, \Lambda, \Gamma[\Gamma.\textbf{key} \leftarrow \pi[src, src+n)]\rangle \\
&(\textbf{copy}\ header\ src\ header\ dst\ n) \rightarrow \langle \pi[dst+\textbf{hoff} \leftarrow \pi[src+\textbf{hoff}, src+\textbf{hoff}+n)], \Lambda, \Gamma\rangle \\
&(\textbf{copy}\ header\ src\ packet\ dst\ n) \rightarrow \langle \pi[dst \leftarrow \pi[src+\textbf{hoff}, src+\textbf{hoff}+n)], \Lambda, \Gamma\rangle \\
&(\textbf{copy}\ header\ src\ key\ dst\ n) \rightarrow \langle \pi, \Lambda, \Gamma[\Gamma.\textbf{key}[dst] \leftarrow \pi[src+\textbf{hoff}, src+\textbf{hoff}+n)]\rangle \\
&(\textbf{copy}\ header\ src\ meta\ dst\ n) \rightarrow \langle \pi, \Lambda, \Gamma[\Gamma.\textbf{meta}[dst] \leftarrow \pi[src+\textbf{hoff}, src+\textbf{hoff}+n)]\rangle
\end{aligned}
$$

Defintion of set function:

$$
\begin{aligned}
&(\textbf{set}\ (meta\ field\ n)\ value) \rightarrow \langle \pi, \Lambda, \Gamma[\Gamma.\textbf{meta}[field, field+n) \leftarrow value]\rangle \\
&(\textbf{set}\ (key\ field\ n)\ value) \rightarrow \langle \pi, \Lambda, \Gamma[\Gamma.\textbf{key}[field, field+n) \leftarrow value]\rangle \\
&(\textbf{set}\ (packet\ field\ n)\ value) \rightarrow \langle \pi, \Lambda, \Gamma[\Gamma.\textbf{packet}[field, field+n) \leftarrow value]\rangle \\
&(\textbf{set}\ (header\ field\ n)\ value) \rightarrow \langle \pi, \Lambda, \Gamma[\Gamma.\textbf{packet}[field+\textbf{hoff}, field+\textbf{hoff}+n) \leftarrow value]\rangle
\end{aligned}
$$

Note: the context variable **header_offset** is abbreviated to **hoff** for brevity.

6 Implementation

Pip was implemented in C++ using Dr. Andrew Sutton's Sexpr and CC libraries [5] [6]. Interpretation of a Pip program happens in five stages: input, sexpr-parsing, translation, name resolution and evaluation. Input and sexpr-parsing are handled by libsexpr and libCC and will be ignored in this paper. The implementation of translation, resolution, and evaluation will be outlined in this chapter. The implementation is hosted on GitHub [7].

6.1 Translation

The translator takes, as a parameter, a libsexpr expression and returns a declaration. It works as a recursive-descent, syntax-directed translator built around the `match_list` framework of libsexpr. The `match_list` framework allows for pattern-matching of user-defined s-expressions, with each unique type being parsed by a callback function.

The types that the translator parses are defined below. Keywords and syntax appear in **sanserif** font.

d ∈ decl ::= **(table** table_id table_kind action_seq rule_seq**)**

decl_seq ::= decl*

a ∈ action ::= (action_name)
| (action_name expr*)
| (action_name action*)

action_seq ::= action*

r ∈ rule ::= (expr action_seq)

rule_seq ::= rule*

e ∈ expr ::= int_expr
| port_expr
| miss_expr
| ref_expr
| bits_expr
| named_field_expr

Here **action_name** refers to the syntactic name of an action as defined in Sections 3.3 and 4.3.

We can see a faithful implementation of this backus-naur form definition is incredibly simple in C++ and `match_list`. Below is the implementation of `decl_seq` and `decl`:

```
decl_seq
translator::trans_decls(const sexpr::expr* e)
{
  decl_seq decls; //decl_seq = std::vector<decl*>
  for(auto el : e) {
    decl* d = trans_decl(e);
    decls.push_back(d);
  }

  return decls;
}

/// decl ::= table-decl
decl*
translator::trans_decl(const sexpr::expr* e)
{
  symbol* sym; // symbol = std::string*
  match_list(e, &sym);
  if (*sym == "table")
    return trans_table(e);
}
```

Translation of all types works similarly: a sequence is translated by iterating through each s-expression within a list of s-expressions. The parser gets recursively translates each element of the s-expression into a Pip AST, storing the completed ASTs in a vector. We see in this example, an overloading of `match_list`

to handle symbols, or string pointers, although the function can take any amount of arguments. Defining overloads of `match_list` functions is beyond the scope of this paper.

6.2 Resolution

Name resolution is responsible for performing name lookup after the first translation phase. It is responsible for assigning meanings to **ref_expr**, or references to table names.

Resolution itself has two phases: first, a table linking locations in memory to identifiers is created; second, those identifiers are replaced with their pointers to their location.

6.3 Evaluation

The final stage of interpretation is evaluation. The evaluator framework is an entire virtual machine, containing the entire Pip state: a context, packet, action list, and stored action list. An evaluator operates upon a single packet; a Pip program that uses n packets will have n evaluators.

Upon construction of an evaluator, all table declarations are initialized with their static rule lists. The first table in the program is set as the **current_table** and its prep-action-list is queued into the evaluator. The evaluator will execute each action in its queue until the queue is empty. An abridged pseudocode version of evaluator construction is defined below:

Algorithm 2 Evaluator Construction

```
let tables = vector of declarations
foreach table ∈ program
    insert table into tables
    foreach rule ∈ table
        insert rule.key into hashtable
```

The main algorithm of the evaluator is the `step` algorithm, which switches on the next action in the queue, and recursively executes them in turn.

Algorithm 3 step

```
while !empty(eval) do
    action a ← front(eval)
    if get_kind(a) = advance_action then return eval_advance(a)
    else if get_kind(a) = copy_action then return eval_copy(a)
    ...
```

The most complex algorithms are found in the evaluation of actions. The evaluation of the copy action, for example, involves copying bits in place from integer values into byte arrays, odd numbers of bites from one byte array to another, and from byte arrays back into integers. These become incredibly dense very fast and will not be examined. In the name of brevity, we will only study the most important action, **match**. The `eval_match` algorithm looks up the key register's value in a hashtable of the table's rule keys. If there is a match, the current rule's action list will be added to the evaluator queue. The hashtable find function returns the integer identifier of the rule, so that the matched rule can be kept track of. If there is no match, then the miss rule's action list will be enqueued. If a miss rule is not defined and the packet is not matched, the packet will be dropped implicitly.

Eventually no more actions will be left to evaluate. When this happens, the stored action list will get its chance to execute. The process is exactly the same.

Algorithm 4 eval_match

for rule ∈ current_table.rules
    key ← rule.key
    id ← find(hashtable, key)
    if(id ≠ 0)
        insert(eval, current_table.rules$_{id}$.action_list)
insert(eval, current_table.rules$_{miss}$.action_list

7 Discussion

In this chapter, we will examine some undefined behaviors and their mitigations if any.

7.1 Goto

The most glaring case of undefined behavior is the "infinite loop", caused by a two tables jumping back and forth.

```
(pip
  (table error1 exact
    (actions
      (set (bits key 0 64))
      (match)
    ) ; actions
    (rules
      (rule (miss)
        (actions (goto (error2)))
      )
    )
  )
  (table error2 exact
    (actions
      (set (bits key 0 64))
      (match)
    ) ; actions
    (rules
      (rule (miss)
        (actions (goto (error1)))
      )
    )
  )
)
```

This is an invalid program in Pip. All tables have sequential integer identifiers, and in order for a **goto** action to be valid, the table being jumped to must have a higher integer identifier than the current table. In other words, the table being jumped to must come later in the program.

7.2 Terminators

The actions **drop**, **output**, and **match** are *terminator* actions, meaning they can only come as the last action in an action list. All action lists end in a terminator. Thus the following undefined-behavior-invoking action list is ill-formed.

```
(actions (output (reserved_port controller))
         (output (reserved_port in_port)))
```

The **match** terminator action is slightly different than the others. All exact-match tables *must* begin their matching phase, even if no rules are defined. The **match** action *must* appear at the end of the table's prep-list, or action list that sets the key. Thus, the following program is ill-formed.

```
(pip
  (table error exact
    (actions
      (set (bits key 0 64))
    ) ; actions
    (rules
      (rule (miss)
        (actions (output (reserved_port controller)))
      )
    )
  )
)
```

As is a program that matches outside of the prep-list.

```
(rule (miss)
  (actions (clear) (match)))
```

7.3 Overlapping Writes

Due to Pip's bitwise, storage-based memory model, there is no mechanism preventing a programmer from writing over only part of a buffer that was already written. Although this could possibly be useful for some advanced bit-manipulation algorithms, more often than not, reading this memory will invoke undefined behavior.

```
(actions (copy (bits meta 0 48)
               (eth.dst)
               48)
         (copy (bits packet 48 16)
               (bits packet 0 16)
               16))
```

Here, the Ethernet destination mac address (the first 48 bits of `packet`) is written to in full by the first copy action. The second copy action writes only the first 16 bits of the same header, resulting in an unknown value. Pip does not currently mitigate this error.

8 Future Work

Pip has a type system that ensures instructions can be written correctly, but does not guarantee that they produce meaningful results. For example, the programmer can copy from anywhere in the buffer to anywhere else in the buffer. Using named bitfields gives the programmer a safe alternative to bit manipulation, but the programmer is not restricted to them. Pip's storage-based model means that it lacks an object model or declaration system (beyond declaring tables). A declaration system would mitigate, and possibly eliminate memory access violations at the expense of freedom. The similar dataplane programming language, Steve, attempted to create a declaration system for packet switching.

Steve's declaration system was described in a paper by H. Nguyen [1]. Steve has a user-defined type known as a *layout* that allows programmers to dynamically define named bitfields within the packet. An example Steve layout follows:

```
layout ethernet {
  dst : uint(48);
  src : uint(48);
  type : uint(16);
}
```

Once a layout has been defined, the programmer can *extract* the individual headers and use them like variables.

```
extract ethernet.type;

if(eth.type == 0x600) {
 // ...
}
```

Until the programmer has explicitly *extracted* a named field, it is not usable. Thus, the only part of the buffer accessible to the programmer are the fields they have explicitly defined and stated they wish to access. Barring poor definition, there is no possible way for the programmer to overwrite their working buffer or access beyond its boundary [1]. Defining a similar declaration system in Pip would be a step toward fixing this issue.

As stated before, outputting to a controller program removes any provability of a Pip program. To help programmers reason about dynamic programs, we wish to create a debugger.

Bibliography

[1] H. Nguyen, "Steve - a programming language for packet processing," University of Akron, 2016.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, "Syntax definition," in *Compilers: Principles, Techniques and Tools*, Pearson Education, 2007, pp. 40–52, ISBN: 0-321-48681-1.

[3] B. C. Pierce, "Untyped arithmetic expressions," in *Types and Programming Languages*, MIT Press, 2000, pp. 20–30.

[4] F. Turbak, D. Glifford, and M. Sheldon, "Operational semantics," in *Design Concepts in Programming Languages*, MIT Press, 2008, pp. 70–127, ISBN: 978-0-262-20175-9.

[5] A. Sutton, "Libcc," 2017. [Online]. Available: `https://gitlab.com/andrew.n.sutton/cc`.

[6] ——, "Libsexpr," 2017. [Online]. Available: `https://gitlab.com/andrew.n.sutton/sexpr`.

[7] S. Goodrick and A. Sutton, "Pip," 2018. [Online]. Available: `https://github.com/asutton/pip/`.