The $P4_{16}$ Programming Language

Mihai Budiu
VMware Research
mbudiu@vmware.com

Chris Dodd
Barefoot Networks
cdodd@barefootnetworks.com

Abstract

P4 is a language for expressing how packets are processed by the data-plane of a programmable network element such as a hardware or software switch, network interface card, router or network function appliance. This document describes the most recent version of the language, $P4_{16}$, and the reference implementation of the $P4_{16}$ compiler.

1 Introduction

One of the most active areas in computer networking is Software Defined Networking (SDN) [10]. SDN separates the two core functions of a network element (e.g., router): the control-plane and the data-plane. Traditionally both these functions were implemented on the same device; SDN decouples them, and allows multiple control-plane implementations for managing each data-plane. A standard SDN example is the Open Flow protocol [14], which specifies the API between the control-plane and the data-plane.

Despite the additional flexibility brought by separating these functions, SDN still assumes that the behavior of the network data-plane is fixed. This is a significant impediment to innovation; for example, the deployment of the VXLAN protocol [18] took 4 years between the initial proposal and its commercial availability in high-speed devices.

As a reaction to this state of affairs there is a new impetus to make computer networks even more programmable by making the behavior of the *data-plane* expressible as software. Industry and academia are converging on a new domain-specific programming language called P4 (Programming Protocol-Independent Packet Processors). The P4 specification is open and public [16]. The P4 language is developed by the p4.org [2] consortium, which currently includes more than 60 organizations in the area of netwrking, cloud systems, network chip design, and academic institutions. Reference implementations for compilers, simulation and debugging tools are available with a permissive license at the GitHub P4 repository [3]. While initially P4 was designed for programming network switches, its scope has been broadened to cover a large variety of packet-processing systems (e.g., network cards, FPGAs, etc.).

1.1 P4 evolution

The original paper [7] that proposed the P4 programming language was written in July 2014. The first version of the language (currently named $P4_{14}$), including a specification, a reference implementation of a compiler, and various tools, including a simulator, were released in the fall of 2015.

Following the initial release, the p4.org consortium grew rapidly. The p4.org consortium organized a series of P4 workshops and P4 developer days. The P4 consortium assembled a formal design committee for designing the next version of the language, based on feedback from the $P4_{14}$ language users. The committee released the final specification for the newest version of the language $P4_{16}$ in May 2017; we describe the new language in Section 2. This new specification is accompanied by a reference implementation of a compiler, described in Section 3.

2 The design of $P4_{16}$

P4 is a relatively simple, statically-typed programming language, with a syntax based on C, designed to express computations on network packets. P4 programs are usually comprised of several distinct parts: parsers, that transform packets from byte arrays into structured representations of packet headers, deparsers, that convert headers into packets, and control blocks, that express the

packet processing as sequences of operations that transform headers (by inserting, deleting, or modifying the contents of headers). A table is a core primitive of P4; it is similar to a Java Map [15]. P4 provides no support for pointers, dynamic memory allocation, floating-point numbers, or recursive functions; looping constructs are only allowed within parsers. The core abstractions provided by the P4 language are listed in Figure 1.

2.1 $P4_{16}$ design goals

The design of $P4_{16}$ was informed by the experience accumulated from using $P4_{14}$.

Incremental change: Change the $P4_{14}$ language only when absolutely needed. Preserve all of the desirable features of $P4_{14}$, including all the core abstractions (headers, metadata, tables, actions, parsers — see Figure 1), and also the limited expressivity (absence of loops and pointers, etc.). Syntactic backward-compatibility is *not* a goal.

Expressivity: the new language should be strictly richer, and all $P4_{14}$ programs should be expressible in $P4_{16}$. Ideally, the conversion from $P4_{14}$ to $P4_{16}$ should be performed automatically. (Indeed, the reference compiler implementation provides this functionality.)

Support for many targets: The $P4_{14}$ specification includes a fixed target architecture, representing a switch with an ingress and an egress pipeline. A goal of $P4_{16}$ is to enable P4 to run on many other targets, which may have different architectures and capabilities (e.g., FPGAs, network cards, software switches, etc.).

Simplicity: $P4_{14}$ has a large number of keywords and many "primitive actions". In $P4_{16}$ many of these actions have been replaced by a simple and familiar expression language (e.g., `add_to_field(a, b)` becomes `a = a + b`). $P4_{16}$ is statically-typed and has a rich type system.

Modularity: $P4_{14}$ had no support for modular programs; all identifiers are global. $P4_{16}$ adds lexical scoping, local variables, and other information hiding mechanisms.

Precise definition: the semantics of many $P4_{14}$ constructs is not completely specified (e.g., arithmetic on values with different widths, runtime exceptions, how deparsing is done, where intrinsic metadata is available, the behavior of some built-in actions — such as **`drop`**, what happens when invoking a **`table`** or **`control`** multiple times, how parallel execution of statements in an action is actually performed, etc.). $P4_{16}$ attempts make the semantics as simple and explicit as possible and to reduce the number of constructs with undefined semantics to a minimum.

Extensibility: Many $P4_{14}$ language constructs were actually introduced to model features of a specific target architecture, which may not be useful or present on all targets (e.g., action selectors for tables). Such constructs have been converted into **`extern`** objects and functions, which are part of target-specific libraries rather than being part of the core language. In addition, the language provides annotations as an extensibility mechanism (similar to Java annotations [11]), which should make it easier for the language to adapt to a variety of unforeseen targets, hopefully without requiring changes to the core language syntax and semantics.

Robustness: Provide support for run-time error handling, but do not enforce run-time error handling on targets that do not want to pay the additional costs.

Compatibility: Attempt to minimize future disruptions of the language. Ideally all future versions of P4 should be be backwards-compatible with $P4_{16}$, in the sense that programs written in $P4_{16}$ today would continue to compile and execute in the same way using future language versions; in practice the goal set for $P4_{16}$ was to make the migration to future language versions as painless as possible for language users.

One non-goal was innovating in the area of programming languages; the design of $P4_{16}$ is based on very well-understood, and hopefully familiar programming language concepts (e.g., C-like syntax, sequential code, libraries, declarations, simple object-oriented programming, simple generic types).

2.2 $P4_{16}$ datatypes

The core datatype in P4 is a bistring of specified width; for example **`bit`**`<128>` specifies a bistring of 128 bits. Other base types include signed integers, booleans, and several flavors of enumeration types (similar to C **`enum`**s), including an **`error`** type for indicating error codes. Users can construct derived types such as tuples, structures (similar to C **`struct`** types), headers, arrays of headers, and unions of headers. Headers are a special case of structures which contain a hidden "valid" bit, intended to indicate whether the header was found in an input packet or should be emitted as part of an output packet. Union types are similar to C **`union`**s.

Headers describe the format (the set of fields, their ordering and sizes) of each header within a network packet.

User-defined metadata are user-defined data structures associated with each packet.

Intrinsic metadata is information provided or consumed by the target, associated with each packet (e.g., the input port where a packet has been received, or the output port where a packet has to be forwarded).

Parsers describe the permitted header sequences within received packets, how to identify those header sequences, and the headers to extract from packets. Parsers are expressed as state-machines.

Actions are code fragments that describe how packet header fields and metadata are manipulated. Actions may include parameters supplied by the control-plane at run time (actions are closures created by the control-plane and executed by the data-plane).

Tables associate user-defined keys with actions. P4 tables generalize traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, and other user-defined table types, including complex decisions depending on many fields. At runtime tables behave as match-action units [8], processing data in three steps:

- Construct lookup keys from packet fields or computed metadata,
- Perform lookup in a table populated by the control-plane, using the constructed key, and retrieving an action (including the associated parameters),
- Finally, execute the obtained action, which can modify the headers or metadata.

Control blocks are imperative programs describing the data-dependent packet processing including the data-dependent sequence of table invocations.

Deparsing is the construction of outgoing packets from the computed headers.

Extern objects are library constructs that can be manipulated by a P4 program through well-defined APIs, but whose internal behavior is hardwired (e.g., checksum units) and hence not programmable using P4.

Architecture definition: a set of declarations that describes the programmable parts of a network processing device.

Figure 1: *Core abstractions of the P4$_{16}$ programming language. The last two abstractions are new to P4$_{16}$, the others are inherited essentially unchanged from P4$_{14}$.*

2.3 P4$_{16}$ Architectures

One major change in P4$_{16}$ compared to P4$_{14}$ is in allowing programs to execute on arbitrary targets. Targets differ in the kind of processing they perform, (e.g., a switch has to forward packets, a network card has to receive or transmit packets, and a firewall has to block or allow packets), and also in their custom capabilities (e.g., ASICs may provide associative TCAM memories or custom checksum computation hardware units, while an FPGA switch may allow users to implement custom queueing disciplines). P4$_{16}$ embraces this diversity of targets and provides some language mechanisms to express it.

Figure 2 is an abstract view of how a P4$_{16}$ interacts with the data-plane of a packet-processing engine. The data-plane is a fixed-function device that provides several programmable "holes". The user writes a P4$_{16}$ program to specify the behavior of each hole. The target manufacturer describes the interfaces between the programmable blocks and the surrounding fixed-function blocks. These interfaces are target-specific. Note that the fixed-function part can be software, hardware, firmware, or a mixture.

A P4$_{16}$ architecture file is expected to contain declarations of types, constants, and a description of the control and parser blocks that the users need to implement. Here is a possible contents of the an architecture file:

```
// File arch.p4

// core.p4 contains the packet_in and
// packet_out declarations
#include <core.p4>

/// Ports are specified using 4 bits
typedef bit<4> PortId;
```

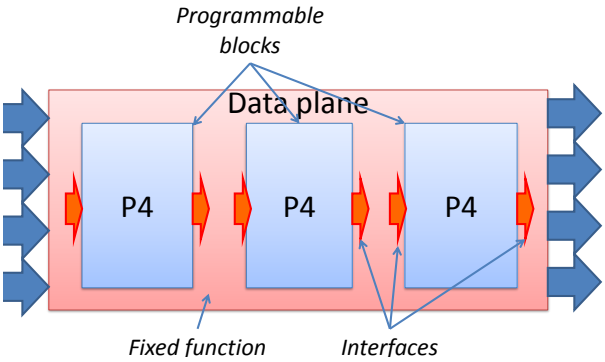

Figure 2: *Generic abstract packet-processing engine programmable in P4. The blocks labeled with P4 are programmable in P4; the surrounding block is fixed-function logic.*

```
/// Metadata accompanying an input packet
struct InControl {
  PortId input_port;
}

/** Metadata that must be computed
   for outgoing packets */
struct OutControl {
  PortId output_port;
}
// special output port values
/// Data sent to this port is dropped
const PortId DROP_PORT = 0xF;
/// Data sent to this port goes to the
/// control-plane.
const PortId CPU_OUT_PORT = 0xE;

/* Prototypes for programmable blocks */
parser Parse<H>(packet_in b,
                out H parsedHeaders);
control Pipe<H>(inout H headers,
                in error parseError,
                in InControl inCtrl,
                out OutControl outCtrl);
control Deparse<H>(in H outputHeaders,
                   packet_out b);
/// top-level package to instantiate
package Switch<H>(Parse<H> p,
                  Pipe<H> map,
                  Deparse<H> d);
```

The architecture file indicates that the user has to provide instantiations for three programmable blocks, called `Parse` (a **`parser`**), `Pipe` (a **`control`** block) and `Deparse` (another **`control`** block). The parser interface to the outside world indicates that it will receive as input a packet and will emit as output the parsed headers. The type of the parsed headers is `H`, a type variable, which indicates that the user can choose this type when writing the program.

2.4 A P4$_{16}$ Example

In this section we present a simple annotated P4 program as a concrete example. This program performs simple switching based on the IPv4 header. This is a simple program: it does not modify the IPv4 time-to-live field, and it does not recompute the IPv4 checksum.

P4 programs usually start by including the standard library `core.p4` and a library describing the target architecture.

```
#include <core.p4>
#include <arch.p4>
```

The user program continues with definitions of the types of headers that are handled in the incoming and outgoing packets; these are just type definitions, resembling C structure definitions. Headers are structures that are tightly packed, with no padding, and their fields are stored in network order.

```
typedef bit<48>  Ethernet_address;
typedef bit<32>  IPv4_address;

header Ethernet_h {
  Ethernet_address dstAddr;
  Ethernet_address srcAddr;
  bit<16>          ether_type;
}

header IPv4_h {
  bit<4>    version;
  bit<4>    ihl;
  bit<8>    diffserv;
  bit<16>   totalLen;
  bit<16>   identification;
  bit<3>    flags;
  bit<13>   frag_offset;
  bit<8>    ttl;
  bit<8>    protocol;
  bit<16>   checksum;
  IPv4_address srcAddr;
  IPv4_address dstAddr;
}

// List of all recognized headers
struct Parsed_packet {
  Ethernet_h ethernet;
  IPv4_h     ip;
}
```

When programming this target, users have to provide a parsing program; this program recognizes and extracts legal sequences of headers from incoming packets:

```
parser Parser(packet_in b,
              out Parsed_packet p) {
  state start {
    b.extract(p.ethernet);
    transition select(p.ethernet.ether_type) {
      0x0800: parse_ipv4;
      default: reject;
    }
  }
  state parse_ipv4 {
```

```
    b.extract(p.ip);
    transition accept;
  }
}
```

The parser language describes a state-machine. Each state is usually responsible for extracting one or more headers form the incoming byte stream (described by the `packet_in` data type), and transitioning to another state depending on the extracted data. In this example, the parser produces a structure p of type `Parsed_packet` as an output, parsing first an Ethernet header followed by an IPv4 header (without IPv4 options).

A $P4_{16}$ program typically contains one or more **`control`**s, which perform a sequence of match-action operations. A control contains declarations for actions and tables; a table is in fact a match-action unit, which maps a user-defined key to an action. The contents of tables is populated dynamically by the control-plane.

```
control Ctr(inout Parsed_packet headers,
       in InControl iCtr, // input port
       out OutControl oCtr) { // output port
  IPv4_address nextHop; // local variable

  action Drop_action()
  { oCtr.port = DROP_PORT; }

  action Set_nhop(IPv4_address ipv4_dest,
                  PortId port) {
    nextHop = ipv4_dest;
    oCtr.output_port = port;
  }

  /* Computes address of next IPv4 hop
  and output port based on the IPv4
  destination of the current packet. */
  table ipv4_match {
    key = { headers.ip.dstAddr: lpm; }
          // longest-prefix match
    actions = { Drop_action; Set_nhop; }
    default_action = Drop_action;
  }

  action Set_dmac(Ethernet_address dmac)
  { headers.ethernet.dstAddr = dmac; }

  /* Set the destination Ethernet
  address of the packet based on
  the next hop IP address. */
  table dmac {
    key = { nextHop: exact; }
    actions = { Drop_action; Set_dmac; }
    default_action = Drop_action;
  }

  action Set_smac(Ethernet_address smac)
  { headers.ethernet.srcAddr = smac; }

  /* Set the source ethernet address
  based on the output port. */
  table smac {
    key = { oCtr.output_port: exact; }
    actions = { Drop_action; Set_smac; }
    default_action = Drop_action;
  }

  apply { // body of the control
    // Writes result in nextHop
    ipv4_match.apply();
    if (oCtr.output_port == DROP_PORT)
      return;
    dmac.apply(nextHop);
    if (oCtr.output_port == DROP_PORT)
      return;
    smac.apply();
  }
}
```

Besides **`action`** and **`table`** declaration, a control has a body (indicated by **`apply`**) this is a loop-free imperative program that indicates the order in which tables are "applied" to packets. Our sample program uses 3 tables, as follows:

- The `ipv4_match` table uses the destination IPv4 address in the packet to find the IP address of the next network hop and the output port that connects to that hop.
- The `dmac` table uses the output port to rewrite the destination Ethernet address.
- The `smac` table uses the output port to rewrite the source Ethernet address.

Devices which rewrite the packet may need a deparser section; the deparser writes the headers of the outgoing packet.

```
control Deparser(in Parsed_packet p,
                 packet_out b) {
  apply {
    b.emit(p.ethernet);
    b.emit(p.ip);
  }
}
```

Finally, the P4 program contains a `main` declaration; this indicates how the various modules written by the user are assembled together. In this example the `Switch` symbol is declared in the included `arch.p4` file, and it expects three module arguments: a parser, a control and a deparser. The `main` declaration binds the user-specified modules to the expected parameters, instantiating the complete switch.

```
Switch(Parser(),Ctr(),Deparser()) main;
```

2.5 Limitations of $P4_{16}$

P4 is not designed for expressing arbitrary computations; it is narrowly defined for performing data-path packet processing. Surprisingly, there are even many packet-processing tasks that cannot be expressed in P4. $P4_{16}$

supports extern functions or methods; these are computational functions that are implemented outside of P4 and can be invoked from P4 programs. There is currently an effort to standardize a set of such methods; however, each P4 target platform can provide additional extern methods, e.g., to model hardware accelerators. Invoking extern methods is one way that P4 programs can perform otherwise impossible tasks.

- There is no iteration construct in P4. Loops can only be created by the parser state-machine. There is no support for recursive functions. In consequence, the work performed by a P4 program depends linearly only on the header sizes.
- There is no dynamic memory allocation in P4. Resource consumption can be statically estimated (at compile-time).
- There is are no pointers or references.
- Multicast or broadcast are achieved by means external to P4. The typical way a P4 program performs multicast is by setting a special intrinsic metadata field to a "broadcast group", triggering a mechanism that is outside of P4, which performs the required packet replication.
- P4 cannot describe queueing, scheduling or multiplexing.
- P4 is unsuitable for deep-packet inspection. In general, due to the absence of loops P4 programs cannot do any interesting processing of the packet payload.
- No support for processing packet trailers.
- All the state in a P4 program is created when a packet is received and destroyed when the processing is complete. To maintain state across different packets (e.g., per-flow counters) P4 programs must use **`extern`** methods. We expect that the standard architecture library will contain support for such persistent arrays (counters, registers, meters). Even given support for registers or counters, P4 programs cannot iterate over all counters to compute statistics.
- There is no standard way to communicate between the data-plane and the control-plane; this is usually achieved using custom `extern` methods ("learning").
- There is no support for performing packet segmentation or reassembly; thus protocols such as TCP cannot be described in P4.
- There is no support for generating new packets (e.g., an ICMP reply), only for processing existing packets.

3 P4$_{16}$ reference compiler implementation

A compiler providing a reference implementation [1] for P4$_{16}$ was developed concurrently with the language specification. The specification and implementation informed each other; in particular, the implementation uncovered many corner cases that the specification did not cover, and it also prevented the specification from making unreasonable requirements.

3.1 Compiler design goals

We set the following goals for the P4$_{16}$ reference implementation compiler:

- It should provide a solid basis to support the past, present and future versions of P4.
- It should support multiple back-ends for application-specific integrated circuits (ASICs), network-interface cards (NICs), field-programmable gate arrays (FPGAs), software switches and other targets.
- It should provide support for writing other software developmenet tools (debuggers, integrated development environments, P4 control-planes, testing, formal verification tools, etc.)
- The compiler front-end should be open-source, enabling anyone to quickly boostrap a compiler for a new architecture.
- The compiler should have an extensible architecture, making it easy to add new passes and optimizations, and to hook up new back-ends.
- The implementation should rely on modern compiler techniques (immutable intermediate representation, visitor patterns, strong type checking, etc.).
- Rely on powerful tools for testing the compiler.

3.2 Compiler architecture

P4$_{14}$ came with a wealth of available software tools: a compiler implemented in Python, a behavioral simulator, many example programs and tutorials. We considered adapting the old compiler to handle P4$_{16}$, but refactoring the old code base proved daunting, so we decided to start a new implementation using a statically-typed language.

We have settled on writing the new compiler using C++11, using garbage collection with the Boehm conservative garbage-collection engine [6]. The compiler is available using an Apache 2 open-source license. The compiler data-flow is shown in Figure 3.

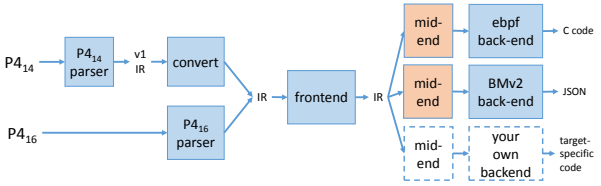

Figure 3: *P4$_{16}$ reference compiler dataflow.*

The compiler has two parsers, for the two existing P4 dialects. The P4$_{14}$ parser reads P4$_{14}$ programs and converts them into P4$_{16}$ programs. This parser is paired with a P4$_{16}$ architecture called `v1model`, which is a description written in P4$_{16}$ of the old fixed architecture implied by P4$_{14}$. When compiling a P4$_{14}$ program it is automatically converted to make use of the `v1model` architecture.

The reference compiler comes with three sample mid-end/back-end combinations. `p4test` is used for testing the compiler; it is used mostly as a source-to-source translator. This back-end is not tailored to any particular architecture. `p4c-bm2-ss` is a compiler intended to support P4$_{14}$ programs. It compiles programs written (either in P4$_{14}$ or P4$_{16}$) for `v1model` and generates code in JSON for the existing behavioral model toolkit BMv2 [5]. `p4c-ebpf` is a compiler that generates stylized C code that is intended to be compiled to extended BPF [13, 9], which can be run in the Linux kernel.

The compiler flow is divided into three parts:

Front-end The front-end does all the target-independent processing. After parsing it performs a series of syntactic and semantic checks, and it performs type-checking. It also performs a series of simplification and optimization steps. Besides traditional optimizations such as dead/unreachable-code elimination and constant folding, it also makes explicit the order of side-effects by converting statements with multiple side-effects into sequences of statements. Currently the front-end consists of roughly 25 distinct passes, some of which are run multiple times.

Mid-end We provide a library of useful passes which can be assembled by users into a desired mid-end (currently we have roughly 25 distinct passes in the library, but their number is growing). Mid-end passes are usually fairly generic, but they are driven by generally simple target-specific policies. The mid-end passes use the same intermediate representation as the front-end passes.

Back-end Back-ends are supposed to contain all target-specific code. They should do register allocation, scheduling, instruction selection, generate code in the specific language supported by the target (e.g., C, Verilog, JSON, etc.). Also, back-ends are supposed to provide support for all architectural-specific features, such as extern function and objects.

The compiler has been designed such that new mid-end/back-end combinations can be added easily. New back-ends can reside in separate respositories; to add a new back-end one merely needs to create a symbolic link in the `extensions` folder and to provide a compatible Makefile.

3.3 Intermediate representation and visitors

An important choice that had to be made upfront is whether to reuse an existing compiler infrastructure with its associated intermediate representation (e.g, LLVM [12]). Since one of the goals of P4$_{16}$ is to be executed on a very diverse variety of targets, including custom ASICS and FPGAs, we have decided that reusing an existing representation designed for traditional software targets (such as LLVM IR), would not provide sufficient benefits.

The P4$_{16}$ compiler uses a custom intermediate representation (IR) to represent the P4 program internally. The core IR representation is relatively high level, but the IR is designed so that various back-ends can extend the representation independently (e.g., adding custom representations for various target-specific resources, such as registers).

The IR data structures are immutable: once an IR representation is created, it can never be modified. A program is represented as a rooted directed acyclic graph (DAG). For example, the root node of an IR representation is always the whole program; the root node has as children a sequence of declarations, representing all declarations in the P4 program. An assignment statement IR node has two Expression IR children: the left-hand side and the right-hand side of the assignment.

The compiler architecture is based on a visitor pattern [4]: in a visitor pattern the data structure (IR) is separated from the transformations that are performed on the data structure (visitors, that modify the program). We provide a base implementation of several visitor patterns, and users create new visitors by subclassing these visitors and handling only the kinds of IR nodes that they worry about in each transformation.

The compilation is organized as a sequence of relatively simple passes; each pass is a separate visitor that consumes an immutable IR DAG and produces a new IR DAG. Using immutable representations together with garbage-collection makes it much easier to focus on the

logic of the compiler transformations, instead of memory management. It also allows one to implement easily multi-threaded compilation and back-tracking if desired (e.g., for exploring a large state space of optimizations).

The compiler front-end and mid-end use the exact same IR data structures; many passes can be seen as simplifying the IR, replacing some IR nodes with other simpler equivalent representations (e.g., convert `enums` to integers).

The IR data structures contain a lot of C++ boiler-plate code (e.g., serialization, equality testing, validation), all of which is automatically generated; the generated code implements the double-dispatch pattern needed for the visitors. IR classes are described in a very simple language which resembles a subset of C++.

The following code fragment shows a (slightly simplified) fragment that describes the IR representation of declarations (an interface implemented by several classes), the addition operation, and of an assignment statement.

```
// IR definition file (fragment)

/// declarations are objects with names
interface IDeclaration {
    /// @return an identifier
    virtual ID getName() const = 0;
}

abstract Expression {}

abstract Operation_Binary : Expression {
    Expression left;
    Expression right;
}

class Add : Operation_Binary {
    stringOp = "+";
}

abstract Statement {}

class AssignmentStatement : Statement {
    Expression left;
    Expression right;
}
```

The set of IR classes can be extended by each back-end; the users just need to add new IR definition files to the Makefiles.

The IR representation is strongly-typed (e.g., the children of an assignment statement must have type `Expression`); this makes it harder to create incorrect IR representations. The IR representation can be serialized to/from to a textual JSON representation.

All or the passes that we supply for the front-end and mid-end maintain the invariant that program IR is always convertible back to a $P4_{16}$ program. This feature significantly simplifies, debugging, testing and learning the IR:

- One does not need to understand the IR to discover compiler bugs; one can dump the internal representation as a P4 program at any time during compilation.
- We use the P4 representation to validate the compiler, by saving the representation at key points during compilation, and by recompiling the P4 code that is output.
- Optionally, the dumped P4 program can contain as comments the actual IR data structures annotating the P4 code. This makes it much easier to learn the IR.

3.4 Testing

Program testing is hard; testing a compiler is doubly hard, because a compiler consumes and outputs programs. We have built the following facilities to simplify testing the compiler:

- We maintain a large and growing collection of sample $P4_{14}$ and $P4_{16}$ input programs that are recompiled on every commit.
- For each of the sample programs we save the $P4_{16}$ representation of the program at three steps during compilation. The tests expect that these reference outputs do not change. Occasionally adding new optimizations will change the reference outputs; the changed outputs have to be manually vetted by an expert before being changed.
- The compiler re-compiles the programs that it outputs.
- For P4 programs written for the `v1model` (both $P4_{14}$ and $P4_{16}$ programs) we provide a simple language that can specify the contents of the tables, input packets and expected output packets. These programs are compiled by `p4c-bm2-ss` and executed using the BMv2 simulator. This provides end-to-end testing of the compiler.

In the future we hope to also build verification tools based on translation validation [17], relying on the compiler's ability to generate P4 sources from the IR.

4 Conclusions

While P4 is a relatively simple language, it needs to execute into a very complex *environment* (as proven by the fast-paced evolution of $P4_{14}$: at some point new features were added to the $P4_{14}$ specification every couple of weeks).

The language designers realized that this situation is not sustainable: a language has to be relatively stable to encourage adoption, investment and portability. So the $P4_{16}$ language was designed to try to reconcile many conflicting trade-offs: fast evolution, support for a very diverse set of targets, while providing a stable base and portability. This was done by separating $P4_{14}$ into a fixed language core, which is supposed to be stable, and a set of target-specific libraries, which can evolve quickly. $P4_{16}$ contains other extensibility mechanisms, such as the ability to specify architectures, and target-specific annotations.

Whether the $P4_{16}$ design is successful it is too early to tell; this will be validated by the number of available implementations and especially by the number of data-plane applications that people will develop. To some degree the goals of flexibility and extensibility seem to be justified by the fact that at least 5 different (open-source and proprietary) back-ends have been already created for the reference compiler, supporting a large variety of very different targets (FPGAs, ASICs, software devices, simulators). Our goal is for $P4_{16}$ to enable the same kind of programmability for network data-planes as the CUDA language did for graphics cards.

5 Acknowledgements

This document represents the work of a large (and growing) community of contributors over the span of 2 1/2 years, both for the language design process and for the reference compiler implementation. The following is an incomplete list of contributors that have made significant contributions, listed in alphabetical order: Leo Alterman, Michael Attig, Antonin Bas, Gordon Brebner, Călin Caşcaval, Andy Fingerhut, Nate Foster, Seth Fowler, Vladimir Gurevich, Robert Halstead, Andy Keep, Changhoon Kim, Chaitanya Kodeboyina, Nick McKeown, Peter Newman, Edwin Peer, Ben Pfaff, Cole Schlesinger, Anirudh Sivaraman, Steffen Smolka, Dan Talayco, Johann Tönsing, Han Wang.

References

[1] P4-16 compiler reference implementation. `https://github.com/p4lang/p4c`. Retrieved May 2017.

[2] P4 Consortium. `http://p4.org`.

[3] P4 github repository. `https://github.com/p4lang`. Retrieved May 2017.

[4] Visitor pattern. `https://en.wikipedia.org/wiki/Visitor_pattern`, Retrieved May 2017.

[5] Antonin Bas. The P4 behavioral model version 2. `https://github.com/p4lang/behavioral-model`, Retrieved May 2017.

[6] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. Programming protocol-independent packet processors. In *ACM SIGCOMM Computer Communications Review (CCR)*, volume 44, July 2014.

[8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. pages 99–110. ACM, 2013.

[9] Jonathan Corbet. Attaching eBPF programs to sockets. `https://en.wikipedia.org/wiki/LWN.net`, December 2014.

[10] Evangelos Haleplidis, Kostas Pentikousis, Spyros Denazis, Jamal Hadi Salim, David Meyer, and Odysseas Koufopavlou. Software-defined networking (SDN): Layers and architecture terminology. `https://tools.ietf.org/html/rfc7426`, January 2015. RFC 7426.

[11] M. M. Islam. Java annotations: An introduction. `http://www.developer.com/java/other/article.php/3556176`, October 2005.

[12] Chris Lattner and Vikram Adve. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, September 2004.

[13] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Conference*, January 1993.

[14] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[15] Sun Microsystems. Interface Map¡K, V¿. https://docs.oracle.com/javase/7/docs/api/java/util/Map.html, 1993.

[16] P4.org. P4-16 language specification. https://github.com/p4lang/p4-spec/tree/master/p4-16/spec, May 2017.

[17] Amir Pnueli, M. Siegel, , and Eli Singerman. Translation validation. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 151–166, 1998.

[18] M. Mahalingam Storvisor, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual eXtensible Local Area Network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks. https://tools.ietf.org/html/rfc7348, August 2014.