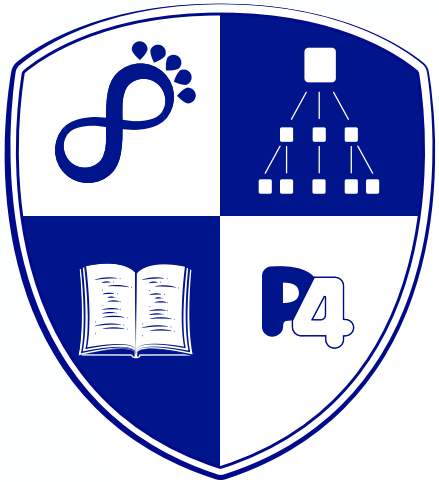

BAREFOOT ACADEMY

Some Ideas for $P4_{16}$

June 22, 2020

Vladimir Gurevich
Principal Engineer. Director, Barefoot Academy.

Agenda

- Composability in P4
- Automatic API generation
- Miscellaneous

Making P4 More Modular

Motivation

- **Goals**
  - Better P4 Source Code Reuse
    - Ability to have standardized, reusable source code modules
    - Ability to extend P4 programs without the need to rewrite the “base”
    - Ability to create many variants of the same program
  - Eliminate the need to use preprocessors as a “poor-man’s module system”
  - Get people thinking
- **Non-Goals**
  - Separate compilation of P4 modules
  - Namespaces
  - Offer specific solutions

Adding a new protocol to an existing base

- **Problem:**
  - Most published P4 programs cannot be used in production networks
- **Solution:**
  - Use ”base” code to handle standard L2/L3 protocols if possible
  - Add a new protocol on top (as L3, L4, ... L7 extension)
- **Challenges:**
  - Modifying the parser
  - Amending the controls
  - Modifying the deparser

Parser Modifications

```
enum ip_proto_t {
    ICMP = 1, IGMP = 2, TCP = 6, UDP=17
#ifdef MY_PROTOCOL_SUPPORT
    , MY_PROTOCOL = 1234
#endif
}

struct ingress_headers_t {
    . . .
#ifdef MY_PROTOCOL_SUPPORT
    my_protocol_h    my_protocol;
#endif
}

parser IngressParser(packet_in        pkt,
    out ingress_headers_t             hdr,
    out ingress_metadata_t            meta,
    out ingress_intrinsic_metadata_t  ig_intr_md)
{
     state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        meta.l4_lookup = pkt.lookahead<l4_lookup_t>();

        transition select(hdr.ipv4.frag_offset,
                          hdr.ipv4.protocol,
                          hdr.ipv4.ihl) {
            ( 0, ip_proto_t.ICMP, 5 ) : parse_icmp;
            ( 0, ip_proto_t.IGMP, 5 ) : parse_igmp;
            ( 0, ip_proto_t.TCP,  5 ) : parse_tcp;
            ( 0, ip_proto_t.UDP,  5 ) : parse_udp;
#ifdef MY_PROTOCOL_SUPPORT
            ( 0, ip_proto_t.MY_PROTOCOL, 5) : parse_my_protocol;
#endif
            ( 0, _             ,  5 ) : parse_first_fragment;
            default : accept;
        }
    }
```

- **Relatively easy with CPP, but**
  - The base code requires extensive modifications
  - Modifications are spread around and cannot be easily consolidated
  - The post-processed code is difficult to read
    - ... once additions start to exceed one line
- **Required functionality:**
  - Ability to add a new enum value
  - Ability to add a new struct member
  - Ability to add a new parser state to the existing parser
  - Ability to add a new transition to a select() statement
    - ... or a group of related select() statements
    - ... **somewhere** among the existing transitions
- **Ideally all changes can be kept in <u>one</u> place**
- **Ideally the base code will provide a list of what needs or can be modified**

Control Modifications (1)

```
action set_port_properties(vlan_id_t default_vlan,
                           pcp_t     default_priority
                     #ifdef MY_PROTOCOL_SUPPORT
                          , bool     allow_my_protocol
                     #endif
) {
     meta.default_vlan      = default_vlan;
     meta.default_priority  = default_priority;
#ifdef MY_PROTOCOL_SUPPORT
     meta.allow_my_protocol = allow_my_protocol;
#endif
}

table l2_station {
    key = {
        ig_intr_md.ingress_port : ternary;
        hdr.ethernet.dst_addr   : ternary;
    }
    actions = {
        process_ipv4; process_ipv6; process_mpls;
#ifdef MY_PROTOCOL_SUPPORT
        process_my_protocol;
#endif
    }
}
```

- **Relatively easy with CPP, but**
  - The base code requires extensive modifications
  - Modifications are spread around and cannot be easily consolidated
  - The post-processed code is difficult to read
    - ... once additions start to exceed one line
- **Required functionality:**
  - Ability to add a new parameter to an existing action
  - Ability to add code to an existing action
    - Before or after existing action code
  - Ability to add a new action to an existing table

Control Modifications (2)

```
table ipv4_acl {
    key = {
        ig_intr_md.ingress_port : ternary;
        hdr.ip.dst_addr         : ternary;
        . . .
#ifdef MY_PROTOCOL_SUPPORT
        hdr.my_protocol.field_1 : ternary;
#endif
    }
    actions = {
        drop; copy_to_cpu; mirror; ...
    }
}

apply {
    . . .
    switch(l2_station.apply().action_run) {
        process_ipv4 : { ipv4_control.apply(hdr, meta, ...); }
        process_ipv6 : { ipv6_control.apply(hdr, meta, ...); }
        process_ipv4 : { ipv4_control.apply(hdr, meta, ...); }
#ifdef MY_PROTOCOL_SUPPORT
        process_my_protocol : {
                  my_protocol_control.apply(hdr, meta, ...);
        }
#endif
        default : { process_l2.apply(); }
    }
```

- **Relatively easy with CPP, but**
  - The base code requires extensive modifications
  - Modifications are spread around and cannot be easily consolidated
  - The post-processed code is difficult to read
    - ... once additions start to exceed one line
- **Required functionality:**
  - Ability to add a new parameter to an existing action
  - Ability to add code to an existing action
    - Before or after existing action code
  - Ability to add a new action to an existing table
  - Ability to add another field to the key of an existing table
  - Ability to add code to the existing control in some specific placed
    - Ability to redefine controls, defined in the base code

Additional Flexibility

```
control calc_ipv6_hash(
    in    my_ingress_headers_t   hdr,
    in    my_ingress_metadata_t  meta,
    out   bit<32>                hash)
    (bit<32> poly)
{
    CRCPolynomial<bit<32>>(
        coeff    = poly,
        reversed = true,
        msb      = false,
        extended = false,
        init     = 0xFFFFFFFF,
        xor      = 0xFFFFFFFF) poly;
    Hash<bit<32>>(HashAlgorithm_t.CUSTOM, poly) hash_algo;

    action do_hash() {
        hash = hash_algo.get({
                hdr.ipv6.src_addr,
                hdr.ipv6.dst_addr,
                hdr.ipv6.next_hdr,
                meta.l4_lookup.word_1,
                meta.l4_lookup.word_2
            });
    }

    apply {
        do_hash();
    }
}
```

This is not something that can be easily abstracted with typedef

- **Problem:**
  - Non-top-level control definitions cannot be generic
- **Required functionality**
  - Allow non-top-level controls and parsers to be generic
    - Can somewhat be mitigated with **typedef**

Automatic API Generation

Motivational Example. Ascribing an API to an extern

```
enum e1_t { e1_value1, e1_value2 }
enum e2_t { e2_value1, e2_value2, e2_value3 }

extern ext<S> {
    ext(bit<32> param1, e1_t param2);
    e2_t method1(in S     param1,
                 in e2_t  param2);
    e2_t method1(in S param1);
}

enum PSA_MeterType_t  { PACKETS, BYTES }
enum PSA_MeterColor_t { RED, GREEN, YELLOW }

extern Meter<S> {
    Meter(bit<32> n_meters, PSA_MeterType_t type);
    PSA_MeterColor_t execute(in S index,
                             in PSA_MeterColor_t color);
    PSA_MeterColor_t execute(in S index);
}

/*
    @ControlPlaneAPI {
        reset(in MeterColor_t color);
        setParams(in S index, in MeterConfig config);
        getParams(in S index, out MeterConfig config);
    }
*/
```

- **Problem:**
  - Data Plane and Control Plane APIs are completely orthogonal and cannot be derived from each other
  - What is the Control Plane API for the extern “ext”?
  - How do we know what is the Control Plane API for the extern “Meter”?
    - Can the comment below be related to the code?
      - Can we say what MeterConfig is?
      - What language is this written in?
- **Solution:**
  - We need to have a separate (sub)language to describe the control-plane interface to any object
  - This description must be a part of the architecture definition (not the user program)

Meters are special objects in P4 Runtime

P4 Code

```
enum PSA_MeterType_t  { PACKETS, BYTES }
enum PSA_MeterColor_t { RED, GREEN, YELLOW }

extern Meter<S> {
    Meter(bit<32> n_meters, PSA_MeterType_t type);
    PSA_MeterColor_t execute(in S index,
                             in PSA_MeterColor_t color);
    PSA_MeterColor_t execute(in S index);
}

/*
    @ControlPlaneAPI {
        reset(in MeterColor_t color);
        setParams(in S index, in MeterConfig config);
        getParams(in S index, out MeterConfig config);
    }
*/
```

reset/setParams/getParams are still nowhere to be found in .proto file

p4runtime.proto

```
message Entity {
 oneof entity {
     ExternEntry extern_entry = 1;
     TableEntry table_entry = 2;
     ActionProfileMember action_profile_member = 3;
     ActionProfileGroup action_profile_group = 4;
     MeterEntry meter_entry = 5;
     DirectMeterEntry direct_meter_entry = 6;
     CounterEntry counter_entry = 7;
     DirectCounterEntry direct_counter_entry = 8;
 }
}

message MeterEntry {
 uint32 meter_id = 1;
 int64 index = 2;
 MeterConfig config = 3;
}

message MeterConfig {
 int64 cir = 1;
 int64 cburst = 2;
 int64 pir = 3;
 int64 pburst = 4;
}
```

These didn't come from P4 code

Why is this important?

- **Either we find a generic solution or P4 Runtime will:**
  - Either remain tied to v1model or PSA
  - Get polluted with tons of incompatible extensions
- **Once we know how to ascribe APIs to arbitrary externs...**
- **We can create APIs for Fixed Function Components too**
  - Packet Replication (Multicast) Engine
  - Traffic Manager (Buffering/Scheduling/Queueing) Engine
  - Ports
  - ...
- **Just add corresponding extern definitions and API descriptions to the architecture file**

Miscellaneous

Special Statements for Language/Architecture

```
/* Replace with the indication that this is P4_16.
 * Make sure that there is a standard way to enforce
 * the required language version */
#include <core.p4>

/* Replace with the indication that this is v1model
 * architecture. Make sure that there is a standard way
 * to enforce the required architecture version */
#include <v1model.p4>

/* Replace with a first-class language construct */
control c() {
#if P4_16_VERSION >= 0x010201
    my_struct_t s = { ... };
#else
    /* Do not forget to change this when you change
     * the struct definition! */
    my_struct_t s = { 0, 0, 0, 0 };
#endif

#if PSA_VERSION >= 0x010101
    PSA_NewCoolExtern(...) cool_extern;
#else
    #error "This program requires PSA version 1.1.1 or later"
#endif
```

- **Problem:**
  - Currently, all P4 programs use the same extension (.p4)
  - It is difficult for the tools to figure out:
    - The language dialect being used
      - $P4_{14}$ or $P4_{16}$
      - The actual version of the language (esp. for P4_16)
    - The architecture
      - The required version of the architecture
  - Writing programs that can be compiled across multiple versions of language/architecture requires CPP
- **Current solution:**
  - Run the preprocessor
    - Use heuristics to determine language/architecture
    - Use preprocessor variables (if defined) to deal with versioning
- **Proposal:**
  - Define special statements instead

Better Naming Control

```
control A() {
    @name(".a1")
    action a1() { }

    @name(".t")
    table t {
    }
}

control B() {
    A() a;
    apply {
        a.apply();
    }
}

control Ingress() {
    B() b;
    apply {
        b.apply();
    }
}
```

In common practice, **.t** or **.Ingress.t** are preferred over **.Ingress.b.a.t**

- **Problem:**
  - Currently, the names of P4 objects reflect full hierarchy of the controls
  - Most practical P4 programs are peppered with unnecessary **@name**() annotations
    - Especially annoying for actions, since actions can live only in two places:
      - top-level
      - the same namespace as the table (originally)
- **Required Functionality**
  - Global way to control names better:
    - Pull them into top-level
    - Pull them into top-level controls/parsers

Table Key Fields

```
table t {
    key = {
        hdr.ipv4.isValid() : ternary;
        hdr.ipv4.dst_addr  : ternary;
        ig_intr_md.ingress_port[6:0] : ternary
                                       @name("ingress_port");
    }
    ...
}

/* Proposal */
table t {
    key = {
        ipv4_valid = hdr.ipv4.isValid()           : ternary;
                     hdr.ipv4.dst_addr            : ternary;
        pipe_port  = ig_intr_md.ingress_port[6:0] : ternary;
    }
    ...
}
```

- **Problem:**
  - Table key fields have long, structured names
  - When we use expressions, @name() annotation is almost always required
  - Action names are always simple
- **Proposal (credits: Steffen Smolka):**
  - Allow for simple names
  - For a field/variable use the last portion of the name
  - Require “name =” for expressions and in ambiguous cases

Structs as Keys and Action Parameters

```
struct l2_address_t {
#ifdef VIRTUAL_L2_NETWORKS
  #ifdef VLAN_IS_BD
    vlan_id_t       vid;
  #else
    bd_id_t         bd;
  #endif
#endif
    mac_address_t   mac_addr;
}

struct ingress_metadata_t {
    ...
    user_l2_meta_t   um_l2;
    user_l3_meta_t   um_l3;
    l2_address_t     l2_dst_addr;
}

/* This allows for better program composability */
action set_l2_properties(user_l2_meta_t l2_props) {
    meta.um_l2 = l2_props;
}

table dmac {
   key  = {
      meta.l2_dst_addr : exact;
   }
   ....
```

- **Problem:**
  - Allow adding new fields to a key or action
- **One of the solutions:**
  - Allow structs
- **Challenges:**
  - Lack of P4 Runtime Support

Thank you