Towards P4 v1.2
- a proposal -

Mihai Budiu

Barefoot Networks

February 2016

Goals of 1.2

- Simplify and clean-up P4
- Provide a “stable” language version
  - I.e., we work very hard to keep the language **backwards compatible** from 1.2 onwards
- Provide a precise specification of the language
  - Including semantics of all constructs
- Provide a reference implementation of the language
  - Compiler front-end
  - Example programs
  - Behavioral simulator
- Address feedback received on P4 v1.0 and v1.1

P4 v1.2 vs 1.1

- An incremental evolution of P4 v1.1
- Same abstraction level
- Same core constructs
  - Parsers, control, match/action tables, actions, headers, metadata
- Same computational restrictions
  - No unbounded loops, no FP, no pointers, no recursion, constant work per header byte
- Simplified and clarified
- Avoid inventing new language constructs
  - Reuse well-understood tools and techniques as much as possible
- Prepare language for future evolution through *growth*
  - Architectural features caused most of P4's growth

Language clean-up

- Break language into three parts
  - Core language (part of the language spec)
    - Packet processing language
    - Language constructs to describe architectures
  - Standard library (e.g., common to *all* architectures)
  - A standard architecture spec
    - Prototypes for architectural blocks and intrinsic metadata
    - Library with extern blocks declarations (e.g., checksums)
- Write a specification for the control/data-plane API
- Libraries and architectures are written in P4
- These separate specifications evolve independently
- Architecture evolution becomes much easier

Desirable P4 v1.2 features

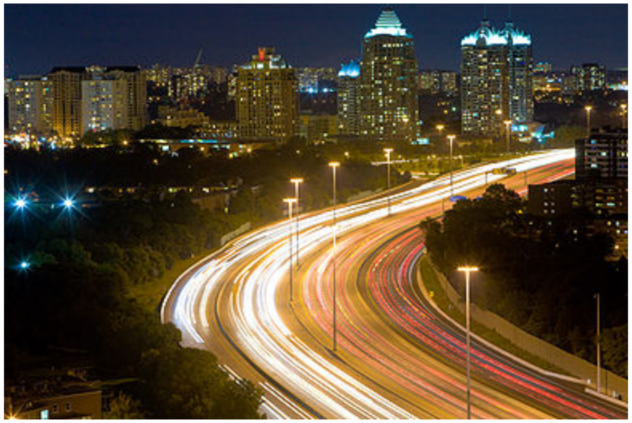
- Strong static typing
- Simpler syntax
- Few undefined behaviors
- No runtime exceptions/traps
- Explicit deparser specification
- Clear evaluation results (“declarative” => “deterministic”)
- Lexical scoping
- Support for writing modular programs
- Support for error handling
- Parameterization (e.g., “how many bits to specify an output port?”)
- Compile-time resource allocation (e.g., checksum units, tables, etc.)
- Simple extensibility hooks (e.g., Java-like annotations)

Details for some proposed constructs

Architecture specification

A proposal for architecture specification was given in December 2015

- That presentation is included as an appendix
- The architectural specification language included the following features:
    - struct/header types
    - parsers/control/packages architectural blocks
    - prototypes for architectural blocks
    - generic types (templates)
    - parameterized architectural blocks
    - separation of declaration vs. instantiation



Interaction with architecture

- Intrinsic metadata
  - Action occurs at the “end” of the pipeline
- Extern object method invocations
  - Action occurs instantly
- No “delayed” execution
  - E.g., drop, field_list_calculation, generate_digest
  - Order of delayed executions was unspecified
  - Order of side-effects and delayed executions was unspecified
- The meaning of a P4 program should be unambiguous

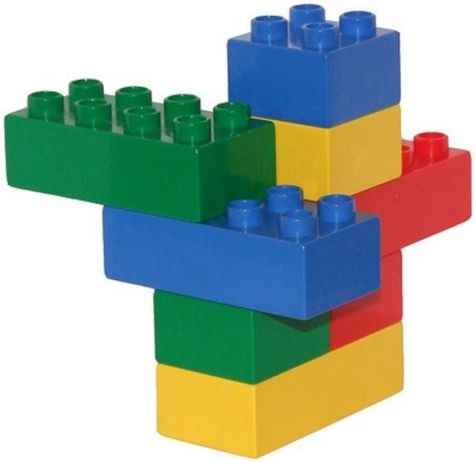
Moving constructs from P4 to libraries

- Custom primitive actions declarations
- field_list_calculation (e.g., checksums, modify_field_with_hash_based_offset)
- parser_value_set
- generate_digest
- cloning, recirculation, resubmission, mirroring
- Counters, meters, registers
- Action profiles
- Saturated types
- In general, all constructs which “look” non-portable across all architectures

Explicit packet (in standard library)

```
extern packet_in {
        void extract<T>(out T hdr);
        void extract<T>(out T varSizeHeader, in bit<32> size);
        T lookahead<T>();
}
extern packet_out {
     void emit<T>(in T hdr);
}
parser prs(packet_in p, Headers h) {
    p.extract(h.eth);
}
```

Deparsers



- In P4 v1.1
  - Sometimes impossible to infer
  - Users have no control
  - Hacks for creating fabric headers
- In P4 v1.2
  - Just another control block
  - Should clearly specify sequence of actions (emit, checksums)

```
control deparser(in headers h, packet_out p)
{
  Checksum16() ck;
  apply {
      ck.clear();
      h.ip.hdrChecksum = 0;
      ck.update(h.ip);
      p.emit(h.ethernet);
      h.ip.hdrChecksum = ck.get();
      p.emit(h.ip);
   }
}
```

Parameterization support

- Writing portable and modular programs
- **typedef**
  - E.g., **typedef** bit<8> Port_t;
- **enum**
  - E.g., **enum** ChecksumType { crc16, crc32 }
- constant declarations
  - **const** Port_t CPU_PORT = 16;
- Generics (templates)
  - E.g., parser<**H**>(packet_in p, out H headers)
- Constructors
- (See also the architectural description proposal)

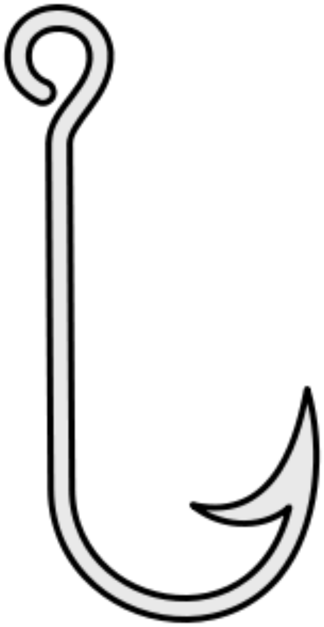
Simpler syntax

- modify_field, set_metadata => assignment statements
  - **modify_field**(a, b) => a = b
  - **set_metadata**(a, b) => a = b
- Add a few useful operators: masking, concatenation, bit selection, mux, range
- Convert keywords to methods or fields
  - **valid(a**) => a.valid
  - **add_header**(a) => a.setValid(true);
  - **remove_header**(a) => a.setValid(false);
  - **copy_header**(a, b) => a = b
  - hs[**last**] => hs.last
  - **push**(hs, 2) => hs.push_front(2)

Richer type system

- **enum**
- **error**
- **header/struct**
- **header_union**
- Stacks of unions => option parsing
- Typed architecture blocks
  - parsers, control blocks, packages
  - (See also the architectural description proposal)

Extensibility hooks



- @annotation(expression)
- Allows for some language evolution without spec changes
- Pragma-like
- Typed
- Apply to specific language elements
- Similar to Java @annotations and C# [Attributes]
- Some annotations could become part of standard

Scoping

- Create lexical scopes
- Remove global variables
- Introduce local variables and parameters
  - Clarifies scope of intrinsic metadata
    (See also the architectural description proposal)
  - Enables modular programs
- Declarations must precede uses (except parser states)

Error handling

- Accommodate various architectural constraints
  - (E.g., encoding of error codes)
- Add an **error** type (special enum-like type):
  error { IncorrectVersion, HeaderTooShort }
- Parser exceptions => “reject” parser state
- Introduce an “assert” method, usable in parsers
  assert(h.ip.version == 4, IncorrectVersion);
  - Assertion failure sets error code and transitions to the reject state
- Expose errors explicitly to control blocks
  control ingress(**in error** parser_error, …)

Flexible control-flow

- Control blocks:
  - Add a **return** statement
  - Add an **exit** statement
- Parsers
  - rename “**return**” to “**transition**”

To be continued…

- We will produce the following:
  - Draft design with all of these features
  - A “migration guide” mapping P4 v1.1 constructs to P4 v1.2
  - Example programs and program fragments
  - A written P4 v1.2 specification draft

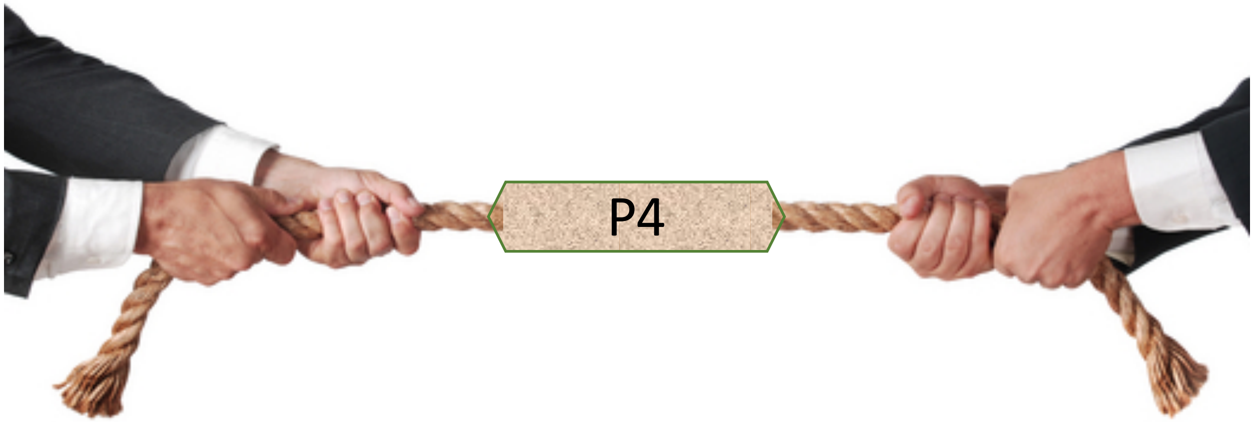
Appendix

- The following slides include for reference the presentation from December 2015 on a proposal for describing architectures in P4

Abstracting switch architectures
- a proposal -

November 30, 2015

The P4 tension



Universal

Customizable

P4 v1:
Fixed Abstract Forwarding Model

1.1 The P4 Abstract Model *1 INTRODUCTION*

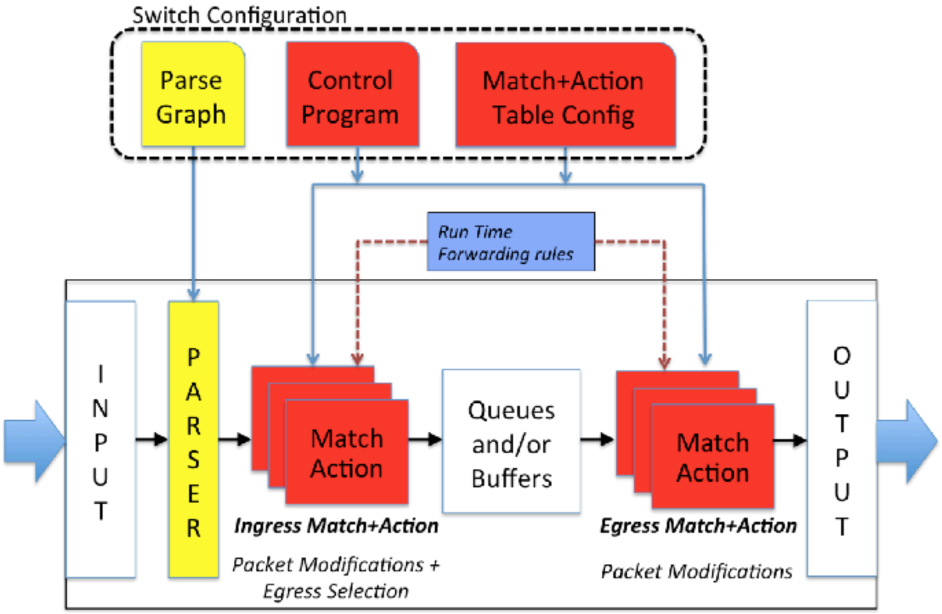
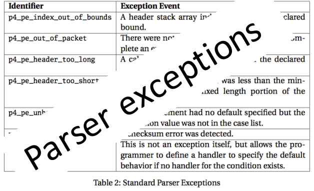


Figure 1: Abstract Forwarding Model

P4 v1: Details



Parser exceptions

Identifier	Exception Event
p4_pe_index_out_of_bounds	A header stack array in[illegible] clared bound.
p4_pe_out_of_packet	There were no[illegible] om-plete an e[illegible]
p4_pe_header_too_long	A ca[illegible] the declared
p4_pe_header_too_short	[illegible] was less than the min-[illegible] ixed length portion of the
p4_pe_unh[illegible]	[illegible] ement had no default specified but the [illegible] ion value was not in the case list.
[illegible]	[illegible] checksum error was detected.
[illegible]	This is not an exception itself, but allows the programmer to define a handler to specify the default behavior if no handler for the condition exists.

Table 2: Standard Parser Exceptions

Action profiles

Mirroring

field_list_calculation

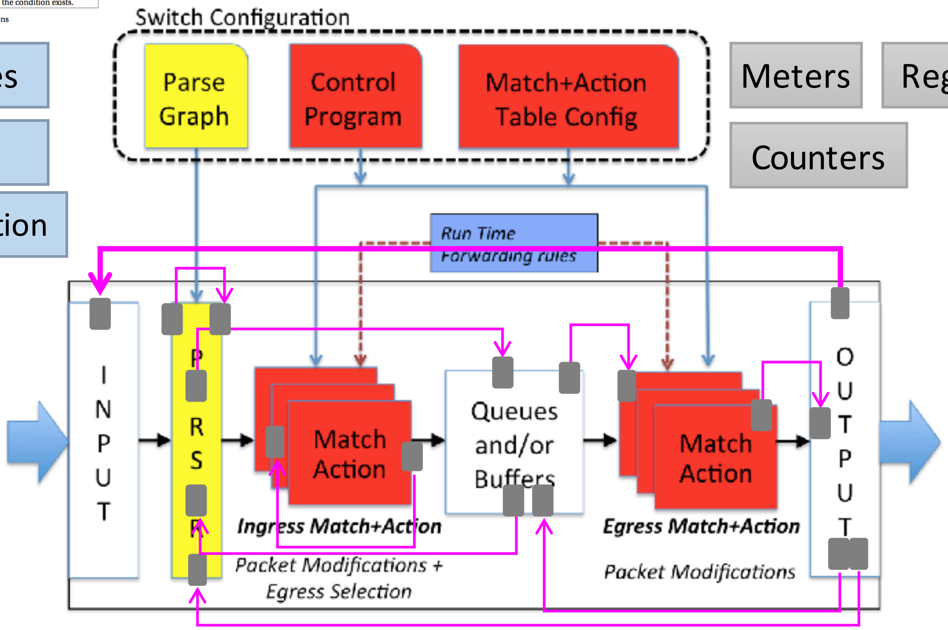
xor16

csum16

crc16

crc32

programmable



Meters

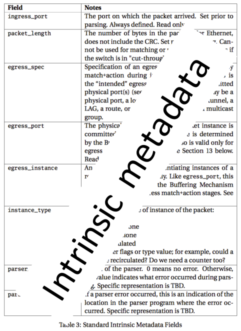
Registers

Counters

LPM

Exact

Ternary

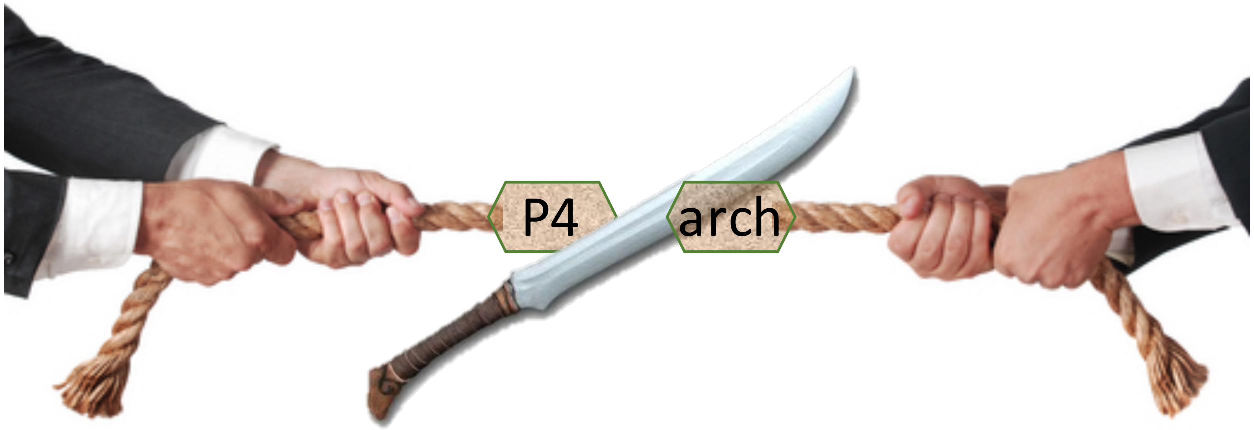


Intrinsic metadata

Field	Notes
ingress_port	The port on which the packet arrived. Set prior to parsing. Always defined. Read only
packet_length	The number of bytes in the pac[illegible] or Ethernet, does not include the CRC. Set [illegible]. Cannot be used for matching or [illegible] if the switch is in "cut-throug[illegible]
egress_spec	Specification of an egres[illegible] y match+action during i[illegible] is the "intended" egress[illegible] itted physical port(s) (se[illegible] ay be a physical port, a lo[illegible] unnel, a LAG, a route, or [illegible] multicast group.
egress_port	The physica[illegible] ket instance is committed[illegible] e is determined by the B[illegible] so is valid only for egress [illegible] e Section 13 below. Read[illegible]
egress_instance	An [illegible] ntiating instances of a r[illegible] ly. Like egress_port, this [illegible] the Buffering Mechanism [illegible] ess match+action stages. See
instance_type	[illegible] of instance of the packet: [illegible] one [illegible] one [illegible] ulated [illegible] er flags or type value; for example, could a [illegible] recirculated? Do we need a counter too?
parser[illegible]	[illegible] of the parser. 0 means no error. Otherwise, [illegible] value indicates what error occurred during pars[illegible] g. Specific representation is TBD.
par[illegible]	[illegible] f a parser error occurred, this is an indication of the location in the parser program where the error occurred. Specific representation is TBD.

Table 3: Standard Intrinsic Metadata Fields

Divide and conquer

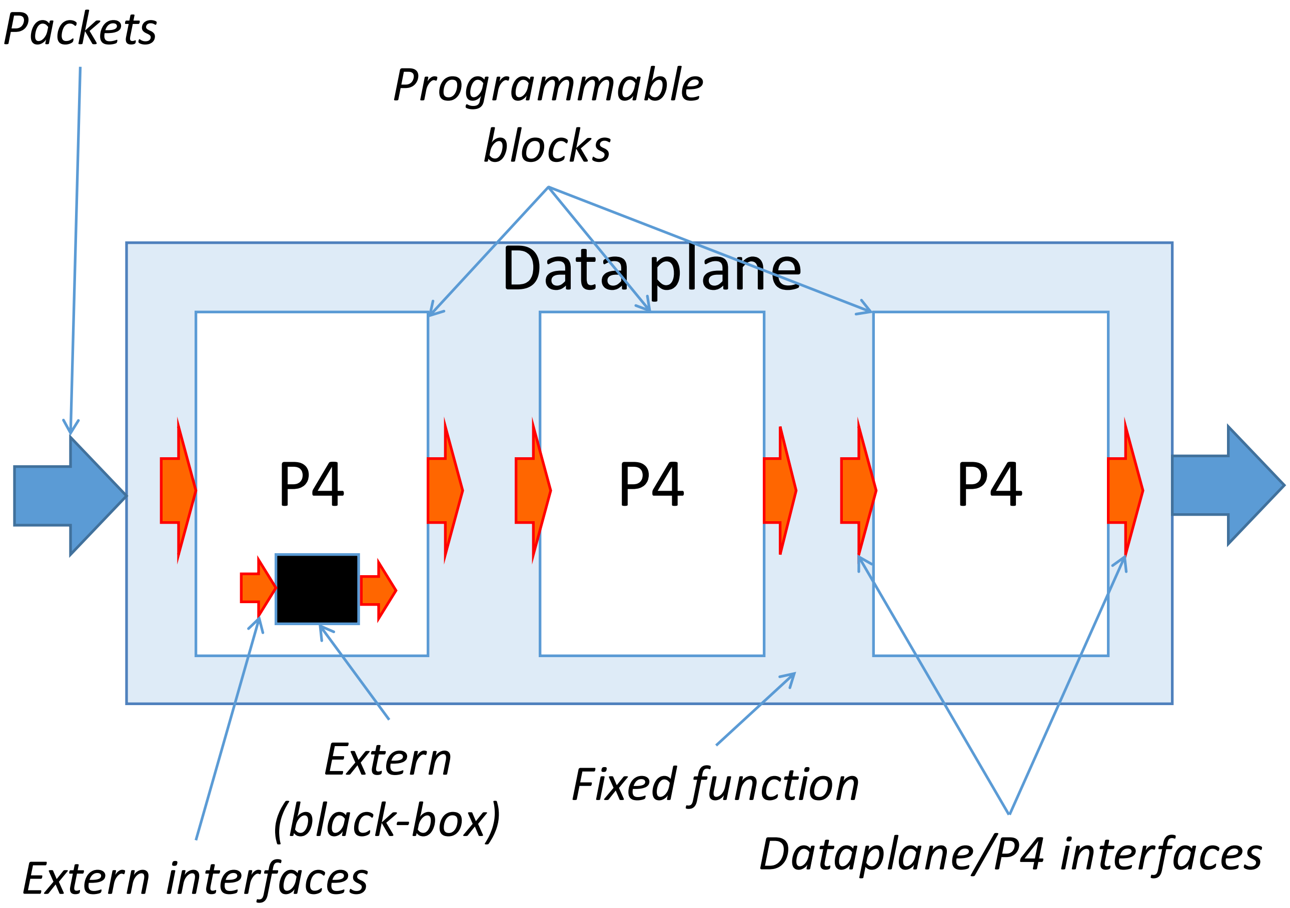


- Separate language definition from architecture definition
- Evolve them independently

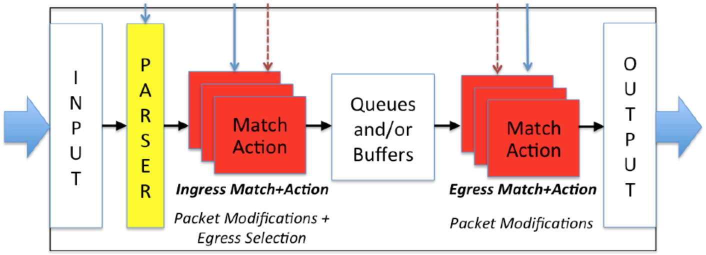
Specifying architectures

- A device model describes what parts of a forwarding device can be programmed in P4.
- Each manufacturer can publish custom device models.
- The community defines a standard switch model for portability.
- Even if without custom switch models, this approach is useful, because it decouples the language evolution from the model evolution: new versions of the standard switch model do not require changing the language.

Generic Programmable Dataplane Model



P4 Support for multiple architectures



INPUT

PARSER

Match Action

Ingress Match+Action

Packet Modifications + Egress Selection

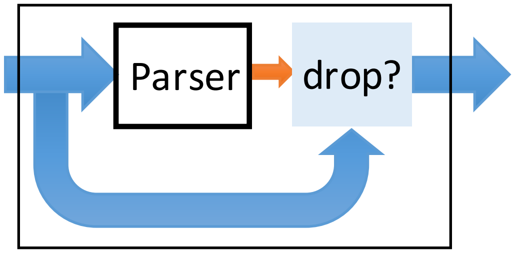
Queues and/or Buffers

Match Action

Egress Match+Action

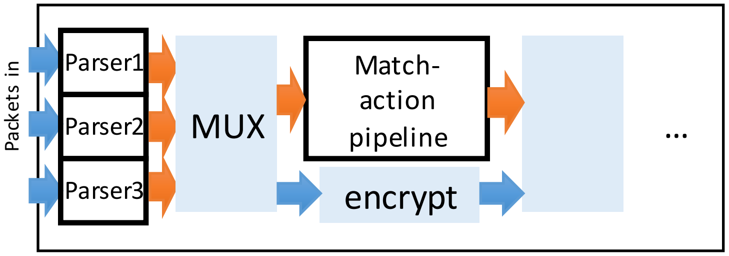
Packet Modifications

OUTPUT



Parser

drop?



Packets in

Parser1

Parser2

Parser3

MUX

Match-action pipeline

encrypt

…

Inventing new language constructs

- Don’t
- You will get them wrong
- Reuse constructs from other languages

- How would I do this in Java/C++?

Switch architecture in C++

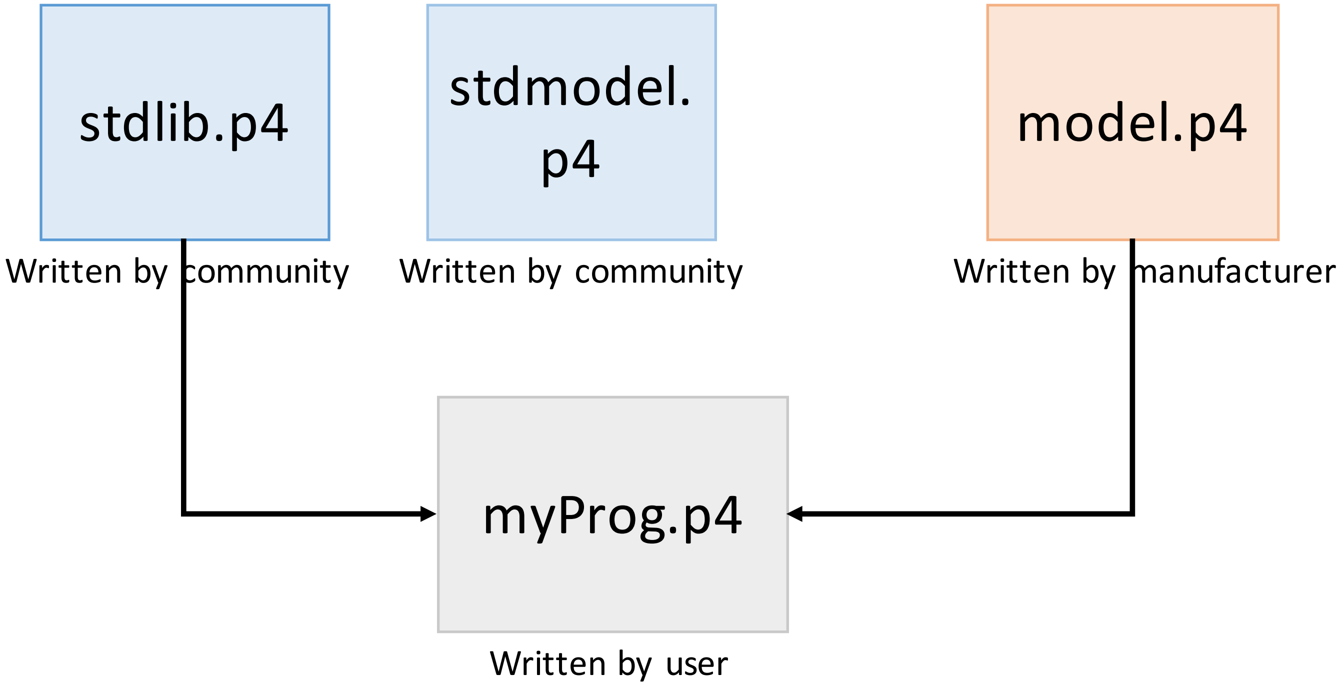
```
// switch.hpp: written by manufacturer
struct MetaIn { int inputPort; }
struct MetaOut {
    int outputPort;
    bool drop;
}

template<class T> class switch
{
    virtual void parser(const packet &p, T& headers)=0;
    virtual void control(T& headers,
                        const MetaIn &in, MetaOut& out)=0;
}
```

target-defined metadata

user-defined metadata

abstract methods = implemented by user

Structure of a P4 program



stdlib.p4

Written by community

stdmodel.p4

Written by community

model.p4

Written by manufacturer

myProg.p4

Written by user

Detailed Design

How is this different from “whitebox”?

- This is a revision of the previous whitebox proposal
- Accomplishes same goals
- Slightly different approach
  - Break out whitebox into multiple simple constructs
    - parser, control, package
  - Allows for separate type-checking
  - Modeled after C++/Java OO
  - We can provide an operational semantics for all these constructs

P4 program skeleton

```
// standard definitions
#include “stdlib.p4”
// architecture description; includes Switch decl.
#include “arch.p4”
// user code
parser myParser …
// architecture instantiation
Switch(myParser(), myControl()) main;
```

Preliminaries

- Add a proper “struct” type
- Can be used for metadata (replacing the **metadata** keyword)
- The **header** type is just for headers
- Structs can be nested (but not headers)

header ethernet { … }

header ipv4 { … }

struct headers_t { ethernet e; ipv4 ip; … }

Basic building blocks

- **parser** and **control**
- They look like functions
    - Local scope
    - Arguments with directionality
    - They are typed
- **Rationale:**
    - **in** and **out** arguments indicate the scope of metadata and the user data. For example, the parser metadata cannot be accessed in the Ingress block.
    - Signatures allow type checking
    - Help with resource allocation, by delimiting the scope of various structures.

Parsers

- Rename **parser** -> **state**
- Use **parser** for grouping states

```
parser P(in parser_metadata_t pm,
         out header_t headers)
{
  state start { … extract(headers.e); … }
  state parse_ip { … }
}
```

Control blocks

```
control Ingress(inout headers_t headers,
           in control_metadata_in cmi,
           out control_metadata_out cmo)
{
  table t { … }
  action a { … }
  apply { /* control body */ }
}
```

Declarations

- Architecture declares **prototypes** for programmable blocks
- Users define blocks with matching prototypes

```
parser P<H>(in parser_metadata pm, out H headers);
control Ingress<H>(inout  H headers,
             in control_metadata_in cmi,
             out control_metadata_out cmo);
```

- **Rationale**:
  - type variables indicate user-specified types
  - Type variables are only allowed in architectural specifications
  - users cannot write code containing type variables

Persistent Resources

- Compiler must allocate resources
- E.g., extern objects, tables, and blocks containing such objects
- Parsers and control blocks are persistent resources

```
parser name(arguments)
{
    stateful_Instantiations
    state { … }
}

control name(arguments)
{
    stateful_Instantiations
    apply { /* control flow here */ }
}
```

Instantiating a resource

```
extern Checksum16 { … }

parser MyParser(...)
{
   Checksum16 ck; // checksum unit instantiation
   ...
   state start { … }
   state ipv4 { … ck.verify(h.ipv4); … }
}
```

Types and instances

- **parser** and **control** block declare types
- Types must be instantiated to be used

```
control IPv4Control(inout  Headers headers)
{ ... }

control Ingress(inout Headers headers, ...)
{
   IPv4Control() ipv4control; // instantiate control block
   table acl { ... } // table instantiation

   apply {
     ...
     ipv4control.apply(headers); // invoke control instance
     acl.apply();                // invoke table instance
   }
}
```

Rationale for instantiations

- Parsers and control blocks are similar to classes in OO languages.
- Separating type declaration from instantiation allows one type to be instantiated multiple times.
- E.g,: configure a switch that has 4 ingress pipelines where each of them can be programmed independently: the programmer can write one type, and instantiate it 4 times, once for each pipeline.
- Instantiation is denoted using constructor invocation.

Packages

- A **package** is a container which may contain other packages, parsers and control blocks.
- The toplevel forwarding element is declared as a package by the architecture manufacturer and instantiated by the user.

Switch instantiation outline

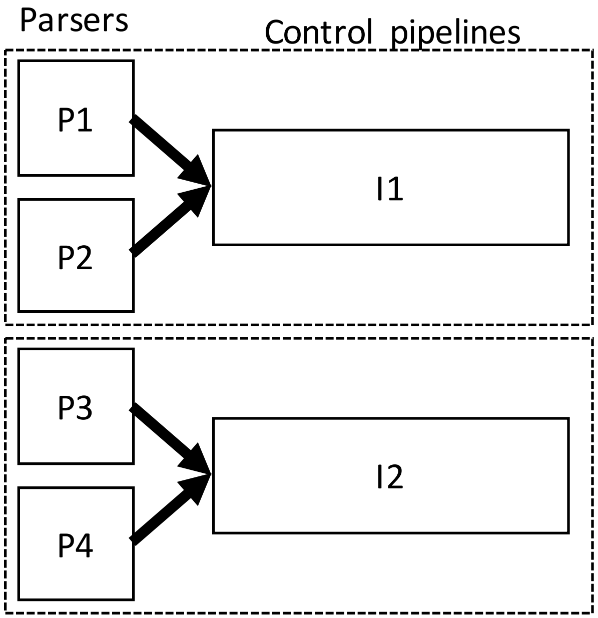
```
// Architecture declaration by manufacturer
parser Parser<H>(out H headers, …);
control Ingress<H>(inout H headers, …);
control Deparser<H>(in H headers, …);
// toplevel element:
package Switch<H>(Parser<H> p,
                  Ingress<H> ingress,
                  Deparser<H> deparser);

// Program written by user
struct head { … }
parser MyParser(out head h, …) { … }
control MyIngress(inout head h) { … }
control MyDeparser(in head h...) { … }

// toplevel element instantiation
Switch(MyParser(),
       MyIngress(),
       MyDeparser()) main;
```

H = **struct** head –
inferred by compiler

Rationale

- The manufacturer can specify complex switches, with many programmable surfaces.
- The type parameters allow various switch components to be linked with each other
  - (e.g., the headers from the parser are the input/output of the ingress pipeline and the input to the deparser
  - the user cannot write an ingress pipeline that accidentally processes different headers from the parser).
- The manufacturer can expose multiple switch models, and the user can choose which one to instantiate (e.g., a standard model, or a model with additional features).
- The user can explicitly instantiate each programmable surface of the switch with the desired implementation.

A Complex Example



```
parser P<H>(out H headers);
control I<H>(inout H headers, …);

package module<H>(P<H> p1, P<H> p2, I<H> pipe);

package switch<H1, H2>(module<H1> mod1,
                       module<H2> mod2);
```

Different headers allowed

Parameterization

- Third parties can write pre-packaged P4 code, which can be reused in a modular way.
- To suit the needs of arbitrary users, these blocks may be parameterized
- Similar to C++/Java/ML Functors

```
control ACLControl(bool  largeSize) // parameters
             (inout Headers h) // arguments
{ ... }


#include  "aclControl.p4"


control Ingress(inout Headers headers)
{
    ACLControl(false) aclControl; // instantiate with largeSize=false

    apply {
      aclControl.apply(headers);
    }
}
```