general interface between the controller and the switches. That is, we believe that future generations of OpenFlow should allow the controller to *tell the switch how to operate*, rather than be constrained by a fixed switch design. The key challenge is to find a "sweet spot" that balances the need for expressiveness with the ease of implementation across a wide range of hardware and software switches. In designing P4, we have three main goals:

- **Reconfigurability.** The controller should be able to redefine the packet parsing and processing in the field.
- **Protocol independence.** The switch should not be tied to specific packet formats. Instead, the controller should be able to specify (i) a packet parser for extracting header fields with particular names and types and (ii) a collection of typed match+action tables that process these headers.
- **Target independence.** Just as a C programmer does not need to know the specifics of the underlying CPU, the controller programmer should not need to know the details of the underlying switch. Instead, a compiler should take the switch's capabilities into account when turning a target-independent description (written in P4) into a target-*de*pendent program (used to configure the switch).

The outline of the paper is as follows. We begin by introducing an abstract switch forwarding model. Next, we explain the need for a new language to describe protocol-independent packet processing. We then present a simple motivating example where a network operator wants to support a new packet-header field and process packets in multiple stages. We use this to explore how the P4 program specifies headers, the packet parser, the multiple match+action tables, and the control flow through these tables. Finally, we discuss how a compiler can map P4 programs to target switches.

Related work. In 2011, Yadav et al. [4] proposed an abstract forwarding model for OpenFlow, but with less emphasis on a compiler. Kangaroo [1] introduced the notion of programmable parsing. Recently, Song [5] proposed protocol-oblivious forwarding which shares our goal of protocol independence, but is targeted more towards network processors. The ONF introduced table typing patterns to express the matching capabilities of switches [6]. Recent work on NOSIX [7] shares our goal of flexible specification of match+action tables, but does not consider protocol-independence or propose a language for specifying the parser, tables, and control flow. Other recent work proposes a programmatic interface to the data plane for monitoring, congestion control, and queue management [8, 9]. The Click modular router [10] supports flexible packet processing in software, but does not map programs to a variety of target hardware switches.

2. ABSTRACT FORWARDING MODEL

In our abstract model (Fig. 2), switches forward packets via a programmable parser followed by multiple stages of match+action, arranged in series, parallel, or a combination of both. Derived from OpenFlow, our model makes three generalizations. First, OpenFlow assumes a fixed parser, whereas our model supports a programmable parser to allow new headers to be defined. Second, OpenFlow assumes the match+action stages are in series, whereas in our model they can be in parallel or in series. Third, our model assumes that actions are composed from protocol-independent primitives supported by the switch.

Our abstract model generalizes how packets are processed in different forwarding devices (e.g., Ethernet switches, load-balancers, routers) and by different technologies (e.g., fixed-function switch ASICs, NPUs, reconfigurable switches, software switches, FPGAs). This allows us to devise a common language (P4) to represent how packets are processed in terms of our common abstract model. Hence, programmers can create target-independent programs that a compiler can map to a variety of different forwarding devices, ranging from relatively slow software switches to the fastest ASIC-based switches.

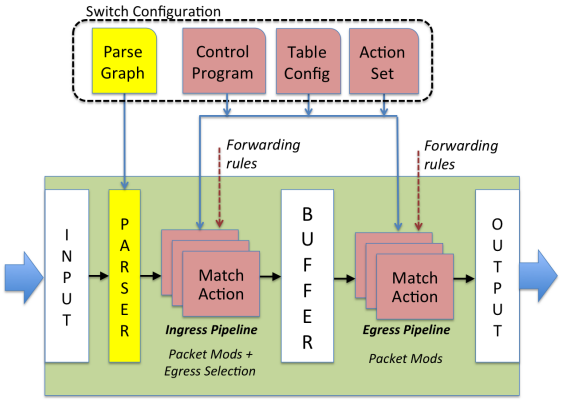

Figure 2: The abstract forwarding model.

The forwarding model is controlled by two types of operations: Configure and Populate. *Configure* operations program the parser, set the order of match+action stages, and specify the header fields processed by each stage. Configuration determines which protocols are supported and how the switch may process packets. *Populate* operations add (and remove) entries to the match+action tables that were specified during configuration. Population determines the policy applied to packets at any given time.

For the purposes of this paper, we assume that configuration and population are two distinct phases. In particular, the switch need not process packets during configuration. However, we expect implementations will allow packet processing during partial or full reconfiguration enabling upgrades with no downtime. Our model deliberately allows for, and encourages, reconfiguration that does not interrupt forwarding.

Clearly, the configuration phase has little meaning in fixed-function ASIC switches; for this type of switch, the com-