P4DDG: Data-Dependent Grammars for Packet Specification and Parsing in P4

Tommaso Pacciani
t.c.pacciani@uva.nl
Informatics Institute, University of Amsterdam
Amsterdam, The Netherlands

Damian Frolich
dfrolich@acm.org
Informatics Institute, University of Amsterdam
Amsterdam, The Netherlands

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org
Informatics Institute, University of Amsterdam
Amsterdam, The Netherlands

Chrysa Papagianni
c.papagianni@uva.nl
Informatics Institute, University of Amsterdam
Amsterdam, The Netherlands

Abstract

In software-defined networking, the domain-specific language P4 allows developers to program the behavior of networking devices at a comparatively high level of abstraction. A P4 program defines a state machine to parse incoming packets. The parsers are flexible and efficient but may be sensitive to bugs and difficult to maintain. As a possible alternative, data-dependent grammars (DDGs) can describe both packet structure and parser at a higher level of abstraction.

In this work, we investigate the use of DDGs in P4 programs. In particular, we demonstrate how DDGs can be used to simultaneously define packets and parsers. We describe a DDG to P4 compiler and evaluate our approach empirically. Input to our evaluation is a collection of P4 programs with for each: the original (handwritten) parser, the (handwritten) DDG alternative, and the compiler-generated parser. The handwritten and generated parsers are compared for equivalence and performance.

Our results show that the generated parsers are three times slower for grammars that do not utilize features distinguishing DDGs from context-free grammars. When parameterized nonterminals are used, a key feature of DDGs, the generated parsers are around six times slower.

CCS Concepts: • **Software and its engineering** → **Parsers**; • **Networks** → Programmable networks.

Keywords: P4, software-defined networking, domain-specific languages, data-dependent grammars, parser generators, compilers

ACM Reference Format:
Tommaso Pacciani, Damian Frolich, L. Thomas van Binsbergen, and Chrysa Papagianni. 2025. P4DDG: Data-Dependent Grammars for Packet Specification and Parsing in P4. In *Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '25), July 3–4, 2025, Bergen, Norway.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3742876.3742879

GPCE '25, Bergen, Norway

ACM ISBN 979-8-4007-1995-0/25/07
https://doi.org/10.1145/3742876.3742879

1 Introduction

Modern-day networks are no longer just plumbing, moving data around. They have become complex and programmable systems that run multiple services, or programs, that provide the functionality needed to adhere to the different requirements of the networks, such as guarantees about performance or isolation. Traditional networking makes adapting to new requirements, such as a new protocol, difficult and time and resource-intensive intensive as modifications to hardware are often necessary. Software-Defined Networking (SDN) is more flexible by making the data plane – in which packets are parsed, possibly transformed, and forwarded – programmable. By programming the data plane, a network can be adapted to support new functionality without requiring modifications to the hardware. Data plane programming is achieved via a programmable interface that is generally low-level and vendor-specific, making development error-prone and not portable. Various domain-specific languages (DSLs) have been introduced in the context of SDN to raise the level of abstraction and increase portability [10]. In this paper we focus on the Programming Protocol-independent Packet Processors language P4 [4].

P4 programs encode several parts of a packet processing pipeline, starting with the transformation of a sequence of bytes (a packet 'on the wire') into a sequence of headers. A header is effectively a structure in which each field has a particular size (a number of bits). Parsers, defined in P4 as state machines, structure the packet in accordance to header specifications. At a relatively low level of abstraction, P4 parsers give the programmer flexibility and the ability to write parsers with good performance characteristics. However, this flexibility increases the potential for various kinds of bugs, which are not uncommon as a result [16]. Moreover,

the way parsers are defined can complicate understanding exactly which packets are accepted or rejected by a P4 parser.

Parsing, i.e., recognizing a specified structure in an unstructured input format, is a common problem across domains with a wide range of possible solutions. In some cases, such as for programming languages and DSLs themselves, the structure can be specified using a context-free grammar from which parsers can be generated according to various strategies. In the case of packets, however, parsing is *context-sensitive*: the contents of the packet (earlier) may determine the expected structure of the packet (later).

The data-dependent grammar (DDG) formalism extends context-free grammars with exactly these kinds of dependencies [12]. Specifically, grammar rules can be conditioned using predicates expressed in terms of variables bound to previously obtained parse results. In this sense, DDGs are an ideal fit for specifying the structure of packets. Moreover, DDGs admit the generation of (context-sensitive) parsers, suggesting we can use a DDG to specify packet structure and packet parser simultaneously, with parsers that are correct by construction.

In this paper, we contribute to the efforts of raising the level of abstraction in data plane programming by investigating the use of data-dependent grammars (DDGs) in domain-specific languages for software-defined networking. Specifically, we investigate whether DDGs are a viable alternative to the current mechanisms of the P4 language for specifying headers and implementing parsers. Concretely, we make the following contributions:

- A variant of P4 extended with data-dependent grammars, called P4DDG, and a compiler from P4DDG code to P4 evaluated for correctness.
- An evaluation of data-dependent grammars as an alternative to handwritten P4 parsers in terms of runtime performance.

In Section 2, we introduce the running example that is progressively converted from a DDG to a P4 parser throughout the paper. In Section 3, we provide background information on parsing, SDN, and P4, detail how parsers are defined in P4, and provide the formal definitions that form the foundation for this work. In Section 4 we present our compiler, discuss the design decisions, and show how the compiler produces P4 code for the running example. In Section 5 we empirically evaluate our approach.

2 Running Example

Throughout this paper, we use a running example to demonstrate our contributions and to aid understanding. Our running example is a `basic tunnel` program from the P4 tutorial. The tutorial scenario begins with a P4 program that switches packets based solely on an optional IPv4 protocol header. The program is extended to support packets that can have a custom, optional `MyTunnel` header. The packet is passed on through the network differently depending on the presence/absence of MyTunnel.

```
1 Hdr = hdr.ethernet Tunnel(hdr.ethernet.etherType)
2 Tunnel(bit<16> type) =
3     [type == TYPE_IPV4] hdr.ipv4
4     | [type == TYPE_TUNNEL] hdr.tunnel Tunnel(hdr.
          tunnel.protoid)
5     | //empty
```

Figure 1. A DDG for the running example written in functional style.

The structure of the expected (top-level) headers can be described as follows at the abstract level, where both the MyTunnel and IPv4 headers are optional (indicated by the question mark):

$$\text{Hdr} = \text{Ethernet}\ (\text{MyTunnel?})\ (\text{IPv4?})$$

The above describes four kinds of acceptable sequences of *abstract* headers, i.e., where the MyTunnel is (or is not) included and the IPv4 header is (or is not) included. However, the *concrete* contents of the Ethernet header determine whether a MyTunnel header can/should be present and, likewise, the contents of the MyTunnel header (if present) determine whether an IPv4 header can/should be present. As such, certain sequences of headers are acceptable at the abstract level, which are not acceptable at the concrete level. A data-dependent grammar makes it possible to encode such dependencies on concrete values in an otherwise still abstract specification. A DDG for the running example that captures this data-dependency is given in Figure 1. The nonterminal `Tunnel` is called with the parameter `tunnel` set to `etherType`, from the ethernet header, to determine which header to parse next. If `type` is equal to the constant `TYPE_IPV4`, the `ipv4` header is parsed. If instead it is equal to `TYPE_TUNNEL`, the `tunnel` header is parsed, followed by a recursive call to `Tunnel` using the `protoid` field from the tunnel header. The empty alternative captures cases where no further headers are to be parsed.

3 Background

In this section, we build upon data-dependent grammars and focus on (parsers in) P4. In this section we detail the background knowledge needed to understand our contributions.

3.1 SDN with OpenFlow

The SDN approach separates the network into two parts, a control plane and a data plane. The control plane contains the logic for the network functions, such as path finding, firewalls, load balancing etc. Whereas the data plane handles moving and modifying packets at line rate based on a configuration provided by the control plane. In this architecture, the network is programmable from the top (the

control plane) down (to the data plane). Applications run at the above the controller and communicate their network requirements to the controller across the northbound API. These requirements may rely on the network view provided by the controller or on external inputs. The controller then configures the data plane devices across the southbound API in accordance with the requirements.

For many years, OpenFlow was a staple of SDN. In OpenFlow, data plane devices are configured by match-action rules. A match-action rule specifies that when some specified part of a packet matches a given value, the action specified by the rule must be taken. Examples of actions are: modifying fields of a packet, dropping a packet, or sending it to a specific (output) port. Crucially, OpenFlow supports a fixed set of networking protocols, pre-determining which (type of) headers can be matched and which actions can be taken in response. The P4 language addresses this limitation by generalizing the match-action paradigm to user-defined protocols and actions.

3.2 P4

P4 is a language for programming network devices such as switches and NICs. Compared to OpenFlow, P4 enables a fully programmable packet processing pipeline, including for novel networking protocols. That is, P4 allows users to define packet formats and arbitrary operations on packets, providing flexibility in the choice of networking protocols.

The P4 language covers specifying packets and packet transformations. To deal with the heterogeneity of networking devices, P4 programs must include the architecture they use. The architecture specifies a signature for every component of the packet-processing pipeline and declares objects and externs for device-specific functionality. The Simple Switch architecture, used in this paper for evaluating generated parsers, is depicted in Figure 2. The packet processing pipeline starts with a parser that maps bits to a header struct and one or more variables that capture metadata, and ends with a deparser, that maps the headers back to bits. Beginning with a parser and ending with a deparser is common to every architecture, although the signatures could differ between architectures.

Between the parser and deparser, there can be multiple control blocks. Control blocks contain the logic for transforming the header struct, which is eventually deparsed to obtain the modified packet. Control blocks follow the match+action paradigm established by OpenFlow. In P4 this works as follows: the programmer specifies a collection of match+action tables, and for each declares what parts of the header struct are to be matched against the keys of the table and what action can be taken when a match is found. At runtime, the control plane is responsible for instantiating the tables with pairs of values and actions.

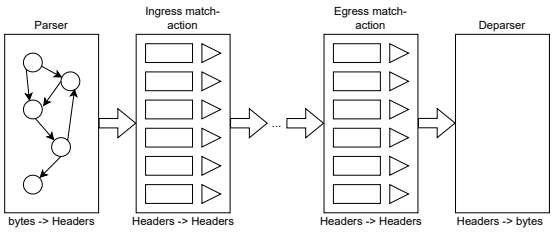

Figure 2. A diagram depicting the packet-processing pipeline of the Simple Switch architecture

Specifying Headers. The `header` keyword is used to declare a structure that represents a single header. To give an idea of header definitions and packet structure definitions, two headers and the global `headers` struct from our running example are defined as follows, where we have omitted the IPv4 header definition for brevity.

```
typedef bit<48> macAddr_t;
header ethernet_t {
  macAddr_t dstAddr;
  macAddr_t srcAddr;
  bit<16> etherType;
}
header myTunnel_t {
  bit<16> proto_id;
  bit<16> dst_id;
}
struct headers {
    ethernet_t ethernet;
    myTunnel_t myTunnel;
    ipv4_t ipv4;
}
```

Both headers contain several fields of varying sizes. Besides bits of specific lengths, header fields can also be among others of type `int`, `boolean`, or a `struct`. In P4, structs are similar to headers but differ in the fact that for the former, every field must be non-empty. While for the latter, any combination of fields can be set.

Parsing Packets in P4. In P4, parsers are defined as state machines comprised of a set of states and transitions between states. A state in the parser consists of a body that can contain imperative statements followed by one or more outgoing transitions. For our running example, the parser starts with the following state.

```
state start {
  transition parse_ethernet;
}
```

The snippet above defines a state that does not perform any operations and transitions to the state `parse_ethernet`, shown below. In case of multiple outgoing transitions, the

`select` construct can be used to select between alternative transitions based on the value of an expression. In the snippet below, the next transition is chosen based on the value of `hdr.ethernet.etherType`.

```
state parse_ethernet {
  packet.extract(hdr.ethernet);
  transition select(hdr.ethernet.etherType) {
      TYPE_MYTUNNEL: parse_myTunnel;
      TYPE_IPV4: parse_ipv4;
      default: accept;
  }
}
```

The snippet above contains a single statement, invoking the `extract` operation, extracting a number of bits from `packet`, equal to the type of `hdr.ethernet`, and assigns these bits to the `ethernet` field of the `hdr` struct.

The `select`$(x)\{\ S_0 : s_0\ ;\ \dots\ \}$ construct defines a conditional transition to the state s_0 if x is a member of the set S_0. In the example above, the `TYPE_IPV4` evaluates to a singleton containing the `TYPE_IPV4` constant.

3.3 Languages, Grammars & Parsing

In the context of syntax analysis, a language is defined as the set of all sentences generated by the grammar defining the syntax of the language. A grammar [5] defines the set of words that can be used in sentences of the language and a number of derivation rules for generating sentences of the right structure. A subset of the class of all grammars, the context-free grammars, is generally used to specify the (concrete) syntax of programming languages.

Definition 3.1. A context-free grammar (CFG) is a tuple $\langle \Sigma, \Delta, A_0, P \rangle$, where Σ is a finite set of terminals (the words of the language); Δ is a finite set of nonterminals; $A_0 \in \Delta$ is the start nonterminal; and $P \subseteq \Delta \times (\Sigma \cup \Delta)^*$ is the set of derivation rules, with $*$ denoting the free monoid of a set.

The derivation rules of a context-free grammar, also referred to as production rules, are often denoted as $X \longrightarrow \alpha$, where the *left-hand side* X is a non-terminal symbol and the *right-hand side* α is a sequence of non-terminal and terminal symbols. The set P of a context-free grammar can thus be seen to map each non-terminal to its right-hand sides. A derivation rule is applied to an occurrence of a non-terminal in a sequence of symbols by replacing the non-terminal with one of the right-hand sides of the nonterminal. A derivation is formed by repeatedly applying derivation rules to a sequence of symbols until only terminal symbols remain. The set of sentences in a language is the set of terminal sequences that can be obtained through repeatedly applying derivation rules starting from the start symbol A_0. Intuitively, by occurring in the right-hand side of derivation rules, non-terminal symbols thus give structure to the sentences of the language.

The derivation process is inherently non-deterministic as a non-terminal can have multiple right-hand sides in which case multiple derivation rules can be applied to the nonterminal. This non-determinism is the main problem tackled by parsing algorithms, which are tasked with finding one or more derivations of an input sentence if there are any. A parser can look ahead one or more symbols into the input sentence to help choosing between alternatives. However, for every choice of amount of lookahead, example grammars and inputs exist for which lookahead is not sufficient. With insufficient lookahead, a parser can apply backtracking to correct choosing a failing alternative. Backtracking is not regularly used in practice because of the performance drawbacks. For a more extensive take on grammars and parsing, the reader is referred to [3, 11].

In our work we adopt data-dependent grammars, a generalisation of context-free grammars that supports data-dependent conditions in right-hand sides of nonterminals. To avoid backtracking, we assume that the conditions associated with right sides in the data-dependent grammars are mutually exclusive (independent of lookahead).

3.4 Data-Dependent Grammars

Data-dependent grammars [12] provide a grammar formalism that incorporates data-dependencies within the grammar specification. Using these data-dependencies, disambiguation strategies can be specified [2] and some context-sensitive parsing [15] can be performed.

Definition 3.2. A data-dependent grammar (DDG) is a tuple $\langle \Sigma, \Delta, A_0, \mathcal{R} \rangle$, where Σ, Δ, A_0 are as defined in the CFG definition and $\mathcal{R} \subseteq \Delta \times r$ relates nonterminals to regular right sides as defined below.

Definition 3.3. Regular right sides are inductively defined by the following grammar.

$$
\begin{array}{rll}
r ::= & \epsilon & \text{(empty string)} \\
\mid & t & \text{(terminal)} \\
\mid & A(e) & \text{(nonterminal call with P4 expression)} \\
\mid & i_1, \dots i_n & \text{(P4 statements)} \\
\mid & (r.r) & \text{(concatenation)} \\
\mid & (r \mid r) & \text{(alternation)} \\
\mid & (r^*) & \text{(Kleene closure)} \\
\mid & [e] & \text{(A P4 expression as a condition)}
\end{array}
$$

The definition above is an adaptation of the original from [12] ignoring some unused operators, embedding P4 statements and utilising P4 expressions as conditions.

Data-dependent grammars incorporate several key elements that distinguish them from conventional context-free grammars. *Parameterized nonterminals* take a number of

parameters to provide context to their right sides. In the formalization, a single parameter is used without loss of generality. We use the abstract *param* function to obtain the parameter identifier of a non-terminal. A call $A(e)$ is realized by evaluating the argument-expression e to a value and binding the parameter to the value. The parameter is then available for use as a *variable* in conditions, which are themselves P4 expressions. A *condition* $[e]$ serves as a guard for right sides, where e is a P4 expression. That is, a right side can only yield a derivation (or successfully parse) if all its conditions evaluate to true in the given context. Right-sides are defined akin to regular expressions, with the Kleene-closure operator and binary concatenation and alternation operators. The definition is completed by empty right sides ϵ, terminal symbols t, and P4 statements $i_1, \ldots, i_n$. Compared to the original definition, we do not use a separate binding operator $\{x = e\}$. Instead, a P4 assignment statement can be used wherever binding the value of an expression e to a variable x is desired.

The following example, taken from the original DDG paper [12], shows a DDG that captures fixed-width strings.

```
StringFW(n) = [n = 0] ϵ
            | [n > 0] char8 StringFW(n - 1)
```

This example highlights conditions and parameterized non-terminals. Conditionals are recognized as expressions enclosed by square brackets, and parameterized non-terminals are recognized by the fact that the `StringFW` non-terminal has a parameter (n) as part of its definition. In this example, `char8` denotes a terminal, capturing an 8-bit character. Throughout this paper, we use the convention that terminals start with a lowercase character while non-terminals start with an uppercase character.

The example uses a functional style, since it performs multiple calls to a non-terminal with a modified argument, simulating the idea of (recursive) functions. Alternatively, the example can be encoded using a more procedural or imperative style:

```
StringFW(n) = ([n > 0] char8 {n= n - 1})* [n = 0]
```

This style uses iteration with the Kleene closure operator rather than recursion and uses a binding, `n= n - 1`, instead of a recursive call with a modified argument. Both styles use conditionals, but the procedural style uses conditionals at the start and end of the rule, while the functional style only uses conditionals at the start of rules. Conditionals at the end of a rule function like post-conditions and, in this example, guide the application of the Kleene closure, which now unfolds until n is equal to 0. Without the conditional at the end, this rule would accept any string with a length within the bounds $[1, n]$. In our implementation, iteration proceeds (greedily) as long as the pre-condition of the iterated statement holds, relying on the assumption that every iterated statement has a pre-condition that eventually evaluates to false.

Data-dependent grammars generalize the class of context-free grammars and can be implemented using generalized parsing algorithms such as GLR [17] and GLL [2, 14]. In [12], the generalized parsing algorithm by Earley [7] was adapted for data-dependent grammars and combined with a scanner-less approach. In [12], *transducers* are used as an intermediate abstraction to express and prove correct the semantics of the presented parsing algorithm. In this paper we use transducers as an intermediate representation for generating the P4 parsers, taking advantage of the similarities between transducers and P4 parsers.

DDG Transducers. Transducers are a type of automata that include states that produce output.

Definition 3.4. A parsing transducer T is a tuple $(\Sigma, \Delta, Q, A_0, q_0, \xrightarrow{l}, \mapsto)$, where:

- Σ is a finite set of terminals;
- Δ is a finite set of nonterminals;
- Q is a finite set of states;
- A_0 is the start nonterminal, $A_0 \in \Delta$;
- q_0 is the initial state, $q_0 \in Q$;
- $\xrightarrow{l}$ is the transition relation where label l is either:
  - ϵ or simply omitted,
  - a terminal t,
  - a guard e, with e a P4 expression (here and below),
  - a sequence of P4 statements $i_1, ..., i_n$,
  - a call to a nonterminal $call(A(e))$,
  - or a return from a call $return(A(e))$.
- $\mapsto \subseteq Q \times \Delta$ is the output relation.

The translation from DDGs to transducers is given by an extended version of Thompson's construction provided by [12] and presented here in Figure 3. The first judgment $S \vdash r \rightsquigarrow (s, F, \mathcal{T})$ defines how a regular right side r is translated to a transducer graph $\mathcal{T}$. S is a mapping from nonterminals to states, relating a nonterminal to the start state of the (sub-)automata implementing that nonterminal. In accordance with [12], the meta-variable $\mathcal{T}$ is overloaded to represent both a transducer graph, a tuple of states and transitions, and a transducer with a start state. The second judgment $G \rightsquigarrow \mathcal{T}$ uses the first judgment to build a transducer graph for each regular right side of a nonterminal in the data-dependent grammar G. The semicolon operator in $\mathcal{T}_1; \mathcal{T}_2$ indicates the unions of the states and transitions of both transducers. The transducer for our running example is visualized in Figure 4

Note that we do not utilize the output relation of transducers. This is because in the generated parser, a terminal t gives rise to an application of the `extract` operator making the parsed data available through the packet header as a global variable (under the field name t).

$$\boxed{S \vdash r \rightsquigarrow (s, F, \mathcal{T})}$$

$$\text{T-Eps}\ \frac{s, F \text{ fresh}}{S \vdash \epsilon \rightsquigarrow (s, F, [s \to F])}$$

$$\text{T-Term}\ \frac{s, F \text{ fresh}}{S \vdash t \rightsquigarrow (s, F, [s \xrightarrow{t} F])}$$

$$\text{T-Pred}\ \frac{s, F \text{ fresh}}{S \vdash [e] \rightsquigarrow (s, F, [s \xrightarrow{e} F])}$$

$$\text{T-P4Stmts}\ \frac{s, F \text{ fresh}}{S \vdash i_i, \ldots, i_n \rightsquigarrow (s, F, [s \xrightarrow{i_i, \ldots, i_n} F])}$$

$$\text{T-Call}\ \frac{s, F \text{ fresh} \quad l_1 = return(A(e)) \quad l_2 = call(A(e))}{S \vdash A(e) \rightsquigarrow (s, F, [s \xrightarrow{l_1} F]; [s \xrightarrow{l_2} S(A)])}$$

$$\text{T-Seq}\ \frac{\begin{array}{c} s, F \text{ fresh} \\ S \vdash r_1 \rightsquigarrow (s_1, F_1, \mathcal{T}_1) \quad S \vdash r_2 \rightsquigarrow (s_2, F_2, \mathcal{T}_2) \\ \mathcal{T}_3 = [s \to s_1]; [F_1 \to s_2]; [F_2 \to F] \end{array}}{S \vdash r_1 . r_2 \rightsquigarrow (s, F, \mathcal{T}_1; \mathcal{T}_2; \mathcal{T}_3)}$$

$$\text{T-Alt}\ \frac{\begin{array}{c} s, F \text{ fresh} \\ S \vdash r_1 \rightsquigarrow (s_1, F_1, \mathcal{T}_1) \quad S \vdash r_2 \rightsquigarrow (s_2, F_2, \mathcal{T}_2) \\ \mathcal{T}_3 = [s \to s_1]; [s \to s_2]; [F_1 \to F]; [F_2 \to F] \end{array}}{S \vdash r_1 \mid r_2 \rightsquigarrow (s, F, \mathcal{T}_1; \mathcal{T}_2; \mathcal{T}_3)}$$

$$\text{T-*}\ \frac{\begin{array}{c} s, F \text{ fresh} \\ S \vdash r \rightsquigarrow (s_1, F_1, \mathcal{T}_1) \\ \mathcal{T}_2 = [s \to s_1]; [F_1 \to F]; [s \to F]; [F_1 \to s_1] \end{array}}{S \vdash r^* \rightsquigarrow (s, F, \mathcal{T}_1; \mathcal{T}_2)}$$

$$\boxed{G \rightsquigarrow \mathcal{T}}$$

$$\text{T-G}\ \frac{\begin{array}{c} \mathcal{R} = [A_0 = r_{A_0}, \ldots, A_k = r_{A_k}] \\ Q = \{s_{A_0}, \ldots, s_{A_k}\} \text{ fresh} \\ S = [A_0 = s_{A_0}, \ldots, A_k = s_{A_k}] \\ S \vdash r_{A_i} \rightsquigarrow (s_i, F_i, \mathcal{T}_i) \quad (\text{for } i = 0, \ldots k) \\ \mathcal{T}_{\text{init}} = [s_{A_0} \to s_0]; \ldots; [s_{A_k} \to s_k] \\ \mathcal{T}_{\text{final}} = [F_0 \to A_0]; \ldots; [F_k \to A_k] \\ \mathcal{T} = \mathcal{T}_0; \ldots; \mathcal{T}_k; \mathcal{T}_{\text{init}}; \mathcal{T}_{\text{final}} \end{array}}{\langle \Sigma, \Delta, A_0, \mathcal{R} \rangle \rightsquigarrow (\Sigma, \Delta, Q, A_0, s_{A_0}, \mathcal{T})}$$

Figure 3. Adapted translation from data-dependent grammars to transducers.

4 P4DDG

In this section we introduce P4DDG, a compiler from DDGs to P4 parsers. A visual view of the pipeline of our compiler and the integration of the generated parser with the final P4 program is displayed in Figure 5. P4DDG consists of two steps. Step (a) transforms a DDG into a transducer, and step (b) transforms this transducer into a P4 program.

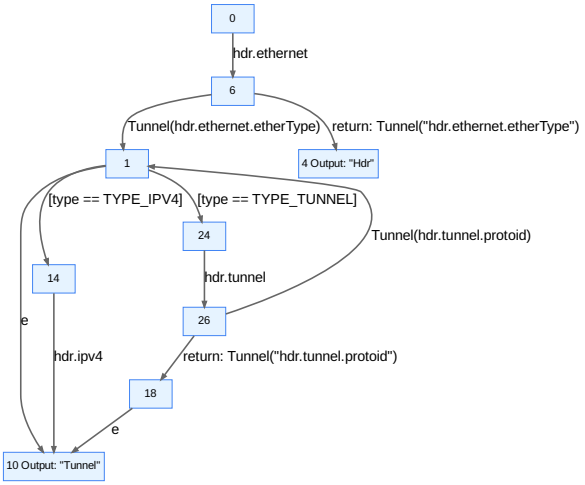

Figure 4. The transducer generated from the DDG of the running example from Figure 1.

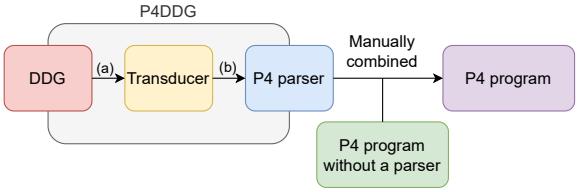

Figure 5. The translation and utilization pipeline of P4DDG.

The first step of P4DDG was formalized in Section 3 as an adaptation of the approach in the original DDG paper [12]. Compared to the original, we directly include P4 code (statements and expressions) in the right sides of DDGs. Moreover, terminals are not characters but instead refer to header struct fields that are expressed in a P4 program (using `extract`). Furthermore, we do not support grammars that require some form of backtracking to parse.

In this section we formalize step (b) of the pipeline. This translation occurs in two steps. In Section 4.1, a transducer is translated to P4-IR, an intermediate representation of a P4 parser simplifying code generation. In Section 4.2, a straightforward translation from P4-IR gives the actual P4 code.

4.1 Translating from Transducers to P4-IR

In this subsection, we describe how we generate P4-IR code from a transducer. The general idea behind the translation is that most labels on the edges of the transducer become imperative statements in the body of a P4 state definition. The abstract syntax of P4-IR is given in Figure 6. Here, t is a placeholder for a P4 field name (instantiating a terminal),

$$
\begin{aligned}
\mathit{stmt} ::= \ & \text{extract}(t) \\
\mid \ & \text{assign}(x, e) \\
\mid \ & \text{push}(l) \\
\mid \ & \mathit{p4s}(i^*) \\
\mathit{trans} ::= \ & \text{if}(e, s) \\
\mid \ & \text{goto}(s)
\end{aligned}
$$

Figure 6. Abstract syntax for P4-IR.

$$
\boxed{[\![\,]\!]^{\text{abs}} : (Q \times l \times Q) \mapsto (\mathit{stmt} \cup \mathit{trans})^*}
$$

$$
\begin{aligned}
[\![s_1 \xrightarrow{\epsilon} s_2]\!]^{abs} &\doteq \text{goto}(s_2) \\
[\![s_1 \xrightarrow{\{i_1,\ldots,i_n\}} s_2]\!]^{abs} &\doteq \text{p4s}(i_1, \ldots, i_2), \text{goto}(s_2) \\
[\![s_1 \xrightarrow{[e]} s_2]\!]^{abs} &\doteq \text{if}(e, s_2) \\
[\![s_1 \xrightarrow{t} s_2]\!]^{abs} &\doteq \text{extract}(t), \text{goto}(s_2) \\
[\![s_1 \xrightarrow{\text{call } A(x)} s_2]\!]^{abs} &\doteq \text{assign}(p, x) \\
&\quad , \text{push}(\psi(s_1, A, x)) \\
&\quad , \text{goto}(s_2) \\
&\quad \textbf{where } p = \mathit{params}(A)
\end{aligned}
$$

Figure 7. Translation from transducers to P4-IR. The ψ function generates a continuation that identifies the return to some state s_3 for the transition $s_1 \xrightarrow{\text{return } A(x)} s_3$.

x for a P4 variable name, e for a P4 expression, i for a P4 statement, l is an integer identifying a continuation used to implement the *call* and *return* labels, and s is an integer identifying a state.

The translation of a single transition in the transducer to P4-IR is given in Figure 7. In this figure, the first equation translates an epsilon transition to a single `goto` to the target state. The second equation translates a sequence of P4 statements into its intermediate representation, followed by a `goto` to the target state. The third equation translates a conditional transition guarded by an expression e to target state s using *if*. The fourth equation translates a transition matching a terminal t into an `extract(t)` followed by a `goto` to the target state. The fifth equation generates instructions to bind the argument of a call to the parameter of the nonterminal f being called and to store a continuation on the stack to handle the eventual return from the call, followed by a `goto` to the target state.

Using these rules, a full transducer T translates as follows:

$$
\mathcal{P}^{IR}(T) = \{(s, \{[\![s \xrightarrow{a} s']\!]^{\text{abs}} \mid (s, a, s') \in \xrightarrow{l}\}) \mid s \in Q\} \quad (1)
$$

$$
\boxed{\Downarrow : \mathit{trans}^* \mapsto \mathit{string}}
$$

$$
\boxed{b_\top : e \mapsto \mathit{string}} \quad \boxed{b_\bot : e \mapsto \mathit{string}}
$$

goto(s_0)

$\Downarrow$

```
transition state_s0 ;
```

if(e_1, s_1), ..., if(e_n, s_n), (goto(s_0))**?**

$\Downarrow$

```
transition select(e_1, . . . , e_n) {
  (b⊤(e1), b⊥(e2), . . . , b⊥(en)) : state_s1;
  (b⊥(e1), b⊤(e2), . . . , b⊥(en)) : state_s2;
  ⋮
  (b⊥(e1), b⊥(e2), . . . , b⊤(en)) : state_sn;
  (default : state_s0;)?
}
```

$b_\top(!e) =$ `false`

$b_\top(e) =$ `true`

$b_\bot(!e) =$ `true`

$b_\bot(e) =$ `false`

Figure 8. The translation from a sequence of transitions (*trans*), from P4-IR, to concrete P4 syntax. Expression $!e$ is the application of the logical negation operator ! of P4.

$$
\boxed{\Rightarrow_{\mathit{stmt}} : \mathit{stmt} \mapsto \mathit{string}}
$$

extract(t) $\Rightarrow_{\mathit{stmt}}$

```
  packet.extract(t);
```

push(c) $\Rightarrow_{\mathit{stmt}}$

```
  return_stack_index = return_stack_index + 1;
  return_stack[return_stack_index].val = c;
```

assign(p, x) $\Rightarrow_{\mathit{stmt}}$

```
  p = x;
```

p4s(i^*) $\Rightarrow_{\mathit{stmt}}$ i^*

Figure 9. The translation of a P4-IR *stmt* to concrete P4.

The translation to P4-IR thus gives a mapping, for every state in the transducer, to a set of sequences, each prefixed by a (possibly empty) sequence of P4-IR statements and ending with a P4-IR transition, i.e., every sequence in the set is of

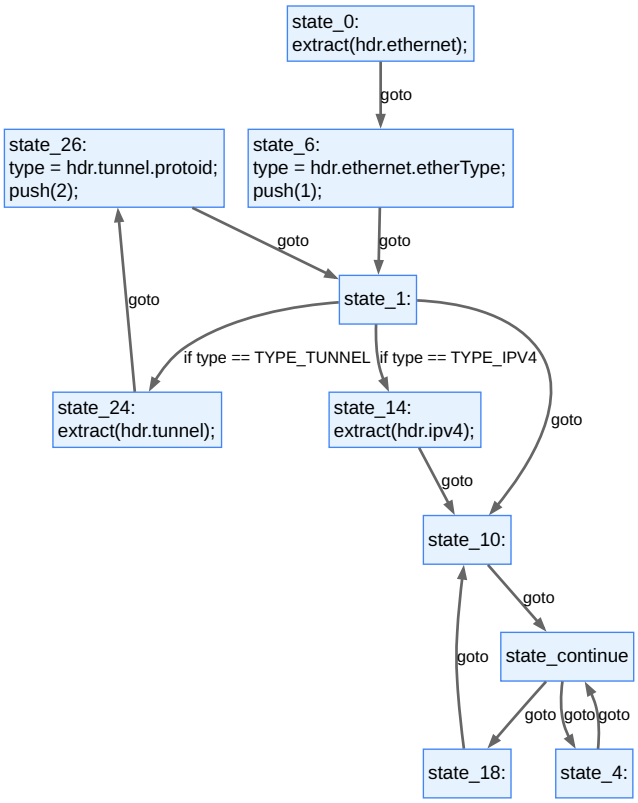

Figure 10. A visualization of the control flow of the running example expressed in P4-IR produced by translating the transducer of Figure 4

the form $stmt^*\ trans$. Note that because of our assumptions on the input DDG, there can be at most one such sequence for which it holds that $stmt^*$ is non-empty. And because of the same assumption, only one such sequence ends with a `goto` transition.

4.2 P4 Code Generation from P4-IR

This subsection describes the production of P4 code as a concrete syntax string via a final translation step. The generated code includes a configurable preamble described towards the end of this subsection.

In Figures 8 and 9 we give separate translations for a sequence of transitions and individual statements respectively. The full translation for a tuple $(s, P) \in \mathcal{P}^{IR}(T)$ generates the code that declares a P4 state for s with a body consisting of a sequence of statements followed by a (selection between) transition(s). The statements are produced by applying $\Rightarrow_{stmt}$ to all statements in the only sequence $stmt^*\ trans \in P$ with non-empty $stmt^*$ (in the order respecting the order of the sequence). The (selection between) transition(s) is produced by applying $\Downarrow$ to all the transitions of P together, ordered as a sequence $trans^*$ such that the `goto` transition is last in the sequence (if it exists).

The first equation in Figure 8 defines P4 code for the simplest sequence of $trans$, a single `goto`. The second translates a sequence of $n-1$ `if`(e_i, s_i) transitions, optionally followed by a single `goto`. The generated code contains a `select` statement with an n-tuple with n P4 expressions e. Every case in the `select` contains an n-tuple of booleans, with one position set to true, for each possible branch. If a `goto(s)` is present, then a default case is added that transitions to `s`.

The translation of individual statements in Figure 9 is straightforward for most P4-IR constructors. In the case of `push(c)`, P4 code is generated that adds the continuation (identifier) c to a global array utilized as a stack. The code for handling returns by popping continuation identifiers from the stack is part of the preamble code explained next.

Every generated P4 parser opens with a preamble containing the instructions to allocate the continuation array, a global index into this array, and the P4 instructions that declare all the parameters of all the nonterminals, which is necessary because P4 transitions do not accept arguments. The size of the continuation array is user-defined. In the case of our running example, the preamble looks as follows. The `return_stack` is the continuations array, which is represented, in this program, as an array of size 16 containing 8-bit values. The last line declares the parameter for the *MyTunnel* nonterminal.

```
header return_stack_type { bit<8> val;}
return_stack_type[16] return_stack;
bit<8> return_stack_index = 0;
bit<16> type;
```

Next, a single P4 state is generated that handles returns modeled via continuations. This state has a single select statement with a case for every continuation. The case that matches the last element of the continuation array results in a transition to the state identified by the continuation. If there are no continuations, the packet is accepted, since an empty continuation array corresponds to a successful return from the top-level nonterminal.

For the running example, the return handling state is:

```
state handle_continuations {
    bit<8> tmp_return;
    if (return_stack_index == 0) {
        tmp_return = 0;
    } else {
        tmp_return = return_stack[return_stack_index].
            val;
        return_stack_index = return_stack_index - 1;
    }
    transition select (tmp_return) {
          1 : state_4;
          2 : state_18;
          0 : accept;
    }
}
```

The behavior of this state is similar for every generated parser. We introduce a local variable (`tmp_return`) in which we store the continuation with which to continue. When the continuation stack is empty, we continue with continuation 0, which corresponds to a transition to the *accept* state, indicating acceptance of the packet. Otherwise, we pop the stack and use the popped value as the next continuation. To determine which state a continuation refers to, the state ends with a `transition select` on the popped continuation value. This `transition select` is essentially a lookup table, associating continuations with concrete states. After this `transition select`, the parser transitions to the state identified by the current continuation. For our running example, we have two possible continuations, corresponding to state 4 and state 18 in Figure 10, and the accepting continuation when the continuation stack is empty.

4.3 Additional Syntax

Finally, we propose a specific extension to the syntax for DDGs in P4 that makes it easier to express common patterns observed in P4 parsers. In contrast to conventional P4 parsers, the proposed syntax allows control flow to be denoted using the if-then-else construct and switch construct instead of state transitions. An example of a concrete DDG that shows this difference is shown in Figure 11. The abstract syntax of P4DDG contains the additional constructs $\text{if}(e, stmt^*, stmt^*?)$ and $\text{case}(e, (\langle e, stmt^* \rangle)^*)$. These constructs are desugared using the following rewriting rules:

$$\begin{aligned}
&\text{if}(e, stmt_1^*, stmt_2^*?) = \\
&\quad [e]\ stmt_1^*\ (|\ [(!e)]\ stmt_2^*)? \\
&\text{case}(e_0, \langle e_1, stmt_1^* \rangle, \ldots, \langle e_n, stmt_n^* \rangle) = \\
&\quad [e_0 == e_1]\ stmt_1^*\ |\ \ldots\ |\ [e_0 == e_n]\ stmt_n^*
\end{aligned}$$

5 Evaluation

In this section, we evaluate the use of DDGs in P4. The software and programs used for our experiments are available as a Zenodo publication.

5.1 Experimental Setup

Program Collection. The experiments are run using P4 programs collected from various online repositories (e.g., those supplementing paper or thesis publications). For each collected program we write a DDG definition that can replace the header specification and parser of the program without modifying the functional behavior of the program. From the DDG definition we generate a parser for comparison with the original parser. The original parser and generated parser are compared for functional equivalence to establish the correctness of the generated parser. Moreover, the original and generated parsers are compared for runtime performance.

Processing Pipeline. An overview of our experimental setup is given in Figure 12. The original P4 program and the program with the generated parser are run on a BMv2 [6] virtual switch to process packets. We use BMv2 which, as a target for P4, aims to be feature-rich and easy to use but explicitly does not prioritize performance. Therefore we do not intend to draw conclusions from the presented experiments about the real-world performance on (any specific) hardware targets. To compare functional behavior and performance of the two programs, packets are generated using P4testgen [13]. P4testgen is given the original P4 program as input and uses the parser contained in the program to generate packets of a meaningful variety. This is achieved by traversing the (possibly many) paths that can be traversed by following the transitions between parser states. The resulting set of packets is used as input in the runs for both programs. Only the first stage of packet processing, the parsing stage, is relevant to the experiments.

Correctness. To determine that the parsers generated from DDGs behave identically their hand-written counterparts, we use the logs produced by BMv2 to determine the sequence of parsed headers and check these for equality. Note that we do not test if the modifications to the metadata or other variables global to the parser are the same. We chose not to include these assignments in the correctness check as differentiating between variables originating form the program and those introduced by the P4 compilation process proved costly due to the lacking integration with the reference P4 compiler. Furthermore we observed that programs that relied on assignments to metadata passed our correctness check.

All pairs of programs that we have experimented with are included in the supplementary material accompanying this paper. In all cases the check for equality holds, suggesting indeed that the parser generator of P4DDG is correct.

5.2 Runtime Performance

In this section we discuss the results of comparing the running times of the parsers of the two programs. The timings are measured by collecting timestamps from the logs generated by the switch. The timestamps of interest are those produced at the beginning and end of the parsing stage.

The running time for the different parsers can be seen in Figure 13. For the firewall, cache, and fastreact programs, the difference in delay compared to the original parser is the lowest. The reason for this is that the DDGs for these programs do not use parameterized nonterminals and thus, avoid the runtime overhead from the continuation array. For the other three programs, the delay is about seven times as large. The main reason for this is the overhead from allocating and modifying the continuation array.

```
state parse_gtpu {
    packet.extract(hdr.gtpu);
    transition select(hdr.gtpu.ex_flag, hdr.gtpu.seq_flag, hdr.gtpu.
        npdu_flag) {
        (0, 0, 0): parse_inner_ipv4;
        default: parse_gtpu_options;
    }
}
state parse_gtpu_options {
    packet.extract(hdr.gtpu_options);
    bit<8> gtpu_ext_len = packet.lookahead<bit<8>>();
    transition select(hdr.gtpu_options.next_ext, gtpu_ext_len) {
        (0x85, 8w1): parse_gtpu_ext_psc;
        default: accept;
    }
}
state parse_gtpu_ext_psc {
    packet.extract(hdr.gtpu_ext_psc);
    fabric_metadata.spgw.qfi = hdr.gtpu_ext_psc.qfi;
    transition select(hdr.gtpu_ext_psc.next_ext) {
        0x0: parse_inner_ipv4;
        default: accept;
    }
}
```

```
GTPU =
hdr.gtpu
if hdr.gtpu.ex_flag, hdr.gtpu.seq_flag, hdr.gtpu.npdu_flag == (0, 0,
    0) then
    InnerIpv4()
else
    hdr.gtpu_options
    {bit<8> gtpu_ext_len = packet.lookahead<bit<8>>();}
    if hdr.gtpu_options.next_ext, gtpu_ext_len == (0x85, 8w1) then
        hdr.gtpu_ext_psc
        {fabric_metadata.spgw.qfi = hdr.gtpu_ext_psc.qfi;}
        if hdr.gtpu_ext_psc.next_ext == 0x0 then InnerIpv4 () else
            ()
    else ()
```

Figure 11. Concrete syntax example for P4-specific DDGs. On the left is the original P4 parser taken from onos-spgw-int. On the right is the P4DDG alternative in the proposed concrete syntax using the if-construct.

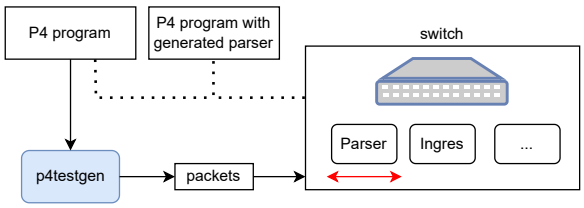

Figure 12. Experimental setup to compare the performance of hand-written P4 parsers with those generated from DDGs.

Analyzing the Impact of Continuations. As we saw in the previous experiment, using the continuation array causes a significant increase in the time it takes to parse a packet. For the second experiment, we attempt to measure the precise impact of the continuation array on the parsing time. The continuation array is used to handle return from, for example, the recursive call in the running example from Figure 1.

For this experiment, we use the following grammar. A packet starts with a number that indicates how many 128-bit headers are present. The size of 128 bits is irrelevant to the experiment; all we are concerned with is the number that specifies how many headers will follow, as this number will determine the number of times we make a recursive call and, thus the usage of the continuation array.

```
Hdr = hdr.num P(hdr.num.val)
P(bit<16> n) =
    [n == 0] //empty
    | [n !=0 ] hdr.header_stack.next P(n-1)
```

The DDG below accepts the same packets but avoids the need for a parameterized nonterminal and, thus, a continuation by using the Kleene star operator.

```
Hdr = hdr.num {n = hdr.num.val;}
    ([n != 0] hdr.header_stack.next {n = n - 1;})*
    [n == 0]
```

In Figure 14 the average parsing delay is compared for the two DDGs given above against the delay for a hand-written parser. The results show that usage of the stack comes with a large increase in the parsing delay. For the DDG that uses recursion, the difference between the other two results can be explained by the following two sources of delay. First, the delay from setting up the continuation array. This is incurred no matter the path taken by the packet. Second, the delay from the operations that modify the continuation stack, that is, looking up the last continuation and transitioning to the correct next state. Furthermore, we see that using the

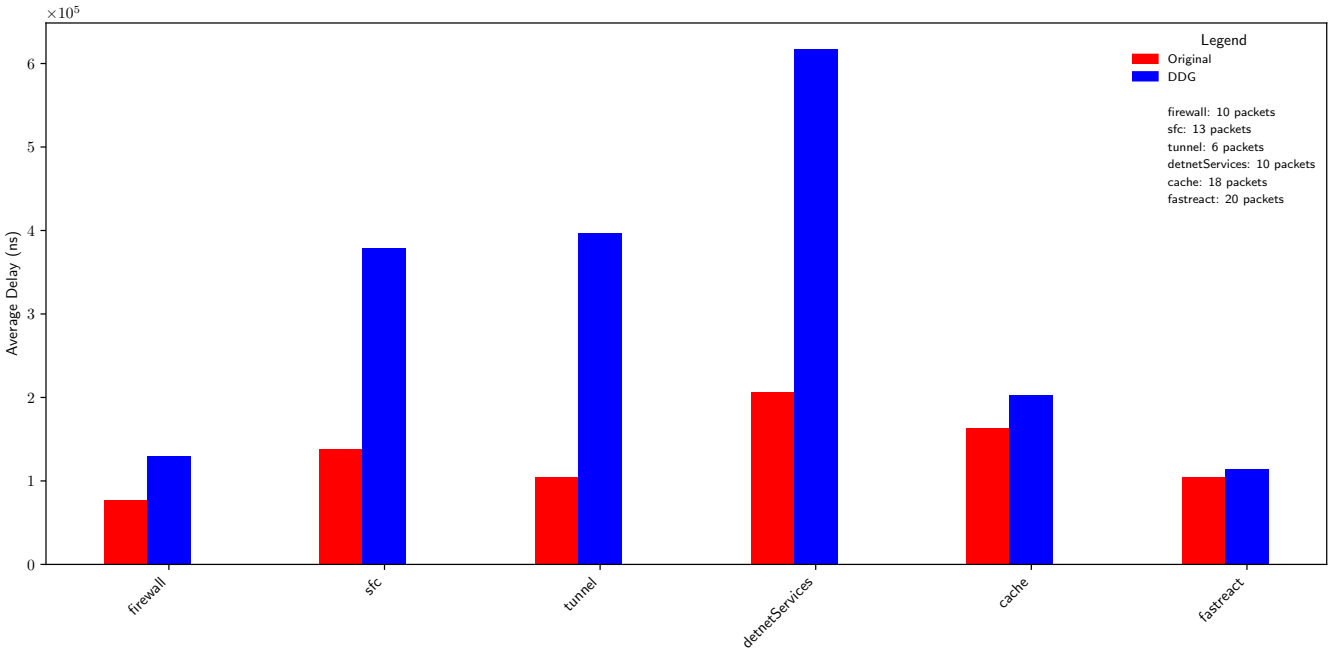

Figure 13. The average parsing time of a DDG-derived parser compared to the original parser for a collection of P4 programs.

imperative style over the functional style results in a parser that is closer in performance to the hand-written parser.

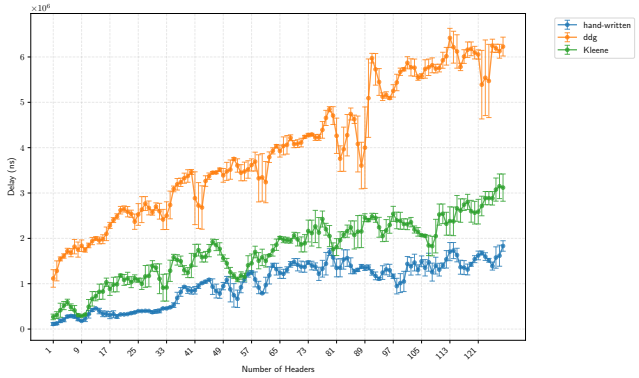

Figure 14. A comparison of the parsing delay for two different styles of specifying iterations in a DDG. The first uses a recursive call, in orange. The second uses the Kleene star operator, indicated in green. The advantage of the Kleene star is that no continuation array is required. Third, a hand-written parser is used as a control.

6 Discussion

From the results, we can see that using an array that holds continuations for returning from parameterized nonterminals comes with a significant increase in parsing time. The delay can be split into two parts. There is a constant delay incurred for every packet from the allocation of the array. Furthermore, significant overhead is incurred for every return that is handled, as can be seen in the second experiment.

From the same experiment, we can see that for parameterized nonterminals with a single callee, the continuations can be avoided for better performance. In the experiment, we avoid generating continuations by using the Kleene star operator. But this same performance benefit could be achieved by not using continuations for parameterized nonterminals with a single callee. Note, in case continuations are used for other nonterminals, then avoiding continuations for some nonterminals will only result in the total overhead from returns being lower, while the overhead from allocating the continuation array would remain.

Limitations. Due to the early-stage nature of this work, there are a few limitations that we find important to discuss. First, there is the low number of programs used in the evaluation section. We encountered multiple challenges collecting programs, with the main one being the fact that there is a limited number of novel protocols that have a P4 definition. Nevertheless, we feel that the included programs contain sufficient non-trivial protocols to show a benefit of our approach, despite the incurred slowdown.

The second limitation is the incurred slowdown for the DDG-generated parser compared to hand-written ones. While the parsing stage generally does not take up a significant amount of time in full P4 programs, the slowdown may still limit the real-world applicability of this work. However, we consider this acceptable for preliminary work. Our research

has shown that DDGs can be a suitable alternative to the current state machines in P4, thus motivating further research into a code generation approach that eliminates the current performance gap.

Finally, we spent some time on optimizations on the P4-IR, of which we implemented one that combines states when possible. This optimization aimed to decrease the number of states in the generated P4 code as much as possible, as we hypothesized that this could be important for the Tofino hardware target. In the end, we did not expand our evaluation to Tofino and found the optimization to result in only a minor decrease in the number of states.

An Alternative Code Generation Strategy. Instead of using a continuation stack, we could specialize nonterminal calls. That is, instead of having one nonterminal being called in two places, we have two copies of the nonterminal such that they both have a single callee, thus removing the need to store a continuation for the two nonterminal calls. Specializing the nonterminal calls, while avoiding the increase in delay from allocating the continuation array, results in a parser with more states, as the complete body of every nonterminal is repeated as many times as the nonterminal is called with different arguments.

7 Related Work

Π4 [8] is a dependently-typed version of P4. With Π4, the type system is used to determine correctness of P4 programs, including the written parsers. This way, certain bugs in parsers are caught by the type system since those programs do not type check. The way in which parsers are written is similar to the way they are written in P4, except Π4 does not support the transition command, instead parsers are written using *if-else* chains. Compared to our work, the contributions of Π4 are more focused on the verification aspects, while our work is more focused on expressiveness and maintainability. Nevertheless, it would be interesting for future work to use our DDGs to verify certain properties of the grammars, or to add Π4 as a new compilation target to our compiler.

In the context of data-dependent grammars, previous work introduced Iguana [1, 2], a framework for data-dependent parsing. As part of this framework, they make several extensions to the data-dependent grammar formalism, and introduce several forms of syntactic sugar for common disambiguation actions. One of the extensions is support for global variables that are reachable to all the rules in the grammar. This aligns with our handling of terminals except more general. In future work, it would be interesting to explore how suitable the other extensions proposed in the framework are within the context of P4DDG.

An alternative approach to writing a grammar is specifying the structure of the data and generating parsers from that description [9]. A common idiom with these frameworks is the use of unions to represent alternatives. This is not directly possible with P4, since P4 does not support unions inside structs. It does support unions for headers. Instead of defining a new language, an interesting alternative is to use the P4 header and struct definitions directly and generate parsers from those definitions. However, it is still unclear how to model the data dependencies between these headers in the definition. Nevertheless, it could be an interesting avenue to explore in future work.

8 Conclusion

We explored the use of data-dependent grammars (DDGs) in the context of P4, a programming language for networking devices. We introduced P4DDG, a P4-specific variant of DDGs, as a means to specify both packet structures and parsers at a higher level of abstraction. We presented a compiler from P4DDG to P4 parsers. By empirically evaluating the equivalence and performance of existing parsers and generated parsers, we demonstrated that P4DDGs offer an alternative to P4 parsers, albeit with some performance trade-offs. Future work will focus on shrinking the performance gap by exploring alternative code generation strategies and refining the current generation approach.

Acknowledgments

This research was funded by the Dutch 6G flagship project "Future Network Services".

References

[1] Ali Afroozeh and Anastasia Izmaylova. 2015. One Parser to Rule Them All. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Gail C. Murphy and Guy L. Steele Jr. (Eds.). ACM, 151–170. https://doi.org/10.1145/2814228.2814242

[2] Ali Afroozeh and Anastasia Izmaylova. 2016. Iguana: A Practical Data-Dependent Parsing Framework. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 267–268. https://doi.org/10.1145/2892208.2892234

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley. I–X, 1–796 pages.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *Comput. Commun. Rev.* 44, 3 (2014), 87–95. https://doi.org/10.1145/2656877.2656890

[5] N. Chomsky and M.P. Schützenberger. 1963. The Algebraic Theory of Context-Free Languages*. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 35. Elsevier, 118–161. https://doi.org/10.1016/S0049-237X(08)72023-8

[6] The P4 Language Consortium. 2025. BMv2: reference P4 software switch. https://github.com/p4lang/behavioral-model.

[7] Jay Earley. 1970. An Efficient Context-free Parsing Algorithm. *Commun. ACM* 13, 2 (1970), 94–102. https://doi.org/10.1145/362007.362035

[8] Matthias Eichholz, Eric Hayden Campbell, Matthias Krebs, Nate Foster, and Mira Mezini. 2022. Dependently-Typed Data Plane Programming.

Proc. ACM Program. Lang. 6, POPL, Article 40 (Jan. 2022), 28 pages. https://doi.org/10.1145/3498701

[9] Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-Specific Language for Processing Ad Hoc Data. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 295–304. https://doi.org/10.1145/1065010.1065046

[10] Nate Foster, Arjun Guha, Mark Reitblatt, Alec Story, Michael J. Freedman, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Jennifer Rexford, Cole Schlesinger, David Walker, and Rob Harrison. 2013. Languages for software-defined networks. *IEEE Commun. Mag.* 51, 2 (2013), 128–134. https://doi.org/10.1109/MCOM.2013.6461197

[11] Dick Grune. 2010. *Parsing Techniques: A Practical Guide* (2nd ed.). Springer Publishing Company, Incorporated. https://doi.org/10.1007/978-0-387-68954-8

[12] Trevor Jim, Yitzhak Mandelbaum, and David Walker. 2010. Semantics and Algorithms for Data-dependent Grammars. *SIGPLAN Not.* 45, 1 (Jan. 2010), 417–430. https://doi.org/10.1145/1707801.1706347

[13] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtech Havel, Hanneli Tavante, Rob Sherwood, Vladyslav Dubina, Volodymyr Peschanenko, Anirudh Sivaraman, and Nate Foster. 2023. P4Testgen: An Extensible Test Oracle For P4. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 136–151. https://doi.org/10.1145/3603269.3604834

[14] Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 177–189. https://doi.org/10.1016/j.entcs.2010.08.041

[15] Bostjan Slivnik. 2022. Context-sensitive parsing for programming languages. *J. Comput. Lang.* 73 (2022), 101172. https://doi.org/10.1016/J.COLA.2022.101172

[16] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, Sergey Gorinsky and János Tapolcai (Eds.). ACM, 518–532. https://doi.org/10.1145/3230543.3230548

[17] Masaru Tomita. 2012. *Generalized LR Parsing*. Springer Science & Business Media. https://doi.org/10.1007/978-1-4615-4034-2

Received 2025-04-10; accepted 2025-05-19