Reducing P4 Language’s Voluminosity using Higher-Level Constructs

Albert Gran Alcoz*  Coralie Busse-Grawitz*  Eric Marty  Laurent Vanbever

ABSTRACT

Over the last years, P4 has positioned itself as the primary language for data-plane programming. Despite its constant evolution, the P4 language still “suffers” from one significant limitation: the voluminosity of its code. P4 applications easily reach thousands of lines of code, becoming hard to develop, debug, and maintain. The reason is two-fold: P4 requires many characters to express individual concepts (verbosity), and it relies on code repetition (lack of parametrization).

Today, P4 users overcome this limitation by relying on templating tools, hand-crafted scripts, and complicated macros. Unfortunately, these methods are not optimal: they make the development process difficult and do not generalize well.

In this work, we propose reducing the voluminosity of P4 code by introducing higher-level constructs to the language. We present O4, an extended version of P4, that includes three higher-level abstractions: arrays (which group same-type entities together), loops (which reduce simple repetitions), and factories (which enable code parametrization).

We evaluate O4 on several state-of-the-art programs and show how, with respect to P4: (i) it reduces code volumes by up to 80%, (ii) it decreases code verbosity by 44% on average, and (iii) it cuts duplicated code by 60%. We show how these benefits come with just a 3.5% increase in compilation time.

1 INTRODUCTION

P4 was proposed in 2014 as a high-level language to program the data planes of programmable network devices [22, 43]. Since its inception, the P4 language has been widely adopted, enabling innovation in numerous areas of networking: e.g., traffic engineering [35], telemetry [32], and security [31, 38].

Despite its continuous evolution (with new features and syntaxes introduced [2, 43]), the P4 language still “suffers” from one limitation: its voluminosity. Even simple P4 programs span thousands of lines of code. The origin of P4’s voluminosity is two-fold. First, the language is verbose: P4 primitives require more characters than their equivalents in other languages (cf. Listing 1). Second, P4 primitives are poorly parametrized, generating code duplicates (cf. Listing 2).

P4 language’s voluminosity convolutes the development process of P4-based applications: i.e., the writing, debugging, deployment, and maintenance phases. Indeed, voluminous code takes longer to be written, read and understood; larger codes have increased probability of containing errors [37]; and errors can propagate across multiple code repetitions.

Listing 1: Simple primitives, such as register updates, are more verbose in P4 than in other languages.

```c
// P4
RegisterAction<...>(my_register) inc = {
  void apply(...) { val = val + 1; ... }
};

// C++
(*reg)++;```

Listing 2: The lack of parametrization, e.g. in registers, leads to code duplication in P4 programs.

```c
Register<...>() my_register_1;
RegisterAction<...>(my_register_1) inc1 = {
  void apply(...) { val = val + 1; ... }
};

Register<...>() my_register_2;
RegisterAction<...>(my_register_2) inc2 = {
  void apply(...) { val = val + 1; ... }
};```

P4 users today are well aware of the consequences of P4’s voluminosity, and try to overcome it by relying on various tools. We surveyed a group of 27 P4 programmers and found that all of them rely on some tool to reduce P4’s voluminosity (cf. Figure 1). Some of them rely on copy-pasting code fragments, templating tools (e.g., [1, 25, 29]), or hand-crafted scripts. While these tools may solve the problem temporarily, they do not generalize well, being impractical in the long run. Others rely on pre-processor macros, which are too generic since they do not inherently convey the restrictions of P4.

```
How do you deal with P4’s voluminosity?

Hand-crafted Scripts 4
Templating Tools 8
Pre-processor Macros 14
Copy-pasting 24
```

Figure 1: P4 users leverage tools to reduce its volume.
Recent research works have also tried simplifying the P4 development process, by either directly fitting P4 programs into the target hardware resources [28, 29, 34, 40], or adding macro-like annotations into P4 code [39]. These approaches significantly increase P4’s expressivity (e.g., allowing functionalities well beyond the scope of P4, potentially giving programmers a false sense of what is possible), or abstract the low-level parameters that enable P4’s fine-tuning capabilities.

In the long term, the ideal solution would be to integrate a set of higher-level constructs—directly into the P4 language—to reduce its voluminosity in a target-agnostic manner. While reducing code volumes may be straightforward in other programming languages, in P4 it is challenging due to the need of preserving the language’s expressivity and fine-grained control. Indeed, any P4 extension needs to convey the constraints of programmable data planes (e.g., the lack of dynamic memory allocation and the lack of recursion), and to preserve the language’s fine-tuning capabilities for high performance.

**Our work.** In this paper, we propose the addition of three simple constructs to P4—arrays, loops, and factories—which we specifically design to match the strict requirements of programmable data planes. We show how these primitives reduce P4’s voluminosity significantly while preserving the language’s expressivity. At the high level, arrays group together variables of the same type, loops reduce code-block repetitions, and factories introduce code parameterization. The proposed abstractions are backwards compatible with existing P4 programs and can be easily adopted. We also present a compiler that translates programs in our higher-level P4 extension (which we call “O4”), into P4 programs.

**Performance.** We evaluate O4 on various applications and show how, compared to P4, it manages to reduce code volumes up to 80%, code verbosity by 44% on average, and duplicate code by 60% on average. Overall, O4 performs on par with state-of-the-art target-dependent languages such as P4All [34] while providing seamless integration with the original P4 language. Finally, we show how the proposed abstractions only increase by 3.5% the total compilation time.

**Contributions.** Our main contributions are:

- A set of language constructs that manage to reduce the code voluminosity of the P4 language (§2).
- A compiler, written in Racket, that transpiles O4 code to P4, and compiles the result into hardware code\(^1\) (§3).
- A comprehensive evaluation of O4, showing its effectiveness in reducing P4’s voluminosity, while only negligibly increasing the total compilation time (§4).

\(^1\)Code available at https://github.com/nsg-ethz/O4

## 2 HIGHER-LEVEL ABSTRACTIONS

In this section, we propose three new language abstractions—arrays, loops, and factories—that reduce P4’s voluminosity while preserving its expressivity. First, we introduce arrays, which group together same-type variables (§2.1). Then, we introduce loops, which reduce code-block repetitions (§2.2), and factories, which enable code parameterization (§2.3).

### 2.1 Arrays

The first type of code repetition that one can encounter in P4 programs is variable repetitions. In P4, programmers have to instantiate every new variable from scratch, regardless of whether they have already instantiated other variables of the same type (cf. Listing 3). For complex P4 programs, this process becomes tedious. In other programming languages, this problem has long been solved by introducing arrays. Arrays are data structures allowing the definition of multiple variables of the same type with a single instantiation [12, 13].

**Listing 3: Simple example for O4 arrays.**

```plaintext
// P4: variables
bit<32> a_0_0;
bit<32> a_0_1;
bit<32> a_1_0;
bit<32> a_1_1;

// O4: array
bit<32>[2][2] a;

// O4: array
bit<32>[2][2][2] a;
```

As of today, the P4 language does not support arrays yet: the closest abstractions are header stacks and tuple types, which are either limited to header fields or are not indexable.

Unfortunately, adding arrays to P4 is not straightforward. The main challenge is that the P4 language does not allow arbitrary memory allocation. We overcome this limitation by introducing fixed-length arrays, in which the length must be specified at compile time. During compilation, the O4 compiler translates each array declaration into individual P4 variable declarations (cf.§3). With this design, arrays preserve the P4-language’s expressivity (i.e., they neither narrow nor extend the functionality of P4). Indeed, each array directly corresponds to a group of same-type P4 variables, and every group of same-type P4 variables can be expressed as an array.

**Array types.** We define arrays as `type<width>[length]`. We support most P4 data types: all base types, specialized types, and extern-derived types [43] (e.g., `int`, `bit<@>` or `int<@>`). Arrays can be used wherever their types are allowed in P4, and are accessed using the subscript operator: `array[index]`. Both, array sizes and indices, have to be known at compile time. Multidimensional arrays are allowed.

**Array literals.** We also define array literals, i.e., arrays that are not associated with a name. Array literals can be useful e.g. when looping over indices in a loop, or when assigning initial values to an array. The O4 compiler directly maps them to simple P4 expressions (e.g., `[0, 1, 2]` in Listing 4).
2.2 Loops

The second type of code repetition that one can encounter in P4 programs is the repetition of code blocks. The lack of parametrization in P4 language makes repeated code blocks common within P4 programs. Other high-level programming languages usually reduce code-block repetitions by introducing loops [15, 16]. Loops allow programmers to compress repeated segments of code into a single instantiation.

The P4 language, however, does not currently support any iteration construct within its main control body (i.e., loops are only allowed within the P4 parsers). The main reason is that programmable data planes do not support general recursion. Indeed, performing a loop in the data plane would require sending the packet across the pipeline multiple times, which breaks the throughput guarantees. Even though recursive packet processing is not possible in programmable data planes, iterative data structures at the language level can significantly reduce code voluminosity. As such, the key challenge in introducing loops to P4 is to just do it at the language level (i.e., to reduce code-block repetitions), ensuring that they do not extend the P4 language’s expressivity (i.e., that they do not allow more computation than is supported).

We solve this challenge by introducing fixed-depth loops, where the loop’s depth needs to be specified at compile time. During compilation, the O4 compiler unrolls the declared loops, performing in-place replacement of the loop iterators to prevent introducing additional variables [17]. Specifically, we propose a for/in loop primitive, which aggregates a fixed set of code repetitions into a single concise instantiation (cf. Listing 4). This would not be the case for primitives such as while or do-while, which enable infinite loops. With this design, we restrict the expressive power of the loop abstraction and prevent arbitrarily long loops in P4, making it fit the constraints of programmable devices. The resulting loops preserve P4 language’s expressivity: any set of repeated P4 blocks can be represented by an O4 loop, and vice versa.

Listing 4: Simple example for O4 loops.

```p4
my_call(0); for (int index in [0, 1, 2]) my_call(index); my_call(2);
```

Loop support. We support loops in the control-blocks of P4 programs and provide support for nested loops. We define loops as for (type name in expression) statement. The expression can represent any P4 expression, but it can only support one iterating variable per loop (i.e., the compiler only accepts as input one-dimensional arrays). The loop iterator needs to be immutable. The body statement of the loop does not have to be a block statement, and, therefore, the compiler must guarantee that the scopes are not violated.

2.3 Factories

While arrays and loops already significantly reduce P4’s voluminosity, they cannot address its lack of parametrization. As of today, P4 provides limited parametrization support for a few primitives such as controls and actions, and no support for primitives such as tables and registers. As such, in order to e.g., declare multiple actions performing the same operations on (i) different registers, or (ii) multiple tables only differing in their key fields, one needs to declare them individually.

Other high-level programming languages enable code parametrization by introducing factories (or constructors). Factories allow modularizing code into library-like structures that the compiler can then expand at compile time [14, 18].

Introducing factory support for most P4 constructs (e.g., registers, tables, and externs) is intricate. The key challenge in doing so, once again, is to preserve expressivity. Indeed, a naive factory design can become a Turing-complete primitive subject to the halting problem [20]. We solve this problem by designing factories that can only call existing P4 primitives or other factories, which will eventually call existing P4 primitives (i.e., no transitive recursion). With this design, since factories do not introduce any additional program logic per se, we guarantee that they preserve P4 language’s expressivity.

Listing 5: Simple example for O4 factories.

```p4
// P4: repetitive table structure
control my_control(my_header hdr, ...) {
  table my_table_0 {
    key = { hdr.field_0: exact; }
    actions = { my_action; }
  }
  table my_table_1 {
    key = { hdr.field_1: exact; }
    actions = { my_action; }
  }
  apply {
    my_table_0.apply();
    my_table_1.apply();
  }
}

// O4: abstracts the table structure
control my_control(my_header hdr, ...) {
  factory my_factory(bit<8> field) {
    table my_table {
      key = { field: exact; }
      actions = { my_action; }
    } return my_table;
  }
  apply {
    for (int i in [0, 1]) {
      my_factory(hdr.field[i]).apply();
    }
  }
}
```
**Factory support.** We design O4 factories by borrowing concepts from object-oriented factories and from P4 constructors. We provide support for factories in actions, tables, and externs. We declare factories as: 

```o4
factory name(params) {body ... return body-name ;},
```

where body and params are their corresponding P4 counterparts. Factories allow the parametrization of any of their wrapped primitives (cf. Listing 5). Indeed, their body can include an action declaration, a table declaration, or an extern instantiation. O4 factories can call other factories, which allows O4 programs to follow the architectural principles of P4. For example, table factories can call action factories, which in turn can again call extern factories.

**Factory instantiation.** A factory declaration, by itself, is not functional. Similarly to P4 externs [43], after factories are declared, they have to be instantiated (or called). A factory instantiation looks like all P4 instantiations: `factory(args)`. When a factory is called with a set of arguments, the O4 compiler substitutes the factory call for its respective body instance, instantiated with the given arguments. If a factory is called twice with the same arguments, the compiler will instantiate two code segments with identical body. Same as for loops, factories are compiled using in-place replacement of their factory declarations by their body instances [17].

3 IMPLEMENTATION

In this section, we introduce the design of the O4 compiler. We implement the O4 compiler using Racket [10, 26, 27], a functional programming language to write compilers for domain-specific languages (DSLs). ²

The O4 compiler is composed of: a front end, and a back end (cf. Figure 2) [21]. The front end analyzes the input source code (in O4) and extracts an intermediate representation (IR) by performing lexical, syntactic, and semantic analysis on the input. This intermediate representation is then processed by the back end, which generates the target P4 code.

**Front end.** The compiler’s front end consists of two parts. First, a tokenizer processes the input source code (in O4) as a stream of characters and converts it into a stream of tokens. A token is just a tuple composed by a string and a label assigned to the string (e.g., a string can be labeled as an integer, a comment, a keyword, an identifier, or whitespace). Second, a parser takes as input these tokens, and verifies that their format matches the O4 grammar. The O4 grammar is a set of language rules in which we define what tokens we expect in O4, and in what order. As a result, the parser produces a data structure representation of the O4 code, as a nested tree of function calls, which can then be expanded by the compiler’s back end. In the O4 compiler, this representation follows the format of an abstract syntax tree (AST) [11], where each node in the tree corresponds to a rule in the O4 grammar. This AST representation returned by the parser is the intermediate representation (IR) of O4 (cf. Figure 2).

**Back end.** The compiler’s back end is responsible for expanding each of the nodes in the AST representation. First, it provides bindings to all nodes, describing how each node should be handled. Then, it expands the nodes by replacing the function calls by their P4 instantiations. The P4 code generation is performed in a distributed manner, with each node generating its own P4 codelets, before they are merged. The compiler’s back end is divided into multiple modules, each defining expansions for a number of symbols in the O4 grammar. For instance, the control backend expands all nodes related to the control blocks, including factories. The expansion process heavily utilizes Racket’s macro system, reshaping the IR, checking and rewriting all new O4 primitives to their equivalent P4 representations.

4 EVALUATION

We now evaluate our proposed solution on a set of state-of-the-art P4 program examples. First, we introduce our setup (§4.1). Second, we evaluate the performance of O4 in terms of volume reduction (§4.2), verbosity reduction (§4.3), code-clones reduction (§4.4), and finally compilation time (§4.5).

4.1 Test setup

We create a dataset of P4-16 programs, which cover a wide range of use-cases and program sizes. It is composed by programs from the P4-Learning tutorials [8], which run on the v1model architecture [9], and programs from the Open Tofino collection [7], which run on the tna architecture [5]. Before running the experiments, we apply a consistent formatting to all codes with a “pretty-printer”. We compare: (i) the original P4 code (including macros), denoted by P4*, and (ii) the P4 code with macros expanded (to not mix the effects of macros and O4), denoted by P4, to their hand-translated O4 counterparts, resulting in Diff* and Diff respectively.

---

²We leverage Racket to build a proof-of-concept compiler, which can show the benefits of O4. For production, we would directly modify the actual P4 compiler to support the proposed higher-level abstractions.
4.2 Volume reduction

We evaluate O4’s voluminosity reduction with respect to P4 by measuring their respective lines of code. Lines of Code (LOC) is a widely-used metric for measuring code volumes, which measures the number of physical lines of code in a file [19]. We compute LOC for P4 and O4 programs, ignoring the lines containing only whitespaces, comments, and braces.

Our evaluation shows that O4 decreases the LOC by $\approx 42\%$ and $\approx 44\%$ on average, with respect to P4* and P4. We also find that the effectiveness of O4 increases with the program size, and that O4 performs on par with more-complex higher-level state-of-the-art programming languages [28, 34, 39, 40].

<table>
<thead>
<tr>
<th>Program</th>
<th>P4*</th>
<th>P4</th>
<th>O4</th>
<th>Diff. * (%)</th>
<th>Diff. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>heavy_hitter</td>
<td>127</td>
<td>123</td>
<td>113</td>
<td>-11.0</td>
<td>-8.1</td>
</tr>
<tr>
<td>cm_sketch</td>
<td>125</td>
<td>125</td>
<td>89</td>
<td>-28.8</td>
<td>-28.8</td>
</tr>
<tr>
<td>loss_detect.</td>
<td>300</td>
<td>285</td>
<td>181</td>
<td>-39.7</td>
<td>-36.5</td>
</tr>
<tr>
<td>aes_one.</td>
<td>319</td>
<td>443</td>
<td>201</td>
<td>-37.0</td>
<td>-54.6</td>
</tr>
<tr>
<td>conquest</td>
<td>647</td>
<td>630</td>
<td>271</td>
<td>-58.1</td>
<td>-57.0</td>
</tr>
<tr>
<td>acc-turbo</td>
<td>1354</td>
<td>1352</td>
<td>270</td>
<td>-80.1</td>
<td>-80.0</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td>-42.4</td>
<td>-44.2</td>
</tr>
</tbody>
</table>

Table 1: O4 reduces the LOC of a program. The first three programs run on v1model, the last three on tna.

Contrary to the LOC results (cf. §4.2), the macros in P4* programs tend to help with verbosity. The macros help the most in the case where they also reduce LOC (aes-oneround). Nevertheless, all P4* programs are still more verbose than their O4 counterparts. Without the help of macros (P4), the difference is even more extreme. Again, the beneficial effect of O4 tends to get more emphasized with program size.

4.3 Verbosity reduction

We measure O4’s verbosity by using the Halstead volume [33], and observe an average reduction of 33% (P4*) and 44% (P4).

The Halstead volume measures program verbosity as $V = N \log_2 \eta$ [33], where: $N$ is the sum of all the operators and operands in the program, and $\eta$ is the sum of the distinct operators and operands in the program. We define the Halstead operators as the reserved characters and keywords of O4 and P4, respectively. Before computing the Halstead volume, we remove whitespaces, comments, and annotations.

<table>
<thead>
<tr>
<th>Program</th>
<th>P4*</th>
<th>P4</th>
<th>O4</th>
<th>Diff. * (%)</th>
<th>Diff. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>heavy_hitter</td>
<td>7.0</td>
<td>7.2</td>
<td>6.6</td>
<td>-5.7</td>
<td>-8.3</td>
</tr>
<tr>
<td>cm_sketch</td>
<td>5.0</td>
<td>7.3</td>
<td>4.9</td>
<td>-2.0</td>
<td>-32.9</td>
</tr>
<tr>
<td>loss_detect.</td>
<td>25.4</td>
<td>25.4</td>
<td>14.1</td>
<td>-44.5</td>
<td>-44.5</td>
</tr>
<tr>
<td>aes_one.</td>
<td>16.9</td>
<td>28.5</td>
<td>14.2</td>
<td>-16.0</td>
<td>-50.2</td>
</tr>
<tr>
<td>conquest</td>
<td>39.6</td>
<td>40.3</td>
<td>20.1</td>
<td>-49.2</td>
<td>-50.1</td>
</tr>
<tr>
<td>acc-turbo</td>
<td>93.3</td>
<td>92.1</td>
<td>20.3</td>
<td>-78.2</td>
<td>-78.0</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td>-32.6</td>
<td>-44.0</td>
</tr>
</tbody>
</table>

Table 2: O4 reduces the Halstead volume of a program.

Figure 3: O4 is more beneficial for large programs.

Figure 4: O4 performs on par with state-of-the-art.

We compare the performance of O4 with other high-level languages [28, 34, 39, 40], by gathering their published LOC measurements and computing their average reduction.
4.4 Code-clones reduction

We measure O4’s code-clone reduction with respect to P4, by using the Levenshtein distance [36]. We find that, on average, O4 reduces the number of clones by $\approx 56\%$ (P4*) or 60% (P4).

The Levenshtein distance [36] measures the minimum number of single-character edits required to make two strings equivalent, where edits can be insertions, deletions, and substitutions. We identify clones as every pair of lines which has a Levenshtein distance smaller or equal to a threshold $\theta$.

Since we want to minimize the number of false positives, we set $\theta = 1$. Moreover, we remove comments, trim whitespaces, and ignore empty lines and lines only containing braces.

[Table 3: O4 reduces the percentage of code clones.]

<table>
<thead>
<tr>
<th>Program</th>
<th>P4*</th>
<th>P4</th>
<th>O4</th>
<th>Diff* (%)</th>
<th>Diff. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>heavy_hitter [4]</td>
<td>7.1</td>
<td>7.3</td>
<td>7.1</td>
<td>-0.1</td>
<td>-3.2</td>
</tr>
<tr>
<td>cm_sketch [3]</td>
<td>18.4</td>
<td>18.4</td>
<td>4.5</td>
<td>-75.6</td>
<td>-75.6</td>
</tr>
<tr>
<td>loss_detect. [6]</td>
<td>32.3</td>
<td>32.6</td>
<td>12.7</td>
<td>-60.7</td>
<td>-61.1</td>
</tr>
<tr>
<td>aes_oneround [23]</td>
<td>29.2</td>
<td>50.1</td>
<td>10.9</td>
<td>-62.5</td>
<td>-78.2</td>
</tr>
<tr>
<td>conquest [24]</td>
<td>52.7</td>
<td>53.8</td>
<td>15.5</td>
<td>-70.6</td>
<td>-71.2</td>
</tr>
<tr>
<td>acc-turbo [31]</td>
<td>77.5</td>
<td>77.7</td>
<td>24.4</td>
<td>-68.5</td>
<td>-68.6</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
<td>-56.3</td>
<td>-59.6</td>
</tr>
</tbody>
</table>

In Table 3, we observe that O4 reduces the number of code clones for all test programs. Contrary to the volume and verbosity tests (cf. §4.2 and §4.3), we see a similar percentual reduction for small and large programs. This shows that even for smaller programs, containing fewer repetitions in absolute terms, O4 is still effective in reducing duplicates.

4.5 Compilation time

Even though compiler efficiency is not a main goal of our work, we want to show that it is both feasible and practical to compile our new abstractions to P4. We show that, on average, the O4 compiler adds only 3.5% to the total compilation time, when compiling an O4 program to a Tofino target [5]. We measure the O4 and P4 compilation times with UNIX’s time command. All reported times are averaged over three measurements. We observe that the O4 compiler translates each O4 test program to P4 in a few seconds (cf. Table 4).

[Table 4: O4 keeps a small compilation time.]

<table>
<thead>
<tr>
<th>Program</th>
<th>Compilation Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>heavy_hitter [4]</td>
<td>1.46 N/A</td>
</tr>
<tr>
<td>cm_sketch [3]</td>
<td>1.37 N/A</td>
</tr>
<tr>
<td>loss_detection [6]</td>
<td>2.01 N/A</td>
</tr>
<tr>
<td>aes_oneround [23]</td>
<td>2.01 131.4</td>
</tr>
<tr>
<td>conquest [24]</td>
<td>2.76 129.8</td>
</tr>
<tr>
<td>acc-turbo [31]</td>
<td>3.12 46.0</td>
</tr>
</tbody>
</table>

In Table 4, we observe that O4 keeps a small compilation time.

5 RELATED WORK

Macro-based languages  P4All [34] adds elastic data structures to P4, to automatically fit P4 programs to hardware. pcube [39] proposes annotation primitives to synchronize state variables across multiple P4 switches. Both P4All and pcube include language constructs similar to O4, such as arrays and loops. However, the purpose of their usage is radically different. O4 is the first work that focuses squarely on reducing P4 language’s voluminosity, proving empirically the benefit of using such language constructs in reducing P4-program sizes. Further, O4 introduces a new construct–factories—which has not been used by either P4All nor pcube.

Synthesized languages  Domino [40] and Chipmunk [30] automatically compile packet-processing specifications into target-specific code. Lyra [28] generalizes this idea to allow concurrent execution of P4 programs across multiple devices. These languages have different design goals than O4, not trying to extend P4, but directly replacing it by higher-level program specifications. In doing so, they abstract the low-level hardware parameters that facilitate fine-tuning in P4, and increase the language expressivity. Further, they require users to learn a new language that is broadly different from P4.

Modularization languages  Languages such as $\mu$P4 [42], ClickP4 [44], or Lucid [41] aim at increasing the modularity in P4, often by changing its architecture model. Thus, while they may produce concise code, they are orthogonal to O4.

6 CONCLUSION

We presented O4, a lightweight extension of the P4 language that reduces code voluminosity by just introducing three higher-level abstractions: arrays, loops, and factories. We show that these basic abstractions already manage to reduce code volumes, code verbosity, and code duplicates at the level of complex state-of-the-art languages such as Lyra [28] and P4All [34]. O4 does so while preserving the language’s expressivity and fine-tuning capabilities. We also show that O4 only increases the overall compilation time marginally.
REFERENCES