Reducing P4 Language's Voluminosity using Higher-Level Constructs

Albert Gran Alcoz[*]
ETH Zürich

Coralie Busse-Grawitz[*]
ETH Zürich

Eric Marty
ETH Zürich

Laurent Vanbever
ETH Zürich

Figure 1: Pairwise comparison of line count for original P4 programs (blue) and their O4 counterparts (green).

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 twofold: 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 beyond one codebase.

In this work, we propose reducing the voluminosity of P4 code by introducing higher-level language constructs. We present O4, an extended version of P4, that includes *three* such constructs: *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 contribute a compiler implementation that provides said benefits with just a 3.5% increase in compilation time.

CCS CONCEPTS

• **Networks** → **Programmable networks**; • **Software and its engineering** → **Domain specific languages**.

KEYWORDS

P4 Language, Programmable Networks, Programming Languages

ACM Reference Format:
Albert Gran Alcoz, Coralie Busse-Grawitz, Eric Marty, and Laurent Vanbever. 2022. Reducing P4 Language's Voluminosity using Higher-Level Constructs. In *P4 Workshop in Europe (EuroP4 '22), December 9, 2022, Roma, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3565475.3569078

1 INTRODUCTION

P4 was proposed in 2014 as a high-level language to program the data planes of programmable network devices [23, 44]. Since its inception, the P4 language has been widely adopted, enabling innovation in numerous areas of networking: e.g., traffic engineering [36], telemetry [33], packet scheduling [22], and security [32, 39].

Despite its continuous evolution (with new features and syntax introduced [2, 44]), 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 programs have increased probability of containing errors [38]; and errors can propagate across multiple code repetitions. The latter are particularly hard to catch when the repetition is *implicit*, i.e. not a one-to-one copy but rather the repetition of an algorithmic assumption or structure.

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

```
// 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.

```
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; ... }
};
```

[*]The first two authors contributed equally to this research.

EuroP4 '22, December 9, 2022, Roma, Italy
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9935-7/22/12...$15.00
https://doi.org/10.1145/3565475.3569078

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 2). Some of them rely on copy-pasting code fragments, templating tools (e.g., [1, 26, 30]), 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.

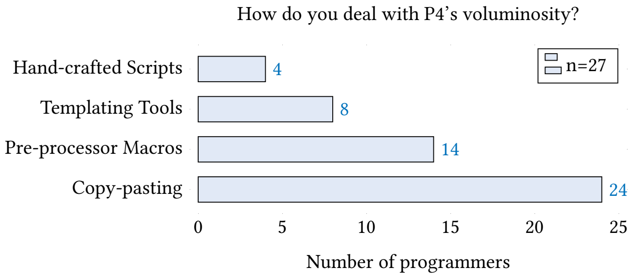

Figure 2: P4 users leverage tools to reduce its volume.

Recent research works have also tried simplifying the P4 development process, by either directly fitting P4 programs into the target hardware resources [29, 30, 35, 41], or adding macro-like annotations into P4 code [40]. These approaches either significantly increase P4's expressivity (e.g., allowing functionalities well beyond the scope of P4, potentially giving programmers a false sense of what is computable on network hardware); or they abstract away 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 of recursion), and to preserve the language's fine-tuning capabilities for high performance.

Our work. 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. *Arrays* group together variables of the same type, *loops* reduce code-block repetitions, and *factories* introduce code parametrization. We show how these primitives reduce P4's voluminosity significantly while preserving the language's expressivity. These abstractions are backward-compatible with existing P4 programs and can easily be adopted. We also present a compiler that translates programs in our higher-level P4 extension (which we call "O4"), into P4 programs.

While previous works [35, 40] include language constructs similar to O4, such as arrays and loops, the purpose of their usage is radically different. Indeed, O4 is the first work that focuses *squarely* on reducing P4 language's voluminosity, empirically proving the benefit of its constructs to this end. Further, O4 introduces a new construct–*factories*–which was not proposed by previous works.

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 [35] while providing seamless integration with the original P4 language. 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).

2 HIGHER-LEVEL ABSTRACTIONS

In this section, we propose three new language abstractions–*arrays*, *loops*, and *factories*–that reduce the voluminosity of P4 while preserving its expressivity. First, we introduce *arrays*, which group together variables of the same type (§2.1). Second, we introduce *loops*, which reduce repetitions of code blocks (§2.2). Finally, we introduce *factories*, which enable code parameterization (§2.3).

2.1 Arrays

The first type of 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.

```
// P4: variables          // O4: array
bit<32> a_0_0;            bit<32>[2][2] a;
bit<32> a_0_1;
bit<32> a_1_0;
bit<32> a_1_1;
```

As of today, the P4 language does not support arrays yet: the closest supported abstractions are header stacks and tuple types. These primitives 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): 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.

[1] Code available at https://github.com/nsg-ethz/O4

Array types. We define arrays as **type**`<width>[length]`. We support most P4 data types: all base types, specialized types, and extern-derived types [44] (e.g., **int**, **bit**`<W>` or **int**`<W>`). Arrays can be used wherever their types are allowed in P4, and are accessed using the subscript operator: `array[index]`. 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 variable name. Array literals can be useful for "single-use" arrays (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 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 code segments 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 in 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 *only* do this at the language level (i.e., to reduce code-block repetitions), ensuring that they do not extend the expressivity of the P4 language (i.e., that they do not allow more computation than is actually 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 the expressivity of the P4 language: any set of repeated P4 blocks can be represented by an O4 loop, and vice versa.

Listing 4: Simple example for O4 loops.

```
// P4:          // O4: loops
my_call(0);     for (int index in [0, 1, 2])
my_call(1);        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. Hence, 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*. For instance, multiple actions performing the same operations on (i) different registers, or (ii) multiple tables that only differ in their key fields, need to be declared individually.

Other high-level programming languages enable code parametrization by introducing *factories* (or constructors). With factories, one can modularize code into library-like structures (e.g., a parametrized sketch) 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 is, once again, preserving expressivity. Indeed, a naive factory design can become a Turing-complete primitive, thus subject to the halting problem [20] and violating throughput guarantees. 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). Thanks to this, factories do not introduce any additional program logic *per se*. Accordingly, we can guarantee that they preserve P4 language's expressivity.

Listing 5: Simple example for O4 factories.

```
// 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; }
   }
}

// 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;
   }
   my_table_0 = my_factory(hdr.field_0);
   my_table_1 = my_factory(hdr.field_1);
}
```

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: **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 [44], 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 an identical body. Analogously to loops, factories are compiled using *in-place* replacement of their factory declarations with their body instances [17].

3 IMPLEMENTATION

We now introduce the design of the O4 compiler. We implement the O4 compiler using Racket [10, 27, 28][2], a functional programming language to write compilers for domain-specific languages (DSLs).

The O4 compiler is composed of a *front end* and a *back end* (cf. Figure 3) [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. The back end then processes this IR in order to generate the target P4 code.

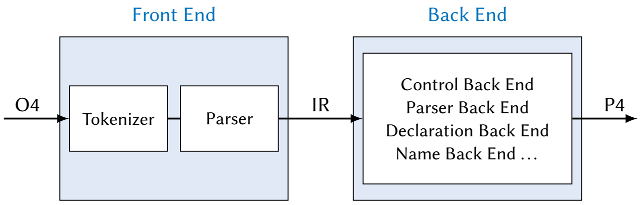

Figure 3: The O4 compiler uses a two-stage design.

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 (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 that represents 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 3).

Back end. The compiler's back end is responsible for expanding each node 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 is generated 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 back end 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

In this section, we evaluate our proposed language primitives 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), and code-clones reduction (§4.4). Finally, we evaluate O4's compilation time (§4.5).

4.1 Test setup

We create a dataset of P4-16 programs, covering a wide range of use-cases and program sizes. It is composed by programs from the P4-Learning tutorials [9], which run on the *v1model* architecture [8], 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 codebases 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 differences *Diff** and *Diff* respectively.

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 code volume that measures the number of physical lines of code in a file [19]. We compute LOC for the P4 programs with and without expanding macros, and their O4 counterparts. We ignore the lines with only whitespaces, comments, and braces.

Our evaluation shows that O4 decreases the LOC by ≈ 42% and ≈ 44% on average with respect to P4* and P4, respectively. We also find that the effectiveness of O4 increases with the program size, and that O4 performs on par with the current state of the art on more complex higher-level programming languages [29, 35, 40, 41].

Table 1 details O4's LOC-reduction for all test examples. Interestingly, the macro expansion from P4* to P4 often *reduces* LOC. This is the case because macros are often used to define constants. Directly inserting them reduces the LOC, however it hurts the easy adaptability of the code. Even in the single case where macros help (the *aes-oneround* program), O4 improves the LOC by ≈ 37%.

[2] We 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.

	LOC				
Program	P4*	P4	O4	Diff.* (%)	Diff. (%)
heavy_hitter [4]	127	123	113	-11.0	-8.1
cm_sketch [3]	125	125	89	-28.8	-28.8
loss_detect. [6]	300	285	181	-39.7	-36.5
aes_one. [24]	319	443	201	-37.0	-54.6
conquest [25]	647	630	271	-58.1	-57.0
acc-turbo [32]	1354	1352	270	-80.1	-80.0
Unweighed average				-42.4	-44.2

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

We also note that the reduction in LOC seems to increase with the size of the program. This is interesting, given that real-world programs running in production tend to have higher LOC counts.

We further analyze this relationship in Figure 4. We see an upwards trend in LOC reduction with increasing program size. This behavior could be explained by the fact that smaller files contain fewer repetitions (cf. §4.4), which makes our new abstractions less efficient. However, given that we have only investigated a handful of programs, we cannot make stronger claims beyond this general trend.

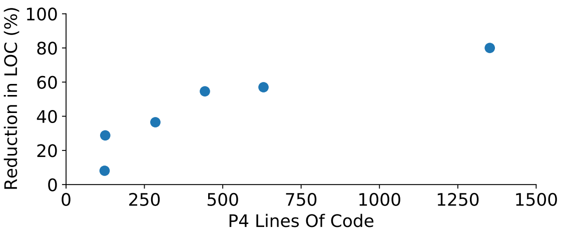

Figure 4: O4 is more beneficial for large programs.

We compare the performance of O4 with other high-level languages [29, 35, 40, 41]. We do this by gathering their published LOC measurements and computing their average LOC reduction.

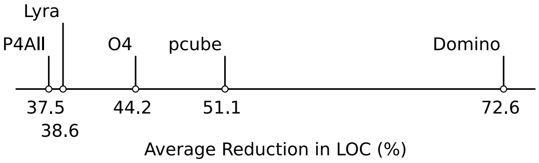

Figure 5: O4 performs on par with state-of-the-art.

We see that O4 performs similarly to state-of-the-art higher-level languages, only being significantly outperformed by Domino [41] (cf. Figure 5). These results have to be taken with a grain of salt, given that not all related work evalutes the same set of programs.

4.3 Verbosity reduction

We measure the verbosity of O4 with the Halstead volume [34]. We observe an average reduction of ≈ 33% (P4*) and ≈ 44% (P4).

The Halstead volume is a metric that measures program verbosity as $V = N \log_2 \eta$ [34], where: N is the sum of *all* the operators and operands in the program, and η 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 on a given program, we remove all contained whitespaces, comments, and annotations.

	Halstead Volume (10^3)				
Program	P4*	P4	O4	Diff.* (%)	Diff. (%)
heavy_hitter [4]	7.0	7.2	6.6	-5.7	-8.3
cm_sketch [3]	5.0	7.3	4.9	-2.0	-32.9
loss_detect. [6]	25.4	25.4	14.1	-44.5	-44.5
aes_one. [24]	16.9	28.5	14.2	-16.0	-50.2
conquest [25]	39.6	40.3	20.1	-49.2	-50.1
acc-turbo [32]	93.3	92.1	20.3	-78.2	-78.0
Unweighed average				-32.6	-44.0

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

In contrast to the LOC results (cf. §4.2), the macros in P4* programs tend to help with verbosity compared to the P4 counterparts. 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. As in §4.2, the beneficial effect of O4 tends to get more emphasized with increasing program size.

4.4 Code-clones reduction

We measure O4's code-clone reduction with respect to P4* and P4 by using the Levenshtein distance [37]. We find that, on average, O4 reduces the number of clones by ≈ 56% (P4*) and ≈ 60% (P4).

The Levenshtein distance [37] 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 with a Levenshtein distance smaller or equal to threshold θ. Since we want to minimize the number of false positives, we set $\theta = 1$. We remove comments, trim whitespaces (incl. line breaks), and ignore lines only containing braces.

Usually, the number of clones in a program is given as a percentage of the program containing the duplicates (denoted by %LOC). Accordingly, a 50 %LOC means that half of the lines of the program are duplicates of (at least) one other line in the same program.

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

	Levenshtein (%LOC)				
Program	P4*	P4	O4	Diff.* (%)	Diff. (%)
heavy_hitter [4]	7.1	7.3	7.1	-0.1	-3.2
cm_sketch [3]	18.4	18.4	4.5	-75.6	-75.6
loss_detect. [6]	32.3	32.6	12.7	-60.7	-61.1
aes_one. [24]	29.2	50.1	10.9	-62.5	-78.2
conquest [25]	52.7	53.8	15.5	-70.6	-71.2
acc-turbo [32]	77.5	77.7	24.4	-68.5	-68.6
Unweighed average				-56.3	-59.6

Table 3: O4 reduces a program's percentage of code clones.

	Compilation Time (s)	
Program	O4 -> P4	P4 -> Tofino
heavy_hitter [4]	1.46	N/A[1]
cm_sketch [3]	1.37	N/A[1]
loss_detection [6]	2.01	N/A[1]
aes_oneround [24]	2.01	131.4
conquest [25]	2.76	129.8
acc-turbo [32]	3.12	46.0

Table 4: O4 keeps a small compilation time.

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 see that the O4 compiler translates each O4 test program to P4 in a few seconds (cf. Table 4).

5 RELATED WORK

Macro-based languages P4All [35] adds elastic data structures to P4, to automatically fit P4 programs to hardware. pcube [40] 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 the voluminosity of the P4 language, empirically proving the benefit of using such language constructs for reducing P4-program sizes. Furthermore, O4 introduces a new construct–*factories*–which was neither used by P4All nor pcube.

Synthesized languages Domino [41] and Chipmunk [31] automatically compile packet-processing specifications into target-specific code. Lyra [29] 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 different from P4.

Modularization languages Other programming languages such as μP4 [43], ClickP4 [45], or Lucid [42] aim at increasing the modularity in P4, often by changing its architecture model. Therefore, 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 [29] and P4All [35]. O4 does so while preserving P4 language's expressivity and fine-tuning capabilities. We show that O4 only increases the overall compilation time marginally.

[1] It uses the v1model architecture, and can not be compiled to Tofino.

REFERENCES

[1] 2007. Jinja Documentation. https://jinja.palletsprojects.com/.

[2] 2020. P4 Language Specifications. https://p4.org/specs/.

[3] 2021. Count-Min Sketch Exercise, P4-Learning (Pulled: 22 September 2022). https://github.com/nsg-ethz/p4-learning/tree/master/exercises/07-Count-Min-Sketch.

[4] 2021. Heavy-Hitter Detection Exercise, P4-Learning (Pulled: 22 September 2022). https://github.com/nsg-ethz/p4-learning/tree/master/exercises/06-Heavy_Hitter_Detector.

[5] 2021. Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[6] 2021. Loss Detection Exercise, P4-Learning (Pulled: 22 September 2022). https://github.com/nsg-ethz/p4-learning/tree/master/exercises/11-Packet-Loss-Detection.

[7] 2021. Open Tofino (Pulled: 22 September 2022). https://github.com/barefootnetworks/Open-Tofino.

[8] 2021. P4 Compiler (Pulled: 22 September 2022). https://github.com/p4lang/p4c.

[9] 2021. P4-Learning (Pulled: 22 September 2022). https://github.com/nsg-ethz/p4-learning.

[10] 2021. Racket Language. https://racket-lang.org.

[11] 2022. Abstract Syntax Tree. https://en.wikipedia.org/wiki/Abstract_syntax_tree).

[12] 2022. Arrays in Java. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html.

[13] 2022. Arrays in Python. https://docs.python.org/3/library/array.html.

[14] 2022. Constructors in Object-Oriented Programming. https://en.wikipedia.org/wiki/Constructor_(object-oriented_programming).

[15] 2022. Control Flow Statements in Java. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html.

[16] 2022. Control Flow Tools in Python. https://docs.python.org/3/tutorial/controlflow.html.

[17] 2022. In-Place Algorithm. https://en.wikipedia.org/wiki/In-place_algorithm.

[18] 2022. Python Constructors. https://www.javatpoint.com/python-constructors).

[19] 2022. Source Lines of Code. https://en.wikipedia.org/wiki/Source_lines_of_code).

[20] 2022. The Halting Problem. https://en.wikipedia.org/wiki/Halting_problem.

[21] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

[22] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. SP-PIFO: Approximating Push-in First-out Behaviors Using Strict-priority Queues. In *USENIX NSDI*. Santa Clara, CA, USA.

[23] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming Protocol-independent Packet Processors. (2014).

[24] Xiaoqi Chen. 2020. Implementing AES Encryption on Programmable Switches via Scrambled Lookup Tables. In *ACM SPIN*. Virtual Event.

[25] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. 2019. Fine-Grained Queue Measurement in the Data Plane. In *ACM CoNEXT*. Orlando, Florida, USA.

[26] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. 2020. Beaucoup: Answering Many Network Traffic Queries, One Memory Update At a Time. In *ACM SIGCOMM*. 226–239.

[27] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *SNAPL*. Asilomar, CA, USA.

[28] Matthew Flatt and PLT. 2021. The Racket Reference. https://docs.racket-lang.org/reference/index.html.

[29] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *ACM SIGCOMM*. Virtual.

[30] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2020. Switch code generation using program synthesis. In *ACM SIGCOMM*. 44–61.

[31] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2020. Switch Code Generation Using Program Synthesis. In *ACM SIGCOMM*. Virtual.

[32] Albert Gran Alcoz, Martin Strohmeier, Vincent Lenders, and Laurent Vanbever. 2022. Aggregate-Based Congestion Control for Pulse-Wave DDoS Defense. In *ACM SIGCOMM*. Amsterdam, The Netherlands.

[33] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven Streaming Network Telemetry. In *ACM SIGCOMM*. Budapest, Hungary.

[34] Maurice H Halstead. 1977. *Elements of Software Science.* Elsevier North-Holland.

[35] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. 2020. Elastic Switch Programming with P4All. In *ACM HotNets*. Virtual.

[36] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast Connectivity Recovery Entirely in the Data Plane. In *USENIX NSDI*. Boston, MA, USA.

[37] Vladimir I Levenshtein et al. 1966. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In *Doklady Physics*. Soviet Union.

[38] Steve McConnell. 2004. *Code Complete.* Pearson Education.

[39] Roland Meier, Petar Tsankov, Vincent Lenders, Laurent Vanbever, and Martin Vechev. 2018. NetHide: Secure and Practical Network Topology Obfuscation. In *USENIX Security*. Baltimore, MD, USA.

[40] Rinku Shah, Aniket Shirke, Akash Trehan, Mythili Vutukuru, and Purushottam Kulkarni. 2018. pcube: Primitives for Network Data Plane Programming. In *IEEE ICNP*. Cambridge, UK.

[41] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-level Programming for Line-rate Switches. In *ACM SIGCOMM*. Florianópolis, Brazil.

[42] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: A Language for Control in the Data Plane. In *ACM SIGCOMM*. Virtual.

[43] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. 2020. Composing Dataplane Programs with μP4. In *ACM SIGCOMM*. Virtual.

[44] The P4 Language Consortium. 2021. P4-16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.2.2.html.

[45] Yu Zhou and Jun Bi. 2017. ClickP4: Towards Modular Programming of P4. (2017).