Acta Cybernetica **27** (2025) 53–65.

P4 Specific Refactoring Steps*

Máté Tejfel[ab], Dániel Lukács[ac], and Péter Hegyi[ad]

Abstract

P4 is a domain-specific programming language for programmable switches running inside next-generation computer networks. The language is designed to use the software defined networking (SDN) paradigm which separates the data plane and the control plane layers of the program. The paper introduces tool-supported refactoring steps for P4. The challenge in this task is that P4 has special domain-specific constructs that cannot be found in other languages and as such there is no existing methodology yet for refactoring these constructs. The proposed steps are implemented using P4Query, an analyzer framework dedicated to P4.

Keywords: P4 language, refactoring steps

1 Introduction

P4 [2] is a domain-specific programming language which enables a new approach to programming computer networks. It adopts the software defined networking (SDN) paradigm [10] which separates the data plane and the control plane layers of the program. P4 focuses on the data plane, while we need some other tool to create the control plane. It facilitates the implementation of the concept of fully programmable networks making possible the development of programmable data planes.

The paper introduces tool-supported refactoring steps for P4 with two-fold objectives. On one hand, we aim to assist developers to take full advantage of the programmability of P4, by providing standard refactoring services commonly found in IDEs of modern high-level languages. On the other hand, we want to enable P4 code optimizations that are aware of the unique make-up of this language.

*This research is in part supported by the project no. FK_21 138949, provided by the National Research, Development and Innovation Fund of Hungary. The research was partly supported by Ericsson Hungary.

[a]Faculty of Informatics, Eötvös Loránd University, ELTE, Budapest, Hungary
[b]E-mail: matej@inf.elte.hu, ORCID: 0000-0001-8982-1398
[c]E-mail: dlukacs@inf.elte.hu, ORCID: 0000-0001-9738-1134
[d]E-mail: immsrb@inf.elte.hu

DOI: 10.14232/actacyb.308085

The difficulty of the task is that P4 has many unique language constructs, for which there is no existing methodology for refactoring. One example is the declaration and application of lookup tables. As these components are performance-critical, their definitions are target-dependent: compilers have to consider the capabilities of their target architectures, and choose where to map tables to get the most efficient outcomes. Unfortunately, this is less straightforward than it looks, in particular because the intended content of the tables is unknown at compile-time, and so is the traffic that will be processed by these tables. As such, performing such optimizations is usually out of the scope of standard compilers, and it falls on specialized optimization tools, or maybe the well-informed developer, who can take into account these application-specific factors. We expect neither of these approaches to be prepared for handling machine-specific representations, and so it is preferable to perform the optimizing code transformation on the highest level, that is, directly on the P4 code.

1.1 Related work

There are currently many different tools that support the development of P4 programs. Some of them concentrate on error checking of P4 programs. For example, BF4 [3] is created as a P4C backend, which can not only detect error possibilities, but it is able to repair them by adding new keys to the lookup tables of the program and modify the table contents. Another tool p4-data-flow [1] uses data flow analysis to detect potential bugs in P4 switch codes.

Some other tools have been created for different purposes. For example, p4pktgen [9] uses symbolic execution for automatically generating test cases. SafeP4 [4] is a language which has precise semantics and a static type system that can be used to obtain guarantees about the validity of all headers which are used or modified by the program. The type checker of the language (P4Check) can also check P4 programs executing some static analysis on them.

P4 refactorings can be particularly useful for dataplane disaggregation, a problem recently addressed by Flightplan [13]. The objective here is to optimally segment P4 programs so that individual program segments can be assigned to resources, in turn transforming P4 into a programming language for the "one big switch" networking model. Flightplan can solve the allocation problem, and refactoring can help with the realization of program splitting, moreover, semantics-preserving program transformations may allow new, previously unseen forms of segmentations, enabling further optimization of allocations.

Another potential application for P4 refactorings can be found in [8]. The authors apply the normal form concept of relational database theory to lookup tables in P4 programs. Due to dependencies between their fields, large tables often contain significant amount of redundancy. By recognizing these dependencies, tables can be decomposed into smaller, irredundant tables. This irredundant representation is called a normal form, and normalization can be realized by vertically splitting tables. The authors find that on many targets normalization leads to better efficiency, because smaller, simpler tables can be updated by the controller with less work,

and because it is easier for compilers to find optimal representations for simpler tables.

The refactorings presented in this paper build on P4Query [7], whose program graph representation was mostly inspired by [5], a similar, but more established static analysis tool for Erlang, also developed at ELTE. A key idea in RefactorErl is that persisting pre-calculated semantic information in a database can both simplify and speed up refactorings. This also enables incremental refactorings where syntactical changes automatically trigger semantic analysis. Beyond being a tool for Erlang developers, RefactorErl is also a framework aiming to support quick and correct implementation of new refactorings.

2 P4 language

Every P4 program contains at least three main components: these are the parser, the match/action pipeline and the deparser [11]. The parser reads a packet from the network (as a bitstream) and builds up its header structure and its metadata information (while leaving the payload part of the packet unchanged). The match/action pipeline can read and modify the headers and the metadata (add, delete headers, modify header fields or metadata). While the deparser part creates the new packet using the original payload (for example, by changing the order of headers or omitting headers). This new packet will be sent forward on the network.

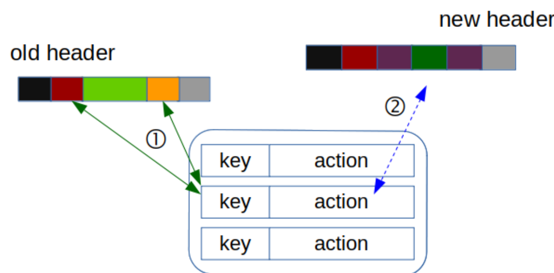

Figure 1: Match/Action table application

One of the most important part of the execution of a P4 program is the application of the match/action tables[1]. Figure 1 illustrates a match/action table application. The different colours in the headers represent different fields, which may vary in size. Every row in the table contains a key and an action part. The key refers to some fields of the header structure. The table application – when processing a specific packet – first searches the appropriate row in the table based on the concrete field values of the headers using a given lookup algorithm (exact, longest prefix match or ternary lookup). If the algorithm finds the appropriate

[1]More information about P4 can be found on the official website: `https://p4.org/`.

row, it will execute the action part of the row, which will modify the headers of the packet. If no appropriate line exists, the program executes the default action of the table. If the default action is not defined and no entry matches, then the table does not affect the packet.

It is worth mentioning that a P4 program defines only the data plane layer of a packet processing algorithm, namely it will define only the structure of match/action tables. Listing 1 introduces an example declaration of a match/action table in P4.

The declaration defines the key fields, the used lookup mechanisms, the possible actions, the maximum size (maximum number of rows), the default action and the const entries (the explicitly defined rows) of the table. The last three are optional. Specific data in the table (which actions will be executed for which field values) are specified by the control plane layer of the algorithm which is out of the scope of the P4 program.

```
table ipv4_lpm {
    key = {
        hdr.ipv4.srcAddr: exact;
        hdr.ipv4.dstAddr: lpm;
    }
    actions = {
        ipv4_forward;
        modify_dst;
        drop;
        NoAction;
    }
    size = 1024;
    default_action = drop();
    const entries = {
      ...
    }
}
```

Listing 1: Example of a match/action table declaration in P4

3 P4Query

The refactorings were realized with P4Query [7], a static analysis framework for P4[2]. The framework is centered around an extensible internal graph representation where the results of the different static analysis methods are stored also as part of the graph. The information in the knowledge graph is accessed using graph queries written in the Gremlin query language [12]. This way the framework guarantees unique standard representation both for the stored data, and for the data access mechanism. As the graph instance is detached from the code analysis, users and developers alike can access it by external tools for visualising, monitoring and

[2]The P4Query framework is available in GitHub: https://github.com/P4ELTE/P4Query.

validating purposes. Each static analysis has to declare which other static analyses it depends on, and a central component ensures that all analyses are executed in order, and without collisions. This also ensures that only necessary analyses are performed.

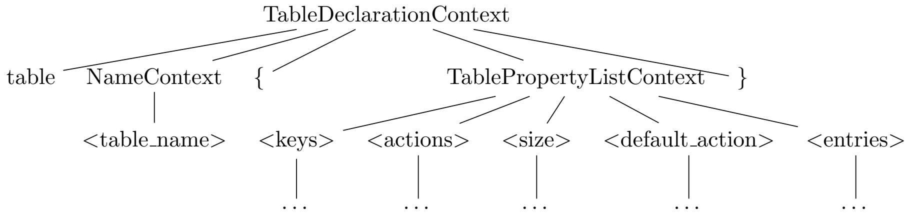

Figure 2: Representation of a match/action table in P4Query

Figure 2 illustrates the representation of a match/action table in P4Query. As the steps analyse and manipulate this representation, they can use the built-in analyzers of the tool to make checking the prerequisites much more easier.

4 Refactoring steps

In this section, we present the refactorings we defined for P4. Ultimately, refactorings are transformations of the syntax tree, satisfying the assumption that the resulting tree is semantically equivalent to the original one. Due to the complexity of the refactorings, circumspection must be exercised when executing the transformation, requiring to a considerable overhead. The general scheme of the refactoring is depicted by Algorithm 1, executing an R set of refactorings over a P4Query program graph G.

As we will see, more complex transformations often have various preconditions: for example, splitting a match/action table by its keys naturally requires a table with at least two keys. Thus, the first step of the algorithm is to check such transformation-specific preconditions. If the precondition is not satisfied, it does not make sense to start the refactoring. As P4 match/action tables are usually filled at runtime (by the SDN controller), it may be impossible to check all preconditions at analysis-time: in this case, the user should be warned that some preconditions could not be checked and it will be the responsibility of the user to ensure that the P4 program is being executed in the right environment. Additionally, the user should be subsequently given an option to cancel the refactoring, if they cannot guarantee this.

Checking the semantic equivalence between input and output syntax tree is one of the most important properties regarding implementation correctness. Unfortunately, exhaustive checking is not feasible due to the halting problem, which means we have to resort to non-exhaustive testing. While post-compile-time testing is the standard method for ensuring implementation correctness, it is difficult

Algorithm 1 General scheme of refactorings

Procedure Refactor(R, G):
Input: R is a list of refactorings
Input: G is graph, includes AST
Result: G conditionally transformed by refactorings in R

1: **for all** $r \in R$ **do**
2: **if** $\neg$ r.CheckPreconditions(G) **then**
3: Exit(*"Preconditions failed:"*, r.FailedPreconditions(G))
4: **end if**
5: **if** r.HasExternPreconditions(G) **then**
6: Warn(*"Preconditions could not be checked:"*, r.FailedPreconditions(G))
7: **end if**
8: G.StartTransaction()
9: r.Execute(G)
10: **if** $\neg$CheckConsistency(G) **then**
11: Warn(*"Consistency test failed, reverting."*)
12: G.RollbackTransaction()
13: **else**
14: G.FinishTransaction()
15: **end if**
16: **end for**

to sufficiently test complex systems with varied runtime parameters. For example, the graph backend under P4Query can be switched with relative ease, but this may have unexpected effects due to differences between graph backend implementations. Another issue could be parallelization (not yet a feature of P4Query, but could be a feature in graph backends) that – due to non-determinism – can also have unexpected effects. For this reason, the algorithm supports testing graph consistency after a refactoring was executed: if the test fails, the transformation is reverted so that the graph is left in a consistent state.

One of the promises of using a graph database in P4Query was that the database provides built-in support for operations such as atomic transactions and transaction rollback. Unfortunately, this is not always the case in practice: in the implementation, we had to implement rollbacks manually, since the in-memory database engine version (TinkerPop 3.4.4) used by default in P4Query does not support these features.

With having the general outline discussed, we can now focus on individual refactorings.

4.1 Table structure modification

The P4 program can modify the incoming packets (namely the headers) by applying match/action tables. For a given program, determining the optimal table structure is a difficult task. It can often be the case that in a given hardware environment

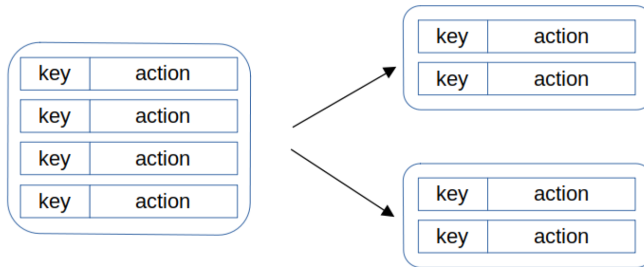

Figure 3: Horizontal splitting

using fewer but larger tables, while in another environment using more but smaller tables may yield better results. Therefore our implemented refactorings mainly focus on the modification of the match/action tables, namely horizontal and vertical splitting of tables, merging tables, changing the execution order of tables.

4.1.1 Horizontal table splitting

One of the simplest ways of table splitting is horizontal splitting introduced by Figure 3. In this case, the basic table structure remains the same, but we duplicate the table by halving the maximum size.
The prerequisites of the transformation are the following.

- The maximum size of the original table is explicitly defined.
- The table uses exact lookup mechanism.
- The application of the table appears in the match/action pipeline.

If the prerequisites hold, the following transformation steps must be taken.

1. Creating a new table with the same structure (using also the same default action).
2. Halving the maximum sizes.
3. Modifying the default action of the original table to `NoAction`. (A built-in action that changes nothing.)
4. If there exist more explicitly defined rows in the original table as the new maximum, copying the excess into the new table. (It is a possibility in P4 to defining explicit rows to a table.)
5. Searching every point in the match/action pipeline where the original table was applied. Modifying these applications to the execution of a sequence which first applies the original table and if no appropriate row was found during the lookup (the default action was executed) applies the newly created table.

4.1.2 Vertical table splitting

A much more complex case is when the table is split vertically. This step is executable if the table key contains two different fields. The parameter of the step is a value set which determines the likely values of the first key part which can appear in the table. It is worth noting that the control plane layer can change table contents dynamically, therefore this parameter has to be determined manually by an expert or based on some information coming from the control plane layer. Using this set, we can split the original table in such a way that first we just lookup the first key part in the determined set and then lookup the second key part in independent tables.

Considering more precisely the prerequisites of the transformation, we obtain the following conditions.

- The key of the table contains two different field values.
- The table uses the exact lookup mechanism for the first key part.
- The parameter set contains valid values for the first key part.
- The application of the table appears in the match/action pipeline.

If the prerequisites hold, the following transformation steps must be taken.

1. Creating a new dispenser table. Adding an explicit row for every values from the parameter set using a specific variant of the built-in `NoAction` as action.
2. Creating one executor table for every value from the parameter set. Copying the appropriate rows from the explicitly defined rows of the original table into the new tables.
3. Modifying the explicitly defined rows of the original table leaving only key values which do not appear in the parameter set.
4. Searching every point in the match/action pipeline where the original table was applied. Modifying these applications to the execution of a sequence which first applies the dispenser table and after that a `switch` branch based on the executed action, where every branch executes the application of the corresponding new executor table. Add the application of the original table as the default branch.

As an illustration, consider the table shown in Listing 1 and assume that the control plane layer will set the entries described in Table 1 into the table (for simplicity, we use only small numbers instead of real addresses in the example).

Executing a vertical splitting with the parameter set $\{1, 2, 3\}$, first we have to create three new specific version of the action `NoAction`. The new actions are introduced by Listing 2.

Table 1: Before vertical splitting

src_addr	*dst_addr*	action
1	1	*ipv4_forward*
1	2	*ipv4_forward*
1	3	*drop*
2	1	*drop*
2	2	*modify_dst*
3	1	*modify_dst*
4	4	*NoAction*

```
  action case1() {  }
  action case2() {  }
  action case3() {  }
```

Listing 2: Three new empty actions

Then a new table (called dispenser) should be created which determines based on the first key value which new table will be used during the lookup. The table declaration of the dispenser table of our example is described by Listing 3.

```
table dispenser {
  key = {
    hdr.ipv4.srcAddr: exact;
  }
  actions = {
    case1;
    case2;
    case3;
  }
  size=1024;
}
```

Listing 3: Example dispenser table

After that we have to create a new table for every value in the parameter set (called executor tables) to execute the remaining part of the lookup (based on the second key value). Listing 4 defines the declaration of the first executor table in the example (the other two executor tables have very similar declarations).

```
table executor1 {
      key = {
          hdr.ipv4.dstAddr: lpm;
      }
      actions = {
          ipv4_forward;
          modify_dst;
```

```
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
    }
```

Listing 4: Example executor table

Finally the `ipv4_lpm.apply()` match/action table application should be replaced with the code snippet described by Listing 5 at each point in the program where it originally appears.

```
switch(dispenser.apply().action_run) {
    case1 : { executor1.apply(); }
    case2 : { executor2.apply(); }
    case3 : { executor3.apply(); }
    default : { ipv4_lpm.apply(); }
}
```

Listing 5: New code snippet

Using the new structure, we need to use the entries described by Table 2 and Table 3 in the new dispenser table, in the original table and in the new executor tables to provide the same functionality.

Table 2: Vertical dispenser and the original table after splitting

src_addr	action
1	case_1
2	case_2
3	case_3

src_addr	*dst_addr*	action
4	4	*NoAction*

Table 3: Executor tables

executor_1	
dst_addr	action
1	*ipv4_forward*
2	*ipv4_forward*
3	*drop*

executor_2	
dst_addr	action
1	*drop*
2	*modify_dst*

executor_3	
dst_addr	action
1	*modify_dst*

4.2 Changing the execution order of tables

In addition to the structure of the tables, P4 programs also define the order in which they are executed. However in many cases the applied tables use independent fields of the header structure, so changing the execution order of them will not modify the results of the program. Changing the order can lead to more optimal memory usage or can allow further modifications, such as table merging.

More formally we check the independence between tables based on the dependency relations defined by Lavanya et al. [6]:

- **Match dependency** where the actions of the first table can modify a field which is used as a key in the subsequent second table.
- **Action dependency** where the first table and a subsequent second table both can change the same field.
- **Successor dependency** where the match result of the first table determines whether a second table should be executed or not.
- **Reverse match dependency** where the first table matches on a field that can be modified by a subsequent second table, and the first table must finish matching before the second table changes the field.

Based on these definitions we have implemented a dependency analysis in P4Query which – using the existing control flow and data flow analyses – can determine the dependency between two tables. We have defined and implemented a simple version of execution order changing based on this analysis.

The prerequisites of the transformation are the following.

- The applications of the two table are successive applications in the match/action pipeline of the P4 program.
- There does not exist match, action, successor or reverse match dependency relation between the two tables.

If the prerequisites hold, the transformation swaps the application of the two table in the program.

4.3 Further refactoring step

In addition to those described above, we have defined additional P4 specific steps. One is the merging operation corresponding to the previously described splitting transformations. We also defined a variant of the horizontal splitting which split the table based not on the size, but on some much likely used action appearing in the table and using as parameter of the transformation. We have defined some generic (not P4 specific) transformation steps too (e.g. parameter renaming and magic number replacing).

5 Conclusion

We have presented the definition of refactoring steps for P4. The proposed transformations focus mainly on the manipulation of match/action tables which are basic language elements in P4. The steps were implemented using the P4 specific analyzer tool, P4Query. The transformations are executed on the level of the internal representation of the tool which helps in performing the analyses of the prerequisites. Our current solution applies some generic consistency check provided by P4Query on the resulted graph. In the future we plan to implement further, more complex refactoring steps using the created refactoring infrastructure and extend the actual consistency checks with more specific verification methods.

References

[1] Birnfeld, K., da Silva, D. C., Cordeiro, W., and de França, B. B. N. P4 switch code data flow analysis: Towards stronger verification of forwarding plane software. In *NOMS 2020 – 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–8, 2020. DOI: 10.1109/NOMS47738.2020.9110307.

[2] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming protocol-independent packet processors. *SIGCOMM Computer Communication Review*, 44(3):87–95, 2014. DOI: 10.1145/2656877.2656890.

[3] Dumitrescu, D., Stoenescu, R., Negreanu, L., and Raiciu, C. Bf4: Towards bug-free P4 programs. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 571–585, New York, NY, USA, 2020. Association for Computing Machinery. DOI: 10.1145/3387514.3405888.

[4] Eichholz, M., Campbell, E., Foster, N., Salvaneschi, G., and Mezini, M. How to avoid making a billion-dollar mistake: Type-safe data plane programming with SafeP4. In Donaldson, A. F., editor, *33rd European Conference on Object-Oriented Programming*, Volume 134 of *LIPIcs*, pages 12:1–12:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. DOI: 10.4230/LIPIcs.ECOOP.2019.12.

[5] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Tóth, M., Bozó, I., and Király, R. Modeling semantic knowledge in Erlang for refactoring. In *International Conference on Knowledge Engineering, Principles and Techniques*, pages 38–53, Cluj-Napoca, Romania, 2009.

[6] Jose, L., Yan, L., Varghese, G., and McKeown, N. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems*

Design and Implementation, pages 103–115, Oakland, CA, 2015. USENIX Association. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jose.

[7] Lukács, D., Tóth, G., and Tejfel, M. P4Query: Static analyser framework for P4. *Annales Mathematicae et Informaticae*, 2023. DOI: 10.33039/ami.2023.03.002.

[8] Németh, F., Chiesa, M., and Rétvári, G. Normal forms for match-action programs. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 44–50, New York, NY, USA, 2019. Association for Computing Machinery. DOI: 10.1145/3359989.3365417.

[9] Nötzli, A., Khan, J., Fingerhut, A., Barrett, C., and Athanas, P. P4pktgen: Automated test case generation for P4 programs. In *Proceedings of the Symposium on SDN Research*, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery. DOI: 10.1145/3185467.3185497.

[10] Nunes, B. A. A., Mendonca, M., Nguyen, X.-N., Obraczka, K., and Turletti, T. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys and Tutorials*, 16(3):1617–1634, 2014. DOI: 10.1109/SURV.2014.012214.00180.

[11] P4 language specification, 2021. URL: https://p4.org/p4-spec/docs/P4-16-v1.2.2.html.

[12] Rodriguez, M. A. The Gremlin graph traversal machine and language. *Proceedings of the 15th Symposium on Database Programming Languages*, 2015. DOI: 10.1145/2815072.2815073.

[13] Sultana, N. and et al. Flightplan: Dataplane disaggregation and placement for P4 programs. In Mickens, J. and Teixeira, R., editors, *18th USENIX Symposium on Networked Systems Design and Implementation*, pages 571–592. USENIX Association, 2021. URL: https://www.usenix.org/conference/nsdi21/presentation/sultana.