Acta Cybernetica — *online–first paper version* — pages 1–27.

Overlaying Control Flow Graphs on P4 Syntax Trees with Gremlin*

Dániel Lukács[ab] and Máté Tejfel[ac]

Abstract

Our overall research aim is to statically derive execution cost and other metrics from program code written in the P4 programming language. For this purpose, we extract a detailed control flow graph (CFG) from the code, that can be turned into a full, formal model of execution, to extract properties – such as execution cost – from the model. While CFG extraction and analysis is well researched area, details are dependent on code representation and therefore application of textbook algorithms (often defined over unstructured code listings) to real programming languages is often non-trivial. Our aim is to present an algorithm for CFG extraction over P4 abstract syntax trees (AST). During the extraction we create direct links between nodes of the CFG and the P4 AST: this way we can access all information in the P4 AST during CFG traversal. We are utilizing Gremlin, a graph query language to take advantage of graph databases, but also for compactness and to formally prove algorithm correctness.

Keywords: control flow graph, static analysis, P4, Gremlin, graph database, proof of correctness

1 Introduction

Our long-term research goal – that also motivates this current work – is to develop an adaptable, scalable, and efficient static cost analysis tool for programs written in the P4 programming language [7]. P4 is a new domain-specific programming language running on programmable network switches. P4 programs describe network communication protocols: more specifically, a P4 programs tells the switch how process (transform, forward, or drop) an incoming network packet. Static cost

*Supported by the ÚNKP-21-4 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund. This research is in part supported by the project no. FK_21 138949, provided by the National Research, Development and Innovation Fund of Hungary.

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

DOI: 10.14232/actacyb.298770

analysis tools – that can estimate performance, energy needs, and other metrics of a P4 program automatically and without actually executing the program code – have several industrial use cases. Unfortunately, cost analysis is NP-hard (it solves the halting problem). While algorithms exist, they do not scale well for large, industrial size P4 programs: the time it takes to compute the solution exceeds the bounds of what is considerable usable in the industry.

As we attempted to realise a cost analysis tool for P4, we found that control flow analysis has a central role in all our efforts. Control flow analysis [1, Chapter 8.4.3.] concerns discovering the order of execution of the program statements in compile-time (i.e. based on the source code, or its equivalent representation, the syntax tree). Due to branching structures, there are usually multiple possible selections of executable statements, so the appropriate representation of the results is a graph, called control flow graph (CFG). We refer to paths in the CFG as execution paths.

Cost analysis requires a representation where implementation-dependent information (abstractions of the implementation, for example, cost formulas) can be easily inserted. CFGs turned out to be such representations. In our earlier works [10, 9], we discussed an approach based on enumerating all possible execution paths in the CFG of a P4 program to produce the average (or minimum or maximum) cost for that program. Our preliminary measurements have also shown that CFG path enumeration scales up for CFGs having as much as one hundred-thousand execution paths.

CFGs can also be considered Kripke-structures or transition systems [5, Chapter 2.1.], with the program counter being the only visible variable in the state, while the actual program state stays implicit in the start state and its subsequent transformations by the program instructions. A clear consequence of this is that CFGs – as traditionally understood – are not full program representations, but graphs that connect program points to program points, mostly with no formal references to the actual data that is being processed during execution. For example, it may happen that during execution, we update a variable in a way such that one branch of a conditional is never executed. Taking the cost of this branch into account in the average cost then possibly leads to a significant overestimation of the true program cost.

For this reason, in our latest work [11], we decided to try a new approach instead of CFG path enumeration and relied on probabilistic model checking to cost analyse P4 code. Here, we translate P4 code into model checkable representation, and – together with the specification of cost requirements – we delegate this to a model checker tool. Yet, even in this approach, we rely on CFGs in order to generate the model checkable representation. And here as well, we need more information about the program points than what textbook definition CFGs store.

Thus, our first aim in this paper is to develop a variant of CFG representations that is also capable of meaningfully representing program data and instructions over this data. To achieve this, we will define control flow analysis over the abstract syntax tree (AST) [1, Chapter 2.5.1.] of P4 programs. The AST is a hierarchical representation of the program sources, describing how various program structures and expressions are nested into each other. As AST is a full program representation,

we can exploit the fact that the AST already has every information about data and instructions, and to make this information available during control flow analysis, we just have to establish the links between the corresponding nodes in the AST and the CFG. As a result, when we traverse the CFG, any further information about the current program point is just one link away, in the form of an AST subtree.

As it can be seen, interconnected graph representations will be central to our efforts. To make sure we use graphs in the most efficient way possible, we host these graphs (AST and CFG) in a graph database (GDB). A GDB is a database that stores data in graph data structure and provides a query language with graph semantics. Compared to relational databases (storing data in tables), GDBs eliminate the need for expensive join-operations, making them more efficient (both in terms of computation and usage) for storing and traversing heavily interconnected data. An extensive meta-analysis on the concept can be found in Angles et al. [3].

Then, our second aim is to address the problem of implementing a CFG algorithm in the form of a Gremlin query. By doing so, we can leverage built-in optimisations in Gremlin-compatible GDBs, such as parallelisation and bulking (see Rodriguez [14]). As Gremlin is a domain-specific language with somewhat unusual syntax and semantics, we found this task challenging enough to deem it necessary to discuss it in depth. We also hope that this discussion will be helpful for all future Gremlin programmers aiming to implement non-trivial algorithms in Gremlin.

Contributions In this work, we present an approach to intraprocedural CFG extraction from ASTs (formalised as a Gremlin query), and prove its correctness. At the same time, we explore the expressive power of Gremlin for specifying fairly complex static analysis procedures. We define both ASTs and CFGs in Section 3.1, in a way that can handle most of the P4 language control flow, and makes extending the AST for the rest is a straightforward process. Then, in Section 3.2, we describe the extraction algorithm in pseudocode. In Section 4.1 we formalised the semantics of a subset of Gremlin. Section 4.2 contains the extraction algorithm in Gremlin. We use the formal description to prove the correctness of the algorithm in Section 4.3. Finally, we conclude the paper with a few words about limitations and future work.

2 Related work

Recently, Dumitrescu et al. [8] introduced *Bf4*, a program verification tool for P4 programs, that also builds heavily on the CFG representation of P4. They rely on a preceding instrumentation step, and extend the CFG with "bug nodes" (nodes, guarded with a condition that can only be satisfied if there is a bug), and then perform program slicing (using SSA and various dependency analyses) to compute reachability of the bug nodes using an SMT solver. They do not discuss their internal CFG representation, but they do tell that they realised the tool as a P4C backend, and the size of the implementation is around 25000 lines of C++ code (not

counting the P4C infrastructure). We suspect GDB-integrated deep CFGs could complement the Bf4 implementation in order to reach all necessary the information more easily than what visitors over the P4C intermediate program representation can currently provide.

The work of Amighi et al. [2] shares some goals with ours. They extract CFGs from Java bytecode, which is a more difficult problem since it involves handling stack (implicit the bytecode) and exception flow as well. First, they translate bytecode to an intermediate representation that makes the stack explicit. Then, for each instruction they declaratively define the transition relation between program points. The final CFG is simply the union of the transitions resulting from evaluating the relation over program points and instructions of a given bytecode. Like us, they also prove the correctness of the extraction. In their proof, they establish the existence of a simulation relation between states induced by the bytecode instructions and states induced by the extracted CFG.

An important application of CFGs is that it is the representation on which data flow analysis (DFA) [1, Chapter 9.2.] operates. The results of DFA can be represented e.g. in the form of a definition-use graph, establishing links between the definitions of variable names and the usages of these names. In turn, it should be possible to store such a definition-use graph in our GDB, interlinked to AST and CFG, in order to enable even more applications. For example, Birnfeld et al. [6] combine CFG and definition-use graphs to discover potential faults in P4 code, to detect e.g. that there are execution paths where the P4 program processes invalid packets.

In our work, we extract CFGs from a structured AST, not from unstructured code (with features such as `gotos`, no nesting, etc.). This is also the approach of Söderberg et al. [15], who – analysing Java – recognise that by superimposing the CFG on the AST, "high-level abstractions are not compiled away during the translation to intermediate code". The authors utilise elegant reference attribute grammars for control flow and data flow analysis. In this approach control-related AST nodes get a reference attribute (e.g. *successor*) that points to another AST node where control is supposed to flow from the previous node. Another interesting feature of this work is that it is easily extensible to handle new language elements in novel versions of Java, by incrementally adding new grammar rules.

Another inspiring example for this concept is the RefactorErl framework, that also superimposes control flow (and many other static analysis results) on the AST [16] for the Erlang programming language.

3 Problem and solution idea

3.1 Basic definitions

In this section, we define what we mean by ASTs and control flow graphs in the following sections. In the correctness proof in Section 4.3, these definitions constitute the precondition and postcondition.

An illustration of these concepts is depicted by Figure 1. One the left, there is a simplified excerpt from `basic_routing-bmv2.p4`, a P4 program describing an L2/L3 routing protocol, used in the testing of the P4 reference compiler, P4C [13]. This code describes a control declaration named `ingress`. It declares a few match-action tables (their external definition is linked at compile-time), and then it specifies the actual control flow determining the (sometimes conditional) invocation of these tables. In the middle is the corresponding AST with `d`, `b`, `c`, `s` labelled nodes denoting control declaration, block, conditional, and statement nodes respectively. (We do not analyse control flow inside expressions.) On the right is the corresponding CFG with matching node names. The thin lines are the association edges between AST nodes and CFG nodes.

```
control ingress(inout headers hdr,
                [...]) {
 table bd {[...]}
 table ipv4_fib {[...]}
 table ipv4_fib_lpm {[...]}
 table nexthop {[...]}
 table port_mapping {[...]}
 [...]
 apply {
   if (hdr.ipv4.isValid()) {
     port_mapping.apply();
     bd.apply();
     if ([...]) {
       ipv4_fib_lpm.apply();
     } else {}
     nexthop.apply();
   } else {}
  }
}
```

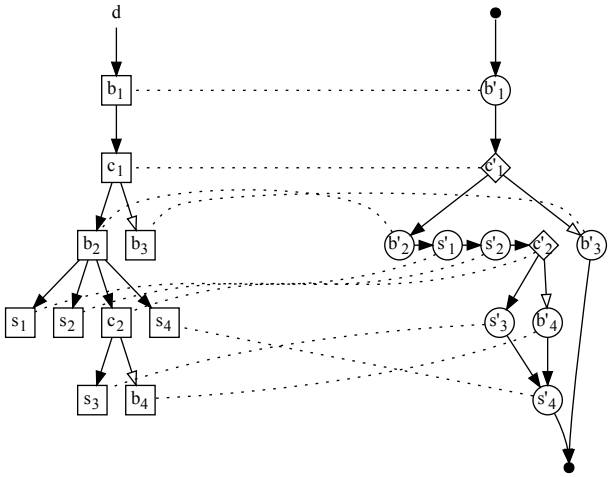

Figure 1: Source code, AST, and CFG of a control declaration

In the rest of this section, we define these concepts based on labelled graphs and related notations. GDBs support sophisticated attribute-based labelling, but for ease of understanding we use a simpler scheme in this paper.

Definition 1. *A labelled graph is a* (V, E, l) *tuple of a* V *node set, an* E *edge set, and an* $l : (V \cup E) \to L$ *labelling function (where* L *is an arbitrary set of labels).*

Notations. In case a distinction must be made between multiple graphs, we write e.g. (V_g, E_g, l_g) to denote the components of a particular graph g. We use underlined lowercase letters to denote a node with a specific label: for example $\underline{x}$ denotes a node $n \in V$ for which $l(n) = \texttt{x}$ (a node with label `x`). We use indexes to distinguish between multiple nodes: for example, $\underline{x}_1$ and $\underline{x}_2$ denotes n_1, n_2 nodes for which $l(n_1) = l(n_2) = \texttt{x}$. We write $n_1 \xrightarrow{\texttt{x}} n_2$ to denote an edge $(n_1, n_2) \in E$ for which $l((n_1, n_2)) = \texttt{x}$. In the case graph g is a tree, $root_g$ denotes the root of tree g, and $children_g(n, \texttt{x})$ denotes those child nodes of node n in g whose incoming edge has label `x`.

Notations. In the AST, we use labels `d`, `b`, `c`, and `s` to denote control declarations, blocks, conditionals, and statements, respectively (so e.g. $\underline{d} \in V$ denotes any $n \in V$ node that is a control declaration). Syntactical edges between the AST nodes are appropriately labelled with labels to distinguish from other edges introduced into G, e.g. `assoc` (see later). These edge labels are `body` (between a control declaration node and the top-level block forming its body), `nest` (between a block and its nested blocks), `statement` (between a block and its statements), `true`, and `false` (between a conditional node and its branches).

Definition 2. *A (V, E, l) graph is an abstract syntax tree (AST), if it is a tree, and $\forall n \in V : l(n) \in \{\mathtt{d}, \mathtt{b}, \mathtt{c}, \mathtt{s}\}$, and*

1. *If $(n_1 \xrightarrow{\mathtt{body}} n_2) \in E$, then $l(n_1) = \mathtt{d}$, and $l(n_2) = \mathtt{b}$*
2. *If $(n_1 \xrightarrow{\mathtt{nest}} n_2) \in E$, then $l(n_1) = \mathtt{b}$, and $l(n_2) = \mathtt{b}$*
3. *If $(n_1 \xrightarrow{\mathtt{statement}} n_2) \in E$, then $l(n_1) = \mathtt{b}$, and $l(n_2) = \mathtt{s}$,*
4. *If $(n_1 \xrightarrow{\mathtt{true}} n_2) \in E$, then $l(n_1) = \mathtt{c}$, and $l(n_2) = \mathtt{b}$*
5. *If $(n_1 \xrightarrow{\mathtt{false}} n_2) \in E$, then $l(n_1) = \mathtt{c}$, and $l(n_2) = \mathtt{b}$*
6. *If $\underline{s} \in V$, then $children(\underline{s}) = \varnothing$*
7. *If $\underline{d} \in V$, then $\exists!\ \underline{b} \in V : (\underline{d} \xrightarrow{\mathtt{body}} \underline{b}) \in E$,*
8. *If $\underline{c} \in V$, then $\exists!\ \underline{b}_1, \underline{b}_2 \in V : ((\underline{c} \xrightarrow{\mathtt{true}} \underline{b}_1) \in E \ \wedge\ (\underline{c} \xrightarrow{\mathtt{false}} \underline{b}_2) \in E)$,*

The definition asserts that all P4 control declaration has an AST made of blocks (containing 0,1 or more ASTs), conditionals (containing exactly two ASTs, one per branch), and statements (primitives, along with empty blocks).

In addition, we assume (without explicitly featuring) that all AST nodes have a unique identifier incremented in depth-first order. Such identifier attributes (labels) can be inserted in the graph straightforwardly (preferably directly after graph construction). The recursive top-down traversal of ASTs guarantees termination and allows us to use a common idea in correctness proofs of functional programs. By inductively assuming that the previous elements were correctly processed, and applying a proven correct procedure to the current element, we only have to assure that the previous and the current elements are aggregated in an appropriate manner.

In later sections, we will define our algorithm over ASTs of P4 control declarations. For this reason, we restrict our discussion here to these, and omit discussing `goto`-like flows in the P4 packet parser declarations. In P4, packet parsers are defined in the form of state machines. Their control flow analysis is straightforward, so omit this for simplicity. P4 has no construct for user-defined loops except for match-action tables (lookup tables that match packet headers to actions). The implementation of these table algorithms is not part of the language, only the node of

the table application appears in the syntax tree (and similarly, table applications will be featured as single nodes in the control flow graph). For languages with loops, extending the definition is straightforward, albeit cumbersome. P4 also has one-way and multi-way conditionals, but those are processed similarly to two-way conditionals, and so we also omit them for simplicity.

Now, we prepare for defining deep CFGs. The main problem we have to solve as we translate an AST to a CFG is that ASTs give no explicit clue about the order of execution of its elements, while CFGs aims to describe precisely that. It is well-known [4], that by transforming CFGs to static single-assigment form, blocks can be treated as functions, and directed flows as calls between these functions: the result is a functional program, where the continuation (i.e. the rest of program) at each program point explicitly appears as a function. While we aim for a more direct definition, this gives us a clue that a recursive approach can be successful. Specifically, we will define CFGs as compositions of sub-CFGs, with each sub-CFG relating to a subtree of the control declaration AST.

Definition 3. *Let $G = (V, E)$ contain syntax subtree t and subgraph c. We say that c is the sub-CFG corresponding to t, with source $n \in V_c$ and return points $R \subseteq V_c$ given the following conditions are satisfied in G:*

1. *If $root_t = \{\underline{s}\}$, then $(n \xrightarrow{assoc} \underline{s}) \in E, \quad R = \{n\}$*

2. *If $root_t = \{\underline{b}\}$ and $children_t(\underline{b}) = \varnothing$, then $(n \xrightarrow{assoc} \underline{b}) \in E, \quad R = \{n\}$*

3. *If $root_t = \{\underline{b}\}$ and $children_t(\underline{b}) = \{t_1, \ldots, t_k\}$ and c_i is the sub-CFG with source n_i, return points R_i corresponding to the subtree rooted in t_i $(\forall i = 1, \ldots, k)$, then*

$$
\begin{aligned}
&(n \xrightarrow{assoc} \underline{b}) \in E,\\
&(n \xrightarrow{flow} n_1) \in E_c,\\
&(r \xrightarrow{flow} n_i) \in E_c \ \ (\forall i = 2 \ldots k, \ \ \forall r \in R_{i-1}),\\
&R = R_k
\end{aligned}
$$

4. *If $root_t = \{\underline{c}\}$ and $children_t(\underline{c}) = \{t_1, t_2\}$, then*

$$
\begin{aligned}
&(n \xrightarrow{assoc} \underline{c}) \in E,\\
&c_i \text{ is the sub-CFG corresponding to } t_i \text{ with source } n_i,\\
&\quad \text{return points } R_i \ \ (\forall i = 1, 2),\\
&(n \xrightarrow{flow} n_i) \in E_c \ \ (\forall i = 1, 2),\\
&R = R_1 \cup R_2
\end{aligned}
$$

According to the definition, the sub-CFG of statements and empty blocks is a single node, relating to the syntax node of the statement or empty block itself. The sub-CFG of non-empty blocks is composed of a CFG node relating to the syntax node of the block, and the sub-CFGs of its children. We set up flow from the block to the first child CFG, and also between the siblings. The sub-CFG of a conditional is composed of a CFG node relating to the syntax node of the conditional, and of the sub-CFGs of the children. Here, we omitted true and false labels on the flows, but these can be easily identified by querying the incoming edge (labelled with `true` or `false`) of the associated node in the AST.

We may note that control declaration nodes (`d`-labelled nodes) are missing from the sub-CFG definition. This is because the CFG corresponding to such a node is a top-level CFG, that we call deep CFG. This node does not require much analysis compared to its descendants, it simply identifies the entry and exit nodes of the CFG.

Definition 4. *Let $G = (V, E)$ contain control declaration AST u and subgraph g. We say that g is the deep CFG corresponding to u, with entry $e \in V_g$ and exit $f \subseteq V_g$ given the following condition is satisfied in G:*
If $root_u = \underline{d}$, $child(\underline{d}) = \{t\}$ and g is the sub-CFG corresponding to the tree rooted in t, with source n and return points R, then

$$(\underline{d} \xrightarrow{entry} e) \in E,\ \ (\underline{d} \xrightarrow{exit} f) \in E,\ \ (e \xrightarrow{flow} n) \in E_g,\ \ (r \xrightarrow{flow} f) \in E_g\ \ (\forall r \in R)$$

Edges labelled with `entry` and `exit` are similar to `assoc`, linking the AST declaration node to the CFG entry and exit points. Control will flow from the entry node to the first block (the source of the sub-CFG corresponding to the declaration body). From the return points of this first block, control flows into the exit node. Flows inside the CFG are determined by Definition 3.

3.2 CFG extraction

First, we present the idea of our CFG extraction algorithm by translating the CFG definition into an informal, imperative description. Later on we formalise this as a Gremlin traversal. We split the operation in two.

Algorithm 1 iterates over the control declaration nodes in the AST, creates one entry and one exit CFG node (e and f) for each in graph G, and then calls Algorithm 2 on b (the top-level block of the declaration). An edge will be sent from e to b by that other procedure (as b's CFG is the subsequent continuation of e). Finally, we send an edge from R (the return nodes returned by that procedure) to f (as the exit is the final continuation).

Algorithm 2 creates a CFG for the AST of some b, either a block or a conditional. The algorithm expects a set of predecessor CFG nodes: while ProcNode is initially called with (a set of) just a single predecessor, later it is called again recursively with the R set of return points. The resulting CFG is the subsequent continuation of whatever is in R, and so right after we create its starting point n, we link the contents of R to n. In case b is a block or a statement, we recursively apply the

procedure to the children. R is initially set to n, since the continuation after node b is the CFG of the first child. After the child CFG was produced, we assign the return points of that CFG to R as this needs to be linked to the next continuation (that is either a sibling, or a sibling of an ancestor). Note that in case b is an empty block or a statement, it has no children and $\{n\}$ is the returned return point (this is the node that will be followed by the sibling of an ancestor).

In case b is a conditional, n has to be linked to the two possible subsequent continuations, and we collect into R the return points of both branch CFGs, as anything that follows will be the subsequent continuation of both branches. Algorithm 2 always terminates because it is progressing from child to child, and for any of its calls, the longer the call's stack trace, the shorter the distance between b and the AST leaves.

Procedure `CFG(`G`)`**:**
Input: G is graph, includes AST
Result: CFG of each control declaration is added to G
begin
forall $v \in V_G$ **do**
if v *is control declaration* **then**
$e :=$ new CFG entry node
$f :=$ new CFG exit node
$V_G := V_G \cup \{e, f\}$
$b := child_G(v, \texttt{body})$
$R :=$ `ProcNode(`G, b, $\{e\}$`)`
forall $r \in R$ **do**
$E_G := E_G \cup \{r \xrightarrow{\texttt{flow}} f\}$
end
end
end
return
end

Algorithm 1: Control declarations

4 Formalising CFG extraction in Gremlin

4.1 Semantics of Gremlin

To formally prove our CFG extraction operation in Section 4.2, we have to have a formal semantics for the Gremlin Traversal Machine (GTM). In the white paper [14], the authors give a mathematical description of the GTM, but one of their goals is to keep it general and enable adapters to implement many possible – including parallel – evaluation strategies. In this section, we intend to reiterate this description with two important modifications: we formalise it as an axiomatic

Procedure `ProcNode(`G,b,P`)`**:**
Input: G is graph, includes AST and CFG
Input: b is block, statement or conditional in the AST
Input: P is set of predecessor CFG nodes
Output: Return points of CFG of b
Result: CFG of b is added to G
begin
$n :=$ new CFG node
$V_G := V_G \cup \{n\}$
forall $p \in P$ **do**
$E_G := E_G \cup \{p \xrightarrow{flow} n\}$
end
if *b is block* $\vee$ *b is statement* **then**
$R := \{n\}$
forall $c \in children_G(b, \{\texttt{nest}, \texttt{statement}\})$ **do**
$R :=$ `ProcNode(`G, c, R`)`
end
return R
else
if *b is conditional* **then**
$R := \varnothing$
forall $c \in children_G(b, \{\texttt{true}, \texttt{false}\})$ **do**
$R := R \cup$ `ProcNode(`G, c, $\{n\}$`)`
end
return R
end
end
end

Algorithm 2: Blocks, conditionals

semantics so that we can use it in proofs (see Section 4.3), and we restrict the traverser set (see later) to be an ordered set (or list). This restriction ensures that sibling nodes are processed sequentially, and in a fixed order. In our experience, the default evaluation strategy in the native Gremlin Java graph implementation (called `TinkerGraph`) satisfies this restriction.

The GTM executes a program called graph traversal, which is effectively a sequence of instructions (with some higher-order instructions also accepting graph traversal programs as parameters). While "traversal" is a notion employed in the Gremlin-literature to denote programs, it may cause some confusion that it is also used colloquially to denote the execution of such programs. Therefore, in the formal treatment we use the notion of traversal to denote the program text, and use other phrases (semantics, state, etc.) to characterise execution. We now proceed first to define the state of the GTM. The global memory state of the GTM consists of the

graph contents and other auxiliary storages. There are stores in the memory that can store both graph objects (e.g. nodes, edges) and other objects (e.g. collections of graph objects).

Definition 5. *The **global memory** of the GTM state is a $\Gamma = (V, E, H, K)$ tuple, where (V, E) is the graph itself, H is an object heap for processing non-graph objects during the traversal, and K is a global key-value store (often, referred to as side-effect store).*

Besides the global memory, graph traversals have a current local state called traverser. A GTM state stores multiple traversers at the same time. Informally, we can imagine each traversers as a worker (in Gremlin materials often depicted as the green little monster) that jumps from node to node (as per the instructions of the traversal program), and collects data about the nodes into various data structures. In addition, the worker can "clone" itself when the traversal program branches: the clone starts with the same data as the original, but they will move around following the instructions of a different program branch.

Definition 6. *A **traverser** is a (p, k, s) triple, where p is a pointer to a graph element or an object in H, k is a local key-value store, and s is the sack, a local store for sum-like operations (aggregation).*

The white paper [14] lists some other components of traversers as well, but we will not use those in this work.

Finally, a traversal is effectively a program, whose instructions (called *steps* and denoted by σ) transform a Γ global memory (e.g. by changing the graph) and list of (p, k, s) traversers (e.g. by moving the individual traversers forward in the graph).

Definition 7. *A **traversal** is a program defined by the following simple grammar:*

$$\Psi ::= \varepsilon \mid \sigma \rightsquigarrow \Psi$$
$$\sigma ::= \sigma_1 \mid \sigma_n(\Psi, \ldots, \Psi)$$

A traversal Ψ may be empty (ε), or it may consist of a step σ, sequentially followed by the rest of the traversal. Steps are $2^\Gamma \times 2^T \longrightarrow 2^\Gamma \times 2^T$ functions, and come in two variants: higher-order steps ($\sigma_n(\Psi, \ldots, \Psi)$) are parameterised by a number of different subtraversals and transform the GTM state by executing these subtraversals in the current GTM state, while first-order and zeroth-order steps (σ_1) take only ordinary parameters to transform the state. In these terms, we can think of $\rightsquigarrow$ as forward function composition.

Notations. In the grammar of Definition 7, σ, σ_1, σ_n and Ψ are non-terminals. Later in the text, we use them to denote concrete traversals and steps. In particular, we will use σ to denote steps in the Gremlin language, e.g. $\sigma_{\texttt{flatMap}}$ will denote the `flatMap` step of Gremlin.

Gremlin defines over 30 type of steps, and we have no place here – nor do we find it indispensable – to include a formal definition for each of them. After defining machine state and semantics, we will include in Table 1 short informal descriptions for the ones we used in our CFG extraction traversal (e.g. $\sigma_{\texttt{outE}}$, $\sigma_{\texttt{inV}}$, $\sigma_{\texttt{flatMap}}$, $\sigma_{\texttt{sideEffect}}$), and hope that our readers can reconstruct a formal definition in case needed.

Definition 8. *The state of the GTM is a triple* $(\Gamma,\ T,\ \Psi)$*, where* Γ *is the state of the global memory,* T *is a traverser list (storing the current local state of multiple traversals), and* Ψ *is a traversal.*

Below, we formalise the evaluation of Gremlin traversals as an axiomatic semantics similar to Hoare-logic. A thorough introduction on defining language semantics with inferential systems and proving it using inferential trees can be found in Nielson & Nielson [12, Chapter 6.2].

Note that we intend Rules (4), (5), and (6) as templates that highlight and help in formalizing the three main categories of concrete steps.

$$\frac{}{\{\Gamma;\varnothing\}\quad\Psi\quad\{\Gamma;\varnothing\}} \tag{1}$$

$$\frac{}{\{\Gamma;T\}\quad\varepsilon\quad\{\Gamma;T\}} \tag{2}$$

$$\frac{\{\Gamma;T\}\ \ \sigma\ \ \{\Gamma_1;T_1\}\qquad\{\Gamma_1;T_1\}\ \ \Psi\ \ \{\Gamma_2;T_2\}}{\{\Gamma;T\}\quad\sigma\rightsquigarrow\Psi\quad\{\Gamma_2;T_2\}} \tag{3}$$

$$\frac{\sigma_1(\Gamma;\ t_1)\mapsto(\Gamma_1;\ R_1)\qquad\cdots\qquad\sigma_1(\Gamma_{n-1};\ t_n)\mapsto(\Gamma_n;\ R_n)}{\{\Gamma;\ t_1,\ldots,t_n\}\quad\sigma_1\quad\{\Gamma_n;\ R_1\cup\ldots\cup R_n\}} \tag{4}$$

$$\frac{\{\Gamma;t_1\}\ \ \Psi'\ \ \{\Gamma_1;R_1\}\qquad\cdots\qquad\{\Gamma_{n-1};t_n\}\ \ \Psi''\ \ \{\Gamma_n;R_n\}}{\{\Gamma;\ t_1,\ldots,t_n\}\quad\sigma_n(\Psi)\quad\{\Gamma_n;\ R_1\cup\ldots\cup R_n\}} \tag{5}$$

$$\frac{\{\Gamma;T\}\ \ \Psi_1'\ \ \{\Gamma_1;R_1\}\qquad\cdots\qquad\{\Gamma_{n-1};T\}\ \ \Psi_n''\ \ \{\Gamma_n;R_n\}}{\{\Gamma;\ T\}\quad\sigma_n(\Psi_1,\ldots,\Psi_n)\quad\{\Gamma_i;\ R_i\}}\quad i=\min_{\substack{j=1\ldots n\\ R_j\neq\varnothing\vee j=n}} j \tag{6}$$

Axioms (1) and (2) correspond to termination in case the traversal mapped to an empty traverser list or in case all steps were executed in the traversal.

Rule (3) corresponds to the classic sequencing rule: the first step and the rest of the traversal is executed in the program state resulting from the first step. What may not be evident at first glance, is that this step prescribes a **breadth-first traversal**: step σ is applied to all traversers in T (possibly modifying the global state) before the following steps are applied to any of them.

Rule (4) is a template rule for describing how first-order steps are evaluated. This rule also emphasizes our restriction that traversers are processed in some fixed order.

Note that some steps may map to any number of traversers (including zero). Rule (5) is a template rule for higher-order traversals: it goes through the traversers

in some fixed order, and applies the subtraversal one-by-one to each traverser (possibly modifying the global state in the mean time). Additionally, we allow instances of this rule to apply some modifications to the Ψ traversal (denoting the resulting traversal as Ψ') so a large variation of traversals (such as conditional traversals) can be expressed.

Finally, Rule (6) formalises special higher-order traversals knowns as branching. This rule executes its subtraversals in a way such that if Ψ_1 terminates with an empty traverser list, then Ψ_2 is executed, and so and so on either until one of the traversers terminates with a non-empty traverser list, or until Ψ_n is executed. While failed traversers may modify the state, only the traverser list of the first successful traverser will be returned. For generality, we also allow instances of this rule to modify their Ψ_i subtraversals. Note that higher-order traversals enable us to define **depth-first traversals** as well, since subtraversal has to be completely executed to complete the step.

In Table 1, we informally describe individual Gremlin steps we use in this paper. For space reasons, we omit formally defining the semantics of each step, and just highlight some of them to familiarise our readers with the notation. Based on the white paper [14], the online Gremlin documentation, and our algorithm description in our algorithm description in Section 4.2, we believe it is not too difficult to reconstruct the semantics of the steps.

4.2 CFG extraction in Gremlin

It is common for graph procedures (and as such, static analysis procedures as well) to have a short, intuitive textual specification, that – when implemented in executable code – blows up into an entangled web of nested loops, custom data structures, and reliance on language built-in dispatch mechanisms for distinguishing involved objects.

To avoid the gap between presentation and implementation, we formalised CFG extraction as a Gremlin (v3.4.4) traversal. The traversal in Figure 2 – consisting of around 60 steps – can be typed into any Gremlin language variant (e.g. Gremlin Java) with minimal amount of language-specific modifications (for example defining a function expression for σ_{clear}, using function expressions to enable lazy recursive calls of subtraversals, using type hints, etc.). In fact, we automatically generated the formulas in this paper directly from our implementation code, and then manually simplified the aforementioned elements. (Regarding recursion, see limitations in Section 5.) Our Gremlin Java implementation – that was extended to handle a slightly more elaborate graph schema that distinguishes between AST overlays and CFG overlays –, is slightly below 110 lines of code with each line having one or two steps. Beyond those mentioned, the requirement of navigating the labels and properties in the more complex graph schema was solely responsible for having to add in additional steps.

We show in Section 4.3 that this compact representation combined with Gremlin's simple semantics is very effective for formally deriving its correctness proof by hand. The alternative – proving a less abstract, less compact executable rep-

Table 1: Informal description of selected Gremlin steps

Step	**Description**
σ_{outE}	"Moves the traverser forward", i.e. it replaces the nodes in the traverser list with their outgoing edges.
σ_{inV}	"Moves the traverser forward", i.e. it replaces the edges in the traverser list with the nodes they enters
$\sigma_{\text{flatMap}}(\Psi)$	Applies Ψ to all $t \in T$ (see Definition 8) as described by Rule (5).
$\sigma_{\text{sideEffect}}(\Psi)$	Abbreviated as $\sigma_{\text{sEffect}}(\Psi)$. Also a Rule (5) step; it may modify the global state Γ, but it discards the traverser list produced by Ψ and returns the original T.
$\sigma_{\text{coalesce}}(\Psi_1, \ldots, \Psi_n)$	A Rule 6 step, that returns the traversers from the first of traversals $\Psi_1, \ldots \Psi_n$ which is successful (has non-empty result).
$\sigma_{\text{aggregate}_x}$	Adds the current traverser list into a collection assigned in K to name x.
σ_{cap_x}	Loads the content stored in x into the traverser list.
σ_{unfold}	Replaces collections in the traverser list with the elements of the collections.
$\sigma_{\text{sEffect}}(\text{clear})$	Empties a collection. Useful for discarding the traversers stored earlier in x, before adding a new element. *clear* is a custom Java function expression that empties the collection.
$\sigma_{\text{sack}_{\text{in}}}$	Applied to (p, k, s) will put p inside store s. This step enables additional sum-like operations, that we will not use.
$\sigma_{\text{sack}_{\text{out}}}$	Applied to (p, k, s) will replace p with the content of store s.
$\sigma@x$	Given that some σ step produces a (p, k, s) traverser, $\sigma@x$ will assign p to name x in local store k.
σ_{tail_1}	Removes all elements of the traverser list except for the last one.

resentation (such as direct Java code) with the same level of formality – would have likely required machine assistance, and in that case it likely would have been impossible to fit the complete proof of such a representation into this paper.

We now describe how this traversal implements the previously described CFG extraction procedure. The procedure is a traversal consisting of five subtraversals depicted in Figure 2. Ψ_{control} and Ψ_{icontrol} are corresponding to Algorithm 1 in Section 3.2. Ψ_{control} selects nodes in the AST that correspond to a P4 control declaration d, and then executes Ψ_{icontrol} (the internal part of Ψ_{control} we separated for readability). Here, we make use of Rule 5 semantics to make sure that steps in

Ψ_{icontrol} affect just one control declaration subtree at a time. Ψ_{icontrol} clears global register `r`, stores the entry CFG node of d into `r`, invokes subtraversal Ψ_{node} over the body block of d, and finally sends a `flow`-edge from the contents of `r` (supposedly filled by Ψ_{node}) to the exit CFG node of d. The second and third side-effects are simply there because we do not need the results from those subtraversals, and instead we want to continue from the same place they started. The side-effect with Ψ_{node} simply performs the invocation of the subtraversal. $\sigma_{\text{flatMap}}(\sigma_{\text{cap}_\texttt{r}} \rightsquigarrow \sigma_{\text{unfold}})$ is an idiom that makes the content of `r` the current traversal. σ_{flatMap} here is an implementation detail: σ_{cap} loses the traverser data, but σ_{flatMap} reattaches it to the new traversers.

$$
\begin{aligned}
\text{control} &= \left(\begin{array}{l} V_{\texttt{ControlDeclaration}} \\ \rightsquigarrow\ \text{sEffect(icontrol)} \end{array}\right) \\
\text{icontrol} &= \left(\begin{array}{l}
\text{sEffect}(\text{clear}_\texttt{r}) \\
\rightsquigarrow\ \text{sEffect}\left(\begin{array}{l} \text{outE}_{\texttt{entry}} \\ \rightsquigarrow\ \text{inV} \\ \rightsquigarrow\ \text{aggregate}_\texttt{r} \end{array}\right) \\
\rightsquigarrow\ \text{sEffect}\left(\begin{array}{l} \text{outE}_{\texttt{body}} \\ \rightsquigarrow\ \text{inV} \\ \rightsquigarrow\ \text{sEffect(node)} \end{array}\right) \\
\rightsquigarrow\ \text{outE}_{\texttt{exit}} \\
\rightsquigarrow\ \text{inV@}\texttt{exit} \\
\rightsquigarrow\ \text{flatMap}\left(\begin{array}{l} \text{cap}_\texttt{r} \\ \rightsquigarrow\ \text{unfold} \end{array}\right) \\
\rightsquigarrow\ \text{addE}\ \begin{array}{l} \textit{label} = \texttt{flow}, \\ \textit{to} = \texttt{exit} \end{array}
\end{array}\right) \\
\text{node} &= \left(\begin{array}{l}
\texttt{\_\_@synB} \\
\rightsquigarrow\ \text{addV}_{\texttt{block}}\texttt{@newB} \\
\rightsquigarrow\ \text{sEffect}\left(\begin{array}{l}
\text{addE}\ \begin{array}{l} \textit{label} = \texttt{assoc} \\ \textit{from} = \texttt{synB} \\ \textit{to} = \texttt{newB} \end{array} \\
\rightsquigarrow\ \text{flatMap}\left(\begin{array}{l} \text{cap}_\texttt{r} \\ \rightsquigarrow\ \text{unfold} \end{array}\right) \\
\rightsquigarrow\ \text{addE}\ \begin{array}{l} \textit{label} = \texttt{flow} \\ \textit{to} = \texttt{newB} \end{array}
\end{array}\right) \\
\rightsquigarrow\ \text{coalesce}\left(\begin{array}{c} \text{cond} \\ \text{block} \\ \left(\begin{array}{l} \text{sEffect}(\text{clear}_\texttt{r}) \\ \rightsquigarrow\ \text{aggregate}_\texttt{r} \end{array}\right) \end{array}\right)
\end{array}\right) \\
\text{cond} &= \left(\begin{array}{l}
\text{sack}_{\text{in}} \\
\rightsquigarrow\ \text{select}_{\texttt{synB}} \\
\rightsquigarrow\ \text{has}_{\textit{label}=\texttt{Conditional}} \\
\rightsquigarrow\ \text{outE}_{\{\texttt{true},\texttt{false}\}} \\
\rightsquigarrow\ \text{order}_{\{\textit{in}=\texttt{asc},\textit{by}=\texttt{id}\}} \\
\rightsquigarrow\ \text{inV} \\
\rightsquigarrow\ \text{flatMap}\left(\begin{array}{l} \text{sEffect}\left(\begin{array}{l} \text{sack}_{\text{out}} \\ \rightsquigarrow\ \text{aggregate}_\texttt{r} \end{array}\right) \\ \rightsquigarrow\ \text{flatMap(node)} \end{array}\right) \\
\rightsquigarrow\ \text{sEffect}(\text{clear}_\texttt{r}) \\
\rightsquigarrow\ \text{aggregate}_\texttt{r}
\end{array}\right) \\
\text{block} &= \left(\begin{array}{l}
\text{sEffect}(\text{clear}_\texttt{r}) \\
\rightsquigarrow\ \text{aggregate}_\texttt{r} \\
\rightsquigarrow\ \text{select}_{\texttt{synB}} \\
\rightsquigarrow\ \text{outE}_{\{\texttt{nest},\texttt{statement}\}} \\
\rightsquigarrow\ \text{order}_{\{\textit{in}=\texttt{asc},\textit{by}=\texttt{id}\}} \\
\rightsquigarrow\ \text{inV} \\
\rightsquigarrow\ \text{flatMap}\left(\begin{array}{l} \text{flatMap(node)} \\ \rightsquigarrow\ \text{fold} \\ \rightsquigarrow\ \text{sEffect}(\text{clear}_\texttt{r}) \\ \rightsquigarrow\ \text{sEffect}\left(\begin{array}{l} \text{unfold} \\ \rightsquigarrow\ \text{aggregate}_\texttt{r} \end{array}\right) \end{array}\right) \\
\rightsquigarrow\ \text{tail}_1 \\
\rightsquigarrow\ \text{unfold}
\end{array}\right)
\end{aligned}
$$

Figure 2: CFG extraction in Gremlin

Ψ_{node}, Ψ_{cond}, and Ψ_{block} are corresponding to Algorithm 2 in Section 3.2. Here, `synB` and `newB` are just arbitrary variables (names of local store keys). Ψ_{node} assigns the local `synB` name to the AST node in its traverser (there is always just one) and creates a new CFG node corresponding to the AST node (again with a name). It links the two with an `assoc`-edge, and sends `flow`-edges from the traversers stored in `r`. This is the CFG entry node when Ψ_{node} is called initially, and later in the recursion `r` stores those CFG nodes that directly precede `newB` (see later). Finally,

Ψ_{node} calls σ_{coalesce} (see Rule 6): first, Ψ_{cond} is called, if terminates early (`newB` is not a conditional), then Ψ_{block} is called, and if that also terminates early (`newB` is an empty block), then we simply store `newB` in register `r`. This means that `newB` will be the direct predecessor of a CFG node created later (or of the CFG exit). The returned node will be that of σ_{coalesce} (`newB` in case both subtraversals terminate early).

Ψ_{cond} first stores its traverser (a CFG node) into a path-local store, and makes `synB` the current traverser. In case this node is not a conditional, Ψ_{cond} terminates early, otherwise it processes its branches. Ordering the branches ensures that we can retrace later which CFG flows correspond to the true and false branches (as a lengthier alternative we could store the labels and use them to label the flow appropriately). For each branch, we copy the CFG node (of the conditional) from the local store to the global `r` and recursively invoke Ψ_{node}: this means that Ψ_{node} will create a CFG of the branch, and send a flow from the conditional to the source-node of the branch CFG. Finally, Ψ_{cond} stores into `r` the traversers returned from applying Ψ_{node} to the branches, and also returns these.

Finally, Ψ_{block} iterates over all the children (specifically statements and nested blocks) of the syntactic blocks and sets up the flows between them. It first stores the current CFG node into `r`, as this will be the predecessor node for the CFG of the first child. In case there are no children (the current block is an empty block), the subtraversal terminates early. Children are traversed in ascending order and per Rule 5 σ_{flatMap} ensures that each child is fully processed before we start processing the next. After Ψ_{node} was invoked for a child, we store all returned traversers into `r` as these will be the preceding CFG nodes for the next child. The only traversers we want to return are the traversers returned from the processing of the last child (these will be the predecessors of the continuation CFG). For this reason, each child is mapped to a collection of all its return nodes (using σ_{fold}): the traversers after σ_{flatMap} will be collections of CFG nodes (not just nodes). Then, σ_{tail_1} will keep only the last collection, which is then unfolded so that we return the CFG nodes instead of a collection. Note that these are also stored in `r`.

4.3 Proving the algorithm

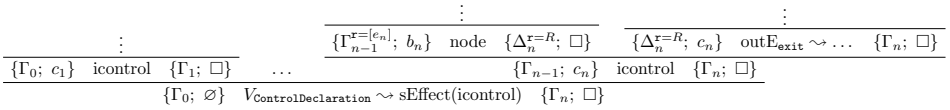

Figure 3: Structure of the proof tree

In this section, we describe the idea of proving the correctness of our CFG extraction algorithm in Section 4.2. Using our formalisation of Gremlin semantics in Section 4.1, we can formally prove that each traversals in the algorithm will result in

a state, that can be shown to guarantee correctness inductively. Despite operating with non-linear data structures, such as graphs, the recursive top-down traversal of ASTs guarantees termination and allows us to use structured correctness proofs. We include a complete, semi-formal proof tree in Appendix A, together with an informal description.

The claim of correctness is that in any Γ_0 initial state – containing the correct AST of each control declaration c_i –, the algorithm ultimately produces a Γ_n state, that contains also the correct CFG of each c_i control declaration.

Figure 3 formally depicts the beginning of the proof, including an inductive $(n-1) \longrightarrow n$ step. To prove the claim, we construct a proof tree, which consist of explicitly (formally) writing out the preconditions and postconditions (effects) of each steps of the algorithm, as dictated by the formal Gremlin semantics.

In the root of the proof tree, we insert the claim itself: starting from Γ_0 and an empty traverser list, the algorithm should lead to Γ_n and an arbitrary traverser list (denoted using the wildcard or placeholder symbol $\square$). As per Rule 5, during this step the c_i syntax nodes of each P4 control declarations are stacked into the traverser list, and Ψ_{icontrol} is applied to each of them one by one. Since this step is using induction, the proof tree depicts only the processing of the last declaration (c_n). The inductive assumption is that the former $n-1$ applications of traversal Ψ_{icontrol} result in a Γ_{n-1} state that is already correct, apart from missing the CFG of the nth declaration. For the last Ψ_{icontrol} to be correct, we need to prove that Ψ_{node} is correct, starting from $\Gamma_{n-1}^{r=[e_n]}$ (i.e. the state we get from Γ_{n-1} by storing CFG entry e_n in registry `r` as per the first steps of Ψ_{icontrol}). Given that Ψ_{node} results in some state $\Delta_{n-1}^{r=R}$ (where R is a variable, denoting – in case Ψ_{node} is correct – the set of return points), we also need to prove that in this state, the last steps of Ψ_{icontrol} (i.e. the steps that link return points to the exit node) result in the the expected Γ_n. In turn, both propositions can be proved by building the proof tree further. Due to its technical nature, we do not include complete proof tree here, but for our readers interested in the details, we include it – together with an informal description – in Appendix A.

5 Conclusion

Summary In this work, we defined deep CFGs, and presented a CFG extraction algorithm that superimposes CFGs over ASTs inside a Gremlin graph database: this way all information of the AST is at hand during CFG traversals, and can be readily accessed through a uniform graph interface. We already rely on deep CFGs for code generation in our latest paper on P4 cost analysis [11]. There, we used the CFG to transform P4 code to a custom, low-level instruction language which is then passed to a sophisticated probabilistic model checker tool. In the implementation of that transformation, we simply traverse the CFG, and at each node, we follow the association links in the graph to collect further information needed to create instructions from the node. Our other goal in the current paper was to explore how the expressive power of Gremlin can be used for specifying and

proving fairly complex static analysis procedures. For this reason, we formalised our algorithm as an executable Gremlin query, and used the formal semantics of Gremlin to formally prove the correctness of our CFG extraction algorithm.

Limitations We highlight two limitations of this CFG extraction algorithm. First, our Gremlin formalisation relies on (shallow) recursion to realise depth-first traversal. Recursive queries are made possible only by host language features such as lambdas, and these are not serialisable. This means that in a client-server environment this query has to be stored on the server-side as a stored procedure. Second, complex Gremlin traversals are – at least in our experience – not easy maintain. For example, the graph query in Figure 2 has to keep track of program state as it stores return points in the global register `r`. If a developer intends to extend the algorithm for handling further P4 language elements, they have to understand how this register is used in the algorithm, in order to make sure not to cause unintended side effects. For production environments, a simpler – although less efficient – approach is to iterate over the AST in multiple stages, e.g. first discovering the R return points of each node n, and in a second iteration link R to the continuation of n. This way the developer can either store global state in the host language (arguably more suitable for handling global state) or eliminate it entirely by storing intermediate information in the graph. Even without considering state and side effects, a sequence of simple, small queries is usually much easier to read, test, and debug.

Future work We started to utilise the deep CFG concept in this paper in the code generation phase of our model checking-based cost analysis [11]. One missing element for this is intraprocedural control flow (i.e. translating function calls), which we currently handle with an ad-hoc solution. One advantage of the CFG being superimposed on the AST is that this element can be added in as a separate analysis on the AST, and we can reach this information in every traversal. We would also like to explore more of the possibilities this opens up, including graphs resulting from data flow analysis (see Section 2) as well.

References

[1] Aho, A.V., Lam, M.S., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] Amighi, A., de C. Gomes, P., Gurov, D., and Huisman, M. Sound control-flow graph extraction for Java programs with exceptions. In Eleftherakis, G., Hinchey, M., and Holcombe, M., editors, *Software Engineering and Formal Methods*, pages 33–47, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-33826-7_3.

[3] Angles, R. and Gutierrez, C. Survey of graph database models. *ACM Computing Surveys*, 40(1), 2008. DOI: 10.1145/1322432.1322433.

[4] Appel, A.W. SSA is functional programming. *SIGPLAN Not.*, 33(4):17–20, 1998. DOI: 10.1145/278283.278285.

[5] Baier, C. and Katoen, J.-P. *Principles of Model Checking (Representation and Mind Series)*. ISBN: 978-0-262-02649-9. The MIT Press, 2008.

[6] 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 *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, page 1–8. IEEE Press, 2020. DOI: 10.1109/NOMS47738.2020.9110307.

[7] 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 Comput. Commun. Rev.*, 44(3):87–95, 2014. DOI: 10.1145/2656877.2656890.

[8] Dumitrescu, D., Stoenescu, R., Negreanu, L., and Raiciu, C. Bf4: Towards bug-free P4 programs. In *Proceedings of SIGCOMM'20*, page 571–585, New York, NY, USA, 2020. Association for Computing Machinery. DOI: 10.1145/3387514.3405888.

[9] Lukács, D., Pongrácz, G., and Tejfel, M. Are graph databases fast enough for static P4 code analysis? In *Proceedings of the 11th International Conference on Applied Informatics*, pages 213–223. CEUR Workshop Proceedings, 2020. URL: http://ceur-ws.org/Vol-2650/#paper22.

[10] Lukács, D., Pongrácz, G., and Tejfel, M. Control flow based cost analysis for P4. *Open Computer Science*, 11:70–79, 2020. DOI: 10.1515/comp-2020-0131.

[11] Lukács, D., Pongrácz, G., and Tejfel, M. Model checking-based performance prediction for P4. *Electronics*, 11(14), 2022. DOI: 10.3390/electronics11142117.

[12] Nielson, H.R. and Nielson, F. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., USA, 1992.

[13] P4 Language Consortium. basic_routing-bmv2.p4, a small test case for the official P4 reference compiler, P4C, 2018. URL: https://github.com/p4lang/p4c/blob/master/testdata/p4_16_samples/basic_routing-bmv2.p4.

[14] Rodriguez, M.A. The gremlin graph traversal machine and language (invited talk). *Proceedings of the 15th Symposium on Database Programming Languages*, 2015. DOI: 10.1145/2815072.2815073.

[15] Söderberg, E., Ekman, T., Hedin, G., and Magnusson, E. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Sci. Comput. Program.*, 78(10):1809–1827, 2013. DOI: 10.1016/j.scico.2012.02.002.

[16] Tóth, M. and Bozó, I. Building dependency graph for slicing Erlang programs. *Periodica Polytechnica Electrical Engineering*, 55(3-4):133–138, 2011. DOI: 10.3311/pp.ee.2011-3-4.06.

Appendix A Proof of correctness

In this section, we give a semi-formal proof for the correctness of our CFG extraction algorithm in Section 4.2. Using our formalisation of Gremlin semantics in Section 4.1, we formally prove that each traversals in the algorithm will result in a state, that can be shown to guarantee correctness inductively. Despite operating with non-linear data structures, such as graphs, the recursive top-down traversal of ASTs guarantees termination and allows us to use structured correctness proofs. We depict these formal proofs as proof trees in Figures 4, 5, 6, and 7. We describe the contents of these proof trees, and give the informal induction steps in the course of this section.

In the proof trees, we heavily rely on the notation introduced in the previous sections (especially the semantics of Gremlin). In addition to that, we utilize a small number of additional notations: these are related to the semantics of individual Gremlin steps, or to the proof tree syntax.

Notations. Earlier, we used Γ and its subscripted variants to denote the state of global memory: in the proof, we also use Δ and $\tilde{\Delta}$ to denote intermediate memory states. In the proofs, we rarely write out the state explicitly, rather we treat it as a set of statements which are true for the state. For example, we write $\Gamma^{\texttt{r}=R}$ to highlight the assumption that in Γ, the value of register `r` is R. `synB`, `newB`, `exit` are referring to the local store keys assigned in CFG algorithm. In order to not to waste variable names, we use the "wildcard" variable $\square$ as a placeholder, to denote intermediate values (usually traverser lists) that are not important (not used elsewhere). Since it is a wildcard symbol, two occurrences of $\square$ may contain different values.

In the proof tree leafs, ✓ means that an axiom has been reached (the tree branch has been proved); numbers between parentheses (e.g. (1), (2), (3)) mean that the proof is continued in another proof tree (with the number in its root); dots (...) mean some steps were omitted (we elaborate these in the explanations). Note that in some cases we compressed multiple steps into one movement, specifically where effects of the individual steps were simple (e.g. in case of $\sigma_{outE} \rightsquigarrow \sigma_{inV}$).

As noted before, σ_{coalesce} (Rule 6) is a higher-order step (it is parameterised with sub-traversals), that returns the traversers from the first successful sub-traversal. We use the $\underset{\rightarrow}{\vee}$ shorthand to denote its semantics: given Ψ,Ψ' traversals, the proposition $(\{\Delta_0; T_0\}\Psi\{\Delta_n; T_n\}) \underset{\rightarrow}{\vee} (\{\Delta_0; T_0\}\Psi'\{\Delta_n; T_n\})$ is solved for (Δ_0, T_0, Δ_n, T_n) either by solving the left-hand side operand of $\underset{\rightarrow}{\vee}$, or in case this would result in $T_n = \varnothing$, then by solving the right-hand side operand.

In the rest of this section, we go through each of the traversals defined in Section 4.2, state its correctness and prove that claim. In each of the proofs, we refer to the corresponding formal proof tree, include a descriptive commentary to explain and clarify the proof tree contents, and then finish the proof with an informal argument about satisfaction of the requirements posed by the proof tree. Our first claim concerns Ψ_{control} and Ψ_{icontrol}. The precondition and postcondition posed by this claim refers to our definition of ASTs and control flow graphs in Section 3.1.

Claim 1. *Given that Γ_0 contains a correct AST, together with previously prepared entry and exit nodes (e_i and f_i) for each control declarations c_i in the tree, Γ_n contains for each of c_i in the tree the CFG of the control declaration, rooted in the entry node e_i and all its return points linked to the exit node f_i.*

Proof. Figure 4 formally depicts the inductive ($(n-1) \longrightarrow n$) step of this proof. The rest of this paragraph is commentary for that diagram. In short, we process each control declaration one after the other, and for each, we run Ψ_{node} and link its returned return points to the exit node. In detail, the syntax nodes of each P4 control declarations are stacked into the traverser list, and Ψ_{icontrol} is applied to each of them one by one (as per Rule 5). Each application is expected to fill Γ_0 with the CFGs of each declaration c_i, ultimately reaching Γ_n containing all the CFGs. The final contents of the traverser list is irrelevant (we denote it with the "wildcard" variable $\square$ that can match anything). Since this is the inductive step, the proof tree depicts only the processing of the last declaration (c_n). To process the last declaration, we store the corresponding CFG entry (e_n) into the global variable r (used later in n), and move the traverser to the first syntax block (b_n) of the declaration. We expect side effect traversal Ψ_{node} to solve the rest of the problem, resulting in global state Δ_n, in which global variable r is set to the set of those CFG nodes (R) that are the return points of the CFG built by Ψ_{node}. After that we move the traverser from c_n to the corresponding exit node f_n, save it to path-local name `exit`, replace the traverser list with R (path-local names are preserved) and send links from R to f_n, and the processing ends.

From this, we can see that the inductive step has two requirements. The first requirement is that Ψ_{node} produces the correct CFG of the nth declaration apart from missing the links between the return points and the exit node. The other requirement (base case) is that the former $n-1$ applications of traversal Ψ_{icontrol} result in a Γ_{n-1} state that is already correct, apart from missing the CFG of the nth declaration.

We give the rest of this proof informally. The first requirement (namely that *node* produces $\Delta_n^{\mathrm{r}=R}$ from Γ_{n-1} by creating the CFG of c_n rooted at entry e_n with R containing the return nodes of this CFG in R) is satisfied by Claim 2. To see that the inductive hypothesis (Γ_{n-1} has the correct CFGs for $c_1, \ldots, c_{n-1}$) is satisfied, recognize that c_1 is applied in state Γ_0, and state Γ_0 already has the 0 correct CFGs for $\varnothing$. (For a non-degenerate case, we can build a very similar proof tree for c_1, and see that Γ_1 has the correct CFG for $\{c_1\}$.)

Then, it follows that $\Gamma_n = \Delta_n^{\forall r_1 \ldots r_k \in R:\ ((r_1 \to f_n) \in E, \ldots, (r_k \to f_n) \in E)}$, s.t. r_i are

$$\dfrac{\dfrac{\vdots}{\{\Gamma_0;\ c_1\}\quad \text{icontrol}\quad \{\Gamma_1;\ \square\}}\quad \ldots\quad \dfrac{\dfrac{(1)}{\{\Gamma_{n-1}^{\mathtt{r}=[e_n]};\ b_n\}\quad \text{node}\quad \{\Delta_n^{\mathtt{r}=R};\ \square\}}\quad \dfrac{\dfrac{\checkmark}{\{\Delta_n^{\forall r_1 \ldots r_k \in R:\ ((r_1 \to f_n)\in E, \ldots, (r_k \to f_n)\in E)};\ \square\}\quad \varepsilon\quad \{\Gamma_n;\ \square\}}}{\{\Delta_n;\ R \times \{\mathtt{exit} = f_n\}\}\quad \text{addE}_{\substack{label\ =\ \mathtt{flow} \\ to\ =\ \mathtt{exit}}}\quad \{\Gamma_n;\ \square\}}}{\{\Delta_n^{\mathtt{r}=R};\ (c_n,\ \mathtt{exit} = f_n)\}\quad \text{flatMap}(\ldots) \rightsquigarrow \ldots \{\Gamma_n;\ \square\}}}{\{\Delta_n^{\mathtt{r}=R};\ c_n\}\quad \text{outE}_{\mathtt{exit}} \rightsquigarrow \ldots\quad \{\Gamma_n;\ \square\}}}{\{\Gamma_{n-1};\ c_n\}\quad \text{icontrol}\quad \{\Gamma_n;\ \square\}}}{\{\Gamma_0;\ \varnothing\}\quad V_{\mathtt{ControlDeclaration}} \rightsquigarrow \text{sEffect(icontrol)}\quad \{\Gamma_n;\ \square\}}$$

Figure 4: Proof of Claim 1

$$\dfrac{\dfrac{\dfrac{\dfrac{(2)}{\{\Delta_1; (x, \{\mathtt{synB} = b\})\}\ \text{cond}\ \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}}\quad \underset{\rightarrow}{\vee}\quad \dfrac{(3)}{\{\Delta_1; (x, \{\mathtt{synB} = b\})\}\ \text{block}\ \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}}\quad \underset{\rightarrow}{\vee}\quad \dfrac{\checkmark}{\{\Delta_1^{\mathtt{r}=[x]}; x\}\ \varepsilon\ \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}}}{\{\Delta_1;\ (x, \{\mathtt{synB} = b\})\}\quad \text{coalesce}(\ldots)\quad \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}}}{\{\Delta^{\mathtt{r}=U, x\in V};\ (x, \{\mathtt{synB} = b, \mathtt{newB} = x\})\}\quad \text{sEffect}(\ldots) \rightsquigarrow \ldots\quad \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}}\ \Delta_1 = \Delta^{\left(\begin{array}{l} x \in V, \\ \forall u \in U : (u \xrightarrow{\mathtt{flow}} x) \in E, \\ (b \xrightarrow{\mathtt{assoc}} x) \in E \end{array}\right)}}{\{\Delta^{\mathtt{r}=U};\ b\}\quad \text{node}\quad \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}}$$
$$(1)$$

Figure 5: Proof of Claim 2

return nodes, and f_n is exit node, and so Γ_n is the correct graph for $c_1, \ldots, c_n$. □

We now continue with stating the correctness of traversal Ψ_{node}. While we needed this claim in the proof of Claim 1, we will also use this in proving Claim 3 and 4. For this reason, we state it in a more general form than what is used in Claim 1. We make use of a heuristic, namely the recognition that Ψ_{node} is always called with one traverser. (On the other hand, r may contain multiple nodes: Ψ_{block} loads its results into r and may call Ψ_{node}, and Ψ_{node} can call Ψ_{cond} which is guaranteed to have multiple results.)

The returned traversers of Ψ_{node} are used in Ψ_{block} to return the return points of its last child. The contents of R are used by Ψ_{block} to correctly process the right sibling of b (we also use it in Ψ_{icontrol} to link the exit nodes).

Notice as well that while, at first glance, the proof of Ψ_{node} requires using mutual induction with the proofs of Ψ_{cond} and Ψ_{block} (because of mutually recursive calls), this can be easily eliminated. To linearise the induction, we just have to inline traversals Ψ_{cond} and Ψ_{block} in traversal Ψ_{node}. (Indeed, we only introduced these traversals to increase readability.)

Claim 2. *Given that U contains the predecessor nodes, then Ψ_{node} produces $\tilde{\Delta}^{\mathtt{r}=R}$ from Δ a correct CFG, rooted at some node x of a given syntax block b, links the predecessor nodes to x, and returns the return nodes of this CFG in R, and returns these as traversers as well.*

Proof. Figure 5 formally depicts the inductive $((n-1) \longrightarrow n)$ step of this proof. The rest of this paragraph is commentary for that diagram. Here, we expect that by calling Ψ_{node} in some state $\Delta^{\mathtt{r}=U}$ (that is, a global state where global variable r is set to a set of nodes U) with a traverser pointing to a syntax block b, Ψ_{node} produces the expected $\tilde{\Delta}$ state, with global variable r storing the expected R. We assign the path-local name `synB` to b, create a new CFG node x in the graph, and move the traverser to x in order to assign it to the path-local name `newB`. Then, in a σ_{sEffect}, we link all the predecessors in U to our new x, and set up the association between x and b as well, resulting in global state Δ_1. (Notice that both *block* and *cond* resets r, and *node* never directly calls itself, so since its contents are not used anymore, we can omit it from Δ_1.) Then, we call the σ_{coalesce} step in state Δ_1. As per the description of the $\underset{\rightarrow}{\vee}$ symbol in Rule 6, we call Ψ_{cond} which either correctly creates the rest of this CFG (if b is a conditional), or terminates without side-effects, in which case we call Ψ_{block}. Again, Ψ_{block} either correctly creates the rest of this CFG (if b is a non-empty block), or terminates without side-effects, in which case we expect b to be an empty block, and so our new x is the appropriate return point. In this last case, we modify Δ_1 by storing x in r, it also stays in the traverser list, and the traversal ends.

From this, we can see that the inductive step has three requirements. The first two requirement is that both Ψ_{cond} and Ψ_{block} continues the correct processing of b according to its type, and they result in the expected state $\tilde{\Delta}^{\mathtt{r}=R}$. The third requirement (posed by Ψ_{cond} and Ψ_{block}) is the satisfaction of the base step, i.e. that

the last element of the chain is processed correctly. In more precise term, given b is an empty block such that it is the syntactic child of a previously processed syntactic node, and U contains the return points of the CFG resulting from this, then Ψ_{node} ends in the expected state.

We give the rest of this proof informally. The first requirement (namely that in case b is a conditional, *cond* produces $\tilde{\Delta}$ by including in Δ_1 the nodes and edges of the branches and sets their return nodes as return nodes in R, and returns these as traversers as well) is satisfied by Claim 3. Then, the claim is true, since $\{x \in V, \forall u \in U : u \xrightarrow{\texttt{flow}} x \in E, b \xrightarrow{\texttt{assoc}} x \in E\} \subset \Delta_1$. The second requirement (namely that in case b is a non-empty block, *block* produces $\tilde{\Delta}$ by including all remaining nodes and edges of the nested statements and blocks in Δ_1, and sets the last nest as the return node in R, and returns these as traversers as well) is satisfied by Claim 4. Then, the claim is true, since $\{x \in V, \forall u \in U : (u \xrightarrow{\texttt{flow}} x) \in E, (b \xrightarrow{\texttt{assoc}} x) \in E\} \subset \Delta_1$.

The third requirement (base case) is satisfied, because in every step we get closer to the bottom of the AST (containing only empty blocks or statements), and since in case b is not a conditional – as Ψ_{cond} terminated before doing any side effect – , and b is an empty block – as Ψ_{block} terminated before doing any side effect – , then $\tilde{\Delta} = \Delta^{\mathbf{r}=[x],\ x\in V,\ \forall u\in U:(u\xrightarrow{\texttt{flow}}x)\in E,\ (b\xrightarrow{\texttt{assoc}}x)\in E}$, and the returned traverser is x. That is, one CFG node x is created, it is linked from each $u \in U$, and associated with syntax node b, and finally, $R = [x]$ contains the correct return node since the only node x should be the return node. □

With that, we now proceed to state the correctness of Ψ_{cond} and Ψ_{block}, starting with the former one. As mentioned, these proofs seemingly have mutual dependence with the proof of Ψ_{node}, but we can easily reduce mutual induction to linear induction by inlining traversals Ψ_{cond} and Ψ_{block} into traversal Ψ_{node}.

Claim 3. *In case b is a conditional, then cond produces $\tilde{\Delta}^{\mathbf{r}=R}$ by including in Δ the nodes and edges of the branches, sets their return nodes as return nodes in R, and returns these as traversers as well. Otherwise terminates without side-effects.*

Proof. Figure 6 formally depicts the inductive $((n-1) \longrightarrow n)$ step of this proof. The rest of this paragraph is commentary for that diagram. We expect that in some state Δ, with one traverser standing on a CFG x (created earlier by Ψ_{node} for syntactical node b), Ψ_{cond} produces the expected state $\tilde{\Delta}^{\mathbf{r}=R}$. We treat the termination requirement in the informal part of the proof. First, we store in x in the path-local store, also known as sack ($\varsigma(\cdot)$) and set the current traverser to b. (We have to use a path-local store, as the processing of the branches may modify the global stores (e.g. r), and in that case we would lose our reference to x after the processing the branch.) We continue the processing in case $\mathit{label}(b) = \texttt{Conditional}$ (otherwise the algorithm terminates). In this case, it is always true (by our definition of the AST) that b has two branches: one true-branch starting in syntactical node y, and one false-branch starting in syntactical node n. We move a traverser to each of these nodes (the sack is preserved). In the σ_{flatMap} step, we process the two branches (first the branch of y, then the branch of n) by

traversal Ψ_{node} (storing x in r, as Ψ_{node} will have to link x to the produced CFG as predecessor), with the expectation that Δ_1 contains the CFG produced for y, and Δ_2 contains the CFGs both for y and n, and that Ψ_{node} returns y's CFG return points in T_y and n's CFG return points in T_n. (The processing order of the two branches does not matter as long as it is consistent across all applications of the traversal.) After that, we store the return points of both of T_y and T_n in r, and we also return this return points as traversers.

From this, we can see that the inductive step has two very similar requirements regarding the correct processing of y and n by Ψ_{node}. It is worth noting the analysis of the two branches never interferes with each other: r is reseted and *node* will traverse different subtrees of the tree. A possible third requirement of a base case (it is possible that the called Ψ_{node} may call Ψ_{cond} again, but ultimately Ψ_{node} will stop when statements or empty blocks are encountered) was already discussed in the proof of Ψ_{node}.

We give the rest of this proof informally. The first requirement (namely that *node* produces Δ_1 from Δ by creating the CFG of the true branch y, linking it to the (conditional) CFG node x, and returns in traverser list T_y all the return nodes of the CFG, is satisfied by Claim 2. Similarly, the second requirement (namely that *node* produces Δ_2 from Δ_1 by creating the CFG of the false branch y, linking it to the (conditional) CFG node x, and returns in traverser list T_n all the return nodes of the CFG is also satisfied by Claim 2.

Then it follows, that Δ_2 contains CFGs of the branches rooted at x, and $R = T_y \cup T_n$ contains the return nodes of the branches, and returns these as traversers as well.

Regarding the termination requirement of the claim, in case b is not a conditional, the *has* filter fails and the traversal terminates without side-effects. □

Finally, we state the correctness of Ψ_{block}. This is a case where we use induction in two axes: Ψ_{block} as part of a chain initiated from Ψ_{node}, and at the same time, use induction in proving that a sequence of sibling nodes are processed correctly.

Claim 4. *In case b is a non-empty block, assume that block produces $\tilde{\Delta}^{\mathbf{r}=R}$ by including all remaining nodes and edges of the nested statements and blocks in Δ, and sets the last nest as the return node in R, and returns these as traversers as well. Otherwise terminates without side-effects.*

Proof. Figure 7 formally depicts the inductive $((n-1) \longrightarrow n)$ step of this proof. The rest of this paragraph is commentary for that diagram. We expect that in some state Δ, with one traverser standing on a CFG x (created earlier by Ψ_{node} for syntactical node b), Ψ_{block} produces the expected state $\tilde{\Delta}^{\mathbf{r}=R}$. We treat the termination requirement in the informal part of the proof.

First, we store in x in the global store r (we expect this to be modified as children of b are processed and use its contents to create links between the children), and set the traverser list to S, denoting the blocks and statements nested in b (in case there is non, the traversal terminates). Through ordering S we ensure that σ_{flatMap} processes these nested nodes from left-to-right (guaranteed by definition of

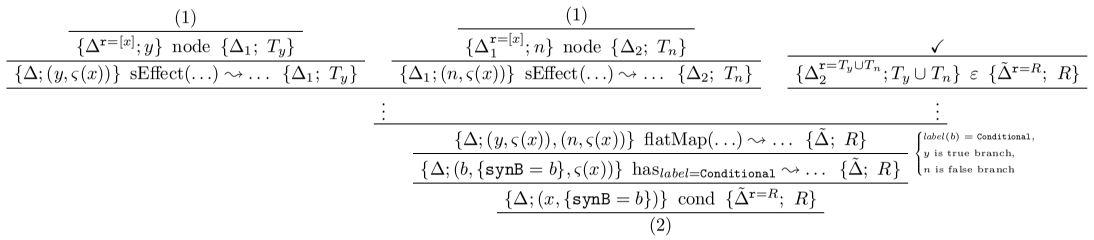

Figure 6: Proof of Claim 3

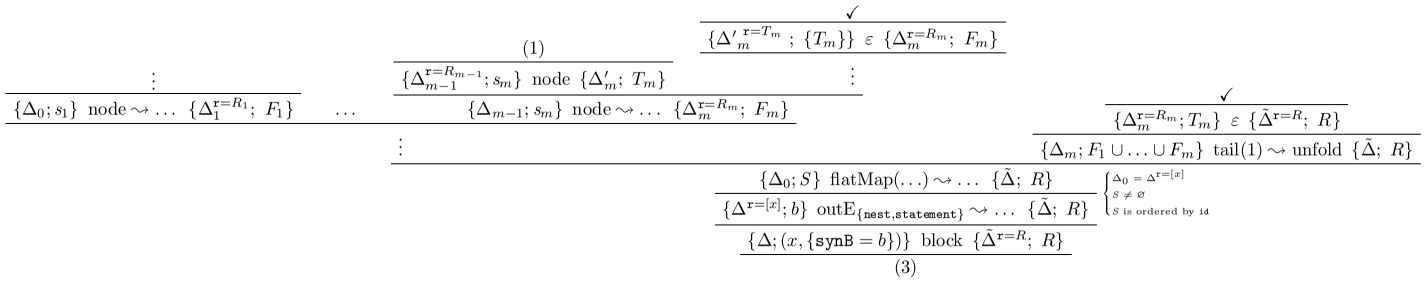

Figure 7: Proof of Claim 4

the AST). σ_{flatMap} starts out in state Δ_0 when it starts processing the first nested node, and finishes in state Δ_m after processing the last nested node.

We will check that the σ_{flatMap} step processes S correctly by induction, and so we only included the proof tree corresponding to the last child (s_m). Here, we expect that given all previous siblings have their CFGs in global state Δ_{m-1} with r storing the return points (R_{m-1}) of the closest previous sibling's CFG, and Ψ_{node} produces Δ'_m containing (and properly linking) all the CFGs of the children, and returns the return points (T_m) of the last child of b as the traverser list. Then this nested traversal stores T_m into r and at the same folds T_m into a single collection-element $F_m = \{T_m\}$ that is returned in the traverser list. After σ_{flatMap} processed everyone, we are in state Δ_m with r storing the return points of the last child of b, and the traverser list containing all the collection-elements return from processing the children. Now, keep only the last traverser (a collection), and unfold this collection again into the individual traversers (still pointing to the return points of the last child's CFG), and return these.

From this, we can see that the inductive step has two requirements. The first – about Δ'_m – is that Ψ_{node} creates a correct node CFG for any child of b, links this CFGs node from the return points of the previous child's CFGs, and returns the return points of the currently processed child's CFG as the traverser list, while also storing it in r. The other requirement (base case) is that the former $n-1$ applications of traversal Ψ_{node} result in a Δ_{m-1} state contains the correct CFGs of all the previous children.

We give the rest of this proof informally. The first requirement (namely that *node* produces Δ'_m from $\Delta_{m-1}^{\mathtt{r}=R_{m-1}}$ by creating the CFG of s_m, linking it to the nodes in R, and returns in traverser list T_m all the return nodes of the CFG) is satisfied by Claim 2. The next requirement is that the inductive hypothesis ($\Delta_{m-1}^{\mathtt{r}=R_{m-1}}$ contains the chain of CFGs produced from the first $m-1$ blocks of b, and R_{m-1} contains the return points of the last CFG in this chain) is satisfied. Again, we may consider the degenerate case (Δ_0, with $R = [x]$ satisfies the hypothesis) or alternatively, we can build a very similar proof tree for s_1, and from the first requirement it follows that Δ_1 has the correct CFG for $\{s_1\}$, with r storing its T_1 return points, and F_1 in the traverser list is T_1 folded.

Then, the claim follows, since $\tilde{\Delta}^{\mathtt{r}=R} = \Delta_m^{\mathtt{r}=R_m} = {\Delta'}_m^{\mathtt{r}=T_m}$, with $R = R_m = T_m$ also being returned.

Regarding the termination requirement of the claim, in case b is an empty block, *outE* maps to $\varnothing$ and the traversal terminates without side-effects. □