Field-Sensitive Program Slicing[⋆]

Carlos Galindo[a], Jens Krinke[b], Sergio Pérez[a], Josep Silva[a,*]

[a] *VRAIN, Universitat Politècnica de València, Camino de Vera s/n, 46022, Valencia, Spain*
[b] *CREST Centre, University College London, Gower Street, London, WC1E 6BT, United Kingdom*

Abstract

The granularity level of the traditional program dependence graph (PDG) for composite data structures (tuples, lists, records, objects, etc.) is inaccurate when slicing their inner elements. We present the constrained-edges PDG (CE-PDG) that addresses this accuracy problem. The CE-PDG enhances the representation of composite data structures by decomposing statements into a subgraph that represents the inner elements of the structure, and the inclusion and propagation of data constraints along the CE-PDG edges allow for accurate slicing of complex data structures. Both extensions are conservative with respect to the traditional PDG, in the sense that all slicing criteria (and more) that can be specified in the PDG can be also specified in the CE-PDG, and the slices produced with the CE-PDG are always smaller or equal to the slices produced by the PDG. An evaluation of our approach shows a reduction in the size of the slices of around 10%.

Keywords: Program Analysis, Program Slicing, Composite Data Structures.

1. Introduction

The *Program Dependence Graph* (PDG) [1, 2] represents the statements of a program as a collection of nodes, and their control and data dependencies as edges. The PDG is used in *program slicing* [3, 4], a technique for program analysis and transformation whose main objective is to extract from a program the set of statements, the so-called *program slice* [5], that affect the values of a set of variables v at a program point p ($\langle p, v\rangle$), which is known as the *slicing criterion* [1]. Program slicing is applied in many disciplines such as software maintenance [6], debugging [7], code obfuscation [8], and program specialization [9], among others.

The original PDG is not able to handle all the features that most modern programming languages offer. Therefore, several extensions and enhancements of the PDG have been proposed to represent features like arbitrary control-flow [10, 11]; exception handling [12, 13]; interprocedural behaviour [14, 15]; or concurrency [16, 17]; among others. Nevertheless, there is still a largely unaddressed problem that is a source of imprecision and that affects all programming languages: the slicing of composite data structures. Finite composite data structures can be atomized [18] and then sliced as usual, however, infinite or recursive data structures cannot be atomized and slicing them is therefore imprecise.

In this paper, we propose a general method that solves the problem of accurately representing and slicing any composite data structure, even if it is recursive (infinite data structures can be also sliced) or if it is collapsed and expanded again (we solve the *slicing pattern matching* problem [19], which is explained in Section 2). The key ideas are (i) to expand the PDG with new nodes to precisely represent the subexpressions of the data structures, and (ii) to introduce the concept of *constrained edges*: we label the PDG edges with information about the data structures so that this information can be used at slicing time to know exactly which edges should be traversed. We call the new resulting graph the *Constrained-Edges PDG (CE-PDG)*. Finally, (iii) we provide a new slicing algorithm that takes advantage of constrained edges, limiting the traversal when necessary, and obtaining more accurate slices in the presence of composite data structures.

The main goal of our technique is to handle recursive data structures in combination with pattern matching to make program slicing field-sensitive in order to improve the accuracy of slicing programs with composite data structures.

This paper is an extended version of work [20] presented at the 20th International Conference on Software Engineering and Formal Methods (SEFM 2022). In this version, we extend the technical results with new formalizations, theoretical results, and their proofs. Moreover, we provide extended examples, showing that, e.g., other state-of-the-art techniques, such as atomization, cannot solve the addressed problem. The slicing algorithm pro-

[⋆]This work has been partially supported by grant PID2019-104735RB-C41 funded by MCIN/AEI/10.13039/501100011033 and by the *Generalitat Valenciana* under grant CIPROM/2022/6 (FassLow). Carlos Galindo was partially supported by the Spanish Ministerio de Universidades under grant FPU20/03861.

[*]Corresponding author

Email addresses: cargaji@vrain.upv.es (Carlos Galindo), j.krinke@ucl.ac.uk (Jens Krinke), serperu@dsic.upv.es (Sergio Pérez), jsilva@dsic.upv.es (Josep Silva)

posed has been revised and is more efficient and precise than the one presented before. All experiments have been repeated to evaluate the new performance, which has improved by one order of magnitude.

The rest of the paper is structured as follows: Section 2 demonstrates the problems in slicing composite data structures. Section 3 gives a short introduction to program slicing with basic definitions. Section 4 presents the CE-PDG and how it is used for slicing. Section 5 presents an implementation and an empirical evaluation of the proposed technique. It is followed by a discussion of related work (Section 6) and conclusions (Section 7).

2. Slicing Composite Data Structures

In this section, we show the inaccuracy problems caused by the traditional PDG when it is used to slice programs with complex data structures.

It is important to remark that these problems can be studied and solved at the level of the PDG (i.e., for intraprocedural programs). Because we can present the fundamental ideas and solutions of field-sensitive slicing at this level, we omit the more complex representation in the *System Dependence Graph* (SDG) [21] (i.e., for interprocedural programs). In this way, we keep the presentation easier to understand, avoiding the complexity introduced by the SDG (procedure calls, input/output edges, summary edges...). Of course, an extension of our work for the SDG is possible and will increase the precision of our technique by propagating dependencies throughout procedures.

Example 1 (PDG's Composite Data structures). Consider the four fragments of code with different data structures shown in Figure 1. We are interested in the values computed at the slicing criterion (the underlined variable in blue). The only part of the code that can affect the slicing criterion (i.e., the minimal slice) is coloured in green. Nevertheless, the slice computed with the traditional PDG contains the whole program in the four cases.

The complexity of these structures, together with the lack of granularity of the PDG (each node of the PDG represents a statement) results in a lack of accuracy when slicing these structures. Slicing algorithms cannot remove unnecessary inner components of statements using only the information of the PDG. Unfortunately, this is only one part of the imprecision, because all variables wrongly captured by the slice trigger a snowball effect, effectively including in the slice all the parts of the program that potentially influence them. For instance, in Figure 1c, including in the slice variable `arg` implies also including in the slice each expression that is passed as argument to `foo`.

In some cases, it is possible to solve the situation with a program transformation [22, 23, 24]. For instance, in Figure 1a we could replace `person = {"John",36};` by:

```
foo() {
  struct S {
    string name;
    int age;
  };
  S person = {"John",36};
  int maxAge = person.age;
  std::cout << maxAge;
}
```

(a) Records (C++)

```
enum Light {
  Red = 0,
  Yellow = 1,
  Green = 2
}
void Main() {
  Light pass = Light.Yellow
             | Light.Green;
  Console.WriteLine(pass);
}
```

(b) Enums (C#)

```
void foo(int arg){
  int[] nums = {2,arg,27};
  int x = nums[2];
  System.out.println(x);
}
```

(c) Arrays (Java)

```
class Person:
  def __init__(self,name,age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1.age)
```

(d) Objects (Python)

Figure 1: Slicing composite data structures (slicing criterion underlined and blue, minimal slice in green).

```
1 foo(X,Y) ->
2   {A,B} = {X,Y},
3   Z = {[8],A},
4   {[C],D} = Z.
```

(a) Original Program

```
1 foo(X,Y) ->
2   {A,B} = {X,Y},
3   Z = {[8],A},
4   {[C],D} = Z.
```

(b) PDG Slice

```
1 foo(X,Y) ->
2   {A,B} = {X,Y},
3   Z = {[8],A},
4   {[C],D} = Z.
```

(c) Minimal Slice

Figure 2: Slicing Erlang tuples (slicing criterion underlined and blue, slice in green)

`person.name = "John"; person.age = 36;`. Or, similarly in Figure 1c, we could replace `nums = {2,arg,27};` by `nums[0] = 2; nums[1] = arg; nums[2] = 27;`. This transformation, called *atomization* [18], decomposes data structures in simpler assignments for each of their components. It uses the qualified name `person.age` or the indexed array `nums[0]` as the name of an independent variable [24]. Note that only finite data structures can be atomized. An alternative approach uses the AST nodes of a program as PDG nodes [19]. Unfortunately, despite solving several problems, these approaches are also imprecise because they are unable to resolve the most problematic constructs: recursive (infinite) data types and pattern matching. Let us illustrate the problem induced by pattern matching with an example.

Example 2 (Pattern matching). Consider the fragment of Erlang code in Figure 2a, where we are interested in the values computed at variable `C` (the slicing criterion is $\langle 4, \texttt{C} \rangle$). The only part of the code that can affect the values at `C` (i.e., the minimal slice) is coloured in green in Figure 2c. Nevertheless, the slice computed with the PDG (shown in Figure 2b) contains the whole program. This is again a potential source of more imprecisions outside this function because it wrongly includes in the slice the parameters of function `foo` and, thus, in calls to `foo` their arguments and the code in which they depend are also

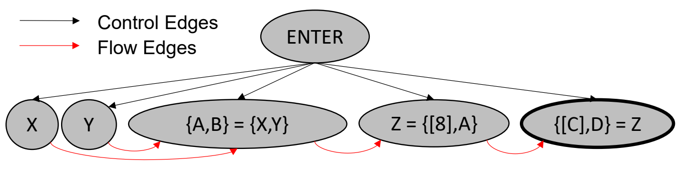


Figure 3: Erlang program and corresponding PDG.

included.

Consider Figure 3, which represents the PDG of the code in Figure 2, where the slicing criterion is the node marked with a bolded border. In the example, a whole data structure (the tuple `{[8],A}`) has been collapsed to a variable (`Z`) and then expanded again (`{[C],D}`). Therefore, the list `[C]` depends on the list `[8]`. Nevertheless, the traditional PDG represents the equality as a whole, making `[C]` flow dependent on `Z`, and in turn, `Z` flow depends on `A`. Because flow dependence is transitive, slicing the PDG wrongly infers that `C` depends on `A` (`A` is wrongly included in the slice for `C`), and this lack of precision is propagated to parameter `X`.

This problem worsens in the presence of recursive data types. For instance, trees or objects (consider a class A with a field of type A, which produces a potentially infinite data type) can prevent the slicer from knowing statically what part of the collapsed structure is needed. Späth et al. [25, pp. 2–3] present an interesting discussion and example about this problem.

3. Background: The Program Dependence Graph

This section briefly summarizes the theoretical background of the PDG that is needed to keep the paper self-contained. Extended explanations about the applications of the PDG can be found in program slicing surveys [4, 24, 3]. Readers already familiar with the PDG can skip this section.

Given a program, two kinds of dependencies can be defined over it to construct the PDG [2]: control dependence and flow dependence (aka data dependence).

Definition 1 (Control Dependence). Let G be a CFG (Control-Flow Graph). A node n_1 post-dominates a node n_2 in G if all paths in the CFG from n_2 to the *Exit* node of the CFG traverse n_1. Node n_1 is *control dependent* on node n_2 if and only if n_1 post-dominates one but not all of n_2's successors.

Definition 2 (Flow Dependence). A node n_2 is *flow dependent* on a preceding node n_1 if (1) n_1 defines a variable x, (2) n_2 uses x, and (3) there exists a control-flow path from n_1 to n_2 where x is not defined.

Definition 3 (Program Dependence Graph). Given a procedure p, its *Program Dependence Graph* (PDG) is a graph $G = (N, E)$, where there is a node in N to represent each statement in p; and E is a set of edges that represent all control and flow dependencies between the nodes in N.

4. Constrained-Edges Program Dependence Graph

This section introduces the CE-PDG, in which the key idea is to expand all PDG nodes where a composite data structure is defined or used. This expansion augments the PDG with a tree representation for composite data structures. We describe how this structure is generated and we introduce a new kind of dependence edge used to build this tree structure. For this, we formally define the concepts of *constraint* and *constrained edge*, describe the different types of constraint, and how they affect the graph traversal in the slicing process.

4.1. Extending the PDG

Figure 2b shows that PDGs are not accurate enough to differentiate the elements of composite structures. For instance, the whole statement in line 4 is represented by a single node, so it is not possible to distinguish the data structure `{A,B}` nor its internal subexpressions. This can be solved by transforming the PDG into a CE-PDG. The transformation consists of three steps.

Step 1. The first step is to decompose all nodes that contain composite data structures so that each component is represented by an independent node. As in most ASTs, we represent data structures with a tree-like representation (similar to the one used in object-oriented programs to represent objects in calls [26, 27]). The decomposition of PDG nodes into CE-PDG nodes is straightforward from the AST. It is a recursive process that unfolds the composite structure by levels, i.e., if a subelement is another composite structure, it is recursively unfolded until the whole syntax structure is represented in the tree. The CE-PDG only unfolds data types to the level they are represented in the source code, thus unfolding is always finite (unlike atomization). In contrast to the PDG nodes (which represent complete statements), the nodes of this tree structure represent expressions. Therefore, we need a new kind of edge to connect these intra-statement nodes. We call these edges *structural edges* because they represent the syntactical structure of the program.

Definition 4 (Structural Edge). Let $G = (N, E)$ be a CE-PDG where N is the set of nodes and E is the set of edges. Given two CE-PDG nodes $n, n' \in N$, there exists a *structural edge* $n \dashrightarrow n'$ if and only if:

- n contains a data structure for which n' is a subcomponent, and
- $\forall n'' \in N : n \dashrightarrow n' \wedge n' \dashrightarrow n'' \rightarrow n \not\dashrightarrow n''$.

Structural edges point to the components of a composite data structure, composing the inner skeleton of its abstract syntax tree. More precisely, each field in a data type is represented with a separate node that is connected to the PDG node that contains the composite data structure. For instance, the structural edges of the CE-PDG in

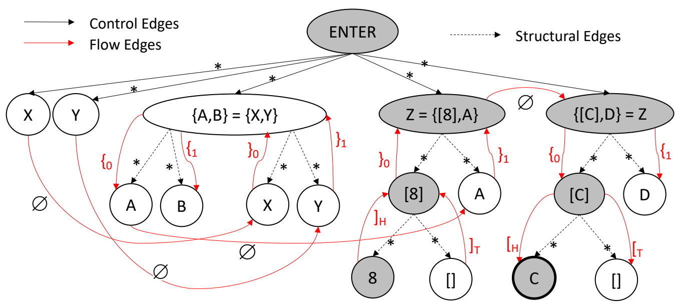


Figure 4: CE-PDG of the code in Figure 2.

Figure 4 represent the tuples of the code in Figure 2. The second condition of the definition enforces the tree structure as otherwise "transitive" edges could be established. For example, without the second condition a structural edge between `{[C],D} = Z` and `C` could exist.

Step 2. The second step is to identify the flow dependencies that arise from the decomposition of the data structure. Clearly, the new nodes can be variables that flow-depend on other nodes, so we need to identify the flow dependencies that exist among the new (intra-statement) nodes. They can be classified according to two different scenarios: composite data structures being (i) defined and (ii) used. In Figure 2 we have a definition (line 4), a use (line 3) and a definition and use in the same node (line 2). The explicit definition of a whole composite data structure (e.g., a tuple in the left-hand side of an assignment, see line 4) always defines every element inside it, so the values of all subelements depend on the structure that immediately contains them. Hence, the subexpressions depend on the structure being defined (i.e., flow edges follow the same direction as structural edges. See `{[C],D}=Z` in Figure 4). Conversely, the structure being used depends on its subexpressions (i.e., flow edges follow the opposite direction than structural edges. See `Z={[8],A}` in Figure 4). Additionally, because the decomposition of nodes augments the precision of the graph, all flow edges that pointed to original PDG nodes that have been decomposed, now point to the corresponding node in the new tree structure. An example of a flow edge that has been moved due to the decomposition is the flow edge between the new `A` nodes. In the original PDG, this flow edge linked the nodes `{A,B}={X,Y}` and `Z={[8],A}`.

Step 3. The last step to obtain the CE-PDG is labelling the edges with constraints that are later used during the slicing phase. The idea is that the slicing algorithm traverses the edges and collects the labels in a stack that is used to decide what edges should be traversed and what edges should be ignored. We call the new labelled edges *constrained edges* because the labels act as constraints for the graph traversal.

For the sake of simplicity, and without loss of generality, we distinguish between tuples and functional (algebraic) lists. The position in a tuple is indicated with an integer, while the position in a list is indicated with head (H) or tail (T). The case of objects, records, or any other structure can be trivially included by just specifying the position with the name of the field.

Definition 5 (Constraint). A constraint C is a label defined with the following grammar:

$$\begin{aligned} C &::= \varnothing \mid * \mid Access \\ Access &::= Tuple \mid List \\ Tuple &::= \{_{int} \mid \}_{int} \\ List &::= [_{Pos} \mid]_{Pos} \\ Pos &::= H \mid T \end{aligned}$$

where int is a positive integer.

The meaning of each kind of constraint is the following:

Empty Constraint ($n \xrightarrow{\varnothing} n'$). It specifies that an edge can always be traversed by the slicing algorithm.

Asterisk Constraint ($n \xrightarrow{*} n'$). It also indicates that an edge can always be traversed; but it ignores all the collected restrictions so far, which means that going forward, the whole data structure is needed. This kind of constraint is the one used in control and structural edges, which are traversed ignoring the previous constraints collected.

Access Constraint ($n \xrightarrow{op_{position}} n'$). It indicates that an element is the *position*-th component of another data structure that is a tuple if op=$Tuple$ or a list if op=$List$. op also indicates whether the element is being defined ("{", "[") or used ("}", "]").

Example 3 (Labeling edges). All edges in Figure 4 are labelled with constraints. Because `B` is the second element being defined in the tuple `{A,B}`, the constraint of the flow dependence edge that connects them is $\{_1$. Also, because 8 is the head in the list `[8]`, the constraint of the flow dependence edge that connects them is $]_H$.

At this point, the reader can see that the constraints can be used to accurately slice the program in Figure 2a. In the CE-PDG (Figure 4), the slicing criterion (`C`) is the head of a list (indicated by the constraint $[_H$), and this list is the first element of a tuple. When traversing backwards the flow dependencies, we do not want the whole `Z`, but only the head of its first element (i.e., the cumulated constraints $[_H\{_0$). Then, when we reach the definition of `Z`, we find two flow dependencies (`[8]` and `A`). But looking at their constraints, we exactly know that we want to traverse first $\}_0$ and then $]_H$ to reach the `8`. So far, no structural edge is traversed during the slice in the above example. Structural edges represent the syntactical structure of composite data structures in trees. Therefore, after a structural edge has been traversed, no flow edges are allowed to be traversed (note that every structural edge has

$$
\begin{aligned}
P &::= C\,O\\
C &::= \}_i C \mid]_p C \mid R\,C \mid \varnothing C \mid *P \mid \epsilon\\
R &::= \{_i R\,\}_i \mid [_p R\,]_p \mid \varnothing R \mid \epsilon\\
O &::= \{_i O \mid [_p O \mid R\,O \mid \varnothing O \mid *P \mid \epsilon
\end{aligned}
$$

(a) Realizable paths grammar

$$
\begin{aligned}
W &::= O'\\
O' &::= \{_i O' \mid [_p O' \mid \epsilon
\end{aligned}
$$

(b) Stack words

Figure 5: Grammars defining allowed constraints ($p \in \{H, T\}$ and $i \in \mathbb{Z}$).

a corresponding flow edge in the same or opposite direction). The slice computed in this way is composed of the grey nodes, and it is exactly the minimal slice in Figure 2c.

The CE-PDG is a generalization of the PDG because the PDG is a CE-PDG where all edges are labelled with empty constraints (∅). In contrast, all edges in the CE-PDG are labelled with different constraints:

- Structural and control edges are always labelled with asterisk constraints.
- Flow edges for definitions inside a data structure are labelled with opening ({,[) access constraints.
- Flow edges for uses inside a data structure are labelled with closing (},]) access constraints.
- The remaining data edges are labelled with empty constraints.

The behaviour of access constraints and asterisk constraints in the graph traversal is further detailed in the next section, where we also formalize the slicing algorithm that performs the traversal of the CE-PDG.

4.2. Constrained traversal

In this section, we show how constraints can improve the accuracy of the slices computed with the CE-PDG. In order to represent the paths of the CE-PDG that can be traversed, we use a grammar. The label of an edge can be seen as a terminal. Therefore, by traversing the edges we build words. But not all edges can be traversed; paths are only realizable when the word induced by the path belongs to a language for which the grammar is shown in Figure 5a. The realizable path grammar is modelled after the grammars by Reps et al. [28]. In this grammar, P is the initial symbol, C, R, and O represent sequences that contain closing, resolved, and opening constraints, respectively. $\varnothing$ and $*$ stand for empty and asterisk constraints, respectively.

The key point of this grammar is *resolved* constraints. A resolved constraint is an opening constraint followed by *the complementary* closing constraint (e.g., $\{_2$ followed by $\}_2$). The paths of the CE-PDG that can be traversed are formed by any combination of closing constraints followed by opening constraints. Any number of empty constraints ($\varnothing$) can be placed along the path. On the other hand, asterisk constraints ($*$) always ignore any constraints already collected. Therefore, after traversing an asterisk constraint, the paths that can be traversed are the same as if no constraint was previously collected. The reason behind this behaviour is later detailed in Example 6.

Example 4 (Traversing constraints). Consider Figure 4 again and the slicing criterion (`C`). To compute the slice we traverse edges backwards, tracing a path from `C` to `8` formed by the following sequence of constraints: $[_H\{_0\varnothing\}_0]_H$, which can be derived from the grammar in Figure 5a:

$$
\begin{aligned}
P &\xrightarrow{P\to CO} CO \xrightarrow{C\to\epsilon} O \xrightarrow{O\to RO} RO \xrightarrow{O\to\epsilon} R\\
&\xrightarrow{R\to[_H R]_H} [_H R]_H \xrightarrow{R\to\{_0 R\}_0} [_H\{_0 R\}_0]_H\\
&\xrightarrow{R\to\varnothing R} [_H\{_0\varnothing R\}_0]_H \xrightarrow{R\to\epsilon} [_H\{_0\varnothing\}_0]_H
\end{aligned}
$$

Opening access constraints can be seen as queries that can be solved along the path traversal. Let us explain this view with an example. Consider Figure 2 again and assume that we are interested in the value of variable `A` in line 2. In this case, `A` is being defined. Since `A` is inside a particular position of the composite structure (position 0 inside the tuple `{A,B}`), it can only receive a value from the same position in an analogue composite structure. This may happen in the same statement or in any other previous one. For this reason, we metaphorically "open" a query to find the value of variable `A`, and indicate it with the corresponding structure and position symbols ($\{_0$). When we reach the statement level, we notice that the expression giving value to the whole data structure is an analogue data structure (the right-hand side of the equality, tuple `{X,Y}`). Thus, the expression that gives value to variable `A` must be inside this data structure. Because we are looking for a specific position with an open query ($\{_0$), we can choose the element we are interested in between the possible ones (`X`, reachable through $\}_0$), "closing" this query. As a result, we reach a state where the pending query has been "resolved", and we can now focus on another open query, if any. To sum up, with this point of view, each variable definition contained in a composite structure is considered as the opening of a flow dependence query, and each variable use inside an analogous structure as the closing of this flow dependence query. Then, every time a query is successfully closed we say that the opened query has been resolved.

To ensure that only realizable paths are visited, the slicing algorithm uses a stack to store the word generated by traversing the CE-PDG. When a node is selected as the slicing criterion, the algorithm starts from this node with an empty stack ($\perp$) and accumulates constraints with each edge traversed. Only opening constraints impose a restriction on the symbols that can be pushed onto the stack: when an opening constraint is on the top of the stack, the only closing constraint accepted to build a realizable word

Table 1: Processing edges with a stack. x and y are positions (*int* or H/T). $\varnothing$ and $*$ are empty and asterisk constraints, respectively. S is a stack, $\perp$ the empty stack.

	Input Stack	Edge Constraint	Output Stack
(1)	S	$\varnothing$	S
(2)	S	$\{_x$ or $[_x$	$S\{_x$ or $S[_x$
(3)	$\perp$	$\}_x$ or $]_x$	$\perp$
(4)	$S\{_x$ or $S[_x$	$\}_x$ or $]_x$	S
(5)	$S\{_x$ or $S[_x$	$\}_y$ or $]_y$	*error*
(6)	S	$*$	$\perp$

is its complementary closing constraint. Therefore, the only information necessary to determine whether an edge can be traversed is the sequence of non-resolved (opening) constraints at the top of the stack. They form the words that remain in the stack when a path is traversed (see the grammar in Figure 5b).

Example 5 (Evolution of the stack). To produce the derivation shown in Example 4 we start from the slicing criterion with an empty stack that is filled and emptied during the traversal:

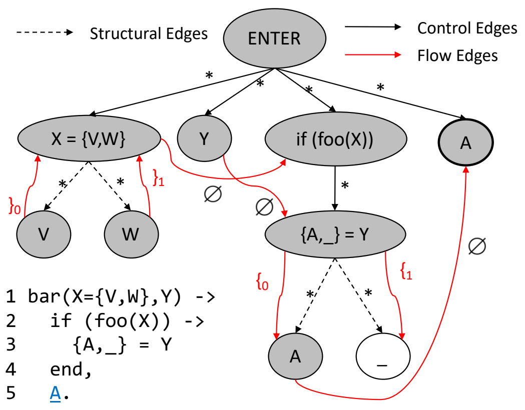
$$\perp \xrightarrow{[_H} [_H \xrightarrow{\{_0} [_H\{_0 \xrightarrow{\varnothing} [_H\{_0 \xrightarrow{\}_0} [_H \xrightarrow{]_H} \perp$$

All words in the stack are only formed from opening access constraints, as defined by the grammar in Figure 5b.

Table 1 shows how the stack is updated in all possible situations. The constraints are collected or resolved depending on the last constraint added to the word (the one at the top of the *Input stack*) and the new one to be treated (column *Edge Constraint*). All cases shown in Table 1 can be summarized in four different situations:

- **Traverse constraint (cases 1 and 3):** The edge is traversed without modifying the stack.
- **Collect constraint (case 2):** The edge can be traversed by pushing the edge's constraint onto the stack.
- **Resolve constraint (cases 4 and 5):** There is an opening constraint at the top of the stack and an edge with a closing constraint that matches it (case 4), so the edge is traversed by popping the top of the stack; or they do not match (case 5), so the edge is not traversed.
- **Ignore constraints (case 6):** Traversing the edge empties the stack.

Previous examples have shown the behaviour of access and empty constraints. Example 6 shows a program in which asterisk constraints are necessary to select all the nodes that are needed.



```
1 bar(X={V,W},Y) ->
2   if (foo(X)) ->
3     {A,_} = Y
4   end,
5   A.
```

Step	Start Node	Edge		Node Reached	Stack
0	-	-	-	(5,`A`)	$\perp$
1	(5,`A`)	flow	$\varnothing$	(3,`A`)	$\perp$
2	(3,`A`)	flow	$\{_0$	(3,`{A,_} = Y`)	$\{_0$
3a	(3,`{A,_} = Y`)	flow	$\varnothing$	(1,`Y`)	$\{_0$
4a	(1,`Y`)	control	$*$	(1,ENTER)	$\perp$
3b	(3,`{A,_} = Y`)	control	$*$	(2, `if(foo(X))`)	$\perp$
4b	(2, `if(foo(X))`)	flow	$\varnothing$	(1,`X={V,W}`)	$\perp$
5b	(1,`X={V,W}`)	flow	$\}_0$	(1,`V`)	$\perp$
5c	(1,`X={V,W}`)	flow	$\}_1$	(1,`W`)	$\perp$

Figure 6: Erlang function, associated CE-PDG, and slice step by step.

Example 6 (The need for asterisk constraints). Consider the Erlang function and its associated CE-PDG in Figure 6, where variable `A` in line 5 is the slicing criterion. Consider also the table in Figure 6, where the most relevant steps of the backward traversal of the graph are shown. Each row represents the traversal of one edge, except the initial row, which represents the selection of the first node (the slicing criterion). In the table, column **Step** represents the number of the current step during the traversal. Different alternative paths are shown with letters (a,b,c). Columns **Start Node** and **Node Reached** represent the start and end nodes (`line`,`expression`) when traversing the current edge, **Edge** represents the type and constraint of the traversed edge; and **Stack** represents the stack computed after traversing the edge.

After reaching `{A,_} = Y` (step 2), the stack contains the opening constraint $\{_0$, and there are two possible paths: (a) a flow path to the parameter variable `Y` (step 3a), and (b) a control path to the `if` condition (step 3b). Let us focus on the second path to show the necessity of asterisk constraints. When we traverse the control edge, all the constraints stacked due to the traversal of previous flow edges must be dropped from the stack (case 6 in Table 1). The reason is simple: when we reach a statement by a control edge, we are no longer interested in the value of the uses of variables that the traversal has accumulated in the stack, but in the value of the variables used in this controller statement. The fact is that keeping the previous stack constraints may result in erroneous slices. For instance, consider a scenario where, if we do not empty the stack in step 3b, we would reach the `X={V,W}` statement with the stack $\{_0$, and the traversal would only reach the first element of the tuple (V) traversing $\}_0$. Therefore, W would be never included in the slice because it can only be reached traversing the constraint $\}_1$ that does not match the constraint of the stack. In contrast, emptying the stack in step 3b when traversing the control edge forces the slice to correctly include both V and W. Note also that the constraint $\{_0$ collected in step 2 is not entirely useless. It is still used in the flow path to Y (we only want the first component of Y).

4.3. Subsumed constraints

The inclusion of constraints in the PDG's edges induces interesting properties that we can take advantage of. An important property is that the constraints of some stacks subsume the constraints of other stacks. The subsumed constraints can be ignored, thus, improving the efficiency and the precision in the slicing traversal.

When we reach a node of a data structure with a non-empty stack, we are explicitly requiring a part of the data structure represented by the stack. For instance, if we reach a list with the constraint $[_T$ we require the *whole* tail of the list, but not the head. Therefore, if we later reach the same data structure with a stack $[_H[_T$, which requires only the second element of the list (the head of the tail), we can ignore the second stack because it is subsumed by the previous one.

In general, the number of cumulated constraints in a stack when we reach a data structure indicates the depth where the element that we are searching for is located.

Example 7. Consider the tuple $\{\{\{1,2\},\{3,4\}\},\{\{5,6\},\{7,8\}\}\}$, that we reach with the following four stacks:

$\bot$: If the stack is empty, we need all the elements of the data structure.

$\{_0$: We need the whole tuple in position 0: $\{\{1,2\},\{3,4\}\}$.

$\{_1$: We need the whole tuple in position 1: $\{\{5,6\},\{7,8\}\}$.

$\{_1,\{_0$: We need a part of element 0, but not all. We only need the element in position 1: $\{3,4\}$.

The following inter-relations exist: $\{_0 \subseteq \bot$, $\{_1 \subseteq \bot$, $\{_1,\{_0 \subseteq \bot$, and $\{_1,\{_0 \subseteq \{_0$.

The above example reveals an important property: if we reach a node with two different stacks and *the first is a suffix of the second*, then the second stack can be ignored. This relationship imposes an order among stacks.

Definition 6 (Stack ordering). Given two stacks S and S', $S' \subseteq S$ if and only if S is a suffix of S'.

This ordering is a partial order because it is reflexive ($S \subseteq S$), antisymmetric ($S_1 \subseteq S_2 \wedge S_2 \subseteq S_1 \iff S_1 = S_2$) and transitive ($S_1 \subseteq S_2 \wedge S_2 \subseteq S_3 \implies S_1 \subseteq S_3$).

We can use this property to sometimes stop the traversal of the CE-PDG: The traversal is stopped when a node is reached with a stack, and this node has been already reached by another previous stack that is a suffix of the new one. Note that this includes the case where a stack is empty because the empty stack is always a suffix of any other stack. Hence, when a node is traversed with the empty stack, it must not be traversed again.

Algorithm 1 Intraprocedural slicing algorithm for CE-PDGs

Input: The slicing criterion node n_{sc}.
Output: The set of nodes that compose the slice.

```
function SLICINGALGORITHMINTRA($n_{sc}$)
  $slice \leftarrow \emptyset$; $processed \leftarrow \emptyset$
  $workList \leftarrow \{\langle n_{sc}, \bot, [] \rangle\}$
  while $workList \neq \emptyset$ do
    select some $state \in workList$;
    $newItems = \{\}$
    $\langle node, stack, traversedEdges \rangle \leftarrow state$
    for all $edge \in$ GETINCOMINGEDGES($node$) do
      $\langle node', type, \_ \rangle \leftarrow edge$
      if GETLASTEDGETYPE($traversedEdges$) $= structural$ then
        if $type = flow$ then
          continue for all
      $newStack \leftarrow$ PROCESSCONSTRAINT($stack, edge, traversedEdges$)
      if $newStack \neq error$ then
        if $\nexists (node', s, t) \in processed \cup workList$
            $|$ $s$ is suffix of $newStack$ then
          $w \leftarrow \langle node', newStack, traversedEdges \cup \{edge\} \rangle$
          $works \leftarrow works \cup \{w\}$
    $processed \leftarrow processed \cup \{state\}$
    for all $(n, s, t) \in works$ do
      $workList \leftarrow workList \setminus \{(n', s', t') \in workList \mid$
                              $n = n' \wedge s$ is suffix of $s'\}$
    $workList \leftarrow workList \cup works$
    $slice \leftarrow slice \cup \{node\}$
  return $slice$

function PROCESSCONSTRAINT($stack$, $edge$, $traversedEdges$)
  $\langle \_, \_, constraint \rangle \leftarrow edge$
  $\langle op, position \rangle \leftarrow constraint$
  if $constraint = AsteriskConstraint$ then                    ▷ Case 6
    return $\bot$
  if $edge \in traversedEdges$ then                          ▷ Check cycles
    $loop \leftarrow$ FINDLOOP($traversedEdges$)               ▷ Check loops
    if $loop \neq error \wedge$ ISINCREASINGLOOP($loop, edge$) then
      return $\bot$                    ▷ Inc. loop replaced by * constraint
  if $constraint = EmptyConstraint$ then                      ▷ Case 1
    return $stack$
  else if $op = \{ \vee op = [$ then                               ▷ Case 2
    PUSH($constraint, stack$)
  else if $stack = \bot$ then                                   ▷ Case 3
    return $\bot$
  else if $op = \} \wedge$ TOP($stack$) $= \langle \{, position \rangle$ then          ▷ Case 4
    POP($stack$)
  else if $op = ] \wedge$ TOP($stack$) $= \langle [, position \rangle$ then          ▷ Case 4
    POP($stack$)
  else          ▷ Non-matching closing constraint                ▷ Case 5
    return $error$
  return $stack$
```

4.4. The slicing algorithm

Algorithm 1 illustrates the process to slice the CE-PDG. It works similarly to the standard algorithm [29], traversing backwards all edges from the slicing criterion and collecting nodes to form the final slice. The algorithm uses a work list with the states that must be processed. A state represents the (backward) traversal of an edge. It includes the node reached, the current stack, and the sequence of already traversed edges (line 7). In every iteration, the algorithm processes one state. First, it collects all edges that target the current node (function getIncomingEdges in line 8). If the previously traversed edge is structural, we avoid traversing flow edges (lines 10–12) and only traverse structural or control dependence edges. The reason for this is that structural edges are only traversed to collect the structure of a data type so that the final slice is syntactically correct (for instance, to collect the tuple to which an element belongs). Flow edges are not further traversed to avoid collecting irrelevant dependencies of the structural parent. The function processConstraint checks the existence of a loop (reaching an already traversed edge) during the slicing traversal and implements Table 1 to produce the new stack generated by traversing the edge to the next node (line 13). If the edge cannot be traversed according to Table 1 (line 14), or because the new stack is subsumed (see Section 4.3) by one in the previously processed states or the work list (line 16), then the reachable node is ignored. Otherwise, the node is stored together with the new stack (line 18). Finally, the state is added to a list of processed states, used to avoid multiple evaluations of the same state (line 19), the work list is updated (removing any element that would be subsumed by new items – lines 20–22, and then adding said items – line 23), and finally the current node is included in the slice (line 24).

The function processConstraint computes a new stack for all possible types of constraint, taking into account Table 1 and Section 4.5 (loop detection and avoidance). It checks first for an asterisk constraint and returns an empty stack (line 29). Then, the condition in line 31 checks the existence of a cycle (reaching an already traversed edge) during the slicing traversal. The function findLoop (line 32) returns the shortest suffix of the sequence of traversed edges that form the last loop, while function isIncreasingLoop (line 33), whose rationale is extensively explained in Section 4.5, consequently empties the stack when needed. If no dangerous loop is detected, the function checks for each of the cases in Table 1, pushing or popping elements from the stack accordingly (lines 35–47).

Example 8 (Applying Algorithm 1). Consider function `foo` in the code of Figure 2a again, the selected slicing criterion ($\langle 4, \texttt{C}\rangle$), and its CE-PDG, shown in Figure 4. The slicing process starts from the node that represents the slicing criterion (the expanded representation of the CE-PDG allows us to select `C`, the bold node, inside the tuple structure, excluding the rest of the tuple elements). Algorithm 1 starts the traversal of the graph with an empty stack ($\perp$). The evolution of the stack after traversing each flow edge is the following:

$$\perp \xrightarrow{[_H} [_H \xrightarrow{\{_0} [_H\{_0 \xrightarrow{\varnothing} [_H\{_0 \xrightarrow{\}_0} [_H \xrightarrow{]_H} \perp$$

Due to the traversal limitations imposed by row 5 in Table 1, node `A` is never included in the slice because the following transition is not possible: $[_H\{_0 \xrightarrow{\}_1} \mathit{error}$.

The slicing algorithm will also traverse the structural edges reaching the traversed nodes and generate new states in the worklist with empty stacks due to the asterisk constraint, however, no flow dependence edge is traversed after a structural edge (lines 10–12) and therefore despite the node `Z={[8],A}` being encountered with an empty stack, the flow edge to `A` is not traversed.

As already noted, the resulting slice provided by Algorithm 1 is exactly the minimal slice shown in Figure 2c.

4.5. Dealing with loops

In static slicing we rarely know the values of variables (they often depend on dynamic information), so we cannot know how many iterations will be performed in a program loop[1] (see the programs in Figure 7, where the value of `max` is unknown). For the sake of completeness, we must consider any number of iterations, thus program loops are often seen as potentially infinite. Program loops produce cycles in the PDG. Fortunately, the traversal of cycles in the PDG is not a problem, since every node is only visited once. In contrast, the traversal of a cycle in the CE-PDG could produce a situation in which the stack grows infinitely (see Figure 7d[2] and its CE-PDG in Figure 9), generating an infinite number of states. Fortunately, not all cycles produce this problem:[3] To keep the discussion precise, we need to formally define when a cycle in the CE-PDG is a *loop*.

Definition 7 (Loop). A cyclic flow dependence path $P = n_1 \xleftarrow{C_1} n_2 \ldots \xleftarrow{C_n} n_1$ is a *loop* if P can be traversed more than once with an initially empty stack ($\perp$) following the rules of Table 1.

Example 9 (Cycles vs. loops). Cycles that are not loops are not dangerous because the cycle's edges constraints prevent us to traverse them infinitely. The code in Figure 7a contains a cycle that is not a loop because it can be traversed only once with an empty stack. This can be observed in Figure 8, where we have the flow dependence cycle: $(6,x) \xleftarrow{\}_0} (6,a) \xleftarrow{\varnothing} (5,a) \xleftarrow{\{_0} (5,d) \xleftarrow{\varnothing} (4,d) \xleftarrow{\{_1}$

[1]Note the careful wording in this section, where we distinguish between "program loops" (`while`, `for`...), "cycles" (paths in the PDG that repeat a node), and "loops" (repeated sequence of nodes during the graph traversal).

[2]It is easier to see how the stack changes by reading the code backwards from the slicing criterion.

[3]The interested reader has a developed example for each kind of loop, which includes their CE-PDGs, in the technical report at `https://mist.dsic.upv.es/techreports/2023/02/case-studies-field-sensitive.pdf`.

```
1 read(max);
2 x = init_tuple();
3 for(int i=0; i<max; i++){
4   {e,d} = x;
5   {a,b} = d;
6   x = {a,5};
7 }
8 {c,d} = x;
9 print(c);
```

(a) A cycle but not a loop

```
1 read(max);
2 read(b);
3 x = init_tuple();
4 for(int i=0; i<max; i++){
5   a = {x,i};
6   x = {a,b};
7 }
8 {c,d} = x;
9 print(c);
```

(b) Loop: decreasing stack size

```
1 read(max);
2 x = init_tuple();
3 for(int i=0; i<max; i++){
4   {a,b} = x;
5   x = {a+i,b+i};
6 }
7 {c,d} = x;
8 print(c);
```

(c) Loop: same stack size

```
1 read(max);
2 x = init_tuple();
3 for(int i=0; i<max; i++){
4   {e,d} = x;
5   {a,b} = d;
6   x = {a,b};
7 }
8 {c,d} = x;
9 print(c);
```

(d) Loop: increasing stack size

Figure 7: Slicing flow-dependence cycles in the CE-PDG (slicing criterion underlined and blue, slice in green).

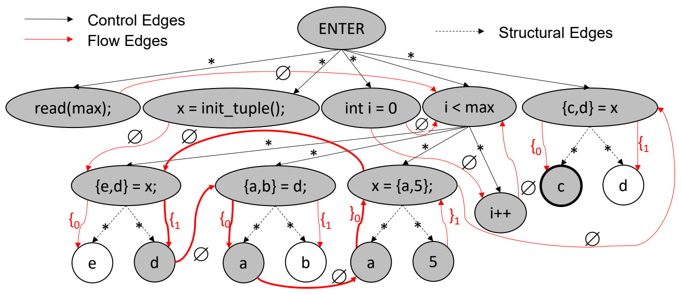


Figure 8: CE-PDG of Figure 7a. The cycle is represented with bold red edges.

$(4, x) \xleftarrow{\emptyset} (6, x)$. But this is not a loop because no matter the stack we enter the cycle with, when $\{_1$ is pushed on the stack, the cycle cannot be entered again, as the $\}_0$ constraint cannot be matched against the top of the stack.

There exist three kinds of loops:

1. Loops that decrease the size of the stack in each iteration (Figure 7b) can only produce a finite number of states because the stack will eventually become empty. Such loops can be traversed collecting the elements specified by the stack, without a loss of precision.

2. Loops that maintain the stack's size constant in each iteration (Figure 7c) are also not a problem because traversing the loop multiple times does not generate new states. Again, they can be traversed as many times as required by the stack, without a loss of precision.

3. Loops that increase the size of the stack in each iteration (Figure 7d) produce an infinite number of states because the stack grows infinitely. This can be observed in the CE-PDG (Figure 9) of the code in Figure 7d, where we can see the increasing loop: $(6, x) \xleftarrow{\}_1} (6, b) \xleftarrow{\emptyset} (5, b) \xleftarrow{\{_1} (5, d) \xleftarrow{\emptyset} (4, d) \xleftarrow{\{_1} (4, x) \xleftarrow{\emptyset} (6, x)$

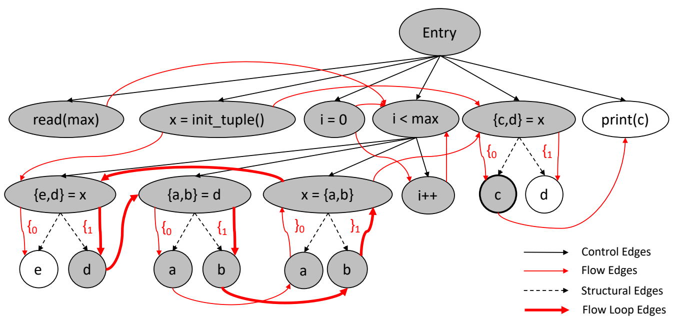


Figure 9: CE-PDG of Figure 7d. The increasing loop is represented with bold red edges.

We formally define a special kind of loop which is the only one with potential danger: the *increasing loop*.

Definition 8 (Increasing loop). A loop L is an *increasing loop* if the number of opening constraints along L is greater than the number of closing constraints.

Identifying increasing loops. To detect the increasing loops (those that can grow the stack infinitely) we have designed the pushdown automaton (PDA) shown in Figure 10. The input of this automaton is the sequence of constraints that form a dependence cycle. The PDA tracks two stacks (for closing and opening constraints) across two states. Initial state 0 represents the case where all opening constraints of the sequence are balanced by the corresponding closing constraint. When a closing constraint is reached, the PDA pushes the constraint into the closing stack ($push_c$). When an opening constraint is processed, the PDA pushes the opening constraint into the opening stack ($push_o$) and moves to state 1. Final state 1 represents the case where an opening constraint has been processed but not balanced yet. In state 1, when a closing constraint that matches a previous opening constraint (condition M_o) is processed, we pop the opening constraint from the stack (pop_o). If the popped element of the opening stack is the last element of the stack (condition E_o), the PDA returns to state 0. Finally, if a path is accepted by this automaton, the path forms an increasing loop if and only if:

the reversed stack S_c is a prefix of S_o and they are not equal.

The rationale of this condition is that it ensures that, in each iteration, there are more opening constraints (those in S_o) than closing constraints (those in S_c), and all the closing constraints close some but not all opening constraints (because they are a prefix), thus the number of opening

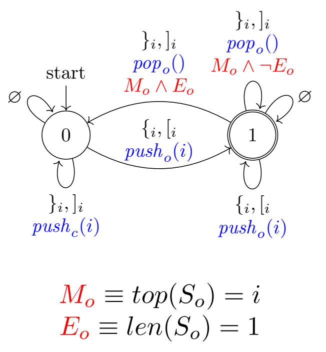

start
$\varnothing$
$\varnothing$
$\}_i,]_i$ $pop_o()$ $M_o \wedge E_o$
$\}_i,]_i$ $pop_o()$ $M_o \wedge \neg E_o$
$\{_i, [_i$ $push_o(i)$
$\}_i,]_i$ $push_c(i)$
$\{_i, [_i$ $push_o(i)$
0
1
$M_o \equiv top(S_o) = i$
$E_o \equiv len(S_o) = 1$

Figure 10: Pushdown automaton to recognize increasing loops.

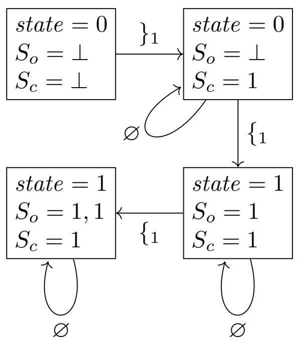

$state = 0$ $S_o = \bot$ $S_c = \bot$
$\}_1$
$state = 0$ $S_o = \bot$ $S_c = 1$
$\varnothing$
$\{_1$
$state = 1$ $S_o = 1, 1$ $S_c = 1$
$\{_1$
$state = 1$ $S_o = 1$ $S_c = 1$
$\varnothing$
$\varnothing$

Figure 11: States produced by the PDA in Figure 10 with the word $\}_1\varnothing\{_1\varnothing\{_1\varnothing$

constraints grows infinitely. Note that asterisk constraints (*) do not appear in the PDA because they cannot appear in an increasing loop (an asterisk constraint empties the stack and thus the same state would be repeated).

Example 10 (Detecting an increasing loop). Consider the dependence cycle formed from lines 4, 5, and 6 of Figure 7d (see its CE-PDG in Figure 9): $(6, x) \xleftarrow{\}_1} (6, b) \xleftarrow{\varnothing} (5, b) \xleftarrow{\{_1} (5, d) \xleftarrow{\varnothing} (4, d) \xleftarrow{\{_1} (4, x) \xleftarrow{\varnothing} (6, x)$, which contains the word: $\}_1\varnothing\{_1\varnothing\{_1\varnothing$.

Now, if we parse this word with the PDA we produce the sequence of states shown in Figure 11. The final state is an accepting state, and the reverse of S_c (1) is a prefix of S_o (1, 1) (but they are not equal), so this path corresponds to an increasing loop. Moreover, the PDA also detects that this loop adds $\{_1$ (the remainder of S_o once the prefix is removed) to the stack in every iteration.

More formally, an increasing loop $n_1 \xleftarrow{C_1} n_2 \xleftarrow{C_2} \ldots \xleftarrow{C_n} n_1$ can be identified because $C_1C_2 \ldots C_n$ belongs to the language induced by the PDA and the two final stacks computed with the PDA (S_c and S_o) satisfy that $reverse(S_c)$ is a prefix of S_o and $reverse(S_c) \neq S_o$.

Detecting loops. When a loop is detected, the PDA can determine whether it is increasing. Fortunately, it is not necessary to preprocess the CE-PDG to detect the loops. The loops can be detected at slicing time during the traversal (with Algorithm 1). The strategy proposed in [20] to detect a loop and prevent traversing it infinitely was:

1. To traverse the CE-PDG backwards following all the edges until we traverse the same edge twice.

2. If the same node is reached with the same stack twice, then the traversal of this node is stopped.

3. Otherwise, use the PDA to determine whether the loop is increasing.

4. If the loop is increasing, the constraint of the traversed edge is considered as an asterisk constraint (which empties the stack). This ensures that a loop is only traversed at most twice (the second time that the stack is emptied, the state is repeated) and slicing is ensured to terminate.

The new algorithm (Algorithm 1) modifies point 2: the traversal of a node is stopped if this node has been previously reached *with a stack that is a suffix of the current stack* (see Section 4.3). This change significantly reduces the number of times that the PDA is triggered as can be seen in Example 11.

Example 11 (Increased efficiency of Algorithm 1). Consider the CE-PDG in Figure 13 and let c be the slicing criterion. A traversal with the approach of [20] is shown in the following table:

Step	Start Node	Edge		Node Reached	Stack
0	-	-	-	c	$\bot$
1	c	flow	$\{_0$	{c,d}=x	$\{_0$
2	{c,d}=x	flow	$\varnothing$	x	$\{_0$
3	x	flow	$\{_0$	{x,a}=b	$\{_0\{_0$
4	{x,a}=b	flow	$\varnothing$	b=x	$\{_0\{_0$
5	b=x	flow	$\varnothing$	x	$\{_0\{_0$
6	x	flow	$\{_0 \rightsquigarrow *$	{x,a}=b	$\bot$
7	{x,a}=b	flow	$\varnothing \rightsquigarrow *$	b=x	$\bot$
8	b=x	flow	$\varnothing \rightsquigarrow *$	x	$\bot$
9	x	flow	$\{_0 \rightsquigarrow *$	{x,a}=b	$\bot$

In step 6, the edge $x \leftarrow \{x, a\} = b$ is visited for the second time and, thus, the PDA is triggered to determine whether the loop is increasing, which it is. Therefore, the edge's constraint is ignored, and an asterisk constraint is applied instead (emptying the stack). The same happens in steps 7, 8, and 9. In step 9, for the first time, a node $\{x, a\} = b$ is visited twice with the same stack and, thus, the traversal terminates.

In contrast, if we use Algorithm 1, steps 6–9 are never executed; and the PDA is never triggered. This happens because, in step 5, we visit node x for the second time with the stack $\{_0\{_0$. The first visit to x (step 2) was done with the stack $\{_0$, which is a suffix of the new stack. Therefore, the traversal terminates at step 5.

Skipping steps 6–9 also skip the traversal of $x = init\_tuple()$ with the empty stack (see step 7) and subsequent traversals from this node with the empty stack.

Not only the efficiency, but also the precision of Algorithm 1 is augmented with respect to [20]. This is explained in Example 12.

Example 12 (Increased precision of Algorithm 1). Continuing Example 11, after step 4, where we reach node $b = x$ with the stack $\{_0\{_0$, we can exit the loop and reach node $x = init\_tuple()$ with the stack $\{_0\{_0$. This means that we only want the first element of the first element of the tuple assigned to x. Nevertheless, with the approach of

[20], after step 7, we can exit the loop again and reach the same node ($x = \mathit{init\_tuple}()$) with a different stack (the empty stack). Unfortunately, this means that we want the whole tuple, which is clearly imprecise.

Only increasing loops can produce non-termination. For this reason, Algorithm 1 detects loops (lines 31–32) and checks whether they are increasing with function isIncreasingLoop (line 33). This function uses the PDA of Figure 10 to determine whether the loop is increasing and in such a case the stack is emptied (line 34), i.e., the traversal continues unconstrained.

At this point, the reader might be wondering whether the PDA is actually needed, because Example 12 showed that even with an increasing loop, the PDA was not used. However, with some loops, the PDA is necessary. This is illustrated in Example 13.

Example 13 (Necessity of the PDA). Consider again the CE-PDG in Figure 13 and this time let d be the slicing criterion. A traversal with Algorithm 1 is shown in the following table:

Step	Start Node	Edge		Node Reached	Stack
0	-	-	-	(d)	$\bot$
1	(d)	flow	$\{_1$	({c,d}=x)	$\{_1$
2	({c,d}=x)	flow	$\varnothing$	(x)	$\{_1$
3	(x)	flow	$\{_0$	({x,a}=b)	$\{_1\{_0$
4	({x,a}=b)	flow	$\varnothing$	(b=x)	$\{_1\{_0$
5	(b=x)	flow	$\varnothing$	(x)	$\{_1\{_0$
6	(x)	flow	$\{_0 \rightsquigarrow *$	({x,a}=b)	$\bot$
7	({x,a}=b)	flow	$\varnothing \rightsquigarrow *$	(b=x)	$\bot$
8	(b=x)	flow	$\varnothing \rightsquigarrow *$	(x)	$\bot$
9	(x)	flow	$\{_0 \rightsquigarrow *$	({x,a}=b)	$\bot$

In this case, the algorithm would never stop the traversal because there is never a previous stack that is a suffix of the current stack. To stop the traversal we need the PDA. In step 6, we visit for the first time a previously visited edge (we find a cycle). At this point, the PDA is triggered and it determines that this is an increasing loop. Therefore, the PDA forces the change of the loop edges' constraints to $*$. This makes us stop the traversal at step 9 because the same node ($\{x, a\} = b$) is visited twice with the same stack ($\bot$) (one is a suffix of the other).

Theorem 1 ensures termination of the whole slicing process.

Theorem 1 (Termination of slicing)**.** *Let $G = (N, E)$ be a CE-PDG and let $n_{sc} \in N$ be a slicing criterion for G. Algorithm 1 terminates when it slices G with respect to n_{sc}.*

The proof of this theorem can be found in Section 4.7.

The reader could think that it would be a good idea to identify all increasing loops at CE-PDG construction time. Unfortunately, finding all cycles has an average complexity $\mathcal{O}(N^2EL)$, where L is the number of cycles. The maximum number of cycles in a graph is exponential ($L = 2^N$), and thus the worst-case complexity is $\mathcal{O}(2^N)$ [30]. Our approach avoids the problem of finding all loops. We just treat them on demand, when they are found by the slicing algorithm (i.e., we do not search for loops, we just find them during the CE-PDG traversal). So we only process those loops found in the slicing process; and processing a loop has a linear cost (in the worst case $\mathcal{O}(N)$, if the loop includes all program statements).

```
...
for(int i=0; i<max; i++){
  b = x;
  {x,a} = b;
}
```

(a) Increasing loop

$$\begin{array}{llll} b = x; & b = x; & b_1 = \mathbf{x_1}; & b_1 = x_1; \\ \{x, a\} = b; & x = \mathbf{b_1}; & b_2 = \mathbf{x_2}; & b_2 = x_2; \\ & a = \mathbf{b_2}; & x = b_1; & x_1 = \mathbf{b_{11}}; \\ & & a = b_2; & x_2 = \mathbf{b_{12}}; \\ & & & a = b_2; \end{array}$$

(b) The first steps of atomization for the increasing loop

Figure 12: Infinite unfolding of atomization.

4.6. Recursive data structures: constrained traversal versus atomization

In this section we show that our technique is not only an alternative to atomization but a fundamental improvement that solves an important problem that cannot be solved with atomization. Atomization cannot handle recursive data types such as lists, trees, linked lists (often implemented via a node that contains a value and a reference to another node), etc. The fundamental problem is that atomization would need to infinitely unroll the recursive data type.

Example 14 (Recursive unfolding cannot be atomized). Consider the program in Figure 12a, in which a recursive tuple x is unfolded in a loop. The inside of the loop cannot be atomized, as the loop contains two instructions: $b = x$ and $\{x, a\} = b$. Atomization would convert structures to multiple assignments. However, the depth of x and thus b is unknown.

Consider the atomization sequence shown in Figure 12: A first step (second column) is to split the pattern matching assignment into two separate assignments for x and a. However, b_1 and b_2 are not defined, so we must split b's assignment. The result (third column) still has undefined variables (x_1 and x_2). Defining those would require splitting b_1 (last column), resulting in a situation analogous to the first split, starting an infinite loop.

In contrast, the CE-PDG represents each field explicitly, performing the unfolding process as many times as

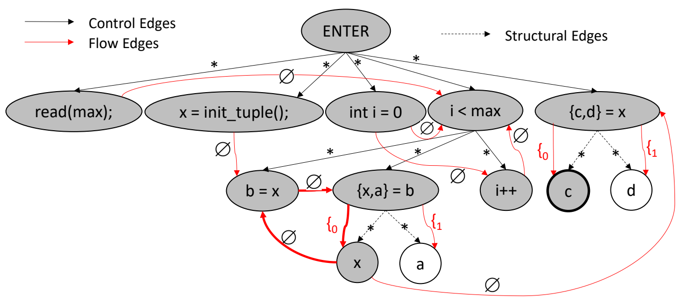


Figure 13: The CE-PDG for a program (Figure 12a) with recursively unfolding data structures.

required during the slicing traversal and thus, yielding to the correct slice, as can be seen in Figure 13.

4.7. Proof of Algorithm 1's termination

In this section, we prove Theorem 1 by showing that the traversal of all non-increasing loops is terminating with any initial stack. And the same happens with all loops with an asterisk constraint. Because emptying the stack is equivalent to traversing an asterisk constraint, then emptying the stack of increasing loops is enough to ensure that their traversal is also terminating with any initial stack.

In Section 4.5 we define loops and increasing loops (Definitions 7 and 8), the remaining kinds of loops can be defined in a similar fashion:

Definition 9 (Decreasing loop). A loop L is a *decreasing loop* if the number of closing constraints along L is greater than the number of opening constraints.

Definition 10 (Balanced loop). A loop L is a *balanced loop* if the number of closing and openings constraints along L is the same.

It is important to note here that, according to definitions 8, 9, and 10, a loop can only be decreasing, balanced, or increasing.

We also need to prove some properties of a loop that are needed to prove the theorem. These properties are captured by the following lemmas.

Lemma 1. *Let L be a non-increasing loop and S be a stack. The traversal of L with S is terminating.*

Proof. Non-increasing loops can be decreasing loops (Definition 9) or balanced loops (Definition 10). We prove each case separately.

Decreasing loops. First of all, for any initial stack, a first traversal of all the edges of L must be possible to explore this scenario. Otherwise, there cannot be a traversal loop due to unmatched constraints. Moreover, according to Definition 9 the path of L contains more closing constraints than opening constraints. When slicing the graph, and before iterating into a decreasing loop, there are two stack possibilities: the empty stack or a stack with a sequence of opening constraints at the top (see Figure 5b):

Empty stack ($\perp$). Since the constraint in L can be in any order, we need to consider two different scenarios:

1. *There is no suffix of L where the number of opening constraint is greater than the number of closing constraints (e.g., $[\{_0, \{_0, \}_0, \}_0, \}_0]$).* Since L is a decreasing loop, in this scenario we can summarise L as a sequence of n closing constraints. In this case, according to case 3 of Table 1, pushing any number j of closing constraint to an empty stack results in the empty stack.

$$\perp \xrightarrow{\}_{i,\forall i \in 1..j}}{}^{*} \perp \quad \text{and} \quad \perp \xrightarrow{]_{i,\forall i \in 1..j}}{}^{*} \perp$$

Then, the same node is reached twice with the same stack ($\perp$) and a second traversal is not processed as indicated by line 22 in Algorithm 1.

2. *There is a suffix of L where the number of opening constraints is greater than the number of closing constraints (e.g., $[\}_0, \}_0, \{_0, \{_0, \}_0]$).* In this scenario, there are always closing constraints that are consumed by the initial empty stack ($\perp$) according to case 3 of Table 1. On the other hand, the final part of L pushes more opening constraints than closing constraints and the stack calculated after the first traversal of the loop is not empty. As L is a decreasing loop, all the constraints pushed in the first traversal are consumed at the beginning of the second traversal. Finally, the final part of L generates the same sequence of positive constraints at finishing the second traversal. Therefore, as the same node is reached twice with the same stack, the traversal stops.

$$\perp \xrightarrow{L} S \xrightarrow{L}{}^{*} S$$

Stack with a sequence of opening constraints ($[C_{O_1}, \ldots, C_{O_n}]$). Cases 4 and 5 of Table 1 represent the two possibilities that can occur when pushing a closing constraint to a non-empty stack: the closing constraint balances the opening constraint at the top of the stack or it fails.

- According to case 5, if the closing constraint does not balance the opening constraint at the top of the stack, the traversal is aborted by an error. Hence, the traversal is finite.

$$[\ldots, \{_i] \xrightarrow{\}_j, \forall i \neq j} \mathit{error} \text{ and } [\ldots, [_i] \xrightarrow{]_j, \forall i \neq j} \mathit{error}$$

- On the contrary, case 4 shows that if the closing constraint balances the opening constraint at the top of the stack, the constraint is popped and the traversal continues. Since the number of elements of the stack is finite (an infinite stack would have been forever growing in an increasing loop), popping each opening constraint on the top of the stack with a closing constraint of the loop sequence will result in an empty stack at some point of the traversal. Note that this point of the traversal is not necessarily after traversing the whole loop L, but can be after traversing any intermediate edge of the loop. As the processing of any decreasing loop L with an empty stack ($\perp$) has already been proved terminating, then we can state that the traversal is finite.

$$[\{_1, \ldots, \{_n] \xrightarrow{L}^{m} S \xrightarrow{L}^{*} S$$

Balanced loops. With the same reasoning as in the decreasing loops, we differentiate two scenarios according to the initial form of the stack:

Empty stack ($\perp$). Since the constraint in L can be in any order, we need to consider the same two scenarios we did before:

- *There is no suffix of L where the number of opening constraint is greater than the number of closing constraints (e.g., $[\{_0, \{_0, \}_0, \}_0]$).* Since L is a balanced loop, in this scenario we can summarise L as an empty sequence of constraints. The result of iterating into a balanced loop of this type is the same stack, since each iteration of the loop itself balances the opening constraints with their complementary closing constraints. Hence, the initial node of the graph will be reached twice with the same stack, and the traversal will be stopped as indicated by line 22 in Algorithm 1.

$$\perp \xrightarrow{\varnothing}^{*} \perp$$

- *There is a suffix of L where the number of opening constraints is greater than the number of closing constraints (e.g., $[\}_0, \}_0, \{_0, \{_0]$).* In this scenario, there are always closing constraints that are consumed by the initial empty stack ($\perp$) according to case 3 of Table 1. On the other hand, the final part of L pushes more opening constraints than closing constraints and the stack calculated after the first traversal of the loop is not empty. As L is a balanced loop, all the constraints pushed in the first traversal are consumed at the beginning of the second traversal. Finally, the final part of L generates the same stack when the traversal of L finishes. Therefore, as the same node is reached twice with the same stack, the traversal stops.

$$\perp \xrightarrow{L} S \xrightarrow{L}^{*} S$$

Stack with a sequence of opening constraints ($[C_{O_1}, \ldots, C_{O_n}]$). Cases 4 and 5 of Table 1 represent the two possibilities that can occur when pushing a closing constraint to a non-empty stack: the closing constraint balances the opening constraint at the top of the stack or it fails.

- According to case 5, if the closing constraint does not balance the opening constraint at the top of the stack, the traversal is aborted by an error. Hence, the traversal is finite.

$$[\ldots, \{_i] \xrightarrow{\}_j, \forall i \neq j} \mathit{error} \text{ and } [\ldots, [_i] \xrightarrow{]_j, \forall i \neq j} \mathit{error}$$

- On the contrary, case 4 shows that if the closing constraint balances the opening constraint at the top of the stack, the constraint is popped and the traversal continues. Since the number of opening and closing constraints in L is the same, if any constraint from the stack is consumed by the initial closing constraints it will be restored later by the corresponding opening constraint contained in L. Then, the stack obtained after traversing L will be the same. As a result, since the same node is reached twice with the same stack, the traversal finishes.

$$[\{_1, \ldots, \{_n] \xrightarrow{L}^{*} [\{_1, \ldots, \{_n]$$

Finally, note that if we consume every opening constraint during the traversal of the closing constraints at the beginning of L, we will find ourselves in the previous scenario where the stack was empty and, thus, the traversal finishes too. □

Lemma 2. *Given a CE-PDG increasing loop L. There exists a stack S for which it is possible to infinitely traverse L with S.*

Proof. Following the same reasoning that we did in some particular decreasing loops, each iteration of an increasing loop can be summarised as a sequence of opening constraints of the form $[\{_1, ..., \{_n]$ independently of the sequence of constraints. Case 2 of Table 1 shows that opening constraints can always be traversed independently of the top of the stack. For this reason, the loop can be infinitely traversed generating an infinite stack. □

$$S \xrightarrow{\{_i}^{*} S \mathbin{++} [\{_1, ..., \{_n, \{_1, ..., \{_n, \ldots]$$

Lemma 3. *Given a CE-PDG increasing loop L, a stack S that allows Algorithm 1 to iterate at least once into it, and a loop L' which corresponds to L but replacing any access constraint by an asterisk constraint, then L' is not an increasing loop and it is not possible to infinitely traverse L' with S.*

Proof. According to Definition 8, a cyclic flow dependency path is an increasing loop if the sequence of constraints generated by traversing it belongs to the language induced by the PDA in Figure 10. The PDA cannot reach the final state if an asterisk constraint exists in L, thus, it cannot be an increasing loop. Moreover, the traversal of an asterisk constraint described in case 6 of Table 1 always results in an empty stack ($\perp$). Therefore, if an asterisk constraint is included in an edge of L, then the second time that this edge is traversed the same node will be reached again with the same stack ($\perp$). Therefore, a second traversal is never done as indicated by line 22 in Algorithm 1. □

Finally, we can prove Theorem 1:

Proof. The traversal of any sequence of nodes that is not a cycle (i.e., that does not represent a loop in the program) trivially terminates. Only loops can produce non-termination in Algorithm 1. But all loops are detected by the algorithm in line 22. According to Lemma 1, the traversal of all non-decreasing loops always terminates. On the other hand, as shown in Lemma 2 increasing loops can produce non-termination. However, all of them are detected by the PDA in Figure 10. When a increasing loop is detected by the algorithm the stack is emptied.

We know by case 6 in Table 1 that including an asterisk constraint in a path is equivalent to emptying the stack. Therefore, according to Lemma 3 it is not possible to infinitely traverse the increasing loops found in the traversal made by the algorithm. Hence, Algorithm 1 always terminates. □

4.8. Properties of the CE-PDG

In this section, we prove two important properties of the CE-PDG. We prove that it represents an improvement: slices are equal or smaller in size when compared to the PDG. We also prove that the slices produced contain all relevant elements.

Theorem 2 (Precision of the CE-PDG)**.** *Let P be a program, let $G = (N, E)$ and $G' = (N', E')$ be their corresponding PDG and CE-PDG. Let each edge in E be labelled with an empty constraint ($\varnothing$) and each edge in E' be labelled according to Section 4.1. Let sc represent each possible slicing criterion in the PDG, and* SLICE *be a function that applies Algorithm 1 for the given graph and slicing criterion.*

$$\forall sc \in N \mid \textit{SLICE}(G', sc) \subseteq \textit{SLICE}(G, sc)$$

According to Theorem 2, the slices produced by the CE-PDG are at least as precise as the ones produced by the PDG. To prove Theorem 2, we first prove two lemmas:

Lemma 4 (CE-PDG node equivalence)**.** *Let $G = (N, E)$ be a PDG, and $G' = (N', E')$ be a CE-PDG. Let $E'_s \subset E'$ be the set of structural edges in G', let ${E'}^*_s$ be the reflexive and transitive closure of E'_s and let* $\textsc{subtree}(n) = \{n\} \cup \{m \mid (n, m) \in {E'}^*_s\}$.

$$\forall n' \in N' \mid \exists! n \in N \mid n' = n \vee n' \in \textit{SUBTREE}(n)$$

Proof. Trivial, by construction of the CE-PDG. The CE-PDG has the following two properties:

1. $\forall n \in N \ . \ n \in N'$
2. $\forall n' \in N', n \notin N \ . \ n' \in \textsc{subtree}(n''), n'' \in N$

Additionally, each node in the PDG is unique because each represents a different statement (even if two statements have the same code). Therefore, a node can either be common to both graphs (given 1), or represents a member of a data structure in a given node (given 2), fulfilling either the first or the second part of the lemma. □

We use Lemma 4 to create a function $\mathcal{N} :: N' \to N$, defined as $\mathcal{N}(x) = y \in N \mid x = y \vee x \in \textsc{subtree}(y)$, given a node from a CE-PDG, returns its matching node in the corresponding PDG.

Lemma 5 (CE-PDG edge equivalence)**.** *Let $G = (N, E)$ be a PDG, $G' = (N', E')$ be a CE-PDG, and $n'_a, n'_b \in N'$ be two nodes in G'.*

$$\forall (n'_a, n'_b) \in E' \mid (\mathcal{N}(n'_a), \mathcal{N}(n'_b)) \in E \vee \mathcal{N}(n'_a) = \mathcal{N}(n'_b)$$

Proof. The condition will be true if any of the $\vee$ operands is true, so we can analyse them separately.

There are two different possibilities for the condition $(\mathcal{N}(n'_a), \mathcal{N}(n'_b)) \in E$ to hold:

1. The edge exists between the same nodes in the CE-PDG and in the PDG: $n'_a, n'_b \in N$.
2. The edge exists in the PDG and its source and/or target in the CE-PDG points to a data structure node n', with $n' \notin N$.

On the other hand, there is one possibility for the condition $\mathcal{N}(n'_a) = \mathcal{N}(n'_b)$ to hold:

3. The edge does not exist in the PDG, but connects a data structure element to its parent or vice-versa, both nodes represents the same statement $n \in N$, with $\mathcal{N}(n'_a) = n$ and $\mathcal{N}(n'_b) = n$.

The CE-PDG features three kinds of edges, all of them backed by one of the previous three cases:

- Control edges (case 1): these edges remain unchanged in the CE-PDG.
- Flow edges: these can be classified according to their label (which can be an access or empty constraint).
  - Empty constraints (cases 1 and 2): these edges connect a variable definition to a usage. If the source and target are not members of a data structure, they remain unchanged (case 1). Otherwise, the source and/or target points to a data structure element (case 2).
  - Access constraints (case 3): these edges connect a member of a data structure to its parent or vice-versa.

- Structural edges (case 3): these edges connect a node that contains a data structure to a member of that data structure.

Thus, all edges present in the CE-PDG either correspond to a edge in the PDG (cases 1 and 2) or represent a reflexive edge (case 3), which has no effect on program slicing. □

We can now prove Theorem 2:

Proof. The PDG and CE-PDG are equivalent node by node (Lemma 4) and edge by edge (Lemma 5), so the slices produced by them would be equivalent. However, the PDG is labelled only with empty constraints and does not contain structural edges. For this reason, in some cases, the constraints present in the CE-PDG (see case 5 in Table 1 or line 45 in Algorithm 1) and the combination of structural and flow edges (lines 10-12 in Algorithm 1) do not allow the traversal to continue, making CE-PDG slices smaller. Thus, every slice produced by the CE-PDG will be equal to or smaller than the equivalent slice produced by the PDG. □

For the next theorem we need to provide a formal definition of complete slice:

Definition 11 (Complete slice). Let P be a program, $G = (N, E)$ its PDG, and $n \in N$ a slicing criterion for P. A set of nodes $S \subseteq N$ is a *complete slice* if and only if the sequence of values produced in the variables of n when P is executed is a prefix of the sequence of values produced in n when S is executed.

Lemma 6. *Let P be a program. Let $G = (N, E)$ be the PDG of P, $G' = (N', E')$ be its corresponding CE-PDG, and $sc \in N'$ be a slicing criterion. Not traversing flow edges after structural edges during the computation of* SLICE(G', sc) *does not affect the completeness of said slice.*

Proof. In the CE-PDG, a structural edge is only traversed by the slicing algorithm (Algorithm 1) after reaching a node inside a data structure. This node can either be a *definition* or a *usage*. We prove that the traversal restriction applied to the CE-PDG (Algorithm 1, lines 10-12) generates complete slices in both situations:

- **The reached node ($n' \in N'$) is a definition**. Let $n = \mathcal{N}(n')$ be the CE-PDG node with the whole data structure that contains n'. According to the CE-PDG construction algorithm (Section 4.1), there is a path p_s from n to n' formed by structural edges (labelled with asterisk constraints), but also another parallel path p_f from n to n' formed by flow edges (labelled with their associated opening constraints). Both paths are traversed by Algorithm 1 and both of them reach the same nodes in the data structure. If we reach n' from sc with a stack S and we traverse p_s from n', then the stack will be emptied (see Table 1, case (6)) with the first structural edge; thus traversing flow edges after the structural edges would be done with an empty stack (this would cause the algorithm to lose the context accumulated in the stack, which, in turn, would probably lead it to collect nodes that do not affect the slicing criterion). However, if we traverse p_f we will reach the same nodes in the data structure but with a properly updated stack S' (see Table 1, case (2)). Moreover, Algorithm 1 traverses all flow edges from the nodes in p_f with the proper stacks. Therefore, the traversal of flow edges (with an empty stack) after traversing structural edges is unnecessary and can be ignored, preserving completeness.
- **The reached node ($n' \in N'$) is a use**. Let $n = \mathcal{N}(n')$ be the CE-PDG node with the whole data structure that contains n'. According to the CE-PDG, when n' is a use, there is only one path p_s from n to n', formed by structural edges. Traversing p_s is only necessary to include in the slice the hierarchical structure of n' inside the data structure. Therefore, there is no need to traverse flow edges after reaching a node with structural edges because the value of n' cannot depend on the nodes reachable through structural edges. In fact, if the value of n' depends on a node n_0, then there must exist a flow edge (n_0, n') in the graph, and this edge is traversed by Algorithm 1. Thus, the traversal of flow edges after traversing structural edges is unnecessary and can be ignored, preserving completeness.

□

Theorem 3 (Completeness of the CE-PDG). *Let P be a program. Let G be the PDG of P, G' be its corresponding CE-PDG, and $sc \in N'$ be a slicing criterion. If* SLICE$(G, \mathcal{N}(sc))$ *is a complete slice, then* SLICE(G', sc) *is also a complete slice.*

Proof. The CE-PDG slicing algorithm contains two traversal limitations compared to the standard PDG slicing algorithm : (i) flow edges are not traversed after a structural edge is traversed and (ii) the traversal limitation imposed by access constraints. As proved in Lemma 6, limitation (i) does not affect completeness. On the other hand, to prove limitation (ii) we consider all possible situations that concern the management of access constraints. We divide them into two different cases:

1. **SLICE$(G, \mathcal{N}(sc))$ does not include any node that contains a data structure**. This case is trivially proved using Lemmas 4 and 5. The nodes without data structures and the edges connecting them in the PDG remain unchanged in the CE-PDG. Then, during the algorithm, the same edges are traversed and the same nodes are included in the slice in both graphs (SLICE$(G, \mathcal{N}(sc))$ = SLICE(G', sc)). Thus, since the PDG slice is complete, the CE-PDG slice is also complete.

2. **SLICE$(G, \mathcal{N}(sc))$ includes at least one node with a data structure**. Inside this case, we consider two different scenarios:

(i). No element included in the slice has been defined inside a data structure (e.g. `{A,B} = X` or `X = {A,B}`). According to Table 1, only opening constraints (associated to definitions inside data structures) are pushed into the stack (case (2)). Opening constraints at the top of the stack are the ones that can limit the traversal during the CE-PDG slicing algorithm (see case (5) in Table 1). However, since no opening constraint is collected during the traversal (because all elements included in the slice have been defined outside a data structure), we reach a scenario similar to the previous case. The elements outside data structures are connected by flow and control dependencies in the same way in both the PDG and CE-PDG (as stated in Lemma 5 (case 1)). Additionally, since closing constraints can be traversed while the stack is empty, the elements inside data structures which are used are also included in the slice, ensuring the CE-PDG to include the same code as the PDG. Thus, the PDG and the CE-PDG include the same code in the slice, resulting in the completeness of the CE-PDG slice.

(ii). At least one element in the slice has been defined inside a data structure (e.g. `{A,B} = X`). In this case, an opening constraint is pushed into the traversal stack, enabling the possibility to limit the traversal. When slicing the CE-PDG after pushing one (or more) opening constraints, there are two possible situations:

(a) No closing constraints are reached during the same CE-PDG traversal inside the procedure. In this case the traversal is not limited by the CE-PDG, and the slice includes only those elements in the data structures that contain the definitions included, which are reached by flow dependences. In this case, the CE-PDG makes possible to exclude the elements defined in the same data structure that are not reached by flow edges, which do not influence the slicing criterion. Thus, $\text{SLICE}(G', sc) \subseteq \text{SLICE}(G, \mathcal{N}(sc))$, being both complete.

(b) Other closing constraints are reached during the same CE-PDG traversal inside the procedure. We prove this case by induction on the depth d of the definition inside its data structure. In the proof, S represents the value of the stack before reaching this node, O_i represents the opening constraint with index i, and C_i represents the closing constraint with the same index i.

(Base case: $d = 1$). First, in the case where the definition is at depth level 1, the traversal collects the constraint at the flow edge (O_i) and pushes it into the stack (S, O_i). When reaching the data structure with closing constraints, only the edge with the complementary constraint (C_i) is traversed (case (4) in Table 1). As a result, the element required at the data structure with uses is the one at the same position that the one at the data structure with definitions. Since no more elements are required to compute the value of the slicing criterion, and according to the flow dependences defined at the CE-PDG (which are extracted from the flow dependences of the PDG), the CE-PDG slice is complete.

(Induction hypothesis: $d = n$). We assume as the induction that completeness holds for definitions inside data structures located at depth level n, which generates a sequence of opening constraints S_O.

(Inductive case: $d = n+1$). We prove that the theorem holds for any expression with depth $n + 1$. When a definition node inside a data structure at depth $n+1$ is reached with an initial stack S, the traversal of the flow edge with constraint O_j pushes it to the slice resulting in the stack S, O_j. Then, the traversal reaches depth n and, according to the induction hypothesis it includes to the stack a sequence of n opening constraints (S_O). As a result the stack results in the new stack S, O_j, S_O. Considering the induction hypothesis, when the algorithm reaches a data structure that contains a use, it pops this sequence of constraints reaching depth level n and preserving completeness (then, the stack becomes S, O_j again). Hence, we are in the same situation as the base case, where only the edge with the complementary constraint (C_j) is traversed according to case (4) in Table 1, generating a complete slice.

□

5. Implementation and empirical evaluation

Comparing our implementation against other slicers is not the best way to assess the proposed stack extension to the PDG because we would find big differences in the PDG construction time, slicing time, and slicing precision due to differences in the libraries used, different treatments for syntax constructs such as list comprehensions, guards, etc. Therefore, we would not be able to assess the specific impact of the stack on the slicer's precision and performance. The only way to do a fair comparison is to implement a single slicer that is able to build and slice the PDG with and without constraints.

All the algorithms and ideas described in this paper have been implemented in a slicer for Erlang called e-Knife. e-Knife can produce slices based on either the PDG or the CE-PDG. Thus, it allows us to know exactly the additional cost required to build and traverse the constraints, and the extra precision obtained by doing so. e-Knife is a Java program with 13355 LOC (excluding comments and empty lines). It is an open-source project and is publicly available[4].

Additionally, anyone can slice a program via a web interface[5], without the need to build the project locally. Large or very complex programs may run into the memory and time limitations that are in place to avoid abuse.

To evaluate e-Knife, we used Bencher[6], a program slicing benchmark suite for Erlang. All the benchmarks were interprocedural programs, so we have created a new intraprocedural version of them (by inlining functions). This intraprocedural version has been made publicly available (every benchmark in Bencher has a link to its intraprocedural version). To evaluate the techniques proposed throughout this work, we have built both graphs (PDG and CE-PDG) for each of the intraprocedural benchmarks. Then, we sliced both graphs with respect to all possible slicing criteria[7], which guarantees that there is no bias in the selection of slicing criteria.

The benchmarks were run on a Intel Xeon E-2136 CPU running Debian Linux 11 with 32GB of DDR4 RAM available. All processes were stopped, except for *init* and *sshd*. We strictly followed the methodology proposed by Georges et al. [31]. Each program's graph was built 1001 times, and the graphs were sliced 1001 times per criterion. To ensure real independence, the first iteration was always discarded (to avoid the influence of dynamically loading libraries to physical memory, data persisting in the disk cache, etc.). From the 1000 remaining iterations, we retained a window of 10 measurements when steady-state performance was reached, i.e., once the coefficient of variation (CoV, the standard deviation divided by the mean) of the 10 iterations falls below a preset threshold of 0.01 or the lowest CoV if no window reached it. It is with these 10 iterations that we computed the average time taken by each operation (building each graph or slicing each graph w.r.t. each criterion).

The results of the experiments performed are summarized in Table 2. The two columns (**PDG**, **CE-PDG**) display the average time required to build each graph. Building the CE-PDG, as in the PDG, is a quadratic operation; and the inclusion of labels in the edges is a linear operation w.r.t. the amount of nodes in the graph. Thus, building the CE-PDG is only slightly slower than its counterpart. The other columns are as follows (average values are w.r.t. all slicing criteria):

DSs: the number of data structure access constraints in the CE-PDG. It is a metric to measure the amount and size of the composite data structures.

Function: the name of the function where the slicing criterion is located.

#SCs: the number of slicing criteria in that function.

PDG, CE-PDG: the average time required to slice the corresponding graph.

Slowdown: the average additional time required (with 95% error margins), when comparing the CE-PDG with the PDG. For example, on average (last row), the computation of each slice is 3.54 times slower in the CE-PDG.

Red. Size: the average reduction in the sizes of slices (with 95% error margins)[8]. It is computed as $(A - B)/A$ where A is the size (number of AST nodes) of the slice computed with the standard (field-insensitive) algorithm and B is the size (number of AST nodes) of the slice computed with the field-sensitive algorithm (Algorithm 1). This way of measuring the size of the slices is much more precise and fair. LOC is not proper because it can ignore the removal of subexpressions. PDG/CE-PDG nodes is also not a good solution because the CE-PDG includes nodes and edges not present in their PDG counterparts, therefore they are incomparable.

The averages shown at the bottom of the table are the averages of all slicing criteria and not the averages of each function's average.

The first 13 benchmarks (set A) are benchmarks with complex data structures but without cycles, while the rest of the benchmarks (set B) do contain cycles. In set A, each slice produced by the CE-PDG is around four times slower. However, this has little impact, as each slice consumes just hundreds of milliseconds. As can be seen in each row, generating the graph is at least 3 orders of magnitude slower than slicing it. This increase in time is offset by the average reduction of the slices, which is 11.27%. This reduction goes up to 37.14% in function `main/1` from `bencher1A`, as it contains complex data structures that can be efficiently sliced with the CE-PDG. Therefore, our technique reduces the size of the slices by $11.27 \pm 2.91\%$ at almost no cost (only a few μs).

[4]`https://mist.dsic.upv.es/git/program-slicing/e-knife-erlang`

[5]`https://mist.dsic.upv.es/e-knife-constrained/`

[6]`https://mist.dsic.upv.es/bencher/`

[7]Each variable use or definition in all functions that contain complex data structures.

[8]The minimum value for Red. Size is 0, even if some error margins hint at otherwise. The slices produced by the PDG can never be smaller than those produced by the CE-PDG.

Table 2: Summary of experimental results for *Bencher*, comparing the PDG (without constraints) to the CE-PDG (with constraints).

	Graph Generation			**Slice**					
Program	**PDG**	**CE-PDG**	**DSs**	**Function**	**#SCs**	**PDG**	**CE-PDG**	**Slowdown**	**Red. Size**
`bench1A.erl`	3230.08ms	3233.42ms	515	`getLast/2`	26	59.12μs	269.26μs	5.68 ± 1.38	$14.88 \pm 3.16\%$
				`getNext/3`	174	228.10μs	1014.41μs	4.62 ± 0.29	$13.09 \pm 1.57\%$
				`getStringDate/1`	11	22.56μs	164.10μs	8.62 ± 2.05	$8.67 \pm 3.88\%$
				`main/1`	57	800.96μs	1644.59μs	2.21 ± 0.20	$37.14 \pm 6.97\%$
`bench3A.erl`	37.53ms	37.54ms	4	`tuples/2`	22	30.36μs	100.91μs	3.36 ± 0.20	$5.46 \pm 2.03\%$
`bench4A.erl`	53.22ms	53.26ms	20	`main/2`	31	64.19μs	163.78μs	2.65 ± 0.19	$20.79 \pm 5.38\%$
`bench5A.erl`	32.87ms	32.90ms	8	`lists/2`	18	44.67μs	136.81μs	3.12 ± 0.17	$6.51 \pm 2.02\%$
`bench6A.erl`	236.33ms	236.42ms	37	`ft/2`	34	65.87μs	249.55μs	4.11 ± 0.28	$8.71 \pm 2.45\%$
				`ht/2`	16	20.37μs	77.71μs	4.07 ± 0.25	$10.79 \pm 3.69\%$
`bench9A.erl`	117.30ms	117.45ms	16	`main/2`	18	136.84μs	210.77μs	1.47 ± 0.07	$1.38 \pm 1.04\%$
`bench11A.erl`	15.10ms	15.13ms	6	`lists/2`	16	35.92μs	108.67μs	2.92 ± 0.33	$6.47 \pm 2.15\%$
`bench12A.erl`	1042.13ms	1042.94ms	103	`add/4`	26	78.11μs	273.66μs	5.02 ± 1.13	$15.38 \pm 4.13\%$
				`from_ternary/2`	9	17.28μs	86.14μs	6.18 ± 1.84	$3.56 \pm 2.61\%$
				`main/3`	39	75.28μs	196.66μs	4.02 ± 0.79	$8.43 \pm 6.19\%$
				`mul/3`	21	41.43μs	143.00μs	4.95 ± 1.17	$2.74 \pm 1.27\%$
				`to_ternary/2`	13	57.37μs	126.01μs	4.13 ± 1.91	$1.02 \pm 1.32\%$
`bench14A.erl`	2300.65ms	2301.05ms	75	`main/2`	81	73.50μs	284.47μs	3.89 ± 0.34	$8.62 \pm 2.46\%$
`bench15A.erl`	1182.25ms	1182.66ms	44	`main/4`	71	165.03μs	268.53μs	3.84 ± 0.75	$1.72 \pm 1.53\%$
`bench16A.erl`	159.12ms	159.26ms	16	`word_count/5`	36	59.85μs	130.93μs	2.67 ± 0.23	$7.72 \pm 2.68\%$
`bench17A.erl`	44.94ms	45.05ms	8	`mug/3`	19	41.23μs	92.11μs	2.43 ± 0.22	$5.59 \pm 3.02\%$
`bench18A.erl`	49.41ms	49.52ms	8	`mbe/2`	19	58.36μs	117.12μs	2.28 ± 0.23	$7.38 \pm 4.59\%$
Totals and averages for set A					**757**	164.67μs	491.93μs	3.88 ± 0.50	$11.27 \pm 2.91\%$
`bench1B.erl`	2640.65ms	2643.99ms	493	`main/1`	273	1538.26μs	6648.83μs	4.24 ± 0.23	$24.02 \pm 1.93\%$
`bench2B.erl`	73.61ms	73.62ms	2	`main/2`	17	79.86μs	168.92μs	3.07 ± 0.70	$0.43 \pm 0.58\%$
`bench3B.erl`	35.16ms	35.17ms	4	`tuples/2`	18	51.44μs	133.12μs	2.57 ± 0.07	$4.33 \pm 1.21\%$
`bench4B.erl`	26.49ms	26.52ms	20	`main/2`	39	89.57μs	240.93μs	2.66 ± 0.17	$13.11 \pm 3.79\%$
`bench5B.erl`	18.31ms	18.34ms	8	`lists/2`	11	54.50μs	136.21μs	2.50 ± 0.10	$6.88 \pm 0.85\%$
`bench6B.erl`	54.35ms	54.42ms	25	`tuples/2`	42	46.94μs	129.73μs	3.01 ± 0.22	$8.25 \pm 1.62\%$
`bench8B.erl`	87.29ms	87.39ms	16	`main/2`	42	204.09μs	607.50μs	3.06 ± 0.18	$0.73 \pm 0.68\%$
`bench9B.erl`	34.89ms	34.95ms	10	`main/2`	17	187.93μs	294.62μs	1.51 ± 0.06	$1.16 \pm 0.85\%$
`bench10B.erl`	97.22ms	97.39ms	18	`main/1`	35	263.35μs	656.71μs	2.85 ± 0.26	$2.23 \pm 1.15\%$
`bench11B.erl`	12.95ms	12.98ms	8	`lists/2`	13	45.68μs	118.99μs	2.56 ± 0.13	$8.02 \pm 2.08\%$
`bench12B.erl`	294.70ms	295.25ms	79	`main/3`	88	917.43μs	2692.51μs	3.23 ± 0.40	$2.61 \pm 2.67\%$
`bench13B.erl`	27.08ms	27.11ms	4	`main/0`	22	135.10μs	253.26μs	1.92 ± 0.11	$0.48 \pm 0.39\%$
`bench14B.erl`	147.19ms	147.48ms	49	`main/2`	51	112.06μs	371.92μs	3.21 ± 0.44	$13.34 \pm 4.49\%$
`bench15B.erl`	217.26ms	217.49ms	38	`main/4`	65	286.95μs	495.93μs	2.70 ± 0.41	$8.78 \pm 2.84\%$
`bench16B.erl`	102.53ms	102.62ms	16	`word_count/5`	40	131.26μs	349.56μs	3.11 ± 0.26	$4.14 \pm 1.53\%$
`bench17B.erl`	57.63ms	57.64ms	8	`mug/3`	19	166.01μs	352.78μs	2.11 ± 0.09	$4.96 \pm 2.38\%$
`bench18B.erl`	62.95ms	63.03ms	8	`mbe/2`	19	273.12μs	629.87μs	2.34 ± 0.11	$0.05 \pm 0.10\%$
Totals and averages for set B					**811**	704.73μs	2735.10μs	3.30 ± 0.26	$11.79 \pm 2.05\%$
Totals and averages					**1568**	444.00μs	1652.14μs	3.54 ± 0.38	$11.54 \pm 2.47\%$

If we consider programs with cycles (set B), due to the analysis of loops, the slowdown is around three to four times slower (the slowdown is 3.30) and the reduction in the size of the slices is 11.79 ± 2.05. If we consider all benchmarks, our technique reduces the size of the slices by 11.54% with a slowdown of 3.54. This is a good result: for many applications (e.g., debugging) reducing the suspicious code over 11.54% with the cost of increasing the slicing time by only a few milliseconds is a good trade-off to make.

Regarding the effect of data structures on the results, we can see that benchmarks with more data structures have the potential to produce more precise slices with the CE-PDG.

5.1. Slicing larger programs

Although Bencher is useful because it contains challenging slicing programs specifically designed to test program slicers, and because it helps determine whether a slicer is complete by comparing the slices produced by a given implementation against a known minimal slice. However, the size of its benchmarks is rather small, with files ranging from 12 to 114 LOC. To test the efficiency of the CE-PDG on larger programs with more complex functions, we have performed an experimental evaluation of the *erlsom* Erlang library, which implements an XML parser[9]. This library contains 13k LOC across 22 source files. The evaluation followed the same methodology and process as Bencher's evaluation, generating over 4k graphs and analysing over 35 million slices. The results can be seen on

Table 3: Summary of experimental results for the *erlsom* library, comparing PDG to the CE-PDG.

		Graph Generation			Slice				
Program	**LOC**	**PDG**	**CE-PDG**	**DSs**	**#SCs**	**PDG**	**CE-PDG**	**Slowdown**	**Red. Size**
`erlsom.erl`	182	0.279s	0.280s	100	172	29.48ms	49.69ms	1.68 ± 0.55	5.85 ± 8.17%
`erlsom_add.erl`	28	0.224s	0.224s	34	60	46.87ms	81.74ms	1.69 ± 0.84	4.23 ± 12.19%
`erlsom_compile.erl`	693	25.662s	25.665s	1002	1335	57.09ms	102.70ms	1.71 ± 0.58	7.00 ± 10.72%
`erlsom_example_value.erl`	270	1.515s	1.518s	299	349	35.59ms	49.64ms	1.51 ± 0.78	11.31 ± 12.18%
`erlsom_lib.erl`	939	6.729s	6.736s	754	881	87.35ms	159.15ms	1.67 ± 0.66	11.41 ± 8.81%
`erlsom_parse.erl`	893	84.991s	84.995s	891	1224	134.94ms	232.84ms	1.67 ± 0.48	14.21 ± 7.62%
`erlsom_parseXsd.erl`	504	52.615s	52.629s	1608	50	20.00ms	34.38ms	1.65 ± 0.39	0.24 ± 3.67%
`erlsom_pass2.erl`	605	15.463s	15.467s	695	1166	52.07ms	81.51ms	1.60 ± 0.59	10.47 ± 9.81%
`erlsom_sax.erl`	74	0.818s	0.818s	60	111	44.70ms	71.36ms	1.59 ± 0.63	6.96 ± 9.04%
`erlsom_sax_latin1.erl`	1140	18.396s	18.401s	733	1564	54.80ms	96.75ms	1.69 ± 0.68	9.69 ± 9.35%
`erlsom_sax_latin9.erl`	1140	18.527s	18.532s	734	1559	55.02ms	96.84ms	1.69 ± 0.68	9.84 ± 9.46%
`erlsom_sax_lib.erl`	160	0.591s	0.593s	272	261	43.76ms	92.83ms	1.97 ± 0.61	9.00 ± 7.38%
`erlsom_sax_list.erl`	1140	18.703s	18.711s	1249	1806	60.69ms	135.42ms	1.91 ± 1.64	10.77 ± 8.09%
`erlsom_sax_utf8.erl`	1140	21.556s	21.561s	736	1600	56.13ms	98.26ms	1.69 ± 0.68	9.51 ± 9.42%
`erlsom_sax_utf16be.erl`	1140	21.521s	21.526s	736	1600	55.80ms	97.75ms	1.68 ± 0.68	9.51 ± 9.42%
`erlsom_sax_utf16le.erl`	1140	20.414s	20.420s	748	1664	66.19ms	109.91ms	1.66 ± 0.68	9.21 ± 9.42%
`erlsom_simple_form.erl`	126	2.589s	2.590s	201	201	48.19ms	71.45ms	1.39 ± 0.83	15.36 ± 9.50%
`erlsom_type2xsd.erl`	190	1.831s	1.832s	289	324	84.39ms	300.61ms	2.17 ± 1.66	9.29 ± 8.00%
`erlsom_ucs.erl`	158	3.356s	3.358s	222	309	94.11ms	130.81ms	1.43 ± 0.58	12.02 ± 9.20%
`erlsom_write.erl`	606	16.591s	16.598s	719	1010	118.65ms	207.84ms	1.66 ± 0.64	11.51 ± 9.14%
`erlsom_writeHrl.erl`	240	1.446s	1.447s	201	294	46.68ms	73.63ms	1.51 ± 0.87	14.16 ± 12.53%
`ucs.erl`	351	4.075s	4.078s	211	351	53.06ms	65.13ms	1.30 ± 0.61	13.90 ± 12.52%
Totals and averages	**12859**	15.359s	15.363s	**12494**	**17891**	67.37ms	121.40ms	1.69 ± 0.88	10.24 ± 9.45%

Table 3, which is structured like Table 2.

From the graph generation columns, we can see that the cost of building the CE-PDG is almost equal to the cost of building the PDG. As already known, the cost scales with program size, with the average growing around three orders of magnitude (from milliseconds to seconds) with respect to Bencher, reaching a maximum of 85s. Any improvement on the algorithms used to compute the PDG would benefit the CE-PDG.

Regarding the time needed to slice the CE-PDG, it also grows three orders of magnitude with respect to Bencher, due to size. On the bright side, the increase in the size of the programs lowers the relative speed of the graphs. In Bencher, the CE-PDG was 3 times slower than the PDG and, in this case, the slowdown is much lower (1.69), which may indicate that this technique scales well with size.

Finally, the reduction in slice size is much more stable in *erlsom* than in Bencher, achieving an average of 10.24%, with a maximum of 15% in `erlsom_simple_form.erl`. However, these results have a much higher variance, probably due to a higher variance in the size of slices. In conclusion, in larger programs we achieve a reduction in the size of the slices of 10.24% at the cost of having a slowdown of 1.79 to produce slices. This slowdown is 50 ms on average.

[9]The exact version used corresponds to commit hash `e00d2c7426fe7a8c49f8959da38a11d21b8b2168`, and is available for download at `https://github.com/willemdj/erlsom/archive/e00d2c7.zip`.

6. Related work

Transitive data dependence analysis has been extensively studied [28, 32]. Less attention has received, however, the problem of field-sensitive data dependence analysis [33, 18, 34, 25]. The existing approaches can be classified into two groups: those that treat composite structures as a whole [35, 1, 36, 33], and those that decompose them into small atomic data types [23, 37, 24, 38, 18, 39, 40, 41]. The latter approach is often called *atomization* or *scalar replacement*, and it basically consists of a program transformation that recursively disassembles composite structures to their primitive components. However, slicing over the decomposed structures usually uses traditional dependence graph based traversal [39, 40, 41] which limits the accuracy. Other important approaches for field-sensitive data dependence analysis of this kind are [34, 25, 33]. Litvak et al. [33] proposed a field-sensitive program dependence analysis that identifies dependencies by computing the memory ranges written/read by definitions/uses. Späth et al. [25] proposed the use of pushdown systems to encode and solve field accesses and uses. Snelting et al. [42] present an approach to identify constraints over paths in dependence graphs. Our approach combines atomization with the addition of constraints checked by pushdown systems to improve the accuracy of slicing composite data structures.

There exist approaches for the field-sensitive slicing of some specific data structures. If we refer to arrays, some static proposals consider the whole array as a variable, and each access as a definition or use of that variable [35]. However, this technique produces complete, but unnecessarily large program slices [24]. The PDG variant of Ottenstein and Ottenstein [1] represents composite data types providing a node for each one of its subexpressions, and provides special *select* and *update* operators to access the elements of an array. Other static approaches rely on determining whether two statically unknown vector accesses can refer to the same memory location during runtime [43, 44]. Some papers [45, 46, 47] propose algorithms that demonstrate the absence of a flow dependence between array accesses under certain conditions.

Some approaches [48, 41] have been also proposed to accurately represent the inner structure of objects and the dependencies between their data members. Most object-oriented approaches [26, 27, 41, 49] are based on the same principle: object variables and their inner data members are unfolded in a tree-like representation when used at function calls. This allows for the generation of dependencies between data members of a particular object and to accurately slice off those data members of an object that are not affecting the slicing criterion. Our representation is inspired by this tree-like structure, but with some differences. In our representation, the tree structure is connected with a new kind of edges (structural edges) instead of control edges. This allows us to apply a different slicing behaviour for structural edges without interfering in the traversal restrictions given to control edges in some slicing algorithms [50]. Additionally, our tree structure is connected not only with structural edges but also with flow edges; providing a more realistic representation of the dependencies between a composite structure and all its elements.

Severals works have tried to adapt the PDG for functional languages dealing with tuple structures in the process [51, 52, 53, 34]. Some of them with a high abstraction level [54], and other ones with a low granularity level. Silva et al. [19] propose a new graph representation for the sequential part of Erlang called the Erlang Dependence Graph. Their graph, despite being built with the minimum possible granularity (each node in the graph corresponds to an AST node) and being able to select subelements of a given composite data structure, does not have a mechanism to preserve the dependency of the tuple elements when a tuple is collapsed into a variable; i.e., they do not solve the *slicing pattern matching* problem (for instance, they cannot solve the program in Figure 2). In contrast, although our graph is only fine-grained at composite data structures, we overcome their limitations by introducing an additional component to the graph, the constrained edges, which allow us to carry the dependence information between definition and use even if the composite structure is collapsed in the process.

7. Conclusion

Static analyses often use a representation of the program being analyzed, and this representation strongly influences their correctness, completeness, and performance. In the particular case of intraprocedural program slicing, the standard representation used is the PDG. Unfortunately, the PDG's data dependencies are imprecise when modelling composite data structures. In particular, the information stored in the nodes of the PDG (i.e., statements) is often inappropriate when representing composite data structures.

To solve this problem, we present a generalization of the PDG called CE-PDG where (i) the inner components of the composite data structures are unfolded into a tree-like representation, providing an independent representation for their subexpressions and allowing us to accurately define intra-statement data dependencies, and (ii) the edges are augmented with constraints (constrained edges), which allows the propagation of the component dependence information through the traversal of the graph during the slicing process. As a result, the CE-PDG allows the user to select any subexpression of a data structure as the slicing criterion and it computes accurate slices for (recursive) composite data structures. It ignores irrelevant elements inside the same statement and allows for the transference of data dependence information through the compression and expansion of composite structures. An evaluation of our approach shows a slowdown of 3.88/3.30 and a reduction of the slices of 11.27%/11.79% for programs without/with cycles.

References

[1] K. J. Ottenstein, L. M. Ottenstein, The program dependence graph in a software development environment, SIGSOFT Software Engineering Notes 9 (3) (1984) 177–184. doi:10.1145/390010.808263.
URL http://doi.acm.org/10.1145/390010.808263

[2] J. Ferrante, K. J. Ottenstein, J. D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9 (3) (1987) 319–349.

[3] J. Silva, A vocabulary of program slicing-based techniques, ACM Computing Surveys 44 (3) (June 2012).

[4] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (3) (1995) 121–189.

[5] M. Weiser, Program Slicing, in: Proceedings of the 5th international conference on Software engineering (ICSE '81), IEEE Press, Piscataway, NJ, USA, 1981, pp. 439–449.

[6] A. Hajnal, I. Forgács, A demand-driven approach to slicing legacy COBOL systems, Journal of Software Maintenance 24 (1) (2012) 67–82.
URL http://dblp.uni-trier.de/db/journals/smr/smr24.html#HajnalF12

[7] R. A. DeMillo, H. Pan, E. H. Spafford, Critical slicing for software fault localization, SIGSOFT Softw. Eng. Notes 21 (3) (1996) 121–134. doi:10.1145/226295.226310.
URL http://doi.acm.org/10.1145/226295.226310

[8] A. Majumdar, S. J. Drape, C. D. Thomborson, Slicing obfuscations: Design, correctness, and evaluation, in: Proceedings of the 2007 ACM Workshop on Digital Rights Management, DRM '07, ACM, New York, NY, USA, 2007, pp. 70–81. doi:10.1145/1314276.1314290.
URL http://doi.acm.org/10.1145/1314276.1314290

[9] C. Ochoa, J. Silva, G. Vidal, Lightweight program specialization via Dynamic Slicing, in: Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, WCFLP '05, ACM, New York, NY, USA, 2005, pp. 1–7.

doi:10.1145/1085099.1085101.
URL http://doi.acm.org/10.1145/1085099.1085101

[10] C. Galindo, S. Pérez, J. Silva, Slicing unconditional jumps with unnecessary control dependencies, in: M. Fernández (Ed.), Logic-Based Program Synthesis and Transformation, Vol. 12561 of Lecture Notes in Computer Science (LNCS), Springer International Publishing, Cham, 2021, pp. 293–308.

[11] T. Ball, S. Horwitz, Slicing programs with arbitrary control-flow, in: Proceedings of the First International Workshop on Automated and Algorithmic Debugging, AADEBUG '93, Springer-Verlag, London, UK, UK, 1993, pp. 206–222.

[12] C. Galindo, S. Pérez, J. Silva, Exception-sensitive program slicing, Journal of Logical and Algebraic Methods in Programming 130 (2023) 100832. doi:https://doi.org/10.1016/j.jlamp.2022.100832.
URL https://www.sciencedirect.com/science/article/pii/S2352220822000852

[13] M. Allen, S. Horwitz, Slicing Java programs that throw and catch exceptions, SIGPLAN Not. 38 (10) (2003) 44–54.

[14] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, T. Teitelbaum, Program slicing for VHDL, International Journal on Software Tools for Technology Transfer 4 (1) (2002) 125–137. doi:https://doi.org/10.1007/s100090100069.

[15] D. Binkley, Precise executable interprocedural slices, ACM Letters on Programming Languages and Systems 2 (1-4) (1993) 31–45.

[16] J. Krinke, Context-sensitive slicing of concurrent programs, in: Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2003, p. 178–187. doi:10.1145/940071.940096.

[17] Z. Chen, B. Xu, Slicing concurrent Java programs, SIGPLAN Not. 36 (4) (2001) 41–47. doi:10.1145/375431.375420.
URL http://doi.acm.org/10.1145/375431.375420

[18] G. Ramalingam, J. Field, F. Tip, Aggregate structure identification and its application to program analysis, in: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '99, Association for Computing Machinery, New York, NY, USA, 1999, p. 119–132. doi:10.1145/292540.292553.
URL https://doi.org/10.1145/292540.292553

[19] J. Silva, S. Tamarit, C. Tomás, System dependence graphs in sequential Erlang, in: Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012), Vol. 7212 of Lecture Notes in Computer Science (LNCS), Springer, 2012, pp. 486–500.

[20] C. Galindo, J. Krinke, S. Pérez, J. Silva, Field-sensitive program slicing, in: Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings, Vol. 13550, Springer Nature, 2022, pp. 74–90.

[21] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions Programming Languages and Systems 12 (1) (1990) 26–60.

[22] B. Korel, J. Laski, Dynamic program slicing, Information Processing Letters 29 (3) (1988) 155 – 163. doi:https://doi.org/10.1016/0020-0190(88)90054-3.
URL http://www.sciencedirect.com/science/article/pii/0020019088900543

[23] B. Korel, J. Laski, Dynamic slicing of computer programs, Journal of Systems and Software 13 (3) (1990) 187–195. doi:https://doi.org/10.1016/0164-1212(90)90094-3.

[24] D. Binkley, K. B. Gallagher, Program slicing, Advances in Computers 43 (2) (1996) 1–50. doi:10.1016/S0065-2458(08)60641-5.
URL http://dx.doi.org/10.1016/S0065-2458(08)60641-5

[25] J. Späth, K. Ali, E. Bodden, Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems, Proc. ACM Program. Lang. 3 (POPL) (Jan. 2019). doi:10.1145/3290361.
URL https://doi.org/10.1145/3290361

[26] D. Liang, M. J. Harrold, Slicing objects using system dependence graphs, in: Proceedings of the International Conference on Software Maintenance, ICSM '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 358–367.
URL http://dl.acm.org/citation.cfm?id=850947.853342

[27] N. Walkinshaw, M. Roper, M. Wood, The Java system dependence graph, in: Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation, 2003, pp. 55–64.

[28] T. Reps, S. Horwitz, M. Sagiv, Precise interprocedural dataflow analysis via graph reachability, in: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '95, Association for Computing Machinery, New York, NY, USA, 1995, p. 49–61. doi:10.1145/199448.199462.
URL https://doi.org/10.1145/199448.199462

[29] T. Reps, S. Horwitz, M. Sagiv, G. Rosay, Speeding up slicing, SIGSOFT Softw. Eng. Notes 19 (5) (1994) 11–20.

[30] X. Gongye, Y. Wang, Y. Wen, P. Nie, P. Lin, A simple detection and generation algorithm for simple circuits in directed graph based on depth-first traversal, Evolutionary Intelligence (April 2020). doi:10.1007/s12065-020-00416-6.

[31] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous Java performance evaluation, SIGPLAN Not. 42 (10) (2007) 57–76. doi:10.1145/1297105.1297033.
URL http://doi.acm.org/10.1145/1297105.1297033

[32] M. Sridharan, S. J. Fink, R. Bodik, Thin slicing, in: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 112–122. doi:10.1145/1250734.1250748.
URL https://doi.org/10.1145/1250734.1250748

[33] S. Litvak, N. Dor, R. Bodik, N. Rinetzky, M. Sagiv, Field-sensitive program dependence analysis, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 287–296. doi:10.1145/1882291.1882334.
URL https://doi.org/10.1145/1882291.1882334

[34] P. K. K., A. Sanyal, A. Karkare, S. Padhi, A static slicing method for functional programs and its incremental version, in: Proceedings of the 28th International Conference on Compiler Construction, CC 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 53–64. doi:10.1145/3302516.3307345.
URL https://doi.org/10.1145/3302516.3307345

[35] J. R. Lyle, Evaluating variations on program slicing for debugging (data-flow, ada), Ph.D. thesis, USA (1984).

[36] S. S. Muchnick, Advanced Compiler Design and Implementation, chapter 8.12, Morgan Kaufmann, 1997.

[37] H. Agrawal, R. A. DeMillo, E. H. Spafford, Dynamic slicing in the presence of unconstrained pointers, in: Proceedings of the symposium on Testing, Analysis, and Verification, 1991, pp. 60–73.

[38] S. S. Muchnick, Advanced Compiler Design and Implementation, chapter 12.2, Morgan Kaufmann, 1997.

[39] J. Krinke, Advanced slicing of sequential and concurrent programs, Ph.D. thesis, Universität Passau (2003).

[40] P. Anderson, T. Reps, T. Teitelbaum, Design and implementation of a fine-grained software inspection tool, IEEE Trans. Softw. Eng. 29 (8) (2003) 721–733.

[41] J. Graf, Speeding up context-, object- and field-sensitive SDG generation, in: 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, 2010, pp. 105–114. doi:10.1109/SCAM.2010.9.

[42] G. Snelting, T. Robschink, J. Krinke, Efficient path conditions in dependence graphs for software safety analysis, ACM Trans. Softw. Eng. Methodol. 15 (4) (2006) 410–457. doi:10.1145/1178625.1178628.

[43] W. Landi, B. G. Ryder, A safe approximate algorithm for interprocedural aliasing, ACM SIGPLAN Notices 27 (7) (1992)

235–248.

[44] J.-D. Choi, M. Burke, P. Carini, Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects, in: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1993, pp. 232–245.

[45] U. K. Banerjee, Dependence Analysis for Supercomputing, Vol. 60, Springer US, USA, 1988.

[46] D. E. Maydan, J. L. Hennessy, M. S. Lam, Efficient and exact data dependence analysis, in: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, 1991, p. 1–14. `doi:10.1145/113445.113447`.

[47] W. Pugh, D. Wonnacott, Eliminating false data dependences using the omega test, in: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, PLDI '92, Association for Computing Machinery, New York, NY, USA, 1992, p. 140–151. `doi:10.1145/143095.143129`.
URL `https://doi.org/10.1145/143095.143129`

[48] Z. Chen, B. Xu, Slicing object-oriented Java programs, SIGPLAN Not. 36 (4) (2001) 33–40. `doi:10.1145/375431.375418`.
URL `https://doi.org/10.1145/375431.375418`

[49] C. Galindo, S. Pérez, J. Silva, Program slicing of java programs, Journal of Logical and Algebraic Methods in Programming 130 (2023) 100826. `doi:https://doi.org/10.1016/j.jlamp.2022.100826`.
URL `https://www.sciencedirect.com/science/article/pii/S2352220822000797`

[50] S. Kumar, S. Horwitz, Better slicing of programs with jumps and switches, in: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002), Vol. 2306 of Lecture Notes in Computer Science (LNCS), Springer, 2002, pp. 96–112.

[51] D. Cheda, J. Silva, G. Vidal, Static slicing of rewrite systems, in: Proceedings of the 15th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2006), Elsevier ENTCS 177, 2007, pp. 123–136.

[52] C. M. Brown, Tool Support for Refactoring Haskell Programs, Ph.D. thesis, School of Computing, University of Kent, Canterbury, Kent, UK (2008).

[53] M. Tóth, I. Bozó, Z. Horváth, L. Lövei, M. Tejfel, T. Kozsik, Impact analysis of erlang programs using behaviour dependency graphs, in: Proceedings of the Third summer school conference on Central European functional programming school, CEFP'09, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 372–390.
URL `http://dl.acm.org/citation.cfm?id=1939128.1939139`

[54] N. F. Rodrigues, L. S. Barbosa, Component identification through program slicing, in: In Proc. of Formal Aspects of Component Software (FACS 2005). Elsevier ENTCS, Elsevier, 2005, pp. 291–304.