Field-Sensitive Program Slicing*

Carlos Galindo[1][0000−0002−3569−6218], Jens Krinke[2][0000−0003−1009−2861], Sergio Pérez[1][0000−0002−4384−7004], and Josep Silva[1][0000−0001−5096−0008]

[1] VRAIN, Universitat Politècnica de València, Valencia, Spain
{serperu,cargaji,jsilva}@vrain.upv.es
[2] CREST Centre, University College London, London, U.K.
j.krinke@ucl.ac.uk

Abstract. The granularity level of the 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 allows for accurate slicing of complex data structures. Both extensions are conservative with respect to the 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 of the slices of 11.67%/5.49% for programs without/with loops.

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

1 Introduction

The *Program Dependence Graph* (PDG) [18] represents the statements of a program as a collection of nodes; and their control and data dependencies are represented as edges. The PDG is used in *program slicing* [23], 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* [30], 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* [18].

Unfortunately, the original PDG is not able to properly handle the slicing of composite data structures. Finite composite data structures can be atomized [19]

* This work has been partially supported by grant PID2019-104735RB-C41 funded by MCIN/AEI/ 10.13039/501100011033, by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust), and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215. Carlos Galindo was partially supported by the Spanish Ministerio de Universidades under grant FPU20/03861.

and then sliced as usual, however, infinite 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 [24], which is explained in Section 2).

The rest of the paper is structured as follows: The next section demonstrates the problems in slicing composite data structures. Section 3 presents the CE-PDG and how it is used for slicing. In Section 4 we present an implementation and an empirical evaluation of the proposed technique. It is followed by a discussion of related work and the conclusions.

2 Slicing Composite Data Structures

In this section, we show the inaccuracy problems caused by the PDG when it is used to slice programs with complex data structures. It is important to remark that the problem of data structure slicing can be studied and solved at the level of the PDG (i.e., for intra-procedural programs). Because we can present the fundamental ideas and solutions of field-sensitive slicing at this level, we avoid the representation in the *System Dependence Graph* (SDG) [9] (i.e., for inter-procedural 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[3]. We also want to highlight that, for the sake of clarity, we ignore aliasing, pointers, and other programming features that are orthogonal to the problem we want to solve: slicing (recursive) data structures. The pointer analysis needs to be field-sensitive in the same way our approach is.

Example 1. Consider the fragment of Erlang code in Figure 1a, where we are interested in the values computed at variable `C` (the slicing criterion is $\langle 4, \texttt{C} \rangle$).

[3] Our implementation is already inter-procedural. However, due to lack of space, and because it is an important problem by itself, we have limited the paper to the intra-procedural version.

```
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

Fig. 1: Slicing Erlang tuples (slicing criterion underlined and blue, slice in green)

The only part of the code that can affect the values at `C` (i.e., the minimal slice) is coloured in green in Figure 1c. Nevertheless, the slice computed with the PDG (shown in Figure 1b) contains the whole program. This is a potential source of more imprecisions outside this function because it wrongly includes in the slice the parameters of function `foo` and, thus, all calls to `foo` are also included together with their arguments and the code on which they depend.

The fundamental problem in this particular example is pattern matching: 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 that `[C]` flow depends on `Z`, and in turn, `Z` flow depends on `A`. Because flow dependence is usually considered to be transitive, slicing the PDG wrongly infers that `C` depends on `A` (`A` is in the slice for `C`). This problem becomes worse in presence of recursive data types. For instance, trees or objects (consider a class A with a field of type A, which produces an infinite data type) can prevent the slicer to know statically what part of the collapsed structure is needed. An interesting discussion and example about this problem can be found in [26, pp. 2–3]. In the next section we propose an extension of the PDG that solves the above problem.

3 Constrained-Edges Program Dependence Graph

This section introduces the CE-PDG, for which the key idea is to expand all those 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, and how they affect the graph traversal in the slicing process.

3.1 Extending the PDG

Figure 1b 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 is made following 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 [13,29]). 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 as much as they are 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.

Definition 1 (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 a child of the PDG node that contains the composite data structure. For instance, the structural edges of the CE-PDG in Figure 2 represent the tuples of the code in Figure 1. 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: (i) composite data structures being defined, and (ii) composite data structures being used. In Figure 1 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

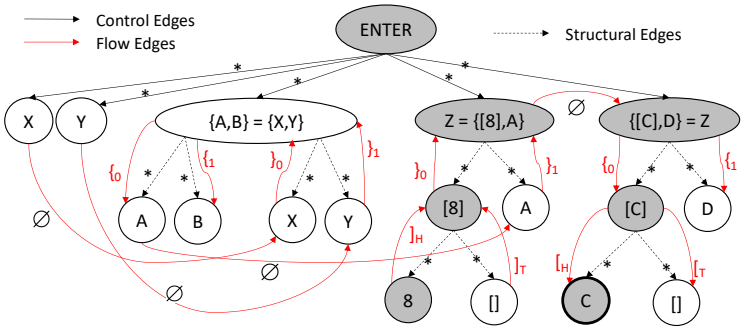


Fig. 2: CE-PDG of the code in Figure 1.

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 2). 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 2). 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 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*.

Definition 2 (Constraint). *A constraint C is a label defined as follows:*

$$C ::= \varnothing \mid * \mid Tuple \mid List \qquad Pos ::= H \mid T$$
$$Tuple ::= \{_{int} \mid \}_{int} \qquad List ::= [_{Pos} \mid]_{Pos}$$

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, meaning that 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 ("}", "]").

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. Arrays where the position is a variable imply that any position of the array may be accessed. Hence, arrays with variable indices are treated as $\{_*$ constraints, which would match a constraint $\}_x$ for any x.

Example 2. All edges in Figure 2 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 accurately slice the program in Figure 1a. In the CE-PDG (Figure 2), 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 backward the flow dependencies, we do not want the whole `Z`, but 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`. The slice computed in this way is composed of the grey nodes, and it is exactly the minimal slice in Figure 1c. Note that no structural edge is traversed during the slice in the above example. How structural edges are handled during slicing is discussed in the next section.

The CE-PDG is a generalization of the PDG because the PDG is a CE-PDG where all edges are labelled with empty constraints ($\varnothing$). 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.

3.2 Slicing the CE-PDG: Constrained traversal

In this section, we show how constraints can improve the accuracy of the slices computed with the CE-PDG. 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 slicing algorithm uses a stack to store the words while it traverses 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 is its complementary closing constraint.

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

Table 1: Processing edges' stacks. 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	$*$	$\perp$

(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.

3.3 The slicing algorithm

Algorithm 1 illustrates the process to slice the CE-PDG. It works similar to the standard algorithm [21], 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 6). In every iteration the algorithm processes one state. First, it collects all edges that target the current node (function getIncomingEdges in line 7). If the previous traversed edge is structural, we avoid traversing flow edges (lines 9–10) 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. 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 11). If the edge cannot be traversed according to Table 1 ($newStack == error$), then the reachable node is ignored (line 12). Otherwise, the node is added to the work list together with the new stack (line 13). Finally,

Algorithm 1 Intraprocedural slicing algorithm for CE-PDGs

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

```
1:  function slicingAlgorithmIntra(n_sc)
2:    slice ← ∅; processed ← ∅
3:    workList ← {⟨n_sc, ⊥, []⟩}
4:    while workList ≠ ∅ do
5:      select some state ∈ workList;
6:      ⟨node, stack, traversedEdges⟩ = state
7:      for all edge ∈ getIncomingEdges(node) do
8:        ⟨sourceNode, type, _⟩ ← edge
9:        if getLastEdgeType(traversedEdges) = structural
                ∧ type = flow then
10:         continue for all
11:       newStack ← processConstraint(stack, edge)
12:       if newStack ≠ error then
13:         workList ← workList ∪
              {⟨sourceNode, newStack, traversedEdges ++ edge⟩}
14:     processed ← processed ∪ {state}
15:     workList ← {(n, s, t) ∈ workList | (n, s, _) ∉ processed}
16:     slice ← slice ∪ {node}
17:   return slice

18: function processConstraint(stack, edge)
19:   ⟨_, _, constraint⟩ ← edge
20:   if constraint = AsteriskConstraint then return ⊥
21:   else
22:     if edge ∈ traversedEdges then
23:       if isIncreasingLoop(findLoop(traversedEdges),edge) then return ⊥
24:     if constraint = EmptyConstraint then return stack
25:     else return processAccess(stack, constraint)

26: function processAccess(stack, constraint = ⟨op, position⟩)
27:   if stack = ⊥ then
28:     if op = { ∨ op = [ then return push(constraint, stack)
29:     else return ⊥
30:   lastConstraint ← top(stack)
31:   if (op = } ∧ lastConstraint = ⟨{, position⟩)
        ∨ (op = ] ∧ lastConstraint = ⟨[, position⟩) then
32:     return pop(stack)
33:   else
34:     if op = } ∨ op = ] then return error
35:     else return push(constraint, stack)
36:   return stack
```

the state is added to a list of processed states, used to avoid the multiple evaluation of the same state, and the current node is included in the slice (lines 14–16).

Function processConstraint computes a new stack for all possible types of constraint: First, it returns an empty stack for asterisk constraints (line 20), Then, the condition in line 22 checks the existence of a loop (reaching an already traversed edge) during the slicing traversal. Function findLoop (line 23) returns the shortest suffix of the sequence of traversed edges that form the last loop, while function isIncreasingLoop (line 23), whose rationale is extensively explained in Section 3.4, consequently empties the stack when needed. If no dangerous loop is detected, the function returns the same stack for empty constraints (line 24), or it processes access constraints following Table 1 with function processAccess (line 25).

Example 3. Consider again function `foo` in the code of Figure 1a, the selected slicing criterion ($\langle$`4`, `C`$\rangle$), and its CE-PDG, shown in Figure 2. 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 ($\bot$). The evolution of the stack after traversing each flow edge is the following: $\bot \xrightarrow{[_H} [_H \xrightarrow{\{_0} [_H\{_0 \xrightarrow{\varnothing} [_H\{_0 \xrightarrow{\}_0} [_H \xrightarrow{]_H} \bot$. Due to the traversal limitations imposed by the row 5 in Table 1, node `A` is never included in the slice because the following transition is not possible: $[_H\{_0 \xrightarrow{\}_1} error$. As already noted, the resulting slice provided by Algorithm 1 is exactly the minimal slice shown in Figure 1c.

3.4 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[4] (see the programs in Figure 3, 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 3a[5]), generating an infinite number of states. Fortunately, not all cycles produce this problem:[6] To keep the discussion precise, we need to formally define when a cycle in the CE-PDG is a *loop*.

Definition 3 (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* $n > 1$ *times with an initial empty stack* ($\bot$) *following the rules of Table 1.*

There exist three kinds of loops:

(1) Loops that decrease the size of the stack in each iteration 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.

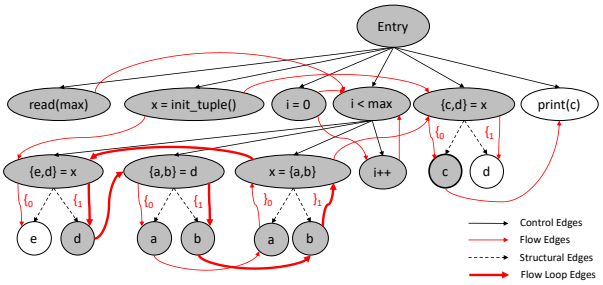
(2) Loops that keep the same stack in each iteration are also not a problem because traversing the loop multiple times does not generate new states. Again,

[4] 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).

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

[6] The interested reader has a developed example for each kind of loop, which includes their CE-PDGs, in the technical report `https://mist.dsic.upv.es/techreports/2022/06/field-sensitive-program-slicing.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,b};
7 }
8 {c,d} = x;
9 print(c);
```



(a) Increasing stack size (b) CE-PDG of code in Figure 3a

Fig. 3: Slicing flow-dependence cycles in the CE-PDG (slicing criterion underlined and blue, slice in green).

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 3a) could produce an infinite number of states because the stack grows infinitely. It is important to remark that not all cycles formed from more opening constraints than closing constraints are increasing loops. They may not even be loops (see Definition 3). Cycles that are not loops are not dangerous because the cycle's edges constraints prevent us to traverse them infinitely. One illustrative example is the code in Figure 3a where we have the flow dependence cycle $(6,x) \xleftarrow{\}_0} (6,a) \xleftarrow{\varnothing} (5,a) \xleftarrow{\{_0} (5,d) \xleftarrow{\varnothing} (4,d) \xleftarrow{\{_1} (4,x) \xleftarrow{\varnothing} (6,x)$. But this is not a loop because no matter with what stack we enter the cycle, when $\{_1$ is pushed on the stack, the cycle cannot be entered again due to the constraint $\}_0$ that does not match the top of the stack. In contrast, in the same code there exist a loop (highlighted in bold red) that can infinitely increase the stack with $\{_1$ in each iteration: $(6,x) \xleftarrow{\}_1} (6,b) \xleftarrow{\varnothing} (5,b) \xleftarrow{\{_1} (5,d) \xleftarrow{\varnothing} (4,d) \xleftarrow{\{_1} (4,x) \xleftarrow{\varnothing} (6,x)$.

We formally define a special kind of loop which is the only potentially dangerous: *increasing loop*.

Definition 4 (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.*

To define and detect the increasing loops (those that can grow the stack infinitely) we have designed the pushdown automaton (PDA) of Figure 4. The input of this automaton is the sequence of constraints that form a dependence cycle. The PDA contains two states and two different stacks (closing stack and opening stack). 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

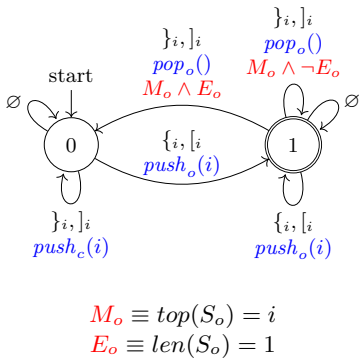


Fig. 4: Pushdown automaton to recognize increasing loops.

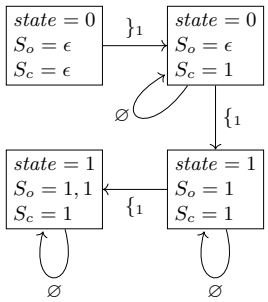


Fig. 5: States produced by the PDA in Figure 4 with the word $\}_1\varnothing\{_1\varnothing\{_1\varnothing$

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

Example 4. Consider the dependence cycle formed from lines 4, 5, and 6 of Figure 3a: $(6, x) \xleftarrow{\}_1} (6, b) \xleftarrow{\varnothing} (5, b) \xleftarrow{\{_1} (5, d) \xleftarrow{\varnothing} (4, d) \xleftarrow{\{_1} (4, x) \xleftarrow{\varnothing} (6, x)$, 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 5. 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.

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 in Figure 4 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 $reverse(S_c) \neq S_o$.

Only increasing loops can produce non-termination. For this reason, Algorithm 1 detects loops (Line 22) and checks whether they are increasing with function ProcessEdgeCircuit (Line 23). This function uses the PDA of Fig-

ure 4 to determine whether the loop is increasing and in such a case the stack is emptied, i.e., the traversal continues unconstrained.

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 worst complexity is exponential $\mathcal{O}(2^N)$ [7]. 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).

4 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 treatment 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 12186 LOC (excluding comments and empty lines). It is an open-source project and is publicly available[7].

Additionally, anyone can slice a program via a web interface[8], 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, 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[9]. 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[10], which guarantees that there is no bias in the selection of slicing criteria.

We strictly followed the methodology proposed by Georges et al. [6]. Each program's graph was built 1001 times, and the graphs were sliced 1001 times per

[7] `https://mist.dsic.upv.es/git/program-slicing/e-knife-erlang`
[8] `https://mist.dsic.upv.es/e-knife-constrained/`
[9] `https://mist.dsic.upv.es/bencher/`
[10] Each variable use or definition in all functions that contain complex data structures.

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

	Graph Generation		Slice					
Program	**PDG**	**CE-PDG**	**Function**	**#SCs**	**PDG**	**CE-PDG**	**Slowdown**	**Red. Size**
bench1A.erl	5468.10ms	5474.38ms	getLast/2	26	82.55μs	392.39μs	4.40 ± 0.50	$14.88 \pm 3.23\%$
			getNext/3	174	308.66μs	1645.44μs	4.94 ± 0.16	$13.06 \pm 1.58\%$
			getStringDate/1	11	30.18μs	93.71μs	3.22 ± 0.18	$8.67 \pm 4.07\%$
			main/1	57	1121.93μs	2869.79μs	2.59 ± 0.30	$38.76 \pm 7.22\%$
bench3A.erl	49.58ms	49.59ms	tuples/2	22	38.39μs	153.21μs	3.69 ± 0.44	$5.46 \pm 2.08\%$
bench4A.erl	79.70ms	79.76ms	main/2	31	89.80μs	376.33μs	4.23 ± 0.42	$20.79 \pm 5.47\%$
bench5A.erl	48.69ms	48.73ms	lists/2	18	60.92μs	265.87μs	3.82 ± 0.43	$6.51 \pm 2.08\%$
bench6A.erl	403.52ms	403.66ms	ft/2	34	82.59μs	333.51μs	3.60 ± 0.36	$12.25 \pm 2.72\%$
			ht/2	16	21.39μs	71.55μs	2.94 ± 0.29	$10.79 \pm 3.81\%$
bench9A.erl	199.53ms	199.71ms	main/2	18	197.94μs	458.68μs	2.25 ± 0.14	$1.38 \pm 1.07\%$
bench11A.erl	15.49ms	15.52ms	lists/2	16	43.09μs	141.87μs	3.30 ± 0.16	$6.47 \pm 2.22\%$
bench12A.erl	1661.91ms	1663.25ms	add/4	26	104.88μs	454.55μs	4.27 ± 0.49	$15.21 \pm 4.29\%$
			from_ternary/2	9	22.92μs	103.17μs	4.28 ± 0.44	$3.56 \pm 2.76\%$
			main/3	39	103.42μs	408.30μs	4.03 ± 0.51	$8.43 \pm 6.27\%$
			mul/3	21	55.05μs	261.14μs	4.57 ± 0.35	$2.74 \pm 1.31\%$
			to_ternary/2	13	71.93μs	199.70μs	3.05 ± 0.28	$1.02 \pm 1.37\%$
bench14A.erl	3841.95ms	3842.62ms	main/2	81	85.94μs	451.66μs	4.01 ± 0.40	$8.76 \pm 2.56\%$
bench15A.erl	1948.76ms	1949.37ms	main/4	71	246.97μs	609.24μs	2.94 ± 0.19	$2.31 \pm 1.73\%$
bench16A.erl	276.60ms	276.79ms	word_count/5	36	83.79μs	289.83μs	3.96 ± 0.30	$8.91 \pm 2.93\%$
bench17A.erl	63.47ms	63.60ms	mug/3	19	55.44μs	202.33μs	3.78 ± 0.18	$5.59 \pm 3.10\%$
bench18A.erl	71.38ms	71.50ms	mbe/2	19	83.69μs	278.30μs	3.73 ± 0.31	$7.38 \pm 4.71\%$
Totals and averages for set A				**757**	218.65μs	814.51μs	3.88 ± 0.10	**11.67±3.02%**
bench1B.erl	4689.59ms	4695.39ms	main/1	273	2375.91μs	52978.07μs	19.04 ± 1.48	$5.78 \pm 2.23\%$
bench2B.erl	122.07ms	122.10ms	main/2	17	100.30μs	160.02μs	2.54 ± 0.47	$0.25 \pm 0.34\%$
bench3B.erl	53.70ms	53.71ms	tuples/2	18	73.09μs	283.20μs	3.70 ± 0.42	$4.33 \pm 1.25\%$
bench4B.erl	38.34ms	38.40ms	main/2	39	136.43μs	351.29μs	2.98 ± 0.33	$11.78 \pm 3.70\%$
bench5B.erl	24.67ms	24.72ms	lists/2	11	83.64μs	316.45μs	3.83 ± 0.20	$6.88 \pm 0.89\%$
bench6B.erl	89.36ms	89.49ms	tuples/2	44	64.04μs	241.37μs	3.65 ± 0.39	$6.54 \pm 1.65\%$
bench8B.erl	144.54ms	144.67ms	main/2	42	317.21μs	19641.19μs	57.75 ± 7.30	$0.73 \pm 0.68\%$
bench9B.erl	53.57ms	53.65ms	main/2	17	305.20μs	588.48μs	2.02 ± 0.16	$1.16 \pm 0.88\%$
bench10B.erl	146.72ms	146.98ms	main/1	35	415.38μs	7368.92μs	26.06 ± 5.94	$2.23 \pm 1.17\%$
bench11B.erl	15.10ms	15.15ms	lists/2	13	69.71μs	248.10μs	3.58 ± 0.18	$8.02 \pm 2.17\%$
bench12B.erl	526.36ms	527.29ms	main/3	88	1445.05μs	7244.07μs	5.15 ± 1.32	$2.61 \pm 2.69\%$
bench13B.erl	41.00ms	41.05ms	main/0	22	212.20μs	307.64μs	1.88 ± 0.35	$0.48 \pm 0.40\%$
bench14B.erl	257.98ms	258.50ms	main/2	52	167.99μs	522.23μs	3.20 ± 0.40	$12.84 \pm 4.48\%$
bench15B.erl	376.22ms	376.62ms	main/4	73	394.71μs	770.11μs	2.39 ± 0.16	$8.78 \pm 2.86\%$
bench16B.erl	170.25ms	170.42ms	word_count/5	40	200.22μs	3490.60μs	30.73 ± 6.76	$3.70 \pm 1.53\%$
bench17B.erl	93.42ms	93.55ms	mug/3	19	248.47μs	442.49μs	1.88 ± 0.22	$4.96 \pm 2.45\%$
bench18B.erl	102.34ms	102.48ms	mbe/2	19	393.15μs	607.97μs	1.55 ± 0.15	$0.05 \pm 0.11\%$
Totals and averages for set B				**822**	1060.16μs	19742.28μs	13.43 ± 1.18	**5.49±2.16%**

criterion. To ensure real independence, the first iteration was always discarded (to avoid 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. 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):

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, in the first row, the computation of each slice is on average 4.40 times slower in the CE-PDG.
Red. Size: the average reduction in the slices sizes (with 95% error margins). 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 nor a good solution because the CE-PDG includes nodes and arcs 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 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 8.45%. This increase goes up to 38.76% in function `main/1` from `bencher1A`, as it contains complex data structures that can be efficiently sliced with the CE-PDG. The same happens in set B, but due to the analysis of loops, the slowdown is around thirteen times slower.

If we consider programs without cycles, and taking into account that this is an intra-procedural technique, the time required to compute a slice will be of at most a few hundred μs. Therefore, our technique reduces the size of the slices by $11.67 \pm 3.02\%$ at almost no cost (only a few μs). If we consider programs with cycles, the slowdown is 13.43, but since the technique has more opportunities for improvement (because, contrarily to the CE-PDG, the PDG includes the whole cycle in the slice in all cases), the reduction in the slices size is $5.49 \pm 2.16\%$. This is a very good result: for many applications, e.g., debugging, reducing the suspicious code over 11.67% with a cost of increasing the slicing time by only a few milliseconds is a good trade-off to make.

5 Related work

Transitive data dependence analysis has been extensively studied [20,27]. Less attention has received, however, the problem of field-sensitive data dependence

analysis [14,19,10,26]. The existing approaches can be classified into two groups: those that treat composite structures as a whole [15,18,17,14], and those that decompose them into small atomic data types [11,1,3,16,19,12,2,8]. The later 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 [12,2,8] which limits the accuracy. Moreover, atomization cannot deal with recursive data structures. Other important approaches for field-sensitive data dependence analysis of this kind are [10,26,14]. Litvak et al. [14] proposed a field-sensitive program dependence analysis that identifies dependencies by computing the memory ranges written/read by definitions/uses. Späth et al. [26] proposed the use of pushdown systems to encode and solve field accesses and uses. Snelting et al. [25] 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.

Severals works have tried to adapt the PDG for functional languages dealing with tuple structures in the process [5,4,28,10]. Some of them with a high abstraction level [22], and other ones with a low granularity level. Silva et al. [24] 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 1). 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.

6 Conclusion

To address the imprecision of PDG-based slicing of composite data structures, 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. An evaluation of our approach shows a slowdown of 3.88/13.43 and a reduction of the slices of 11.67%/5.49% for programs without/with cycles.

References

1. H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the symposium on Testing, Analysis, and Verification*, pages 60–73, 1991.
2. P. Anderson, T. Reps, and T. Teitelbaum. Design and implementation of a fine-grained software inspection tool. *IEEE Trans. Softw. Eng.*, 29(8):721–733, August 2003.
3. D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43(2):1–50, April 1996.
4. C. M. Brown. *Tool Support for Refactoring Haskell Programs.* PhD thesis, School of Computing, University of Kent, Canterbury, Kent, UK, 2008.
5. D. Cheda, J. Silva, and G. Vidal. Static slicing of rewrite systems. In *Proceedings of the 15th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2006)*, pages 123–136. Elsevier ENTCS 177, 2007.
6. A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, October 2007.
7. X. Gongye, Y. Wang, Y. Wen, P. Nie, and P. Lin. A simple detection and generation algorithm for simple circuits in directed graph based on depth-first traversal. *Evolutionary Intelligence*, April 2020.
8. J. Graf. Speeding up context-, object- and field-sensitive SDG generation. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pages 105–114, 2010.
9. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions Programming Languages and Systems*, 12(1):26–60, 1990.
10. P. K. K., A. Sanyal, A. Karkare, and 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, page 53–64, New York, NY, USA, 2019. Association for Computing Machinery.
11. B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.
12. J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs.* PhD thesis, Universität Passau, 2003.
13. D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 358–367, Washington, DC, USA, 1998. IEEE Computer Society.
14. S. Litvak, N. Dor, R. Bodik, N. Rinetzky, and M. Sagiv. Field-sensitive program dependence analysis. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 287–296, New York, NY, USA, 2010. Association for Computing Machinery.
15. J. R. Lyle. *Evaluating Variations on Program Slicing for Debugging (Data-Flow, Ada).* PhD thesis, USA, 1984.
16. S. S. Muchnick. *Advanced Compiler Design and Implementation, chapter 12.2.* Morgan Kaufmann, 1997.
17. S. S. Muchnick. *Advanced Compiler Design and Implementation, chapter 8.12.* Morgan Kaufmann, 1997.
18. K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *SIGSOFT Software Engineering Notes*, 9(3):177–184, 1984.

19. G. Ramalingam, J. Field, and 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, page 119–132, New York, NY, USA, 1999. Association for Computing Machinery.
20. T. Reps, S. Horwitz, and 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, page 49–61, New York, NY, USA, 1995. Association for Computing Machinery.
21. T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. *SIGSOFT Softw. Eng. Notes*, 19(5):11–20, December 1994.
22. N. F. Rodrigues and L. S. Barbosa. Component identification through program slicing. In *In Proc. of Formal Aspects of Component Software (FACS 2005). Elsevier ENTCS*, pages 291–304. Elsevier, 2005.
23. J. Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3), June 2012.
24. J. Silva, S. Tamarit, and C. Tomás. System dependence graphs in sequential Erlang. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*, volume 7212 of *Lecture Notes in Computer Science (LNCS)*, pages 486–500. Springer, 2012.
25. G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, Oct. 2006.
26. J. Späth, K. Ali, and E. Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proc. ACM Program. Lang.*, 3(POPL), Jan. 2019.
27. M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 112–122, New York, NY, USA, 2007. Association for Computing Machinery.
28. M. Tóth, I. Bozó, Z. Horváth, L. Lövei, M. Tejfel, and 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, pages 372–390, Berlin, Heidelberg, 2010. Springer-Verlag.
29. N. Walkinshaw, M. Roper, and M. Wood. The Java system dependence graph. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 55–64, 2003.
30. M. Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering (ICSE '81)*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.