

Handling Cyclic Reinforcement of Lattice Values in Incremental Dependency-driven Static Analysis

Jens Van der Plas
Vrije Universiteit Brussel, Belgium
jens.van.der.plas@vub.be

Quentin Stiévenart
Université du Québec à Montréal, Canada
stievenart.quentin@uqam.ca

Coen De Roover
Vrije Universiteit Brussel, Belgium
coen.de.roover@vub.be

Abstract—Nowadays, many developers heavily rely on feedback from bug-detection tools to help ensure the quality of the code they produce. Such tools are underlain by static analysis. It is, however, critical for analysis results to be produced fast. To this end, incremental static analysis can be used. Upon a program change, an incremental analysis updates the previous results rather than recomputing the results from scratch.

Incremental static analyses may suffer from *cyclic reinforcement of lattice values*, where the computation of some values within the analysis relies on the values themselves, due to the abstractions made by the analysis. This can cause the incremental analysis to produce less precise results, reducing its usability.

In this work, we provide a solution to cyclic reinforcement of lattice values for incremental dependency-driven analyses. We compute the information flow within the analysis and show how this information flow can be used to detect cyclic reinforcements. We establish a criterion to detect when a cyclic reinforcement contains outdated information that needs to be removed, and show how precision can be regained. Our results show that using our method, an incremental analysis produces results matching a from-scratch analysis for all but one benchmark program, at the cost of a performance hit in some cases.

Index Terms—Static Program Analysis, Incremental Analysis, Modular Analysis

I. INTRODUCTION

Static program analyses compute behavioural program properties, e.g., the procedures that may be called at a call site, without executing the analysed program. They underlie many tools used to ensure code quality and correctness, such as detectors of bugs and vulnerabilities. These tools must fit seamlessly into the development cycle and produce results sufficiently fast, regardless of whether they are part of the IDE or of a continuous integration pipeline. Late analysis feedback, for example, may require developers to make expensive mental context switches [1], [2]. Yet, running a complex analysis on large programs may take longer than desired. Running analyses in agile contexts where feedback is required about successive program changes exacerbates the need for speed.

To speed up static analysis, incremental designs have been introduced. During the first run, an *initial analysis* analyses the entire program. Upon every subsequent program update, an *incremental update* of the results is performed using the program changes, with the goal to produce results faster than a *full reanalysis* of the updated program. As the impact of program changes is typically small [3]–[6], there is a large

opportunity for saving time. As a result, incremental analyses have already been proposed for various applications, including data-flow analysis [7], [8], reachability analysis [9], [10], points-to analysis [11], and race detection [12].

Van der Plas et al. [13] recently introduced a systematic derivation method for incremental static analyses that leverages computational dependencies within the analysed program, as reified by dependency-driven analyses [14]. Dependency-driven analyses (DDA) divide the behaviour of a program into parts called *components*, e.g., function calls or spawned threads. Every component is analysed in isolation while its dependencies on the global analysis state are monitored, and components are reanalysed when a part of the global state on which they depend is updated. This mechanism is leveraged to obtain incrementality, using the dependencies to propagate the effect of program changes. This way, the impact of the changes is limited to the affected components and the analysis scales in size of the change impact, which is the ideal case.

To our knowledge, the method of Van der Plas et al. [13] is the only systematic derivation method for incremental analyses. It has a broad applicability due to its reliance on the computational dependencies that are available in every program, and its application can yield significant speed-ups over a full reanalysis. Yet, an incremental update may produce less precise results than a full reanalysis in the presence of *cyclic reinforcement of lattice values* [13], [15]–[17], a situation in a lattice-based analysis where, due to the abstractions made by the analysis, a cyclic flow of information exists within the analysis. While the precision loss is minor overall, this may not be acceptable when precision is essential.

Cyclic reinforcement of lattice values occurs when an incremental analysis has lattice-based aggregations that are cyclic and where the incremental analysis does not remove all information that may be affected by the changes upfront. Here, the computation of a reinforced lattice value is influenced by the value itself, for which precision may be lost. Cyclic reinforcements have been studied for recursive, lattice-based aggregations in the context of Datalog-based static analysis [15], [16], but a general solution to this problem is currently lacking. Therefore, in this work, we present a solution for incremental analyses based on reified computational dependencies to handle such cyclic reinforcements. We extend and improve upon the method of Van der Plas et al. [13] to enhance the precision of the resulting incremental analyses. Concretely,

we make the following contributions:

- We show how the presence of cyclic reinforcements can be determined using the information flow within the analysis.
- We demonstrate how the information flow can also be used to determine when a cycle contains outdated information and how this information can be removed. In line with the original work, we formulate our extensions as general as possible to preserve its broad applicability.
- We evaluate our extension in terms of the soundness, precision and running time of an incremental update.

II. BACKGROUND MATERIAL

This section presents background material on dependency-driven static analysis, on the systematic method of Van der Plas et al. [13] to incrementalise such analyses, and on cyclic reinforcement of lattice values.

A. Dependency-driven Static Analysis

Dependency-driven static analysis (DDA) is a recent formulation of static analysis, computing control flow and data flow information, that decomposes a program into parts called *modules*, such as function definitions or classes [14]. Modules may have several run-time instantiations that may be discerned by the analysis (e.g., a function may be called from multiple call sites), each reified as a *component*. The analysis of the program is decomposed into the analysis of its components.

A *compositional analysis* analyses each component in isolation and then combines all partial results. This, however, ignores the fact that components may not be fully independent. For example, functions may call each other, passing arguments and return values. A *DDA*, in contrast, *reifies* these dependencies as *effects* and uses these to drive its fixed-point computation [14].¹ An *inter-component* analysis (CA_{inter}) schedules components for analysis based on the observed effects, and an *intra-component* analysis (CA_{intra}) analyses individual components, emitting effects according to the dependencies of the analysed component. Using these effects, components are scheduled for reanalysis whenever a part of the analysis state on which they depend is updated. It is possible to use different module granularities, e.g., to create function-modular [14] or thread-modular [18], [19] analyses. DDA works well with global-store widening [20], where there is a single global value store σ shared by the analyses of all components, which abstractly represents the heap of the program within the analysis and maps abstract addresses to abstract values. We now describe the DDA algorithm in detail, and then give a detailed example to clarify the explanation. For simplicity, in this text, we will always assume the analysis to be function-modular and for global-store widening to be used.

1) *The Dependency-driven Analysis Algorithm*: The inter-component analysis CA_{inter} , shown in Algorithm 1, drives the fixed-point computation using a worklist of components that need to be (re-)analysed. Initially, the worklist contains MAIN,

```

1 WL := {MAIN}; // The worklist.
2 V := ∅; // The set of visited components.
3 D := λr.∅; // Encountered dependencies (read effects).
4 σ := λa.⊥; // Global value store (represents the heap).
5 while WL ≠ ∅ do
6   α ∈ WL;
7   WL := WL \ {α};
8   (C', R', U', σ') = intra(α, σ); // CAintra.
9   σ := σ';
10  V := V ∪ {α};
11  WL := WL ∪ (C' \ V);
12  foreach r ∈ R' do D := D[r ↦ D(r) ∪ {α}];
13  foreach u ∈ U' do WL := WL ∪ D(u);
14 return (σ, V, D);

```

Algorithm 1: The inter-component analysis CA_{inter} of a DDA.

a component representing the entry point of the program (Line 1). A visited set V stores all analysed components, and a dependency map D stores all dependencies, mapping every address in the global store to the components depending on the value at that address. The global store, σ , initially maps all addresses to bottom, \perp , indicating no information (Line 4).

As long as the worklist is not empty (Line 5), a component α is taken from the worklist (Lines 6–7) and analysed by the CA_{intra} (Line 8). We treat the CA_{intra} as a black box but it must return a set of components encountered during the analysis of α (C), a set of read effects containing all addresses read in σ (R), a set of write effects containing all addresses in σ whose value changed during the analysis of α (U), and the updated global store (σ'). The CA_{inter} then stores σ' and adds α to the visited set (Lines 9–10). Based on the collected effects, the CA_{inter} then schedules components for analysis that are encountered but have not been analysed before (new components, Line 11), and that depend on a value at an address in the store that has been updated (Line 13). Note that the dependencies are updated (Line 12) before adding dependent components to the worklist to ensure that all dependencies are taken into account (e.g., to support for recursive functions). Adding components to the worklist when the values in the store on which they depend are updated, propagates the updated values to the dependent components. The analysis terminates when the worklist is empty, i.e., when a fixed-point has been reached.

2) *Example*: We exemplify the above algorithm using a DDA of the program in Listing 1. Values are abstracted using a product lattice, where we abstract a boolean as either **true** ($\#t$), **false** ($\#f$), or **Bool**, a function as a singleton set containing the closure, a pointer as a singleton set containing the corresponding store address, and any other value as its type. We use no context sensitivity, meaning that every function definition in the program corresponds to one component.

```

(define var 0)
(define (fun) (set! var -1))
(fun)

```

Listing 1: Example program.

A DDA of the program in Listing 1 works as follows:

- Initially, the worklist contains MAIN, representing the entry point. Its analysis proceeds as follows:

- 1) Binding the variable `var` updates its abstract value in the store from \perp to **Int**. Therefore, a *write effect* is generated.

¹In the literature, DDA are often referred to as *effect-driven* analyses. We find the former term more descriptive regarding the nature of the analysis.

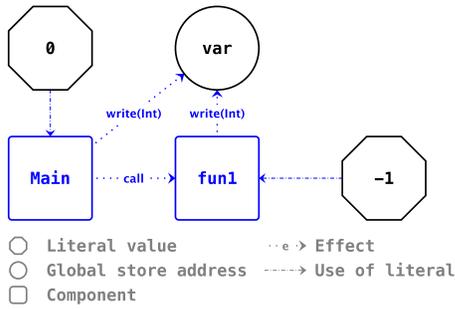


Fig. 1: Visualisation of the DDA of the program in Listing 1, omitting the return values of the components and the address `fun`, and including the use of literals.

- 2) The variable `fun` becomes bound to a closure which is written to the store. Again, a write effect is generated.
 - 3) The call to `fun` is encountered. First, the variable `fun` is read, returning a closure and generating a *read effect*. As a function call denotes the module boundary, the analysis does not step into the call but creates a *call effect* for the corresponding component, `FUN`. Then, the return value of `FUN` is retrieved from σ , which is stored at a dedicated address. As the address is read, a *read effect* is generated by the analysis. This indicates the dependency of `MAIN` on the value stored at the given address in σ .
 - 4) The return value of `MAIN` is written to σ . The generated effects are then processed. For every call effect, the corresponding component is added to the worklist if the component has not been analysed before. Thus, `FUN` is added to the worklist. Then, all read effects are stored in D ; they correspond to the dependencies of `MAIN`. Finally, for every write effect, the dependent components are added to the worklist. Here, the write effect on `fun` causes `MAIN` to be added to the worklist again, as it generated a read effect for this variable.
- The worklist now contains `FUN` and `MAIN`. The order in which the components are analysed does not influence the result [14]; we assume that `FUN` is analysed next:
 - 1) When the `set!` expression is analysed, the new value for `var` is written to σ . However, as the variable's value in σ remains `Int` (given the abstract domain), no write effect is generated as no components need to be reanalysed.
 - 2) Next, the return value of `FUN` is written. As the value of this address now becomes `Int` rather than \perp (assuming that the `set!` expression evaluates to the value written), a write effect is generated.
 - 3) The effects are handled; `MAIN` is added to the worklist.
 - A final reanalysis of `MAIN` concludes the analysis.

The result of the analysis contains the global value store σ , the set of analysed components, and the collected dependencies. This can be used by client analyses requiring control flow and data flow information. Fig. 1 depicts the components and the effects generated during the analysis, together with the use of literal values in the program.

B. Incremental Dependency-driven Static Analysis

Van der Plas et al. [13] propose a general method to incrementalise DDA, as well as other analyses that can be formulated as such. The incrementalisation method does not rely on the language of the analysed program, the analysis performed, nor on the abstract domain, or sensitivity. The core requirement of the method is that the CA_{intra} computes dependencies.

1) *Monotonic Baseline:* Van der Plas et al. [21] propose a foundational incrementalisation method for DDA, consisting of two steps. First, a change-impact analysis finds all components that are *directly affected* by a change, i.e., all components where (part of) the program text was changed. These components are then added to the worklist. Second, the fixed-point computation is restarted. The directly affected components will thus be reanalysed, and – due to the dependency-driven nature of the analysis – so will all other components that are dependent on the changes. Unaffected components are thus not reanalysed and the impact of the changes is restricted to the affected part of the analysis results. The analysis scales in size of the *change impact*, which is the ideal situation.

Consider e.g., the program in Listing 2. Without loss of generality, *change expressions* are used to encode an update to the program, where the first argument of the change represents the expression in the initial program, and the second argument contains the expression of the updated program. The program is thus updated so that `var` initially is "waiting" instead of 0 and so that `fun` no longer sets `var` to -1 but to "ok" instead. The incremental analysis of Van der Plas et al. [21] handles this change as follows. First, all directly affected components are added to the worklist. In this case, the function `fun` is updated. In addition, the text of the program's entry point also got updated. Thus, the corresponding components to be added to the worklist are `MAIN` and `FUN`. Then, the fixed-point computation is restarted. We assume `FUN` is analysed first. This causes the value of `var` to be updated in σ , meaning that a write effect is generated. Here, the dependency-driven nature of the analysis causes the dependent components, in this case `FUN` itself, to be added to the worklist. The analysis of `FUN` now generates a write effect when its return value is updated, so that `MAIN` is added to the worklist again.

```
(define var (<change> 0 "waiting"))
(define (fun) (set! var (<change> -1 "ok")))
(fun)
```

Listing 2: Changes made to the program of Listing 1. `var` is initialised to "waiting" instead of to 0 and `fun` sets `var` to "ok" instead of to -1.

2) *Result Invalidation:* The incremental analysis of Van der Plas et al. [21] has no means to invalidate outdated parts of the result, i.e., all updates to the results are monotonic. As a result, the result of an incremental update is imprecise as the analysis results reflect the behaviour of both program versions. In the example, `var` thus does not become `String` but `{Int, String}`; the outdated information remains in the result.

To mitigate this precision loss, later work [13] proposes three invalidation strategies to remove the outdated parts of the result, allowing outdated components, dependencies, and

writes to the store σ to be removed. The strategies are applied within the CA_{inter} , meaning that the CA_{intra} remains untouched. This preserves the generality of the method, as the requirements for the CA_{intra} remain minimal.

As explained in Section II-A, the CA_{intra} generates effects that correspond to the dependencies of the analysed component. After every CA_{intra} , the CA_{inter} uses the generated effects and store accesses to invalidate outdated parts of the results, by comparing them to the effects and store accesses registered during the previous analysis of the component. This way, for example, the store of the analysis can be made more precise by removing outdated writes of components. However, as the CA_{intra} remains unchanged, CA_{intra} itself remains monotonic.

Consider again the program in Listing 2 to see how the precision can be improved. Due to the changes, MAIN and FUN need to be reanalysed; both write to `var`. Suppose MAIN is reanalysed first. It now writes **String** to `var` instead of **Int**, a non-monotonic change. This updated write is now joined with the previous write of FUN (**Int**, which has not been updated yet), and `var` becomes **{Int, String}**. When FUN is reanalysed, its updated write (**String**) is joined with the updated write of MAIN, and the value of `var` becomes **String**.

However, the techniques cannot regain precision in the parts of the result affected by cyclic reinforcements, which is discussed next. We will henceforth refer to the incrementalisation method of Van der Plas et al. [13] as the MODINC approach.

C. Cyclic Reinforcement of Lattice Values

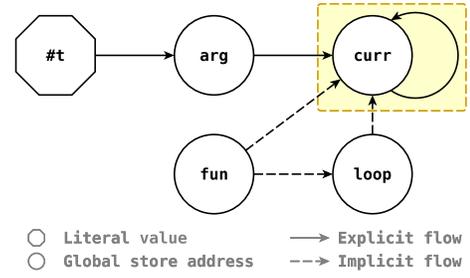
We now describe *cyclic reinforcement of lattice values*. We first repeat some general terminology on information flow.

1) *Information Flow*: We define *information flow* using the general definitions of Denning and Denning [22], [23]:

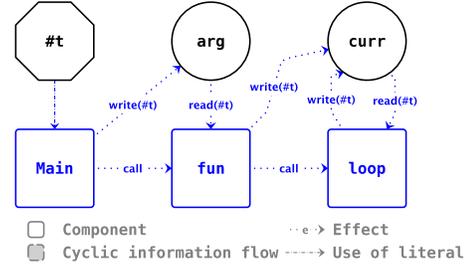
*Information flows from an object x to an object y whenever information stored in x is transferred to, or used to derive information transferred to, object y . An **explicit flow** from x to y occurs whenever the operations generating it are independent of the value of x . An **implicit flow** from x to y occurs whenever a statement specifies a flow from some arbitrary z to y , but the execution depends on the value of x .*

Explicit flows correspond to a *data dependence* and implicit information flow correspond to a *control dependence* [24]. For example, with $y := x + v$, there is an explicit information flow from x to y and from v to y . In contrast, when a piece of code is conditionally executed, there is a flow from the condition to the objects assigned in the branches. For example, if y is assigned to z given a condition depending on the value of x , then there is an implicit flow from x to y .

2) *Cyclic Reinforcements*: Cyclic reinforcement of lattice values is a phenomenon occurring in static analysis when the lattice value stored at a certain address in σ influences itself [13], [15]–[17]. In this case, the value *reinforces* itself as it has become a source for its own value. If the analysis is not aware, it will never be able to refine the value(s) within the cycle due to the monotonicity in the analysis (a value joined together with a value influenced by itself can never be refined).



(a) Information flow. Cyclic flows shown in yellow boxes.



(b) Components and effects, omitting return values, the addresses `fun` and `loop`, and their effects.

Fig. 2: Visualisation of the analysis of the program shown in Listing 3.

The occurrence of these cyclic reinforcements heavily depends on the abstractions within the analysis.

Consider the program in Listing 3. It defines an unary function `fun` which defines a new internal function `loop`. If the argument to `loop` is `#f`, the loop stops and returns the string "stop". Otherwise, the loop continues indefinitely. The function `fun` is called with `#t`.

```
(letrec ((fun (lambda (arg)
  (letrec ((loop (lambda (curr)
    (if curr
      (loop curr)
      "stop"))))
    (loop arg))))))
(fun #t))
```

Listing 3: Program causing cyclic reinforcements within the analysis.

Fig. 2a shows the information flow in (the analysis of) the program. An explicit information flow goes from the literal `#t` to the variable `arg` and from there to `curr`. Importantly, due to the recursive call in `loop`, there is also an explicit flow from `curr` to itself. This has been marked in yellow and it is here that a cyclic reinforcement occurs. (In general, cyclic reinforcements can span any number of addresses and can include both explicit and implicit flows.)

To show the troublesome nature of these cycles, let's assume the program in Listing 3 is updated so that `fun` is no longer called with `#t` but with `#f` instead. The MODINC approach [13] handles this change as follows. MAIN is directly affected so the incremental update starts by reanalysing it. The contribution of MAIN to `arg` becomes `#f`, and using write invalidation, the value is updated precisely in σ . This then triggers the reanalysis of FUN, whose contribution to `curr` becomes `#f` as well (see Fig. 3). Since the contribution of FUN is updated non-monotonically, write invalidation joins the

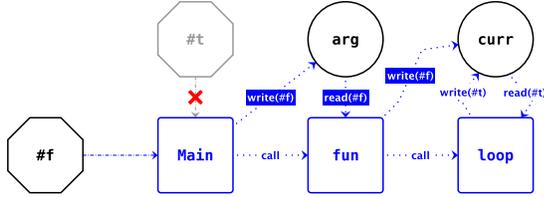


Fig. 3: Incremental update for the program in Listing 3, before the reanalysis of LOOP. Again, return values and the addresses fun and loop have been omitted.

updated contribution of FUN, $\#f$, to the contribution of LOOP, $\#t$, and the value of `curr` becomes **Bool**.

Yet, it is clear that in the updated program, `curr` is always $\#f$. Here, the incremental update has lost precision as a full reanalysis of the updated program would infer correctly that `curr` now is $\#f$. To see what causes this precision loss, consider the contribution of LOOP, which is $\#t$ when the contribution of FUN is updated. Importantly, this contribution is $\#t$ because, when LOOP was first analysed, `curr` was true. The value of `curr` is thus influencing its own computation, meaning that the old value of `curr` influences the computation of the new one, for which precision is lost. It is exactly this phenomenon that is called *cyclic reinforcement of lattice values* [13], [15]–[17], and for which we provide a solution.

III. DETECTING CYCLIC REINFORCEMENTS

We now describe how cyclic reinforcements can be detected within an analysis. In Section IV, we then explain how lattice values within such cycles can be made precise once they have been detected. Our method is designed so that only minor modifications to the underlying analysis are required. Our work thus follows the rationale of the MODINC approach [13], so that it can be seen as a precision-improving extension to the MODINC approach.

To detect cyclic reinforcements, the analysis needs to identify how values flow between addresses in σ . This way, the analysis can infer whether the computation of a value is influenced by itself, either directly or indirectly. This thus boils down to an information-flow problem, where the information flow within the analysis needs to be computed and checked for cycles; Section IV explains how precision can be regained in the presence of these cycles. We now first present a method to detect cyclic reinforcements that only requires minor modifications to the incrementalised analysis.

In a DDA, all values are stored in σ , which is thus the enabler for cyclic reinforcements. Thus, to detect such cyclic reinforcements, flows of information *between different addresses in σ* need to be computed and checked for cycles. We first discuss the inference of the information flow and then explain how it can be used to detect cyclic reinforcements.

A. Information-flow Inference

To infer the information flow within the analysis, we only require minor changes to the analysis itself. Our method relies on an information-flow graph (IG) to determine whether cyclic reinforcements are present within the analysis. The objective

is to infer the information flow between addresses in σ , the location where lattice values are stored. The nodes in the IG then correspond to the addresses in σ and the edges indicate how information flows between these addresses.

1) *Explicit Information Flow*: To track how information flows between the addresses in σ , we annotate every value with labels corresponding to the addresses in the store that the value is influenced by. This happens as follows:

- When an address a is looked up in σ , the resulting value v is labelled with that address, $v_{\{a\}}$. This indicates that the value v depends on the (value stored at) address a .
- When a value v_l is written to an address a in σ , the labels l are first stripped of the value and then the write happens as normal. This avoids labels in σ , so that components are not reanalysed when only the flow information is updated. The information represented by the labels l is stored separately and represents the edges in the IG, it indicates that the value stored at a has been influenced by all addresses in l .
- The result of a join depends on all addresses either of its arguments are depending on. Thus, when two values are joined together, so are their labels: $v_{l_1} \sqcup w_{l_2} \equiv (v \sqcup w)_{l_1 \cup l_2}$
- The application of primitive functions within the analysis follows the same reasoning as for the join operator: the result is dependent on all addresses depended on by any of its arguments: $f(v_{l_1}, w_{l_2}, \dots, x_{l_n}) \equiv f(v, w, \dots, x)_{\cup_{i=1}^n l_i}$

For explicit information flow, the labels with which values are annotated are thus propagated through the operations. When values are written to the store, the labels are stored separately in a map dfR that stores, per address, all addresses influencing the given address. dfR thus stores the *reverse* information flow data. Importantly, this data is also stored on a per-component basis, so that the information-flow data inferred by a component can be removed upon its reanalysis, allowing outdated information-flow data to be removed as well.

There is, however, one other aspect of explicit data flow: the use of *literal values* within the program. So far, we have described how values in σ are tracked as the analysis performs operations on them. However, these values originate from literals that are present in the program text. Literals may be used conditionally or not at all, and their use may change when the program gets updated. To this end, the analysis also needs to track their use. Therefore, whenever a literal value c is evaluated by the analysis, the resulting value is labelled with a specific label l corresponding to the literal expression in the program text, resulting in a value $c_{\{l\}}$. This allows to track the influence of literal values, and will enable the incremental analysis to handle situations in which the use of a literal value changes within a reinforcing cycle (see Section IV-B).

2) *Implicit Information Flow*: Implicit information flows arise when the execution of a piece of code is conditional. In this case, there is a flow between the condition and the code executed conditionally. Due to nested conditions and conditional function applications, there may be many implicit information flows. To detect them, the analysis keeps track of an *implicit flow context* (IFC) that contains all labels that have influenced the current control flow. As the number of

implicit flows may be high, we use extra *flow nodes* in the information-flow graph (IG) to reduce the number of edges.

Implicit information flows are handled as follows:

- When a value v_l is written to an address a in a given flow context i , the labels in i are stored together with l . After all, the labels in i also influence the value that is written to a , but implicitly by determining the control flow.
- When a conditional expression is evaluated, the labels of the predicate value are extracted and added to the IFC during the analysis of the branches. These labels are also added to the labels of the result value of the conditional.
- When a function is called, its labels are extracted and added to the IFC. In addition, its labels also need to be added to the return value of the function. For a function-modular analysis, however, function calls denote the component boundary. As components are analysed separately, the IFC must be remembered. Therefore, for every component, the IFCs of its calls are stored. These are used to compute the inter-component implicit information flow (see Section IV-A). As the return values of components are written to σ during the analysis of the respective components, it is not needed to add the flows to the return value explicitly, as this already happens when the return address is written.
- When a literal value is read and assigned a label l , the labels in the IFC, i , are read and stored in dfR . This way, the analysis keeps track of the fact that the literal value was read conditionally, depending on the values of i .

a) Optimisation: The number of implicit flows may be high, given that all implicit flow labels need to be taken into account upon every write to σ . To reduce the number of edges in the IG, we add extra nodes in the graph for function calls and conditionals. (If another component granularity is used, the optimisation can still apply to that granularity.)

During the CA_{intra} , all labels in the IFC must be taken into account upon every write. In the information-flow graph, this leads to a number of edges that is the product of the number of labels in the IFC and the number of written addresses. Adding an extra node to the graph allows the analysis to reduce the number of edges to the sum of both values. To this end, whenever a function call takes place in a non-empty IFC, the analysis creates a special *flow label* for the called component α . The analysis then records an information flow from the labels in the IFC to the label of α , and records that label as the IFC for the call. Following a similar reasoning, the analysis inserts a special label when a conditional is analysed. This only happens when the size of the IFC in the branches is larger than one, to avoid adding an extra nodes when possibly only few edges can be saved. As an example, the IG with flow nodes for the program in Listing 3 is shown in Figure 4. As the IFCs have size 1 due to the brevity of the example, the optimisation does not lead to a smaller graph in this case.

B. Detection of Reinforcing Cycles

Cyclic reinforcements can be detected by finding the *strongly-connected components* (SCCs) in the IG constructed

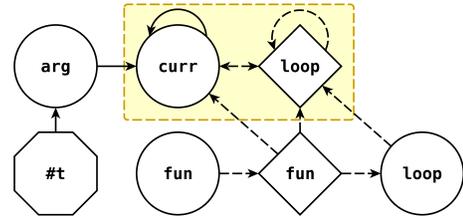


Fig. 4: IG produced for the program in Listing 3. Flow nodes are represented as diamonds, and no addresses are omitted. As `fun` and `loop` are called in non-empty IFCs, flow nodes are created.

as described in the previous section.² We refer to these SCCs as SCAs, or *strongly-connected addresses*, as the vertices in the IG are exactly the addresses in the store (supplemented with the literal nodes and the added flow nodes as explained in Section III-A2). When all the SCAs need to be computed initially, at the beginning of the incremental update (see Section IV), we use Tarjan’s algorithm for computing strongly-connected components in directed graphs [25]. Afterwards we can update the SCAs incrementally, as explained next.

1) Incremental SCC Computation: The IG is updated incrementally as the incremental update progresses. To efficiently handle edge and node additions or removals, we developed an algorithm to incrementally update SCCs. In spirit, our algorithm follows an algorithm from Van Es et al. [26], yet their algorithm assumes nodes are SCCs by themselves. Our algorithm works as follows. First, edge removals are handled. Only removed edges that are internal to an SCC can affect the SCCs in the graph. To this end, we compute the set of SCCs in which an internal edge is removed, and apply Tarjan’s algorithm locally on the nodes of the SCC to see whether it still exists, whether it’s dissolved, or whether new sub-SCCs have arisen. Second, edge additions are handled. Only added edges that are not internal to an SCC can affect the SCCs in the graph. For each such edge, a backward search from the target of the edge to its source is performed. If one or more paths are found, a new SCC is created, containing the paths found (which include the source and target nodes), as well as all SCCs containing a node that is part of a found path.

IV. PRECISION RECOVERY OF CYCLIC REINFORCEMENTS

We now describe how the information-flow inference, discussed in Section III-A, and the SCA computation, discussed in Section III-B, are used to restore the precision that may be lost by an incremental update in the presence of cyclic reinforcements. We first describe when strongly-connected addresses (SCAs) are computed, and then how they are refined.

At the start of the incremental update, all SCAs within the initial results are computed and stored. Then, the incremental update continues following the MODINC approach [13]. After every CA_{intra} , the analysis now computes whether a cyclic reinforcement is present and may cause a precision loss. In this case, the analysis needs to recover precision; we say that

²In the remainder of this section, we will omit the fact that the information flow is stored in reverse, as the direction is irrelevant when computing cycles, as long as all edges in the graph are reversed.

the SCA is *refined*. Our method to regain precision works by executing the following steps after every CA_{intra} :

- 1) The entire information-flow data is computed. This is needed as the inter-component implicit flows have not yet been computed, and these flows are thus missing in the IG.
- 2) The SCAs are computed using the data computed in step 1.
- 3) For every SCA, a *refinement condition* is checked, indicating whether the SCA may contain outdated information.
- 4) For every SCA for which the condition holds, the values at the addresses of the SCA are *refined*, i.e., they are lowered in the lattice to remove any outdated information.

These steps are shown in Algorithm 2. We now discuss steps 1, 3, and 4 in more detail. Step 2 was discussed in Section III-B.

A. Computation of Full Information-flow Data

The explicit and implicit information flow is computed during the analysis. As this is stored on a per-component basis, first, all information needs to be aggregated. Second, when a component was called, the implicit flows in which the calls were made were stored separately. Moreover, these flows were not taken into account *during* the analysis of the called components, avoiding the need to reanalyse components when only the implicit flows are updated. To compute the complete information flow, the implicit flows must be propagated across component calls to account for the inter-component implicit information flow, and be taken into account for the writes of the components. To this end, the analysis can make use of the flow nodes from Section III-A2a as follows:

- 1) For every component that is called in a non-empty implicit flow context (IFC), add an information-flow edge from its flow node to the flow nodes of all its callees as these calls are also made in the IFC of the calling component. Then also propagate over calls made by the callees, which were also made in this IFC.
- 2) For every component with a non-empty IFC, add an edge from the component’s flow node to all addresses it wrote.

Propagating the IFC over component calls ensures that all implicit flows are considered when the SCAs are computed. Thus, the analysis now possesses the entire information flow in which SCAs can be detected.

B. SCA Refinement Condition

Cyclic reinforcements are problematic as the information stored at the addresses within the SCA is used for its own computation, for which the outdated information within an SCA reinforces itself. To this end, the analysis must be able to detect when the values stored at the addresses of an SCA may have become outdated and, thus, when refinement is needed. This is when at least one of the following holds:

- (a) An *externally incoming value* is updated non-monotonically. This is a value at an address that is not part of the SCA but from which there is an edge in the IG to an address that is part of the SCA.
- (b) A value in the store is no longer externally incoming or the SCA is (partially) broken up (this is a special case of a value that is no longer incoming).

- (c) A literal value flowing to the SCA is no longer used.

Using $a \rightsquigarrow b$ to denote an edge from a to b in the IG, using \rightsquigarrow' for edges in the updated IG, and using SCA' to denote the updated set of SCAs, we can express this condition as follows:

$$\forall s \in SCA' : \exists a_t \in s :$$

$$\left[\left[\exists a_s \in \text{dom}(\sigma') : \left[(a_s \rightsquigarrow a_t) \wedge (a_s \rightsquigarrow' a_t) \wedge (\sigma(a_s) \not\sqsubseteq \sigma'(a_s)) \right] \right. \right. \quad (\text{a})$$

$$\left. \vee \left[a_s \rightsquigarrow a_t \wedge \neg(a_s \rightsquigarrow' a_t) \right] \right] \quad (\text{b})$$

$$\vee \left[\exists l \in \text{Lit} : (l \rightsquigarrow a_t) \wedge \neg(l \rightsquigarrow' a_t) \right] \quad (\text{c})$$

$$\implies \text{refine}(s)$$

When (a) holds, a value that is introduced into an SCA has been updated non-monotonically, causing outdated information in the SCA which therefore needs to be refined. (b) holds when an information flow entering the SCA is removed, or – as a special case – when it is (partially) broken. Thus, as less information is entering the SCA, it needs to be refined as the values in the addresses of the SCA may have become imprecise. Finally, when (c) holds, a literal value is no longer entering to the SCA, for which refinement is also needed. Algorithm 2 shows this condition on Lines 1–2.

Consider again the example of Section II-C2, where the argument passed to `fun` changes from `#t` to `#f`. As shown in Figure 2a, the value of `arg` flows to a cycle. In this case, the value is externally incoming and updated non-monotonically: condition (a) holds and the cycle needs to be refined.

1) *Optimisation*: Computing the information flow and the SCAs is computationally expensive since after every CA_{intra} it is checked whether SCAs can be refined. To optimise this, we derived a more lightweight test to determine whether steps 1 to 4 (the entire cycle refinement) can safely be skipped. The test checks whether a non-monotonic store update has taken place or whether information flows have been deleted. If the test does not hold, then the refinement condition specified above cannot hold: if no non-monotonic store update has taken place this excludes case (a), and if no information flows have been deleted this excludes cases (b) and (c). Algorithm 2 shows this optimisation on Line 14.

C. SCA Refinement

When the refinement condition holds for an SCA, it can be refined. To this end, the values of all addresses that are part of the SCA are set to \perp in σ , and the corresponding provenance information, write information, and dataflow information is removed. As information written by components is removed, all components whose contribution is removed are added to the worklist. This allows to recompute the information based solely on precise values. In addition, all components reading an address whose value was updated also need to be added to the worklist, allowing the updates to be propagated. In Algorithm 2, SCA refinement is shown on Lines 19–22.

Continuing the previous example, refining cycle the will set `curr` to \perp and thereby removes the imprecise value **Bool**.

```

// The cache dfR stores the reverse information flow
// on a per-component basis, SCAs stores the
// previously computed SCAs.
1 Function refiningNeeded(s, σ', σ, dfR', dfR) is
   // unify collects the data stored per component.
2   return  $\left\{ w \mid (w, rs) \in \text{unify}(dfR) \wedge \left( (rs \setminus \text{unify}(dfR')(w) \neq \emptyset) \vee (\exists r \in rs : \sigma(r) \sqsubseteq \sigma'(r)) \right) \right\} \neq \emptyset$ ;
3 SCAs := scc_tarjan(dfR);
4 while WL  $\neq \emptyset$  do // Fixed-point computation.
5    $\alpha \in WL$ ; WL := WL  $\setminus \{ \alpha \}$ ;
6   (C', R', U', σ') = intra( $\alpha$ , σ); // CAintra.
7   // σ := σ'; // Omitted for write invalidation.
8   V := V  $\cup \{ \alpha \}$ ; WL := WL  $\cup (C' \setminus V)$ ;
9   foreach r  $\in R'$  do D := D[r  $\mapsto D(r) \cup \{ \alpha \}$ ];
10  foreach u  $\in U'$  do WL := WL  $\cup D(u)$ ;
11  σold := σ; dfRold := dfR;
   // Strategies from Section II-B2 omitted here.
12  if incremental update going on then
13    dfR := dfR[ $\alpha \mapsto \perp$ ]; // dFI computed during CAintra.
14    if non-incremental update or flow deleted then
15      flows := compute all information flow using dfR;
16      SCAs := scc_incremental(flows, SCAs);
17      foreach s  $\in SCAs$  do // SCA precision recovery.
18        if refiningNeeded(s, σ, σold, dfR, dfRold) then
19          foreach a  $\in s$  do
20            σ := σ[a  $\mapsto \perp$ ];
           // Omitted: information removal.
21            foreach
22              β  $\in$  components reading and writing a do
23                WL := WL  $\cup \{ \beta \}$ 
24  return (σ, V, D);

```

Algorithm 2: Incremental update with refinement of cyclic reinforcements of lattice values added to the inter-component analysis of Algorithm 1 in blue. The additions for the invalidation strategies of Section II-B2 have been omitted for brevity.

The subsequent reanalyses of `fun` and `loop` cause `curr` to become `#f`, and a precise analysis result is obtained once the fixed-point computation has terminated.

V. EVALUATION

We evaluated our method to answer the following questions:

RQ1: Precision Does the result of an incremental update with cycle invalidation always match the result of a full reanalysis?

RQ2: Performance How does an incremental update with cycle invalidation perform compared to a full reanalysis of the updated program?

We evaluate the precision of the incremental update with cycle invalidation to verify whether it produces the same results as a full reanalysis of the updated program. Precision was the major shortcoming of the MODINC approach [13], as the incremental analysis could lose precision, and we aim to improve upon its precision. We also evaluate the performance to compare the time needed by an incremental update to the time needed to perform a full reanalysis. In addition to the experiments required to present answers to the above questions, we also verified the correctness of our method by verifying that an incremental analysis always produces at least the same result as a non-incremental analysis, both on the initial program as on the updated program, for all programs in our benchmark suites (discussed later). We did not encounter any issues.

A. Experimental Design

To facilitate the comparison of our results to the MODINC approach [13], we extended their implementation in the MAF framework [27]. Our implementation has also been made

available open source.³ Our experimental setup also matches the setup used to evaluate the MODINC approach [13] to allow for a proper comparison: we used a context-insensitive function-modular analysis for Scheme with a LIFO-ordered worklist and a product lattice.⁴ Our evaluation was run on a 2021 computer with an AMD Ryzen Threadripper 3990X processor and 128GB of ram. We used Scala 3.1.0, OpenJDK 14.0.2, and sbt version 1.6.1, with a maximal heap size of 8GB. For all analyses, we used a timeout of 15 minutes.

We reuse the benchmark suites from [13]. One suite contains 32 *curated* programs, with manually added changes following possible real-world edits, and one suite contains 950 *generated* programs to which changes have been added programmatically according to several edit patterns. Changes are encoded using *change expressions*, e.g., as used in Listing 2. The programs originate from various sources, e.g., university courses with programming exercises, together with the solutions, and benchmarking suites used in the literature. For example, the curated freeze benchmark implements a metacircular interpreter and the changes add the ability to make variables immutable, `peval` contains 38 code changes that introduce and apply a new abstraction, and in `nbody-processed`, lists have been replaced by vectors.

Some of the programs in the curated suite have been explicitly created to contain reinforcing cycles. We have added an extra benchmark that also contains this behaviour and creates a cycle in a specific manner; our curated suite thus contains 33 programs instead of 32. The evaluation of the MODINC approach [13] shows that the incremental analysis loses precision on 12 curated benchmarks, meaning that our curated suite contains 13 benchmarks in which cyclic reinforcements are present. The MODINC approach also yields in imprecise results on 163 of the generated benchmarks, implying that cyclic reinforcements are present in these benchmarks.

B. RQ1: Precision

To evaluate the precision, we perform a full analysis on the initial program and then an incremental update of the initial result. We also run a full reanalysis on the updated program. We then compare every value in the store of the incremental update to the values in the store of the full reanalysis:

- If a value in the store of an incremental update is more precise than the corresponding value in the store of a full reanalysis, we consider the result unsound (incorrect).
- If a value is less precise than the corresponding value produced by a full reanalysis, precision is lost.
- Values matching the values produced by the full reanalysis indicate equal precision. This is the desired case.

We did not encounter unsound results, and treated soundness as a requirement for our incremental analysis. We found that on all but one benchmark, the incremental update produces

³A repository containing our implementation can be found online at <https://github.com/softwarelanguageslab/maf> (release SCAM 2025).

⁴We abstract a boolean as either `#t`, `#f`, or `Bool`, a function as a singleton set containing the closure, a pointer as a singleton set containing the corresponding store address, and any other value as its type.

results that match the result of a full reanalysis. The only program on which the incremental analysis still loses precision is a generated benchmark called R5RS_WeiChenRompf2019_the-little-schemer_ch3-5.scm, where 3 of the 121 addresses in σ contain a less precise value (ignoring addresses of primitive functions). The reason for this minor precision loss seems to be the presence of a cycle that is not refined. This may point to a small mistake in our implementation or to a rare edge case not covered by our method.

Answer RQ1. Our method successfully improves the precision of an incremental update. In all but one occurrence, the result of an incremental update matches the result of a from-scratch analysis, meaning that the precision loss to which the MODINC approach [13] was subjected to has been mediated successfully for all practical goals and purposes.

C. RQ2: Performance

To evaluate the performance, we compare the time needed by an incremental update to the time needed by a full reanalysis. To this end, we measured these times for every benchmark program 15 times, after a warm-up of 15 repetitions or maximally 15 minutes, and where a garbage collect is triggered prior to each measurement. Our results are shown in Fig. 5.

The results have been grouped based on the benchmark suite, and, for the generated suite, based on the time taken by (1) the initial analysis and by (2) the full reanalysis of the updated program, as every situation poses different possibilities and challenges for an incremental analysis. For example, on benchmarks with short-running analyses, there is less room to obtain speed-ups. In addition, this grouping allows for a comparison with the original work which divides the results in an identical manner.

On about half of the curated programs, an incremental update outperforms a full reanalysis. The same holds for the short-running generated benchmarks, comprising the vast majority of the suite. The latter sees more outliers. Hence, big speed-ups are possible but also big slow-downs are seen on some programs. We see a slowdown of a factor 10 or more on less than half of the long-running generated programs. On programs with a long initial analysis time but with a short reanalysis time, the incremental performance is the poorest, showing a slowdown of more than a factor 100 on about half of the programs and outliers above 10^4 . In this category, Van der Plas et al. [13] also reported a median slowdown of about 100: the changes in these programs may significantly alter program behaviour, so it is difficult for an incremental analysis to outperform a full reanalysis; as the updated program runs faster than the initial analysis, a lot of program behaviour may have been pruned away, precluding an incremental analysis to outperform a full reanalysis. We, however, see a higher slowdown in this category, as in the results of the MODINC approach [13], the top whisker was below 10^3 .

In general, the performance of an incremental update heavily depends on the changes and on their *impact* on the analysis result. When comparing our results to the results shown in

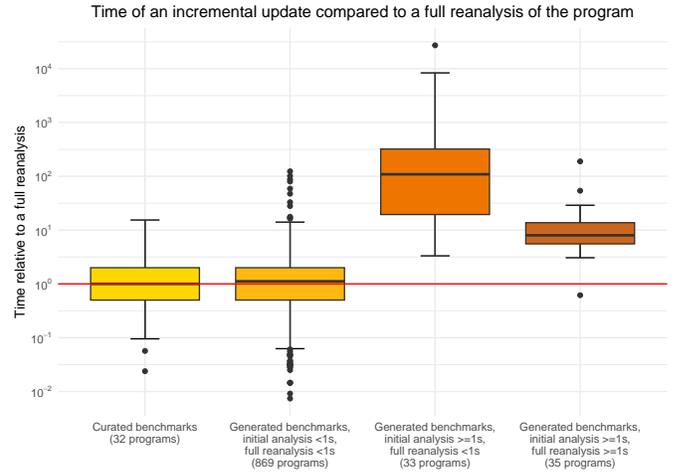


Fig. 5: Analysis time of an incremental update relative to a full reanalysis. Lower is better. Benchmarks for which the latter completed in 0ms or for which the log scaling rendered an infinite value have been omitted but have been counted in the categories. Benchmarks for which the incremental update timed out have been omitted entirely. The horizontal, red line at 1 marks the normalised analysis time needed by a full reanalysis. Data points below this line indicate execution times shorter than a full reanalysis; data points above this line indicate longer execution times.

the evaluation of the MODINC approach [13], we see that our method introduces an overhead in the incremental update, for which it is slower than a full reanalysis in many cases, although some slowdowns were already present in the original work. It is difficult to pinpoint a single reason why this may be the case. Possible reasons can include the following:

- Our SCA computation may cause an overhead, as SCAs need to be computed multiple times. We have tried to alleviate this overhead by reducing the size of the IG (see Section III-A2a), by computing SCAs incrementally as well (see Section III-B1), and by avoiding the computation of SCAs altogether when in some cases (see Section IV-B1).
- When SCAs are invalidated, the values at the addresses are set to \perp . However, in the MAF framework, the analysis of a component stops when \perp is encountered. We assume that, in this case, the result invalidation strategies from Section II-B2 may invalidate too many components, dependencies and writes to σ . In this case, the analysis has to recompute the data later, causing a performance loss.

We find that there may still plenty be of room to improve the performance of the incremental analysis, e.g., by optimising the implementation and by improving the worklist algorithm. However, these improvements are not the focus of this work: our objective has been to improve the precision of the incremental analyses (RQ1).

Answer RQ2. Handling cyclic reinforcements may cause slowdowns of the incremental update in comparison to an update without cycle invalidation and to a full reanalysis. Yet, performance has not been our main concern, for which multiple performance improvements may still be possible.

VI. RELATED WORK

We divide the work in the field of incremental static analysis into three categories that coincide with a general categorisation of work in the field of incremental computation [28]. We now discuss the most relevant related work in every category.

A. Bespoke Incremental Analyses

Bespoke incremental analyses are designed for a specific analysis to which the incrementalisation is tailored.

Yu et al. [30] present an incremental predicate analysis for regression verification of C programs, based on abstract interpretation. Every point in the control-flow graph (CFG) is annotated with an assertion on the abstract program state. The analysis syntactically invalidates outdated assertions and only recomputes these. However, the reliance on the CFG makes the approach unsuited for programs in which control flow and data flow are intertwined, as opposed to DDA. Another incremental analysis for C is presented by McPeak et al. [31]. To this end, the analysis is divided into *work units* of which the results are computed in parallel and cached. Contrary to components in DDA, work units can be analysed individually and do not require reanalysis, although a full analysis requires five passes over the entire program. The authors create one work unit per function, containing summaries of its callees and calling contexts to enable inter-procedural analysis. The modular incremental analysis of Garcia-Contreras et al. [32] achieves incrementality on the inter-modular and intra-modular level. It differs from DDA in that the division into modules is programmer-defined and lexical. The analysed program needs to be encoded in constrained Horn clauses. Nichols et al. [37], [38] introduce an incremental analysis for JavaScript programs, where a mapping is made from old to new program points. This allows result reuse for mapped points but an imprecise mapping may cause the analysis to lose precision. Also, the incremental analysis must reanalyse every program point at least once, i.e., the analysis does not bound the impact of changes to the affected parts of the result, unlike ours.

Other bespoke incremental analyses include an incremental resource-usage analysis for a sequential Java-like language [39], [40], analyses for classical data-flow problems [7], [8], [41], [42], points-to analysis [11], demand-driven taint analysis [36], race detection [12], reachability analysis [9], [10], incremental analyses built on attribute grammars [33]–[35], and a general incrementalisation for abstract interpretation based on generic local solvers [29].

B. Incremental Analysis Frameworks

Incremental analysis frameworks provide a DSL in which analyses can be specified and internally take care of the incrementalisation. They often support specific classes of analyses.

IncA [15], [16], [43]–[45] is a Datalog-based DSL for the specification of analyses using pattern functions, abstracting over graph patterns of interest. These patterns express relations over AST nodes using Datalog rules. IncA then provides an incremental Datalog solver, allowing to update the analysis result incrementally. It supports inter-procedural analyses with

recursive yet monotonic user-defined aggregations on custom lattices. Szabó et al. [15], [16] provide a solution to cyclic reinforcements in the context of Datalog-based analyses, utilising the uniform structure of Datalog while alternating the monotonic and anti-monotonic phases of their $DRed_L$ algorithm, an updated version of the $DRed$ algorithm [46]. It is clear, however, that our context differs from the one of IncA. Other incremental frameworks relying on logic programming are the framework of Saha and Ramakrishnan [52], JavaDL [51] and iQL [53]. Reviser [47], [48] supports incremental inter-procedural data-flow analyses implemented in the IDE [49] or IFDS [50] frameworks. Upon a code change, first a diff is generated, specifying updates to the CFG; this requires rebuilding the CFG from scratch. Then, all information is removed for affected nodes, after which Reviser’s solver propagates the analysis information to all predecessor nodes.

Unlike many of the frameworks, analyses built in a dependency-driven style do not require the conversion of the program and of the analysis to logical facts and rules, and they are not constrained to analyses specified in IDE or IDFS.

C. Derivation Methods for Incremental Static Analyses

This category comprises step-by-step incrementalisation methods to render existing analyses incremental, whose application to an analysis results in an incremental version of the original analysis. Just as with frameworks, derivation methods are often restricted to specific classes of analyses.

To the best of our knowledge, the only work fitting into this category is the work of Van der Plas et al. [13], [17], [21], which have already discussed in detail in Section II-B.

VII. CONCLUSION AND FUTURE WORK

We introduced a systematic method to handle cyclic reinforcement of lattice values in incremental dependency-driven analysis. Our method computes the information flow within the analysis and detects the SCCs in it; these indicate the presence of cyclic reinforcements. We then showed when the SCCs should be refined and presented an optimisation to avoid this process when the refinement condition cannot be true. Finally, we showed how outdated information can be removed.

We applied our method to extend the MODINC approach [13], which incrementalises static analyses using the computational dependencies within the program. However, its application lead to a loss in precision due to cyclic reinforcements. We have extended the approach in a general way and improved the precision of the resulting analyses up to the point where the precision of a full reanalysis is matched in all but one case. However, our method may lead to slowdowns of the incremental update. As performance was not our main concern, we foresee that there are multiple possible performance improvements that can still be investigated in future work. We plan to investigate a custom worklist algorithm, heuristics to find when it may be worthwhile to perform a from-scratch analysis rather than an incremental update, and whether SCA refinement could take place less frequently to reduce the overhead of computing the SCA refinement condition.

REFERENCES

- [1] B. Beizer, *Software Testing Techniques (2nd Ed.)*. Boston, MA, USA: International Thomson Computer Press, 1990.
- [2] M. Harman and P. W. O'Hearn, "From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis," in *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*. Los Alamitos, CA, USA: IEEE, 2018, pp. 1–23.
- [3] A. Alaboudi and T. D. LaToza, "Edit-Run Behavior in Programming and Debugging," in *Proceedings of the 2021 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2021, Saint Louis, MO, USA, October 10-13, 2021*, K. J. Harms, J. Cunha, S. Oney, and C. Kelleher, Eds. Los Alamitos, CA, USA: IEEE, 2021, pp. 1–10.
- [4] A. Alali, H. H. Kagdi, and J. I. Maletic, "What's a Typical Commit? A Characterization of Open Source Software Repositories," in *Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, R. L. Krikhaar, R. Lämmel, and C. Verhoef, Eds. Los Alamitos, CA, USA: IEEE, 2008, pp. 182–191.
- [5] L. Hattori and M. Lanza, "On the Nature of Commits," in *Workshop Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE Workshops 2008, L'Aquila, Italy, September 15-16, 2008*. Los Alamitos, CA, USA: IEEE, 2008, pp. 63–71.
- [6] R. Purushothaman and D. E. Perry, "Toward Understanding the Rhetoric of Small Source Code Changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 511–526, 2005.
- [7] T. J. Marlowe and B. G. Ryder, "An Efficient Hybrid Algorithm for Incremental Data Flow Analysis," in *Proceedings of the 17th ACM Symposium on Principles of Programming Languages, POPL 1990, San Francisco, CA, USA, January, 1990*, F. E. Allen, Ed. New York, NY, USA: ACM, 1990, pp. 184–196.
- [8] F. K. Zadeck, "Incremental Data Flow Analysis in a Structured Program Editor," in *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, Montreal, Canada, June 17-22, 1984*, M. S. V. Deussen and S. L. Graham, Eds. New York, NY, USA: ACM, 1984, pp. 132–143.
- [9] Y. Lu, L. Shang, X. Xie, and J. Xue, "An Incremental Points-to Analysis with CFL-Reachability," in *Proceedings of the 22nd International Conference on Compiler Construction, CC 2013, Rome, Italy, March 16-24, 2013*, R. Jhala and K. De Bosschere, Eds. Berlin, Heidelberg, Germany: Springer, 2013, pp. 61–81.
- [10] L. Shang, Y. Lu, and J. Xue, "Fast and Precise Points-to Analysis with Incremental CFL-Reachability Summarisation: Preliminary Experience," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, Essen, Germany, September 3-7, 2012*. New York, NY, USA: ACM, 2012, pp. 270–273.
- [11] B. Liu, J. Huang, and L. Rauchwerger, "Rethinking Incremental and Parallel Pointer Analysis," *ACM Transactions on Programming Languages and Systems*, vol. 41, no. 1, pp. 6:1–6:31, 2019.
- [12] S. Zhan and J. Huang, "ECHO: Instantaneous In Situ Race Detection in the IDE," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds. New York, NY, USA: ACM, 2016, pp. 775–786.
- [13] J. Van der Plas, Q. Stiévenart, and C. De Roover, "Result Invalidation for Incremental Modular Analyses," in *Proceedings of the 24th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2023, Boston, MA, USA, January 15-21, 2023*, ser. Lecture Notes in Computer Science, C. Dragoi, M. Emmi, and J. Wang, Eds., vol. 13881. Cham, Switzerland: Springer, 2023, pp. 296–319.
- [14] J. Nicolay, Q. Stiévenart, W. De Meuter, and C. De Roover, "Effect-Driven Flow Analysis," in *Proceedings of the 20th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2019, Cascais, Portugal, January 13-15, 2019*, C. Enea and R. Piskac, Eds. Cham, Switzerland: Springer, 2019, pp. 247–274.
- [15] T. Szabó, "Incrementalizing Static Analyses in Datalog," Doctoral dissertation, Johannes Gutenberg-Universität Mainz, Mainz, Germany, 2021.
- [16] T. Szabó, G. Bergmann, S. Erdweg, and M. Voelter, "Incrementalizing lattice-based program analyses in Datalog," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.
- [17] J. Van der Plas, "Incremental Static Program Analysis through Reified Computational Dependencies," Doctoral dissertation, Vrije Universiteit Brussel, Brussels, Belgium, 2024.
- [18] Q. Stiévenart, N. Van Es, J. Van der Plas, and C. De Roover, "A parallel workload algorithm and its exploration heuristics for static modular analyses," *Journal of Systems and Software*, vol. 181, p. 111042, 2021.
- [19] Q. Stiévenart, "Scalable Designs for Abstract Interpretation of Concurrent Programs: Application to Actors and Shared-Memory Multi-Threading," Doctoral dissertation, Vrije Universiteit Brussel, Brussels, Belgium, 2018.
- [20] D. Van Horn and M. Might, "Abstracting Abstract Machines," in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, MD, USA, September 27-29, 2010*, P. Hudak and S. Weirich, Eds. New York, NY, USA: ACM, 2010, pp. 51–62.
- [21] J. Van der Plas, Q. Stiévenart, N. Van Es, and C. De Roover, "Incremental Flow Analysis through Computational Dependency Reification," in *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 27-28, 2020*. Los Alamitos, CA, USA: IEEE Computer Society, 2020, pp. 25–36.
- [22] D. E. Denning, "A Lattice Model of Secure Information Flow," *Communications of the ACM*, vol. 19, no. 5, p. 236–243, 1976.
- [23] D. E. Denning and P. J. Denning, "Certification of Programs for Secure Information Flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.
- [24] J. Van der Plas, J. Nicolay, W. De Meuter, and C. De Roover, "MODINF: Exploiting Reified Computational Dependencies for Information Flow Analysis," in *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2023, Prague, Czech Republic, April 24-25, 2023*, H. Kaindl, M. Mannion, and L. A. Maciaszek, Eds., INSTICC. Setúbal, Portugal: SciTePress, 2023, pp. 420–427.
- [25] R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.
- [26] N. Van Es, Q. Stiévenart, and C. De Roover, "Garbage-Free Abstract Interpretation Through Abstract Reference Counting," in *Proceedings of the 33rd European Conference on Object-Oriented Programming, ECOOP 2019, London, UK, July 15-19, 2019*, A. F. Donaldson, Ed., 2019, pp. 10:1–10:33. [Online]. Available: <https://doi.org/10.4230/LIPIcs.ECOOP.2019.10>
- [27] N. Van Es, J. Van der Plas, Q. Stiévenart, and C. De Roover, "MAF: A Framework for Modular Static Analysis of Higher-Order Languages," in *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 27-28, 2020*. Los Alamitos, CA, USA: IEEE Computer Society, 2020.
- [28] Y. A. Liu, "Incremental Computation: What Is the Essence? (Invited Contribution)," in *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, PEPM 2024, London, UK, January 16, 2024*, G. Keller and M. Wang, Eds. New York, NY, USA: ACM, 2024, pp. 39–52.
- [29] H. Seidl, J. Erhard, and R. Vogler, "Incremental Abstract Interpretation," in *From Lambda Calculus to Cybersecurity Through Program Analysis - Essays Dedicated to Chris Hankin on the Occasion of His Retirement*, A. D. Pierro, P. Malacaria, and R. Nagarajan, Eds. Cham, Switzerland: Springer, 2020, pp. 132–148.
- [30] Q. Yu, F. He, and B.-Y. Wang, "Incremental Predicate Analysis for Regression Verification," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, 2020.
- [31] S. McPeak, C. Gros, and M. K. Ramanathan, "Scalable and Incremental Software Bug Detection," in *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013, Saint Petersburg, Russia, August 18-26, 2013*. New York, NY, USA: ACM, 2013, pp. 554–564.
- [32] I. Garcia-Contreras, J. F. M. Caballero, and M. V. Hermenegildo, "An Approach to Incremental and Modular Context-Sensitive Analysis," ETSI Informatica, Madrid, España, Technical Report CLIP-2/2018.0, 2018, <https://oa.upm.es/53067>.
- [33] G. Hedin, "Incremental Attribute Evaluation with Side-effects," in *Proceedings of the 2nd Workshop on Compiler Compilers and High Speed Compilation, CCHSC 1988, Berlin, German Democratic Republic, October 10-14, 1988*, ser. Lecture Notes in Computer Science, D. K.

- Hammer, Ed., vol. 371. Berlin, Heidelberg, Germany: Springer, 1988, pp. 175–189.
- [34] —, “Incremental Static Semantic Analysis for Object-Oriented Languages using Door Attribute Grammars,” in *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems, SAGA 1991, Prague, Czechoslovakia, June 4-13, 1991*, ser. Lecture Notes in Computer Science, H. Alblas and B. Melichar, Eds., vol. 545. Berlin, Heidelberg, Germany: Springer, 1991, pp. 374–379.
- [35] —, “Incremental Semantic Analysis,” Doctoral dissertation, Lund University, Lund, Sweden, 1992.
- [36] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, “Andromeda: Accurate and Scalable Security Analysis of Web Applications,” in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE 2013, Rome, Italy, March 16-24, 2013*, V. Cortellessa and D. Varró, Eds. Berlin, Heidelberg, Germany: Springer, 2013, pp. 210–225.
- [37] L. Nichols, M. Emre, and B. Hardekopf, “Fixpoint Reuse for Incremental JavaScript Analysis,” in *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, N. Grech and T. Lavoie, Eds. ACM, 2019, pp. 2–7. [Online]. Available: <https://doi.org/10.1145/3315568.3329964>
- [38] —, “Fixpoint Reuse for Incremental JavaScript Analysis (Extended Version),” 2019, <https://cs.ucsb.edu/sites/default/files/documents/worklist-reuse.pdf>, visited on 2023-08-23.
- [39] E. Albert, J. Correas, G. Puebla, and G. Román-Díez, “Incremental Resource Usage Analysis,” in *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, PA, USA, January 23-24, 2012*, O. Kiselyov and S. J. Thompson, Eds. New York, NY, USA: ACM, 2012, pp. 25–34.
- [40] —, “A multi-domain incremental analysis engine and its application to incremental resource analysis,” *Theoretical Computer Science*, vol. 585, pp. 91–114, 2015. [Online]. Available: <https://doi.org/10.1016/j.tcs.2015.03.002>
- [41] M. D. Carroll and B. G. Ryder, “Incremental Data Flow Analysis via Dominator and Attribute Updates,” in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988, San Diego, CA, USA, January 10-13, 1988*, J. Ferrante and P. Mager, Eds. New York, NY, USA: ACM, 1988, pp. 274–284.
- [42] L. L. Pollock and M. L. Soffa, “An Incremental Version of Iterative Data Flow Analysis,” *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1537–1549, 1989. [Online]. Available: <https://doi.org/10.1109/32.58766>
- [43] T. Szabó, S. Erdweg, and G. Bergmann, “Incremental Whole-Program Analysis in Datalog with Lattices,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021, Virtual Event, June 20-26, 2021*, S. N. Freund and E. Yahav, Eds. New York, NY, USA: ACM, 2021, pp. 1–15.
- [44] T. Szabó, G. Bergmann, and S. Erdweg, “Incrementalizing interprocedural program analyses with recursive aggregation in Datalog,” 2019, Presented at the Second Workshop on Incremental Computing, IC 2019, Athens, Greece, October 21, 2019.
- [45] T. Szabó, S. Erdweg, and M. Voelter, “IncA: A DSL for the Definition of Incremental Program Analyses,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, D. Lo, S. Apel, and S. Khurshid, Eds. New York, NY, USA: ACM, 2016, pp. 320–331.
- [46] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, “Maintaining Views Incrementally,” in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD 1993, Washington, DC, USA, May 26-28, 1993*, P. Buneman and S. Jajodia, Eds. New York, NY, USA: ACM, 1993, pp. 157–166.
- [47] S. Arzt and E. Bodden, “Efficiently updating IDE-based data-flow analyses in response to incremental program changes,” EC SPRIDE, Technical Report TUD-CS-2013-0253, September 2013, <https://bodden.de/pubs/TUD-CS-2013-0253.pdf>.
- [48] —, “Reviser: Efficiently Updating IDE-/IFDS-Based Data-Flow Analyses in Response to Incremental Program Changes,” in *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, Hyderabad, India, May 31-June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. New York, NY, USA: ACM, 2014, pp. 288–298.
- [49] M. Sagiv, T. Reps, and S. Horwitz, “Precise interprocedural dataflow analysis with applications to constant propagation,” *Theoretical Computer Science*, vol. 167, no. 1, pp. 131–170, 1996.
- [50] T. W. 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 1995, San Francisco, CA, USA, January 23-25, 1995*, R. K. Cytron and P. Lee, Eds. New York, NY, USA: ACM, 1995, pp. 49–61.
- [51] A. Dura, C. Reichenbach, and E. Söderberg, “JavaDL: Automatically Incrementalizing Java Bug Pattern Detection,” *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–31, 2021. [Online]. Available: <https://doi.org/10.1145/3485542>
- [52] D. Saha and C. R. Ramakrishnan, “Incremental and Demand-driven Points-To Analysis Using Logic Programming,” in *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP 2005, Lisbon, Portugal, July 11-13 2005*, P. Barahona and A. P. Felty, Eds. New York, NY, USA: ACM, 2005, pp. 117–128.
- [53] T. Szabó, “Incrementalizing Production CodeQL Analyses,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. New York, NY, USA: ACM, 2023, pp. 1716–1726.