Incremental Flow Analysis through Computational Dependency Reification

Jens Van der Plas, Quentin Stiévenart, Noah Van Es, Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium
{jens.van.der.plas,quentin.stievenart,noah.van.es,coen.de.roover}@vub.be

Abstract—**Static analyses are used to gain more confidence in changes made by developers. To be of most use, such analyses must deliver feedback fast. Therefore, incremental static analyses update previous results rather than entirely recompute them. This reduces the analysis time upon a program change, and makes the analysis well-suited for environments where the code base is frequently updated, such as in IDEs and CI pipelines.**

In this work, we present a general approach to render a modular static analysis for highly dynamic programs incremental, by exploiting dependencies between intermediate analysis results. Modular analyses divide a program in interdependent parts that are analysed in isolation. The dependencies between these parts stem, for example, from the use of shared variables within the program. Our incrementalisation approach leverages the modularity of the analysis together with the dependencies that it reifies to compute and bound the impact of changes. This way, only the affected parts of the result need to be reanalysed, and unnecessary recomputations are avoided.

We apply our approach to both a function-modular and a thread-modular analysis and evaluate it by comparing an incremental update of an existing result to a full reanalysis. We find reductions of the analysis time from 6% to 99% on 14 out of 16 benchmark programs, and on most programs the impact on precision is limited. On 7 of the programs, reanalysis time is reduced by more than 75%, showing that our approach results in fast incremental updates.

Index Terms—**Static Program Analysis, Incremental Analysis, Modular Analysis, Dynamic Languages**

I. Introduction

As software is developed, often, multiple small changes are made. Such changes usually impact only a limited part of the program [1]–[3]. Organisations use static analyses to gain more confidence in changes made by developers. Such analyses help developers review their code by reasoning about the program's behaviour to verify program properties and to detect potential bugs. Recent literature has shown that timely feedback of static analysis results is crucial and significantly impacts developer response [4]. Moreover, an analysis that produces results fast can be integrated into the software development process, by deploying it as part of a continuous integration system or within a development environment.

To deliver results fast upon a program change, incremental static analyses reuse and update results obtained from the analysis of a prior version of the program, which takes less time than a full reanalysis. To do so, incremental analyses must *efficiently* link code changes to the analysis results impacted by the change, and update these results accordingly while guaranteeing a correct result.

Although incremental analyses are not novel, previously presented techniques are often tailored to specific analyses [5], or require a statically known call graph [6]–[15], making them unable to support highly dynamic languages. Moreover, even for languages where the call graph can be known statically, the code changes may affect the structure of the call graph, which is often not accounted for by incremental analyses [16]. As the call graph is modified upon the introduction of a new function (call) or when code is inlined, for example, changes to a program can easily lead to an altered call graph.

In this work, we introduce a novel approach for rendering modular static analyses incremental. Our incrementalisation approach can be used to construct incremental modular analyses that support dynamic, higher-order languages as well as changes to the call graph, independently of the specific analysis. The approach is instantiated with a modular static analysis, which divides a program into parts that are analysed in isolation and reifies the dependencies between these parts [17]–[19]. We observe that these dependencies can be exploited to track the parts of the result impacted by a change, and to bound the impact of this change to only those parts of the result that are directly or indirectly affected (a possible dependency is, for example, the use of a shared variable–see Sections II-A2 and II-A3).

This paper makes the following contributions:

- We present a novel approach to incrementalise modular analyses. Our approach makes use of the intra-program dependencies reified by the analysis, and takes advantage of the division into modules to bound the impact of changes (Section III).
- To demonstrate the generality of our approach, we instantiate it for two context-insensitive modular analyses: a function-modular analysis and a thread-modular analysis.
- We perform a thorough evaluation of analysis time and precision using the two instantiations described previously (Section IV). We find that our approach leads to a reduction of the analysis time from 6% to 99% on all but 2 benchmark programs, and that the impact on precision is limited on most programs.

Without loss of generality, we present our approach from the viewpoint of a static analysis for Scheme programs. We claim that our approach is applicable to other languages too. Even though we only use context-insensitive instantiations for the evaluation, the approach is also applicable to context-sensitive analyses, as is shown by the example in Section II-A2.

II. Background

In this section, we cover background material on modular static analysis and on how we represent *changes* to a program.

A. Modular Static Analysis

Modular static analysis [20] is an approach to static analysis that scales well, and that can achieve good precision with low memory consumption [17], [19], [21]–[23]. Our incrementalisation approach is enabled by the design of modular analyses.

In a modular static analysis, the analysis of a program is decomposed into the analysis of elements of the program called *modules*. These modules can for example be function definitions [17], classes, or thread definitions [19], [22]. The analysis approximates the run-time equivalent of these modules as *components*. For example, a component corresponding to a function definition is a function invocation, containing not only the function definition but also its arguments and lexical environment. Other examples of components are class instances and threads. Hence, one module can correspond to multiple components created by the analysis.

Each component is analysed in isolation from every other. In the ideal case, the analysis result is obtained by composing the analysis results of all components. Since modules are usually a fraction of the program size, the analysis of each component is performed quickly and can be tuned to have a high precision.

In practice, however, components may interfere with each other: functions can call each other, classes interact, and threads may spawn other threads or read from shared variables. In a modular analysis, these interferences are reified as *dependencies* between components. When a new dependency is found or an existing dependency is updated, the analysis schedules the affected components for reanalysis, possibly triggering more dependencies until a fixed point is reached.

1) Algorithm for a Modular Analysis: We now briefly present a general approach to modular program analysis more formally, independently of the actual definitions of module and component. A modular analysis consists of two alternating phases: an *intra-component analysis* that analyses a single component while inferring its dependencies, and an *inter-component analysis* that schedules intra-component analyses based on their dependencies until a fixed point is reached.

The inter-component analysis keeps track of:

1) the analysis state, σ, which we detail shortly;
2) a set of visited components V;
3) a mapping of dependencies to components *Deps*;
4) a work list of components to visit *Work*.

The first three items constitute the analysis result (the work list will always be empty at the end of an analysis). The goal of the analysis is to compute the analysis state so that it over-approximates its concrete counterpart. More specifically, the analysis over-approximates all possible values for each variable in the program. To that end, the analysis state is a *global store* σ, which maps *abstract addresses* to *abstract values*. Each program variable will be associated to an abstract address, which will in turn be mapped (by σ) to an abstract value approximating the variable's possible concrete values.

Algorithm 1: Inter-component analysis.

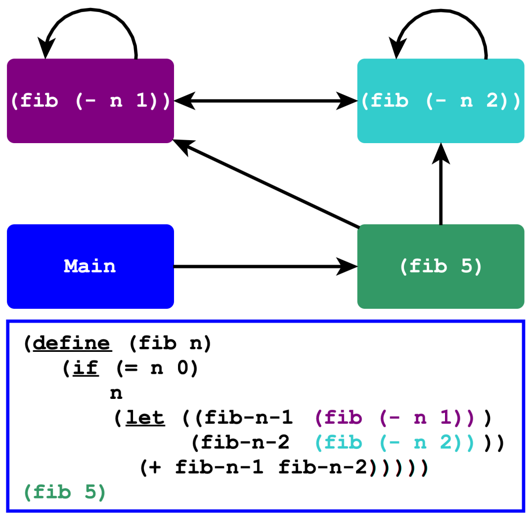
```
1  Function interComponentAnalysis() is
       // Work, V, Deps, and σ are globally available.
2      $Work := \{\textbf{Main}\},\ V := \varnothing,\ Deps := \lambda d.\varnothing,\ \sigma := \lambda x.\bot$;
3      computeFixedpoint();
4  end
5  Function computeFixedPoint() is
6      while $Work \neq \varnothing$ do
7          $cmp \in Work$;
8          $Work := Work \setminus \{cmp\}$;
9          $(\sigma', C, U, T) = intra(cmp, \sigma)$;
10         $\sigma := \sigma'$;
11         foreach $d \in U$ do
               $Deps := Deps[d \mapsto Deps(d) \cup \{cmp\}]$;
12         foreach $d \in T$ do $Work := Work \cup Deps(d)$;
13         $V := V \cup \{cmp\}$;
14         $Work := Work \cup (C \setminus V)$;
15     end
16 end
```

Function `interComponentAnalysis` in Algorithm 1 shows the inter-component analysis. Initially, the work list contains a special component **Main**, representing the entry point of the program under analysis (e.g., the top-level expression, the `Main` class, or the initial thread of the program). The initial visited set, store and dependency map are empty (line 2). Then, a fixed-point computation is started (line 3), which finishes when the work list is empty (line 6).

Each iteration, a component is selected from the work list and analysed by the intra-component analysis (lines 7–9). The intra-component analysis is performed by a function *intra* and depends on the programming language considered and on the definitions of module and component. The intra-component analysis returns the updated analysis state σ', a set of discovered components C, a set of inferred dependencies for the current component U, and a set of dependencies *triggered* T. A triggered dependency indicates that a specific part of the analysis state has been updated by the intra-component analysis. As other components may depend on this part of the analysis state, a dependency in T indicates to the inter-component analysis that the dependent components must be reanalysed to take this update into account.

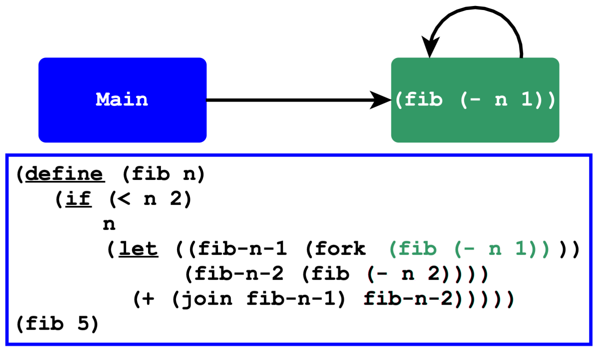
Next, the state of the analysis is updated (line 10) and the dependencies are registered in the dependency map *Deps* (line 11). Components are then added to the work list as follows: all components that depend on a dependency that was triggered by the analysis of the current component need to be reanalysed (line 12). The current component is marked as visited (line 13), and all new components, i.e., the components discovered except the ones that have been visited already, are added to the work list as well (line 14). Then, the algorithm proceeds with the next component in the work list.

Algorithm 1 illustrates how a modular analysis reifies intra-program dependencies as inter-component dependencies, and



```
(define (fib n)
   (if (= n 0)
       n
       (let ((fib-n-1 (fib (- n 1)))
             (fib-n-2 (fib (- n 2))))
         (+ fib-n-1 fib-n-2)))))
(fib 5)
```

Fig. 1. ModF components for the (incorrect) sequential computation of the nth Fibonacci number. An arrow from c_1 to c_2 indicates that c_2 was discovered during the intra-component analysis of c_1. Component contexts consist of the call expression, which is used to denote the components in the figure. No components are created for calls to built-in functions such as +.



```
(define (fib n)
   (if (< n 2)
       n
       (let ((fib-n-1 (fork (fib (- n 1))))
             (fib-n-2 (fib (- n 2))))
         (+ (join fib-n-1) fib-n-2)))))
(fib 5)
```

Fig. 2. ModConc components for the concurrent computation of the nth Fibonacci number using empty component contexts. An arrow from c_1 to c_2 indicates that c_2 was discovered during the intra-component analysis of c_1.

how these dependencies are used to drive the analysis: a component is only reanalysed when another component updates a part of the analysis state upon which the former component depends. This way, the analysis of one component takes into account the analysis results of all other components it depends on. To demonstrate the generality of our incrementalisation approach, we will apply it to two types of modular analyses, each with different definitions for module and component.

2) Function-modular Analyses: In the function-modular analysis we consider, modules correspond to function definitions, and components correspond to function calls. Such an analysis is referred to as a *ModF* analysis in the literature [17].

In a ModF analysis, when a function is called, this function is not directly analysed. Rather, a component corresponding to the function call is created and scheduled for analysis if the component has not been analysed before. A new dependency is registered from the caller to a specific address in the store that will contain the return value of the callee. When the analysis of a component terminates, this specific address is written to and the corresponding dependencies are triggered. For the analysis of programs in a language with first-class functions such as Scheme, a component consists of a function definition, its defining environment, and an optional *component context* that can be used to tune the precision of the analysis.

A ModF analysis knows two types of inter-component dependencies: *read* and *write*. Using these dependencies, the inter-component analysis is made aware of how addresses in the store are used by components, and can schedule the reanalysis of components accordingly.

Figure 1 illustrates a ModF analysis. The figure shows the components created by the analysis of a program computing the nth Fibonacci number, and indicates how components are discovered. In the example, the call expression is used as the component context. The analysis starts with the top-level expression of the program, which is represented by the **Main** component. When `fib` is called, a new component is created. The analysis of this new component will itself create two new ModF components. As components are only distinguished by a closure (`fib` in all cases) and a context (the calling expression), no other components are created.

3) Thread-modular Analyses: In the thread-modular analysis we consider, a module corresponds to a thread definition, and a component corresponds to a spawned thread. These are referred to as *ModConc* analyses in the literature [18], [19].

In a ModConc analysis, a component is created when a new thread is spawned. For a concurrent extension of Scheme, a component consists of the expression to be evaluated concurrently, the lexical environment of this expression, and an optional context. As in the case of ModF, ModConc knows two types of dependencies: *read* and *write*.

Figure 2 illustrates a ModConc analysis for a program that computes the nth Fibonacci number in parallel. The figure shows the components and indicates how components are discovered. In the example, component contexts are empty. The analysis starts with the **Main** component, and a second component is created upon the analysis of `fork`. This component encounters the `fork` expression again during its analysis, but since component contexts are empty, the same component is encountered and no new component is created. Therefore, the analysis result only contains two components.

B. Change Representation

To represent changes in a program, we take inspiration from Palikareva et al.'s *patch annotations* [24], and annotate programs with *change expressions*. As such, both the original and updated version of a program are represented by a single annotated program. The use of these change expressions avoids the trouble of unifying different program versions, allowing us to focus on the core problem of incrementalisation, rather than on the difficult tasks of change distilling [25] and change analysis [26], [27]. We assume that, upon a program change or from a version control system, a change distiller inserts the required annotations in the program AST. To facilitate

```
(define (fib n)
  (if (<change> (= n 0) (< n 2))
      n
      (let ((fib-n-1 (fib (- n 1)))
            (fib-n-2 (fib (- n 2))))
        (+ fib-n-1 fib-n-2))))
```

Listing 1: An annotated Fibonacci, fixing a bug in the end condition.

```
(define (fib n)
  (if ((<change> = <) n (<change> 0 2))
      n
      (let ((fib-n-1 (fib (- n 1)))
            (fib-n-2 (fib (- n 2))))
        (+ fib-n-1 fib-n-2))))
```

Listing 2: A Fibonacci with fine-grained annotations.

the evaluation of our approach, we currently insert change expressions manually in the program text.

As an example, consider the annotated program in Listing 1, fixing a bug in the program of Figure 1. In this program, the annotation denotes a change to the condition of the `if`-expression: the condition has changed from `(= n 0)` in the original program to `(< n 2)` in the updated version.

In contrast to Palikareva et al., our change expressions really are expressions and not annotations. This way, no invasive changes to the parser of the analysis are needed. However, some parts of the program cannot be edited as freely as with annotations: the entire program –including the change expressions– must still be a valid expression. It is therefore sometimes necessary to use more coarse-grained changes than with annotations. For example, to change the parameter list of a function, the entire function definition needs to be put inside a change expression, and to change a `let` special form into a `let*` special form, the entire `let`-expression must be put inside a change expression.

In general, change expressions may be applied at several levels of granularity. Consider for example the program in Listing 2. Here, the two annotations together represent the same change as the single one in Listing 1.

III. Approach

In this section, we describe our approach to incrementalise a modular analysis. Our approach is applicable to modular analyses of which the intra-component analysis infers dependencies, such as in Algorithm 1 for example.

To analyse the original program, a modular analysis as explained in Section II-A1 is performed. As a matter of convenience, our initial analysis already takes an annotated program, but ignores the annotations and treats the program as if it does not contain changes. We first give some additional details about how we denote program changes, before outlining how the analysis is incrementalised using our approach. For our ModF analysis, we assume function calls to be consistently updated when the arity of a function changes.

A. Change Categories

There are three categories of changes that can appear in a Scheme program:

i. Adding an expression to a sequence of expressions.
ii. Removing an expression from a sequence of expressions.
iii. Modifying the lexical environment of expressions. Changes of this type include moving an expression to a different scope and adding or removing bindings to/from an existing scope.

Any change can be regarded as a combination of changes in these categories. For example, modifying an expression is a combination of removing and adding an expression. Note that some changes, such as inlining a function call in a ModF analysis, may impact the creation of components. Also note that changes of type iii. may sometimes coincide with types i. and ii., e.g., when an expression is moved to a different scope.

The change expressions we use can represent all type of changes. Adding an expression is represented as `(<change> #f new-exp)`, where `#f` indicates the absence of an expression, and removing an expression is represented as `(<change> old-exp #f)`.

Changes of type iii. can be represented by enclosing all expressions affected by the change in the lexical environment inside the change expression. For example, defining a new variable may be represented as `(<change> (+ x 1) (let ((y 1)) (+ x y)))`. This however may lead to changes that are too coarse-grained. For example, adding a new definition at the beginning of a program will affect the entire program, and the incremental analysis will degenerate into a full reanalysis.

Using this kind of coarse-grained changes can be avoided by updating the internal data structures of the analyser to account for the changes to the environment of expressions. These updates are difficult as environments may be stored in several of the internal data structures of the analyser. For example, in lexically-scoped languages such as Scheme, environments are stored in closures. As closures are (abstract) values within the analysis, they are present in the store σ of the analysis. The complexity of these updates depends, for example, on the parameters used to tune the precision of the analysis. For example, if component contexts need to be updated, this may affect the number of components, and hence the dependency map *Deps*, as well as the addresses in the store. Hence, dealing with this kind of updates to the internal data structures of the analyser may not be trivial.

Instead, we avoid this complexity by introducing new bindings in the original program or the updated program as follows. In case a new variable is defined in the change, we represent this as `(let ((y (<change> #f 1))) (+ x (<change> 1 y)))`. When a variable is removed, we similarly replace the value it is bound to by `#f`. The environments therefore remain unchanged, but the binding is updated. Hence, the analyser's data structures do not need to be updated upon a program change, and changes of type iii. can be fine-grained as well.

Algorithm 2: The `analyze` function of the intra-component analysis.

```
1  Function analyze(e: Expr, ρ: Env, cmp: Comp) is
       // cmp is the current component.
       // trackMap is globally available.
       // trackMap :: Map[Expr → Set[Comp]]
2      trackMap := trackMap[e ↦ trackMap(e) ∪ {cmp}];
3      switch type of e do
4          case variable(id) do return lookup(id, env);
5          case fnCall(f, args) do
6              return analyzeCall(f, args, ρ);
7          end
8          case if(pred, then, else) do
9              return analyzeIf(pred, then, else, ρ);
10         end
11         case ... do ...;
12     end
13 end
```

Algorithm 3: The `findAffected` function.

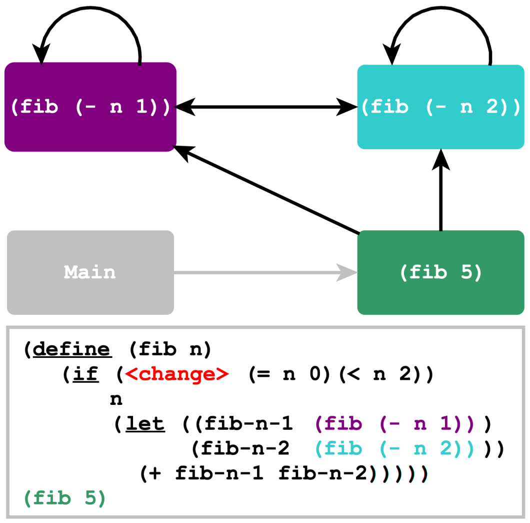
```
1  Function findUpdated(e: Expr): Set[Expr] is
2      switch type of e do
3          case change(old, new) do return {old};
4          otherwise do
5              return subExpressions(e).flatMap(findUpdated);
6          end
7      end
8  end
9  Function findAffected() is
       // program is globally available.
10     affectedExpr := findUpdated(program);
11     affectedComp := ∅;
12     foreach e ∈ affectedExpr do
         affectedComp := affectedComp ∪ trackMap.get(e);
13     return affectedComp;
14 end
```

B. Change Impact Calculation

Upon a change to the program, the analysis result needs to be updated. The result of a modular analysis consists of the analysis state, the set of visited components, and a mapping of dependencies to components. Hence, upon a change within the program, the analysis has to infer which components are impacted by the change and reanalyse them. Due to the modular design of the analysis, only the components that are directly impacted by the change must be explicitly scheduled for reanalysis. Components that are transitively impacted by a change need to be reanalysed as well. These, however, will be scheduled for reanalysis by the modular analysis itself when a dependency on which the components depend is triggered. This step in our approach already demonstrates how our incremental analysis benefits from modularity: the modular analysis will not only ensure that *all* components that are impacted directly or indirectly are reanalysed, but also that *only* those components are reanalysed.

To infer the components that are directly impacted by program changes, different approaches are possible depending on the type of modular analysis. For example, when using a ModF analysis, the components directly impacted by a change can be inferred lexically from the source code, by inspecting which function definitions are impacted. This is however not possible for a ModConc analysis, for example, as threads may execute code from multiple functions and it cannot always be inferred lexically which parts of the program a thread executes. We therefore propose a tracking approach, which is applicable to every type of modular analysis.

Our tracking approach works as follows. An intra-component analysis performs a fixed-point computation during which it steps through the code corresponding to the component. The analysis steps through the expressions one by one. Typically, an `analyze` function that checks the type of the expression and acts accordingly is used, as shown in Algorithm 2. Note that neither this `analyze` function nor the case splitting are needed by our approach, but we use them to illustrate how tracking should be incorporated. During the intra-component analysis, every expression that is encountered is registered (line 2). To this end, a mapping of expressions to sets of components is created (*trackMap*), which links an expression to all components during the analysis of which it was encountered. Hence, an expression that is never encountered during an analysis will be mapped to an empty set.



```
(define (fib n)
   (if (<change> (= n 0)(< n 2))
       n
       (let ((fib-n-1 (fib (- n 1)))
             (fib-n-2 (fib (- n 2))))
         (+ fib-n-1 fib-n-2)))))
(fib 5)
```

Fig. 3. Fix for the incorrect program of Figure 1. Only the **Main** component is not directly affected.

After the annotation of the AST with changes by the change distiller, the updated AST is traversed to collect all expressions that change (function `findUpdated` in Algorithm 3). Given these expressions, *trackMap* can be used to infer which components have been impacted directly by the change (function `findAffected` in Algorithm 3).

Consider the change made to the Fibonacci program in Figure 3. Our change tracking algorithm will infer that three out of the four components are directly impacted, only the component corresponding to **Main** is not.

Due to the way the directly impacted components are computed, we expect that the granularity of change expressions

Algorithm 4: Incremental update.

```
Function incrementalUpdate() is
    // V, Deps, and σ remain unchanged.
    Work := findAffected();
    computeFixedpoint();
end
```

can have an influence on the number of components that are inferred to be impacted directly. The reason for this lies in the points of the program where the control flow may follow one of several branches, as is the case for an `if`-expression for example. If the initial analysis for a component is precise enough to infer that only a specific branch is taken, the expressions of the other branches will not be related to the component by *trackMap*. Hence, if a change only spans a branch that was not taken, the corresponding component will not be inferred as being directly impacted. Therefore, changes spanning fewer branches may lead to fewer components being reanalysed.

C. Update of Analysis Results

After the set of directly impacted components has been computed, the analysis results can be updated, so as to obtain an over-approximation of the behaviour of the new program version.

To update the results, the set of directly impacted components is added to the work list of the modular analysis and the analysis is restarted, as shown in Algorithm 4. In the example of Figure 3, these are the components corresponding to `(fib 5)`, `(fib (- n 1))` and `(fib (- n 2))`. Remember that no other components need to be added to the work list, as components that are indirectly impacted by the changes will be scheduled for reanalysis by the modular analysis itself. For this, the analysis uses the inter-component dependencies that were reified in *Deps* during the initial program analysis. Consider the previous example. If the return value of the component `(fib 5)` changes, this will trigger a dependency causing **Main** to be reanalysed as well. However, when this value does not change, **Main** is not impacted by the change and is not reanalysed. Note that even though a value within the program may change, the corresponding abstract value used by the analysis may remain the same. For example, if a value is represented by its type in the abstract, **Main** does not need to be reanalysed.

During reanalysis, more dependencies can be inferred and new components can be discovered, as a regular modular analysis is performed. The results of the analysis are monotonically updated until a new fixed point is reached, i.e., no analysis results are "cleared" by the incremental update. Therefore, values can never be more precise than the values computed by a full reanalysis: the incrementally updated results over-approximate both the old and new program version, whereas a full reanalysis only has to over-approximate the new version. Because incremental analyses are run on small program changes, this is not expected to significantly deteriorate the precision of the analysis. However, throughout a series of incremental reanalyses, this can result in a more significant loss of precision. We consider the invalidation of analysis results as future work.

IV. Evaluation

To evaluate our approach, we have applied our incremental analysis to two different modular analyses for Scheme, a function-modular and a thread-modular analysis. Using these instantiations, we aim to answer the following questions:

RQ1 Does an incrementalised modular analysis result in a reduction of analysis time in comparison to a full reanalysis of the modified program?

RQ2 How precise is an incremental update compared to a full reanalysis of the modified program?

RQ3 What is the impact of the granularity of the components on the effectiveness of our approach?

A. Set-up and Benchmark Programs

Our approach, including the change expressions and two instantiations described above, has been implemented in MAF [28], a research framework for modular static analysis. We now describe the two instantiations of the incremental analysis used for evaluation in more detail, together with the benchmark programs used for each instantiation. Both instantiations approximate values by their type, except for functions that are approximated as sets of abstract closures, and use empty component contexts.

1) ModF Analysis for Scheme: The first instantiation used to evaluate our approach is a ModF analysis for Scheme [17]. The implementation of the intra-component analysis follows a big-step semantics. With this instantiation, we use a set of seven benchmarks programs to which change expressions have been added. These are listed in Table I, which also explains the changes made to the programs[1].

2) ModConc Analysis for Scheme: The second instantiation used to evaluate our approach is a ModConc analysis for Scheme [18], [19]. The ModConc benchmark programs use a version of Scheme that contains threads and locks, which are the concurrency constructs used in the ModConc literature. The implementation of the intra-component analysis follows a small-step semantics. With this instantiation, we use a set of nine benchmarks programs to which change expressions have been added. These are listed in Table II, which also explains the changes made to the programs[1].

B. Evaluation Method

To answer the research questions posed above, we use the following metrics:

1) The *analysis time*: we measure the time needed by (1) the initial analysis of the program, (2) an incremental update of the analysis results and (3) a full reanalysis of the updated program. To gain certainty in our measurements,

[1]The benchmark programs for ModF and ModConc are available at https://github.com/jevdplas/SCAM2020-Benchmarks.

TABLE I
THE BENCHMARK PROGRAMS FOR THE MODF EVALUATION, RETRIEVED FROM VARIOUS SOURCES. LINES OF CODE AS COUNTED WITH `cloc`.

Benchmark	LOC	Description of the Sequential Program	Changes	#Changes
mceval-dynamic	246	Meta-circular evaluator for Scheme, executing a small Scheme program.	Changed the evaluator so procedures become dynamically scoped.	4
leval	378	Lazy Scheme evaluator, used to perform some list computations.	Changes to the evaluator so that only specific arguments are evaluated lazily.	11
multiple-dwelling (fine)	404	Evaluator for a non-deterministic Scheme, used to solve an allocation problem.	Fine-grained changes to the input for the evaluator.	3
multiple-dwelling (coarse)	434	Evaluator for a non-deterministic Scheme, used to solve an allocation problem.	Changed the input for the evaluator.	1
peval	507	Partial evaluator for Scheme, used to evaluate multiple small programs on given input.	Abstracted repeated code to a function and replaced all occurrences by a call to this function.	38
nboyer	636	Version of the Boyer benchmark. Evaluator for logic programs, applied to a small logic program.	Rewrote conditionals with two branches to if-statements.	2
machine-simulator	1116	Compiles a factorial into machine code, then uses a simulator to execute this code.	Modified the compiler to generate faster code for some primitive functions.	7

TABLE II
THE BENCHMARK PROGRAMS FOR THE MODCONC EVALUATION, ADAPTED FROM VARIOUS SOURCES. LINES OF CODE AS COUNTED WITH `cloc`.

Benchmark	LOC	Description of the Concurrent Program	Changes	#Changes
mcarlo2	28	Monte Carlo simulation.	Now creates less threads, to avoid waiting on a thread just created.	2
pc	43	Producer-Consumer problem.	Converted a variable into a function, and replaced variable references with function calls.	2
msort	44	Merge sort.	Updated implementation of `sorted?` to avoid creating useless threads.	3
pps	71	Parallel-Prefix Sum.	Swapped around the statements in the body of a procedure.	1
sudoku	84	Sudoku checker.	Changed the sudoku board by replacing a number by `'oops`.	1
actors	103	An implementation of actors using threads.	Replaced `begin`-expressions with a single expression in their body.	2
stm	138	Implementation of Software-Transactional Memory.	Updated definitions of `every?` and `map-contains?` to improve code style.	2
crypt	170	Implementation of Vigenère cipher cryptanalysis.	Changed the implementation of `fold`.	1
crypt2	174	Implementation of Vigenère cipher cryptanalysis.	Changed the implementation of `argmin` to use `foldl`.	1

every measurement is repeated 35 times, of which the first 5 repetitions are considered warm-up and discarded.
2) The *precision* of values in the store: for every address mapped in the store, we compare the abstract values computed by the incremental update and full reanalysis.
3) The *size of the store*.
4) The *number of components* discovered by the analysis.
5) The *number of dependencies* inferred by the analysis.
6) The *number of intra-component analyses* performed.

Using the above metrics, we compare the analysis time and result obtained after an incremental update to those of a full analysis of the updated program. The lower the number of components and dependencies, and the smaller the size of the store, the more precise the analysis results are. We also compare all abstract values mapped to in the store. As they are part of a lattice, the partial order relation of the lattice is used to see which values are more precise. All experiments were run on a 2015 Dell PowerEdge R730 with 2 Intel Xeon 2637 processors. We used OpenJDK 1.8.0_265, Scala 2.13.3 and a maximal heap size of 4GB.

C. Experimental Results

Tables III, IV and V contain our experimental results.

Table III contains the results for our evaluation of the analysis time. For ModF, we note a reduction of the analysis time from 40% up to 99% for all but one benchmark, `multiple-dwelling (coarse)`, for which the incremental update is a lot slower than a full reanalysis. For ModConc, we see reductions of the analysis time ranging between 6% and 99% on all but one benchmark, `msort`, for which the incremental update is slightly slower than a full reanalysis. These numbers indicate that our approach overall results in reduced analysis times.

There are two versions of `multiple-dwelling`, as, for this program, the same changes were easily applied using different granularities of change expressions. Hence, the difference between the two versions is striking as they both represent the same program with the same code changes, though the granularity of the expressions used to encode the changes differs. In both versions, an input list is changed; in the coarse-grained version, the entire list is updated, whereas in the fine-grained version, the change expressions are put around the elements of the list that change. We find that this difference might be explained by the fact that our analysis cannot invalidate outdated results, which is exacerbated by the exact change: the change to `multiple-dwelling (coarse)` causes an entire new list to be allocated by the analyser, thereby creating a vast amount of pointers. We find that, after the incremental update, the store of the analysis contains almost 60% more pointers for the coarse-grained version than for the fine-grained program version. As pointers cannot be efficiently joined by our implementation, this possibly causes the slowdown.

TABLE III
TIMING RESULTS USING A TIMEOUT. EVERY MEASUREMENT IS REPEATED 30 TIMES, OF WHICH THE AVERAGE IS SHOWN. THE DELTA SHOWS HOW THE TIME NEEDED BY THE INCREMENTAL UPDATE COMPARES TO THE TIME NEEDED BY A FULL REANALYSIS.

ModF					**ModConc**				
Benchmark	Initial Analysis [ms]	Full Reanalysis [ms]	Incremental Update [ms]	Δ	Benchmark	Initial Analysis [ms]	Full Reanalysis [ms]	Incremental Update [ms]	Δ
mceval-dynamic	226	124	72	-41.94%	mcarlo2	9	29	27	-6.90%
leval	1407	1971	489	-75.19%	pc	21	16	11	-31.25%
multiple-dwelling (fine)	8466	8822	2126	-75.90%	msort	117	151	194	+28.48%
multiple-dwelling (coarse)	3527	3533	15694	+344.21%	pps	421	423	1	-99.76%
peval	19753	17644	103	-99.42%	sudoku	86	90	62	-31.11%
nboyer	1397	1271	98	-92.29%	actors	1601	1595	354	-77.81%
machine-simulator	54124	57043	24093	-57.76%	stm	5384	5597	745	-86.69%
					crypt	7568	7351	2812	-61.75%
					crypt2	9315	10277	8340	-18.85%

TABLE IV
PRECISION RESULTS. THE TABLE INDICATES HOW MANY ADDRESSES IN THE STORE AFTER AN INCREMENTAL UPDATE CONTAIN A VALUE THAT IS EQUAL OR LESS PRECISE COMPARED TO A FULL REANALYSIS OF THE UPDATED PROGRAM. 66 ADDRESSES CORRESPONDING TO BUILT-IN FUNCTIONS ARE IGNORED AS THEY ARE NEVER ASSIGNED AND HENCE OF EQUAL PRECISION IN ALL CASES. A FOURTH COLUMN INDICATES THE NUMBER OF ADDRESSES PRESENT IN THE INCREMENTALLY UPDATED STORE MINUS THE NUMBER OF ADDRESSES IN THE STORE AFTER A FULL REANALYSIS.

ModF					**ModConc**				
Benchmark	Equally Precise	Less Precise	Less Precise [%]	Address Count (Δ)	Benchmark	Equally Precise	Less Precise	Less Precise [%]	Address Count (Δ)
mceval-dynamic	158	220	58.20%	10	mcarlo2	28	2	6.67%	1
leval	187	389	67.53%	10	pc	35	4	10.26%	1
multiple-dwelling (fine)	851	0	0.00%	0	msort	27	9	25.00%	1
multiple-dwelling (coarse)	231	817	77.96%	198	pps	99	0	0.00%	0
peval	919	2	0.22%	0	sudoku	101	0	0.00%	0
nboyer	2115	17	0.80%	1	actors	136	0	0.00%	0
machine-simulator	1676	14	0.83%	7	stm	156	0	0.00%	0
					crypt	141	3	2.08%	3
					crypt2	140	6	4.11%	6

TABLE V
NUMBER OF COMPONENTS CREATED, NUMBER OF DEPENDENCIES INFERRED AND NUMBER OF INTRA-ANALYSES PERFORMED BY THE INTIAL ANALYSIS OF THE ORIGINAL PROGRAM (I), THE FULL REANALYSIS OF THE UPDATED PROGRAM (R) AND THE INCREMENTAL UPDATE TO THE INITIAL RESULT (U).

ModF										**ModConc**									
	Components			Dependencies			Intra-Component Analyses				Components			Dependencies			Intra-Component Analyses		
Benchmark	I	R	U	I	R	U	I	R	U	Benchmark	I	R	U	I	R	U	I	R	U
mceval-dynamic	86	85	87	2647	2057	2742	1722	1529	273	mcarlo2	3	2	3	90	62	91	7	4	4
leval	101	107	109	4999	6683	6840	4155	4413	971	pc	3	3	4	66	67	85	8	7	6
multiple-dwelling (fine)	139	139	139	14138	14538	14538	7442	7442	1245	msort	3	2	3	105	77	112	11	6	9
multiple-dwelling (coarse)	139	139	139	14138	14498	22418	5063	5115	7982	pps	3	3	3	138	138	138	6	6	1
peval	90	91	91	23564	23570	24056	4816	5222	20	sudoku	30	30	30	1051	1051	1051	63	63	35
nboyer	45	45	45	20366	20364	20376	1360	1310	33	actors	2	2	2	233	233	233	4	4	1
machine-simulator	282	289	289	55452	56166	56173	43460	48502	7633	stm	2	2	2	268	268	268	7	7	1
										crypt	2	2	2	293	293	299	8	8	3
										crypt2	2	2	2	293	291	303	8	8	6

Table IV shows the results of our precision evaluation, obtained by comparing the abstract values at each address in the store. Recall that the incremental analysis can never be more precise than a full reanalysis. We see however that on a majority of benchmarks, the precision loss is very small to none. On some benchmarks, however, the loss in precision is more important. For example, `multiple-dwelling (coarse)` sees a huge loss in precision, as more than 75% of the values in the store is less precise. This can again be linked to the fact that the incremental update creates a lot of pointers while being unable to remove outdated results, as can be seen in the fourth column of the table. For the `msort` benchmark, we find that the imprecision arises due to the fact that the incremental update does not remove components: after the incremental update, the store contains abstract values at 9 addresses related to components that are not created by the full reanalysis. Hence, the values at these addresses computed by the incremental update are less precise than those computed by a full reanalysis.

Finally, we consider the results in Table V, which shows the number of components and dependencies discovered by the analysis, as well as the number of intra-component analyses performed for the initial analysis, full reanalysis, and incremental update of the initial analysis results. On all but three benchmarks, the incremental update requires less intra-component analyses than a full reanalysis, and hence overall our approach reduces the work required to reach an updated fixed point. The results of an incremental update are less precise than those of a full reanalysis as more components and/or dependencies are inferred for most benchmarks. However as discussed before, the impact on the abstract values in the store is limited. The loss of precision can be mitigated by performing a full reanalysis, e.g., at regular intervals. The point where a full reanalysis is needed may be depend on the actual analysis performed, and should be determined accordingly.

D. Discussion

Our results show that, in general, our approach leads to a reduction of the analysis time, compared to a full reanalysis of the program (**RQ1**). This is also visible when comparing the number of intra-component analyses required to reach the fixed point. On two programs, a slowdown is seen, which is caused by an increased imprecision due to the incremental update. However, in general, the precision of an incremental update seems to be comparable to that of a full reanalysis (**RQ2**). We see that for ModF, on average, our approach results in higher reductions of the analysis time than for ModConc (**RQ3**). This is most likely caused by the fact that the ModF analyses create more components than the ModConc analyses, given our evaluation set-up. Also, the components created by ModConc, that correspond to spawned threads, are generally bigger than the ones created by ModF, which correspond to function calls. Hence, ModConc leads to more coarse-grained incrementality for which the reduction of the analysis time may be smaller.

Our experiments show that our approach leads to a reduction of the reanalysis time when applied on context-insensitive analyses for Scheme, a highly dynamic, higher-order language. We find that our approach is sufficiently general to be applied on different types of modular analyses, as we have demonstrated using our experiments. Hence, our approach improves upon current incremental analyses that require a statically known call graph or are tailored to specific analyses.

As is shown in the example of Section II-A2, which uses call expressions as component contexts, our approach can also be instantiated with context-sensitive analyses. We restricted our evaluation to two context-insensitive analyses, where all components contain empty contexts. If more information is stored in a context, the analysis may create more components, which both impacts the incremental update and full reanalysis. This may also depend on the abstract domain used, as a component context can contain abstract values. Hence, extending the evaluation to context-sensitive analyses would require investigating other abstract domains. However, the abstract domain only impacts the creation of components via the component contexts, hence we do not expect changes in the results for context-insensitive analyses when using different abstract domains.

E. Threats to Validity

We now briefly identify possible threats to the validity of our results, following the classification of Wohlin et al. [29].

1) Analysis Framework: A threat to the *external validity* comes from the framework in which our approach has been implemented. This framework is based on ModF [17] and ModConc [18], [19], both inspired by the work of Cousot and Cousot [20]. Various precision-improving optimisations, such as abstract garbage collection [30], [31], exist, but are not incorporated in our framework. We do not expect detrimental changes to our results should they be integrated. Also, our approach has only been incorporated in a research-oriented framework. An incorporation in a production quality tool may be required to show how our approach performs in practice. We are however unaware of any industry-standard analysis frameworks for dynamic languages that offer heavyweight analyses and follow a modular design.

2) Evaluation: A threat to the *conclusion validity* of our experiments stems from the low number of benchmark programs used. In the literature, there is no standard set of benchmarks used to evaluate incremental static analyses of dynamic languages, for which we had to compose a benchmark suite ourselves. To compose this benchmark suite, we added change expressions to the benchmark programs manually. Another approach would be to add such expressions programmatically, enabling more benchmark programs, but such changes might not reflect real changes made by developers.

To each program, we have manually added changes. We did not possess change histories for the programs. We however made sensible and varied changes that could reflect actual developer edits. For example, some of the ModF benchmark programs have been used during university classes. To these programs, the changes correspond to solving a course assignment. For ModConc, the changes could correspond to refactorings. We therefore believe that our changes are sufficiently varied and realistic to validate our approach, even though the number of changes to some programs is limited.

V. Related Work

We now review related work on incremental static analysis.

A. Incremental Analyses with Static Call Graphs

Many approaches to incremental analysis rely on a statically available call graph that is not impacted by code changes. There have been such incremental adaptations of CFL reachability analyses [13], [14], IDE/DFS analyses [15], alias analyses [10], logic-based analyses [11], classical dataflow problems [6]–[8], interval analyses [9], model checking [12], and incremental analyses through novel analysis mechanisms such as diff-graphs [32], or for specific client analyses such as race detection [5]. Our approach is designed with at its core the ability to support call graphs that are discovered during the modular analysis and that are updated when the code changes.

McPeak et al. [33] propose an incremental and parallel static analysis for C programs. The analysis is split into deterministic work units of which the results are cached. Upon a code change, the cache is updated so that stale results are removed. Special care is taken to avoid including in the cache results that may need to be updated upon code changes, by relying on *stable anchor points* in the source code. To avoid dealing with this consideration, we use change expressions.

B. Incremental Analyses without Static Call Graphs

Liu et al. [16] propose an alternative to incremental points-to analysis that does not require expensive graph reachability computations. Like ours, this approach does not require to know the call graph of program before the incremental run – as it may be modified by the program change– and unlike ours, they achieve the same precision as a full reanalysis of the program. This approach remains however limited to flow-insensitive analyses, while our approach does not pose any restriction on the flow sensitivity of the intra-component analysis. As discussed in Section IV-D, our approach can be extended to context-sensitive analyses.

Seidl et al. [34] propose to use *generic local solvers* to provide incrementalisation in an analysis infrastructure without restricting the design of the analysis (modular, IDE, IFDS). Similar to our approach, this is achieved by modifying a top-down solver to leverage dependencies for incrementality. To reuse results from the previous run of the analysis, functions are matched to functions with the same name in the previous version of the program. For every modified function, results corresponding to any of the program points of the function are invalidated. We avoid matching functions by the use of change expressions which can encode finer-grained changes, and we do not invalidate program results, but update them in a monotone way. As a result, our approach may result in a loss of precision across incremental runs.

Nichols et al. [35] provide an incremental analysis for JavaScript. The analysis creates a mapping of analysis results from the old program points to the new program points. The fixed-point computation can then be restarted and makes use of this mapping to reuse analysis results, thereby accelerating the convergence of the analysis. Because the impact of changes cannot be bounded, all program points need to be reanalysed at least once. In contrast, our analysis bounds program changes to the affected analysis components, and many components can therefore be skipped during the reanalysis.

IncA [36]–[38] is an analysis framework that provides a DSL to specify analyses. Using a projectional editor, programs are represented using graph patterns that are constructed on top of the program's AST. An analysis is constructed by specifying graph patterns of interest, and using an algorithm for incremental graph pattern matching, IncA keeps analysis results up to date with code changes. In contrast, our approach can be applied to existing modular analyses and does not require analyses to be respecified as so called *pattern functions*, which is the case for IncA. Finally, to represent changes, our approach makes use of change expressions, whereas IncA is implemented on top of a projectional editor.

Andromeda [39] is a framework used to perform taint analyses incrementally, in a demand-driven way. Upon changes to the program, Andromeda performs a change impact analysis that computes the part of the analysis result to be invalidated, and the parts that need to be updated. The change impact analysis determines the affected data structures, and uses an auxiliary *support graph* to find outdated taint facts, which are then removed. Hence, unlike our approach, which may lose precision over incremental runs, Andromeda removes outdated facts causing the results to remain precise. Our change impact analysis is more lightweight, as intra-program dependencies are already reified by the modular analysis, and we only need to identify components directly affected by a change. Also, the approach we present merely requires a mapping from expressions to sets of components, no complex auxiliary data structures are involved. Finally, Andromeda is specialised to a taint analysis, while our approach can be applied to any modular analysis as long as dependencies are inferred by the intra-component analysis.

C. Summary

In comparison to the related work presented in this section, our approach results in general incremental modular analyses that can support highly dynamic, higher-order languages and program changes that modify the program's call graph. Our approach is not specific to any particular analysis, and allows the impact of changes to be bounded to the parts of the analysis results that are affected by program changes. Additionally, the analyses must not be reformulated, and only a single lightweight auxiliary data structure is required.

VI. Conclusion

We introduced an approach to incrementalise modular analyses based on reified inter-component dependencies. A change impact calculation infers the components directly affected by a change, which are then reanalysed. The reified dependencies ensure that transitively affected components are also reanalysed. Hence, the modularity of the analysis leads to a relatively straightforward incrementalisation, where only the analysis results for the components directly or indirectly affected by the changes are updated.

We applied our approach to both a function-modular and thread-modular analysis for Scheme, a dynamic, higher-order language. We found that an incremental update is faster than a full recomputation of the result on 14 out of 16 benchmark programs, reducing the analysis time by up to 99%. A high precision is retained for most benchmark programs.

We envision several possible improvements to this approach. First, we want to investigate changes that affect the lexical environment of expressions. Processing such changes requires an update of the internal data structures of the analysis, e.g., to update the lexical environments stored in closures. A second path to follow would be investigating the invalidation of outdated results, to improve the precision of updated results.

Acknowledgements

This work was partially supported by *Research Foundation – Flanders (FWO)* (grant numbers 11F4820N and 11D5718N), and by *"Cybersecurity Initiative Flanders"*.

REFERENCES

[1] R. Purushothaman and D. E. Perry, "Toward Understanding the Rhetoric of Small Source Code Changes," *IEEE Trans. Software Eng.*, vol. 31, no. 6, pp. 511–526, 2005.

[2] A. Alali, H. H. Kagdi, and J. I. Maletic, "What's a Typical Commit? A Characterization of Open Source Software Repositories," in *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. IEEE Computer Society, 2008, pp. 182–191.

[3] L. Hattori and M. Lanza, "On the Nature of Commits," in *23rd IEEE/ACM International Conference on Automated Software Engineering - Workshop Proceedings (ASE Workshops 2008), 15-16 September 2008, L'Aquila, Italy*. IEEE, 2008, pp. 63–71.

[4] M. Harman and P. W. O'Hearn, "From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis," in *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*. IEEE Computer Society, 2018, pp. 1–23.

[5] 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*, 2016, pp. 775–786. [Online]. Available: https://doi.org/10.1145/2950290.2950332

[6] 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. Deusen and S. L. Graham, Eds. ACM, 1984, pp. 132–143. [Online]. Available: https://doi.org/10.1145/502874.502888

[7] M. D. Carroll and B. G. Ryder, "Incremental data flow analysis via dominator and attribute updates," in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, J. Ferrante and P. Mager, Eds. ACM Press, 1988, pp. 274–284. [Online]. Available: https://doi.org/10.1145/73560.73584

[8] L. L. Pollock and M. L. Soffa, "An incremental version of iterative data flow analysis," *IEEE Trans. Software Eng.*, vol. 15, no. 12, pp. 1537–1549, 1989. [Online]. Available: https://doi.org/10.1109/32.58766

[9] M. G. Burke, "An interval-based approach to exhaustive and incremental interprocedural data-flow analysis," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 341–395, 1990. [Online]. Available: https://doi.org/10.1145/78969.78963

[10] J. Yur, B. G. Ryder, and W. Landi, "An incremental flow- and context-sensitive pointer aliasing analysis," in *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*, B. W. Boehm, D. Garlan, and J. Kramer, Eds. ACM, 1999, pp. 442–451. [Online]. Available: https://doi.org/10.1145/302405.302676

[11] 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, July 11-13 2005, Lisbon, Portugal*, P. Barahona and A. P. Felty, Eds. ACM, 2005, pp. 117–128.

[12] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards, "Incremental algorithms for inter-procedural analysis of safety properties," in *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, ser. Lecture Notes in Computer Science, K. Etessami and S. K. Rajamani, Eds., vol. 3576. Springer, 2005, pp. 449–461. [Online]. Available: https://doi.org/10.1007/11513988_45

[13] L. Shang, Y. Lu, and J. Xue, "Fast and precise points-to analysis with incremental cfl-reachability summarisation: preliminary experience," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, 2012, pp. 270–273. [Online]. Available: https://doi.org/10.1145/2351676.2351720

[14] Y. Lu, L. Shang, X. Xie, and J. Xue, "An incremental points-to analysis with cfl-reachability," in *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, 2013, pp. 61–81. [Online]. Available: https://doi.org/10.1007/978-3-642-37051-9_4

[15] S. Arzt and E. Bodden, "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 Press, 2014, pp. 288–298.

[16] B. Liu, J. Huang, and L. Rauchwerger, "Rethinking incremental and parallel pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 1, pp. 6:1–6:31, 2019. [Online]. Available: https://doi.org/10.1145/3293606

[17] 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 International Publishing, 2019, pp. 247–274.

[18] 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.

[19] Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover, "A General Method for Rendering Static Analyses for Diverse Concurrency Models Modular," *Journal of Systems and Software*, vol. 147, pp. 17–45, 2019.

[20] P. Cousot and R. Cousot, "Modular Static Program Analysis," in *Proceedings of the 11th International Conference on Compiler Construction, CC 2002, Grenoble, France, April 8-12, 2002*, R. N. Horspool, Ed. Berlin, Heidelberg, Germany: Springer, 2002, pp. 159–178.

[21] E. Goubault, S. Putot, and F. Védrine, "Modular static analysis with zonotopes," in *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, 2012, pp. 24–40.

[22] A. Miné, "Relational thread-modular static value analysis by abstract interpretation," in *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, 2014, pp. 39–58.

[23] M. Journault, A. Miné, and A. Ouadjaout, "Modular static analysis of string manipulations in C programs," in *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, 2018, pp. 243–262.

[24] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a Doubt: Testing for Divergences Between Software Versions," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. Williams, Eds. New York, NY, USA: ACM, 2016, pp. 1181–1192.

[25] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and Accurate Source Code Differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, I. Crnkovic, M. Chechik, and P. Grünbacher, Eds. New York, NY, USA: ACM, 2014, pp. 313–324.

[26] W. Muylaert and C. De Roover, "Untangling Composite Commits Using Program Slicing," in *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*. IEEE Computer Society, 2018, pp. 193–202.

[27] R. Stevens, T. Molderez, and C. De Roover, "Querying distilled code changes to extract executable transformations," *Empirical Software Engineering*, vol. 24, no. 1, pp. 491–535, 2019.

[28] 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 *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 27-28, 2020*. IEEE Computer Society, 2020.

[29] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, *Experimentation in Software Engineering - An Introduction*, ser. The Kluwer International Series in Software Engineering. Kluwer, 2000, vol. 6.

[30] M. Might and O. Shivers, "Improving Flow Analyses via ΓCFA: Abstract Garbage Collection and Counting," in *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, OR, USA, September 16-21, 2006*, J. H. Reppy and J. L. Lawall, Eds. New York, NY, USA: ACM, 2006, pp. 13–25.

[31] N. Van Es, Q. Stiévenart, and C. De Roover, "Garbage-free abstract interpretation through abstract reference counting," in *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, 2019, pp. 10:1–10:33. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2019.10

[32] J. Krainz and M. Philippsen, "Diff graphs for a fast incremental pointer analysis," in *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, ICOOOLPS@ECOOP 2017, Barcelona, Spain, June 19, 2017*. ACM, 2017, pp. 4:1–4:10. [Online]. Available: https://doi.org/10.1145/3098572.3098578

[33] S. McPeak, C. Gros, and M. K. Ramanathan, "Scalable and incremental software bug detection," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013, pp. 554–564. [Online]. Available: https://doi.org/10.1145/2491411.2501854

[34] 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 International Publishing, 2020, pp. 132–148.

[35] 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

[36] 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*, D. Lo, S. Apel, and S. Khurshid, Eds. New York, NY, USA: ACM, 2016, pp. 320–331.

[37] 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.

[38] T. Szabó, G. Bergmann, and S. Erdweg, "Incrementalizing interprocedural program analyses with recursive aggregation in Datalog," p. 3, Presented at the Second Workshop on Incremental Computing, IC 2019, Athens, Greece, October 21, 2019.

[39] 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.