Cross-Level Debugging for Static Analysers

Mats Van Molle
mats.van.molle@vub.be
Vrije Universiteit Brussel
Belgium

Bram Vandenbogaerde
bram.vandenbogaerde@vub.Be
Vrije Universiteit Brussel
Belgium

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

Abstract

Static analyses provide the foundation for several tools that help developers find problems before executing the program under analysis. Common applications include warning about unused code, deprecated API calls, or about potential security vulnerabilities within an IDE. A static analysis distinguishes itself from a dynamic analysis in that it is supposed to terminate even if the program under analysis does not. In many cases it is also desired for the analysis to be *sound*, meaning that its answers account for all possible program behavior. Unfortunately, analysis developers may make mistakes that violate these properties resulting in hard-to-find bugs in the analysis code itself. Finding these bugs can be a difficult task, especially since analysis developers have to reason about two separate code-bases: the analyzed code and the analysis implementation. The former is usually where the bug manifests itself, while the latter contains the faulty implementation. A recent survey has found that analysis developers prefer to reason about the analyzed program, indicating that debugging would be easier if debugging features such as (conditional) breakpoints and stepping were also available in the analyzed program. In this paper, we therefore propose *cross-level* debugging for static analysis. This novel technique moves debugging features such as stepping and breakpoints to the base-layer (i.e., analyzed program), while still making interactions with the meta-layer (i.e., analysis implementation) possible. To this end, we introduce novel conditional breakpoints that express conditions, which we call *meta-predicates*, about the current analysis' state. We integrated this debugging technique in a framework for implementing modular abstract interpretation-based static analyses called *MAF*. Through a detailed case study on 4 real-world bugs taken from the repository of *MAF*, we demonstrate how cross-level debugging helps analysis developers in locating and solving bugs.

CCS Concepts: • **Software and its engineering → Automated static analysis**; **Software testing and debugging**.

Keywords: Static Program Analysis, Debugging

ACM Reference Format:
Mats Van Molle, Bram Vandenbogaerde, and Coen De Roover. 2023. Cross-Level Debugging for Static Analysers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3623476.3623512

1 Introduction

Static analyses derive run-time properties of programs without actually running them. They provide the foundation for tools such as Integrated Development Environments, optimizing compilers, and quality assurance tooling. *Termination* is an important property for any static analysis, stating that the analysis always terminates even when the program under analysis does not. In application contexts such as compilers, analyses also ought to be *sound*, meaning that their results account for any possible execution of the program under analysis. For example, an analysis that determines whether integers can be stored in a unsigned variable, should only state that an expression will evaluate to a positive integer if this is the case for every possible program execution.

Unfortunately, analysis developers may make mistakes while trying to realise these properties. Such mistakes are often hard to locate and therefore fix. Debuggers have proven themselves as tools for locating the source of problems in an application. However, as Nguyen et al. [7] found in a survey conducted amongst 115 analysis developers, traditional debuggers are ill-suited for debugging a static analysis:

- *Debugging target mismatch:* a traditional source-level debugger targets the code of the analysis implementation. However, a bug usually manifests itself in an analyzed program. Therefore, for analysis developers, it can be easier to reason about the behavior of the analysis by looking at a specific analyzed program, rather than debugging the static analysis as a whole. Stepping features of the debugger should therefore also be able to target the analyzed program, rather than the analysis implementation itself.
- *Generic visualisation:* debuggers show generic information (e.g., the value of variables in the current call frame) about the implementation of a static analysis. As static analyses typically follow the same structure, *domain-specific* visualisations can be developed.

Nguyen et al. find that these domain-specific visualisation help to understand the behaviour of the analysis, and help to locate bugs.

In this paper, we argue furthermore that the breakpoints from traditional debuggers are inadequate:

- *Shifting breakpoints to the base layer:* traditional debuggers that target the analysis implementation do not support placing breakpoints in the analyzed program. This makes debugging more difficult, since the analysis developer cannot easily suspend the analysis when a particular point in the analyzed code is reached.
- *Domain-specific conditional breakpoints:* conditional breakpoints enable developers to limit the number of times a debugged program is suspended at a breakpoint. Similar to regular breakpoints, we argue that the conditional ones ought to be placed in the analyzed code. However, they must be *cross-level*, meaning that they do not only express properties of the analyzed program (base level), such as the contents of an in-scope variable, but also properties about the global analysis state (meta level) at the point of its evaluation. Predicates for conditional breakpoints therefore become domain-specific, which renders expressing properties about the analysis state easier compared to expressing them in terms of implementation-specific data structures.

1.1 Contributions

In this paper, we propose *cross-level debugging* for static analysis. More specifically, we propose a debugger that moves stepping and breakpoints features to the analyzed program (base level), while still allowing for expressing properties about the analysis implementation (meta level) as conditional breakpoints. To this end, we propose domain-specific *meta-predicates* that can be used to formulate analysis-specific conditional breakpoints. Our debugger therefore crosses the boundary between the base level and meta level, and becomes cross-level. In summary our contributions are as follows:

- A novel debugging technique for static analysis called *cross-level* debugging, which includes domain-specific visualisations and stepping features that can step through each individual step of the analysis.
- Three categories of domain-specific *meta-predicates* that can be used as the conditions for our *cross-level conditional breakpoints.*
- An implementation for this debugger using *MAF*, a framework for implementing modular analyses for Scheme.

1.2 Motivating Example

We motivate the need for cross-level debugging of static analysis implementations through a hypothetical sign analysis that does not properly allocate the parameters of a function. The bug manifests itself during the analysis of the following Scheme program:

```
1 ; define a function named "add"
2 (define (add x y)
3   (+ x y))
4 ; call the "add" function
5 (add 5 2)
```

When executed by a concrete interpreter, the program evaluates to 7. The corresponding analysis result for this program should be the + element of the sign lattice of abstract values (or its $\top$ element in case sound imprecisions are allowed). Imagine that the hypothetical sign analysis produces the $\bot$ lattice element instead, denoting the absence of sign information.

Without prior knowledge about this bug, it may be unclear what part of the analysis is to blame. Several analysis components may be at fault: the implementation of the abstraction of literals 5 and 2 to sign lattice elements, the implementation of the abstract + operation on these lattice elements, or the implementation of the abstract semantics of function calls and returns.

Inspecting the state of the analysis at the corresponding points in the analysed program would help to locate the bug in the analysis implementation. Unfortunately, regular debuggers are not well-equipped for this task. First, the analysis implementation does not exactly mirror the structure of the analysed program. Steps through the analysis implementation therefore do not necessarily correspond to steps in the analysed program. This motivates the need for moving stepping and breakpoint features to the analyzed program (base level) rather than the analysis implementation (meta level).

Second, regular debuggers do not understand the structure of the analysis state. For example, at line 3, it is expected that the analysis knows about variable y in the program under analysis. Although a regular debugger can visualize the state of the analysis implementation in terms of local and global variables, it does not provide an effective way to visualize the contents of, for example, the environments and stores that the analysis is manipulating. This motivates the need for *domain-specific visualisations* that show the analysis state on a more abstract level, rather than in terms of implementation-specific data structures.

Third, regular debuggers do not support analysis developers in formulating and testing hypotheses about the source of a bug in terms of program points from the program under analysis. For example, if the analysis developer suspects that the abstract semantics of function calls is to blame, it would be natural to place a breakpoint at the beginning of the `add` function. For the hypothesis that no addresses have been allocated in the store for the function’s parameters, the following *domain-specific conditional breakpoint* can help reduce the number of steps required to test it:

```
(define (add x y)
   (break (not (store:contains "y")))
   (+ x y))
```

Note that this breakpoint has not been formulated in terms of the program under analysis, but in terms of the state of the analysis when it reaches the corresponding program point in the program under analysis. This motivates the need for *cross-level conditional breakpoints* that can be placed in the analyzed code itself.

2 Effect-Driven Modular Static Analysis

In this paper, we focus on static analyses defined as abstract definitional interpreters, which use global store widening and are effect-driven in their worklist algorithm. In the next few sections we introduce each of these parts of our target static analysis, and illustrate how bugs can arise in their implementation.

2.1 Abstract Definitional Interpreters

Abstract interpretation [3] is an approach to static analysis where an analyser is derived by starting from the concrete semantics of the language under analysis, and then abstracting parts of this concrete semantics. As an example, consider a language that consists of numeric literals, addition (+) and subtraction (−) or any combination thereof. In its concrete semantics, numeric literals evaluate to themselves and addition and negation are defined as usual.

An abstraction of this language could abstract each number to its *sign*. In this semantics, the abstraction for 5 would be +. We also write that $\alpha(5) = +$, where α is called the *abstraction function*. The abstract versions of the addition ($\hat{+}$) and subtraction ($\hat{-}$) operations have to be defined differently. For example, summing two positive numbers results in a positive number. However, summing a positive and negative number could result in either a positive or a negative number. To remain sound, an analysis has to account for both possibilities. Hence, a third value is introduced called *top* (denoted by the symbol $\top$) expressing that the sign of the number could be either negative or positive. A final value called bottom (denoted by $\bot$) is included to express the absence of sign information. The set of these values forms a *mathematical lattice*, meaning that a partial order ($\sqsubseteq$) and a join operation ($\sqcup$) can be defined. As illustrated above, abstract operations are often non-trivial to implement and could result in subtle issues with the result of the analysis.

Van Horn et al. [8] propose a recipe for deriving static analyses by systematically abstracting the small-step operational semantics of a programming language. More recently, this recipe has been transposed to the context of *definitional interpreters* [5, 10]. Definitional interpreters are a way of formally specifying programming language semantics through an interpreter implementation. These interpreters are usually formulated in a *recursive* way and proceed by case analysis on the type of expression that is being analyzed. For example, for evaluating a number literal, the following abstracted semantics can be used:

```
eval expr :=
   match expr with ℕ n ⟶ $\alpha(n)$ ; ... end
```

To satisfy our soundness requirement, this semantics needs to be *exhaustive*, meaning that it has to explore any possible program path that might occur at run time. For example, the analysis might be unable to compute the truth value of an `if` condition precisely (e.g., the value of `(> x 0)` is imprecise if x is $\top$). In those cases, the analysis has to explore both the consequent and the alternative branch, as either might be executed in a concrete execution. Such a semantics can be formulated as follows:

```
eval expr :=
   match expr with
      (if cnd csq alt) ⟶
         let $v_{cnd}$ = eval(cnd)
             $v_{csq}$ = if isTrue ($v_{cnd}$) then eval(csq) else ⊥
             $v_{alt}$ = if isFalse($v_{cnd}$) then eval(alt) else ⊥
         in $v_{csq} \sqcup v_{alt}$
      ...
   end
```

Note that `isTrue` and `isFalse` may succeed simultaneously if the truth value of the condition is imprecise. The excerpt depicted above demonstrates that the implementation of an abstract definitional interpreter is non-trivial. Throughout the implementation, there is a need to account for all possible concrete executions —which leads to subtle bugs when implemented incorrectly.

2.2 Memory Abstraction

To analyze programs that include variables, some kind of memory abstraction is required. In the recipe by Van Horn et al. [8] the interpreter's memory is modelled as a combination of an environment, which represents the lexical scope in which a particular program state is executed, and a store which represents the program's memory. An environment is modelled as a mapping from variables to addresses, and a store is modelled as a mapping from these addresses to actual (abstract) values.

The original recipe includes the store in every abstract program state. In the worst case, this results in an exponential number of program states [9]. One solution to this problem is to widen these (per-state) local stores into a single global store. This global store is then used across all global states, reducing the state space from an exponential to a cubic one. The same approach is taken in the static analysis for which we propose a debugger in this paper. However, our ideas also translate to analyses that do not incorporate store widening.

2.3 Effect-Driven Modular Analyses

The analysis presented in this paper is *modular* [4]. In a modular analysis, the program under analysis is split into *components* that are analyzed separately from each other.

Examples of such components are function calls [16], classes or spawned processes [18]. In practice, however, components might depend on each other through shared variables or return values. The result of analyzing some component B might therefore influence the analysis result for a component A, if the analysis of A depends on the analysis results of B.

Nicolay et al. [16] describe an algorithm for the modular analysis of higher-order dynamic programming languages. In higher-order dynamic programming languages, such as Scheme, the exact components of a program and their dependencies are not known before the program is executed. Each time a component is discovered, it is added to a *worklist* that keeps track of the components to analyze next. Whenever the results for a component change, its dependent components (e.g., through a shared variable) are added to this worklist and eventually reanalyzed to take the new results into account. The algorithm repeats itself until the workist is empty. This process results in a *dependency graph* that consists of components and the store addresses (representing shared variables and return values) through which they depend on each other.

3 Approach

In this section, we introduce the design of our analysis-tailored debugger. First we discuss its visualisation, then its stepping and breakpoint functionality, and finally we introduce our novel meta-predicates for cross-level conditional breakpoints.

3.1 Visualising the analysis state

The first feature of our debugger is its visualisation of the analysis state. This visualisation for the `factorial` program is depicted in fig. 1. The visualisation consists of four parts: the code that is being analyzed, a graph of the components and their dependencies, an overview of the global store, and a visualisation of the worklist algorithm.

The component graph (C) visualizes the components discovered so far and their dependencies. Colored in green are the components themselves, and colored in blue are the *store locations* (addresses) on which the components depend. The component currently under analysis is highlighted using a purple border. Each of the edges depicts dependencies on these store locations and their direction indicates the *flow of values*. For example, the call to the factorial function (depicted by the node labeled `Call...`) both reads (from its recursive call) and writes to its return value.

The global store visualisation (D) depicts the addresses and their corresponding values that are currently in the global store. Highlighted in yellow are addresses that are updated during the interval between the previous and current breakpoint, while highlighted in green are addresses that are added during that time frame.

Finally, the worklist visualisation (E) depicts the current contents of the worklist. Its order corresponds to the order in which components will be removed from the worklist and therefore shows their analysis order.

3.2 Stepping and regular breakpoints

Recall that analysis developers prefer to reason about a specific manifestation of a bug in an analyzed program, rather than debugging the analysis implementation as a whole. We achieve this in two ways. First the code is presented prominently in the interface of the debugger (area A). Second, the analysis developers step through and *break* in the analyzed program instead of through the analysis implementation itself. This is important since unsound results often occur in specific parts of the analyzed the program. Setting breakpoints and stepping through the analyzed program makes it easier to pin-point the problem, and reason about how the analysis proceeds for the analyzed program.

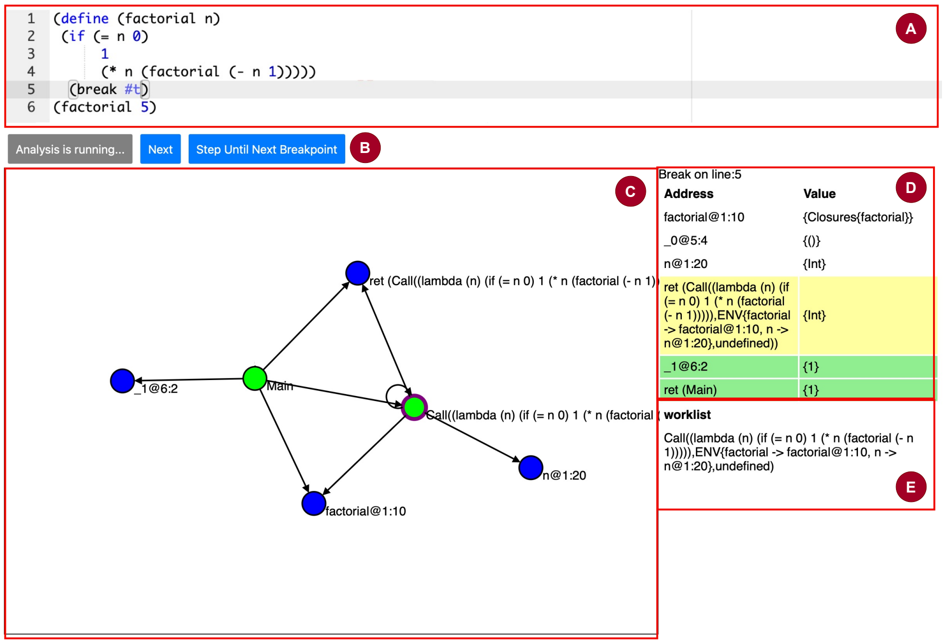
Similar to some debuggers for JavaScript, we choose to represent breakpoints as expressions in the analyzed program. This allows for more expressive freedom, since these expressions can be placed in arbitrary locations inside the program (i.e., in a specific subexpression, rather than on a specific line) and can reuse the same parsing facilities as the one available for the analyzed program.

Our debugger provides two types of stepping (as depicted in area B). The first type continues analysis until the next breakpoint is reached. The second type of stepping allows the developer to step over each expression in the analyzed program. Note that, because of our effect-driven worklist algorithm, this stepping feature never steps into function calls, since those are only analyzed once the component of the caller has been fully analyzed. However, once the analysis of a component is complete, this stepper continues stepping as before in the component that is analyzed next.

Recall that the branches of an if-expression need to be evaluated *non-deterministically*, in case the truth-value of the condition cannot be determined precisely. In that case, the stepper steps over these branches *sequentially* (evaluating the consequent branch first and then the alternative branch), and displays the state of the analysis accordingly.

Table 1. Overview of the meta-predicates in our debugger.

Store predicates	**Lattice predicates**	
`store:lookup`	`lattice:integer?`	`lattice:vector?`
`store:changed?`	`lattice:pair?`	`lattice:car`
Worklist predicates	`lattice:real?`	`lattice:cdr`
`wl:length`	`lattice:real?`	
`wl:prev-length`	`lattice:char?`	
`wl:component`	`lattice:bool?`	
`wl:prev-component`	`lattice:string?`	


A
Analysis is running...
Next
Step Until Next Breakpoint
B
C
ret (Call((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))
_1@6:2
Main
Call((lambda (n) (if (= n 0) 1 (* n (factorial (
n@1:20
factorial@1:10
D
E

```
1  (define (factorial n)
2   (if (= n 0)
3         1
4         (* n (factorial (- n 1)))))
5    (break #t)
6  (factorial 5)
```

Break on line:5

Address	Value
factorial@1:10	{Closures{factorial}}
_0@5:4	{()}
n@1:20	{Int}
ret (Call((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))),ENV{factorial -> factorial@1:10, n -> n@1:20},undefined))	{Int}
_1@6:2	{1}
ret (Main)	{1}

worklist

Call((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))),ENV{factorial -> factorial@1:10, n -> n@1:20},undefined)

Figure 1. Debugger visualisation, which features the following components: the code (A), debugger controls (B), component graph (C), store visualisation (D) and worklist visualisation (E).

.

3.3 Cross-Level Conditional Breakpoints

Conditional breakpoints are used in traditional debuggers to suspend the program once a particular condition is reached. These conditions are usually expressed in terms of program variables and predicates that act upon them. This type of breakpoint is especially important for static analyses where each program part can be analyzed more frequently than in their concrete execution. Hence, analysis developers need conditional breakpoints that can express conditions on the current state of the analysis. We call these kind of predicates *meta-predicates* since they do not express constraints on the program containing the breakpoints (i.e., the analyzed program) but rather on the meta layer above it (i.e., the analysis implementation).

Based on the parts of the analysis' state, we split our meta-predicates in three categories: store-based, worklist-based and lattice-based meta-predicates. A full list of predicates, split according to these categories is depicted in table 1.

3.3.1 Categories of Meta-Predicates.

Store predicates. Our store-based meta-predicates express conditions on the state of the global store. We propose two meta-predicates: `store:lookup` and `store:changed?`. The first predicate enables looking up a value on a specific address in the current global store. The argument of this meta-predicate must correspond to the string representation of the store address as displayed in the visualisation. If the address is absent, the meta-predicate returns `false`. This allows expressing conditional breakpoints that break on the absence of a particular store address. Note that the values returned by this predicate are *abstract* rather than concrete. Therefore, operations on these values can only be applied using the `lattice` meta-predicates.

The second predicate, called `store:changed?`, returns `true` whenever a particular address in the store has changed since the last break. It returns `false` whenever the value on that address has not changed or whenever the address could not be found.

The latter predicate is especially useful when components are (re-)analyzed frequently without actually changing any address of interest. Those re-analyses can simply be executed without breaking, therefore saving the analysis developer's time.

Lattice predicates. To interact with the values returned from `store:lookup`, we provide an interface to the abstract lattice operations as `lattice` meta-predicates. We divide these predicates into two sub-categories: type-checking predicates, and reified abstract operations.

For the former category, we provide type-checking predicates for Scheme's primitive values: integers, reals, characters, strings, booleans, pairs and vectors. The latter category provides operations on these datatypes, such as `lattice:car` and `lattice:cdr` to retrieve the first and second element of a pair respectively.

While the type-checking predicates return a boolean value that can be used for deciding whether to break, the abstract domain operations always return an abstract value. Therefore, a type-checking predicate is always needed to use an abstract operation in the condition of a conditional breakpoint.

Worklist predicates. Finally, we introduce predicates concerning the current state of the worklist: `wl:length` and `wl:component`. The former predicate returns the current length (as a concrete number) of the worklist. This length corresponds to the number of components that are scheduled for analysis. The latter returns the name of the component that is currently being analysed. This predicate is rather redundant for context-insensitive analyses, since the location of the breakpoint already implies which component is being analyzed. However, when the analysis is configured with a form of context sensitivity [17] (e.g., the last-k callers on the stack), multiple components might be created for the same function.

In addition to the aforementioned predicates, we also propose *history-aware* variants of them. These variants correspond to the `wl:prev-length` which returns the length of the worklist at the previous breakpoint, and `wl:prev-component` which returns the previously analyzed component. These predicates can be used to detect unusual behavior of the worklist algorithm. For example, a worklist that does not shrink in size could be indicative of a bug in the worklist algorithm itself. Furthermore, frequent re-analyses of the same component could hint that the analysis is not terminating. Using these meta-predicates, analysis developers can express invariants and expectations about the behavior of the worklist algorithm.

3.3.2 Examples.

In this section, we briefly show some examples of conditional breakpoints to illustrate the synergy between the different categories of meta-predicates.

```
(1) (break #t)
(2) (break (> (abs (- (wl-length) (wl:prev-length))) 100))
(3) (break (and (sto:changed? "adr")
                (lattice:string? (sto:lookup "adr"))))
(4) (break (lattice:char?
               (lattice:car (sto:lookup "adr"))))
```

Some brief examples are depicted in the listing above. (1) depicts a conditional breakpoint that always breaks, thus behaving like a regular breakpoint. In (2), the difference in length of the worklist is computed and some threshold (i.e., 100) is used to break. This breakpoint can be used to detect rapidly growing or shrinking worklists. (3) combines multiple predicates together using a conjunction (i.e., and). In this case the breakpoint will be triggered when the address *adr* has changed and the abstract value at that address in the store can be a `string`. Finally, (4) depicts a combination of type-checking lattice predicates, and lattice operations. In this case, the operation `car` is used to obtain the first value of a pair, and `lattice:char?` is used to check whether the value is a character.

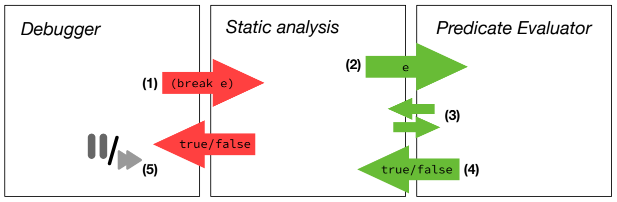


Figure 2. Interactions between the debugger and the meta-predicate evaluator.

Table 2. Summary of bugs from the Github repository.

Commit	Description
a2f43f6	Implemented `car` as `cdr`
1a3c6be	`vector-set!` ignores its own index
08bbe43	Variable arguments are ignored
8b98b9b	Unnecessary triggering of effects

3.3.3 Predicate evaluator.

Conditional breakpoints are evaluated in a separate evaluator which we call the *meta-predicate evaluator*. This evaluator has access to the current state of the analysis but cannot change it. Although the meta-predicate evaluator evaluates arbitrary Scheme expressions, these Scheme expressions cannot influence the results of the program under analysis. We argue that this is necessary for a clear separation between the debugging facilities and the analysis implementation to be maintained. The interactions between the debugger, static analyser, and meta-predicate evaluator are depicted in fig. 2.

The evaluation of a meta-predicate proceeds as follows. First, the `break` expression is analyzed by the static analysis (1). Then, since the static analyzer does not include semantics for evaluating predicates of those break expressions, the predicate expression e is passed to the predicate evaluator (2). Third, the predicate evaluator computes the truth value of the predicate e by querying the state of the static analysis (3). Finally, the computation results in a boolean value (true or false) which is returned to the static analysis (4). Based on this value the debugger decides whether to pause the analysis and show intermediate analysis results in its interface (5).

4 Evaluation

In this section we evaluate our approach through a case study. We first discuss the details of this evaluation method, and then demonstrate how our debugger supports locating 4 real-world bugs in the implementation of *MAF*.

4.1 Evaluation Method

We evaluated our approach by querying the *MAF* repository on Github[1] to find soundness related bugs. To this end, we queried for keywords such as: "bug" and "fix", From the

[1] https://github.com/softwarelanguageslab/maf

results of this query we selected 4 real-world soundness bugs which are summarized in table 2. Additionally, to illustrate how termination issues can be debugged, we introduced a synthetic bug that affects the *worklist algorithm*.

Then, based on the fixes introduced in the aforementioned commits, we reintroduced the bugs back into the analysis itself, adapting the bug to the current state of the framework if necessary. Each case corresponds to one re-introduced bug in isolation, in order to avoid the effects of multiple bugs influencing each other and to replicate the precise environment in which the bug was originally found and fixed.

4.2 Studied Cases

The following cases correspond to re-introduced bugs found in the *MAF* repository, and to one synthetic bug introduced specifically to study the effectiveness of our worklist meta-predicates. For each case, we first detail the corresponding real-world or synthetic bug, then we show how it was resolved, before describing a scenario of successive interactions with our debugger that will lead to the bug being located. In the remainder of this section, we use ● to indicate the location of a breakpoint.

4.2.1 Implemented `car` as `cdr`. In Scheme, pairs are constructed using the primitive `cons`. For example, the expression `(cons 1 2)` denotes a pair that consist of 1 as its first element (also called the `car`) and 2 as its second element (also called the `cdr`). In the bug studied in this first case, the `car` value was used for both the `car` and `cdr` of the pair allocated in the store.

We illustrate this bug in the program depicted below. This program provides an implementation for a *bank account*. The account is represented by a pair consisting of the account name and the current balance of that account (line 1-2). The functions `add-to-balance` (line 3-4) and `balance` (line 5-6) change and retrieve the balance of the bank account respectively.

```
  1   (define bankAccount
● 2     (cons "Lisa" 1983))
  3   (define (add-to-balance account amount)
  4      (set-cdr! account (+ amount(cdr account))))
  5   (define (balance account)
  6      (cdr account))
● 7   (add-to-balance bankAccount 10)
● 8   (balance bankAccount)
```

The analysis result for this program, produced by the buggy analysis implementation, is the pointer to `Lisa` instead of the expected value `integer`, rendering the analysis unsound.

Instrumenting the abstract definitional interpreter to output the analysis state at each evaluation step results in a large amount of unstructured information. Instead, we are interested in the analysis state for specific locations in the analyzed program. Recall that the store shows that the value of balance is `string`. To find the origin of the bug we start by checking whether `bankAccount` is still a `pair` consisting of a `string` and an `integer` after changing its balance. To this end, we place the following breakpoint before '`(balance bankAccount)`' (line 8), which breaks whenever the contents of the store has the expected structure.

```
(break (and
   (lattice:pair? (store:lookup "bankAccount@1:9"))
   (lattice:string? (lattice:car (store:lookup "bankAccount@1:9")))
   (lattice:integer? (lattice:cdr (store:lookup "bankAccount@1:9")))))
```

The inserted breakpoint does not suspend the analysis, meaning that the address does not point to a `cons` cell of the expected structure. Therefore, the bug has already occurred in the previous part of the program. A possible culprit could be the `set-cdr!` primitive, which mutates the `cdr` component of a pair. To test this hypothesis, we place the same breakpoint before `add-to-balance` (line 7). Again our analysis does not suspend, meaning that the structure of the pair is not affected by `set-cdr!`. Therefore the primitive `cons` itself could be the source of the bug. We test this by using the same breakpoint, but placing it right after the allocation (line 2). Again, this breakpoint does not result in a suspension of the analysis. We have now located that the implementation of `cons` itself is most likely to blame. To test this hypothesis, we reduce our conditional breakpoint to break whenever the `cdr` contains a value of an unexpected type (i.e., a value other than an `integer`).

```
  (break (not (lattice:integer?
     (lattice:cdr (store:lookup "bankAccount@1:9"))))
```

Finally, the analysis suspends, which means that the bug resides in the implementation of the abstract allocation of the pair.

4.2.2 `vector-set!` ignores its own index. In Scheme, *vectors* represent collections of a fixed size whose elements can be accessed in constant time. A vector can be allocated using the `make-vector` primitive which needs the length of the vector and an initial value for each position in the vector. Lookup and mutation are provided using primitives `vector-ref` and `vector-set!` respectively. In this example, we investigate a bug in the latter primitive.

The bug is located in the implementation of `vector-set!`. Recall that in order for an analysis to be sound, it must account for all possible program behavior. To this end, the implementation of `vector-set!` must join the previous values of the changed cell with the new value. Unfortunately, in this bug, only the old value was taken into account and the new value was simply ignored.

We demonstrate this bug with the program depicted below:

```
  1   (define (change-age user age)
  2      (vector-set! user 0 age))
  3
  4   (define (paid user)
● 5      (vector-set! user #f))
  6
  7   (define (set-name user name)
● 8      (vector-set! user 1 name))
  9
  10  (define (get-name user)
  11     (vector-ref user 1))
  12
  13    (define new-user (make-vector 3 #f))
  14
  15    (change-age new-user 21)
  16    (paid new-user)
● 17    (set-name new-user "Steve")
  18    (define name (get-name new-user))
● 19    name
```

We expect that the result of the analysis will be the value of the final expression (i.e., the value of the variable name). Since the name of the user is supposed to be a string, the abstract value associated with the address corresponding to name in the store should at least contain a string. However, the analysis results in false only. To debug this problem, we start by placing a breakpoint after name (line 19) .

```
(break (store:lookup "name"))
```

This breakpoint suspends the analysis whenever the variable name is added to the store. We observe that the analysis suspends at this breakpoint, meaning that the analysis reaches the final expression and the variable has been correctly allocated.

We shift our attention to functions paid (line 16) and set-name (line 17), which both change the contents of the vector. We test whether calling these functions has an unexpected effect on the allocation of the vector. To this end we add the following breakpoint after the execution of these functions (line 17).

```
(break (store:lookup "PtrAddr((make-vector 3 #f))"))
```

Since the breakpoint suspends the analysis, the vector is still properly allocated after the calls to these functions have been analysed. We also note that 'Steve'   '21' have been added to the store.

Our set-name and paid functions are both implemented using a vector-set!. The expected semantics for this primitive is that it reads the current contents of the vector and updates the value at the specified index. Therefore, the value at the store address of this vector is supposed to change after the primitive has been executed. To verify whether this is the case, we place a breakpoint on line 5 and on line 8 to suspend the analysis whenever the store *has not* changed.

```
(break (not (store:changed? "PtrAddr((make-vector 3 #f))")"))
```

This results in the analysis suspending at both line 5 and 8, meaning that the vector operations did not have the desired effect. We can conclude that the bug is therefore situated in the implementation of vector-set!.

4.2.3 Variable arguments are ignored.

. Functions in Scheme can be defined to accept a variable number of arguments. This is expressed using a '.' in the function definition, followed by the variable which will collect any excess arguments.

The program depicted below illustrates this feature. The program defines two functions: sum and compute, and calls the compute function as its last expression.

```
  1    (define (sum . vs)
  2      (define (aux l)
● 3        (if (null? l)
  4            0
  5            (+ (car l) (aux (cdr l)))))
  6
● 7      (aux vs))
  8
  9    (define (compute initial)
  10     (+ initial (sum 1 2 3 4)))
  11
● 12   (compute 0)
```

The expected result of the analysis is + (in case of a sign analysis). However, for this bug, the analysis result is ⊥. An analysis result of ⊥ may indicate that the program under analysis does not terminate or that the analysis is incomplete. As the program depicted above clearly terminates with value 10 when executed by a concrete interpreter, this analysis result is unsound.

We add a normal breakpoint to each component (i.e., on lines 3, 7, and 12) of the program to determine which components can be analyzed. The analysis suspends for the main and compute components but does not for the sum component. We conclude that the call to the sum function must have failed, which could be related to its use of a variable number of arguments. However, our debugger cannot determine a more precise cause for the bug, and further debugging in the analysis implementation is required.

4.2.4 Ever-growing worklist.

Although the bug studied in this case is synthetic, it could easily manifest itself while implementing a worklist algorithm. The bug we introduce precludes the worklist from reducing in size as the component under analysis is taken but not removed from the worklist. As a consequence, the analysis never terminates. We illustrate this problem with the factorial depicted below:

```
  1    (define (factorial n)
  2      (if (= n 0)
  3          1
  4          (* n (factorial (- n 1)))))
● 5    (factorial 5)
```

Since the analysis does not terminate, our debugger never displays a visualisation of its final state. To suspend the analysis we use regular breakpoints (i.e., (break #t)), and place them after line 5. We can now step through the analysis state. We notice that each time 'Step Until Next Breakpoint' is pressed, the contents of the worklist remains the same and the analysis' state does not change. To test whether the analysis *makes progress*, we replace our regular breakpoint by a conditional one. This breakpoint suspends the analysis whenever the current component is the same as the previous component, and when the length of the worklist does not change.

```
(break (and (eq? (wl:component) (wl:prev-component))
            (= (wl:length) (wl:prev-length))))
```

Again, the breakpoint suspends on every analysis step, meaning that the same component is re-analyzed in each iteration of the worklist algorithm. This makes it clear that the current component is never removed from the worklist.

4.2.5 Unnecessary triggering of dependencies.

As explained in section 2.3, the analysis is backed by an effect-driven worklist algorithm. Components are re-analyzed when one of their dependencies changes. We say that a dependency *triggers* the reanalysis of a component, meaning that the component is added to the worklist for reanalysis. For example, the analysis result of a function A depends on its arguments, which are represented by store addresses in our global store. Whenever the abstract values for one of these store addresses changes, it triggers the reanalysis of function A. In this bug, dependencies are triggered even though the (abstract) value of their referenced store address no longer changes. This results in a non-terminating analysis, since components continue to be added to the worklist even if no new information can be derived.

We illustrate this bug by reusing the example program from section 4.2.3. The analysis of this program is infinite in the buggy analysis implementation. We add breakpoints to the body of each component of this program to make sure that no component *in particular* is analyzed continuously. Stepping through this program a number of times reveals that a single component *is* being reanalyzed continuously: the `aux` component.

To reduce the number of times the analysis is suspended, we remove all other breakpoints except the breakpoint in the `aux` function. Since a component is only reanalyzed when one of its dependencies changes, we are interested in the argument of `aux`. Therefore, we adapt this breakpoint to suspend the analysis only when the `l` no longer changes:

```
(break (not (store:changed? "l@2:17")))
```

As a result, the analysis suspends less frequently and we can step directly to the problematic infinite behavior of the analysis. Additionally, this debugger interaction gives us an indication of which store address is to blame, and which type of value is associated with it. This makes it easier to find the root cause of the bug in the analysis implementation by focussing on that specific address or looking into the implementation of lists. However, additional logging in the analysis implementation is required to learn more about the dependency triggering mechanism.

4.3 Discussion

Table 3. Overview of all the meta-predicate categories used for solving the bug

	Regular Break	Store	Worklist	Lattice
Bug 1		✓		✓
Bug 2		✓		
Bug 3	✓			
Bug 4	✓		✓	
Bug 5	✓	✓		

Table 3 depicts on overview of the features of our debugger (columns) and the re-introduced bugs considered in the case studies (rows). The table indicates which debugging features were used to understand and locate each bug in the analysis implementation. In the case studies, predicates concerning the worklist are primarily used for solving bugs related to the termination of the analysis (bug 4). The store predicates are used in most of the case studies. The reason for their frequent usage is two-fold. First, the lattice predicates operate on abstract values from the store. Thus, each time a lattice predicate is used, at least one store predicate is required. Second, many bugs involve the store in some capacity. For example, bug 1 occurs because of a mistake in the *allocation* of pairs. Bug 2 is similar, in that the resulting value in the store is incorrectly updated, or not updated at all. The usage of the store meta-predicates in bug 5 is more subtle, here it is used to detect the absence of changes to the store in order to break when dependencies are triggered for addresses that no longer change.

Bug 3 is interesting since it precludes components from being analyzed. Even worse, by preventing a component from being analyzed, its return value is always $\perp$ which causes the analysis to halt early. In the program used for illustrating bug 3, this problem was rather obvious (i.e., the analysis results were empty). However, for larger programs, finding which component was prevented from being analyzed might be more difficult. Breakpoints related to the set of analyzed components (i.e., the seen set) might help to locate these components. However, such breakpoints are not included in our debugger and require further investigation. Therefore, only non-conditional breakpoints were used for debugging bug 3 in the case study.

5 Limitations & Future Work

As illustrated in the debugging scenario for bug 3, our current approach lacks meta-predicates to deal with components that fail to be analyzed. We argue that additional breakpoints that express properties on the dependency graph and the set of seen components can partially solve this problem, but leave this as future work.

Furthermore, our conditional breakpoints are *stateless*, meaning that they cannot keep any state between evaluations of the conditional breakpoints. We solve this problem in our current approach by introducing *history-aware* breakpoints such as `wl:prev-length`. However, as future work, *stateful* predicates can be considered to allow developers to keep track of an arbitrary state between the evaluation of breakpoints. To this end, language extensions and extensions to the predicate evaluator are needed.

Finally, whereas we establish a link between the analyzed code and the analysis implementation through fine-grained meta-predicates and visualisations of the analysis' state, Nguyen et al. [7] establish a clear correspondence between the analyzed code and the code of the analysis implementation by pairing them together visually in the debugger interface itself. Our debugger already keeps track of this information internally, but does not visualize it. However, we acknowledge that this pairing could be beneficial for understanding the analysis implementation as well as for finding the bug in the analysis implementation itself. We consider the integration of our debugger with an *Integrated Development Environment* as future work.

6 Related Work

Charguéraud et al. [2] propose a *double-debugger* for debugging Javascript interpreters using *domain-specific* breakpoints. These domain-specific breakpoints are about the internal interpreter state, and can be anchored within the interpreted program through predicates about line numbers and the contents of local variables. Similarly, Kruck et al. [11, 12] recognize that interpreter developers want to reason about the structure of the interpreted program and propose *multi-level* debugging. Their approach mainly focusses on the representation of call stack frames in a debugging environment, and represents them both from the perspective of the interpreted program as well as from the perspective of the interpreter itself. Similar approaches have been proposed for tailoring debuggers to specific applications or frameworks [14]. This allows developers to reason about the behavior of the interpreter more easily.

Both approaches are, however, not *cross-level*. They either provide domain-specific breakpoints on the meta level (e.g., breakpoints about the current line number of the interpreter), or do not provide them at all. Our breakpoints are placed in the analyzed code (base level) allowing the developer to specify the location where they are evaluated. Furthermore, the conditions in these breakpoints express properties of the analysis state (meta level) rather than the analyzed program (base level). These breakpoints therefore interact with each other and cross the boundaries between the base and the meta level.

Nguyen et al. [7] propose a tool called VisuFlow which is tailored to the visualisation of data flow analyses implemented in the Soot framework [13]. However, they do not propose cross-level domain-specific breakpoints and their approach is only applicable to data flow analyses. Static analyses based on the abstract definitional interpreter approach, however, have been shown to be applicable in many use-cases, including control flow analysis [1, 16, 20], data flow analysis [6] and soft contract verification [15, 19].

7 Conclusion

We proposed *cross-level debugging* for static analysis implementations, which moves stepping and breakpoints from the base level to the meta level. More specifically, we proposed domain-specific visualisations for visually depicting the current state of the analysis. We argue that this visualisation makes it easier to understand the behavior of the analysis and thus to locate the root cause of bugs.

Furthermore, we proposed *domain-specific conditional breakpoints* which enable breaking when a specific analysis state is reached. We divided these meta-predicates into three categories: store-related, worklist-related, and lattice-based predicates.

We implemented our debugger in a framework called *MAF*, and showed the applicability of our debugger on one synthetic and four real-world bugs lifted from the repository of the framework. In this case study, the debugger is highly effective for most bugs that relate to changes of store addresses and their contents, but less so for bugs that prevent analysis progress, or dependency-triggering related bugs. However, we argued that our approach is sufficienlty flexible to support these classes of bugs in future work through additional meta-predicates.

8 Acknowledgements

This work has been partially funded by FWO (Research Foundations Flanders) under grant number 1187122N.

References

[1] Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. 2019. Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.* 3, POPL (2019), 44:1–44:31. https://doi.org/10.1145/3290357

[2] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JSExplain: A Double Debugger for JavaScript. In *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*, Pierre-Antoine Champin, Fabien Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis (Eds.). ACM, 691–699. https://doi.org/10.1145/3184558.3185969

[3] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252.

[4] Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Compiler Construction, 11th International Conference (CC 2002) (Lecture Notes in Computer Science, Vol. 2304)*, R. Nigel Horspool (Ed.). Springer, 159–178.

[5] David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages* 1, International Conference on Functional Programming (2017), 12:1–12:25. https://doi.org/10.1145/3110256

[6] Jens Van der Plas, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. 2023. 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*, Hermann Kaindl, Mike Mannion, and Leszek A. Maciaszek (Eds.). SCITEPRESS, 420–427. https://doi.org/10.5220/0011849900003464

[7] Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. 2020. Debugging Static Analysis. *IEEE Transaction on Software Engineering* 46, 7 (2020), 697–709. https://doi.org/10.1109/TSE.2018.2868349

[8] David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 51–62. https://doi.org/10.1145/1863543.1863553

[9] J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. 2013. Optimizing abstract abstract machines. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 443–454. https://doi.org/10.1145/2500365.2500604

[10] Sven Keidel and Sebastian Erdweg. 2019. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 176:1–176:28. https://doi.org/10.1145/3360602

[11] Bastian Kruck, Stefan Lehmann, Christoph Keßler, Jakob Reschke, Tim Felgentreff, Jens Lincke, and Robert Hirschfeld. 2016. Multi-level debugging for interpreter developers. In *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*, Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki (Eds.). ACM, 91–93. https://doi.org/10.1145/2892664.2892679

[12] Bastian Kruck, Tobias Pape, Tim Felgentreff, and Robert Hirschfeld. 2017. Crossing abstraction barriers when debugging in dynamic languages. In *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng (Eds.). ACM, 1498–1504. https://doi.org/10.1145/3019612.3019734

[13] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*. https://www.bodden.de/pubs/lblh11soot.pdf

[14] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2020. Framework-aware debugging with stack tailoring. In *DLS 2020: Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages, Virtual Event, USA, November 17, 2020*, Matthew Flat (Ed.). ACM, 71–84. https://doi.org/10.1145/3426422.3426982

[15] Cameron Moy, Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2021. Corpse reviver: sound and efficient gradual typing via contract verification. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434334

[16] Jens Nicolay, Quentin Stiévenart, Wolfgang De Meuter, and Coen De Roover. 2019. Effect-Driven Flow Analysis. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11388)*, Constantin Enea and Ruzica Piskac (Eds.). Springer, 247–274. https://doi.org/10.1007/978-3-030-11245-5_12

[17] Olin Grigsby Shivers. 1991. *Control-flow analysis of higher-order languages or taming lambda.* Carnegie Mellon University.

[18] Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. 2019. A general method for rendering static analyses for diverse concurrency models modular. *Journal of Systems and Software* 147 (jan 2019), 17–45. https://doi.org/10.1016/j.jss.2018.10.001

[19] Bram Vandenbogaerde, Quentin Stiévenart, and Coen De Roover. 2022. Summary-Based Compositional Analysis for Soft Contract Verification. In *22nd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Limassol, Cyprus, October 3, 2022.* IEEE, 186–196. https://doi.org/10.1109/SCAM55253.2022.00028

[20] Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged abstract interpreters: fast and modular whole-program analysis via metaprogramming. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 126:1–126:32. https://doi.org/10.1145/3360552

Received 2023-07-07; accepted 2023-09-01