Effect-driven Flow Analysis

Nicolay, Jens; Stiévenart, Quentin; De Meuter, Wolfgang; De Roover, Coen

Published in:
Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Proceedings

DOI:
10.1007/978-3-030-11245-5_12

Publication date:
2019

License:
Unspecified

Document Version:
Accepted author manuscript

Effect-driven Flow Analysis

Jens Nicolay, Quentin Stiévenart, Wolfgang De Meuter, and Coen De Roover

Software Languages Lab, Vrije Universiteit Brussel, Belgium
{jnicolay,qstieven,wdmeuter,cderoove}@vub.ac.be

Abstract. Traditional machine-based static analyses use a worklist algorithm to explore the analysis state space, and compare each state in the worklist against a set of seen states as part of their fixed-point computation. This may require many state comparisons, which gives rise to a computational overhead. Even an analysis with a global store has to clear its set of seen states each time the store updates because of allocation or side-effects, which results in more states being reanalyzed and compared.
In this work we present a static analysis technique, MODF, that does not rely on a set of seen states, and apply it to a machine-based analysis with global-store widening. MODF analyzes one function execution at a time to completion while tracking read, write, and call effects. These effects trigger the analysis of other function executions, and the analysis terminates when no new effects can be discovered.
We compared MODF to a traditional machine-based analysis implementation on a set of 20 benchmark programs and found that MODF is faster for 17 programs with speedups ranging between 1.4x and 12.3x. Furthermore, MODF exhibits similar precision as the traditional analysis on most programs and yields state graphs that are comparable in size.

Keywords: program analysis · static analysis · abstract interpretation · effects

1 Introduction

1.1 Motivation

Traditional machine-based analyses [25] use a worklist algorithm to explore the analysis state space. The worklist contains the program states that still have to be explored by the fixed-point computation. In order to reach a fixed point, every state that is pulled from the worklist has to be checked against a set of seen states. If the state was already analyzed, then it must not be reanalyzed to ensure termination of the analysis.

Comparing states in this manner gives rise to a computational overhead, especially if the store is contained in the program states. To accelerate the fixed-point computation, global-store widening can be applied [22]. Global-store widening lifts the store out of individual states and turns it into a global analysis component, making it a shared component of each state that the analysis produces. This reduces the number of times states have to be reanalyzed and compared, and state comparison itself also becomes cheaper.

Yet, despite improved tractability, an analysis with a global store may still require many state comparisons. Moreover, each time the global store is updated, the set of seen states has to be cleared because all states explored so far were computed with a previous version of the store that is different from the latest one. Although clearing the set of seen states makes checking for membership of this set cheaper, naive approaches do so in an indiscriminate manner. Seen states that are not dependent on a particular store update will still be removed from the set of seen states, and will be reanalyzed without the analysis discovering new information (new states, new store updates, ...). This causes the analysis to reanalyze and compare states unnecessarily.

Therefore, the impact of maintaining a set of seen states on the performance, which is unpredictable in general, motivated the following two research questions.

1. *How to design a static analysis that does not require a set of seen states to reach a fixed point?* and
2. *What are the implications on performance and precision when compared to a traditional technique?*

As the set of seen states plays an important role in ensuring termination of the analysis, our answer focuses on its fixed-point mechanism while assuming regular semantics and configurability (lattices, context-sensitivity, ...).

1.2 Approach

In this work we present a static analysis technique for higher-order, side-effecting programs, called MODF, that does not rely on a set of seen states for computing a fixed point. MODF analyzes one single function execution at a time to completion while tracking read, write, and call effects. These effects trigger the analysis of other function executions, and the analysis terminates when no new effects can be discovered. This means that, unlike existing analyses that rely on effects, MODF uses the effects discovered during analysis to drive the analysis itself. The result of the analysis is a flow graph that can be queried by client analyses for obtaining information about fundamental program properties such as control flow and value flow.

Whenever during a function execution another function is called, a call effect is generated and the cached return value of the called function is retrieved from the store and used as return value. If it is the first call to a particular function, then the called function is added to the worklist for future analysis. Whenever a function reads from the global store, this read dependency is tracked. Upon a write to an address in the store, all read-dependent functions are added to the worklist for reanalysis. When a function returns, its return value is written to the store using the function as the address. Calls beyond the initial one to a particular function do not by themselves trigger that function's reanalysis. Because both the arguments and the return value are stored, writing the argument values at call time and writing the return value at return time ensures that the required (dependent) functions are reanalyzed, thereby honoring the call/return pattern.

By not relying on a set of seen states, MODF avoids the associated state comparisons, and by tracking read and write effects MODF is more selective in reanalyzing program states. The goal of this design is to remove an important source of overhead in naive implementations of machine-based techniques such as AAM [25]. In addition, caching of return values in combination with selective reanalysis acts as a memoization mechanism that a MODF analysis can benefit from.

We applied MODF to a traditional AAM analyzer with global-store widening, and our evaluation shows that for many benchmark programs the MODF analyzer is indeed faster while maintaining precision (Section 4).

Contributions

- The formal definition of a function-modular static analysis design for higher-order, side-effecting programs that does not rely on a set of seen states (MODF).
- The application of MODF to AAM, a well-known and widely used analysis approach.
- The implementation of an AAM-based MODF analyzer and its evaluation in terms of performance and precision.

Overview of the Paper We first introduce MODF (Section 2) and formalize our approach (Section 3). We then compare a MODF analyzer to an AAM analyzer in terms of performance and precision (Section 4). We discuss related work (Section 5) and conclude by presenting open questions for future research (Section 6).

2 Overview of the Approach

We illustrate MODF through a number of examples involving (recursive) function calls, higher-order functions, and side-effects.

Function execution occurs in a particular execution *context*, and the combination of a function `f` and its execution context is denoted by $\kappa_{\texttt{f}}$. The program itself is executed in an initial context denoted by κ_0, but in the remainder of this paper we treat it like any other function execution context and refer to it as such (conceptually a program can be thought of as representing the body of an implicit main function that is called to start the exection of the program). MODF analyzes each function execution separately from other executions, tracking function calls and accesses to the global store. When an address is read from the store, a *read effect* is generated by the analysis: $\mathbf{r}(\texttt{x})$ denotes a read effect on variable `x` (we use variable names as addresses for simplicity). Similarly, when a value is added or modified in the store, a *write effect* is generated: $\mathbf{w}(\texttt{x})$ denotes a write effect on variable `x`. MODF does not immediately step into a function when it is called,

but rather models these calls through *call effects*: $\mathbf{c}(\kappa)$ denotes a call effect on the context κ. Tracking read, write, and call effects enables detecting how changes made by the execution of one function affect other function executions.

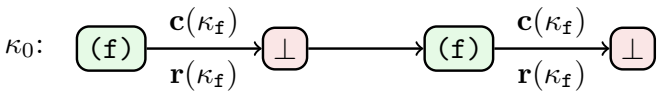
2.1 Simple Function

Consider the following Scheme program, which defines a function `f` and calls it twice.

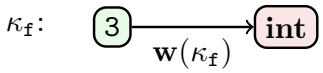
```
(define (f) 3)
(f)
(f)
```

MODF starts by analyzing the initial context κ_0. When encountering the first call to `f`, MODF produces a call effect $\mathbf{c}(\kappa_f)$ and looks up the return value of `f` in the store at address $\kappa_\mathtt{f}$ (contexts are used as addresses in the store). Because `f` has not yet produced a return value, the lookup results in the bottom value $\bot$, which denotes the absence of information. Looking up this return value produces a read effect $\mathbf{r}(\kappa_\mathtt{f})$. The presence of this effect results in κ_0 having a read dependency on address $\kappa_\mathtt{f}$.

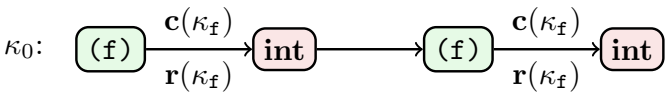
The second call to `f` is treated in the same manner, so that context κ_0 is now fully analyzed. This is represented by the following graph, where green nodes correspond to expressions that have to be evaluated by the program, and red nodes correspond to values reached by evaluating the preceding green node. The edges correspond to transitions in the evaluation of the program and may be annotated with one or more effects. For clarity, we omit some of the read effects in the graphs that follow.



Because the analysis of context κ_0 yielded a function call with context κ_f, and context κ_f was not encountered before, MODF proceeds by analyzing it. This produces an abstract return value **int** (assuming a type lattice as the abstract domain for values), which is written in the store at location κ_f, thereby producing a write effect $\mathbf{w}(\kappa_f)$.



Because context $\kappa_\mathtt{f}$ updates address $\kappa_\mathtt{f}$, and κ_0 has a read dependency on this address, the analysis of the context κ_0 is retriggered with the updated store, during which the resulting values of function calls to `f` are now correctly resolved. No new effects are discovered.



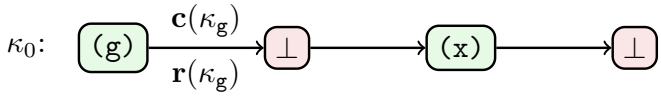
Because all discovered contexts have been analyzed and no new store-changing effects were detected, MODF has reached a fixed point.

2.2 Higher-order Function

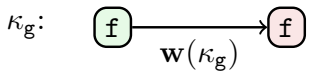
The following example illustrates the analysis of higher-order functions. Function `g` returns a closure `f` which is called on the last line.

```
(define (f) 3)
(define (g) f)
(define x (g))
(x)
```

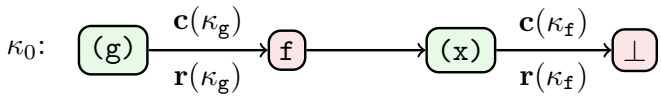
The first iteration of MODF analyzes the initial context κ_0 and detects the call to function `g`, immediately followed by the assignment of value $\bot$ to variable `x` because no return value was previously computed for `g`. The call to variable `x` therefore results in a $\bot$ value as well.



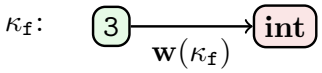
In the next iteration MODF analyzes context κ_g, as it was encountered for the first time during the previous iteration. The analysis detects that `g` returns function `f`, and this return value is stored at address κ_g.



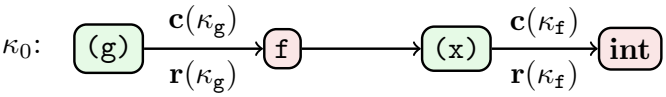
The third iteration reanalyzes κ_0 as one of the addresses read by this context (κ_g) has been written to. The value of variable `x` now is function `f`, and a call effect is generated on context κ_f that immediately returns value $\bot$ because `f` has not been analyzed previously.



The fourth iteration analyzes newly discovered context κ_f and discovers abstract return value **int**, which is stored at address κ_f, generating a write effect.



The fifth and final iteration reanalyzes the initial context, for which the call `(x)` produces the return value **int** residing at address κ_f, and MODF reaches a fixed point.



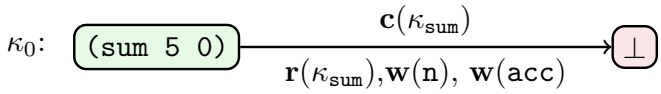
Although this example only considers a program in which a function is returned by another function, MODF supports closures as arguments to or return values of function calls.

2.3 Recursion and Function Arguments

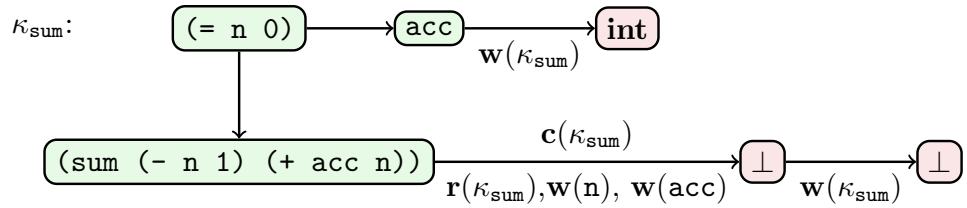
The next example features a recursive function `sum` which computes the sum of natural numbers between 1 and `n`.

```
(define (sum n acc)
  (if (= n 0)
    acc
    (sum (- n 1) (+ acc n))))
(sum 5 0)
```

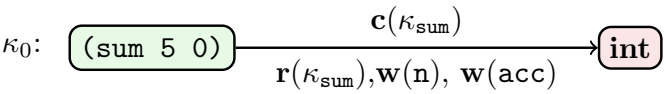
First, the initial context κ_0 is analyzed, and the call to `sum` immediately results in value $\bot$, generating a call effect for context $\kappa_{\texttt{sum}}$. During this iteration (i.e., at the call site), the analysis binds the argument values in the store, generating the corresponding write effects. The store itself is global to the analysis, and only grows monotonically. In our examples we use the name of a variable as the address at which we store its value, so that a particular function parameter is always stored at the same address and multiple calls to `sum` cause the different arguments values to become joined in the store.



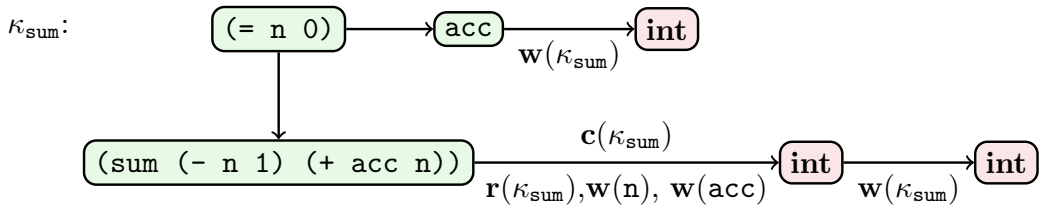
The second iteration of MODF proceeds with the analysis of context $\kappa_{\texttt{sum}}$ and the possible return value **int**, stemming from expression `acc` in the then-branch, is detected. The value corresponding to the recursive call in the else-branch is $\bot$ as the address $\kappa_{\texttt{sum}}$ is not yet bound in the store at this point.



In the third iteration, the analysis has to consider either κ_0 or $\kappa_{\texttt{sum}}$, because both contexts have a read dependency on address $\kappa_{\texttt{sum}}$, which was written during the previous iteration. Although the order in which contexts are analyzed may influence the convergence speed of a MODF analysis, it will not influence its end result. For this example's sake, we assume MODF reanalyzes κ_0 first, in which the call to `sum` now produces return value **int** by reading address $\kappa_{\texttt{sum}}$ in the store.



Finally, context $\kappa_{\texttt{sum}}$ is reanalyzed, and the recursive call now also results in the expected abstract value **int**. Because the return value of the `sum` function was already determined to be **int**, the store does not change and no additional contexts have to be analyzed, concluding the analysis.

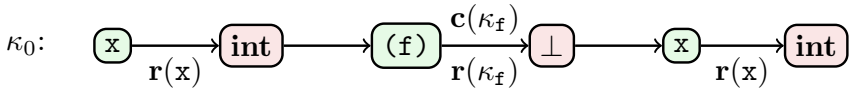


2.4 Mutable state

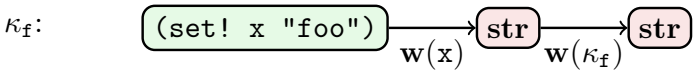
In this final example we mutate state using the `set!` construct. Variable `x` in the program below initially holds an integer value. After evaluating `x` a first time, function `f` is called, which changes the value of `x` to a string, and variable `x` is evaluated again.

```
(define x 0)
(define (f) (set! x "foo"))
(display x)
(f)
(display x)
```

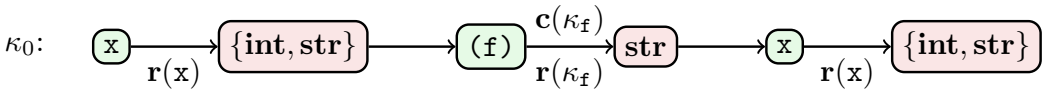
A first analysis of initial context κ_0 correctly infers the first value for `x` (**int**) but incorrectly infers its second value as execution context $\kappa_{\texttt{f}}$ has not yet been analyzed. However, the read dependency of κ_0 on the address of variable `x` is inferred as a read effect.



In the next iteration, context $\kappa_{\texttt{f}}$ is analyzed. It writes to the address of variable `x` and to return location $\kappa_{\texttt{f}}$.



Finally, the reanalysis of context κ_0 is triggered due to changes on addresses on which it depends (`x` and $\kappa_{\texttt{f}}$), and a sound over-approximation of the possible value of `x` is obtained.



3 Formal Definition

We formally define MODF for a higher-order, side-effecting language as a process that alternates between two phases:

1. An *intra-context* analysis analyzes a function execution context given an input store, and tracks the effects performed within this context.
2. An *inter-context* analysis triggers subsequent intra-context analysis based on the effects observed during previous intra-context analyses.

Before introducing these analysis phases, we provide the concrete operational semantics of the language under analysis.

3.1 Input Language

As input language for the formal definition of MODF, we use the untyped λ-calculus in A-Normal Form with support for side-effects through `set!`. A-Normal Form, or ANF, is a restricted syntactic form for λ-calculus programs in which operators and operands are restricted to *atomic expressions*. Atomic expressions *ae* are expressions that can be evaluated immediately without impacting the program state, as opposed to non-atomic expressions that may impact the program state. This syntactic simplification of the language does not limit its expressiveness, as any λ-calculus program can be automatically rewritten into its A-Normal Form [4]. We assume the domain of expressions (Exp) to be finite, as any program contains a finite number of expressions and we only consider expressions appearing in the analyzed program.

We include atomic expressions that denote integer and string primitive values in the language for consistency with the examples given previously and to illustrate that they do not present any complications with respect to the analysis. Other primitive values can be added in a similar fashion.

The `set!` operator modifies the value of a variable x to the value resulting from the evaluation of an atomic expression *ae*. While the presentation in this paper focuses on a functional language with only `set!` as an imperative construct, nothing prevents the MODF approach from being applied to languages with other and additional imperative constructs.

$$
\begin{aligned}
e \in \mathsf{Exp} &::= ae \mid \texttt{(}f\ ae\texttt{)} \\
&\mid \texttt{(set! }x\ ae\texttt{)} \\
&\mid \texttt{(let ((}x\ e\texttt{))}\ e\texttt{)} \\
f, ae \in \mathsf{AExp} &::= x \mid lam \mid n \mid s
\end{aligned}
\qquad
\begin{aligned}
lam \in \mathsf{Lam} &::= \texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e\texttt{)} \\
x &\in \mathsf{Var} \text{ a finite set of identifiers} \\
n &\in \mathbb{Z} \text{ the set of finite integers} \\
s &\in \mathbb{S} \text{ the set of finite strings}
\end{aligned}
$$

3.2 Concrete Semantics

The concrete semantics of the input language is defined as a transition relation, denoted $\varsigma, \sigma \to \varsigma', \sigma'$. It acts on a state ς and a store σ, producing a successor state and store.

State space. A state is composed of a control component c, which can either contain an expression to evaluate in an environment ($\mathbf{ev}(e, \rho)$) or a value ($\mathbf{val}(v)$), and a stack ι, which itself is a sequence of frames representing the continuation of the execution. The values (v) in this language are primitive values such as integers (**int**) and strings (**str**), and closures (**clo**) that bind lambda-expressions with their defining environments. Environments map variables to addresses, and stores map addresses to values. We leave addresses undefined for the sake of generality, but we assume that there are infinitely many concrete addresses.

$$
\begin{aligned}
\varsigma \in \Sigma &::= \langle c, \iota \rangle \\
c \in \mathit{Control} &::= \mathbf{ev}(e, \rho) \mid \mathbf{val}(v) \\
v \in \mathit{Val} &::= \mathbf{clo}(lam, \rho) \mid \mathbf{int}(n) \mid \mathbf{str}(s) \\
\iota \in \mathit{Stack} &::= \phi : \iota \mid \epsilon
\end{aligned}
\qquad
\begin{aligned}
\phi \in \mathit{Frame} &::= \mathbf{let}(a, e, \rho) \\
\rho \in \mathit{Env} &= \mathsf{Var} \to \mathit{Addr} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \to \mathit{Val} \\
a \in \mathit{Addr} &\text{ an infinite set of addresses}
\end{aligned}
$$

Atomic evaluation. Atomic expressions are evaluated by the atomic evaluation function $\mathcal{A} : \mathsf{AExp} \times \mathit{Env} \times \mathit{Store} \to \mathit{Val}$ that, given an environment and a store, computes the value of the atomic expression. Variable references x are evaluated by looking up the address of that variable in the environment and returning the value that resides in the store at that address. The evaluation of a lambda expression results in a closure that pairs the lambda expression with the current environment. Integers and strings are tagged with their respective type during atomic evaluation.

$$\mathcal{A}(x, \rho, \sigma) = \sigma(\rho(x)) \qquad \mathcal{A}(lam, \rho, \sigma) = \mathbf{clo}(lam, \rho) \qquad \mathcal{A}(n, \rho, \sigma) = \mathbf{int}(n) \qquad \mathcal{A}(s, \rho, \sigma) = \mathbf{str}(s)$$

Transition relation. The transition relation is defined using 5 rules.

Function $\mathcal{A}$ is used by the transition relation to evaluate an atomic expression into a value, leaving the stack and the store unmodified.

$$\frac{v = \mathcal{A}(ae, \rho, \sigma)}{\langle \mathbf{ev}(ae, \rho), \iota \rangle, \sigma \to \langle \mathbf{val}(v), \iota \rangle, \sigma}$$

For a function call, first the operator f and the operand *ae* are evaluated atomically. Evaluation then continues by stepping into the body e' of the called function with the environment and store extended with the value v of the argument x at a fresh address generated by the allocation function $\mathit{alloc} : \mathsf{Var} \to \mathit{Addr}$.

$$\frac{\mathbf{clo}(\texttt{(}\lambda\texttt{(}x\texttt{)}e'\texttt{)}, \rho') = \mathcal{A}(f, \rho, \sigma) \qquad v = \mathcal{A}(ae, \rho, \sigma) \qquad a = \mathit{alloc}(x) \qquad \rho'' = \rho'[x \mapsto a]}{\langle \mathbf{ev}(\texttt{(}f\ ae\texttt{)}, \rho), \iota \rangle, \sigma \to \langle \mathbf{ev}(e', \rho''), \iota \rangle, \sigma[a \mapsto v]}$$

A `set!` expression is evaluated by first evaluating the atomic expression ae to obtain the new value v for x, and then updating the value of x in the store.

$$\frac{v = \mathcal{A}(ae, \rho, \sigma)}{\langle \mathbf{ev}(\texttt{(set! } x\ ae\texttt{)}, \rho), \iota \rangle, \sigma \to \langle \mathbf{val}(v), \iota \rangle, \sigma[\rho(x) \mapsto v]}$$

A `let` expression is evaluated in two steps. A first rule pushes a continuation on the stack and evaluates the expression for which x has to be bound to the result. The environment is extended at this point to enable recursion, so that a function can refer to itself in its body (meaning that this `let` is equivalent to Scheme's `letrec`).

$$\frac{a = alloc(x) \qquad \rho' = \rho[x \mapsto a] \qquad \iota' = \mathbf{let}(a, e_2, \rho') : \iota}{\langle \mathbf{ev}(\texttt{(let ((}x\ e_1\texttt{)) } e_2\texttt{)}, \rho), \iota \rangle, \sigma \to \langle \mathbf{ev}(e_1, \rho'), \iota' \rangle, \sigma}$$

A second rule then acts when a value has been computed for the variable x by evaluating the body of the `let` after binding the address of x to its value in the store.

$$\frac{\sigma' = \sigma[a \mapsto v]}{\langle \mathbf{val}(v), \mathbf{let}(a, e, \rho) : \iota \rangle, \sigma \to \langle \mathbf{ev}(e, \rho), \iota \rangle, \sigma'}$$

This completes the rules for the concrete semantics of the input language.

Allocation. The *alloc* function for allocating addresses in the store is a parameter of the analysis. For defining concrete semantics, natural numbers can be used as concrete addresses, i.e., we take $Addr = \mathbb{N}$ and have *alloc* generate fresh addresses each time it is called.

Collecting Semantics. The concrete collecting semantics of a program e is the set of states that the program may reach during its execution. It is defined as the fixed point of a transfer function $\mathcal{F}^e : \mathcal{P}(\Sigma \times Store) \to \mathcal{P}(\Sigma \times Store)$, where function $\mathcal{I}(e) : \mathsf{Exp} \to \Sigma$ injects a program represented by an expression into an initial state.

$$\mathcal{F}^e(S) = \{(\mathcal{I}(e), [])\} \cup \bigcup_{\substack{\varsigma, \sigma \in S \\ \varsigma, \sigma \to \varsigma', \sigma'}} (\varsigma', \sigma') \qquad\qquad \mathcal{I}(e) = \langle \mathbf{ev}(e, []), \epsilon \rangle$$

In the remainder of this section, we discuss approaches to soundly over-approximate the set of concrete states formed by $\mathrm{lfp}(\mathcal{F}^e)$. First, we abstract all elements of the state space except for the stack (Section 3.3), resulting in an infinite abstract interpretation. We then examine and compare different existing approaches for abstracting the stack and introduce and discuss the approach taken by MODF (Section 3.4).

3.3 Abstracting Values

Similar to Earl et al. [3], we perform a first abstraction of the concrete semantics to obtain a baseline abstract interpretation to illustrate the similarities and differences between MODF and related work. In this abstraction, we render the set of values and the set of addresses finite but leave the stack untouched.

State space. This first abstraction consists of rendering the set of addresses finite, which implies that stores now map to *sets of values* rather than a single value. To ensure that the set of values is finite, we also abstract primitive values into their type, although other finite abstractions would work as well. As a result, all components of this state space are finite with the exception of the stack, which can grow infinitely (something we address in the next sections). We highlight the main changes in the formalism in gray.

$$\varsigma \in \Sigma ::= \langle c, \iota \rangle$$
$$c \in Control ::= \mathbf{ev}(e, \rho) \mid \mathbf{val}(\{v, \dots\})$$
$$v \in Val ::= \mathbf{clo}(lam, \rho) \mid \mathbf{int} \mid \mathbf{str}$$
$$\iota \in Stack ::= \phi : \iota \mid \epsilon$$
$$\phi \in Frame ::= \mathbf{let}(a, e, \rho)$$
$$\rho \in Env = \mathsf{Var} \to Addr$$
$$\sigma \in Store = Addr \to \mathcal{P}(Val)$$
$$a \in Addr \text{ a finite set of addresses}$$

Atomic evaluation. The changes in the state space propagate to the atomic evaluation function. The atomic evaluation function $\mathcal{A} : \mathsf{AExp} \times \mathit{Env} \times \mathit{Store} \to \mathcal{P}(\mathit{Val})$ now evaluates to a set of abstract values, losing information about concrete values of integers and strings.

$$\mathcal{A}(x, \rho, \sigma) = \sigma(\rho(x)) \qquad \mathcal{A}(\mathit{lam}, \rho, \sigma) = \{\, \mathbf{clo}(\mathit{lam}, \rho) \,\} \qquad \mathcal{A}(n, \rho, \sigma) = \{\mathbf{int}\} \qquad \mathcal{A}(s, \rho, \sigma) = \{\mathbf{str}\}$$

Transition relation. The rules of the transition relation are updated to account for the changes in the store and atomic evaluation function. Because multiple values can be bound at the same address in the store, it is crucial that store updates become store *joins* instead for the sake of soundness. Store joins are defined as the pointwise lift of the join operation of abstract values (which in our case, is the set union). To avoid unnecessary non-determinism, we introduce the $V \in \mathcal{P}(\mathit{Val})$ metavariable to denote a set of values.

$$\frac{V = \mathcal{A}(ae, \rho, \sigma)}{\langle \mathbf{ev}(ae, \rho), \iota \rangle, \sigma \to \langle \mathbf{val}(V), \iota \rangle, \sigma}$$

$$\frac{\mathbf{clo}(\texttt{(}\lambda\texttt{(}x\texttt{)}e'\texttt{)}, \rho') \in \mathcal{A}(f, \rho, \sigma) \qquad V = \mathcal{A}(ae, \rho, \sigma) \qquad a = \mathit{alloc}(x) \qquad \rho'' = \rho'[x \mapsto a]}{\langle \mathbf{ev}(\texttt{(}f\ ae\texttt{)}, \rho), \iota \rangle, \sigma \to \langle \mathbf{ev}(e', \rho''), \iota \rangle, \sigma \sqcup [a \mapsto V]}$$

$$\frac{V = \mathcal{A}(ae, \rho, \sigma)}{\langle \mathbf{ev}(\texttt{(set! }x\ ae\texttt{)}, \rho), \iota \rangle, \sigma \to \langle \mathbf{val}(V), \iota \rangle, \sigma \sqcup [\rho(x) \mapsto V]}$$

$$\frac{a = \mathit{alloc}(x) \qquad \rho' = \rho[x \mapsto a] \qquad \iota' = \mathbf{let}(a, e_2, \rho') : \iota}{\langle \mathbf{ev}(\texttt{(let ((}x\ e_1\texttt{))}\ e_2\texttt{)}, \rho), \iota \rangle, \sigma \to \langle \mathbf{ev}(e_1, \rho'), \iota' \rangle, \sigma} \qquad \frac{\sigma' = \sigma \sqcup [a \mapsto V]}{\langle \mathbf{val}(V), \mathbf{let}(a, e, \rho) : \iota \rangle, \sigma \to \langle \mathbf{ev}(e, \rho), \iota \rangle, \sigma'}$$

Allocation and context sensitivity. The choice for the abstraction of the set of addresses and for the definition of the *alloc* function influences the context-sensitivity of the abstract interpretation. For example, abstracting addresses to variable names ($\mathit{Addr} = \mathsf{Var}$, with $\mathit{alloc}(x) = x$) results in a context-insensitive 0-CFA analysis. Any allocation function is sound [12], and it is possible to introduce more precise context-sensitivities. We refer to Gilray et al. [7] for a discussion of the impact of allocation on the precision of analyses.

Collecting semantics. Using the abstraction defined here, the abstract collecting semantics of a program e is defined similarly to the concrete collecting semantics. The abstract transfer function $\mathcal{F}^e$ uses the abstract transition relation instead of the concrete one.

$$\mathcal{F}^e(S) = \{(\mathcal{I}(e), [])\} \cup \bigcup_{\substack{\varsigma, \sigma \in S \\ \varsigma, \sigma \to \varsigma', \sigma'}} (\varsigma', \sigma')$$

The fixed point of the abstract transfer function defines the abstract collecting semantics. However, the set defined by $\mathrm{lfp}(\mathcal{F}^e)$ may not be computable as we have not performed abstraction of the stack. It is therefore not suitable for static analysis without additional abstractions as discussed in the following sections.

3.4 Abstracting the Stack

We left the stack untouched until now, which means it can grow infinitely, resulting in an abstract interpretation that may not terminate. Multiple approaches for abstracting the stack have been proposed, which we summarize here before detailing our own approach in MODF.

AAM Abstraction AAM (Abstracting Abstract Machines [25]) is a technique for finitely abstracting machine-based interpreters. AAM eliminates potentially infinite recursion by allocating recursive components in the store and—as we did in Section 3.3—making the set of store addresses finite. With a finite number of addresses in the store the store cannot grow infinitely, and therefore AAM provides a suitable foundation for static analysis. The solution proposed in AAM to abstract the stack therefore is to thread the potentially infinite sequence of stack frames through the store, rendering the stack finite.

State space. The components of the state space that require adaptation are the store, which now also contains stacks, and the stacks themselves, which contain at most a single continuation frame and the address at which the rest of the stack resides in the store. To differentiate between stack addresses and value addresses in the store, we introduce contexts (κ) to represent stack addresses.

$$
\begin{aligned}
\varsigma \in \Sigma &::= \langle c, \iota \rangle \\
c \in \mathit{Control} &::= \mathbf{ev}(e, \rho) \mid \mathbf{val}(\{v, \dots\}) \\
v \in \mathit{Val} &::= \mathbf{clo}(\mathit{lam}, \rho) \mid \mathbf{int} \mid \mathbf{str} \\
\iota \in \mathit{Stack} &::= \phi : \kappa \mid \epsilon \\
\phi \in \mathit{Frame} &::= \mathbf{let}(a, e, \rho)
\end{aligned}
\qquad
\begin{aligned}
\rho \in \mathit{Env} &= \mathsf{Var} \to \mathit{Addr} \\
\sigma \in \mathit{Store} &= (\mathit{Addr} + K) \to \mathcal{P}(\mathit{Val} + \mathit{Stack}) \\
a \in \mathit{Addr} &\text{ a finite set of addresses} \\
\kappa \in K &\text{ a finite set of contexts}
\end{aligned}
$$

Transition relation. The only rules of the transition relation impacted by these changes are the rules that push and pop continuation frames from the stack.

To evaluate a `let` binding, a continuation frame is pushed onto the stack. First a stack address is allocated using stack allocation function *allocCtx*. The store can then be extended at the address returned by the stack allocator to contain the current continuation, and a new stack is used in the resulting state. When the **let** continuation has to be popped from the stack, the rest of the stack is looked up in the store at address κ.

$$\frac{V = \mathcal{A}(ae, \rho, \sigma)}{\langle \mathbf{ev}(ae, \rho), \iota \rangle, \sigma \rightsquigarrow \langle \mathbf{val}(V), \iota \rangle, \sigma}$$

$$\frac{\mathbf{clo}((\lambda(x) e'), \rho') \in \mathcal{A}(f, \rho, \sigma) \qquad V = \mathcal{A}(ae, \rho, \sigma) \qquad a = \mathit{alloc}(x) \qquad \rho'' = \rho'[x \mapsto a]}{\langle \mathbf{ev}((f\ ae), \rho), \iota \rangle, \sigma \rightsquigarrow \langle \mathbf{ev}(e', \rho''), \iota \rangle, \sigma \sqcup [a \mapsto V]}$$

$$\frac{V = \mathcal{A}(ae, \rho, \sigma)}{\langle \mathbf{ev}((\texttt{set!}\ x\ ae), \rho), \iota \rangle, \sigma \rightsquigarrow \langle \mathbf{val}(v), \iota \rangle, \sigma \sqcup [\rho(x) \mapsto V]}$$

$$\frac{a = \mathit{alloc}(x) \qquad \rho' = \rho[x \mapsto a] \qquad \iota' = \mathbf{let}(a, e_2, \rho') : \kappa \qquad \kappa = \mathit{allocCtx}(e_1, \rho) \qquad \sigma' = \sigma \sqcup [\kappa \mapsto \{\iota\}]}{\langle \mathbf{ev}((\texttt{let}\ ((x\ e_1))\ e_2), \rho), \iota \rangle, \sigma \rightsquigarrow \langle \mathbf{ev}(e_1, \rho'), \iota' \rangle, \sigma'}$$

$$\frac{\sigma' = \sigma \sqcup [a \mapsto V] \qquad \iota \in \sigma(\kappa)}{\langle \mathbf{val}(V), \mathbf{let}(a, e, \rho) : \kappa \rangle, \sigma \rightsquigarrow \langle \mathbf{ev}(e, \rho), \iota \rangle, \sigma'}$$

Allocation and context sensitivity. Like function *alloc*, function *allocCtx* is a parameter of the analysis that can be used to influence the context sensitivity of the analysis. To facilitate comparison with MODF (see Section 3.4), we take $K = \mathsf{Exp} \times \mathit{Env}$ and $\mathit{allocCtx} : \mathsf{Exp} \times \mathit{Env} \to K$ with $\mathit{allocCtx}(e, \rho) = (e, \rho)$. With this definition of *allocCtx*, and because environments contain addresses, any context sensitivity introduced by the value address allocator *alloc* will also influence the context-sensitivity of stack addresses.

Collecting semantics. The abstract transfer function for computing the abstract collecting semantics needs to be adapted to use the transition relation developed here. In contrast to the abstract transfer function of Section 3.3, the resulting abstract collecting semantics is guaranteed to be computable, as the abstraction

is finite. This therefore results in an abstract interpretation suitable for a static analysis, although the resulting analysis may not be efficient. If the fixed point is computed with the usual fixed-point computation techniques, such as the commonly used worklist algorithm [19], this requires maintaining a set of states that have been visited in order to avoid re-visiting a state more than once. This in turn ensures that the algorithm terminates.

Because traditional, unoptimized AAM is inefficient, a common performance optimization technique is widening of the global store [25]. Global-store widening widens all reachable stores into a single global store, in effect removing the store as component from individual states.

$$\mathcal{F}^e(S,\sigma) = (\{\mathcal{I}(e)\},[]) \cup (\bigcup_{\substack{\varsigma\in S\\ \varsigma,\sigma\rightsquigarrow\varsigma',\sigma'}} \varsigma', \quad \bigsqcup_{\substack{\varsigma\in S\\ \varsigma,\sigma\rightsquigarrow\varsigma',\sigma'}} \sigma')$$

Although global store-widening improves the performance of the analysis (at the cost of precision), the fixed-point computation of the transfer function using a worklist algorithm has to carefully clear the set of seen states when changes are performed on the global store, as the store is shared with all explored states and changes therefore may impact states that have already been visited. While clearing this set of seen states is crucial for soundness, it does however cause a significant cost, as many of the states present in the set of seen states may not be impacted by the store changes, but will still have to be visited. Our experiments, described in Section 4 and in which we observed that 80% of the reanalyzed states in AAM do not yield new states, confirm this. The worst-case complexity of AAM for computing the context-insensitive 0-CFA analysis is $O(n^3)$ [25], with n representing the size of the program.

Pushdown Abstraction CFA2 [26] and PDCFA [3] are two approaches that use a pushdown automaton instead of a finite state machine to approximate the behavior of a program. However, besides requiring significant engineering effort, using these two techniques as the foundation for static analysis yields a computational costs in $O(2^n)$ for CFA2 and in $O(n^6)$ for PDCFA, resp. [8]. Additionally, CFA2 only supports programs that are written in continuation-passing style.

AAC Abstraction Johnson et al. [10] proposes a variation on stack abstraction found in AAM called AAC (Abstracting Abstract Control). AAC does not allocate stacks in the value store, but instead introduces a different "stack store" for this purpose. This enables the allocation of stack addresses that consist of all components that influence the outcome of a procedure application (assuming the absence of first-class control), i.e., the entire calling context including the value store. This in turn leads to full call/return precision under a given value abstraction, but without requiring the complex implementation of pushdown automata. In AAC the continuation is also split into a local continuation and a meta-continuation. The local continuation represents the intraprocedural stack and is represented as a sequence of frames, i.e., with maximal precision. A local continuation is bounded by a meta-continuation which is allocated in the store at function calls and therefore represents the interprocedural (call) stack. Although AAC offers high precision, the worst-case computational cost of a context-insensitive AAC flow analysis [10] was found to be $O(n^8)$ [8], where n is the size of the input program. P4F [8] is the name for the technique of choosing AAC stack addresses consisting only of an expression and an environment. While this results in a reduced computational cost of $O(n^3)$ for a context-insensitive analysis, maximal (pushdown) call/return precision for side-effecting programs is lost because the store is not a component of the stack address.

Modular Abstraction Rather than abstracting the stack, MODF, the approach presented in this paper, modifies the fixed-point computation to ensure that the stack cannot grow infinitely. Similarly to AAC, during the execution of the body of a function, the stack is modeled as a sequence of frames that can grow. By modifying the fixed-point computation to compute local fixed points for each function call, and because the only means of looping in the input language is recursion, the stack cannot grow infinitely, ensuring the termination of the abstract semantics with MODF. In the next section we describe changes made to the abstract semantics introduced in Section 3.3 in order to obtain a MODF analysis.

3.5 Intra-Context Abstract Semantics for Modf

We first describe changes made to the state space and the abstract transition relation to accomodate for the fixed point computation of Modf.

State space. Contrary to AAM, Modf leaves the stack untouched and preserves its concrete nature. Instead it is the fixed-point computation that ensures that the stack cannot grow infinitely. Two extra components are necessary to approximate the semantics of a program using Modf.

1. Transition relations are annotated with effects (*eff*) in order to denote operations performed during a transition: a write effect ($\mathbf{w}(a)$) indicates that the store has been modified at address a, a read effect ($\mathbf{r}$) indicates that the store has been accessed at address a, and a call effect ($\mathbf{c}(\kappa)$) indicates that there has been a function call to the function denoted by context κ. These effects are used to detect dependencies between the analyzed contexts.
2. The $\mathbf{ret}(\kappa)$ continuation frame is introduced to represent the end of the execution of a function body. Results of function calls are written in the store at addresses that correspond to execution contexts.

$$
\begin{array}{rl@{\qquad}rl}
\varsigma \in \Sigma & ::= \langle c, \iota \rangle & \rho \in \mathit{Env} & = \mathsf{Var} \to \mathit{Addr} \\
c \in \mathit{Control} & ::= \mathbf{ev}(e, \rho) \mid \mathbf{val}(\{v, \dots\}) & \mathit{eff} \in \mathit{Eff} & ::= \mathbf{w}(a) \mid \mathbf{r}(a) \mid \mathbf{c}(\kappa) \\
v \in \mathit{Val} & ::= \mathbf{clo}(\mathit{lam}, \rho) \mid \mathbf{int} \mid \mathbf{str} & \sigma \in \mathit{Store} & = (\mathit{Addr} + K) \to \mathcal{P}(\mathit{Val}) \\
\iota \in \mathit{Stack} & ::= \phi : \iota \mid \epsilon & a \in \mathit{Addr} & \text{ a finite set of addresses} \\
\phi \in \mathit{Frame} & ::= \mathbf{let}(a, e, \rho) \mid \mathbf{ret}(\kappa) & \kappa \in K & \text{ a finite set of contexts}
\end{array}
$$

Atomic evaluation. The atomic evaluation function $\mathcal{A} : \mathsf{AExp} \times \mathit{Env} \times \mathit{Store} \to \mathcal{P}(\mathit{Val}) \times \mathcal{P}(\mathit{Eff})$ may read from the store, and therefore now returns a set of effects indicating whether this was the case.

$$
\begin{array}{rl@{\qquad}rl}
\mathcal{A}(x, \rho, \sigma) &= \sigma(\rho(x)), \{\mathbf{r}(\rho(x))\} & \mathcal{A}(\mathit{lam}, \rho, \sigma) &= \{\mathbf{clo}(\mathit{lam}, \rho)\}, \{\} \\
\mathcal{A}(n, \rho, \sigma) &= \{\mathbf{int}\}, \{\} & \mathcal{A}(s, \rho, \sigma) &= \{\mathbf{str}\}, \{\}
\end{array}
$$

Transition relation. The transition relation is annotated with effects too: $\varsigma, \sigma \overset{E}{\rightsquigarrow} \varsigma', \sigma'$ indicates that the transition has generated the set of effects in E. When evaluating an atomic expression, the transition relation is annotated with the effects generated by the atomic evaluation function.

$$
\frac{V, E = \mathcal{A}(ae, \rho, \sigma)}{\langle \mathbf{ev}(ae, \rho), \iota \rangle, \sigma \overset{E}{\rightsquigarrow} \langle \mathbf{val}(V), \iota \rangle, \sigma}
$$

Function calls are evaluated differently than in the AAM semantics and its variant discussed so far. In Modf each function execution is analyzed in isolation from other function executions. The evaluation of a function call therefore does not step into the body of the called function, but rather generates a call effect $\mathbf{c}$ that will be used by the fixed-point computation to trigger additional intra-context analyses. Immediately after generating a call effect, the return value for the execution context is read from the store, thereby also generating a read effect $\mathbf{r}(\kappa)$. If execution context κ was not analyzed before, then $\sigma(\kappa) = \bot$ and the result of the function call is $\bot$. A function call also results in a write effect $\mathbf{w}(a)$ being generated at address a of the parameter of the function call.

$$
\frac{\begin{array}{c} V_f, E_1 \in \mathcal{A}(f, \rho, \sigma) \qquad \mathbf{clo}((\lambda(x) e'), \rho') \in V_f \qquad V, E_2 = \mathcal{A}(ae, \rho, \sigma) \qquad a = \mathit{alloc}(x) \\ \rho'' = \rho'[x \mapsto a] \qquad \kappa = \mathit{allocCtx}(e', \rho'') \qquad V' = \sigma(\kappa) \qquad E = E_1 \cup E_2 \cup \{\mathbf{w}(a), \mathbf{c}(\kappa), \mathbf{r}(\kappa)\} \end{array}}{\langle \mathbf{ev}((f\ ae), \rho), \iota \rangle, \sigma \overset{E}{\rightsquigarrow} \langle \mathbf{val}(V'), \iota \rangle, \sigma \sqcup [a \mapsto v]}
$$

When evaluating a `set!`, a write effect is generated for the address being modified.

$$\frac{V, E_1 = \mathcal{A}(ae, \rho, \sigma) \qquad E = E_1 \cup \{\mathbf{w}(\rho(x))\}}{\langle \mathbf{ev}(\texttt{(set! } x \text{ } ae\texttt{)}, \rho), \iota \rangle, \sigma \overset{E}{\rightsquigarrow} \langle \mathbf{val}(V), \iota \rangle, \sigma \sqcup [\rho(x) \mapsto V]}$$

Rules for evaluating a `let` remain the same, with the exception that a write effect is generated when the value is bound into the store.

$$\frac{a = alloc(x) \qquad \rho' = \rho[x \mapsto a] \qquad \iota' = \mathbf{let}(a, e_2, \rho') : \iota}{\langle \mathbf{ev}(\texttt{(let ((}x\ e_1\texttt{))}\ e_2\texttt{)}, \rho), \iota \rangle, \sigma \overset{\{\}}{\rightsquigarrow} \langle \mathbf{ev}(e_1, \rho'), \iota' \rangle, \sigma}$$

$$\frac{E = \{\mathbf{w}(a)\} \qquad \sigma' = \sigma \sqcup [a \mapsto V]}{\langle \mathbf{val}(V), \mathbf{let}(a, e, \rho) : \iota \rangle, \sigma \overset{E}{\rightsquigarrow} \langle \mathbf{ev}(e, \rho), \iota \rangle, \sigma'}$$

A new rule is added to account for the **ret** frame, which is reached at the end of a function execution. When a function call with context κ reaches the end of its execution, the resulting value is allocated in the store at address κ and a write effect $\mathbf{w}(\kappa)$ is generated.

$$\frac{E = \{\mathbf{w}(\kappa)\}}{\langle \mathbf{val}(V), \mathbf{ret}(\kappa) : \epsilon \rangle, \sigma \overset{E}{\rightsquigarrow} \langle \mathbf{val}(V), \epsilon \rangle, \sigma \sqcup [\kappa \mapsto v]}$$

Allocation and context sensitivity. Similarly to previous stack abstractions, in Modf the definition of the allocation strategy will influence the context sensitivity of the resulting analysis. While the semantics of Modf is different, the existing allocation strategies for the value store (function *alloc*) developed in the context of AAM [7] can be reused in order to obtain analyses with various context sensitivities.

Modf also requires an allocator for function execution contexts (function *allocCtx*). Unlike in AAM or AAC, however, in Modf contexts are not considered to be stack addresses because they are not used to store continuations (although contexts *are* used as addresses to store return values). Similar to existing approaches that require one, context allocator *allocCtx* in Modf can be used to tune the precision of the analysis. However, because the inter-context analysis must be able to analyze execution contexts, at least the following information must be determinable from a function execution context in Modf: (i) the syntactic function, and (ii) the argument values. For the argument values, one option would be to include the argument values as part of the context. We opt for including the extended environment (the environment after binding function parameters) instead, so that the signature of *allocCtx* is $allocCtx : \mathsf{Exp} \times Env \to K$ with $allocCtx(e, \rho) = (e, \rho)$. With this choice of context allocator, the single mechanism of read-dependency tracking (also of the addresses of the function parameters) suffices to trigger the reanalysis of function executions when they are called with different argument values.

Note that when taking a context-insensitive *alloc* function (0-CFA), then taking only a function as execution context (as we did in the examples in Section 2) is sound, since each parameter is its own address and a function therefore corresponds with a single and unique set of parameter addresses.

3.6 Intra-Context Analysis

Contrary to AAM-style analyses, Modf cannot be used with a traditional transfer function for computing the abstract collecting semantics. Instead, Modf performs local fixed-point computations for each function execution context through an *intra-context analysis*, which will be used by an *inter-context analysis* (presented in the next section) in order to obtain the abstract collecting semantics. The intra-context analysis is defined as a function $Intra : K \times Store \to A \times Store \times \mathcal{P}(Eff)$ which, given a context κ and a store σ, provides:

- Some information computed by the analysis, represented by an element of A; we define A as the set of reachable states within a context ($A = \mathcal{P}(\Sigma)$) because it most closely resembles the collecting semantics computed by a machine-based analysis such as AAM.
- The resulting store, which has to contain the return value of the context at address $\mathbf{ret}(\kappa)$, i.e., for the resulting store σ' we should have $\mathbf{ret}(\kappa) \in \mathrm{dom}(\sigma')$.

– The set of effects generated by the transition relation during the analysis of context κ.

With the definition of the abstract transition relation $\overset{E}{\rightsquigarrow}$, we define the intra-context analysis as the fixed-point computation of a transfer function $\mathcal{F}^{\kappa}_{\sigma_0}$, acting on elements of the domain $A \times \mathit{Store} \times \mathcal{P}(\mathit{Eff})$. For a context under analysis κ and for an initial store σ_0, this domain consists of the set of reachable states, the store, and the set of generated effects.

$$\mathcal{F}^{(e,\rho)}_{\sigma_0}(S, \sigma, E) = \langle \{\varsigma_0\}, \sigma_0, \varnothing \rangle \cup \bigcup_{\substack{\varsigma \in S \\ \varsigma, \sigma \overset{E'}{\rightsquigarrow} \varsigma', \sigma'}} \langle \{\varsigma'\}, \sigma', E' \rangle$$
$$\text{where } \varsigma_0 = \langle \mathbf{ev}(e, \rho), \mathbf{ret}((e, \rho)) : \epsilon \rangle$$

This transfer function deems as reachable the initial state ς_0, and any state ς' that can be reached in one abstract transition from a reachable state ς. The initial stack consists of a single stack frame **ret** to mark the boundary of the function execution. Effects detected during transitions are collected and will be used by the inter-context analysis to detect contexts that need to be reanalyzed. The intra-context analysis is defined as the least fixed point of the transfer function: $\mathit{Intra}(\kappa, \sigma) = \mathrm{lfp}(\mathcal{F}^{\kappa}_{\sigma})$. The only way a state could be reachable from itself would be through a recursive call, but this is delegated to the inter-context analysis. Therefore the computation of this fixed point does not require a set of seen states and associated state comparisons.

3.7 Inter-Context Analysis

The inter-context analysis operates on a worklist of execution contexts, analyzing one execution context until completion before moving on to the next one. A Modf analysis starts with the inter-context analysis on a worklist containing the root context as sole element. The root context represents the initial context in which the input program is evaluated. The inter-context analysis terminates when its worklist is empty, returning the mapping from contexts to intra-context analysis results. The inter-context analysis also keeps track of the read dependencies of contexts on addresses to support the mechanism of reanalyzing contexts when an address they depend on is written to.

Formally speaking, the inter-context analysis is defined by the function $\mathit{Inter} : \mathcal{P}(K) \times (\mathit{Addr} \to \mathcal{P}(K)) \times \mathit{Store} \times (K \to A) \to (K \to A)$. It acts on a worklist of contexts ($\mathcal{P}(K)$), a map that tracks which addresses are read from by which context ($\mathit{Addr} \to \mathcal{P}(K)$), a global store ($\mathit{Store}$), and a map S that stores the most recent results from intra-context analyses ($K \to A$). Since we compute collecting semantics, we have $A = \mathcal{P}(\Sigma)$.

$$\mathit{Inter}(\varnothing, \_, \_, S) = S$$
$$\mathit{Inter}(\kappa \uplus \kappa s, R, \sigma, S) = \mathit{Inter}(\kappa s \cup \bigcup_{\substack{\mathbf{c}(\kappa') \in E \\ \kappa' \notin \mathrm{dom}(\sigma)}} \kappa' \cup \bigcup_{\substack{\mathbf{w}(a) \in E \\ \kappa' \in R(a) \\ \sigma(a) \neq \sigma'(a)}} \kappa', R \sqcup \bigsqcup_{\mathbf{r}(a) \in E} [a \mapsto \{\kappa\}], \sigma', S[\kappa \mapsto S'])$$
$$\text{where } \langle S', \sigma', E \rangle = \mathit{Intra}(\kappa, \sigma)$$

If the worklist is empty, map S is returned. Otherwise, the inter-context analysis pulls a context κ from its worklist and performs an intra-context analysis. Based on the results of the intra-context analysis, additional contexts may be added to the worklist. Adding contexts to the worklist requires comparing contexts which is cheaper than comparing states in a traditional analysis where the worklist contains program states. Also note the absence of a set of seen states, which is needed in traditional algorithms to ensure termination. A context κ' is added to the worklist if at least one of the following two conditions is met.

1. Context κ' was called ($\mathbf{c}(\kappa') \in E$) and has not yet been analyzed (modelled as $\kappa' \notin \mathrm{dom}(\sigma)$).
2. Context κ' has a read dependency on an address a ($\kappa' \in R(a)$) and a was written to ($\mathbf{w}(a) \in E$) in a way that changes the store ($\sigma(a) \neq \sigma'(a)$).

Any address a' that was read during the intra-context analysis of context κ (i.e., $\mathbf{r}(a') \in E$) is registered as being depended upon by κ by updating the mapping of read dependencies R accordingly. The global store is also updated and the result S' of the intra-context analysis is stored in S.

Collecting semantics. Let $\kappa_0 = allocCtx(e, [])$ be the root context for program e. The abstract collecting semantics of program e is obtained by computing $Inter(\kappa_0, \varnothing, \varnothing, \varnothing, \varnothing)$, which results in a mapping $S \in K \to A$ from contexts to the set of states that may be reachable in that context.

3.8 Soundness

Soundness of the Abstract Semantics Except from the rule for function calls, our abstraction of the transition relation rules follows the usual AAM recipe. Their soundness is proven by a case analysis [25]. As our function call rule does not step into the body of the called function, its soundness solely relies on the fact that $\sigma(\kappa)$ holds a sound approximation of the result of a function call —which is proven in the last paragraph of this section.

Soundness of the Intra-Context Analysis We have to show that given an approximation of the store σ, our intra-context analysis $Intra(\kappa, \sigma)$ yields a sound over-approximation for the reachable states, the store, and the effects returned. This is the case because all states reachable from its input store and context will be analyzed, by definition. The soundness of the intra-context analysis therefore also relies on the fact that the given input store is sound.

Soundness of the Inter-Context Analysis The soundness proof for the inter-context analysis amounts to showing that the considered store eventually becomes a sound over-approximation of all possible stores reached by the concrete semantics, and contains an over-approximation of all return values of each function at the address corresponding to their context. This is shown by the fact that intra-context analyses are sound for the store with which they are executed, hence the resulting store is completed with information coming from the analysis of a context given an input store. This in turn will trigger other analyses based on the discovered read dependencies and function calls. We note that a discovered context that has not yet been analyzed will be analyzed by the intra-context analysis, and that a context that has already been analyzed will be analyzed upon changes to values in the store. Eventually, all reachable contexts will be analyzed for all possible changes to the values they depend on, resulting in a sound over-approximation of the store and a sound over-approximation of all possible return values of these contexts. Hence, the inter-context analysis is sound.

3.9 Termination

Termination of the Intra-Context Analysis The intra-context analysis, defined as a fixed-point computation, always terminates. All components of the abstract state space are finite, except for the stack: the abstract address domain *Addr* itself is made finite by the abstraction, and this propagates to the other components of the state space. The resulting set of abstract environments is finite, as there is a finite number of variable names and a finite number of abstract addresses to compose them from. The set of abstract values (Val) is finite as there is a finite number of abstract environments from which closures can be composed. The sets of stores ($Store$), effects (Eff), and contexts (K) become finite too.

Even though the *Stack* abstract domain is not finite, the intra-context analysis cannot construct an infinite stack. Stacks only grow when analyzing `let` expressions, and there can only be a finite number of such expressions within a function body. Constructing an infinite stack requires loops in the analysis, which is precluded by our use of the value cached in the store for a (potentially recursive) function call. The fixed-point computation for the intra-context analysis will therefore always terminate.

Termination of the Inter-Context Analysis The inter-context analysis terminates when its worklist is empty. This worklist grows in two cases.

First, when a function call with context κ is encountered for the first time (i.e., a $\mathbf{c}(\kappa)$ effect is discovered), κ is added to the worklist. There is a finite number of syntactic function calls, and once a function call has already been considered (modelled as $\kappa \in \mathrm{dom}(\sigma)$), it will not be considered again. The worklist can hence not grow indefinitely through function calls.

Second, when a write effect $\mathbf{w}(a)$ is discovered for an address that is read by a context κ, this context is added to the worklist under the condition that the value residing at address a in the store has itself changed ($\sigma(a) \neq \sigma'(a)$). The store being monotone and the number of possible values associated with each address in the store being finite, values will eventually stabilize. This ensures that a given context can be considered for reanalysis only a finite number of times.

Altogether, each context κ will only be analyzed a finite number of times, and there can only be a finite number of contexts for a given program. This ensures that the inter-context analysis always terminates.

3.10 Complexity

Complexity of the Intra-Context Analysis The intra-context analysis can execute at most a number of transitions equal to the number of expressions in the context under analysis. Hence, the complexity of the intra-context analysis given a specific context and specific store is in $\mathcal{O}(|\mathsf{Exp}|)$, where $|\mathsf{Exp}|$ is the size of the program under analysis.

Complexity of the Inter-Context Analysis Each context managed by the inter-context analysis can be analyzed for each change in the store to each of the addresses read in that context. An address can have at most $|\mathsf{Exp}|$ different values, hence the number of changes to an address is bounded by $|\mathsf{Exp}|$. Similarly, with our address allocation strategy, there are at most $|\mathsf{Exp}|$ addresses in the store, hence each context can be analyzed at most $|\mathsf{Exp}|^2$ times.

The inter-context analysis manages at most $|\mathsf{Exp}|$ contexts. With these bounds, one derives a worst-case time complexity of $\mathcal{O}(|\mathsf{Exp}|^4)$: there are at most $|\mathsf{Exp}|$ contexts, each analyzed at most $|\mathsf{Exp}|^2$ times, and the complexity of the analysis of one context being bounded by $|\mathsf{Exp}|$. However, note that for a given program, the number of contexts is inversely proportional to the size of each context: a program with the worst-case context length ($|\mathsf{Exp}|$) can have only one context, as its number of expressions is equal to the size of the program. Conversely, a program with the worst-case number of contexts ($|\mathsf{Exp}|$) has the minimal size for each context. In fact, the number of contexts is related to the size of the contexts in such a way that the worst-case of their multiplication is $|\mathsf{Exp}|$. Hence, MODF has a worst-case time complexity of $\mathcal{O}(|\mathsf{Exp}|^3)$, which is equal to the worst-case time-complexity of a 0-CFA AAM analysis widened with a global store [25]. In practice, as we evaluate in Section 4, MODF executes in a lower analysis time than the equivalent AAM analysis.

4 Evaluation

We implemented an AAM analyzer and a MODF analyzer in the Racket dialect of Scheme, and evaluated them on several Scheme benchmark programs. Compared to the descriptions and formalizations presented in this paper, the analyzers support a more extensive input language and semantics featuring conditionals, lists, vectors, and additional primitives. Our AAM analyzer is a faithful implementation of an AAM analysis, more specifically the AAC variant introduced by Johnson and Van Horn [10], configured with 0-CFA store value allocators and a stack value allocator identical to that of MODF.

To make the comparison possible and fair, we actually derived MODF from the AAM implementation, only changing what was necessary. Therefore both implementations share infrastructure for ASTs, lattices, machine semantics (the intra-context small-step relation), primitives, and so on. We did not optimize the AAM and MODF analyzer and will address applying and evaluating different optimization strategies as future work (also see Section 5). The worklist strategy used by the inter-context analysis of MODF and by the AAM analysis both follow a last-in first-out strategy. The analyzers' implementation and test setup, including benchmark programs, are publicly available[1].

Our set of benchmark programs consists of 20 Scheme programs coming from a variety of sources, including the Gabriel performance benchmarks [6], benchmarks from related work typically used to challenge control-flow analyses (see Section 5), and benchmarks from the Computer Language Benchmarks Game [5]. In the remainder of this section we report on the soundness, precision, and performance of our implementations on these programs.

[1] https://github.com/jensnicolay/modf

4.1 Soundness Testing

We first established the correctness of our AAM-based semantics by running each benchmark program under concrete semantics and checking that it produced a single answer equivalent to the answer computed by Racket for the same program. We then mechanically checked that, for each program and under abstract semantics, the abstract values for all variables produced by our AAM and MODF implementations subsume their corresponding concrete values after abstraction [1]. From these experiments we conclude that both analyzers are sound w.r.t. to our set of benchmark programs.

4.2 Precision Evaluation

We measured the overall precision of each analysis by counting the number of abstract values in all states of the results of each analysis. Because we use a set lattice, set size can be used as a measure of precision: the more elements are in a set representing an abstract value, the lower the precision. Column *Values* in Table 1 lists the results of this experiment (lower is better). In comparison to AAM, MODF may lose precision with respect to function argument values and the return value for a function execution context, as under our test configuration (0-CFA) these values are joined in the store. Upon reanalysis of a context, this may introduce spurious results. In contrast, AAM will not have spurious results for a first call to a function, but may have spurious results for subsequent calls. As seen from the numbers in Table 1, we conclude that in practice the AAM and MODF analyzers are similar in precision for the majority of our benchmark programs.

		AAM		MODF		Difference	
Program	|Exp|	Values	Mono	Values	Mono	Values	Mono
`primtest`	281	66	55	66	55	+0%	+0%
`partialsums`	326	86	73	86	73	+0%	+0%
`treeadd`	354	123	57	128	57	+5%	+0%
`spectralnorm`	400	109	89	109	89	+0%	+0%
`matrix`	351	106	77	109	74	+2%	-4%
`classtree`	430	439	97	441	91	+0%	-6%
`fankuch`	415	131	98	133	97	+2%	-1%
`destruct`	356	167	48	163	48	-2%	+0%
`supermerge`	202	972	21	990	18	+2%	-14%
`churchnums`	194	403	19	403	19	+0%	+0%
`deriv`	331	735	46	756	47	+3%	+2%
`regex`	540	426	69	423	69	-1%	+0%
`triangl`	448	698	40	659	40	-6%	+0%
`graphs`	1407	657	198	657	197	+0%	-1%
`mazefun`	1100	2587	117	2615	116	+1%	-1%
`dderiv`	449	2463	49	2457	49	+0%	+0%
`scm2java`	1769	5908	266	5956	265	+1%	-1%
`browse`	1251	8935	129	8606	129	-4%	+0%
`mceval`	1390	13178	159	13049	159	-1%	+0%
`boyer`	2260	115365	86	115574	86	+0%	+0%

Table 1. Precision comparison between AAM and MODF. |Exp| is the number of atoms present in the program under analysis. *Values* is the sum of the set of abstract values (lower is better), and *Mono* is the number of monomorphic call sites detected by the analysis (higher is better). The *Difference* columns indicate the difference in percentage of the results for AAM and MODF: a positive percentage indicates that MODF has detected more elements. For *Values*, lower percentages are better, and for *Mono*, higher percentages are better.

Although the measurements of the number of values give a good indication of the overall precision of the analysis results, they do not reveal much about "useful" precision. Therefore we also counted the number of singleton sets computed by each abstract analysis (column *Mono* in Table 1, higher is better). Singleton sets indicate abstract values that, in our configuration, represent either a single primitive type, a single closure, or a single address (list or vector). Therefore, this measure of precision is interesting for client analyses such as type analysis, call graph reachability, and monomorphic call inference (a single closure value for an operator

position corresponds to a monomorphic function call). We conclude again that the precision of the AAM analyzer and MODF analyzer are comparable in this respect for the majority of the benchmark programs.

4.3 Performance Evaluation

We measured the time it takes for the analyses to complete, and how many abstract states are computed in this time. Table 2 depicts the results sorted in ascending order by AAM analysis time. The results show an average speedup of 3.7 for MODF over AAM on our set of 20 benchmark programs. MODF finished the analysis faster in 17 out of 20 programs, with speedup factors ranging between 1.4 and 12.3. We registered a slowdown for 3 out of 20 programs, with a doubling of analysis time for one smaller benchmark. The higher the AAM analysis time for a program, the better MODF performs: in Table 2 MODF generally results in a speedup.

		AAM		MODF		Reduction	
Program	|Exp|	States	Time	States	Time	States	Time
`primtest`	281	178	18	183	7	1.0×	**2.6**×
`partialsums`	326	223	18	225	7	1.0×	**2.6**×
`treeadd`	354	263	28	292	35	0.9×	0.8×
`spectralnorm`	400	256	31	269	10	1.0×	**3.1**×
`matrix`	351	237	44	256	27	0.9×	**1.6**×
`classtree`	430	352	72	465	137	0.8×	0.5×
`fankuch`	415	308	73	338	54	0.9×	**1.4**×
`destruct`	356	274	95	306	58	0.9×	**1.6**×
`supermerge`	202	356	295	212	63	**1.7**×	**4.7**×
`churchnums`	194	428	309	318	82	**1.3**×	**3.8**×
`deriv`	331	419	313	400	118	1.0×	**2.7**×
`regex`	540	579	558	491	86	**1.2**×	**6.5**×
`triangl`	448	371	861	373	205	1.0×	**4.2**×
`graphs`	1407	918	2020	898	347	1.0×	**5.8**×
`mazefun`	1100	1066	2655	1140	3379	0.9×	0.8×
`dderiv`	449	750	3003	546	889	**1.4**×	**3.4**×
`scm2java`	1769	2446	24925	1602	2356	**1.5**×	**10.6**×
`browse`	1251	1565	41478	1510	10621	1.0×	**3.9**×
`mceval`	1390	2333	46483	2478	23040	0.9×	**2.0**×
`boyer`	2260	13048	5241215	2154	425915	**6.1**×	**12.3**×

Table 2. Performance comparison between AAM and MODF. For AAM and MODF, column *States* is the number of states that the analysis has explored, and *Time* is the time taken by the analysis to run to completion, in milliseconds. The *Reduction* columns indicate the improvement in number of states and in time resulting from MODF, as a factor of the number of states explored (resp. the time taken) by AAM over the number of states explored (resp. the time taken) by MODF. A higher reduction factor shows more improvement resulting from MODF.

To gain insights to why MODF is faster on these programs, we performed some additional measurements with the following averaged results:

1. 8% of the running time of the AAM analyzer is spent checking seen states, which is avoided by MODF.
2. Adding return value merging to the AAM analyzer—something inherently present in MODF—only improves its running time by 2%.
3. 80% of reanalyzed states by the AAM analyzer do not yield new states, while this is the case for 63% of reanalyzed states by the MODF analyzer.

Considering that, apart from the effect tracking and the fixed-point computation, both analyzers share the same implementation, we can conclude that (i) avoiding the cost of checking for seen states, and (ii) profiting of a worklist strategy that can selectively reanalyze states upon store updates in combination with memoization of return values are the two major factors in explaining why MODF outperforms AAM on our set of benchmark programs.

MODF also tends to produce smaller flow graphs than AAM as analysis time increases, although for the smaller benchmarks MODF slightly outputs more program states. The reason is that MODF immediately continues after a function call with the cached result fetched from the store, even if the function was not previously analyzed. In the latter case, a $\bot$ return value results, which does not occur in AAM. However, for the larger benchmarks, the advantages of MODF and especially its return value memoization outweigh these $\bot$ return flows, and MODF often produces flow graphs with less states than AAM. It is also worth noting that MODF often is faster even when producing more states.

In conclusion, while MODF loses some precision for some programs when compared with AAM, we believe that the tradeoff our technique proposes between performance and precision is worthwhile.

5 Related Work

Modular analysis. MODF can be regarded as a modular analysis in that function execution contexts are analyzed one at a time and to completion. The concept of a modular analysis was formalized by Cousot and Cousot [2], which presents different general-purpose techniques. Sharir and Pnueli [21] introduces a program analysis that integrates an interprocedural analysis with intraprocedural analysis, similarly to MODF. The result of the intraprocedural analysis is a summary that is used by the interprocedural analysis. However, this approach remains limited to a first-order setting while MODF was explicitly designed with support for higher-order, side-effecting programs. Moreover, unlike modular summary-based analyses, MODF *does* reanalyze function execution contexts as new contexts and effects are discovered.

Abstract machines. In this paper we compare a MODF analyzer against an implementation of a variation of AAM, a well-known machine-based analysis approach [10,25] Related work has produced different techniques and extensions for AAM with varying trade-offs between precision and performance, of which a summary can be found in Gilray et al. [8]. MODF itself also uses AAM-style semantics for its small-step analysis of functions, which makes the comparison with the AAM analyzer justified and straightforward, the latter also on a technical level by maximizing code reuse. Applications of AAM are found in various domains, such as detecting function purity [18], performing symbolic execution for contracts [15], analyzing concurrent programs [14,23,24], determining function coupling [16], discovering security vulnerabilities [17] or performing malware analysis [11]. The approach used by MODF could improve the performance of such applications and other client analyses without impacting their precision.

Might and Shivers [13] introduced abstract garbage collection and abstract counting as techniques for increasing the performance and precision of a store-based abstract interpretation. Abstract garbage collection reclaims unused store addresses, but is not straightforward to adapt for MODF because of the per-context analysis in combination with a global store. Abstract counting keeps track of whether an address is allocated exactly once or multiple times in the store. If the address has only been allocated once, a strong update can be used instead of a weak update. Abstract counting is orthogonal to MODF and incorporating it into our approach and evaluating its effects is future work.

In this paper we compare unoptimized implementations of a MODF and an AAM analyzer. Johnson et al. [9] presents OAAM, a series of 6 steps that can be applied to optimize naive global-store AAM implementations for higher-order, functional languages, resulting in two to three order of magnitude speedups. However, OAAM requires heavy semantics and implementation engineering, while MODF is a simple technique that can be applied to side-effecting semantics as well. Some OAAM optimizations can be applied to MODF (e.g., store pre-allocation), while others clearly cannot (e.g., optimizations that rely on the absence of side effects, or those that involve the set of seen states).

Effect systems. MODF relies on effects to drive the inter-context analysis. This however differs from typical usages of effect systems [20]. A first difference is that effect systems usually extend a static type system, while MODF does not make any assumptions about the type system of the analyzed language. Another major difference is that effect systems are used to *reason* about the effects performed in the program under analysis, while MODF *relies* on effects during the analysis to perform a general-purpose static analysis that can serve

a number of client applications. Hence, while both MODF and effect systems use effects, the way effects are used is entirely different.

6 Conclusion

We presented MODF, a technique for the static analysis of higher-order, side-effecting programs in a modular way. MODF analyzes one single function execution at a time to completion while tracking read, write, and call effects. These effects trigger the analysis of other function executions, and the analysis terminates when no new effects can be discovered.

The goal of MODF's design is to reduce the overhead associated with maintaining a set of seen states while exploring the state space. By not relying on a set of seen states, MODF avoids many state comparisons, and by tracking read and write effects MODF is more selective in reanalyzing program states than traditional implementations of machine-based static analyses such as AAM.

We implemented an AAM analyzer and derived a MODF analyzer from it by adding effect tracking and changing the fixed point computation, and evaluated the two implementations on 20 benchmark programs. Our experiments show an average speedup of 3.7 for MODF over AAM on our set of benchmark programs. MODF finished the analysis faster in 17 out of 20 programs, with speedup factors ranging between 1.4 and 12.3. We also found that the AAM and MODF analyzer are similar in precision for the majority of our benchmark programs, while computing flow graphs of similar size or smaller.

In future research we will experiment with different concepts of modules and context sensitivities for analyzing modules, and will also examine opportunities to incrementalize and parallelize the approach. Although MODF is already inherently incremental in the sense that upon a change in the input program initially only the directly affected execution contexts can be (or should be) reanalyzed, the monotonicity of the global store complicates matters in terms of precision. The modular nature of MODF and the fact that the approach tracks interference between modules by design should facilitate its parallelization.

Acknowledgments

Jens Nicolay is funded by the SeCloud project sponsored by Innoviris, the Brussels Institute for Research and Innovation.

References

1. Andreasen, E.S., Møller, A., Nielsen, B.B.: Systematic approaches for increasing soundness and precision of static analyzers. In: Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017. pp. 31–36 (2017)
2. Cousot, P., Cousot, R.: Modular static program analysis. In: Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings. pp. 159–178 (2002)
3. Earl, C., Might, M., Van Horn, D.: Pushdown control-flow analysis of higher-order programs. In: Proceedings of the 2010 Workshop on Scheme and Functional Programming (Scheme 2010) (2010)
4. Flanagan, C., Sabry, A., Duba, B., Felleisen, M.: The essence of compiling with continuations. ACM SIGPLAN Notices **28**(6), 237–247 (1993)
5. Fulgham, B., Gouy, I.: The computer language benchmarks game. http://shootout. alioth. debian. org (2009)
6. Gabriel, R.P.: Performance and evaluation of LISP systems, vol. 263. MIT press Cambridge, Mass. (1985)
7. Gilray, T., Adams, M.D., Might, M.: Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016. pp. 407–420 (2016)
8. Gilray, T., Lyde, S., Adams, M.D., Might, M., Horn, D.V.: Pushdown control-flow analysis for free. In: Proceedings of the 43th Annual ACM Symposium on the Principles of Programming Languages (POPL 2016) (2016)

9. Johnson, J.I., Labich, N., Might, M., Van Horn, D.: Optimizing abstract abstract machines. In: ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013. pp. 443–454 (2013)
10. Johnson, J.I., Van Horn, D.: Abstracting abstract control. In: Proceedings of the 10th ACM Symposium on Dynamic languages. pp. 11–22. ACM (2014)
11. Liang, S., Might, M., Horn, D.V.: Anadroid: Malware analysis of android with user-supplied predicates. Electr. Notes Theor. Comput. Sci. **311**, 3–14 (2015)
12. Might, M., Manolios, P.: *A posteriori* soundness for non-deterministic abstract interpretations. In: Proceedings of the 10th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2009) (2009)
13. Might, M., Shivers, O.: Improving flow analyses via γcfa: abstract garbage collection and counting. In: ACM SIGPLAN Notices. vol. 41, pp. 13–25. ACM (2006)
14. Might, M., Van Horn, D.: A family of abstract interpretations for static analysis of concurrent higher-order programs. In: Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings. pp. 180–197 (2011)
15. Nguyen, P.C., Gilray, T., Tobin-Hochstadt, S., Horn, D.V.: Soft contract verification for higher-order stateful programs. PACMPL **2**(POPL), 51:1–51:30 (2018)
16. Nicolay, J., Noguera, C., Roover, C.D., Meuter, W.D.: Determining dynamic coupling in JavaScript using object type inference. In: Proceedings of the Thirteenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2013). pp. 126–135 (2013)
17. Nicolay, J., Spruyt, V., De Roover, C.: Static detection of user-specified security vulnerabilities in client-side javascript. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. pp. 3–13. ACM (2016)
18. Nicolay, J., Stiévenart, Q., De Meuter, W., De Roover, C.: Purity analysis for JavaScript through abstract interpretation. Journal of Software: Evolution and Process pp. e1889–n/a (2017), e1889 smr.1889
19. Nielson, F., Nielson, H.R., Hankin, C.: Algorithms. In: Principles of Program Analysis, pp. 365–392. Springer (1999)
20. Nielson, F., Nielson, H.R., Hankin, C.: Type and effect systems. In: Principles of Program Analysis, pp. 281–361. Springer (1999)
21. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. Tech. rep., New York Univ. Comput. Sci. Dept. (1978)
22. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie Mellon University Pittsburgh, PA (1991)
23. Stievenart, Q., Nicolay, J., De Meuter, W., De Roover, C.: Detecting concurrency bugs in higher-order programs through abstract interpretation. In: Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming. pp. 232–243. ACM (2015)
24. Stiévenart, Q., Nicolay, J., De Meuter, W., De Roover, C.: Mailbox abstractions for static analysis of actor programs. In: 31st European Conference on Object-Oriented Programming, ECOOP 2017. pp. 25:1–25:30 (2017)
25. Van Horn, D., Might, M.: Abstracting abstract machines. In: ACM Sigplan Notices. vol. 45, pp. 51–62. ACM (2010)
26. Vardoulakis, D., Shivers, O.: Cfa2: A context-free approach to control-flow analysis. In: Programming Languages and Systems, pp. 570–589. Springer (2010)