So You Want To Analyze Scheme Programs With Datalog?

DAVIS ROSS SILVERMAN* and YIHAO SUN*, Syracuse University, USA
KRISTOPHER MICINSKI, Syracuse University, USA
THOMAS GILRAY, University of Alabama at Birmingham, USA

Static analysis approximates the results of a program by examining only its syntax. For example, control-flow analysis (CFA) determines which syntactic lambdas (for functional languages) or (for object-oriented) methods may be invoked at each call site within a program. Rich theoretical results exist studying control flow analysis for Scheme-like languages, but implementations are often complex and specialized. By contrast, object-oriented languages (Java in particular) enjoy high-precision control-flow analyses that scale to thousands (or more) of lines of code. State-of-the-art implementations (such as DOOP on Soufflé) structure the analysis using Horn-SAT (Datalog) to enable compilation of the analysis to efficient implementations such as high-performance relational algebra kernels. In this paper, we present an implementation of control-flow analysis for a significant subset of Scheme (including `set!`, `call/cc`, and primitive operations) using the Soufflé Datalog engine. We present an evaluation on a worst-case term demonstrating the polynomial complexity of our *m*-CFA and remark upon scalability results using Soufflé.

CCS Concepts: • **Software and its engineering → Semantics**; **Domain specific languages**; **Automated static analysis**.

Additional Key Words and Phrases: control-flow analysis, abstract interpretation, m-CFA, datalog

ACM Reference Format:
Davis Ross Silverman, Yihao Sun, Kristopher Micinski, and Thomas Gilray. 2021. So You Want To Analyze Scheme Programs With Datalog?. *J. ACM* 39, 4, Article 111 (August 2021), 16 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Static analysis is a technique to explicate properties of a program's behavior via inspecting only the program's source code (without executing the program) [13]. There exist many frameworks for constructing program analyses, e.g., Cousot and Cousot's abstract interpretation [5]. A static analysis is *sound* when it is strictly conservative in the sense that any true program behavior is reported (at least approximately) as a result of the analysis. Unfortunately—due to the halting problem—no terminating static analysis may be both sound and *complete* (all reported results represent true behavior). While static analyses may be constructed using arbitrary degrees of precision (e.g., via instrumentation-based polyvariance [7]) in principle, in practice balancing precision and complexity while retaining soundness requires significant engineering effort [2].

*Both authors contributed equally to this research.

Authors' addresses: Davis Ross Silverman, dasilver@syr.edu; Yihao Sun, ysun67@syr.edu, Syracuse University, 900 S Crouse Ave, Syracuse, New York, USA, 13244; Kristopher Micinski, Syracuse University, 900 S Crouse Ave, Syracuse, New York, USA, 13244, kkmicins@syr.edu; Thomas Gilray, University of Alabama at Birmingham, 1720 University Blvd, Birmingham, Alabama, USA, 35294, gilray@uab.edu.

0004-5411/2021/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

A central challenge in analyzing Scheme programs is control-flow analysis: for each callsite, which syntactic lambdas (in which contexts, for a context-sensitive analysis) may be invoked? This problem is simple in procedural languages (which include only direct control-flow), but challenging for higher-order languages, as data-flow and control-flow must be performed simultaneously. Shivers defined the k-CFA family of increasingly-precise control-flow analyses for Scheme [16]. However, as Shivers notes, uniform k-CFA quickly becomes intractable for the case $k > 0$, even for reasonably-sized programs [15]. In fact, Van Horn and Mairson later showed that k-CFA is EXPTIME-complete [18].

Compared to Scheme, significant engineering work has been expended into whole-program analysis for object-oriented languages, particularly Java [1, 3, 9, 14]. This has culminated in state-of-the-art systems such as DOOP, whose analysis is written in Datalog for subsequent compilation to efficient relational algebra kernels implemented via C++ [3]. Systems such as DOOP scale to large (multi-thousand line) codebases even when context-sensitivity is considered. This would appear at odds with Van Horn and Mairson's result that k-CFA is EXPTIME-complete. To resolve this paradox, Might et al. show that the degenerative closure structure of object-oriented languages (analogous to the difference between flat and linked closures), context-sensitive control-flow analyses of object-oriented languages is PTIME [11]. The authors also present m-CFA, a technique to analyze arbitrary higher-order languages using flat closures to achieve polynomial complexity.

While the techniques for analyzing object-oriented languages using Datalog are well understood, we are aware of no existing presentation illustrating how Datalog may be used to implement the analysis of Scheme. We find this particularly surprising, as we believe the extreme efficiency of modern Datalog engines may prove a key enabling technology to tackle the inherent complexity of higher-order languages.

In this paper, we present a systematic approach for deriving a Datalog-based implementation of control-flow analysis for Scheme-like languages. Key to our approach is the formulation of the analysis via the abstracting abstract machines (AAM) methodology of Van Horn and Might [19]. We present how this formulation may be translated to Datalog, overcoming several key obstacles unique to Scheme. To evaluate our approach, we implemented an analysis of a significant subset of Scheme including multi-argument lambdas, conditional control-flow, builtins, and first-class continuations (`call/cc`). In section 3, we present a formalization of this language (noting how our choices anticipate a Datalog implementation) via the AAM approach, following in section 4 with its corresponding Datalog transliteration. We validated (manually, via inspection) the correctness of our analysis and in section 5 present the results of a set of experiments benchmarking our implementation of m-CFA for our subset of Scheme.

2 BACKGROUND AND RELATED WORK

In this section, we sketch several key background concepts that underlie our abstract semantics in Section 3 and subsequent Datalog implementation in Section 4.

2.1 Program Analysis and Abstract Interpretation

Kildall first introduced dataflow analysis (of flowchart-style programs in the style of Floyd [6]) to approximate static program behavior for the purpose of compile-time optimization [10]. A central idea of Kildall's work was using a lattice to impose an ordering on analysis results and ensure termination via the finiteness of said lattice. Cousot and Cousot later generalized Kildall's ideas to define the abstract interpretation of a program. Abstract interpretation allows relating an arbitrary pair of lattices, typically a concrete state space (Σ) and its abstraction ($\hat{\Sigma}$), along with a pair, (α, γ), of (adjunctive) mappings between them for abstraction (α) and concretization (γ). Using this abstraction alongside a *collecting semantics* allows iterating a program analysis to some

fixed-point in an arbitrary lattice of abstract results. Assuming this lattice of results is of finite height, the collecting semantics (and therefore analysis) will necessarily terminate via Tarski's fixpoint theorem [17].

We elide a complete presentation of abstract interpretation; there exist several expository texts including those by Miné [12] and Nielsen and Nielsen [13].

2.2 Control Flow Analysis

Languages such as C do not include indirect control flow. Determining control flow (and also data flow) is simple for these languages, as control is syntactically apparent. In Scheme, it is more difficult to tell which values flow to a particular variable because of the pervasive use of higher-order functions.

Consider the following Scheme code:

```
(let* ([f (foo 42)]
       [g (bar 99)]
       [h (if (= a b) (g 30) (f g))])
  (g h))
```

Deciding which branch of the `if` is taken depends on (at least) the values that flow to `a` and `b`. Similarly, to understand data flow, we also must know control flow: knowing which value flows to `h` requires reasoning about the `if`'s control flow. The key is to compute both simultaneously. As the computation continues, data flow information is fed to create a control-flow graph (CFG) on-the-fly, and the new CFG is used to find new data-flow. This is a central idea in the original formulation of k-CFA by Shivers [16], though presentations can also be found elsewhere [12, 13].

2.3 Abstract Abstract Machines

Might and Van Horn presented the Abstracting Abstract Machines (AAM) approach to abstract interpretation for functional languages [19]. The key insight in their work is to redirect all sources of recursion in the analysis through a store which may be finitized by construction. Using this approach, an abstract semantics may be derived from an abstract machine specifying a concrete semantics. The AAM-based approach can compute any type of CFA, and encompasses a broad array of analysis precision including, e.g., object-sensitivity [7]. The machine described in this paper utilizes m-CFA [11], a variant of k-CFA that uses flat closures.

2.4 Datalog

Datalog is a bottom-up logic programming language largely based on Horn-SAT. We refer the reader to the exposition of Ceri et al. for a detailed description of Datalog [4]. Datalog programs consist of a set of Horn clauses of the form $P(x_0, ...) \leftarrow Q(y_0, ...) \wedge ... \wedge S(z_0, ...)$. To evaluate these programs, an extensional database (EDB) is provided as input specifying a set of initial facts. A Datalog engine then runs the rules to a fixed-point to produce an output database. The following example computes the `cousin` relation from the EDB relations of `parent` and `sibling`.

```
cousin(a, c) :- parent(a, p), sibling(p, q), parent(c, q).
```

Relations can be recursive. Calculating ancestry is simple. The base case shows that a parent is trivially an ancestor, but a parent of an ancestor is also an ancestor:

```
ancestor(p, a) :- parent(p, a).
// If p already has some ancestor a,
// the parent b of a is also an ancestor of p.
ancestor(p, b) :- ancestor(p, a), parent(a, b).
```

3 SYNTAX AND ABSTRACT SEMANTICS

Syntactic Classes

$$
\begin{array}{rl}
e \in \mathsf{Exp} ::= & æ \\
& \mid (\texttt{if}\ e\ e\ e) \mid (\texttt{set!}\ x\ e) \\
& \mid (\texttt{call/cc}\ e) \mid \mathit{let} \\
& \mid (\mathit{op}\ e\ e) \mid (e\ e\ e\ \ldots) \\
æ \in \mathsf{AExp} ::= & x \mid \mathit{lam} \mid b \mid n
\end{array}
\qquad
\begin{array}{rl}
b \in \mathbb{B} \triangleq & \{\texttt{\#t}, \texttt{\#f}\} \\
n \in \mathbb{Z} & \\
x \in \mathsf{Var} \triangleq & \text{The set of identifiers} \\
\mathit{let} \in \mathsf{Let} ::= & (\texttt{let}\ ((x\ e)\ \ldots)\ e) \\
\mathit{lam} \in \mathsf{Lam} ::= & (\lambda\ (x)\ e) \\
\mathit{op} \in \mathsf{Prim} \triangleq & \text{The set of primitives}
\end{array}
$$

Fig. 1. Syntactic Classes for the Scheme CESK* Machine

Many famous papers use a significant subset of Scheme, perhaps too small for useful real-world analyses. Might et al.'s m-CFA paper uses a variant of the CPS lambda calculus with multi-argument functions [11]. Shivers' original work on CFA utilized a CPS subset without direct conditionals or mutation [16]. Van Horn et al.'s original Abstracting Abstract Machines adds direct mutation, conditionals, and first order continuations, but does not include multi-argument lambdas, which is a crucial component of CFA [19], as they preserve contexts more thoughtfully. Each of these leaves out important components which are required to build analyses on real languages.

In Figure 1, we partition complex and atomic expressions. We also include a variety of useful syntax such as mutation through `set!` expressions, `let` bindings, and the higher order control flow operator, `call/cc`. Conditional expressions and primitives are also highly important for writing meaningful examples. We support binary primitives, which are needed to analyze real Scheme programs. Primitive ops include constructing lists and mathematical and logical operations.

The subset of Scheme used is capable of supporting many real world programs. Results of analyses using this subset are more clear than an analysis of a heavily compiled subset. Results of analyses are more understandable and easier to implement due to these features. This syntax also clarifies contexts in a CFA and how they grow and shrink. With `let` and multi-argument lambdas, we can place multiple bindings in a single context. This will avoid states trivially ascending to top, by decreasing the number of contexts.

Figure 2 presents a CESK* machine utilizing m-CFA for value analysis, [11, 19]. The states are partitioned into *evaluation* and *application* states, and the store is partitioned into a value store and a continuation store. There are a variety of continuations which give meaning to the program.

The set of states is partitioned into 2 types of sub-state. The control of an *Eval* state is syntax. When transitioning from an *Eval* state, the goal is to produce a value. For example, when an `if` expression is encountered, the resulting state is another *Eval* state with the guard expression set to the control. The second type of state is an *Apply*, where the control is a value. These states will apply the control depending on the continuation. When an `if` expression's guard reaches an *Apply* state, a branch is selected based on the inspected value.

With abstracted abstract machines, both values and continuations are identified through the store with an address. However, they require different address types. Value addresses are determined by the polyvariance and analyses type we are conducting [7]. Continuations, however, are allocated in the Pushdown-For-Free style, which has its own allocator [8]. These stores are combined in the state, but are accessed separately and combined when clear in the semantics for brevity.

Environments in this machine are not a mapping, as shown in many AAM based approaches. Instead, environments are simply a function of context, and stand-in for time-stamps as in k-CFA

Semantic Classes

$$
\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &\triangleq E\langle\widehat{Eval}\rangle + A\langle\widehat{Apply}\rangle \\
\widehat{Eval} &\triangleq \mathsf{Exp} \times \widehat{Context} \\
&\quad \times \widehat{Store} \times \widehat{KAddr} \\
\widehat{Apply} &\triangleq \widehat{Val} \times \widehat{Store} \times \widehat{KAddr} \\
\hat{\sigma} \in \widehat{Store} &\triangleq \widehat{VStore} \times \widehat{KStore} \\
\hat{\sigma}_v \in \widehat{VStore} &\triangleq \widehat{VAddr} \rightharpoonup \mathcal{P}(\widehat{Val}) \\
\hat{\sigma}_k \in \widehat{KStore} &\triangleq \widehat{KAddr} \rightharpoonup \mathcal{P}(\widehat{Kont}) \\
\widehat{ctx} \in \widehat{Context} &\triangleq \mathsf{Exp}^m \\
\hat{a}_v \in \widehat{VAddr} &\triangleq \mathsf{Var} \times \widehat{Context} \\
\hat{a}_{\hat{\kappa}} \in \widehat{KAddr} &\triangleq \mathsf{Exp} \times \widehat{Context}
\end{aligned}
\qquad
\begin{aligned}
\hat{v} \in \widehat{Val} &\triangleq \mathbb{Z} + \mathbb{B} + \widehat{PVal} + \widehat{Clo} + \widehat{KAddr} \\
\widehat{p} \in \widehat{PVal} &\triangleq \mathsf{Prim} \times \widehat{Val} \times \widehat{Val} \\
\widehat{clo} \in \widehat{Clo} &\triangleq \mathsf{Lam} \times \widehat{Context} \\
\hat{\kappa} \in \widehat{Kont} &::= \widehat{\mathbf{mtk}} \mid \widehat{\mathbf{ifk}}(e, e, \widehat{ctx}, \hat{a}_{\hat{\kappa}}) \\
&\quad \mid \widehat{\mathbf{setk}}(\hat{a}_v, \hat{a}_{\hat{\kappa}}) \mid \widehat{\mathbf{callcck}}(\widehat{ctx}, \hat{a}_{\hat{\kappa}}) \\
&\quad \mid \widehat{\mathbf{let}}(\hat{a}_v, e, \widehat{ctx}, \hat{a}_{\hat{\kappa}}) \mid \widehat{\mathbf{fn}}(v, n, \widehat{ctx}, \hat{a}_{\hat{\kappa}}) \\
&\quad \mid \widehat{\mathbf{p1}}(op, e, \widehat{ctx}, \hat{a}_{\hat{\kappa}}) \mid \widehat{\mathbf{p2}}(op, \hat{v}, \hat{a}_{\hat{\kappa}}) \\
&\quad \mid \widehat{\mathbf{arg}}(e \ ..., \widehat{ctx}, \widehat{ctx}, \hat{a}_{\hat{\kappa}})
\end{aligned}
$$

Fig. 2. Semantic Classes for the Scheme m-CFA CESK* Machine

[11]. Here, closures utilize flat-contexts, instead of the usual linked model. Instead of store-fetches being $\hat{\sigma}(\widehat{ctx}(x))$, they are now $\hat{\sigma}(x, \widehat{ctx})$. In m-CFA, the current context is based on the top m stack frames, as opposed to the *latest k* call-sites.

There are a variety of continuation types to enumerate the various semantic features of the Scheme subset. When a `set!` expression is evaluated, a $\widehat{\mathbf{setk}}$ continuation is added to the store and pushed onto the stack. This identifies what to accomplish after the inner expression is fully evaluated.

The *Eval* rules in Figure 3 govern how to break down complex expressions to form an atomic expression. Some rules are straightforward, such as **E-If**. After encountering an `if` expression, the guard is evaluated. The $\widehat{\mathbf{ifk}}$ continuation is created to keep track of what to do when we know the value of the guard. Generally, an evaluation rule focuses on evaluating syntax, and not creating semantic objects such as contexts.

However other transition rules are trickier. The **E-Let** rule transitions to multiple states, one for each binding. For brevity, at least one binding is required, as a 0-binding `let` would have to transition directly to the body with a new context. Let is also an interesting case since it creates the resulting context during syntactic evaluation. Enough information is known at evaluation time to create it, so it is done then. The later application rule can then be more simple.

The **E-Prim** and **E-Call** rules are multi-stage rules. They each have multiple expressions that must be evaluated in a specific order. In a function call, the function is evaluated before the arguments, so there must be a special continuation frame for each juncture in the evaluation.

Atomic evaluation is defined in Figure 4. The function $\widehat{\mathcal{A}}$ produces values from atomic expressions. The rule **E-AE** will transition from an *Eval* state into an *Apply* state using $\widehat{\mathcal{A}}$.

Apply rules, as shown in Figure 5, transition based on the value and the current continuation. These rules primarily govern contexts. Here, the context may be extended, bindings added to it, or it may be returned to a previous version.

Evaluation Rules

$$E\langle(\texttt{if}\ e_g\ e_t\ e_f), \widehat{ctx}, \hat{\sigma}, \hat{a}_{\hat{\kappa}}\rangle \rightsquigarrow E\langle e_g, \widehat{ctx}, \hat{\sigma}', \hat{a}'_{\hat{\kappa}}\rangle$$
$$\text{where } \hat{a}'_{\hat{\kappa}} \triangleq \widehat{alloc_k}(\hat{\varsigma}, e_c, \widehat{ctx}) \qquad \textbf{(E-If)}$$
$$\hat{\kappa} \triangleq \widehat{\textbf{ifk}}(e_t, e_f, \widehat{ctx}, \hat{a}_{\hat{\kappa}})$$
$$\hat{\sigma}'_{\hat{\kappa}} \triangleq \hat{\sigma}_{\hat{\kappa}} \sqcup [\hat{a}'_{\hat{\kappa}} \mapsto \hat{\kappa}]$$

$$E\langle(op\ e_0\ e_1), \widehat{ctx}, \hat{\sigma}, \hat{a}_{\hat{\kappa}}\rangle \rightsquigarrow E\langle e_0, \widehat{ctx}, \hat{\sigma}', \hat{a}'_{\hat{\kappa}}\rangle$$
$$\text{where } \hat{a}'_{\hat{\kappa}} \triangleq \widehat{alloc_k}(\hat{\varsigma}, e_i, \widehat{ctx}) \qquad \textbf{(E-Prim)}$$
$$\hat{\kappa} \triangleq \widehat{\textbf{p1}}(op, e_1, \widehat{ctx}, \hat{a}_{\hat{\kappa}})$$
$$\hat{\sigma}'_{\hat{\kappa}} \triangleq \hat{\sigma}_{\hat{\kappa}} \sqcup [\hat{a}'_{\hat{\kappa}} \mapsto \hat{\kappa}]$$

$$E\langle let, \widehat{ctx}, \hat{\sigma}, \hat{a}_{\hat{\kappa}}\rangle \rightsquigarrow E\langle e_i, \widehat{ctx}, \hat{\sigma}', \hat{a}'_{\hat{\kappa}}\rangle$$
$$\text{where } \widehat{ctx}' \triangleq \widehat{new}(\hat{\varsigma}) \qquad \textbf{(E-Let)}$$
$$let = (\texttt{let}\ ((x_0\ e_0)\ (x_s\ e_s)\ ...)\ e_b)$$
$$(x_i, e_i) \in ([x_0 : x_s], [e_0 : e_s])$$
$$\hat{a}_v \triangleq \widehat{alloc_v}(x_i, \hat{\varsigma})$$
$$\hat{a}'_{\hat{\kappa}} \triangleq \widehat{alloc_k}(\hat{\varsigma}, e_i, \widehat{ctx})$$
$$\hat{\kappa} \triangleq \widehat{\textbf{let}}(e_b, \hat{a}_v, \widehat{ctx}', \hat{a}_{\hat{\kappa}})$$
$$\hat{\sigma}'_{\hat{\kappa}} \triangleq \hat{\sigma}_{\hat{\kappa}} \sqcup [\hat{a}'_{\hat{\kappa}} \mapsto \hat{\kappa}]$$

$$E\langle(\texttt{call/cc}\ e), \widehat{ctx}, \hat{\sigma}, \hat{a}_{\hat{\kappa}}\rangle \rightsquigarrow E\langle e, \widehat{ctx}, \hat{\sigma}', \hat{a}'_{\hat{\kappa}}\rangle$$
$$\text{where } \widehat{ctx}' \triangleq \widehat{new}(\hat{\varsigma}) \qquad \textbf{(E-C/cc)}$$
$$\hat{a}'_{\hat{\kappa}} \triangleq \widehat{alloc_k}(\hat{\varsigma}, e, \widehat{ctx})$$
$$\hat{\kappa} \triangleq \widehat{\textbf{callcck}}(\widehat{ctx}', \hat{a}_{\hat{\kappa}})$$
$$\hat{\sigma}'_{\hat{\kappa}} \triangleq \hat{\sigma}_{\hat{\kappa}} \sqcup [\hat{a}'_{\hat{\kappa}} \mapsto \hat{\kappa}]$$

$$E\langle(\texttt{set!}\ x\ e), \widehat{ctx}, \hat{\sigma}, \hat{a}_{\hat{\kappa}}\rangle \rightsquigarrow E\langle e, \widehat{ctx}, \hat{\sigma}', \hat{a}'_{\hat{\kappa}}\rangle$$
$$\text{where } \hat{a}'_{\hat{\kappa}} \triangleq \widehat{alloc_k}(\hat{\varsigma}, (\texttt{set!}\ x\ e), \widehat{ctx}) \qquad \textbf{(E-Set!)}$$
$$\hat{a}_v \triangleq \widehat{alloc_v}((\texttt{set!}\ x\ e), \hat{\varsigma})$$
$$\hat{\kappa} \triangleq \widehat{\textbf{setk}}(\hat{a}_v, \hat{a}_{\hat{\kappa}})$$
$$\hat{\sigma}'_{\hat{\kappa}} \triangleq \hat{\sigma}_{\hat{\kappa}} \sqcup [\hat{a}'_{\hat{\kappa}} \mapsto \hat{\kappa}]$$

$$E\langle(e_f\ e_0\ e_s\ ...), \widehat{ctx}, \hat{\sigma}, \hat{a}_{\hat{\kappa}}\rangle \rightsquigarrow E\langle e_f, \widehat{ctx}, \hat{\sigma}', \hat{a}'_{\hat{\kappa}}\rangle$$
$$\text{where } \widehat{ctx}' \triangleq \widehat{new}(\hat{\varsigma}) \qquad \textbf{(E-Call)}$$
$$\hat{a}'_{\hat{\kappa}} \triangleq \widehat{alloc_k}(\hat{\varsigma}, (e_f\ e_0\ e_s\ ...), \widehat{ctx}', \hat{a}_{\hat{\kappa}})$$
$$\hat{\kappa} \triangleq \widehat{\textbf{arg}}([e_0 : e_s], \widehat{ctx}, \widehat{ctx}', \hat{\kappa})$$
$$\hat{\sigma}'_{\hat{\kappa}} \triangleq \hat{\sigma}_{\hat{\kappa}} \sqcup [\hat{a}'_{\hat{\kappa}} \mapsto \hat{\kappa}]$$

Fig. 3. Rules to evaluate syntax into smaller expressions

Atomic Evaluation

$$E\langle æ, \_, \hat{\sigma}, \hat{a}_{\hat{\kappa}}\rangle \rightsquigarrow A\langle \hat{v}, \hat{\sigma}, \hat{a}_{\hat{\kappa}}\rangle$$
$$\text{where } \hat{v} \triangleq \widehat{\mathcal{A}}(\hat{\varsigma}) \qquad \textbf{(E-AE)}$$

$$\widehat{\mathcal{A}} :: \widehat{Eval} \to \hat{v}$$
$$\widehat{\mathcal{A}}(b, \_, \_) \triangleq b \quad \widehat{\mathcal{A}}(n, \_, \_) \triangleq n$$
$$\widehat{\mathcal{A}}(x, \widehat{ctx}, \_) \stackrel{\triangle}{\ni} \hat{\sigma}_{\hat{v}}(x, \widehat{ctx})$$
$$\widehat{\mathcal{A}}(lam, \widehat{ctx}, \_) \triangleq (lam, \widehat{ctx})$$

Fig. 4. Atomic Evaluation, which converts syntax into a value.

Apply rules may be triggered at the same time. For example, if an address contains multiple values, then both **A-If** rules must be triggered. In Figure 6, both branches must be taken to soundly approximate this program. In the Figure 6 example, the inner let calls the same function twice, which, in the same context, binds the address x to both `#t` and `#f`. Then, when a is referenced, it contains both possibilities, so both branches will be taken.

The **A-Let** rule will be applied once for every binding in a `let` expression. Because the addresses were calculated in the **E-Let** rule, not much work needs to be done in this rule. However, because

Application Rules

$$A\langle \hat{v}, \hat{\sigma}, \hat{a}_{\hat{\kappa}} \rangle \rightsquigarrow E\langle e_t, \widehat{ctx}, \hat{\sigma}, \hat{a}'_{\hat{\kappa}} \rangle$$
$$\text{where } \hat{\sigma}_{\hat{\kappa}}(\hat{a}_{\hat{\kappa}}) \ni \widehat{\mathbf{ifk}}(e_t, \_, \widehat{ctx}, \hat{a}'_{\hat{\kappa}}) \quad \textbf{(A-IfT)}$$
$$\hat{v} \neq \#\mathbf{f}$$

$$A\langle \hat{v}, \hat{\sigma}, \hat{a}_{\hat{\kappa}} \rangle \rightsquigarrow E\langle e_f, \widehat{ctx}, \hat{\sigma}, \hat{a}'_{\hat{\kappa}} \rangle$$
$$\text{where } \hat{\sigma}_{\hat{\kappa}}(\hat{a}_{\hat{\kappa}}) \ni \widehat{\mathbf{ifk}}(\_, e_f, \widehat{ctx}, \hat{a}'_{\hat{\kappa}}) \quad \textbf{(A-IfF)}$$
$$\hat{v} = \#\mathbf{f}$$

$$A\langle \hat{v}, \hat{\sigma}, \hat{a}_{\hat{\kappa}} \rangle \rightsquigarrow E\langle e_b, \widehat{ctx}, \hat{\sigma}', \hat{a}'_{\hat{\kappa}} \rangle$$
$$\text{where } \hat{\sigma}_{\hat{\kappa}}(\hat{a}_{\hat{\kappa}}) \ni \widehat{\mathbf{let}}(e_b, \hat{a}_v, \widehat{ctx}, \hat{a}'_{\hat{\kappa}}) \quad \textbf{(A-Let)}$$
$$\hat{\sigma}'_{\hat{v}} \triangleq \hat{\sigma}_{\hat{v}} \sqcup [\hat{a}_v \mapsto \hat{v}]$$

$$A\langle \hat{v}, \hat{\sigma}, \hat{a}_{\hat{\kappa}} \rangle \rightsquigarrow A\langle -42, \hat{\sigma}, \hat{a}'_{\hat{\kappa}} \rangle$$
$$\text{where } \hat{\sigma}_{\hat{\kappa}}(\hat{a}_{\hat{\kappa}}) \ni \widehat{\mathbf{setk}}(\hat{a}_v, \hat{a}'_{\hat{\kappa}}) \quad \textbf{(A-Set!)}$$
$$\hat{\sigma}'_{\hat{v}} \triangleq \hat{\sigma}_{\hat{v}} \sqcup [\hat{a}_v \mapsto \hat{v}]$$

$$A\langle \hat{v}, \hat{\sigma}, \hat{a}_{\hat{\kappa}} \rangle \rightsquigarrow E\langle e, \widehat{ctx}, \hat{\sigma}', \hat{a}''_{\hat{\kappa}} \rangle$$
$$\text{where } \hat{\sigma}_{\hat{\kappa}}(\hat{a}_{\hat{\kappa}}) \ni \widehat{\mathbf{p1}}(op, e, \widehat{ctx}, \hat{a}'_{\hat{\kappa}}) \quad \textbf{(A-Prim1)}$$
$$\hat{a}''_{\hat{\kappa}} \triangleq \widehat{alloc_k}(\hat{\varsigma}, e, \widehat{ctx})$$
$$\hat{\kappa} \triangleq \widehat{\mathbf{p2}}(op, \hat{v}, \hat{a}'_{\hat{\kappa}})$$
$$\hat{\sigma}'_{\hat{\kappa}} \triangleq \hat{\sigma}_{\hat{\kappa}} \sqcup [\hat{a}''_{\hat{\kappa}} \mapsto \hat{\kappa}]$$

$$A\langle \hat{v}', \hat{\sigma}, \hat{a}_{\hat{\kappa}} \rangle \rightsquigarrow A\langle \mathbf{primVal}(op, \hat{v}, \hat{v}'), \hat{\sigma}, \hat{a}'_{\hat{\kappa}} \rangle$$
$$\text{where } \hat{\sigma}_{\hat{\kappa}}(\hat{a}_{\hat{\kappa}}) \ni \widehat{\mathbf{p2}}(op, v, \hat{a}'_{\hat{\kappa}}) \quad \textbf{(A-Prim2)}$$

$$A\langle \hat{v}, \hat{\sigma}, \hat{a}_{\hat{\kappa}} \rangle \rightsquigarrow E\langle e_b, \widehat{ctx}, \hat{\sigma}'', \hat{a}'_{\hat{\kappa}} \rangle$$
$$\text{where } \hat{\sigma}_{\hat{\kappa}}(\hat{a}_{\hat{\kappa}}) \ni \widehat{\mathbf{callcck}}(\widehat{ctx}, \hat{a}'_{\hat{\kappa}}) \quad \textbf{(A-C/cc)}$$
$$\hat{v} = ((\lambda\ (x)\ e_b), \widehat{ctx}_{\widehat{clo}})$$
$$\hat{a}_v \triangleq \widehat{alloc_v}(x, \hat{\varsigma})$$
$$\hat{\sigma}'_{\hat{v}} \triangleq \widehat{copy}(\widehat{ctx}_{\widehat{clo}}, \widehat{ctx})$$
$$\hat{\sigma}''_{\hat{v}} \triangleq \hat{\sigma}_{\hat{v}} \sqcup [\hat{a}_v \mapsto \hat{a}_{\hat{\kappa}}]$$

$$A\langle \hat{a}_{\hat{\kappa}}, \hat{\sigma}, \hat{a}'_{\hat{\kappa}} \rangle \rightsquigarrow A\langle \hat{a}'_{\hat{\kappa}}, \hat{\sigma}, \hat{a}_{\hat{\kappa}} \rangle$$
$$\text{where } \hat{\sigma}_{\hat{\kappa}}(\hat{a}'_{\hat{\kappa}}) \ni \widehat{\mathbf{callcck}}(\_, \_) \quad \textbf{(A-C/ccKont)}$$

$$A\langle \hat{v}, \hat{\sigma}, \hat{a}_{\hat{\kappa}} \rangle \rightsquigarrow E\langle e_i, \widehat{ctx}, \hat{\sigma}', \hat{a}'_{\hat{\kappa}} \rangle$$
$$\text{where } \hat{\sigma}_{\hat{\kappa}}(\hat{a}_{\hat{\kappa}}) \ni \widehat{\mathbf{arg}}(e_s, \widehat{ctx}, \widehat{ctx}', \hat{a}'_{\hat{\kappa}}) \quad \textbf{(A-Ar)}$$
$$e_i \in e_s$$
$$\hat{a}'_{\hat{\kappa}} \triangleq \widehat{alloc_k}(\hat{\varsigma}, e_i, \widehat{ctx})$$
$$\hat{\kappa} \triangleq \widehat{\mathbf{fn}}(\hat{v}, i, \widehat{ctx}', \hat{a}'_{\hat{\kappa}})$$
$$\hat{\sigma}'_{\hat{\kappa}} \triangleq \hat{\sigma}_{\hat{\kappa}} \sqcup [\hat{a}_v \mapsto \hat{\kappa}]$$

$$A\langle \hat{v}, \hat{\sigma}, \hat{a}_{\hat{\kappa}} \rangle \rightsquigarrow E\langle e_b, \widehat{ctx}, \hat{\sigma}'', \hat{a}'_{\hat{\kappa}} \rangle$$
$$\text{where } \hat{\sigma}_{\hat{\kappa}}(\hat{a}_{\hat{\kappa}}) \ni \widehat{\mathbf{fn}}(\widehat{clo}, n, \widehat{ctx}, \hat{a}'_{\hat{\kappa}}) \quad \textbf{(A-Call)}$$
$$\widehat{clo} = ((\lambda\ (x_s \ldots)\ e_b), \widehat{ctx}_{\widehat{clo}})$$
$$\hat{a}_v \triangleq \widehat{alloc_v}(x_n, \hat{\varsigma})$$
$$\hat{\sigma}'_{\hat{v}} \triangleq \widehat{copy}(\widehat{ctx}_{\widehat{clo}}, \widehat{ctx})$$
$$\hat{\sigma}''_{\hat{v}} \triangleq \hat{\sigma}_{\hat{v}} \sqcup [\hat{a}_v \mapsto \hat{v}]$$

$$A\langle \hat{v}, \hat{\sigma}, \hat{a}_{\hat{\kappa}} \rangle \rightsquigarrow A\langle \hat{v}, \hat{\sigma}, \hat{a}'_{\hat{\kappa}} \rangle$$
$$\text{where } \hat{\sigma}_{\hat{\kappa}}(\hat{a}_{\hat{\kappa}}) \ni \widehat{\mathbf{fn}}(\hat{a}'_{\hat{\kappa}}, \_, \_, \_) \quad \textbf{(A-CallKont)}$$

Fig. 5. Value Application Rules

```
(let ([f (lambda (x) x)])
  (let ([a (f #t)] [b (f #f)])
    (if a 4 5)))
```

Fig. 6. An example where both branches of a condition must be taken.

every $\widehat{\mathbf{let}}$ continuation results in the same state, only one output state is generated for the body of the expression.

Creating new contexts is only done when variables are bound. Therefore, we only need to create a context in the **E-Let**, **E-C/cc**, and **E-Call** rules. The new context is used to evaluate the expression containing the new bindings. Therefore, the new context only transitions in the corresponding *Apply* rule.

Helper Functions

$$\hat{\mathcal{I}} :: \mathsf{Exp} \to \hat{\Sigma}$$
$$\hat{\mathcal{I}}(e) \triangleq E\langle e, \epsilon, (\bot, \hat{\sigma}_{\hat{\kappa}}), (e, \epsilon)\rangle$$
$$\hat{\sigma}_{\hat{\kappa}} \triangleq \bot \sqcup [(e, \epsilon) \mapsto \widehat{\mathbf{mtk}}]$$

$$\widehat{alloc_v} :: \mathsf{Var} \times \hat{\Sigma} \rightharpoonup \widehat{VAddr}$$
$$\widehat{alloc_v}(x, E\langle \_, \widehat{ctx}, \_, \rangle\rangle \triangleq (x, \widehat{ctx})$$
$$\widehat{alloc_v}(x, A\langle(\_, \widehat{ctx}), \_, \rangle\rangle \triangleq (x, \widehat{ctx})$$

$$\widehat{new} :: \widehat{Eval} \to \widehat{Context}$$
$$\widehat{new}(e, \widehat{ctx}, \_) \triangleq \lfloor e : context \rfloor_m$$

$$\widehat{alloc_k} :: \hat{\Sigma} \times \mathsf{Exp} \times \widehat{Context} \rightharpoonup \widehat{KAddr}$$
$$\widehat{alloc_k}(\_, e, \widehat{ctx}) \triangleq (e, \widehat{ctx})$$

Fig. 7. Auxiliary functions

Transition Relation and Global Store

$$(\rightsquigarrow) :: \hat{\Sigma} \rightharpoonup \mathcal{P}(\hat{\Sigma})$$
$$(\rightsquigarrow_{\hat{\Xi}}) :: \hat{\Xi} \rightharpoonup \mathcal{P}(\hat{\Xi})$$
$$(\hat{r}, \hat{\sigma}) \rightsquigarrow_{\hat{\Xi}} (\hat{r}', \hat{\sigma}')$$
$$\text{where } \hat{s} \triangleq \{\hat{\varsigma} \mid (r, \hat{\sigma}) \rightsquigarrow \hat{\varsigma}\} \cup \{\hat{\mathcal{I}}(e_o)$$
$$\hat{r}' \triangleq \{r \mid (r, \_) \in \hat{s}\}$$
$$\hat{\sigma}' \triangleq \bigsqcup_{(\_, \hat{\sigma}'') \in \hat{s}} \hat{\sigma}''$$

$$\hat{\xi} \in \hat{\Xi} \triangleq \hat{R} \times \widehat{Store}$$
$$\hat{r} \in \hat{R} \triangleq \mathcal{P}(\hat{C})$$
$$\hat{c} \in \hat{C} \triangleq CE\langle \mathsf{Exp} \times \widehat{Context} \times \widehat{KAddr}\rangle$$
$$+ CA\langle \widehat{Val} \times \widehat{KAddr}\rangle$$

Fig. 8. Transition Relation and Global Store

The value and continuation stores are necessarily responsible for an exponential growth of states in the state space. One method to maintain polynomial worst-case complexity when evaluating closures is to separate the stores from the states, and to globalize the stores. After each state transition, the stores are combined and used for the next transitions. Figure 8 gives a collecting semantics which transforms a standard machine with an inline store into a global store. The semantics utilize the injection function $\hat{\mathcal{I}}$, along with the inline store transition function $\rightsquigarrow$.

4 DATALOG IMPLEMENTATION

Soufflé is used as the Datalog engine for its state-of-the-art runtime performance, parallelism, and language features. Algebraic Data Types (ADT) are utilized for contexts, value and continuation types to maintain brevity. This implementation can be replicated without ADTs but it will be much more verbose.

Implementing the above operational semantics as a Datalog program is a straightforward process. Many rules can be converted in a near 1:1 fashion. However, there are some unique relations that differ from the operational semantics. Still, the rules of the machine map closely to the implementation in Datalog. Figure 9 shows the similarity between the two.

Datalog programs being split into rules maps well onto operational semantics. There is a correspondence between the operational semantics and the Datalog implementation. Datalog places the

```
state_e(eguard, ctx, ak),
stored_kont(ak, kont),
flow_ee(e, eguard) :-
    state_e(e, ctx, ak),
    if(e, eguard, et, ef),
    ak = $KAddress(eguard, ctx),
    kont = $If(et, ef, ctx, ak).
```

$$E\langle(\texttt{if}\ e_g\ e_t\ e_f), \widehat{ctx}, \hat{\sigma}, \hat{a}_{\hat{\kappa}}\rangle \rightsquigarrow E\langle e_g, \widehat{ctx}, \hat{\sigma}', \hat{a}'_{\hat{\kappa}}\rangle$$
$$\text{where } \hat{a}'_{\hat{\kappa}} \triangleq \widehat{alloc_k}(\hat{\varsigma}, e_c, \widehat{ctx}) \qquad \textbf{(E-If)}$$
$$\hat{\kappa} \triangleq \widehat{\textbf{ifk}}(e_t, e_f, \widehat{ctx}, \hat{a}_{\hat{\kappa}})$$
$$\hat{\sigma}'_{\hat{\kappa}} \triangleq \hat{\sigma}_{\hat{\kappa}} \sqcup [\hat{a}'_{\hat{\kappa}} \mapsto \hat{\kappa}]$$

Fig. 9. If expression evaluation in Datalog and in the operational semantics.

```
peek_ctx(e, old_ctx, new_ctx) :-
    state_e(e, old_ctx, _),
    ( callcc(e, _) ; call(e, _, _)
    ; let(e, _, _); lambda(e,_,_)),
    old_ctx = $Context(ctx1,ctx0),
    new_ctx = $Context(e,ctx1).
```

```
stored_val(av, v) :-
    copy_ctx(from, to, e),
    freevar(fv, e),
    stored_val(av, v),
    av = $VAddress(fv, to).
```

Fig. 10. The new and copy function analogues in the Datalog implementation.

head of the Horn clause before the body, so the output state is on top, along with the continuation being added to the store. The body of the Datalog rule acts as the input. As rules are computed, new facts are generated by the heads of the clauses. As a result, more rules can be executed, extending the set of total facts.

Figure 10 shows two interesting relations. These highlight a key difference between the operational semantics and the Datalog implementation. On the left is the equivalent to $\widehat{new}$: `peek_ctx`. On the right is the implementation of $\widehat{copy}$. In Datalog, we compute a helper relation, `peek_ctx`, which computes a context as needed. The `copy_ctx` relation will signal a copy operation, from a source to a destination context. When a `copy_ctx` fact is added, it does not immediately do the copy, but it will trigger an inference rule for the `stored_val` relation.

See Appendix A for the full Datalog implementation.

5 EVALUATION

In this section we present an evaluation of our implementation of m-CFA in Soufflé. To gain confidence in our implementation's correctness, we performed manual validation on a set of testcases. To measure scalability and complexity in practice, we construct a family of terms that exhibit worst-case (polynomial) behavior. We used large terms in this family and ran experiments on a 28-core Linux server with 78GB of RAM.

5.1 Constructing worst-case terms for m-CFA

In order to both evaluate the performance and verify the correctness of our implementation, we construct a family of terms that incur maximum-possible work in practice. Van Horn's dissertation details a construction of terms whose analysis require exponential work for k-CFA [18]. Figure 11 shows an example for 1-CFA. The ultimate issue is that `#t` and `#f` will be conflated through the call to `f`, producing two runtime closures but four abstract closures; more bindings, or arguments to w, may be added to add further complexity.

We use Van Horn's technique to generate high-complexity terms for our m-CFA implementation. The key trick to fool [k=1]-CFA from Van Horn's example is the application of the identity function

```
((lambda (f)
   (let ((m (f #t))
         (n (f #f)))
     m))
 (lambda (z)
   ((lambda (x) x)
    (lambda (w) (w z z)))))
```

Fig. 11. Example (from Van Horn) showing exponential behavior for k-CFA

```
((lambda (f)
   (let ((mm (f M)
             ...
             (m1 (f 1))
             (n0 (f 0)))
         m))
   (lambda (z)
     ((lambda (x) (+ z (+ z ...)))
      (lambda (x) x)))))
```

Fig. 12. An example term from our experiments.

Table 1. Running time and memory usage of m-CFA in Datalog. Term size N/K means N calls to `f` and K invocations of `+`.

		0 *padding*		1 *padding*		2 *padding*	
Term size	Polyvariance (m)	Time	Memory	Time	Memory	Time	Memeory
32/4	0	00:09:57	1.27GB	00:09:46	1.86GB	09:48.04	1.27GB
	1	< 1 sec	12.1MB	00:22:49	1.28GB	13:26.88	1.27GB
	2	< 1 sec	12.29MB	< 1 sec	12.5MB	19:09.89	1.27GB
86/3	0	01:08:58	3.55GB	01:09:36	3.55GB	01:02:51	3.56GB
	1	< 1 sec	6.31MB	01:34:37	3.55GB	01:32:10	3.56GB
	2	< 1 sec	6.31MB	< 1 sec	6.32MB	02:13:10	3.56GB

inside the term. This identity function acts as *padding*, as [k=1]-CFA traces only the *most recent* call site. During concrete execution, the intermediate call to the identity function sits (on the stack) in front of the (separate) calls to `f`. In [k=1]-CFA, only the most recent callsite is remembered and thus `#t` and `#f` will flow to same address. However we cannot directly translatiterate this example into m-CFA as we will not see the expected storage blowup. In m-CFA, the context (the contour in k-CFA) only grows in the state when values are bound to a variable. If padding is in function postion when λ `(w) (w x)` is evaluated, the context has not been extended yet. Using Van Horn's example with m-CFA, the padding will be bypassed. In m-CFA, the padding should be moved into argument position and η-expand the precision losing term. This is shown in Figure 12. This modification forces conflation of the values after the padding label is appended into the context.

Table 2. Parallel Performance of m-CFA Soufflé implementation. 32 different clauses in let with 2 padding

	1 core		2 threads		4 threads		8 threads	
m	time	memory	time	memory	time	memeory	time	memory
0	00:09:48	1.27GB	00:13:05	1.43GB	00:18:39	1.55GB	00:22:38	1.61GB
1	00:13:26	1.27GB	00:14:36	1.46GB	00:20:36	1.55GB	00:22:54	1.62GB
2	00:19:09	1.27GB	00:25:14	1.46GB	00:36:11	1.54GB	00:46:57	1.60GB

5.2 Results

We used our worst-case term construction to perform a variety of runs using Soufflé. Table 1 shows the results of our system on a set of two terms: one with 32 `let` bindings and one with 86 `let` bindings. We measure 0, 1, and 2-CFA using a variety of paddings (0, 1, and 2). By construction, we expect terms to explode when the amount of padding is too low as the analysis begins to conflate polynomially-greater callsites. For example, in 32/4, we see 0-CFA explode while 1 and 2-CFA are very fast. This is because both 1 and 2-CFA are fully-precise for the term. As expected, analyses of higher-complexity have longer runtimes. For example, looking at the timings for 2 padding, we can see that 2-CFA is roughly twice as slow as 0-CFA.

We measured parallel performance using a variety of thread counts. Soufflé was built to support scalability and multithreading to speed up parallel execution on a single node. In our analysis, we believed there would be many potentially-parallelizable states. For example, we evaluate different `let` clauses non-deterministically. We used Soufflé version 2.02, compiling from source with OpenMP support enabled. Our results were all compiled to `C++` via Soufflé's `-c` flag. We verified CPU utilization (via `htop`) to ensure multithreading was enabled. However, as detailed in Table 2, our results demonstrated anti-scalability: instead of making the program run faster, the more cores used, the slower execution we observed, while also adding more memory overhead. GitHub issues from the Soufflé authors point to several potential reasons for poor scalability. For example, our implementation uses a large number of rules, and Soufflé does not parallelize across rules.

6 CONCLUSION

Datalog-based implementation of static analysis tools has enabled new frontiers in the scalability of analyses to object-oriented languages. However, we do not know of any presentations that extend these ideas to Scheme-like languages. This paper presented the key ideas necessary to implement analyses of Scheme-like languages using Datalog. We structure our analysis using the AAM-based approach to facilitate translation to Datalog's deductive rules, using the m-CFA allocation strategy of Might et al. to mirror the flat closure structure naturally enabled by Datalog [11]. To our knowledge, this is the first presentation of a Datalog-based analysis for Scheme-like languages.

REFERENCES

[1] George Balatsouras, Kostas Ferles, George Kastrinis, and Yannis Smaragdakis. 2017. A Datalog Model of Must-Alias Analysis. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Barcelona, Spain) *(SOAP 2017)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3088515.3088517

[2] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[3] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*.

243–262.
[4] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. 1989. What you always wanted to know about Datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering* 1, 1 (1989), 146–166.
[5] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252.
[6] Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32. http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf
[7] Thomas Gilray, Michael D Adams, and Matthew Might. 2016. Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 407–420.
[8] Thomas Gilray, Steven Lyde, Michael D Adams, Matthew Might, and David Van Horn. 2016. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 691–704.
[9] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-Sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 423–434. https://doi.org/10.1145/2491956.2462191
[10] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945
[11] Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 305–315.
[12] Antoine Miné. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Found. Trends Program. Lang.* 4, 3–4 (Dec. 2017), 120–372. https://doi.org/10.1561/2500000034
[13] Flemming Nielson, Hanne Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. https://doi.org/10.1007/978-3-662-03811-6
[14] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-Scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/2892208.2892226
[15] Olin Shivers. 2004. Higher-Order Control-Flow Analysis in Retrospect: Lessons Learned, Lessons Abandoned. *SIGPLAN Not.* 39, 4 (April 2004), 257–269. https://doi.org/10.1145/989393.989421
[16] Olin Grigsby Shivers. 1991. *Control-flow analysis of higher-order languages or taming lambda*. Carnegie Mellon University.
[17] Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285 – 309. https://doi.org/pjm/1103044538
[18] David Van Horn and Harry G. Mairson. 2008. Deciding kCFA is Complete for EXPTIME. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 275–282. https://doi.org/10.1145/1411204.1411243
[19] David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. 51–62.

A FULL SOUFFLÉ IMPLEMENTATION

```
.type id <: symbol

.type context = Context{ctx0:id}

.type value = Number { n : number }
              | Bool { b : symbol }
              | Kont { k : address_k }
              | Closure { e: id, ctx: context }
              | PrimVal {op: id , v1: value , v2: value}

.type kont = MT {}
           | Arg {args: id, ctx: context, ectx: context, next_ak: address_k}
           | Fn {fn: value, pos: number, ctx: context, next_ak: address_k}
           | Set {loc: address_v, next_ak: address_k}
           | If {true_branch: id, false_branch: id, ctx: context, next_ak: address_k}
           | Callcc {ectx: context, next_ak: address_k}
           | Let {av: address_v, ebody: id, ctx: context, next_ak: address_k}
           | Prim1 {op: id, e2: id, ctx: context, next_ak: address_k}
           | Prim2 {op: id, v1: value, next_ak: address_k}
```

```
.type address_k = KAddress{e: id, ctx: context}
.type address_v = VAddress{x: symbol, ctx: context}

.decl state_e(e: id, ctx: context, ak: address_k)
.output state_e

.decl state_a(v: value, ak: address_k)
.output state_a

.decl stored_val(av: address_v, v: value)
.output stored_val

.decl stored_kont(ak: address_k, k: kont)
.output stored_kont

.decl lambda(Id: id, Vars: id, BodyId: id)
.input lambda

.decl lambda_arg_list(Id: id, Pos: number, X: symbol)
.input lambda_arg_list

.decl prim(Id: id, OpName: symbol)
.input prim

.decl prim_call(Id: id, PrimId: id, Args: id)
.input prim_call

.decl call(Id: id, FuncId: id, Args: id)
.input call

.decl call_arg_list(Id: id, Pos: number, X: id)
.input call_arg_list

.decl var(Id: id, MetaName: symbol)
.input var

.decl num(Id: id, v: number)
.input num

.decl bool(Id: id, v: symbol)
.input bool

.decl quotation(Id: id, Expr: id)
.input quotation

.decl value_form(Id: id)
value_form(id) :-
    (num(id, _); var(id, _); lambda(id, _, _); quotation(id, _); bool(id, _)).

.decl if(Id:id, GuardId: id, TrueId: id, False: id)
.input if

.decl setb(Id: id, Var: symbol, ExprId: id)
.input setb

.decl callcc(Id: id, ExprId: id)
.input callcc

.decl let(Id: id, BindId: id, BodyId: id)
.input let

.decl let_list(Id: id, X: symbol, EId: id)
.input let_list

.decl top_exp(Id: id)
.input top_exp

.decl freevar(x:symbol, e: id)
.output freevar

freevar(x, e) :- var(e, x).
freevar(x, e) :-
    lambda(e, vars, body),
    freevar(x, body),
    !lambda_arg_list(vars, _, x).

freevar(x, e) :-
    call(e, func, args),
    (freevar(x, func); freevar(x, args)).
```

```
freevar(x, e) :-
    prim_call(e, _, args),
    freevar(x, args).

freevar(x, e) :-
    call_arg_list(e, pos, arg),
    freevar(x, arg).

freevar(x, e) :-
    if(e, eguard, et, ef),
    (freevar(x, eguard); freevar(x, et); freevar(x, ef)).

freevar(y, e) :-
    setb(e, _, ev),
    freevar(y, ev).

freevar(x, e) :-
    callcc(e, ev),
    freevar(x, ev).

freevar(x, e) :-
    let(e, binds, body),
    (freevar(x, binds); freevar(x, body)).

freevar(x, e) :-
    let_list(e, a, bind),
    !let_list(e, x, _),
    freevar(x, bind).

.decl flow_ee(e1: id, e2: id)
.output flow_ee
.decl flow_ea(e1: id, a2: value)
.output flow_ea
.decl flow_aa(a1: value, a2: value)
.output flow_aa
.decl flow_ae(a1: value, e2: id)
.output flow_ae

.decl peek_ctx(e: id, ctx_old: context, ctx_new: context)
.output peek_ctx

.decl copy_ctx(from: context, to: context, e:id)
.output copy_ctx

state_e(e, $Context(""), $KAddress(e, $Context(""))),
peek_ctx(e, $Context(""), $Context(e)),
stored_kont($KAddress(e, $Context("")), $MT) :-
    top_exp(e).

peek_ctx(e, $Context(ctx0), $Context(e)) :-
    state_e(e, $Context(ctx0), _),
    (callcc(e, _) ; call(e, _, _)
    ; let(e, _, _); lambda(e,_,_)).

stored_val($VAddress(fv, to), v) :-
    copy_ctx(from, to, e),
    freevar(fv, e),
    stored_val($VAddress(fv, from), v).

state_e(eguard, ctx, $KAddress(eguard, ctx)),
stored_kont($KAddress(eguard, ctx), $If(et, ef, ctx, ak)),
flow_ee(e, eguard) :-
    state_e(e, ctx, ak),
    if(e, eguard, et, ef).

state_e(elam, ctx, $KAddress(elam, ctx)),
stored_kont($KAddress(elam, ctx), $Callcc(ectx, ak)),
flow_ee(e, elam) :-
    state_e(e, ctx, ak),
    callcc(e, elam),
    peek_ctx(e, ctx, ectx).

state_e(esetto, ctx, $KAddress(esetto, ctx)),
stored_kont($KAddress(esetto, ctx), $Set($VAddress(x, ctx), ak)),
flow_ee(e, esetto) :-
    state_e(e, ctx, ak),
    setb(e, x, esetto).

state_e(efunc, ctx, $KAddress(efunc, ctx)),
stored_kont($KAddress(efunc, ctx), $Arg(eargs, ctx, ectx, ak)),
flow_ee(e, efunc) :-
```

```
    state_e(e, ctx, ak),
    call(e, efunc, eargs),
    peek_ctx(e, ctx, ectx).

state_e(ebnd, ctx, $KAddress(ebnd, ctx)),
stored_kont($KAddress(ebnd, ctx), $Let($VAddress(x, ectx), ebody, ectx, ak)),
copy_ctx(ctx, ectx, e),
flow_ee(e, ebnd) :-
    state_e(e, ctx, ak),
    let(e, ll, ebody),
    let_list(ll, x, ebnd),
    peek_ctx(e, ctx, ectx).

state_e(earg0, ctx, $KAddress(earg0, ctx)),
stored_kont($KAddress(earg0, ctx), $Prim1(op, earg1, ctx, ak)),
flow_ee(e, earg0) :-
    state_e(e, ctx, ak),
    prim_call(e, op, pl),
    call_arg_list(pl, 0, earg0),
    call_arg_list(pl, 1, earg1).

state_a($Number(n), ak),
flow_ea(e, $Number(n)) :-
    state_e(e, ctx, ak),
    num(e, n).

state_a($Bool(b), ak),
flow_ea(e, $Bool(b)) :-
    state_e(e, ctx, ak),
    bool(e, b).

state_a($Closure(e, ctx), ak),
flow_ea(e, $Closure(e, ctx)) :-
    state_e(e, ctx, ak),
    lambda(e, _, _).

state_a(v, ak),
flow_ea(e, v) :-
    state_e(e, ctx, ak),
    var(e, x),
    stored_val($VAddress(x, ctx), v).

state_e(et, ctx_k, next_ak),
flow_ae($Bool("#t"), et) :-
    (state_a($Bool("#t"), ak) ; state_a($Closure(_,_), ak)
        ; state_a($Number(_), ak) ; state_a($Kont(_), ak)),
    stored_kont(ak, $If(et, _, ctx_k, next_ak)).

state_e(ef, ctx_k, next_ak),
flow_ae($Bool("#f"), ef) :-
    state_a($Bool("#f"), ak),
    stored_kont(ak, $If(_, ef, ctx_k, next_ak)).

state_e(ebody, ectx, next_ak),
stored_val($VAddress(x, ectx), $Kont(ak)),
copy_ctx(ctx_clo, ectx, elam),
flow_ae($Closure(elam, ctx_clo), ebody) :-
    state_a($Closure(elam, ctx_clo), ak),
    stored_kont(ak, $Callcc(ectx, next_ak)),
    lambda(elam, params, ebody),
    lambda_arg_list(params, 0, x).

state_a($Kont(ak), bk),
flow_aa($Kont(bk), $Kont(ak)) :-
    state_a($Kont(bk), ak),
    stored_kont(ak, $Callcc(_, _)).

state_e(earg, ctx, $KAddress(earg, ctx)),
stored_kont($KAddress(earg, ctx), $Fn(v, pos, ectx, next_ak)),
flow_ae(v, earg) :-
    state_a(v, ak),
    stored_kont(ak, $Arg(eargs, ctx, ectx, next_ak)),
    call_arg_list(eargs, pos, earg).

state_e(ebody, ectx, next_ak),
stored_val($VAddress(x, ectx), v),
copy_ctx(ctx_clo, ectx, elam),
flow_ae(v, ebody) :-
    state_a(v, ak),
    stored_kont(ak, $Fn($Closure(elam, ctx_clo), pos, ectx, next_ak)),
    lambda(elam, params, ebody),
```

```
    lambda_arg_list(params, pos, x).

state_a(v, callcc_kont),
flow_aa(v, v) :-
    state_a(v, ak),
    stored_kont(ak, $Fn($Kont(callcc_kont), 0, _, _)).

state_e(ebody, ctx, next_ak),
stored_val(av, v),
flow_ae(v, ebody) :-
    state_a(v, ak),
    stored_kont(ak, $Let(av, ebody, ctx, next_ak)).

state_e(earg1, ctx, $KAddress(earg1, ctx)),
stored_kont($KAddress(earg1, ctx), $Prim2(op, v, next_ak)),
flow_ae(v, earg1) :-
    state_a(v, ak),
    stored_kont(ak, $Prim1(op, earg1, ctx, next_ak)).

state_a($PrimVal(op, v1, v2), next_ak),
flow_aa(v2, $PrimVal(op, v1, v2)) :-
    state_a(v2, ak),
    stored_kont(ak, $Prim2(op, v1, next_ak)).

state_a($Number(-42), next_ak),
stored_val(loc, v),
flow_aa(v, $Number(-42)) :-
    state_a(v, ak),
    stored_kont(ak, $Set(loc, next_ak)).
```