Correctly Optimizing Abstract Abstract Machines

J. Ian Johnson Nicholas Labich Matthew Might David Van Horn

Abstract

The technique of *abstracting abstract machines* (AAM) provides a systematic approach for deriving computable approximations of evaluators that are easily proved sound. This article contributes a complementary step-by-step process for subsequently going from a naive analyzer derived under the AAM approach, to an efficient and correct implementation. The end result of the process is a two to three order-of-magnitude improvement over the systematically derived analyzer, making it competitive with hand-optimized implementations that compute fundamentally less precise results.

1. Introduction

Program analysis provides sound predictive models of program behavior, but in order for such models to be effective, they must be efficiently computable and correct. Past approaches to designing program analysis have often featured abstractions that are far removed from the original language semantics, requiring ingenuity in their construction and effort in their verification. The *abstracting abstract machines* (AAM) approach [31, 33] to deriving program analyses provides an alternative: a systematic way of transforming a programming language semantics in the form of an abstract machine into a family of abstract interpreters. It thus reduces the burden of constructing and verifying the soundness of an abstract interpreter.

By taking a machine-oriented view of computation, AAM makes it possible to design, verify, and implement program analyzers for realistic language features typically considered difficult to model. The approach was originally applied to features such as higher-order functions, stack inspection, exceptions, laziness, first-class continuations, and garbage collection. It has since been used to verify actor- [9] and thread-based [21] parallelism and behavioral contracts [30]; it has been used to model Coq [25], Dalvik [24], Erlang [8], JavaScript [32], and Racket [30].

The primary strength of the approach is that abstract interpreters can be easily derived through a small number of steps from existing machine models. Since the relationships between abstract machines and higher-level semantic models—such as definitional interpreters [28], structured operational semantics [27], and reduction semantics [12]—are well understood [6], it is possible to navigate from these high-level semantic models to sound program analyzers in a systematic way. Moreover, since these analyses so closely resemble a language's interpreter (a) implementing an analysis requires little more than implementing an interpreter, (b) a single implementation can serve as both an interpreter and analyzer, and (c) verifying the correctness of the implementation is straightforward.

Unfortunately, the AAM approach yields analyzers with poor performance relative to hand-optimized analyzers. Our work takes aim squarely at this "efficiency gap," and narrows it in an equally systematic way through a number of simple steps, many of which are inspired by run-time implementation techniques such as laziness and compilation to avoid interpretative overhead. Each of these steps is proven correct, so the end result is an implementation that is trustworthy and efficient.

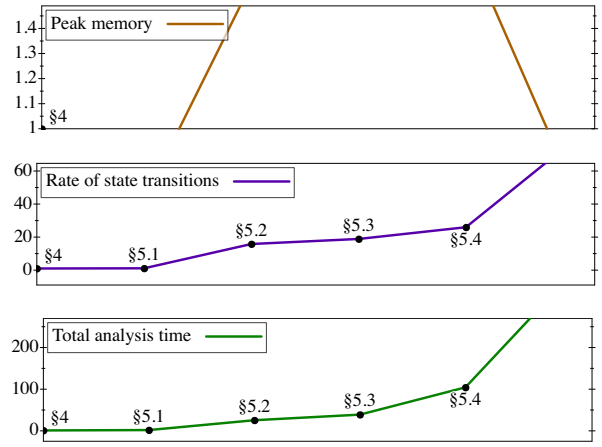


Figure 1. Factor improvements over the baseline analyzer for the Vardoulakis and Shivers benchmark in terms of peak memory usage, the rate of state transitions, and total analysis time. (Bigger is better.) Each point is marked with the section that introduces the optimization.

In this article, we develop a systematic approach to deriving a practical implementation of an abstract-machine-based analyzer using mostly semantic means rather than tricky engineering. Our goal is to empower programming language implementers and researchers to explore and convincingly exhibit their ideas with a low barrier to entry. The optimizations we describe are widely applicable and apparently effective to scale far beyond the size of programs typically considered in the recent literature on flow analysis for functional languages.

2. At a glance

We start with a quick review of the AAM approach to develop an analysis framework and then apply our step-by-step optimization techniques in the simplified setting of a core functional language. This allows us to explicate the optimizations with a minimal amount of inessential technical overhead. Following that, we scale this approach up to an analyzer for a realistic untyped, higher-order imperative language with a number of interesting features and then measure improvements across a suite of benchmarks.

At each step during the initial presentation and development, we evaluated the implementation on a set of benchmarks. The highlighted benchmark in figure 1 is from Vardoulakis and Shivers [34] that tests distributivity of multiplication over addition on Church numerals. For the step-by-step development, this benchmark is particularly informative:

1. it can be written in most modern programming languages,
2. it was designed to stress an analyzer's ability to deal with complicated environment and control structure arising from the use of higher-order functions to encode arithmetic, and
3. its improvement is about median in the benchmark suite considered in section 6, and thus it serves as a good sanity check for each of the optimization techniques considered.

We start, in section 3, by developing an abstract interpreter according to the AAM approach. In the initial abstraction, each state carries a store (what is called per-state store variance). The space of stores is exponential in size; without further abstraction, the analysis is exponential and thus cannot analyze the example in a reasonable amount of time. In section 4, we perform a further abstraction by widening the store. The resulting analyzer sacrifices precision for speed and is able to analyze the example in about 1 minute. This step is described by Van Horn and Might [33, §3.5–6] and is necessary to make even small examples feasible. We therefore take a widened interpreter as the baseline for our evaluation.

Section 5 gives a series of simple abstractions and implementation techniques that, in total, speed up the analysis by nearly a factor of 500, dropping the analysis time to a fraction of a second. Figure 1 shows the step-wise improvement of the analysis time for this example.

The AAM approach, in essence, does the following: it takes a machine-based view of computation and turns it into a *finitary approximation* by bounding the size of the store. With a limited address space, the store must map addresses to *sets* of values. Store updates are interpreted as joins, and store dereferences are interpreted by non-deterministic choice of an element from a set. The result of analyzing a program is a finite directed graph where nodes in the graph are (abstract) machine states and edges denote machine transitions between states.

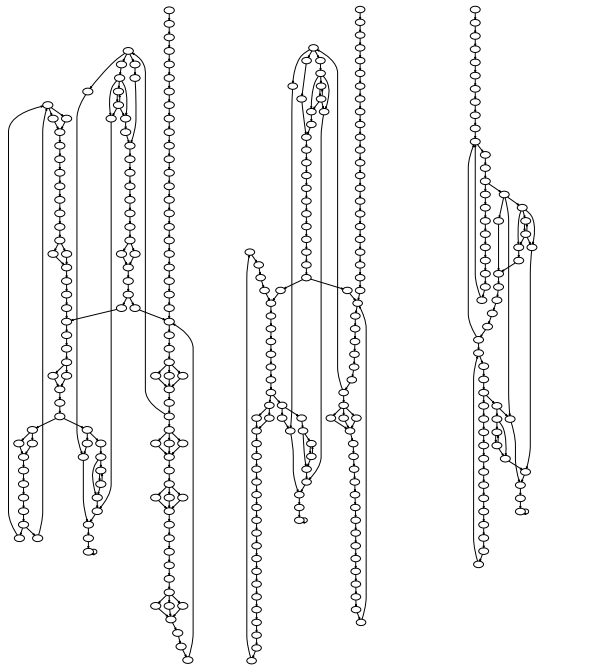
The techniques we propose for optimizing analysis fall into the following categories:

1. generate fewer states by avoiding the eager exploration of non-deterministic choices that will later collapse into a single join point. We accomplish this by applying lazy evaluation techniques so that non-determinism is evaluated *by need*.
2. generate fewer states by avoiding unnecessary, intermediate states of a computation. We accomplish this by applying compilation techniques from functional languages to avoid interpretive overhead in the machine transition system.
3. generate states faster. We accomplish this by better algorithm design in the fixed-point computation we use to generate state graphs.

Figure 2 shows the effect of (1) and (2) for a small example due to Earl, et al. [10]. By generating significantly fewer states at a significantly faster rate, we are able to achieve large performance improvements in terms of both time and space.

Section 6 describes the evaluation of each optimization technique applied to an implementation supporting a more realistic set of features, including mutation, first-class control, compound data, a full numeric tower and many more forms of primitive data and operations. We evaluate this implementation against a set of benchmark programs drawn from the literature. For all benchmarks, the optimized analyzer outperforms the baseline by at least a factor of two to three orders of magnitude.

Section 7 relates this work to the literature and section 8 concludes.



(a) Baseline (b) Lazy (c) Compiled (& lazy)

Figure 2. Example state graphs for the program above. Part (a) shows the result of the baseline analyzer. It has long "corridor" transitions and "diamond" subgraphs that fan-out from nondeterminism and fan-in from joins. Part (b) shows the result of performing nondeterminism lazily and thus avoids many of the diamond subgraphs. Part (c) shows the result of abstract compilation that removes interpretive overhead in the form of intermediate states, thus minimizing the corridor transitions. The end result is a more compact abstraction of the program that can be generated faster.

3. Abstract interpretation of ISWIM

In this section, we give a brief review of the AAM approach by defining a sound analytic framework for a core higher-order functional language: Landin's ISWIM [16]. In the subsequent sections, we will explore optimizations for the analyzer in this simplified setting, but scaling these techniques to realistic languages is straightforward and has been done for the analyzer evaluated in section 6.

ISWIM is a family of programming languages parameterized by a set of base values and operations. To make things concrete, we consider a member of the ISWIM family with integers, booleans, and a few operations. Figure 3 defines the syntax of ISWIM. It includes variables, literals (either integers, booleans, or operations), λ-expressions for defining procedures, procedure applications, and conditionals. Expressions carry a label, ℓ, which is drawn from an unspecified set and denotes the source location of the expression; labels are used to disambiguate distinct, but syntactically identical pieces of syntax. We omit the label annotation in contexts where it is irrelevant.

The semantics are defined in terms of a machine model. The machine components are defined in figure 4; figure 5 defines the transition relation. The evaluation of a program is defined as its set of traces that arise from iterating the machine transition relation. The machine is a very slight variation on a standard abstract machine for ISWIM in "eval, continue, apply" form [6]. It can be systematically derived from a definitional interpreter through a continuation-passing style transformation and defunctionalization,

$$
\begin{array}{lrcl}
\text{Expressions} & e & = & \mathtt{var}^{\ell}(x) \\
 & & | & \mathtt{lit}^{\ell}(l) \\
 & & | & \mathtt{lam}^{\ell}(x, e) \\
 & & | & \mathtt{app}^{\ell}(e, e) \\
 & & | & \mathtt{if}^{\ell}(e, e, e) \\
\text{Variables} & x & = & \mathtt{x} \mid \mathtt{y} \mid \ldots \\
\text{Literals} & l & = & z \mid b \mid o \\
\text{Integers} & z & = & \mathtt{0} \mid \mathtt{1} \mid -\mathtt{1} \mid \ldots \\
\text{Booleans} & b & = & \mathtt{tt} \mid \mathtt{ff} \\
\text{Operations} & o & = & \mathtt{zero?} \mid \mathtt{add1} \mid \mathtt{sub1} \mid \ldots
\end{array}
$$

Figure 3. Syntax of ISWIM

$$
\begin{array}{lrcl}
\text{Values} & v, u & = & \mathtt{clos}\,(x, e, \rho) \mid l \mid \kappa \\
\text{States} & \varsigma & = & \mathtt{ev}\,(e, \rho, \sigma, \kappa) \\
 & & | & \mathtt{co}\,(\kappa, v, \sigma) \\
 & & | & \mathtt{ap}\,(v, v, \sigma, \kappa) \\
\text{Continuations} & \kappa & = & \mathtt{mt} \\
 & & | & \mathtt{fn}\,(v, k) \\
 & & | & \mathtt{ar}\,(e, \rho, k) \\
 & & | & \mathtt{fi}\,(e, e, \rho, k) \\
\text{Addresses} & a, k & \in & \mathit{Addr} \\
\text{Environments} & \rho & \in & \mathit{Var} \rightharpoonup \mathit{Addr} \\
\text{Stores} & \sigma & \in & \mathit{Addr} \rightharpoonup \mathcal{P}(\mathit{Value})
\end{array}
$$

Figure 4. Abstract machine components

$$
\begin{array}{rl}
\multicolumn{2}{r}{\mathit{traces}(e) = \{\mathtt{ev}^{t_0}(e, \varnothing, \varnothing, \mathtt{mt}) \longmapsto\!\!\!\!\!\rightarrow \varsigma\} \text{ where}} \\
\multicolumn{2}{r}{\varsigma \longmapsto \varsigma' \text{ defined to be the following}} \\
\multicolumn{2}{r}{\text{let } t' = \mathit{tick}(\varsigma)} \\
\mathtt{ev}\,(\mathtt{var}\,(x), \rho, \sigma, \kappa) & \longmapsto \mathtt{co}\,(\kappa, v, \sigma) \text{ if } v \in \sigma(\rho(x)) \\
\mathtt{ev}\,(\mathtt{lit}\,(l), \rho, \sigma, \kappa) & \longmapsto \mathtt{co}\,(\kappa, l, \sigma) \\
\mathtt{ev}^t(\mathtt{lam}\,(x, e), \rho, \sigma, \kappa) & \longmapsto \mathtt{co}^{t'}(\kappa, \mathtt{clos}\,(x, e, \rho), \sigma) \\
\mathtt{ev}^t(\mathtt{app}^{\ell}(e_0, e_1), \rho, \sigma, \kappa) & \longmapsto \mathtt{ev}^{t'}(e_0, \rho, \sigma', \mathtt{ar}^t_{\ell}(e_1, \rho, k)) \\
\multicolumn{2}{r}{\text{where } k = \mathit{allockont}^t_{\ell}(\sigma, \kappa)} \\
\multicolumn{2}{r}{\sigma' = \sigma \sqcup [k \mapsto \{\kappa\}]} \\
\mathtt{ev}^t(\mathtt{if}^{\ell}(e_0, e_1, e_2), \rho, \sigma, \kappa) & \longmapsto \mathtt{ev}^{t'}(e_0, \rho, \sigma', \mathtt{fi}^t(e_1, e_2, \rho, k)) \\
\multicolumn{2}{r}{\text{where } k = \mathit{allockont}^t_{\ell}(\sigma, \kappa)} \\
\multicolumn{2}{r}{\sigma' = \sigma \sqcup [k \mapsto \{\kappa\}]} \\
\\
\mathtt{co}\,(\mathtt{mt}, v, \sigma) & \longmapsto \mathtt{ans}\,(\sigma, v) \\
\mathtt{co}\,(\mathtt{ar}^t_{\ell}(e, \rho, k), v, \sigma) & \longmapsto \mathtt{ev}^t(e, \rho, \sigma, \mathtt{fn}^t_{\ell}(v, k)) \\
\mathtt{co}\,(\mathtt{fn}^t_{\ell}(u, k), v, \sigma) & \longmapsto \mathtt{ap}^t_{\ell}(u, v, \kappa, \sigma) \text{ if } \kappa \in \sigma(k) \\
\mathtt{co}\,(\mathtt{fi}^t(e_0, e_1, \rho, k), \mathtt{tt}, \sigma) & \longmapsto \mathtt{ev}^{t'}(e_0, \rho, \sigma, \kappa) \text{ if } \kappa \in \sigma(k) \\
\mathtt{co}\,(\mathtt{fi}^t(e_0, e_1, \rho, k), \mathtt{ff}, \sigma) & \longmapsto \mathtt{ev}^{t'}(e_1, \rho, \sigma, \kappa) \text{ if } \kappa \in \sigma(k) \\
\\
\mathtt{ap}^t_{\ell}(\mathtt{clos}\,(x, e, \rho), v, \sigma, \kappa) & \longmapsto \mathtt{ev}^{t'}(e, \rho', \sigma', \kappa) \\
\multicolumn{2}{r}{\text{where } a = \mathit{alloc}(\varsigma)} \\
\multicolumn{2}{r}{\rho' = \rho[x \mapsto a]} \\
\multicolumn{2}{r}{\sigma' = \sigma \sqcup [a \mapsto \{v\}]} \\
\\
\mathtt{ap}^t_{\ell}(o, v, \sigma, \kappa) & \longmapsto \mathtt{co}\,(\kappa, v', \sigma) \text{ if } v' \in \Delta(o, v)
\end{array}
$$

Figure 5. Abstract abstract machine for ISWIM

or from a structural operational semantics using the refocusing construction of Danvy and Nielsen [7].

Compared with the standard machine semantics, this definition is different in the following ways, which make it abstractable as a program analyzer:

- the store maps addresses to *sets* of values,[1] not single values,
- continuations are heap-allocated, not stack-allocated,
- there are "timestamps" ($t \in \mathit{Time}$) and syntax labels (ℓ) threaded through the computation, and
- the machine is implicitly parameterized by the functions alloc, $\mathit{allockont}$, tick, Δ, and spaces Addr, Time (and initial $t_0 \in \mathit{Time}$).

Concrete interpretation To characterize concrete interpretation, set the implicit parameters of the relation given in figure 5 as follows:

$$\mathit{alloc}(\varsigma) = a \text{ where } a \notin \text{ the } \sigma \text{ within } \varsigma$$
$$\mathit{allockont}^t_{\ell}(\sigma, \kappa) = k \text{ where } k \notin \sigma$$

These functions appear to ignore ℓ and t, but they can be used to determinize the choice of fresh addresses. The $\sqcup$ on stores in the figure is a point-wise lifting of $\cup$: $\sigma \sqcup \sigma' = \lambda a.\sigma(a) \cup \sigma'(a)$. The resulting relation is non-deterministic in its choice of addresses, however it must always choose a fresh address when allocating a continuation or variable binding. If we consider machine states equivalent up to consistent renaming and fix an allocation scheme, this relation defines a deterministic machine (the relation is really a function).

[1] More generally, we can have stores map to any domain that forms a Galois connection with values, enabling Δ to produce elaborate abstractions of base values (e.g., interval or octagon abstractions). We use sets of values for a simpler exposition.

The interpretation of primitive operations is defined by setting Δ as follows:

$$z + 1 \in \Delta(\mathtt{add1}, z) \qquad z - 1 \in \Delta(\mathtt{sub1}, z)$$
$$\mathtt{tt} \in \Delta(\mathtt{zero?}, \mathtt{0}) \qquad \mathtt{ff} \in \Delta(\mathtt{zero?}, z) \text{ if } z \neq \mathtt{0}$$

Abstract interpretation To characterize abstract interpretation, set the implicit parameters just as above, but drop the $a \notin \sigma$ condition. The Δ relation takes some care to not make the analysis run forever; a simple instantiation is a flat abstraction where arithmetic operations return an abstract top element Z, and `zero?` returns both `tt` and `ff` on Z. This family of interpreters is also non-deterministic in choices of addresses, but it is free to choose addresses that are already in use. Consequently, the machines may be non-deterministic when multiple values reside in a store location.

It is important to recognize from this definition that *any* allocation strategy is a sound abstract interpretation [22]. In particular, concrete interpretation is a kind of abstract interpretation. So is an interpretation that allocates a single cell into which all bindings and continuations are stored. The former is an abstract interpretation that is non-computable and gives only the ground truth of a programs behavior; the latter is an abstract interpretation that is easy to compute but gives little information. Useful program analyses lay somewhere in between and can be characterized by their choice of address representation and allocation strategy. Uniform k-CFA [26], presented next, is one such analysis.

Uniform k-CFA To characterize uniform k-CFA, set the allocation strategy as follows, for a fixed constant k:

$$
\begin{aligned}
Time &= Label^* \\
t_0 &= \epsilon \\
alloc(\mathtt{ev}^t(\mathtt{app}^\ell(e_0, e_1), \rho, \sigma, \kappa)) &= \ell t \\
alloc(\mathtt{ap}^t_\ell(\mathtt{clos}\ (x, e, \rho), v, \sigma, \kappa)) &= x\lfloor \ell t \rfloor_k \\
allockont^t_\ell(\sigma, \kappa) &= \ell t \\
tick(\mathtt{ev}^t(e, \rho, \sigma, \kappa)) &= t \\
tick(\mathtt{co}\ (\mathtt{ar}^t(e, \rho, k), v, \sigma)) &= t \\
tick(\mathtt{ap}^t_\ell(u, v, \kappa)) &= \lfloor \ell t \rfloor_k \\
\lfloor t \rfloor_0 &= \lfloor t_0 \rfloor_k = t_0 \\
\lfloor \ell t \rfloor_{k+1} &= \ell \lfloor t \rfloor_k
\end{aligned}
$$

The $\lfloor \cdot \rfloor_k$ notation denotes the truncation of a list of symbols to the leftmost k symbols.

All that remains is the interpretation of primitives. For abstract interpretation, we set Δ to the function that returns $\mathtt{Z}$ on all inputs—a symbolic value we interpret as denoting the set of all integers.

At this point, we have abstracted the original machine to one which has a finite state space for any given program, and thus forms the basis of a sound, computable program analyzer for ISWIM.

4. From machine semantics to baseline analyzer

The uniform k-CFA allocation strategy would make *traces* in figure 5 a computable abstraction of possible executions, but one that is too inefficient to run, even on small examples. Through this section, we explain a succession of approximations to reach a more appropriate baseline analysis. We ground this path by first formulating the analysis in terms of a classic fixed-point computation.

4.1 Static analysis as fixed-point computation

Conceptually, the AAM approach calls for computing an analysis as a graph exploration: (1) start with an initial state, and (2) compute the transitive closure of the transition relation from that state. All visited states are potentially reachable in the concrete, and all paths through the graph are possible traces of execution.

We can cast this exploration process in terms of a fixed-point calculation. Given the initial state ς_0 and the transition relation $\longmapsto$, we define the global transfer function:

$$F_{\varsigma_0} : \wp(State) \times \wp(State \times State) \to \wp(State) \times \wp(State \times State).$$

Internally, this global transfer function computes the successors of all supplied states, and then includes the initial state:

$$
\begin{aligned}
F_{\varsigma_0}(V, E) &= (\{\varsigma_0\} \cup V', E') \\
E' &= \{(\varsigma, \varsigma') \mid \varsigma \in V \text{ and } \varsigma \longmapsto \varsigma'\} \\
V' &= \{\varsigma' \mid (\varsigma, \varsigma') \in E'\}
\end{aligned}
$$

Then, the evaluator for the analysis computes the least fixed-point of the global transfer function: $eval(e) = \mathrm{lfp}(F_{\varsigma_0})$, where $\varsigma_0 = \mathtt{ev}^{t_0}(e, \varnothing, \varnothing, \mathtt{mt})$.

The possible traces of execution tell us the most about a program, so we take $traces(e)$ to be the (regular) set of paths through the computed graph. We elide the construction of the set of edges in this paper.

To conduct this naive exploration on the Vardoulakis and Shivers example would require considerable time. Even though the state space is finite, it is exponential in the size of the program. Even with $k = 0$, there are exponentially many stores in the AAM framework.

In the next subsection, we fix this with store widening to reach polynomial (albeit of high degree) complexity. This widening effectively lifts the store out of individual states to create a single, global shared store for all.

4.2 Store widening

A common technique to accelerate convergence in flow analyses is to share a common, global store. To retain soundness, this store grows monotonically. Formally, we can cast this optimization as a second abstraction or as the application of a widening operator during the fixed-point iteration. In the ISWIM language, such a widening makes 0-CFA quartic in the size of the program. Thus, complexity drops from intractable exponentiality to a merely daunting polynomial.

Since we can cast this optimization as a widening, there is no need to change the transition relation itself. Rather, what changes is the structure of the fixed-point iteration. In each pass, the algorithm will collect all newly produced stores and join them together. Then, before each transition, it installs this joined store into current state.

To describe this process, AAM defined a transformation of the reduction relation so that it operates on a pair of a set of contexts (C) and a store (σ). A context includes all non-store components, *e.g.*, the expression, the environment and the stack. The transformed relation, $\widehat{\longmapsto}$, is

$$
\begin{aligned}
(C, \sigma) &\widehat{\longmapsto} (C', \sigma'), \\
\text{where } C' &= \{c' \mid wn(c, \sigma) \longmapsto wn(c', \sigma^c), c \in C\} \\
\sigma' &= \bigsqcup \{\sigma^c \mid wn(c, \sigma) \longmapsto wn(c', \sigma^c), c \in C\} \\
wn &: Context \times Store \to State \\
wn(\mathtt{ev}\ (e, \rho, \kappa), \sigma) &= \mathtt{ev}\ (e, \rho, \sigma, \kappa) \\
wn(\mathtt{co}\ (v, \kappa), \sigma) &= \mathtt{co}\ (v, \kappa, \sigma) \\
wn(\mathtt{ap}\ (u, v, \kappa), \sigma) &= \mathtt{ap}\ (u, v, \sigma, \kappa) \\
wn(\mathtt{ans}\ (v), \sigma) &= \mathtt{ans}\ (\sigma, v)
\end{aligned}
$$

The new store is computed as the least upper bound of all stores after stepping in order to have only one, and keep the semantics sound.

4.3 Store-allocate all values

The final approximation we make to get to our baseline is to store-allocate all values that appear, so that any non-machine state that contains a value instead contains an address to a value. The AAM approach stops at the previous optimization. However, the `fn` continuation stores a value, and this makes the space of continuations quadratic rather than linear in the size of the program, for a monovariant analysis like 0-CFA. Having the space of continuations grow linearly with the size of the program will drop the overall complexity to cubic (as expected).

To achieve this linearity for continuations, we allocate an address for the value position when we create the continuation. This address and the tail address are both determined by the label of the application point, so the space becomes linear and the overall complexity drops to cubic. This is a critical abstraction in languages with n-ary functions, since otherwise the continuation space grows super-exponentially. We extend the semantics to additionally allocate an address for the function value when creating the `fn` () continuation. The continuation has to contain this address to remember where to retreive values from in the store.

The new evaluation rules follow, where $t' = tick(\varsigma)$:

$$
\begin{aligned}
\mathtt{co}^t(\mathtt{ar}\ (e, \rho, k), v, \sigma) &\longmapsto \mathtt{ev}^{t'}(e, \rho, \sigma', \mathtt{fn}\ (a, k)) \\
\text{where } a &= alloc(\varsigma) \\
\sigma' &= \sigma \sqcup [a \mapsto \{v\}]
\end{aligned}
$$

Now instead of storing the evaluated function in the continuation frame itself, we indirect it through the store for further control on

complexity and precision:

$$\texttt{co}^t(\texttt{fn}\ (a,k),v,\sigma) \longmapsto \texttt{ap}_\ell^{t'}(u,v,\kappa,\sigma') \quad \text{if } \kappa \in \sigma(k), u \in \sigma(a)$$

Associated with this indirection, we now apply all functions stored in the address. This nondeterminism is demanded in order to continue with evaluation. More generally, this is because applied functions are in *strict position* (more about that later).

5. Implementation techniques

In this section, we discuss the optimizations for abstract interpreters that yield our ultimate performance gains. We have two broad categories of these optimizations: (1) pragmatic improvement, (2) transition elimination. The pragmatic improvements reduce overhead and trade space for time by utilizing:

1. timestamped stores;
2. store deltas; and
3. imperative, pre-allocated data structures.

The transition-elimination optimizations reduce the overall number of transitions made by the analyzer by performing:

4. frontier-based semantics;
5. lazy non-determinism; and
6. abstract compilation;

All pragmatic improvements are precision preserving (form complete abstractions), but the semantic changes are not in some cases, for reasons we will describe. We did not observe the precision differences in our evaluation.

We apply the frontier-based semantics combined with timestamped stores as our first step. The move to the imperative will be made last in order to show the effectiveness of these techniques in the purely functional realm.

5.1 Timestamped frontier

The semantics given for store widening in section 4.2, while simple, is wasteful. It also does not model what typical implementations do. It causes all states found so far to step each iteration, even if they are not revisited. This has negative performance *and* precision consequences (changes to the store can travel back in time in straight-line code). We instead use a frontier-based semantics that corresponds to the classic worklist algorithms for analysis. The difference is that the store is not modified in-place, but updated after all frontier states have been processed. This has implications for the analysis' precision and determinism. Specifically, higher precision, and it is deterministic even if set iteration is not.

The state space changes from a store and set of contexts to a set of seen abstract states (context plus store), S, a set of contexts to step (the frontier), F, and a store to step those contexts with, σ:

$$(S,F,\sigma) \widehat{\longmapsto} (S \cup S', F', \sigma')$$

We constantly see more states, so S is always growing. The frontier, which is what remains to be done, changes. Let's start with the result of stepping all the contexts in F paired with the current store (call it I for intermediate):

$$I = \{(c',\sigma') \mid wn(c,\sigma) \longmapsto wn(c',\sigma'), c \in F\}$$

The next store is the least upper bound of all the stores in I:

$$\sigma' = \bigsqcup \{\sigma \mid (\_,\sigma) \in I\}$$

The next frontier is exactly the states that we found from stepping the last frontier, but have not seen before. They must be *states*, so we pair the contexts with the next store:

$$F' = \{c \mid (c,\_) \in I, (c,\sigma') \notin S\}$$

Finally, we add what we know we had not yet seen to the seen set:

$$S' = \{(c,\sigma') \mid c \in F'\}$$

To inject a program e into this machine, we start off knowing we have seen the first state, and that we need to process the first state:

$$\begin{aligned} \mathit{inject}(e) &= (\{(c_0,\bot)\},\{c_0\},\bot) \\ \text{where } c_0 &= \texttt{ev}\ (e,\bot,\texttt{mt}) \end{aligned}$$

Notice that now S has several copies of the abstract store in it. As it is, this semantics is much less efficient (but still more precise) than the previously proposed semantics because membership checks have to compare entire stores. Checking equality is expensive because the stores within each are large, and every entry must be checked against every other. Hashes can sometimes rule out inequality relatively quickly, but the incidence of collisions and actual equality is costly.

And, there is a better way. Shivers' original work on k-CFA was susceptible to the same problem, and he suggested three complementary optimizations: (1) make the store global; (2) update the store imperatively; and (3) associate every change in the store with a version number – its timestamp. Then, put timestamps in states where previously there were stores. Given two states, the analysis can now compare their stores just by comparing their timestamps – a constant-time operation.

There are two subtle losses of precision in Shivers' original timestamp technique that we can fix.

1. In our semantics, the store does not change until the entire frontier has been explored. This avoids cross-branch pollution which would not otherwise happen, e.g., when one branch writes to address a and another branch reads from address a.
2. The common implementation strategy for timestamps destructively updates each state's timestamp. This loses *temporal* information about the contexts a state is visited in, and in what order. The semantics is different - we can't precisely answer questions about the traces of the reduction relation we defined. Our semantics has a drop-in replacement of timestamps for stores in the seen set (S), so we do not experience precision loss.

$$\Sigma \in \mathit{Store}^* \qquad S \subseteq \mathbb{N} \times \mathit{Context} \qquad F \subseteq \mathit{Context}$$

$$\begin{aligned} (S,F,\sigma,\Sigma,t) &\widehat{\longmapsto}^T (S \cup S', F', \sigma', \Sigma', t') \\ &\text{where } I = \{(c',\sigma^c) \mid wn(c,\sigma) \longmapsto wn(c',\sigma^c), c \in F\} \\ &\sigma' = \bigsqcup \{\sigma^c \mid (\_,\sigma^c) \in I\} \\ &(t',\Sigma') = \begin{cases} (t+1,\sigma'\Sigma') & \text{if } \sigma' \neq \sigma \\ (t,\Sigma) & \text{otherwise} \end{cases} \\ &F' = \{c \mid (c,\_) \in I, (c,t') \notin S\} \\ &S' = \{(c,t') \mid c \in F'\} \\ \mathit{inject}(e) &= (\{(c_0,0)\},\{c_0\},\bot,\bot{:}\epsilon,0) \\ &\text{where } c_0 = \texttt{ev}\ (e,\bot,\texttt{mt}) \end{aligned}$$

The observation Shivers made was that the store is increasing monotonically, so all stores throughout execution will be totally ordered (form a chain). This observation allows you to replace stores with pointers into this chain. We keep the stores around in Σ to acheive a complete abstraction. This corresponds to the temporal information about the execution's effect on the store.

Note also that F is only populated with states that have not been seen at the resulting store. This is what produces the more precise abstraction than the baseline widening.

The general fixed-point combinator we showed above can be specialized to this semantics, as well. In fact, $\widehat{\longmapsto}^T$ is a functional relation, so we can get the least fixed-point of it directly.

Lemma 1. *$\widehat{\longmapsto}$ maintains the invariant that all stores in S are totally ordered and σ is an upper bound of the stores in S.*

Lemma 2. *$\widehat{\longmapsto}^T$ maintains the invariant that Σ is in order with respect to $\sqsupset$ and $\sigma = hd(\Sigma)$.*

Theorem 1. *Timestamps are a complete abstraction.*

The proof follows from the Galois connection that, in one direction, sorts all the stores in S to form Σ, and translates stores in S to their distance from the end of Σ. In the other direction, timestamps are replaced by the stores they point to.

5.2 Locally log-based store deltas

The above technique requires joining entire (large) stores together. Additionally, there is still a comparison of stores, which we established is expensive. Not every step will modify all addresses of the store, so joining entire stores is wasteful in terms of memory and time. We can instead log store changes and replay the change log on the full store after all steps have completed, noting when there is an actual change. This uses far fewer join and comparison operations, leading to less overhead, and is precision-preserving.

We represent change logs as $\xi \in \mathit{Store}\Delta = (\mathit{Addr} \times \mathcal{P}(\mathit{Storeable}))^*$. Each $\sigma \sqcup [a \mapsto vs]$ becomes a log addition $(a, vs){:}\xi$, where ξ begins empty (ϵ) for each step. Applying the changes to the full store is straightforward:

$$\mathit{replay} : (\mathit{Store}\Delta \times \mathit{Store}) \to (\mathit{Store} \times \mathit{Boolean})$$

$$\begin{aligned}\mathit{replay}([(a_i, vs_i), \ldots], \sigma) &= (\sigma', vs_i \stackrel{?}{=} \sigma'(a_i) \vee \ldots)\\ &\text{where } \sigma' = \sigma \sqcup [a_i \mapsto vs_i] \sqcup \ldots\end{aligned}$$

We change the semantics slightly to add to the change log rather than produce an entire modified store. The transition relation is identical except for the addition of this change log. We maintain the invariant that lookups will never rely on the change log, so we can use the originally supplied store unmodified.

A taste of the changes to the reduction relation is as follows:

$$\longmapsto_{\sigma\xi} \subseteq (\mathit{Context} \times \mathit{Store}) \times (\mathit{Context} \times \mathit{Store}\Delta)$$

$$\begin{aligned}(\mathtt{ap}^t_\ell(\mathtt{clos}\ (x, e, \rho), v, \kappa), \sigma) &\longmapsto_{\sigma\xi} (\mathtt{ev}^{t'}(e, \rho', \kappa), (a, \{v\}){:}\epsilon)\\ &\text{where } a = \mathit{alloc}(\varsigma)\\ &\quad\ \rho' = \rho[x \mapsto a]\end{aligned}$$

We lift $\longmapsto_{\sigma\xi}$ to accommodate for the asymmetry in the input and output, and change the frontier-based semantics in the following way:

$$\begin{aligned}(S, F, \sigma, \Sigma, t) &\widehat{\longmapsto}_{\sigma\xi} (S \cup S', F', \sigma', \Sigma', t')\\ &\text{where } I = \{(c', \xi) \mid (c, \sigma) \longmapsto_{\sigma\xi} (c', \xi)\}\\ &(\sigma', \Delta?) = \mathit{replay}(\mathit{appendall}(\{\xi \mid (\_, \xi) \in I\}), \sigma)\\ &(t', \Sigma') = \begin{cases}(t+1, \sigma\Sigma) & \text{if } \Delta?\\ (t, \Sigma) & \text{otherwise}\end{cases}\\ &F' = \{c \mid (c, \_) \in I, (c, t') \notin S\}\\ &S' = \{(c, t') \mid c \in F'\}\\ \mathit{appendall}(\varnothing) &= \epsilon\\ \mathit{appendall}(\{\xi\} \cup \Xi) &= \mathit{append}(\xi, \mathit{appendall}(\Xi))\end{aligned}$$

Here *appendall* combines change logs across all non-deterministic steps for a state to later be replayed. The order the combination happens in doesn't matter, because join is associative and commutative.

Lemma 3. *$(c, \sigma) \longmapsto_{\sigma\xi} (c', \xi)$ iff $wn(c, \sigma) \longmapsto wn(c', \mathit{replay}(\xi, \sigma))$*

By cases on $\longmapsto_{\sigma\xi}$ and $\longmapsto$.

Lemma 4 ($\Delta?$ means change)**.** *Let $\mathit{replay}(\xi, \sigma) = (\sigma', \Delta?)$. $\sigma' \neq \sigma$ iff $\Delta?$.*

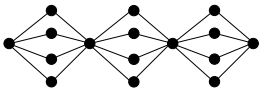
By induction on ξ.

Theorem 2. *$\widehat{\longmapsto}_{\sigma\xi}$ is a complete abstraction of $\widehat{\longmapsto}^T$.*

Follows from previous lemma and that join is associative and commutative.

5.3 Lazy non-determinism

Tracing the execution of the analysis reveals an immediate shortcoming: there is a high degree of branching and merging in the exploration. Surveying this branching has no benefit for precision. For example, in a function application, `(f x y)`, where `f`, `x` and `y` each have several values each argument evaluation induces n-way branching, only to be ultimately joined back together in their respective application positions. Transition patterns of this shape litter the state-graph:



To avoid the spurious forking and joining, we *delay* the non-determinism until and unless it is needed in *strict contexts* (such as the guard for an `if`, a called procedure, or a numerical primitive application). Doing so collapses these forks and joins into a linear sequence of states:

This shift does not change the concrete semantics of the language to be lazy. Rather, it abstracts over transitions that the original non-deterministic semantics steps through. We say the abstraction is *lazy* because it delays splitting on the values in an address until they are *needed* in the semantics. It does not change the execution order that leads to the values that are stored in the address.

We introduce a new kind of value, `addr` (a), that represents a delayed non-deterministic choice of a value from $\sigma(a)$. The following rules highlight the changes to the semantics:

$$\begin{aligned}\mathit{force} &: \mathit{Store} \times \mathit{Value} \to \mathcal{P}(\mathit{Value})\\ \mathit{force}(\sigma, \mathtt{addr}\ (a)) &= \sigma(a)\\ \mathit{force}(\sigma, v) &= \{v\}\\ \mathtt{ev}\ (\mathtt{var}\ (x), \rho, \kappa, \sigma) &\longmapsto_{\mathcal{L}} \mathtt{co}\ (\kappa, \mathtt{addr}\ (\rho(x)), \sigma)\\ \mathtt{co}\ (\mathtt{ar}^t_\ell(e, \rho, k), v, \sigma) &\longmapsto_{\mathcal{L}} \mathtt{ev}^{t'}(e, \rho, \sigma', \mathtt{fn}^t_\ell(a_f, k))\\ &\text{where } a_f = \mathit{alloc}(\varsigma)\\ &\quad\ \sigma' = \sigma \sqcup [a \mapsto \mathit{force}(\sigma, v)]\\ \mathtt{co}\ (\mathtt{fi}^t(e_0, e_1, \rho, k), v, \sigma) &\longmapsto_{\mathcal{L}} \mathtt{ev}^{t'}(e, \rho, \sigma, \kappa)\\ &\text{if } \kappa \in \sigma(k), \mathtt{tt} \in \mathit{force}(\sigma, v)\end{aligned}$$

Since `if` guards are in strict position, we must force the value to determine which branch to to take. The middle rule uses *force* only to combine with values in the store - it does not introduce needless non-determinism.

We have two choices for how to implement lazy non-determinism.

Option 1: Lose precision; simplify implementation This semantics introduces a subtle precision difference over the baseline. Consider a configuration where a reference to a variable and a binding of a variable will happen in one step, since store widening leads to stepping several states in one big "step." With laziness, the reference will mean the original binding(s) of the variable or the new binding, because the actual store lookup is delayed one step (i.e. laziness is administrative). Without laziness, the reference will fan out to all the bindings of the variable before the new binding happens and thus theoretically has an observable precision difference.

Option 2: Regain precision; complicate implementation The administrative nature of laziness means that we could remove the loss in precision by storing the result of the lookup in a value representing a delayed nondeterministic choice. This has problems with the next optimization, leading to unsoundness. Regaining soundness would involve artificial complications to the semantics.

Our choice: option 1 The configurations that lead to precision loss happen too rarely to warrant the significant increase in time and memory needed for this eager non-determinism. Without a widened store, lazy non-determinism is not a complete abstraction with either approach, but the precision gains are not immediately apparent due to fake rebinding (a second reference to the same variable chooses a different value from the first) and other problems. The performance and memory gains are apparent, so we favor the simplicity over invisible gains.

Theorem 3 (Soundness)**.** *If $\varsigma \longmapsto \varsigma'$ and $\varsigma \sqsubseteq \hat{\varsigma}$ then there exists a $\hat{\varsigma}'$ such that $\hat{\varsigma} \longmapsto_{\mathcal{L}} \hat{\varsigma}'$ and $\varsigma' \sqsubseteq \hat{\varsigma}'$*

Here $\sqsubseteq$ is straightforward - the left-hand side store must be contained in the right-hand-side store, and if values occur in the states, forcing the left-hand-side value must be a subset of forcing the corresponding right-hand-side value. The proof is by cases on $\varsigma \longmapsto \varsigma'$.

5.4 Abstract compilation

The prior optimization saved time by doing the same amount of reasoning as before but in fewer transitions. We can exploit the same idea—same reasoning, fewer transitions—with abstract compilation. Abstract compilation transforms complex expressions whose *abstract* evaluation is deterministic into "abstract bytecodes." The abstract interpreter then does in one transition what previously took many. In short, abstract compilation eliminates unnecessary allocation, deallocation and branching. The technique is precision preserving without store widening. We discuss the precision differences with store widening at the end of the section.

The following example illustrates the essence of abstract compilation effect:

$$e := \texttt{app}\,(\texttt{app}\,(\texttt{app}\,(x, e_1), e_2), e_3)$$

makes the following transitions:

$$\texttt{ev}^t(\texttt{app}\,(\texttt{app}\,(\texttt{app}\,(x, e_1), e_2), e_3), \rho, \sigma_0, \kappa) \quad (1)$$
$$\longmapsto \texttt{ev}^{t'}(\texttt{app}\,(\texttt{app}\,(x, e_1), e_2), \rho, \sigma_1, \texttt{ar}\,(e_3, \rho, a_1)) \quad (2)$$
$$\longmapsto \texttt{ev}^{t''}(\texttt{app}\,(x, e_1), \rho, \sigma_2, \texttt{ar}\,(e_2, \rho, a_2)) \quad (3)$$
$$\longmapsto \texttt{ev}^{t'''}(x, \rho, \sigma_3, \texttt{ar}\,(e_1, \rho, a_3)) \quad (4)$$
$$\longmapsto \texttt{co}\,(\texttt{ar}\,(e_1, \rho, a_3), \texttt{addr}\,(\rho(x)), \sigma_4) \quad (5)$$

where $\sigma_4 = \sigma_0 \sqcup [a_1 \mapsto \{\kappa\}, a_2 \mapsto \{\texttt{ar}\,([\![e_3]\!], \rho, a_1)\}, a_3 \mapsto \{\texttt{ar}\,([\![e_2]\!], \rho, a_2)\}]$.

Whereas abstract compilation gives us in one step:

$$[\![e]\!]^t(\sigma_0)(\rho, \epsilon, \kappa) = \texttt{co}\,(\texttt{ar}\,([\![e_1]\!], \rho, a_3), \texttt{addr}\,(\rho(x))), \xi_4$$

$$\begin{aligned}
[\![\text{-}]\!] &: \mathit{Expr} \to \mathit{Store} \\
&\to \mathit{Env} \times \mathit{Store}\Delta \times \mathit{Kont} \times \mathit{Time} \\
&\to \mathit{State} \\
t' &= \mathit{tick}(\ell, \rho, \sigma, t) \\
[\![\texttt{var}\,(x)]\!]_\sigma &= \lambda^t(\rho, \xi, \kappa).\texttt{co}\,(\kappa, \texttt{addr}\,(\rho(x))), \xi \\
[\![\texttt{lit}\,(l)]\!]_\sigma &= \lambda^t(\rho, \xi, \kappa).\texttt{co}\,(\kappa, l), \xi \\
[\![\texttt{lam}\,(x, e)]\!]_\sigma &= \lambda^t(\rho, \xi, \kappa).\texttt{co}\,(\kappa, \texttt{clos}\,(x, [\![e]\!], \rho)), \xi \\
[\![\texttt{app}^\ell(e_0, e_1)]\!]_\sigma &= \lambda^t(\rho, \xi, \kappa).[\![e_0]\!]^{t'}(\rho, \xi', \texttt{ar}^t_\ell([\![e_1]\!], \rho, k)) \\
&\text{where} \quad k = \mathit{allockont}^t_\ell(\sigma, \kappa) \\
&\qquad \xi' = (k, \{\kappa\}):\xi \\
[\![\texttt{if}^\ell(e_0, e_1, e_2)]\!]_\sigma &= \lambda^t(\rho, \xi, \kappa).[\![e_0]\!]^{t'}(\rho, \xi', \texttt{fi}^t([\![e_1]\!], [\![e_2]\!], \rho, a)) \\
&\text{where } k = \mathit{allockont}^t_\ell(\sigma, \kappa) \\
&\qquad \xi' = (k, \{\kappa\}):\xi
\end{aligned}$$

Figure 6. Abstract compilation

$$\begin{aligned}
\mathit{traces}(e) &= \{\mathit{inject}([\![e]\!]^{t_0}_\perp(\perp, \epsilon, \texttt{mt})) \longmapsto\!\!\!\!\!\rightarrow \varsigma\} \text{ where} \\
\mathit{inject}(c, \xi) &= \mathit{wn}(c, \mathit{replay}(\xi, \perp)) \\
\mathit{wn}(c, \sigma) \longmapsto \mathit{wn}(c', \sigma') &\iff c\,[\![\longmapsto]\!]_\sigma\,c', \xi \\
&\xi \text{ is such that } \mathit{replay}(\xi, \sigma) = \sigma' \\
\\
\texttt{co}\,(\texttt{mt}, v)\,[\![\longmapsto]\!]_\sigma\,&\texttt{ans}\,(v'), \epsilon \text{ if } v' \in \mathit{force}(\sigma, v) \\
\texttt{co}\,(\texttt{ar}^t_\ell(k, \rho, k), v)\,[\![\longmapsto]\!]_\sigma\,&k^t(\sigma)(\rho, \xi, \texttt{fn}^t_\ell(a_f, k)) \\
&\text{where } a_f = \mathit{alloc}(\varsigma) \\
&\qquad \xi = (a_f, \mathit{force}(\sigma, v)):\epsilon \\
\texttt{co}\,(\texttt{fn}^t_\ell(a_f, k), v)\,[\![\longmapsto]\!]_\sigma\,&\texttt{ap}^t_\ell(v, v\kappa), \epsilon \\
&\text{if } v \in \sigma(a_f), \kappa \in \sigma(k) \\
\texttt{co}\,(\texttt{fi}^t(k_0, k_1, \rho, a), \texttt{tt})\,[\![\longmapsto]\!]_\sigma\,&k^t_0(\sigma)(\rho, \epsilon, \kappa) \text{ if } \kappa \in \sigma(a) \\
\texttt{co}\,(\texttt{fi}^t(k_0, k_1, \rho, a), \texttt{ff})\,[\![\longmapsto]\!]_\sigma\,&k^t_1(\sigma)(\rho, \epsilon, \kappa) \text{ if } \kappa \in \sigma(a) \\
\\
\texttt{ap}^t_\ell(\texttt{clos}\,(x, k, \rho), v, \kappa)\,[\![\longmapsto]\!]_\sigma\,&k^{t'}(\sigma)(\rho', \xi, \kappa) \\
&\text{where } \rho' = \rho[x \mapsto a] \\
&\qquad \xi = (a, \mathit{force}(\sigma, v)):\epsilon \\
\texttt{ap}\,(o, v, \kappa)\,[\![\longmapsto]\!]_\sigma\,&\texttt{co}\,(\kappa, v'), \epsilon \\
&\text{where } u \in \mathit{force}(\sigma, v), v' \in \Delta(o, u)
\end{aligned}$$

Figure 7. Abstract abstract machine for compiled ISWIM

where

$$\xi_4 = [(a_1, \{\kappa\}), (a_2, \{\texttt{ar}\,([\![e_3]\!], \rho, a_1)\}), (a_3, \{\texttt{ar}\,([\![e_2]\!], \rho, a_2)\})].$$

The compilation step converts expressions into functions that expect the other components of the `ev` state. Its definition in figure 6 shows close similarity to the rules for interpreting `ev` states. The next step is to change reduction rules that create `ev` states to instead call these functions. Figure 7 shows the modified reduction relation. The only change from the previous semantics is that `ev` () state construction is replaced by calling the compiled expression. For notational coherence, we write $\lambda^t(\mathit{args}\ldots)$ for $\lambda(\mathit{args}\ldots, t)$ and $k^t(\mathit{args}\ldots)$ for $k(\mathit{args}\ldots, t)$.

Correctness The correctness of abstract compilation seems obvious, but it has never before been rigorously proved. What constitutes correctness in the case of dropped states, anyway? Applying

an abstract bytecode's function does many "steps" in one go, at the end of which, the two semantics line up again (modulo representation of expressions). This constitutes the use of a notion of stuttering. We provide a formal analysis of abstract compilation *without* store widening with a proof of a stuttering bisimulation [3] between this semantics and lazy non-determinism without widening to show precision preservation.

The number of transitions that can occur in succession from an abstract bytecode is roughly bounded by the amount of expression nesting in the program. The partial order that defines subexpressions smaller than the expression in which they occur is well-founded, so we use this observation to drive our use of a well-founded equivalence bisimulation (WEB) [18]. WEBs are equivalent to the notion of a stuttering bisimulation, but are more amenable to mechanisation (and thus rigorous proof). They also only require reasoning over one step of the reduction relation, which makes the proof concise.

We define a refinement from non-compiled to compiled states by "committing" all the actions of an `ev` () state (defined similarly to $[\![\_]\!]$, but immediately applies the functions), and subsequently changing all expressions with their compiled variants. Since WEBs are for single transition systems, a WEB refinement is over the disjoint union of our two semantics, and the equivalence relation we use is just that a state is related to its refined state (and itself). Call this relation B.

Before we prove this setup is indeed a WEB, we need one lemma that applying an abstract bytecode's function is equal to refining the corresponding `ev` () state:

Lemma 5 (Compile/commit)**.** *Let* $c, \xi' = [\![e]\!]^t_{r(\sigma)}(\rho, \xi, r(\kappa))$. *Let* $wn(c', \sigma') = r(\texttt{ev}^t(e, \rho, \sigma, \kappa))$. $wn(c, replay(\xi', \sigma)) = wn(c', replay(\xi, \sigma'))$.

The proof is by induction on e.

Theorem 4 (Precision preservation)**.** *B is a WEB on* $\longmapsto_{\mathcal{L}} \uplus \longmapsto$

The proof follows by cases on $\longmapsto_{\mathcal{L}} \uplus \longmapsto$ with the WEB *witness* being the well-order on expressions (with a $\bot$ element), and the following $erankt$, $erankl$ functions:

$$\begin{aligned} erankt(\texttt{ev}^t(e, \rho, \sigma, \kappa)) &= e \\ erankt(\varsigma) &= \bot \quad \text{otherwise} \\ erankl(s, s') &= 0 \end{aligned}$$

All cases are either simple steps or appeals to the well-order on $erankt$'s range. The other rank function, $erankl$ is unnecessary, so we just make it the constant 0 function. The $[\![\longmapsto]\!]$ cases are trivial.

Wide store and abstract compilation The fact that many changes happen in one step also makes the comparison with the widened lazy semantics more subtle. It is possible for different stores to occur between the different semantics because abstract compilation can change the order in which the store is changed. Consider the case where before we might call the same function from two different places in 3 and 2 steps respectively, compilation can make that 2 and 1, reversing the order that we bind values in the store, leading to a mismatch from the previous semantics. This kind of cross-step store poisoning is inevitable when a store is shared between several traces. The binding could have just as easily been observed the other way around in the non-compiled semantics. Call $[\![\widehat{\longmapsto}]\!]$ the result of the widening operator from the previous section on $[\![\longmapsto]\!]$.

5.5 Imperative, pre-allocated data structures

Thus far, we have made our optimizations in a purely functional manner. For the final push for performance, we need to dip into the imperative. In this section, we show an alternative representation of the store and seen set that are more space-efficient and are amenable to destructive updates.

The following transfer function has several components that can be destructively updated, and intermediate sets can be elided by adding to global sets. In fact, the log of store deltas can be removed as well, by updating the store in-place, and on lookup, using the first value timestamped $\leq$ the current timestamp. We start with the purely functional view.

5.5.1 Pure setup for imperative implementation

The store maps to a stack of timestamped sets of abstract values.

$$\begin{aligned} \sigma \in Store &= Addr \to ValStack \\ V \in ValStack &= (\mathbb{N} \times \wp(Storeable))^* \end{aligned}$$

Here we formally define what lookup and update mean at a given timestamp.

$$\begin{aligned} lookup(V, t) &= \begin{cases} vs & \text{if } V = (t', vs){:}V', t' \leq t \\ vs' & \text{if } V = (t', vs){:}(t'', vs'){:}V', t' > t \end{cases} \\ \sigma \sqcup_t [a \mapsto vs] &= \sigma[a \mapsto V], \Delta\texttt{?} \\ (V, \Delta\texttt{?}) &= \sigma(a) \sqcup_t vs \\ \epsilon \sqcup_t vs &= (t, vs) \\ (t', vs){:}V \sqcup_t vs' &= (t', vs \sqcup vs'){:}V, \texttt{tt} \text{ if } t' > t \\ V \sqcup_t vs &= (t+1, vs^*){:}V, \texttt{tt} \text{ if } vs' \neq vs^* \\ vs' &= lookup(\sigma(a), t) \\ vs^* &= vs \sqcup vs' \\ V \sqcup_t vs &= V, \texttt{ff} \text{ otherwise} \end{aligned}$$

For the purposes of space, we reuse the $[\![\longmapsto]\!]$ semantics, although the $replay$ of the produced ξ objects should be in-place, and the $lookup$ function should be using this single-threaded store. Because the store has all the temporal information baked into it, we rephrase the core semantics in terms of a transfer function. The least fixed-point of this function gives a more compact representation of the reduction relation of the previous section.

$$\begin{aligned} System &= (\widehat{State} \to \mathbb{N}^*) \times \wp(\widehat{State}) \times Store \times \mathbb{N} \\ \mathcal{F} &: System \to System \\ \mathcal{F}(S, F, \sigma, t) &= (S', F', \sigma', t') \\ \text{where } I &= \{(c', \xi) \mid c \in F, c\,[\![\longmapsto]\!]_{\sigma^*}\, c', \xi\} \\ \sigma^* &= \lambda a.lookup(\sigma(a), t) \\ (\sigma', \Delta\texttt{?}) &= replay(appendall(\{\xi \mid (\_, \xi) \in I\}), \sigma) \\ t' &= \begin{cases} t+1 & \text{if } \Delta\texttt{?} \\ t & \text{otherwise} \end{cases} \\ F' &= \{c \mid (c, \_) \in I, \Delta\texttt{?} \vee S(c) \neq t{:}\_\} \\ S' &= \lambda c. \begin{cases} t{:}S(c) & \text{if } c \in F' \\ S(c) & \text{otherwise} \end{cases} \end{aligned}$$

We prove semantic equivalence with the previous semantics with a lock-step bisimulation with the stack of stores abstraction, which follow from equational reasoning from the following lemmas:

Lemma 6. *Stores of value stacks completely abstract stacks of stores.*

This depends on some wellformedness conditions about the order of the stacks. The store of value stacks can be translated to a stack of stores by taking successive "snapshots" of the store

at different timestamps from the max timestamp it holds down to 0. Vice versa, we replay the changes across adjacent stores in the stack.

We apply a similar construction to the different representation of seen states in order to get the final result:

Theorem 5. *$\mathcal{F}$ is a complete abstraction of $[\![\widehat{\longmapsto}]\!]$.*

5.5.2 Pure to imperative

The intermediate data structures of the above transfer function can all be streamlined into globals that are destructively updated. In particular, there are 5 globals:

1. S: the *seen* set, though made a map for faster membership tests and updates.
2. F: the *frontier* set, which must be persisent or copied for the iteration through the set to be correct.
3. σ: the store, which represents all stores that occur in the machine semantics.
4. t: the timestamp, or length of the store chain - all stores that occur in the semantics are totally ordered due to single-threading the store.
5. $\Delta?$: whether a store change stepped all states in F.

The reduction relation would then instead of building store deltas, update the global store. We would also not view it as a general relation, but a function that adds all next states to F if they have not already been seen. At the end of iterating through F, S is updated with the new states at the next timestamp.

5.5.3 Pre-allocating the store

Internally, the algorithm at this stage uses hash tables to model the store to allow arbitrary address representations. But, such a dynamic structure isn't necessary when we know the structure of the store in advance: we know all possible entries, and we know its maximum size. In a monovariant allocation strategy, the domain of the store is exactly the set of expressions in the program. If we label each expression with a unique natural, the analysis can index directly into the store without a hash or a collision. Even for polyvariant analyses, it is possible to compute the maximum number of addresses and similarly pre-allocate either the spine of the store or (if memory is no concern) the entire store.

6. Evaluation

We have implemented, optimized, and evaluated an analysis framework supporting higher-order functions, state, first-class control, compound data, and a large number of primitive kinds of data and operations such as floating point, complex, and exact rational arithmetic. The analysis is evaluated against a suite of Scheme benchmarks drawn from the literature. For each benchmark, we collect analysis times, peak memory usage, and the rate of states-per-second explored by the analysis for each of the optimizations discussed in section 5, cumulatively applied. The analysis is stopped after consuming 30 minutes of time or 1 gigabyte of space. When presenting *relative* numbers, we use the timeout limits as a lower bound on the actual time required, thus giving a conservative estimate of improvements.

All benchmarks are calculated as an average of 5 runs, done in parallel, on an 12-core, 64-bit Intel Xeon machine running at 2.40GHz with 12Gb of memory.

Many benchmarks cause the baseline analyzer to take longer than 30 minutes or to consume more 1 gigabyte of memory, at which point the analysis is stopped. This is the case for the largest benchmark program, **nucleic**, which is 3,500 lines of code and takes under a minute in the most optimized analyzer. For those benchmarks that did complete on the baseline, the optimized analyzer outperformed the baseline by a factor of two to three orders of magnitude.

Program	LOC	Time (s)		Space (MB)		Speed (state/s)	
nucleic	3492	m	68.2	m	219	58	8K
matrix	747	t	3.7	466	105	25	79K
nbody	1435	t	18.7	516	143	31	65K
earley	667	1.1K	0.4	436	105	15	93K
maze	681	t	2.6	530	105	24	119K
church	42	42.2	0.1	106	105	171	57K
lattice	214	346.4	0.2	282	105	31	103K
boyer	642	m	13.3	m	113	50	39K
mbrotZ	69	356.0	0.1	297	105	134	61K

Figure 8. Overview performance comparison between baseline and optimized analyzer (entries of t mean timeout, and m mean out of memory).

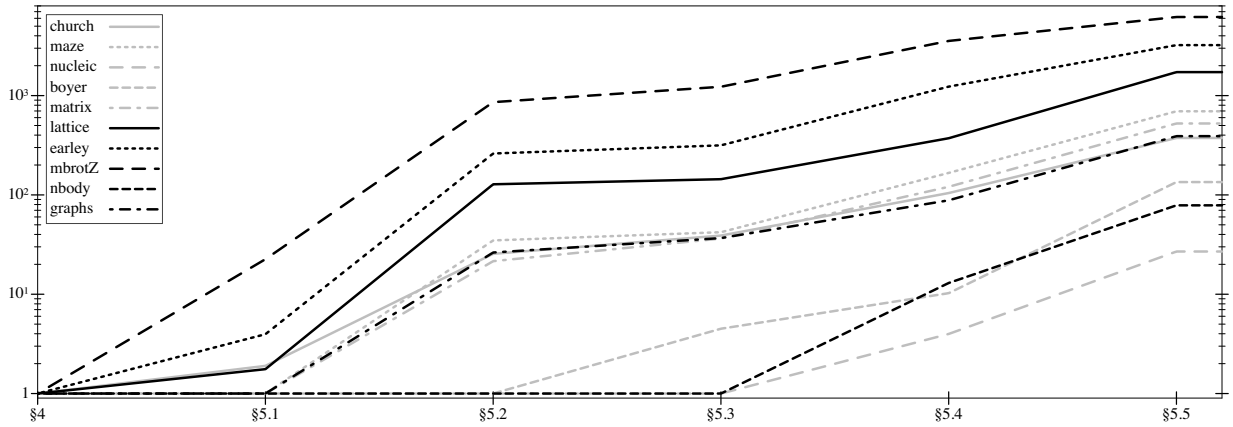
We use the following set of benchmarks:

1. **nucleic**: a floating-point intensive application taken from molecular biology that has been used widely in benchmarking functional language implementations [13] and analyses (e.g. [14, 36]). It is a constraint satisfaction algorithm used to determine the three-dimensional structure of nucleic acids.
2. **matrix** tests whether a matrix is maximal among all matrices of the same dimension obtainable by simple reordering of rows and columns and negation of any subset of rows and columns. It is written in continuation-passing style (used in [14, 36]).
3. **nbody**: implementation [37] of the Greengard multipole algorithm for computing gravitational forces on point masses distributed uniformly in a cube (used in [14, 36]).
4. **earley**: Earley's parsing algorithm, applied to a 15-symbol input according to a simple ambiguous grammar. A real program, applied to small data whose exponential behavior leads to a peak heap size of half a gigabyte or more during concrete execution.
5. **maze**: generates a random maze using Scheme's `call/cc` operation and finds a path solving the maze (used in [14, 36]).
6. **church**: tests distributivity of multiplication over addition for Church numerals (introduced by [34]).
7. **lattice**: enumerates the order-preserving maps between two finite lattices (used in [14, 36]).
8. **boyer**: a term-rewriting theorem prover (used in [14, 36]).
9. **mbrotZ**: generates Mandelbrot fractal using complex numbers.
10. **graphs**: counts the number of directed graphs with a distinguished root and k vertices, each having out-degree at most 2. It is written in a continuation-passing style and makes extensive use of higher-order procedures—it creates closures almost as often as it performs non-tail procedure calls (used by [14, 36]).

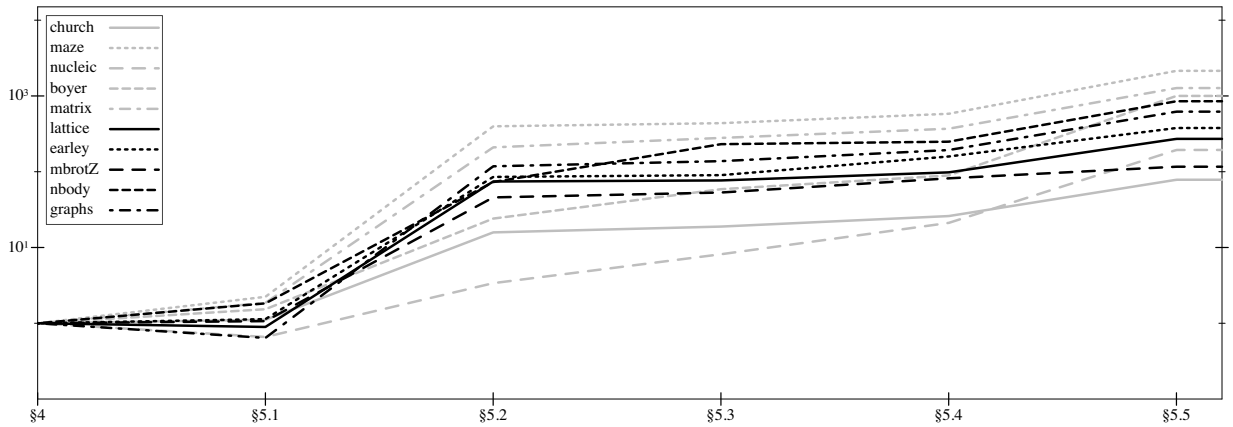
Figure 8 gives an overview of the benchmark results in terms of absolute time, space, and speed between the baseline and most optimized analyzer. Figure 9 plots the factors of improvement over the baseline for each optimization step.

We evaluated the precision of these techniques with a singleton variable analysis to find opportunities to inline constants and closed functions. We found no change in the results across all implementations, including Shivers' timestamp approximation.

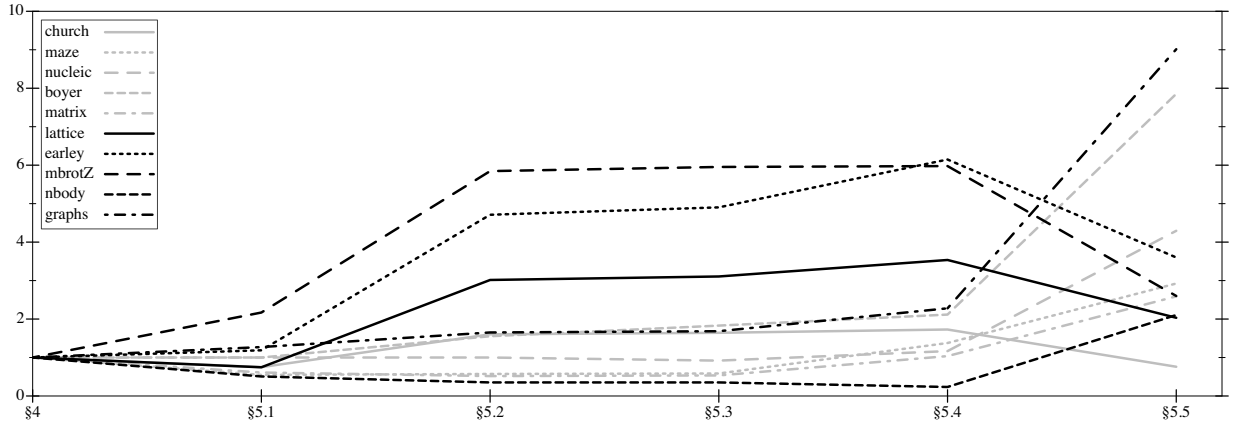
Source code of the implementation and benchmark suite is at:



(a) Total analysis time



(b) Rate of state transitions



(c) Peak memory usage

Figure 9. Factors of improvement over baseline for each step of optimization (bigger is better).

`https://github.com/dvanhorn/oaam`

Comparison with other flow analysis implementations The analysis considered here computes results similar to Earl, et al.'s 0-CFA implementation [10], which times out on the Vardoulakis and Shivers benchmark because it does not widen the store as described for our baseline evaluator. So even though it offers a fair point of comparison, a more thorough evaluation is probably uninformative as the other benchmarks are likely to timeout as well (and it would require significant effort to extend their implementation with the features needed to analyze our benchmark suite). That implementation is evaluated against much smaller benchmarks: the largest program is 30 lines.

Vardoulakis and Shivers evaluate their CFA2 analyzer [34] against a variant of 0-CFA defined in their framework and the example we draw on is the largest benchmark Vardoulakis and Shivers consider. More work would be required to scale the analyzer to the set of features required by our benchmarks.

The only analyzers we were able to find that proved capable of analyzing the full suite of benchmarks considered here were the Soft Typing system of Wright and Cartwright [35] and, in many ways its successor, the Polymorphic splitting system of Wright and Jagannathan [36].[2] Unfortunately, these analyses compute an inherently different and incomparable form of analysis. Consequently, we have omitted a complete comparison with these implementations. The AAM approach provides more precision in terms of temporal-ordering of program states, which comes at a cost that can be avoided in constraint-based approaches. Consequently implementation techniques cannot be "ported" between these two approaches. However, our optimized implementation is within an order of magnitude of the performance of Wright and Jaganathan's analyzer. Although we would like to improve this to be more competitive, the optimized AAM approach still has many strengths to recommend it in terms of precision, ease of implementation and verification, and rapid design. We can get closer to their performance by relying on the representation of addresses and the behavior of $alloc$ to pre-allocate most data structures and split the abstract store out into parts that are more quickly accessed and updated. Our semantic optimizations can still be applied to an analysis that does abstract garbage collection [23], whereas the polymorphic splitting implementation is tied strongly to a single-threaded store.

7. Related work

Abstracting Abstract Machines This work clearly closely follows Van Horn and Might's original papers on abstracting abstract machines [31, 33], which in turn is one piece of the large body of research on flow analysis for higher-order languages (see Midtgaard [20] for a thorough survey). The AAM approach sits at the confluence of two major lines of research: (1) the study of abstract machines [17] and their systematic construction [28], and (2) the theory of abstract interpretation [4, 5].

Frameworks for flow analysis of higher-order programs Besides the original AAM work, the analysis most similar to that presented in section 3 is the infinitary control-flow analysis of Nielson and Nielson [26] and the unified treatment of flow analysis by Jagannathan and Weeks [15]. Both are parameterized in such a way that in the limit, the analysis is equivalent to an interpreter for the language, just as is the case here. What is different is that both give a constraint-based formulation of the abstract semantics rather than a finite machine model.

[2] This is not a coincidence; these papers set a high standard for evaluation, which we consciously aimed to approach.

Abstract compilation Boucher and Feeley [1] introduced the idea of abstract compilation, which used closure generation [11] to improve the performance of control flow analysis. We have adapted the closure generation technique from compositional evaluators to abstract machines and applied it to similar effect.

Constraint-based program analysis for higher-order languages Constraint-based program analyses (e.g. [19, 26, 29, 36]) typically compute sets of abstract values for each program point. These values approximate values arising at run-time for each program point. Value sets are computed as the least solution to a set of (inclusion or equality) constraints. The constraints must be designed and proved as a sound approximation of the semantics. Efficient implementations of these kinds of analyses often take the form of worklist-based graph algorithms for constraint solving, and are thus quite different from the interpreter implementation. The approach thus requires effort in constraint system design and implementation, and the resulting system require verification effort to prove the constraint system is sound and that the implementation is correct.

This effort increases substantially as the complexity of the analyzed language increases. Both the work of maintaining the concrete semantics and constraint system (and the relations between them) must be scaled simultaneously. However, constraint systems, which have been extensively studied in their own right, enjoy efficient implementation techniques and can be expressed in declarative logic languages that are heavily optimized [2]. Consequently, constraint-based analyses can be computed quickly. For example, Jagannathan and Wright's polymorphic splitting implementation [36] analyses the Vardoulakis and Shivers benchmark about 5.5 times faster than the fastest implementation considered here. These analyses compute very different things, so the performance comparison is not apples-to-apples.

The AAM approach, and the state transition graphs it generates, encodes temporal properties not found in classical constraint-based analyses for higher-order programs. Such analyses (ultimately) compute judgments on program terms and contexts, e.g., at expression e, variable x may have value v. The judgments do not relate the order in which expressions and context may be evaluated in a program, e.g., it has nothing to say with regard to question like, "Do we always evaluate e_1 before e_2?" or "Is it always the case that a file handle is opened, read and then closed in that order?" The state transition graphs can answer these kinds of queries, but this does not come for free: respecting temporal order imposes an order in which states and terms may be evaluated *during* the analysis.

8. Conclusion

Abstract machines are not only a good model for rapid analysis development, they can be systematically developed into efficient algorithms that can be proved correct. We view the primary contribution of this work as a systematic path that eases the design, verification, and implementation of analyses using the abstracting abstract machine approach to within a factor of performant constraint-based analyses.

Acknowledgments We thank Suresh Jagannathan for providing source code to the polymorphic splitting analyzer [36] and Ilya Sergey for the introspective pushdown analyzer [10].

References

[1] Dominique Boucher and Marc Feeley. Abstract compilation: A new implementation paradigm for static analysis. In Tibor Gyimóthy, editor, *Compiler Construction: 6th International Conference, CC'96 Linköping, Sweden, April 2426, 1996 Proceedings*, pages 192–207, 1996.

[2] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09: Proceedings of the 24th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2009.

[3] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite kripke structures in propositional temporal logic. *Theor. Comput. Sci.*, 59(1-2):115–131, 1988.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[5] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282. ACM, 1979.

[6] Olivier Danvy. *An Analytical Approach to Program as Data Objects*. DSc thesis, Department of Computer Science, Aarhus University, 2006.

[7] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, 2004.

[8] E. D'Osualdo, J. Kochems, and C.-H. L. Ong. Soter: An automatic safety verifier for Erlang. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, AGERE! '12, pages 137–140. ACM, 2012.

[9] E. D'Osualdo, J. Kochems, and C.-H. L. Ong. Automatic verification of Erlang-style concurrency. In *Proceedings of the 20th Static Analysis Symposium*, 2013.

[10] Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 177–188. ACM, 2012.

[11] Marc Feeley and Guy Lapalme. Using closures for code generation. *Comput. Lang.*, 12(1):47–66, 1987.

[12] Matthias Felleisen, Robert B. Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[13] Pieter H. Hartel, Marc Feeley, Martin Alt, Lennart Augustsson, Peter Baumann, Marcel Beemster, Emmanuel Chailloux, Christine H. Flood, Wolfgang Grieskamp, John H. G. Van Groningen, Kevin Hammond, Bogumil Hausman, Melody Y. Ivory, Richard E. Jones, Jasper Kamperman, Peter Lee, Xavier Leroy, Rafael D. Lins, Sandra Loosemore, Niklas Röjemo, Manuel Serrano, Jean P. Talpin, Jon Thackray, Stephen Thomas, Pum Walters, Pierre Weis, and Peter Wentworth. Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 6(04):621–655, 1996.

[14] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: must-alias analysis for higher-order languages. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 329–341. ACM, 1998.

[15] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 393–407. ACM Press, 1995.

[16] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[17] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[18] Panagiotis Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, 2001.

[19] Philippe Meunier, Robert B. Findler, and Matthias Felleisen. Modular set-based analysis from contracts. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 218–231. ACM, 2006.

[20] Jan Midtgaard. Control-flow analysis of functional programs. *ACM Computing Surveys*, 2011.

[21] Matthew Might and David Van Horn. A family of abstract interpretations for static analysis of concurrent Higher-Order programs. In Eran Yahav, editor, *Static Analysis*, volume 6887 of *Lecture Notes in Computer Science*, pages 180–197. Springer Berlin / Heidelberg, 2011.

[22] Matthew Might and Panagiotis Manolios. A posteriori soundness for non-deterministic abstract interpretations. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 260–274. Springer-Verlag, 2009.

[23] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM International Conference on Functional Programming (ICFP 2006)*, pages 13–25, 2006.

[24] Matthew Might and David Van Horn. Scalable and precise abstractions of programs for trustworthy software. Technical report, 2012.

[25] Greg Morrisett. Harvard university course cs252r: Advanced functional language compilation. `http://www.eecs.harvard.edu/~greg/cs252rfa12/`.

[26] Flemming Nielson and Hanne R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 332–345. ACM Press, 1997.

[27] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, 1981.

[28] John C. Reynolds. Definitional interpreters for Higher-Order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[29] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Trans. Program. Lang. Syst.*, 19(1):48–86, 1997.

[30] Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 537–554. ACM, 2012.

[31] David Van Horn and Matthew Might. Abstracting abstract machines: a systematic approach to higher-order program analysis. *Communications of the ACM*, 54:101–109, 2011.

[32] David Van Horn and Matthew Might. An analytic framework for JavaScript. *CoRR*, abs/1109.4467, 2011.

[33] David Van Horn and Matthew Might. Systematic abstraction of abstract machines. *Journal of Functional Programming*, 22(Special Issue 4-5):705–746, 2012.

[34] Dimitrios Vardoulakis and Olin Shivers. CFA2: a Context-Free approach to Control-Flow analysis. *Logical Methods in Computer Science*, 7(2), 2011.

[35] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Trans. Program. Lang. Syst.*, 19(1):87–152, 1997.

[36] Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Trans. Program. Lang. Syst.*, 20(1):166–207, 1998.

[37] Feng Zhao. An O(N) algorithm for Three-Dimensional N-Body simulations. Master's thesis, MIT, 1987.