Optimizing Abstract Abstract Machines

J. Ian Johnson
Northeastern University
ianj@ccs.neu.edu

Matthew Might
University of Utah
might@cs.utah.edu

David Van Horn
Northeastern University
dvanhorn@ccs.neu.edu

Abstract

Abstracting abstract machines has been proposed as a lightweight approach to designing sound and computable program analyses. The approach derives abstract interpreters from existing machine semantics and has been applied to a variety of languages with features widely considered difficult to analyze. Although sound analyzers are straightforward to build under this approach, they are also prohibitively inefficient.

This article contributes a step-by-step process for going from a naive analyzer derived under the abstracting abstract machine approach to an efficient program analyzer. 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

The *abstracting abstract machines* (AAM) approach [26, 28] to deriving program analyses provides a systematic way of transforming a programming language semantics in the form of an abstract machine into a family of abstract interpreters. The approach parameterizes these families with policies for regulating analytic precision. While flexible and robust, the AAM approach unfortunately 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.

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- [7] and thread-based [18] parallelism and behavioral contracts [25]; it has been used to model Coq [21], Dalvik [20], Erlang [8], JavaScript [27], and Racket [25].

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 [24], structured operational semantics [23], and reduction semantics [11]—are well understood [5], 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.

However, there is a considerable weakness with the approach: an analyzer designed and implemented by following the AAM recipe is prohibitively inefficient without both further approximation and further implementation effort.

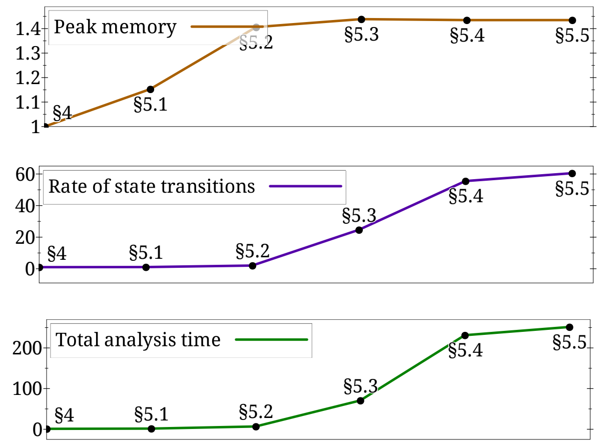


Figure 1. Factor improvements over the baseline analyzer for the Vardoulakis and Shivers benchmark in terms of peak memory usage, the rate of state transitions, and total analysis time. (Bigger is better.) Each point is marked with the section that introduces the optimization.

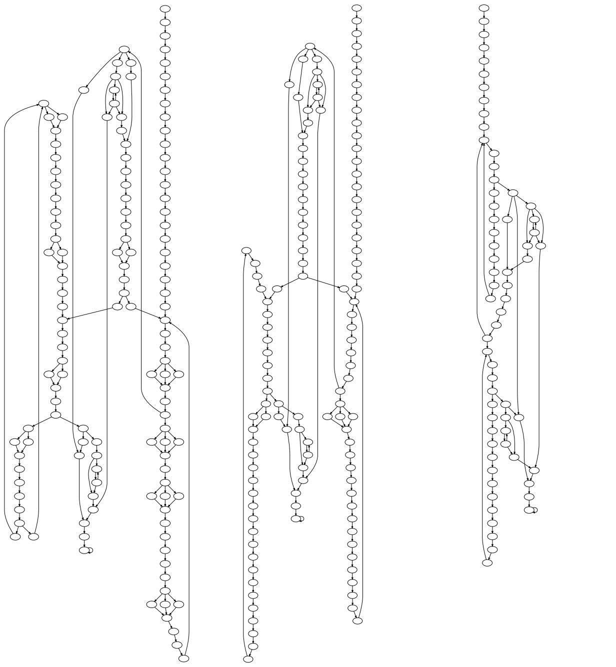
In this article, we develop a systematic approach to deriving the feasible implementation of an abstract-machine-based analyzer.

2. At a glance

This paper is organized in two halfs: in the first, 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. In the second half, 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 evaluate the implementation on a benchmark from Vardoulakis and Shivers [29] 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. it proves to be the *least* improved benchmark of the complete suite considered in section 6, and thus it serves as a good sanity



(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 intepretive 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.

check and lower-bound for each of the optimization techniques considered.

We start, in section 3, by developing an abstract interpreter according to the AAM approach. Without further abstraction, the analysis is exponential due to per-state store variance 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 [28, §3.5–6] and is necessary to make even small examples feasible. We therefore take the 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.

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

Figure 3. Syntax of ISWIM

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. [9]. 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 optimiztion 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.

3. An analyzer for 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 [**?**]. 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 (abstract) 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 compoents are defined in figure 4 and figure 5 defines the

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

Figure 4. Abstract machine components

$$
\mathit{eval}(e) = \{\varsigma \mid \texttt{ev}^{\epsilon}(e,\varnothing,\varnothing,\texttt{mt}) \longmapsto\!\!\!\!\rightarrow \varsigma\} \text{ where}
$$

$$
\begin{array}{rcl}
\texttt{ev}\ (\texttt{var}\ (x),\rho,\sigma,\kappa) & \longmapsto & \texttt{co}\ (\kappa,v,\sigma) \text{ where } v \in \sigma(\rho(x)) \\
\texttt{ev}\ (\texttt{lit}\ (l),\rho,\sigma,\kappa) & \longmapsto & \texttt{co}\ (\kappa,l,\sigma) \\
\texttt{ev}\ (\texttt{lam}\ (x,e),\rho,\sigma,\kappa) & \longmapsto & \texttt{co}\ (\kappa,\texttt{clos}\ (x,e,\rho),\sigma) \\
\texttt{ev}^{\delta}(\texttt{app}^{\ell}(e_0,e_1),\rho,\sigma,\kappa) & \longmapsto & \texttt{ev}^{\delta}(e_0,\rho,\sigma',\texttt{ar}^{\delta}_{\ell}(e_1,\rho,a)) \\
 & & \text{where } a,\sigma' = \mathit{push}^{\delta}_{\ell}(\sigma,\kappa) \\
\texttt{ev}^{\delta}(\texttt{if}^{\ell}(e_0,e_1,e_2),\rho,\sigma,\kappa) & \longmapsto & \texttt{ev}^{\delta}(e_0,\rho,\sigma',\texttt{fi}^{\delta}(e_1,e_2,\rho,a)) \\
 & & \text{where } a,\sigma' = \mathit{push}^{\delta}_{\ell}(\sigma,\kappa) \\
\\
\texttt{co}\ (\texttt{mt},\texttt{v},\sigma) & \longmapsto & \texttt{ans}\ (\sigma,v) \\
\texttt{co}\ (\texttt{ar}^{\delta}_{\ell}(e,\rho,a),v,\sigma) & \longmapsto & \texttt{ev}^{\delta}(e,\rho,\sigma,\texttt{fn}^{\delta}_{\ell}(v,a)) \\
\texttt{co}\ (\texttt{fn}^{\delta}_{\ell}(u,a),v,\sigma) & \longmapsto & \texttt{ap}^{\delta}_{\ell}(u,v,\kappa,\sigma) \text{ where } \kappa \in \sigma(a) \\
\texttt{co}\ (\texttt{fi}^{\delta}(e_0,e_1,\rho,a),\texttt{tt},\sigma) & \longmapsto & \texttt{ev}^{\delta}(e_0,\rho,\sigma,\kappa) \text{ where } \kappa \in \sigma(a) \\
\texttt{co}\ (\texttt{fi}^{\delta}(e_0,e_1,\rho,a),\texttt{ff},\sigma) & \longmapsto & \texttt{ev}^{\delta}(e_1,\rho,\sigma,\kappa) \text{ where } \kappa \in \sigma(a) \\
\\
\texttt{ap}^{\delta}_{\ell}(\texttt{clos}\ (x,e,\rho),v,\sigma,\kappa) & \longmapsto & \texttt{ev}^{\delta'}(e,\rho',\sigma',\kappa) \\
 & & \text{where } \rho',\sigma',\delta' = \mathit{bind}^{\,\delta}_{\,\ell}(\rho,\sigma,x,v) \\
\texttt{ap}^{\delta}_{\ell}(o,v,\sigma,\kappa) & \longmapsto & \texttt{co}\ (\kappa,v',\sigma) \text{ where } v' \in \Delta(o,v)
\end{array}
$$

Figure 5. Abstract abstract machine for ISWIM

transition relation. The evaluation of a program is defined as the set of states reachable by the reflexive, transitive closure of the machine transition relation. The machine is a very slight variation on a standard abstract machine for ISWIM in "eval, continue, apply" form [5]. It can be systematically derived from a definitional interpreter through a continuation-passing style transformation and defunctionalization, or from a structural operational semantics using the refocusing construction of Danvy and Nielsen [6].

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, not single values,
- continuations are heap-allocated, not stack-allocated,
- there are "contour values" (written δ) and syntax labels (ℓ) threaded through the computation, and
- the machine is implicitly parameterized by the functions *push*, *bind*, and Δ.

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

$$
\mathit{push}^{\delta}_{\ell}(\sigma,\kappa) = a, \sigma \sqcup [a \mapsto \{\kappa\}] \text{ where } a \notin \sigma
$$

$$
\mathit{bind}^{\delta}_{\ell}(\rho,\sigma,x,v) = \rho[x \mapsto a], \sigma \sqcup [a \mapsto \{v\}] \text{ where } a \notin \sigma
$$

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, this relation defines a deterministic machine. (The relation is really a function.)

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

$$
\begin{array}{ll}
z+1 \in \Delta(\texttt{add1},z) & z-1 \in \Delta(\texttt{sub1},z) \\
\texttt{tt} \in \Delta(\texttt{zero?},\texttt{0}) & \texttt{ff} \in \Delta(\texttt{zero?},z) \text{ if } z \neq \texttt{0}
\end{array}
$$

Abstract interpretation To characterize abstract interpretation, set the implicit parameters just as above, but drop the $a \notin \sigma$ condition. 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 an abstract interpretation [19]. 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. On the one hand is an abstract interpretation that is non-computable and gives only the ground truth of a programs behavior; on the other 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 [22] is one such analysis.

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

$$
\begin{aligned}
\mathit{push}^{\delta}_{\ell}(\sigma,\kappa) &= \ell\delta, \sigma \sqcup [\ell\delta \mapsto \{\kappa\}] \\
\mathit{bind}^{\delta}_{\ell}(\rho,\sigma,x,v) &= \rho[x \mapsto a], \sigma \sqcup [a \mapsto \{v\}], \delta' \\
\text{where } \delta' &= \lfloor \ell\delta \rfloor_k \\
a &= x\delta' \\
\lfloor \delta \rfloor_0 &= \epsilon \\
\lfloor \ell\delta \rfloor_{k+1} &= \ell \lfloor \delta \rfloor_k
\end{aligned}
$$

where $\sqcup$ on stores is a point-wise lifting of $\sqcup$: $\sigma \sqcup \sigma' = \lambda a.\sigma(a) \sqcup \sigma'(a)$. The $\rfloor \cdot \lfloor_k$ notation denotes the truncation of a list of symbols to the leftmost k symbols.

All that remains is the interpretation of primitives. For abstract intepretation, we set Δ to the function that returns $\texttt{Z}$ 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. Reduction semantics to baseline analyzer

The uniform k-CFA allocation strategy would make *eval* in figure 5 a computable abstraction of reachable states, but not an efficient one. This is not the strategy that AAM, nor we, recommend. Through this and the following section, we will explain a succession of approximations to reach the baseline analysis. We'll compare performance at each stage to identify the criticality of each optimization. We ground this journey 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.

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:

$$F_{\varsigma_0} : \mathcal{P}(State) \rightarrow \mathcal{P}(State).$$

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

$$F_{\varsigma_0}(S) = \{\varsigma_0\} \cup \{\varsigma' \mid \varsigma \in S \text{ and } \varsigma \longmapsto \varsigma'\}.$$

Then, the evaluator for the analysis computes the least fixed-point of the global transfer function:

$$eval(e) = \mathrm{lfp}(F_{\varsigma_0}),$$

where $\varsigma_0 = \mathtt{ev}^{\epsilon}(e, \varnothing, \varnothing, \mathtt{mt})$.

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'll fix this with a widening and reach polynomial (albeit of a high degree) complexity. This widening effictively 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, in one step, complexity drops from intractable exponentiality to merely daunting polynomiality.

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 will install this joined store into current state.

To describe this process, we'll refactor the transition 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 refactored relation, $\widehat{\longmapsto}$, becomes:

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

In effect, the new store is computed as the least upper bound of all subsequent stores.

4.3 Store-allocated results

The final approximation we make to get to our point of departure is store-allocating results of application sub-expressions. 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—even for a monovariant analysis like 0CFA. 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.

If we specialize to 0CFA, the evaluation rules become:

$$\begin{aligned}
\mathtt{ev}\ (\mathtt{app}^{\ell}(e_0, e_1), \rho, \sigma, \kappa) &\longmapsto \mathtt{ev}\ (e_0, \rho, \sigma \sqcup [\ell \mapsto \{\kappa\}], \mathtt{ar}\ (e_1, \ell)) \\
\mathtt{co}\ (\mathtt{ar}\ (e, \rho, \ell), v, \sigma) &\longmapsto \mathtt{ev}\ (e, \rho, \sigma \sqcup [\ell^f \mapsto \{v\}], \mathtt{fn}\ (\ell^f, \ell)) \\
\mathtt{co}\ (\mathtt{fn}\ (\ell^f, \ell), v, \sigma) &\longmapsto \mathtt{ap}_{\ell}(u, \ell^a, \kappa, \sigma \sqcup [\ell^a \mapsto \{v\}]) \\
&\quad \text{where } \kappa \in \sigma(a), u \in \sigma(a^f) \\
\mathtt{ap}_{\ell}(\mathtt{clos}\ (x, e, \rho), \ell^a, \sigma, \kappa) &\longmapsto \mathtt{ev}\ (e, \rho[x \mapsto x], \sigma \sqcup [x \mapsto \sigma(\ell^a)], \kappa) \\
\mathtt{ap}_{\ell}(o, \ell^a, \sigma, \kappa) &\longmapsto \mathtt{co}\ (\kappa, v', \sigma) \\
&\quad \text{where } v' \in \{\Delta(o, v) \mid v \in \sigma(\ell^a)\}
\end{aligned}$$

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) transition elimination and (2) pragmatic improvement. The transition-elimination optimizations reduce the overall number of transitions made by the analyzer by performing:

1. lazy non-determinism;
2. abstract compilation; and
3. uniform literal approximation.

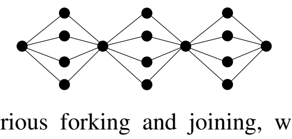
The pragmatic improvements reduce overhead and trade space for time by utilizing:

1. store deltas;
2. timestamped stores; and
3. preallocated data structures.

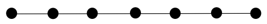
Some techniques preserve the precision of the underlying analysis, and others do not. For any technique that loses precision, we will discuss the design rationale for the move.

5.1 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 two-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:



In case it's unclear, 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 dereferencing an address until its contents are *needed* as values 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, $\texttt{addr}\ (a)$, that represents a delayed lookup of a. The following rules highlight the changes to the semantics:

$$
\begin{aligned}
&\mathit{force} : \mathit{Store} \times \mathit{Value} \to \mathcal{P}(\mathit{Value})\\
\mathit{force}(\sigma, \texttt{addr}\ (b)) &= \sigma(b)\\
\mathit{force}(\sigma, v) &= \{v\}\\
\mathit{ext}(\sigma, a, v) &= \sigma \sqcup [a \mapsto \mathit{force}(\sigma, v)]\\
\texttt{ev}\ (\texttt{var}\ (x), \rho, \kappa, \sigma) &\longmapsto\ \texttt{co}\ (\kappa, \texttt{addr}\ (\rho(x)), \sigma)\\
\texttt{co}\ (\texttt{ar}^{\delta}_{\ell}(e, \rho, a), v, \sigma) &\longmapsto\ \texttt{ev}^{\delta}(e, \rho, \mathit{ext}(\sigma, a^{f}, v), \texttt{fn}^{\delta}_{\ell}(a^{f}, a))\\
\mathit{bind}^{\,\delta}_{\,\ell}(\rho, \sigma, x, v) &= \rho[x \mapsto a], \mathit{ext}(\sigma, a, v), \delta'\\
\text{where } \delta' &= \lfloor \ell\delta \rfloor_{k}\\
a &= x\delta'
\end{aligned}
$$

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. With laziness, the reference will mean the original binding(s) of the variable or the new one, 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 might have 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 duplicating the reduction relation to specialize variable lookup. This works since in the semantics of ISWIM with store-allocated results consumes the value component of states in one step. This is not the case for semantics that replicate the value component across reductions, say for popping off exception handler frames. Further convolution is needed to remove the administrative nature of laziness in these semantics. Due to the increase of conceptual complexity for negligable benefit, we decided against this approach.

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. Indeed, were the variable reference a step later and another binding not made in that step, the results of the two approaches are the same.

5.2 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 is *precision-preserving* and 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 following example illustrates the essence of abstract compilation effect:

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

makes the following transitions:

$$
\begin{aligned}
&\texttt{ev}\ (\texttt{app}\ (\texttt{app}\ (\texttt{app}\ (x, e_1), e_2), e_3), \rho, \kappa, \sigma_0) && (1)\\
\longmapsto\ &\texttt{ev}\ (\texttt{app}\ (\texttt{app}\ (x, e_1), e_2), \rho, \texttt{ar}\ (e_3, \rho, a_1), \sigma_1) && (2)\\
\longmapsto\ &\texttt{ev}\ (\texttt{app}\ (x, e_1), \rho, \texttt{ar}\ (e_2, \rho, a_2), \sigma_2) && (3)\\
\longmapsto\ &\texttt{ev}\ (x, \rho, \texttt{ar}\ (e_1, \rho, a_3), \sigma_3) && (4)\\
\longmapsto\ &\texttt{co}\ (\texttt{ar}\ (e_1, \rho), v, \sigma_4) \text{ where } v \in \sigma(\rho(a)) && (5)
\end{aligned}
$$

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

$$
\begin{aligned}
\llbracket \_ \rrbracket &: \mathit{Expr} \to \mathit{Env} \times \mathit{Store} \times \mathit{Kont} \to \mathit{State}\\
\llbracket \texttt{var}\ (x) \rrbracket &= \lambda(\rho, \sigma, \kappa).\texttt{co}\ (\kappa, \texttt{addr}\ (\rho(x)), \sigma)\\
\llbracket \texttt{lit}\ (l) \rrbracket &= \lambda(\rho, \sigma, \kappa).\texttt{co}\ (\kappa, l, \sigma)\\
\llbracket \texttt{lam}\ (x, e) \rrbracket &= \lambda(\rho, \sigma, \kappa).\texttt{co}\ (\kappa, \texttt{clos}\ (x, \llbracket e \rrbracket, \rho), \sigma)\\
\llbracket \texttt{app}^{\ell}(e_0, e_1) \rrbracket &= \lambda^{\delta}(\rho, \sigma, \kappa).\llbracket e_0 \rrbracket^{\delta}(\rho, \sigma', \texttt{ar}^{\delta}_{\ell}(\llbracket e_1 \rrbracket, \rho, a))\\
&\quad \text{where } a, \sigma' = \mathit{push}^{\delta}_{\ell}(\sigma, \kappa)\\
\llbracket \texttt{if}^{\ell}(e_0, e_1, e_2) \rrbracket &= \lambda^{\delta}(\rho, \sigma, \kappa).\llbracket e_0 \rrbracket^{\delta}(\rho, \sigma', \texttt{fi}^{\delta}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket, \rho, a))\\
&\quad \text{where } a, \sigma' = \mathit{push}^{\delta}_{\ell}(\sigma, \kappa)
\end{aligned}
$$

Figure 6. Abstract compilation

$$\mathit{eval}(e) = \{\varsigma \mid \llbracket e \rrbracket(\epsilon, \varnothing, \varnothing, \texttt{mt}) \longmapsto\!\!\!\!\!\rightarrow \varsigma\} \text{ where}$$

$$
\begin{aligned}
\texttt{co}\ (\texttt{mt}, \texttt{v}, \sigma) &\longmapsto \texttt{ans}\ (\sigma, v)\\
\texttt{co}\ (\texttt{ar}^{\delta}_{\ell}(k, \rho, a), v, \sigma) &\longmapsto k^{\delta}(\rho, \sigma, \texttt{fn}^{\delta}_{\ell}(v, a))\\
\texttt{co}\ (\texttt{fn}^{\delta}_{\ell}(u, a), v, \sigma) &\longmapsto \texttt{ap}^{\delta}_{\ell}(v, u, \kappa, \sigma) \text{ where } \kappa \in \sigma(a)\\
\texttt{co}\ (\texttt{fi}^{\delta}(k_0, k_1, \rho, a), \texttt{tt}, \sigma) &\longmapsto k^{\delta}_{0}(\rho, \sigma, \kappa) \text{ where } \kappa \in \sigma(a)\\
\texttt{co}\ (\texttt{fi}^{\delta}(k_0, k_1, \rho, a), \texttt{ff}, \sigma) &\longmapsto k^{\delta}_{1}(\rho, \sigma, \kappa) \text{ where } \kappa \in \sigma(a)\\[1ex]
\texttt{ap}^{\delta}_{\ell}(\texttt{clos}\ (x, k, \rho), v, \sigma, \kappa) &\longmapsto k^{\delta'}(\rho', \sigma', \kappa)\\
&\text{where } \rho', \sigma', \delta' = \mathit{bind}^{\,\delta}_{\,\ell}(\rho, \sigma, x, v)\\
\texttt{ap}\ (o, v, \sigma, \kappa) &\longmapsto \texttt{co}\ (\kappa, v', \sigma)\\
&\quad \text{where } \kappa \in \sigma(a) \text{ and } v' \in \Delta(o, v)
\end{aligned}
$$

Figure 7. Abstract abstract machine for compiled ISWIM

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.

5.3 Locally log-based store deltas

Every step the analysis makes for the above techniques requires joining large stores together. 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. This uses far fewer join 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 $\sigma \sqcup [a \mapsto \mathit{vs}]$ becomes a log addition $(a, \mathit{vs}){:}\xi$, where ξ begins empty (ϵ) for each step. Applying the changes to the full store is straightforward:

$$\begin{aligned} \mathit{replay} &: \mathit{Store\Delta} \times \mathit{Store} \to \mathit{Store} \\ \mathit{replay}(\epsilon, \sigma) &= \sigma \\ \mathit{replay}((a, vs){:}\xi, \sigma) &= \mathit{replay}(\xi, \sigma \sqcup [a \mapsto vs]) \end{aligned}$$

The transition relation is identical except for the addition of this change log. Lookups will never rely on the change log, and can use the originally supplied store unmodified:

$$(\mathtt{ap}^{\delta}_{\ell}(\mathtt{clos}\ (x, e, \rho), v, \kappa), \sigma, \xi) \longmapsto (\mathtt{ev}^{\delta'}(e, \rho', \kappa), \xi')$$
$$\text{where } \rho', \xi', \delta' = \mathit{bind}^{\,\delta}_{\,\ell}(\rho, \sigma, \xi, x, v)$$

$$\begin{aligned} \mathit{bind}^{\,\delta}_{\,\ell}(\rho, \sigma, \xi, x, v) &= \rho[x \mapsto a], (a, \mathit{force}(\sigma, v)){:}\xi, \delta' \\ \text{where } \delta' &= \lfloor \ell\delta \rfloor_k \\ a &= x\delta' \end{aligned}$$

Compilation changes to additionally take a ξ component, so the above rule's right hand side would instead be $k^{\delta'}(\rho', \sigma, \xi', \kappa)$ where $k = [\![e]\!]$ would be in the closure.

We also lift $\longmapsto$ to accommodate for this asymmetry in the input and output. For each state that is stepped, we feed the output changes to the next so that all changes get accumulated:

$$\begin{aligned} (cs, \sigma) &\mathrel{\widehat{\longmapsto}} (cs \cup cs', \mathit{replay}(\xi, \sigma)) \\ \text{where } (cs', \xi) &= \mathit{step}^*(\varnothing, cs, \epsilon) \\ \mathit{step}^*(S, \varnothing, \xi) &= (S, \xi) \\ \mathit{step}^*(S, \{c\} \cup cs, \xi &= \mathit{step}^*(S \cup cs^*, cs, \xi^*) \\ cs^* &= \{c' \mid (c, \sigma, \xi) \longmapsto (c', \xi^c)\} \\ \xi^* &= \mathit{concat}\{\xi^c \mid (c, \sigma, \xi) \longmapsto (c', \xi^c)\} \end{aligned}$$

Here $\mathit{concat} : \mathcal{P}(\mathit{Store\Delta}) \to \mathit{Store\Delta}$ flattens the lists of changes to one; the order in which it appends does not matter:

$$\begin{aligned} \mathit{concat}(\varnothing) &= \epsilon \\ \mathit{concat}(\{\xi\} \cup D) &= \mathit{append}(\xi, \mathit{concat}(D)) \end{aligned}$$

5.4 Timestamping an imperative store

Thus far, we have made our optimizations in a purely functional manner. For the next series of optimizations, we need to dip into the imperative. We can motivate this entire sequence of optimizations by focusing on the largest bottleneck in the current state-space exploration algorithm: checking to see if a state has already been seen. Given two states, 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 is a subtle loss of precision in Shivers' original timestamp technique that we can fix. For a given abstract state, all writes to the global store need to be delayed until the analysis considers all branches from that abstract state. 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. Fortunately, given our conversion to log-based stores this change is straightforward.

```
eval($e$) :=
  $\sigma$, todo, seen, T := $\varnothing$, $\varnothing$, [], 0
  $[\![e]\!]$($\epsilon$, $\varnothing$, $\varnothing$, $\epsilon$, mt)
  while(true):
    if todo = $\varnothing$: return (keys(seen), $\sigma$)
    else:
      let old := todo
      todo, $\xi$ := $\varnothing$, $\epsilon$
      foreach c $\in$ old: $\Delta$? := false; c()
      unless $\xi$ = $\epsilon$: T += 1; replay!($\xi$, $\sigma$)
```

Figure 8. Imperative algorithm

At this point, we can begin to think of the analysis as an imperative algorithm with six major components:

- σ, the store;
- todo, the workset;
- seen, a map from states to the timestamps at which they were last seen;
- ξ, the store changes for the current step;
- Δ?, a boolean tracking whether the stepped state contributed a store change; and
- T, the timestamp of the store.

The new *eval* calculation is defined in figure 8. To ensure termination, we guard against adding c to todo by the following check:

$$\Delta\texttt{?} \vee \texttt{seen}(c) \neq \texttt{T}$$

If it succeeds, todo gets c and we update seen to map c to T.

After all steps complete, we apply ξ to σ imperatively (with replay!) and increase T as long as there was at least one change in ξ. This logic leads to termination if we know that each (a, vs) in ξ would change the value of a in the current store. Thus, we also guard additions to ξ so that only updates that would change the store are permitted. Each time ξ is successfully extended, we set Δ? to true. Before each individual step, we set it to false.

5.5 Pre-allocating the store

Internally, the algorithm at this stage uses hash tables to model the store. This is because stores used to be distributed to all states, which required a compact, dynamic representation. 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 analysis, 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.

5.6 Abstracting literal compound data

The abstraction of compound data structures like lists has deep implications for precision and performance. The classic list-heavy benchmark boyer—a simplified implementation of the Boyer-Moore theorem-prover—drove our own considerations for abstractions for compound data structures. Boyer's fluent use of literal list dooms the "natural" abstraction of lists to fail from overprecision. In short, if we interpret a literal list '(a b c) as (cons 'a (cons `'b (cons 'c '())))`, even a monovariant allocation stategy for abstract cons cells will precisely and exactly create a 3-cell list.

If code contains a length-300 literal list, then its analysis yields a length-300 abstract list as well. In many cases, the specific contents of the list add little precision to the resulting analysis, and yet the analyzer will dutifully execute recursive functions over the entirety of these structures at every encounter. An alternative in this case is to explicitly flatten literal lists into single abstract cells.

We explore both options here. We will first explain the natural abstraction of tuples, and then we will explain a less precise allocation strategy from uniform k-CFA that we use for large compound data literals that we modified from the implementation in [31].

5.6.1 Option 1: The natural abstraction

The uniform way AAM approaches a simple abstraction strategy is to cut recursion out of the data definition by tying the recursive knot through the abstract store. For Scheme, the grammar for values looks like the following:

$$\mathit{Value} ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{(cons}\ \mathit{Value}\ \ \mathit{Value}\texttt{)} \mid \texttt{'()} \mid \ldots$$

Upon evaluating a `cons` application, we instead allocate two addresses a and d, join them to the respective values in the store, and return the flattened $\texttt{(cons}\ \ a\ \ d\texttt{)}$ value. Since these addresses are all distinguished at different syntactic callsites in the uniform k-CFA allocation strategy, and quoted lists are sugar for a sequence of calls to `cons`, this abstraction explodes the value space. Analyzing a function that counts the number of atoms in a literal s-expression would actually interpret that function at least that number of times (more because of intermediate conses). Indeed, even in our fastest implementation, this abstraction causes the analysis of Boyer to be 430 times slower than the approach we will now describe.

5.6.2 Option 2: A precise yet compact abstraction

The number of syntactic uses of `cons` versus implicit uses via literal lists is smaller in typical Scheme programs. We use the above abstraction for these syntactic uses, but choose to interpret literal lists as not always sugar for cascading `cons`es. In particular, if a list literal is "too big" (in our case length > 1), we interpret the list as a circular data structure; the right address points back to the `cons` itself, and the left address points to all of the elements of the list. We take this farther and join the list elements together in with a coarse type-based abstraction. In effect, large lists or vectors of literal numbers become unbounded lists/vectors of "number." Heterogeneous data is combined to just "data," rather than a union of "number", "string", etc. since primitives are interpreted on every combination of values that flow to them, and unions lead to more primitive interpretations.

Quotation is special because it cannot introduce function values, which is important to enhancing our technique's soundness to conceptual complexity ratio. If we join two values of different type together, we don't get "anything," which has complex meaning in higher-order languages (Shivers' escape semantics) and is overly approximate. We instead get "any quotable value," which has much simpler semantics.

There are a few steps to consider:

- Define special value lattice elements for compound data domains that can be quoted (e.g. `QPair` for $\{(\texttt{cons}\, a_i\, d_i)\}_i$, `QVector` for immutable vectors, etc.)
- Define a "larger" value lattice element for all quotable data, `QData`
- Interpret `(quote (a ...))` as `(qlist a ...)`, a new primitive function defined in figure 9, and similar definitions for immutable vectors.
- Extend the Δ axioms to include conservative meaning for these new values (e.g. `(car QData) = QData`, `(add1 QData) = Number` and log "possible type error") and allow them to allocate addresses and change the store

$$\begin{aligned}
\Delta(\sigma,\ \texttt{qlist}) &= (\texttt{'()}, \sigma) \\
\Delta(\sigma,\ \texttt{qlist}, v\ldots_{+}) &= ((\texttt{cons}\ a\ d), \sigma') \\
\text{where } \sigma' &= \sigma \sqcup [a \mapsto \bigsqcup(v\ldots)] \\
&\qquad \sqcup [d \mapsto \{\texttt{'()}, (\texttt{cons}\ a\ d)\}] \\
\bigsqcup(v) &= v \\
\bigsqcup(v, vs\ldots) &= \mathit{merge}(v, \bigsqcup(vs\ldots)) \\
\mathit{merge}(v, v) &= v \\
\mathit{merge}(n, m) &= \texttt{Number} \\
\mathit{merge}(n, v) &= \texttt{QData} \\
\mathit{merge}((\texttt{cons}\ a\ d), (\texttt{cons}\ a'\ d')) &= \texttt{QPair} \\
\mathit{merge}(\texttt{QPair}, (\texttt{cons}\ a\ d)) &= \texttt{QPair} \\
\mathit{merge}(\texttt{QPair}, v) &= \texttt{QData} \\
&\vdots
\end{aligned}$$

Figure 9. Quoted list primitive

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 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, 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.

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 [12] and analyses (e.g. [13, 31]). 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

Program	LOC	Time (s)		Space (MB)		Speed (state/s)	
nucleic	3492	m	57.8	m	138	m	7K
matrix	747	t	4.9	t	63	t	68K
nbody	1435	t	38.3	t	124	t	53K
earley	667	1.1K	0.7	409	60	252	41K
maze	681	t	4.0	t	60	t	80K
church	42	44.9	0.2	86	60	714	43K
lattice	214	348.5	0.4	231	60	382	72K
boyer	642	m	18.3	m	93	m	33K
mbrotZ	69	373.6	0.1	295	60	540	34K

Figure 10. Overview performance comparison between baseline and optimized analyzer (entries of t mean timeout, and m mean out of memory).

and columns and negation of any subset of rows and columns. It is written in continuation-passing style (used in [13, 31]).

3. **nbody**: implements the Greengard multipole algorithm for computing gravitational forces on point masses distributed uniformly in a cube (used in [13, 31]).
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 [13, 31]).
6. **church**: tests distributivity of multiplication over addition for Church numerals (introduced by [29]).
7. **lattice**: enumerates the order-preserving maps between two finite lattices (used in [13, 31]).
8. **boyer**: a term-rewriting theorem prover (used in [13, 31]).
9. **mbrotZ**: generates Mandelbrot set 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 [13, 31]).

Figure 10 gives an overview of the benchmark results in terms of absolute time, space, and speed between the baseline and most optimized analyzer. Figure 11 plots the factors of improvement over the baseline for each optimization step. The dip we see in transition rate even though time taken decreases is to be expected - fewer "easy" states are added by abstract compilation. It increases again with the introduced algorithmic improvements. Accumulating store changes in addition to maintaining the store accounts for the higher memory usage when using the store delta technique without further improvements.

Comparison with other flow analysis implementations The analysis considered here computes results similar Earl, et al.'s 0-CFA implementation [9], 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 [29] 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 [30] and, in many ways its successor, the Polymorphic splitting system of Wright and Jagannathan [31].[1] Unfortunately, these analyses compute an inherently different and uncomparable 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.

7. Related work

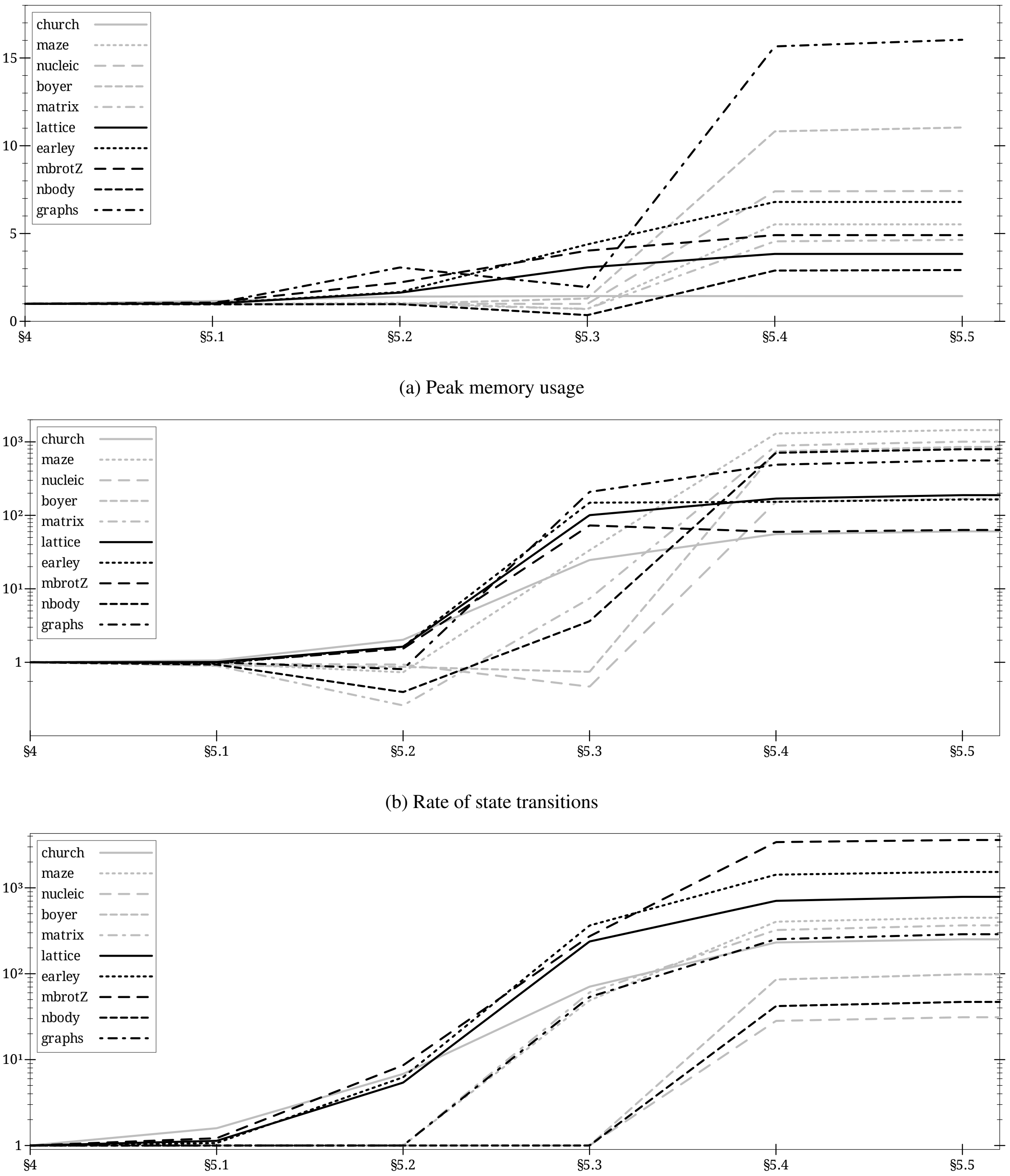
Abstracting Abstract Machines This work clearly closely follows Van Horn and Might's original papers on abstracting abstract machines [26, 28], which in turn is one piece of the large body of research on flow analysis for higher-order languages (see Midtgaard [17] for a thorough survey). The AAM approach sits at the confluence of two major lines of research: (1) the study of abstract machines [15] and their systematic construction [24], and (2) the theory of abstract interpretation [3, 4].

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 [22] and the unified treatment of flow analysis by Jagannathan and Weeks [14]. 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.

Abstract compilation Boucher and Feeley [1] introduced the idea of *abstract compilation*, which used closure generation [10] to improve the performance of control flow analysis. We have adapted the closure generation technique from composition evaluators to abstract machines and applied it to similar effect.

Constraint-based program analysis for higher-order languages Constraint-based program analyses (e.g. [16, 22, 31]) 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.

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



(a) Peak memory usage

(b) Rate of state transitions

(c) Total analysis time

Figure 11. Factors of improvement over baseline for each step of optimization (bigger is better).

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 [31] analyses the Vardoulakis and Shivers benchmark about 25 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. These 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.

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.

8. Conclusion

Abstract machines are not only a good model for rapid analysis development, they can be systematically developed into efficient algorithms.

Acknowledgments We thank Suresh Jagannathan for providing source code to the polymorphic splitting analyzer [31] and Ilya Sergey for the introspective pushdown analyzer [9].

References

[1] Dominique Boucher and Marc Feeley. Abstract compilation: A new implementation paradigm for static analysis. In *Compiler Construction: 6th International Conference*, 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] Patrick Cousot and Radhia Cousot. 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*, pages 238–252, 1977.

[4] 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.

[5] Olivier Danvy. *An Analytical Approach to Program as Data Objects*. DSc thesis, Aarhus University, 2006.

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

[7] E. D'Osualdo, J. Kochems, and C.-H. L. Ong. Automatic verification of Erlang-style concurrency. Technical report, University of Oxford, 2012. `http://mjolnir.cs.ox.ac.uk/soter/papers/erlang-verif.pdf`.

[8] E. D'Osualdo, J. Kochems, and C.-H. L. Ong. Soter: An automatic safety verifier for Erlang. Technical report, University of Oxford, 2012. `http://mjolnir.cs.ox.ac.uk/soter/papers/soter-demo.pdf`.

[9] 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.

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

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

[12] Pieter H. Hartel, Marc Feeley, et al. Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. *J. Func. Prog.*, 6(04):621–655, 1996.

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

[14] 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.

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

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

[17] 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.

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

[19] Matthew Might and David Van Horn. A family of abstract interpretations for static analysis of concurrent Higher-Order programs. In *Static Analysis*, vol. 6887 of *LNCS*, pages 180–197, 2011.

[20] 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*, pages 260–274. Springer-Verlag, 2009.

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

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

[23] 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.

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

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

[26] 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.

[27] 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.

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

[29] David Van Horn and Matthew Might. Systematic abstraction of abstract machines. *J. Func Prog*, 22(4-5):705–746, 2012.

[30] Dimitrios Vardoulakis and Olin Shivers. CFA2: a Context-Free approach to Control-Flow analysis. *LMCS*, 7(2), 2011.

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

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