Systematic constructions for higher-order program analyses

J. IAN JOHNSON

The problem

Higher-order program analyses are difficult to design, engineer, and use.

Designing analyses is difficult for two major reasons: language faithfulness and property expressiveness. Modern languages are large and have semantics that are not "mathematically pure," making them difficult to faithfully model abstractly. Further complications come from higher-order language features, since there is no apparent control-flow graph that most tools rely on as a starting point. "Analysis" covers tools that can discover properties of programs or verify that written properties are valid. Verification properties themselves can be written in pure logic or the tooled language itself. Both have their strengths and weaknesses in terms of designer effort and user effort, which are often at odds with each other.

Engineering correct and performant analyses is also no easy feat. Some problems that need tackling include finding a data representation for good allocation and cache behavior, avoiding dispatch costs, and avoiding re-computation. Performance gains through pure blood, sweat, and tears are respectable, but the resulting code ends up problematic. Code that is not apparently correct is difficult to read, write, maintain, and most importantly, trust.

Using analyses such as model-checkers, interactive theorem provers, proof assistants and a further category of "unsound analyses" requires expert knowledge. The first three commonly have their own language for expressing programs and sometimes yet another language for expressing properties of those programs. Common programmers do not express their correctness properties as, *e.g.*, formulae in monadic second-order logic, but rather assertions or better yet contracts. Moreover, programmers need tools for *their* languages, not the specialized and restricted languages of niche groups of researchers. Full correctness is almost never defined, much less attainable, and thus programmers are happy for any help guaranteeing partial correctness. A key word is "guarantee" because there are several analysis tools that are unsound — they do not report some possible errors when they have not verified the errors are impossible. Unsoundness means we cannot trust the tool's report to justify performance-improving program transformations. Thus either an expert must perform them by hand, or convince the compiler to perform them when they are convinced (sometimes erroneously)

that it is safe. The sound tools like interactive theorem provers and proof assistants have a caveat in their names: the user must assist the tool, which requires expert knowledge.

In short, program analyses require expertise in all domains of design, engineering and use, making them all-around costly.

My thesis

Precise and performant analyses for higher-order languages can be systematically constructed from their semantics.

My focus in analysis is for non-interactive methods such as flow analysis and model-checking (note: not type systems). Outside the scope of this thesis, one can imagine using the methods I describe to design an interactive tool[1]. Flow analysis in higher-order languages encompasses both control-flow analysis (CFA) and data-flow analysis. CFA answers the question, "what functions could be called where? Or, more generally, what are the possible control transfers in the program?" Data-flow analysis answers a similar question, "what data could flow where?" but when functions are values, the two fit together. Traditionally, flow analysis is lightweight (low polynomial time), but its specification is broad enough to include undecidable program verification problems.

Model-checking is defined as exhaustively and automatically checking whether a model meets its specification. In a language with contracts, a program's specification is part of the program itself, and thus specification checking is built into the semantics. Model-checking, though broadly defined, is often attributed to the verification of temporal properties of finite models rather than simply functional correctness properties. As such, we focus on temporal properties in higher-order languages as enforced by temporal higher-order contracts. Flow analysis provides a way to soundly infer the finite models on which model-checkers operate, but can additionally perform the checking too. By connecting specification to semantics with contracts, verification becomes a reachability problem in the abstract: when we run a program "in abstract" and never reach blame, then all contracts hold when run "in concrete."

A fortunate corollary of contract verification is that we can soundly remove their runtime enforcement. Conversely, if in the abstract we reach blame, we have an abstract path to failure that could be further refined for debugging purposes. We can create a concrete counter-example via constraint-solving, or simply report the endpoint or path as a possible failure. The upside to using contracts to enforce properties at runtime is that we can squelch reporting a class of errors and still get blame at runtime. With this approach, model-checking becomes flow analysis, further blurring the line between the different research areas. There are multiple advanges to the method of systematically transforming a concrete semantics into a computable and terminating abstract semantics. To name a few:

[1] *e.g.*, Andy Keep's semantic grep `https://github.com/Ucombinator/Tapas`

1. we can maintain confidence in the correctness of the verification tool,

2. we make the construction process simpler and more broadly applicable to other languages,

3. we can factor out important abstraction points to make a pluggable analysis framework, and

4. analysis mechanisms have a concrete semantics that can be tested prior to a confounding abstraction process.

To demonstrate this thesis, I have explored two research programs: systematic transformations for improving precision and performance of abstract interpreters, and concrete and abstract semantics of temporal contracts. This proposal describes what I have accomplished with each program so far, and what remains to be done.

1 Background

The fields of abstract interpretation and model-checking have historically focused on analyzing first-order languages. Focusing on the first-order world pigeonholed these communities into analyzing C or Fortran, solely, with only a sketch of the semantics of these languages in mind. In the higher-order world, we have first-class functions, objects, control operators, stack inspection and other not-so-simple features that are used every day, but are poorly understood by our analyses. The plethora of opportunities for innovation in the world of programming language semantics creates a demand for more widely applicable analysis techniques as more and more programmers use and depend on them. An approach to solving this problem, called abstracting abstract machines (AAM) [32], is a systematic process of transforming a programming language semantics directly into an analysis for that language.

The philosophy of AAM is to finitize a reduction relation by first redirecting all recursive data-structures through the machine's heap, and second to allocate only finitely many addresses in the heap. The finite source of addresses pigeonholes some values to reside in the same address, so for soundness purposes, the heap must map addresses to sets of values. Heap lookups then do not have the same type, so the reduction relation then non-deterministically chooses a value from the fetched set. The "analysis" is then a calculation of this abstracted reduction relation. While delightfully simple, a direct implementation of an application of AAM is too slow to analyze even the simplest programs. The upside is a straightforward-to-apply and simple-to-prove method for designing one's analysis. Contrasting AAM with the focused work on first-order languages and constraint-based techniques for higher-order lanugages, we see large disparities in performance.

2 Systematic semantics transformations to abstract interpreters

2.1 Optimizing abstract abstract machines (OAAM)

Slogan *Engineering tricks are semantics refactorings*

Practical program analysis implementations employ a wide range of engineering tricks in order to be performant, but in the process obscure correctness with respect to the original semantics. A contributing factor to the lack of rigor or connection to semantics in these techniques is the unchallenged assumption that analysis is only for C- or Fortran-like languages. These languages are essentially Imp with second-class functions and maybe pointers and maybe heap allocation. Higher-order program analysis comes before any "first-orderization" to something C-like, and thus can use higher level guarantees from the high-level semantics to make more precise predictions. Unfortunately, the techniques pioneered for the first-order world of Fortran do not immediately carry over to languages with first-class functions, objects, or control operators. The reason is there is no "syntactic" control-flow graph that the analysis can assume approximates the implicit reduction relation of the underlying semantics. Analyses for higher-order languages thus look much more like interpreters for the language rather than graph algorithms.

The connection between interpreters and higher-order flow analyses lead to the birth of AAM, a new mindset for analysis design. The problem with AAM was that direct implementations were too slow for practical use. AAM produces abstract semantics that have several points of waste in terms of time and memory:

1. the above described non-determinism creates many more states than are necessary, increasing memory pressure and computation time;
2. expressions always perform the same actions, so many states are unnecessarily represented and additionally take time from dispatch on expression type;
3. store widening is necessary for accelerating convergence, but joining several entire stores each step is wasteful; and
4. specialized knowledge of the allocation strategy affords represention specialization.

After studying existing analysis implementations and discovering new techniques myself, patterns emerge that are directly applicable to any abstract machine.

My contribution: performance with proofs

Store-counting, store-allocated values, lazy non-determinism, abstract compilation, store deltas and preallocation give a 1000 factor improvement [17]. Most of

these techniques are known in a slightly different formulation, and are mostly only described informally and thus never proved correct. Each technique in this paper is accompanied with a simple proof of correctness that follows either from subject reduction or a straightforward case of stuttering bisimulation. See the appendix for the evaluation details.

Further work

I have unpublished work on techniques exploiting "sparseness" that preliminary evaluation shows a factor of 8 performance improvement in a store-widened semantics. This should be written down in more detail to present in the dissertation.

2.2 The essence of summarization

Slogan

Summarization is memoization, and memoization is keyed by enough context to encapsulate behavior

Summarization is a technique for interprocedural program analysis first devised by Sharir and Pnueli [29] for a first-order language. The original intention was to get better precision of interprocedural bitvector analyses on recursive programs, and in doing so they found a precise way to match function calls with their associated returns. The article did not mention pushdown automata, so it may be safe to assume that the connection between summarization and pushdown automata was not discovered until later. The added precision of properly matched calls and returns is desirable for any analysis doing heavy lifting; a computationally intensive lattice operation performed on additional spurious paths can lead to significant performance hits.

Summarization in the first-order setting was still connected heavily to an up-front representation of the control-flow graph, so it was not readily applicable to higher-order languages. Vardoulakis and Shivers described an analysis of a higher-order CPS language (called CFA2) that uses an adapted form of summarization to make it "online" in the way that higher-order analyses require. The summarization mechanism in both cases is only described post-abstraction as an imperative worklist algorithm that drives a "local" reduction relation that otherwise would get stuck on continuation invocations. Later the authors of CFA2 described an augmented form of this imperative algorithm so that CFA2 could also analyze programs that use the `call/cc` control operator. This is a semantic extension to a language that breaks well-bracketed calls and returns. Its addition removed the guarantee that the summarization algorithm is a complete abstraction of the semantics with abstracted bindings but still unbounded stacks. However, in the common case CFA2 still improves upon regular analyses in terms of precision and performance.

CFA2 taught us the lesson that summarization is an approach to improving precision that just so happens to completely abstract pushdown automata. Models of computation that do not easily abstract into pushdown automata can still benefit from summarization.

My contribution: summarizing semantics of delimited control and abstract garbage collection

I found that the summarization algorithm in CFA2 has an alternative description as a semantics refactoring that still works in the concrete. A simple abstraction process then produces a direct-style cousin of CFA2. The refactoring is to realize that summarization is memoization for control transfers, and memoization in the context of impure functions, stack-awareness (either stack inspection, capture, replacement or extension) and other non-functiony things is not keyed by the arguments of functions, but rather it is more generally keyed by "enough context" to reproduce the result in the presence of different surrounding context. In CFA2 without `call/cc`, since all bindings are allocated in the heap, "enough context" was simply the function being called, and the heap.

An operational encoding of a memoizing machine is just to add a stack frame at function calls that stores the function and arguments so that when it evaluates to a value, we can store the result in a global memo table for later lookup. What we notice is that the rest of the continuation under such a frame does not matter to the function's execution, so we can throw it into a table also. The only states that we see are within functions, where the continuation is chopped off with a, "go look up where all I should return to when I'm done" directive. When we apply the AAM approach to this, and share the tables across the non-deterministic paths that arise, the result is a polyvariant, direct-style CFA2.

If we naively apply the same method to the abstract machine for delimited control, the resulting analysis may not terminate. The reason is that the "context" we put in a continuation includes the heap, but captured continuations are allocated in the heap. The AAM philosophy directs us to remove the circularity with an indirection through a separate table that is not storeable in the heap. Thus, captured continuations get further approximated by replacing the represented heap with an address to some number of heaps. We call the map of addresses to heaps the "continuation closure" for lack of a better term.

The continuation closure is essentially an extension of the heap, and thus they would then have to map addresses to pairs of heaps and continuation closures! We seem to get into an infinite regress, but we can bottom out with a join; when we would store a closure in the closure, we instead join them together. Memoization and returns must be updated to reflect the new approximate form of context, but that is straightforward.

Consider the CEK machine with a "memo table" and an additional stack frame that we insert at function calls that will direct the semantics to memoize the result to the key stored in said stack frame (Figure 1). This semantics is a complete abstraction of the CEK machine which we can prove given an invariant of the memo table:

$$inv(\langle e, \rho, \kappa, M\rangle) = \forall \kappa, ((e,\rho),(v,\rho')) \in M. \\ \exists \pi \equiv \langle e, \rho, \kappa\rangle \longmapsto^*_{CEK} \langle v, \rho', \kappa\rangle . hastail(\pi, \kappa)$$

We can prove this invariant with subject reduction with a further lemma that traces with a common continuation tail are still valid traces when the continuation

$$\varsigma \longmapsto \varsigma'$$

$\langle x, \rho, \kappa, M\rangle$	$\langle v, \rho, \sigma, \kappa, M\rangle$ if $(v, \rho) = \rho(x)$
$\langle e_0\ e_1, \rho, \kappa, M\rangle$	$\langle e_0, \rho, \texttt{arg}\ (e_1, \rho){:}\kappa\rangle$
$\langle v, \rho, \texttt{arg}\ (e, \rho'){:}\kappa, M\rangle$	$\langle e, \rho', \texttt{fun}\ (v, \rho){:}\kappa, M\rangle$
$\langle v, \rho, \texttt{fun}\ (\lambda x.\ e, \rho'){:}\kappa, M\rangle$	$\langle e, \rho'', \texttt{ret}\ (e, \rho''){:}\kappa, M\rangle$ if $(e, \rho'') \notin \mathbf{dom}(M)$
	or
	$\langle v', \rho''', \kappa, M\rangle$ if $(v', \rho''') = M(e, \rho'')$
where	$\rho'' = \rho'[x \mapsto (v, \rho)]$
$\langle v, \rho, \texttt{ret}\ (e, \rho'){:}\kappa, M\rangle$	$\langle v, \rho, \kappa, M[(e, \rho') \mapsto (v, \rho)]\rangle$

Figure 1: CEKM machine

is replaced by any other continuation. The continuations below **ret** frames don't matter to the result of the function, so we can throw them into a table to look up at return time in order to produce a *summary*. A summary represents who called which functions with what arguments, and what were the observed input/output pairs for functions throughout execution.

Abstract garbage collection [24] is a technique used to improve the precision of flow analyses that requires knowing all addresses that the stack keeps alive. In the pushdown model, such stack introspection is not possible. With summarization, the entirety of the stack is readable through the continuation table that is built during execution, making touched address calculation trivial. Recent work by the original author of abstract GC defined an entirely new model of pushdown automata that has restricted rules for reading the current stack [12]. Again such detours are unnecessary, and do not offer insight into how one might model first-class control.

Further work

The proofs I have for summarizing delimited control are yet to be peer-reviewed, so that is a priority. The stack-capturing and stack-extending capabilities of `shift` and `reset` also give but one case study for the generality of "context" for summarizing analyses. A further case study of stack inspection would be strong evidence that my semantics for summarization is capable of features more difficult to model in pushdown automata. A programming language with the primitives `with-continuation-mark`, `current-continuation-marks`, and `continuation-mark-set->list` provides a generous model of stack inspection to study. I conjecture that the result of `current-continuation-marks` is enough context to include for sound summarization. I propose to evaluate the summarizing semantics against the AAM-style regular semantics on a representative set of benchmarks for performance and precision.

2.3 Related work

Abstracting Abstract Machines

This work clearly closely follows Van Horn and Might's original papers on abstracting abstract machines [33, 34]. AAM in turn is one piece of the large body of research on flow analysis for higher-order languages (see Midtgaard [23]

for a thorough survey). The AAM approach sits at the confluence of two major lines of research: (1) the study of abstract machines [20] and their systematic construction [26], and (2) the theory of abstract interpretation [7, 8].

Frameworks for flow analysis of higher-order programs

Besides the original AAM work, the analysis most similar to that presented in OAAM is the infinitary control-flow analysis of Nielson and Nielson [25] and the unified treatment of flow analysis by Jagannathan and Weeks [16]. 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 [5] introduced the idea of abstract compilation, which used closure generation [13] 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. [25, 38, 22, 30]) 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 [6]. Consequently, constraint-based analyses can be computed quickly. For example, Jagannathan and Wright's polymorphic splitting implementation [38] analyses the Church numeral 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?" The state transition graphs can answer these kinds of queries, but evaluation demonstrated this does not come for free.

Pushdown analyses

The insights into the essence of summarization are directly the product of CFA2 [35], which itself derives from Sharir and Pnueli [29]. By breaking down the semantic features of CFA2 into orthogonal components, summarization and stack frames, we see that PDCFA [11] is simply CFA2 without stack frames. PDCFA's analysis method uses graph traversals to find stack push sites rather than storing context in the state itself for a table lookup. Additionally, PDCFA does these traversals for each stack frame pop instead of at function call boundaries, which is costly and unnecessary.

The work in pushdown garbage collection was done concurrently and independently of Earl et al. [12], based on the original work on abstract garbage collection [24]. I am now a co-author on the journal version of that work, although I maintain that my summarization approach is more general (can handle first-class control), and simpler to apply to an arbitrary language since it has a concrete interpretation.

Further afield, there is work in higher-order recursion schemes (HORS) [19] for model-checking higher-order programs that is co-expressive with a generalization of pushdown systems: collapsible pushdown systems. The translation from arbitrary languages to HORS to begin with is non-trivial unless your language is pure and simply typed. To handle more complex languages, the HORS community developed an undecidable, untyped variant that they would later abstract within their infinite intersection type machinery to approximate [31]. There is work translating the Krivine machine to HORS [27], but the methods employed are not widely applicable to other machines. Our focus with systematic translations directly from the language semantics seeks to make analysis techniques more widely applicable.

Finally, there is a temporal logic based on nested words that uses summarization as part of its saturation method of checking propositions, called NWTL [3]. The theory here again assumes a program model already in a finite nested word automaton, and uses a non-online algorithm so higher-order languages could not be easily analyzed.

3 Temporal higher-order contracts

Temporal higher-order contracts are a runtime monitoring system for specifying the temporal use of higher-order functions [10]. Flow analysis has historically

been used for proving "shallow" properties (one or two quantifiers in a temporal logic formula, if so expressed [28]), whereas model-checkers have been used for proving properties with deeply nested temporal modalities. Furthermore, model-checkers are applied extra-linguistically: a user will pose a formula in a propositional temporal logic for a model-checker to check against a synthesized model, necessarily abstractly. The programmer cannot dynamically test such propositions to debug their specification before attempting verification, because the formula is outside of the language. The field of runtime monitoring has created solutions to the latter, but runtime monitoring has never been the source of model-checking queries.

Contract verification is a large area that has produced techniques that are complementary to temporal contract verification. The first hurdle to overcome for verifying temporal contracts is to define them. The published work on temporal contracts has a problematic formalization and implementation; both violate expectations about their behavior. Specifically, the temporal contracts are given a denotational semantics in terms of *prefixes* of full program traces. Our expectations for a negated temporal contract are that once execution matches the temporal contract under negation, we blame. However, a contract such as $\neg A$ allows AA in its full trace semantics (since the literature defines $[\![\neg T]\!] = \mathit{Traces} \setminus [\![T]\!]$), which in turn allows A in the prefix semantics.

My contributions

I defined a new semantics of temporal contracts that is aware of its prefix interpretation, necessitating a different interpretation of negation. Our expectation is that any (nonempty) trace in the denotation of a contract should not be a *prefix of* any trace in the denotation of the *negated* contract:

$$[\![\neg T]\!] = \{\epsilon\} \cup \{\pi : \forall \pi' \in [\![T]\!] \setminus \{\epsilon\}.\pi' \not\sqsubseteq \pi\}$$

The language of temporal contracts closely resembles regular expressions, but allow for matching on events to later perform equality tests. Regular expression derivatives still apply in this domain, with a slight reworking to accomodate matching and a different rule for negation:

$$\partial_A \neg T \stackrel{\text{def}}{=} \nu(\partial_A T) = \epsilon \to \bot, \neg\ \partial_A T$$

I have proved that this new notion of derivative is indeed a derivative with respect to the denotational semantics ($[\![\partial_A T]\!] = \{\pi : A\pi \in [\![T]\!]\}$). The interpretation is that a runtime monitor is a temporal contract derivative, and it should blame the sender of the event that made the derivative not nullable (*i.e.*, blame ℓ if ℓ sends A and $\nu(\partial_A T) \neq \epsilon$).

Further work

The abstract semantics of temporal contracts has posed some serious challenges. Equality checking during matching is unusably imprecise in the abstract, since syntactically equal closures only *may* be equal. We can import a technique such

as abstract counting to improve equality checking to allow for *may*, *must* and *never*, but there are further problems. The derivatives of temporal contracts have a combinatorial explosion due to the added uncertainty in matching. The derivative of a temporal contract is a set of possible derivatives, each of which must be explored as the possible next state of the runtime monitor. Composing possible derivatives in $\cup, \cap$ and $\cdot$ forms leads to a large space of possibilities. I propose to explore possible widenings and/or representational optimizations to make this approach feasible for moderately sized programs.

3.1 Related work

Runtime monitoring

Monitoring sequences of actions at runtime is a mature and active area of research. Temporal higher-order contracts are themselves a runtime monitoring system. The notion of an action is reminiscent of aspect-oriented programming's notion of a *join-point*, and thus we see several systems built on AspectJ [18] that offer a domain-specific language for running *advice* when the action trace matches a specified pattern, *e.g.*, Tracematches [2] and J-LO [4]. Tracematches use a language similar to temporal contracts but do not support negation; they also only provide a way to execute given *advice*, and not monitor the satisfaction of a specification. J-LO on the other hand is a monitoring system based on LTL propositions with binding constructs (named DLTL). J-LO's goal is closer to temporal contracts, but its language is not; conversely, tracematches match the language and not the goal. Both systems are also tied to Java's class structure, so they cannot natively express properties of higher-order functions or refinements on values.

Interface automata [1] give a lightweight mechanism for monitoring component interactions that send first-order data. They are closely related to temporal contracts, except they do not have binding in the specifications and thus cannot reason about higher-order functions. There is also no discussion of any implementation of the monitoring system.

Further discussion of runtime monitoring systems can be found in Meredith et al. [21].

Model-checking

The behemoth of model-checking higher-order languages is the Java PathFinder (JPF) [37], which offers both an explicit state model-checker, and a symbolic execution engine. Although the language (Java) is higher-order, both the model-checker and symbolic execution engine can only check first-order properties. The JPF is its own JVM implementation, so one could reasonably see temporal higher-order contracts implemented in a straight-forward way.

Additional model-checkers for Java such as Bandera or ESC/Java use different methods. The former uses CFA to extract a finite model to translate into a back-end model checker's language. This has obvious precision deficiencies since data- and path-sensitive information that a model-checker has does not prune the model the way an intermixed model construction and model checking analysis

would. The latter is an unsound invariant checker for first-order assertions that generates verification conditions and uses a theorem prover (Simplify [9]) to discharge them.

4 Research plan

Below are the research projects I've outlined, along with my estimated time investment:

Project	Date range	Time
Summarization	Proposal - Feb 15	∼1.5 *mo.* (1 *mo.* paternity leave)
Temporal contracts	Feb 15 - Apr 15	2 *mo.*
Stack inspection	Proposal - Apr 15	parallel
Writing	Apr 15 - Aug 15	4 *mo.*
Total		**7 *mo.***

References

[1] de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering. pp. 109–120. ACM Press (2001)

[2] Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Moor, O.D., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: In OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications. pp. 345–364. ACM Press (2005)

[3] Alur, R., Arenas, M., Barcelo, P., Etessami, K., Immerman, N., Libkin, L.: First-Order and temporal logics for nested words pp. 151–160 (2007)

[4] Bodden, E.: J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University (2005)

[5] Boucher, D., Feeley, M.: Abstract compilation: A new implementation paradigm for static analysis. In: Gyimóthy, T. (ed.) Compiler Construction: 6th International Conference, CC'96 Linköping, Sweden, April 2426, 1996 Proceedings. pp. 192–207 (1996)

[6] Bravenboer, M., Smaragdakis, Y.: 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)

[7] Cousot, P., Cousot, R.: 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. pp. 238–252. ACM (1977)

[8] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 269–282. POPL '79, ACM (1979)

[9] Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)

[10] Disney, T., Flanagan, C., McCarthy, J.: Temporal higher-order contracts. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) ICFP. pp. 176–188. ACM (2011)

[11] Earl, C., Might, M., Van Horn, D.: Pushdown Control-Flow analysis of Higher-Order programs. In: Workshop on Scheme and Functional Programming (2010)

[12] Earl, C., Sergey, I., Might, M., Van Horn, D.: Introspective pushdown analysis of higher-order programs. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. pp. 177–188. ICFP '12, ACM (2012)

[13] Feeley, M., Lapalme, G.: Using closures for code generation. Comput. Lang. 12(1), 47–66 (1987)

[14] Hartel, P.H., Feeley, M., Alt, M., Augustsson, L., Baumann, P., Beemster, M., Chailloux, E., Flood, C.H., Grieskamp, W., Groningen, J.H.G.V., Hammond, K., Hausman, B., Ivory, M.Y., Jones, R.E., Kamperman, J., Lee, P., Leroy, X., Lins, R.D., Loosemore, S., Röjemo, N., Serrano, M., Talpin, J.P., Thackray, J., Thomas, S., Walters, P., Weis, P., Wentworth, P.: Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. Journal of Functional Programming 6(04), 621–655 (1996)

[15] Jagannathan, S., Thiemann, P., Weeks, S., Wright, A.: 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. pp. 329–341. ACM (1998)

[16] Jagannathan, S., Weeks, S.: 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. pp. 393–407. ACM Press (1995)

[17] Johnson, J.I., Labich, N., Might, M., Van Horn, D.: Optimizing abstract abstract machines. In: Morrisett, G., Uustalu, T. (eds.) Proceedings of the 18th ACM SIGPLAN international conference on Functional programming. ACM SIGPLAN, ACM Press (2013)

[18] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Proceedings of ECOOP 2001 - Object-Oriented Programming: 15th European Conference, Budapest, Hungary, June 18-22, 2001. pp. 327–354. Springer Verlag (2001)

[19] Kobayashi, N.: Model-checking higher-order functions. In: Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming. pp. 25–36. PPDP '09, ACM (2009)

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

[21] Meredith, P., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. International Journal on Software Tools for Technology Transfer (STTT) pp. 1–41 (2011)

[22] Meunier, P., Findler, R.B., Felleisen, M.: Modular set-based analysis from contracts. In: POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 218–231. ACM (2006)

[23] Midtgaard, J.: Control-flow analysis of functional programs. ACM Computing Surveys (2011)

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

[25] Nielson, F., Nielson, H.R.: 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. pp. 332–345. ACM Press (1997)

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

[27] Salvati, S., Walukiewicz, I.: Krivine machines and higher-order schemes. In: Proceedings of the 38th international conference on Automata, languages and programming - Volume Part II. pp. 162–173. ICALP'11, Springer-Verlag (2011)

[28] Schmidt, D.A., Steffen, B.: Program analysis as model checking of abstract interpretations. In: SAS '98: Proceedings of the 5th International Symposium on Static Analysis. pp. 351–380. Springer-Verlag (1998)

[29] Sharir, M., Pnueli, A.: Two Approaches to Interprocedural Data Flow Analysis, chap. 7, pp. 189–233. Prentice-Hall, Inc. (1981)

[30] Steckler, P.A., Wand, M.: Lightweight closure conversion. ACM Trans. Program. Lang. Syst. 19(1), 48–86 (1997)

[31] Tsukada, T., Kobayashi, N.: Untyped recursion schemes and infinite intersection types. In: Proceedings of the 13th International Conference on Foundations of Software Science and Computational Structures. pp. 343–357. FOSSACS'10, Springer-Verlag (2010)

[32] Van Horn, D., Might, M.: Abstracting abstract machines. In: ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. pp. 51–62. ICFP '10, ACM (2010)

[33] Van Horn, D., Might, M.: Abstracting abstract machines: a systematic approach to higher-order program analysis. Communications of the ACM 54, 101–109 (2011)

[34] Van Horn, D., Might, M.: Systematic abstraction of abstract machines. Journal of Functional Programming 22(Special Issue 4-5), 705–746 (2012)

[35] Vardoulakis, D., Shivers, O.: CFA2: a Context-Free Approach to Control-Flow Analysis. Logical Methods in Computer Science 7(2:3), 1–39 (2011)

[36] Vardoulakis, D., Shivers, O.: CFA2: a Context-Free approach to Control-Flow analysis. Logical Methods in Computer Science 7(2) (2011)

[37] Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. In: Ledru, Y., Alexander, P., Flener, P. (eds.) The Fifteenth IEEE International Conference on Automated Software Engineering Proceedings. vol. 10, pp. 203–232. IEEE, Springer Netherlands (2003)

[38] Wright, A.K., Jagannathan, S.: Polymorphic splitting: an effective polyvariant flow analysis. ACM Trans. Program. Lang. Syst. 20(1), 166–207 (1998)

[39] Zhao, F.: An O(N) Algorithm for Three-Dimensional N-Body Simulations. Master's thesis, MIT (1987)

A OAAM evaluation results

The analysis is evaluated against a suite of Scheme benchmarks drawn from the literature.

1. **nucleic**: a floating-point intensive application taken from molecular biology that has been used widely in benchmarking functional language implementations [14] and analyses (e.g. [38, 15]). 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 [38, 15]).

Program	LOC	Time (sec)		Space (MB)		Speed $\frac{state}{sec}$	
nucleic	3492	m	66.9	m	238	44	9K
matrix	747	t	3.4	294	114	68	87K
nbody	1435	t	22.9	361	171	67	57K
earley	667	1.1K	0.4	409	114	252	95K
maze	681	t	2.6	332	114	55	118K
church	42	44.9	0.1	86	114	714	56K
lattice	214	348.5	0.2	231	114	382	104K
boyer	642	m	13.4	m	130	39	39K
mbrotZ	69	373.6	0.1	295	114	540	63K

Figure 2: Overview of benchmark results

3. **nbody**: implementation [39] of the Greengard multipole algorithm for computing gravitational forces on point masses distributed uniformly in a cube (used in [38, 15]).

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 [38, 15]).

6. **church**: tests distributivity of multiplication over addition for Church numerals (introduced by [36]).

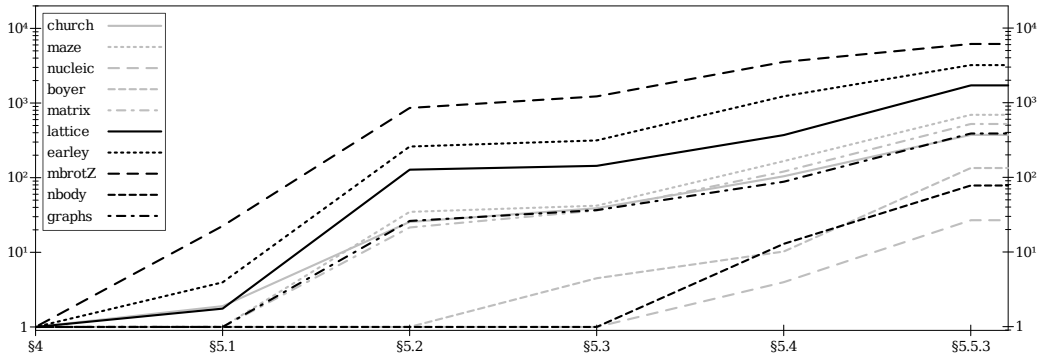
7. **lattice**: enumerates the order-preserving maps between two finite lattices (used in [38, 15]).

8. **boyer**: a term-rewriting theorem prover (used in [38, 15]).

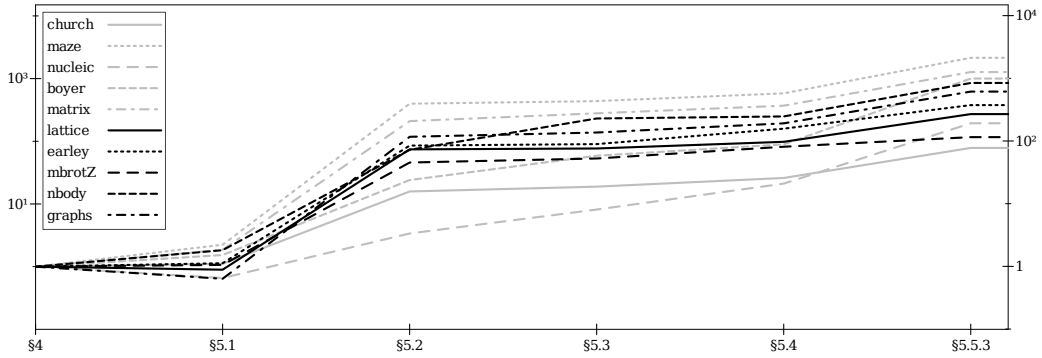
9. **mbrotZ**: generates Mandelbrot fractal using complex numbers.

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 [38, 15]).

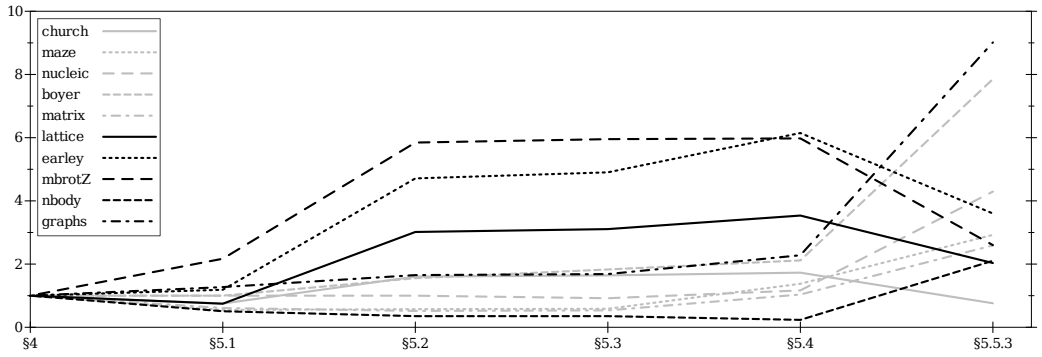
Figure 2 gives an overview of the benchmark results in terms of absolute time, space, and speed between the baseline and most optimized analyzer. Figure 3 plots the factors of improvement over the baseline for each optimization step.



(a) Total analysis time speed-up (baseline / optimized)



(b) Rate of state transitions speed-up (optimized / baseline)



(c) Peak memory usage improvement (baseline / optimized)

Figure 3: Factors of improvement over baseline for each step of optimization (bigger is better).