Multi-core Parallelization of Abstracted Abstract Machines

Leif Andersen
University of Utah
leif@leifandersen.net

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

Abstract

It is straightforward to derive well-known higher-order flow analyses as abstract interpretations of well-known abstract machines. In this paper, we explore multi-core parallel evaluation of one such abstract abstract machine, the CES machine. The CES machine is a variant of CESK machines that runs Continuation Passing Style (CPS) λ-calculus. Using k-CFA, the concrete semantics for a CES machine can be turned into abstract semantics. Analyzing a program for this machine is a state graph walk, which can be run in parallel to increase performance.

1. Introduction

Control Flow Analysis [13] (CFA) is too slow. There are techniques to increase performance by widening the abstract semantics of call sites [2]. However, widening comes at the cost of lost precision. It is natural to ask: can the speed of CFA be improved without reducing precision. The answer appears to be yes.

Unlike an abstract machine's concrete semantics, which will execute deterministically, an abstract machine's abstract semantics may branch and execute multiple nondeterministic paths. Each of these paths are independent, and unless they later merge into one path, the result of analyzing one path is distinct from the other. This independence provides an opportunity to analyze code in parallel.

The execution of a program on an abstract machine can be represented as a graph, as visualized in Figure 1. Points where CFA would benefit from parallelization are the places where the graph follow multiple paths, such as the right half of Figure 1. Portions of the execution that have no nondeterministic behavior, such as the left of Figure 1 will have no benefit.

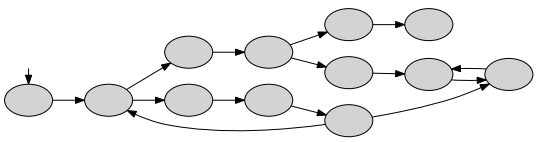

Figure 1. An abstract state space.

Due to the chaotic nature of the splits, and the potential for splits to rejoin, creating a new thread for every split would lead to a significant overhead. However, a queue with states to be explored, along with a fixed number of producers to explore them will cause little overhead.

In Section 2 we discuss what Control Flow Analysis is, and how Control, Environment, Store, and Continuation (CESK) machines are used. In section 3 we discus the language that will be run. In Section 4, the concrete and abstract CES machine will be analyzed. Section 5 discusses what can be run in parallel, and how the changes are made. Section 6 tests the performance of parallel CFA. Section 7 discus related works. Finally, Section 8 concludes this paper.

2. CESK and Control Flow Analysis

Control, Environment, Store, and Continuation [4] (CESK) machines can run versions of the λ-calculus. If the continuations are placed in the code itself (resulting in Continuation Passing Style (CPS) λ-calculus), than the machine need not store the continuations, thus reducing the machine to a Control, Environment, and Store (CES) machine. The standard CES machine is an abstract machine with concrete semantics. However, with a few modification, CES machines can execute abstract semantics.

For the CES machine to execute a program, it must be inserted into the machine. This is done by attaching an empty environment and store to the program, and setting the control to evaluate the input. In a CESK machine a halt continuation would also be required, as the machine would keep track of where the program would flow next. The explicit continuations are not required for a CES machine, as they are stored in the program itself.

From there a small step analyzer will evaluate the expression, modifying the control, environment, and store as expected. If this process is repeated one of three things will happen. The program can terminate, at which point a complete state graph is achieved, alternatively it can loop infinitely, with one state looping back to a previous state, where a complete state graph is still achieved. Both of these outcomes are fine from an analysis standpoint. The final outcome is that the program will loop infinitely, generating a new state each iteration. This generates an infinite amount

of states, and thus is impossible to analyze fully. Figure 2 visualizes the third case.

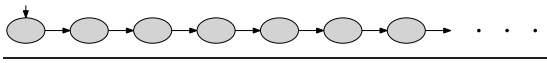

Figure 2. A concrete state space.

The problem with generating an infinite state space with these concrete semantics is undecidability. An infinite state space can not be explored. Abstract machines with abstract semantics fix this problem. While the input for an abstract machine with abstract semantics is equivalent to its concrete semantics counterpart, the abstract semantics changes input such that a small step function will eventually reach a fixed point, and terminate. In order to do this, the abstract semantics cannot guarantee that every state will produce exactly one next state or halt. Rather, with the abstract semantics, every state can produce a finite set of next states, which the analysis will follow non-deterministically. Figure 1 demonstrates the program flow through an abstract machine.

When the abstract semantics perfectly resemble the concrete semantics, every small step will have exactly one output state. The further the abstract semantics is from properly representing the concrete semantics, the more states are returned when evaluating each state.

3. The Language

In this section, we start with the continuation passing style (CPS) variant of the λ-calculus. Figure 3 describes the input language for the machine.

$$
\begin{array}{rcl}
pr \in \mathsf{Prog} & ::= & \mathsf{CExp} \\
v \in \mathsf{Var} & \text{is} & \text{a set of identifiers} \\
lam \in \mathsf{Lam} & ::= & (\lambda\ (v_1\ ...\ v_n)\ ce) \\
f, \text{æ} \in \mathsf{AExp} & ::= & lam \\
 & | & v \\
ce \in \mathsf{CExp} & ::= & (f\ \text{æ}_1\ ...\ \text{æ}_n) \\
 & | & \mathbf{halt}
\end{array}
$$

Figure 3. Syntax for CPS Style λ-Calculus.

The main difference between the language in Figure 3 and the regular untyped λ-calculus is that atomic expressions (æ) are not complex expressions (ce), which makes incapable of returning variables or literals. As every expression requires an additional expression to be evaluated, **halt** allows program to terminate. Figure 4 demonstrates the identity function in CPS λ-calculus.

```
((λ (x) (x x)) (λ (x) halt))
```

Figure 4. CPS λ-Calculus Expression

4. Abstracting the CES Machine

The first step towards creating a parallel algorithm for CFA is to specify a concrete semantics. The abstract semantics will follow. The specific version of CFA that we are using is k-CFA [11].

4.1 Concrete Semantics

Figure 5 shows the concrete CES machine. A state is comprised of an expression as declared in Figure 3, and an environment and store which maps variables to closures.

$$
\begin{array}{rcl}
\varsigma \in \Sigma & = & \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \\
\rho \in \mathit{Env} & = & \mathsf{Var} \to \mathit{Addr} \\
\sigma \in \mathit{Store} & = & \mathit{Addr} \to D \\
d \in D & = & \mathit{Clo} + \mathbf{halt} \\
\mathit{clo} \in \mathit{Clo} & = & \mathsf{Lam} \times \mathit{Env}
\end{array}
$$

$a \in \mathit{Addr}$ is an infinite set of addresses

Figure 5. Concrete CES state space.

An inject function as defined below takes the initial program and injects it into and empty environment and state.

$$\mathcal{I} : \mathsf{Prog} \to \Sigma$$
$$\mathcal{I}(pr) = (pr, [], [])$$

Atomic expression evaluation determines the value of an atomic expression. It takes an expression, environment, and store. It is defined as:

$$\mathcal{A} : \mathsf{EXP} \times \mathit{Env} \times \mathit{Store} \to D$$
$$\mathcal{A}(v, \rho, \sigma) = \sigma(v, \rho(v))$$
$$\mathcal{A}(lam, \rho, \sigma) = (lam, \rho)$$

There is also a transition network that maps every state to the following state.

$$(\Rightarrow) \subseteq \Sigma \times \Sigma$$

The relationship between transitions requires the expression to be evaluated, as well as the environment and store. It extends the environment to map the atomic expression to a fresh address, and the store to map the new address to its value. It is defined as:

$$([\![(f\ \text{æ}_1\ \ldots\ \text{æ}_n)]\!], \rho, \sigma) \Rightarrow (ce, \rho'', \sigma')$$

where $([\![(\lambda\ (v_1 \ldots v_n)\ ce)]\!], \rho') = \mathcal{A}(f, \rho, \sigma)$
$d_i = \mathcal{A}(\text{æ}_i, \rho, \sigma)$
a_i is fresh in σ
$\rho'' = \rho'[v_i \mapsto a_i]$
$\sigma' = \sigma[a_i \mapsto d_i]$

Explore gives the state graph. It takes an initial state, and using the transition relation, determines the next state. This is repeated until the halt state is reached, and there are no more states to explore. If no halt state occurs, a state may still loop back to a previous state, which would cause the evaluation to terminate. Additionally, the explore function may not terminate, and generate an infinite amount of states.

$$explore : \Sigma \to \mathcal{P}(\Sigma)$$
$$explore(\varsigma) = \{\varsigma' \mid \varsigma \Rightarrow^* \varsigma'\}$$

To evaluate the expression with the concrete semantics, the expression will be injected into the initial state space, and then the graph will be explored, the graph may be infinite in size.

4.2 Abstract Semantics

Figure 6 shows the abstraction of the CES machine using k-CFA [11]. To abstract the CES semantics, the store is limited to a finite amount of addresses. The effect of this change affects the store. Unlike the concrete semantics where the store is a mapping between addresses and closures, in the abstract semantics the store is a mapping between addresses and sets of closures.

$$\begin{array}{rcl} \varsigma \in \widehat{\Sigma} & = & \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \\ \rho \in \widehat{Env} & = & \mathsf{Var} \to \widehat{Addr} \\ \sigma \in \widehat{Store} & = & \widehat{Addr} \to \mathcal{P}(\widehat{D}) \\ d \in \widehat{D} & = & \widehat{Clo} + \mathbf{halt} \\ clo \in \widehat{Clo} & = & \mathsf{Lam} \times \widehat{Env} \end{array}$$

$a \in \widehat{Addr}$ is a finite set of addresses

Figure 6. Abstract CES state space.

The abstract semantics also injects the input into an initial state. The injection works the same as concrete injection, and is defined below.

$$\mathcal{I} : \mathsf{Prog} \to \widehat{\Sigma}$$
$$\mathcal{I}(pr) = (pr, [], [])$$

The atomic expression evaluation for the abstract semantics is also analogous to the concrete semantics, and is defined as:

$$\mathcal{A} : \mathsf{EXP} \times Env \times Store \to \widehat{D}$$
$$\mathcal{A}(v, \rho, \sigma) = \sigma(v, \rho(v))$$
$$\mathcal{A}(lam, \rho, \sigma) = (lam, \rho)$$

Like the concrete semantics, the abstract semantics has a transition relation. Unlike the concrete semantics, each state generates a set of next states.

$$(\rightsquigarrow) \subseteq \Sigma \times \Sigma$$

Evaluation of the states works analogously to the concrete semantics. They differ in the address space. Rather than an infinite set of addresses, the abstract semantics has a finite set of addresses. The store becomes a mapping from addresses to sets of values. This is what causes each state to have multiple next states. The abstract semantics are defined below as:

$$(\llbracket(f\ æ_1\ \ldots\ æ_n)\rrbracket, \rho, \sigma) \Rightarrow (ce, \rho'', \sigma')$$

$$\begin{array}{l} \text{where } (\llbracket(\lambda\ (v_1 \ldots v_n)\ ce)\rrbracket, \rho') = \mathcal{A}(f, \rho, \sigma) \\ \quad d_i = \mathcal{A}(æ_i, \rho, \sigma) \\ \quad a_i \text{ is fresh in } \sigma \\ \quad \rho'' = \rho'[v_i \mapsto a_i] \\ \quad \sigma' = \sigma[a_i \mapsto d_i] \end{array}$$

Like the concrete semantics, explore gives the state graph. Explore takes an abstract state, and will return all of the states that can be reached. This is often achieved using fixed points. The transition function determines the set of next states, if all of these states have already been seen, the explore is finished, otherwise the transition for the next states is found. This is done until there are no new states to find. The definition of explore is:

$$\widehat{explore} : \widehat{\Sigma} \to \mathcal{P}(\widehat{\Sigma})$$
$$\widehat{explore}(\varsigma) = \{\varsigma' \mid \varsigma \rightsquigarrow^* \varsigma'\}$$

Unlike the concrete semantics, the abstract semantics will always reach a fixed point. There will be a finite amount of states to explore. The cause of this is the finite amount of addresses.

If this were a CESK, rather than only CES, machine, the continuations would also need to be finitized. However as the code itself tracks the continuations, this is not required.

5. Parallelization

There are two parts of k-CFA that can be parallelized. The first is during state transition, and the second is during function application. In both of these parts, the state graph will branch non-deterministically, and be independent of other branches.

In the abstract semantics, transition will map one state, to a set of next states. Each of these next states can be explored independent of each other. This is what allows k-CFA to be parallelizable.

Two branches of states can merge into a single state later in the state graph. Thus, when a state produces multiple states, it is inefficient to analyze each state space independent of each other. Doing so would lead to a significant amount of duplicated work.

A better model would be a producer and consumer model. Figure 7 depicts this model. The initial state is injected into the queue. One of the producers takes the state off of the queue, and finds its next states. The state is then placed in the visited set, and all of the next states that have not already been visited are added back into the queue. This cycle is repeated until the queue is empty.

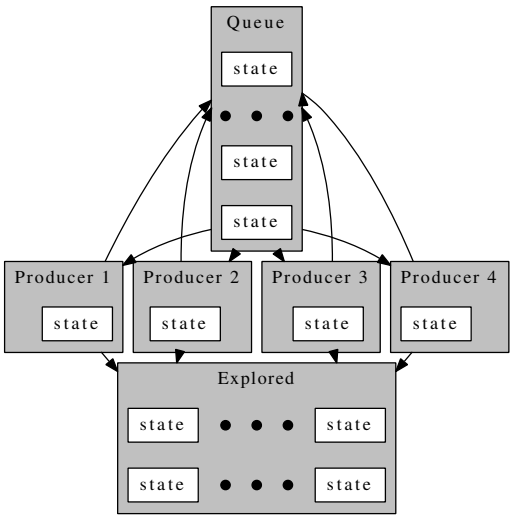

Figure 7. Queue and actor architecture for parallelizing k-CFA.

Scala's actors system makes this an easy system to implement [5]. Figure 8 depicts the code that implements the parallel explore.

```
def explore(in: Map[State,Set[State]]):
    Map[State,Set[State]] = {
  var next = in
  var producers =
    new StateIterator[(State,StateProducer)]
  for(i <- in)
    getProducers(producers, i._2)
  for(i <- producers) {
    var tmpStep = Set[State]()
    for(j <- i._2.iterator)
      tmpStep ++= j
    next += (i._1 -> tmpStep)
    getProducers(producers, tmpStep)
  }
  return next
}
```

Figure 8. Parallel explore function for CES machine in Scala.

Function application can also be run in parallel. This has little benefit for the CES machine as function application contains only atomic members, and thus do not generate new states. In a full CESK machine functions application would generate new states. Figure 9 demonstrates this faster version of function application.

```
case ApplyState(f, x, s) => {
  val tmpProducers =
    for(c <- x) yield new EvalProducer(c);
  val b = for (c <- tmpProducers) yield {
    var tmpSet = Set[Closure]()
    for(i <- c.iterator) tmpSet ++= i
    tmpSet
  }
  for(a <- aevalState(f))
    yield closureToEval(aapply(a, b, s), s)
}
```

Figure 9. Parallel function application in Scala.

The code in Figure 9 uses the same queue depicted in Figure 7, however rather than placing the result back in the queue, it is kept for the application.

6. Results

Both regular k-cfa and parallel k-cfa compute the same resulting state space. Thus the most important metric is run time.

Writing benchmark suites directly in the lambda calculus is impractical. Therefore, we created a richer language to run the test suites in.

Figure 10 is a richer language than the one in Figure 3. The language is compiled down into the untyped λ-calculus, and is then CPS converted, resulting in the same input language as described in Figure 3.

We use a factorial program to test the performance of parallel CFA. Figure 11 shows a sample of the factorial program calculating the factorial of 5. Similar programs were also tested for factorial of 0, 10, 15, and 20.

```
(letrec ((f (λ (n)
         (if (= n 0)
             1
             (* n (f (- n 1)))))))
  (f 5))
```

Figure 11. Factorial of five.

The program was tested on multiple processors, in particular an Intel i7-3770k, and an Intel i7-3630QM. Both processors have 4 cores, and both processors have hyper-threading, which was enabled during the tests.

Table 1 shows the results of running the code on the Intel i7-3770k. The test is initially ran a few times to allow any Just In Time (JIT) compiler Scala uses to run, and the speedup is taken by calculating the average runtime of 20 runs.

$$
\begin{array}{rcl}
pr \in \mathsf{Prog} & ::= & \mathsf{exp} \\
v \in \mathsf{Var} & \text{is} & \text{a set of identifiers} \\
n \in \mathsf{Nat} & \text{is} & \text{a positive integer} \\
lam \in \mathsf{Lam} & ::= & (\lambda\ (v_1\ ...\ v_n)\ ce) \\
prim \in \mathsf{Prim} & ::= & \#t \\
 & | & \#f \\
 & | & n \\
ce \in \mathsf{CExp} & ::= & (f\ ce_1\ ...\ ce_n) \\
 & | & lam \\
 & | & v \\
 & | & (if\ ce\ ce\ ce) \\
 & | & (zero?\ ce) \\
 & | & (-\ ce\ ce) \\
 & | & (=\ ce\ ce) \\
 & | & (+\ ce\ ce) \\
 & | & (*\ ce\ ce) \\
 & | & (letrec\ ((v\ lam))\ ce)
\end{array}
$$

Figure 10. Syntax for a Lisp Subset

Size	Speedup
0	6.8
5	8.8
10	9.2
15	9.1
20	8.5

Table 1. Results of parallel CFA analyzing factorial on an i7-3770k.

This demonstrates that the program is CPU bound. The average speedup of the parallel algorithm is 8.5 times that of the naive algorithm. The Scala runtime will create new actors when there is both work to be done, and unused CPU cores.

Table 2 runs the same test except on an Intel i7-3630QM. Like the i7-3770k, it also has four cores and hyper-threading. It was also released around the same time as the i7-3770k.

Size	Speedup
0	4.7
5	8.8
10	7.3
15	7.4
20	7.4

Table 2. Results of parallel CFA analyzing factorial on an i7-3630QM.

The average speedup of the parallel algorithm to the naive one is 7.16. This speedup fits the model that the program is running approximately eight different actors simultaneously.

The program was run on multiple processors to show that the results of running parallel CFA is not limited to one computer, but will lead to benefits on any multi-core platform it is run on.

6.1 A larger example

To further test parallel CFA, we created a Collatz Conjecture program. The input language in Figure 10 does not contain many of the functions required to calculate the hailstone sequence of a number, so many helper functions make it possible. Figure 12 is the hailstone sequence with 5 as its input.

```
(letrec ((even? (λ (n)
    (if (= n 0)
      #t
      (if (= n 1)
        #f
        (even? (- n 2)))))))
  (letrec ((div2* (λ (n s)
      (if (= (* 2 n) s)
        n
        (if (= (+ (* 2 n) 1) s)
          n
          (div2* (- n 1) s))))))
    (letrec ((div2 (λ (n)
        (div2* n n))))
      (letrec ((hailstone* (λ (n count)
          (if (= n 1)
            count
            (if (even? n)
              (hailstone*
                (div2 n) (+ count 1))
              (hailstone*
                (+ (* 3 n) 1)
                (+ count 1)))))))
        (letrec ((hailstone (λ (n)
            (hailstone* n 0))))
          (hailstone 5))))))
```

Figure 12. Collatz Conjecture on a sequence starting with 5.

The code will evaluate to the amount of steps the sequence took to complete. Table 3 demonstrates the results of running the Collatz Conjecture program on an i7-3630QM.

Size	Speedup
5	2.3

Table 3. Results of analyzing the Collatz Conjecture on an i7-3630QM.

The speedup for the Collatz Conjecture is significantly slower than that of factorial. This can be explain with the state graph, shown in Figure 13. Only a small portion of the graph can be run in parallel.

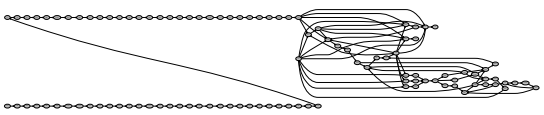

Figure 13. State space for Collatz Conjecture

This demonstrates a limitation in parallel CFA. Not every state space is highly parallelizable.

6.2 Pathologically Bad Cases

The goal of k-CFA is to model the concrete semantics of a language as accurately as possible, while still remaining decidable. k-CFA will be an accurate model to varying degrees for various programs. If it does a perfect job modeling the program, then it will be a series of sequential states, such as in Figure 2. This state graph is the worst possible case for parallel CFA. Each state produces exactly zero or one next states. Even though one CPU core is kept busy, the remaining ones will remain idle.

Figure 14 demonstrates an example where parallel CFA will do poorly (the U Combinator applied to itself). No matter how big the store is allowed to be, one state will always produce exactly one additional state. Figure 15 demonstrates this with a relatively small store.

```
((λ (x) (x x)) (λ (x) (x x)))
```

Figure 14. Pathologically bad code for parallel CFA.

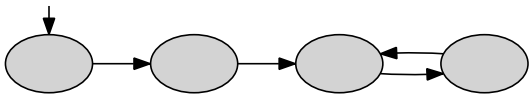

Figure 15. State space for pathologically bad case.

This means that parallel CFA can not be used as an optimization to k-CFA for every state space. It does however serve as a protection when k-CFA has its worst case, when a state branches into many next state. With parallel CFA, these results will return fairly quickly.

7. Related Works

The study of interpretation of the λ-calculus with abstract machines began with Landin's SECD machine [7], as well as the POPL papers from Coust and Coust [2, 3]. And Jones's static analysis of the λ-calculus [6]. However, combining abstract machines and interpretation has been a very recent thing [9, 10].

Analyzing abstract machines in parallel has had very little research. However, Prabhu's, Ramalingam's, Might's, and Hall's paper [12] on EigenCFA discus accelerating flow analysis through the use of GPUs. However, EigenCFA can only run on Two CPS code, which limits the expressiveness of the input language. Unlike the parallel CFA analyzed in this paper, it is not trivial to modify the EigenCFA implementation to run on CPS with additional arguments, let alone code that isn't CPS at all.

Burtscher, Narse, and Pingali also published [1] a paper using GPUs for general purpose computation. Part of this general purpose computation was control flow analysis. Méndez-Lojo, Matthew, and Pingali also published a paper on analysis in parallel [8]. In this paper they are performing constraint solving in parallel, while parallel CFA concerns performing higher order program analysis in parallel.

8. Conclusion

Our goal was to create a parallel version of k-CFA that balances the concerns of simplicity and scalability. We wanted to increase the speed at which it runs, while keeping the underlying theory behind k-CFA in tact. While it does not work for cases like in Figure 15, most programs will have some speedup. Additionally, we have shown that there significance speedup for simple programs.

Other possible avenues of research is to run it on a richer language. If a parallel CESK, rather than a CES, machine is created, we can analyze the full untyped λ-calculus, rather than the continuation passing style discussed in this paper.

This work has the potential to speed up k-CFA proportionally to the number of processors that a CPU contains.

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0106. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.

[2] P. Cousot and R. 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. ACM, 1977.

[3] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979.

[4] M. Felleisen. Calculi of lambda-nu-cs conversion: a syntactic theory of control and state in imperative higher-order programming languages. Technical report, Indiana Univ., Bloomington (USA), 1987.

[5] P. Haller. Scala actors: A short tutorial, 2008.

[6] N. D. Jones. *Flow analysis of lambda expressions*. Springer, 1981.

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

[8] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *ACM Sigplan Notices*, volume 45, pages 428–443. ACM, 2010.

[9] J. Midtgaard and T. Jensen. A calculational approach to control-flow analysis by abstract interpretation. In *Static Analysis*, pages 347–362. Springer, 2008.

[10] J. Midtgaard and T. P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. *ACM Sigplan Notices*, 44(9):287–298, 2009.

[11] M. Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, June 2007.

[12] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. Eigencfa: accelerating flow analysis with GPUs. *ACM SIGPLAN Notices*, 46(1):511–522, 2011.

[13] O. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Citeseer, 1991.