Partitioning 0-CFA for the GPU

Thomas Gilray, James King, Matthew Might

University of Utah
{tgilray, jsking2, might}@cs.utah.edu

Abstract. We generalize EigenCFA, a linear formulation of 0-CFA, to features needed for real-world intermediate languages and apply our approach to an analysis of Scheme. EigenCFA lacks the full precision of 0-CFA and is restricted to programs in the subset of pure λ-calculus conforming to binary continuation-passing-style. This constraint is borne of the need to encode a monolithic transfer function as a GPU kernel free of thread-divergence. We show the soundness and precision of partitioning this transfer function to obtain an encoding with both the fine-grained parallelism of matrix operations as well as coarse-grained parallelism between transfer functions. Our approach supports real-world languages with a diverse array of forms and has the full precision of a traditional 0-CFA. An implementation effort is in progress and preliminary results are promising. Both CPU and GPU versions of our encoding have been tested for correctness against a standard worklist implementation.

1 Introduction

The aim of static analysis is to make guarantees about the behavior of a program before runtime. For example, if we wanted to know the possible types for a variable in a dynamic language (type-recovery), or if we wanted to know where sensitive data might exit a program (taint-propagation), we are asking for a data-flow analysis. If we wanted to remove code which can never be run (dead-code elimination) or determine which code is likely to be run most frequently (static profiling), we are asking for a control-flow analysis of our program. The goal of these analyses is to give a computable approximation of the propagation of data or control through a program which is precise enough to answer useful questions for the purposes of compiler optimization, bug-finding, or security, among other applications.

A flow-analysis allows us to answer a question like: "What are the possible return values of the following Scheme snippet?"

```
(let ([add5 (lambda (a) (+ a 5))])
  (add5 10))
```

In this particular case, the code is guaranteed to halt with a specific answer, and so it may simply be executed. In the general case however, static analysis brings us up against the halting problem and fundamental limitations on computability. Rice's theorem shows that proving any non-trivial program property is incomputable in the general case. The challenge of designing a static analysis is therefore the unavoidable trade off between precision and scalability. Effectively reformulating a flow-analysis so that parallelism may be easily exploited allows solutions to be found using throughput-oriented hardware and benefits the scalability of an analysis at no cost to precision.

EigenCFA is a preliminary attempt at doing just that [13]. It derives a rather clever encoding of 0-CFA (a specific control-flow analysis) as linear-algebraic operations which are efficient to compute on SIMD architectures like the GPU. Unfortunately, it is restricted by its need to encode the entire analysis as a single GPU kernel. This restriction has made it impossible to encode an analysis of simple language features like primitive operations and conditions. The `+` operation in our code snippet places even such a simple example outside the ability of EigenCFA to handle efficiently. Our approach solves this problem with a technique we call *transfer function partitioning*, allowing the essential encoding to be extended to real-world analyses. As 0-CFA is nearly cubic in complexity and impractical to compute for large programs, such parallelism is of vital importance in bringing sophisticated program-analysis to bear on everyday problems.

In addition, EigenCFA is less precise than it should be as it uses the trivially sound approximation for control-flow behavior. Consider the following sample taken from a larger program where `f`, `g`, and `h` are unreachable functions.

```
(define (f x) (h x 0))
(define (g) (h 0 0))
```

Proving that these are in-fact unreachable requires a precise control-flow analysis. EigenCFA models only data-flow and assumes the reachability of all expressions in a program. Because of this, the callsites `(h 0 0)` and `(h x 0)` will be examined and their data-flows propagated. As `x` is unbound, even after handling propagations for `(h x 0)`, the corresponding formal parameter for h will remain unbound. An unbound variable is an implicit indication that a function (in this case `h`) is in-fact unreachable. In this case, an approximation of data-flow has resulted in a bound on control-flow, but this cannot be relied on to give the same precision as a traditional 0-CFA which models control explicitly. Because the callsite in g applies h on two constant values, once the analysis is complete, it will appear that h may be reachable as possible values will have been found for both its parameters. Our linear encoding solves this problem by explicitly modeling both the control-flow and data-flow aspects of a program.

In the following sections, we will give the necessary background on static analysis by abstract interpretation, introduce a concrete and abstract semantics for a Scheme intermediate-representation, and derive a linear encoding for 0-CFA. Our encoding has been implemented both as a single-threaded CPU version and as single-stream and multi-stream GPU versions. All these have been tested for correctness against a standard worklist implementation and produce identical results for a suite of Scheme benchmarks. While the optimization effort is ongoing, preliminary results are promising, showing potential speedups of 20x or better over a single-threaded version of the encoding.

2 Background

Abstract interpretation is a very general framework for static analysis which allows us to perform an approximate evaluation of a program [2] [3]. The result is a single, time-bounded execution which represents the set of all possible exact executions. The approximation used, formally known as a Galois-connection, is a precisely defined relationship between a concrete semantics and an abstract semantics. In the code snippet on the front page for example, approximating concrete integers as `INT` could instantiate the framework to perform a type-recovery and we would determine the result to be at least some integer. If instead they were approximated as one of `POS`, `ZERO`, or `NEG`, we could perform a sign-analysis and discover that the result can only be a positive integer more specifically.

While there are a number of common formulations, our presentation uses a small-step operational semantics defined over the configurations of an abstract-machine. Such a semantics is determined by a series of inference rules which define, given a machine configuration, what configurations may immediately succeed it. A static analysis in this style proceeds by evaluation of an abstract abstract-machine starting with an initial configuration and exploring all abstract states reachable by the semantic rules. The simulation may be called *sound* if the defined Galois-connection strictly bounds the concrete executions represented by an abstract execution (for example, over-approximation or under-approximation). This property is proved inductively by showing that soundness before state-transition implies soundness after state-transition in all cases [10].

In Shivers' seminal work on control-flow analysis (CFA) of higher-order languages, he introduced a hierarchy of increasingly precise analyses for Scheme known as k-CFA [14]. The most basic of these, 0-CFA, has been extended and improved in a variety of ways since then and may be considered the foundational analysis of functional languages by abstract interpretation. For $k > 0$ the analysis is known to be EXPTIME-complete [5] and therefore intractable, but 0-CFA and many of its related algorithms are merely of a large polynomial complexity. Specifically, 0-CFA is in $O(\frac{n^3}{log(n)})$, but remains stubbornly difficult to compute in practice.

2.1 GPU programming

GPUs have become increasingly popular in recent years for solving computationally intense problems outside of graphics processing [11]. The FLOP throughput on GPUs has continued to increase exponentially, significantly outpacing CPUs. Many high performance computers now rely on the GPU as a work-horse for the computationally demanding linear-algebra which is often needed in large-scale scientific and engineering applications. GPUs achieve this performance due to the use of a streaming SIMD (single-instruction/multiple-data) architecture. This allows a group of small lightweight cores to perform the same operation on a vector of data and is ideal for certain tasks such as graphics and linear-algebra. With the addition of soldered memory, GPUs are also able to attain a significantly higher memory bandwidth than that of traditional CPUs.

Modern GPUs have thousands of cores and can achieve parallelism on multiple levels. The first is fine-grained SIMD parallelism which relies on many threads operating in lock-step. The second is parallelism from concurrent program execution. *Thread-divergence* occurs when a branch instruction separates a group of threads into different control-flow paths. This causes threads on one path to continue while others wait to run sequentially and is disastrous for performance.

2.2 EigenCFA

EigenCFA is a linear encoding of 0-CFA [13] and the primary inspiration for our approach. This formulation of 0-CFA allows the flow-analysis to be computed on a GPU; however, it is constrained to a very simple language without support for fundamental and important features like mutation and conditions. This extreme simplicity is required so the entire analysis can be implemented as a single GPU kernel free of thread-divergence. Prior to analysis, programs must be reduced to a subset of the pure λ-calculus conforming to binary continuation-passing-style (binary CPS). This restricts programs to values ranging over closures that accept precisely two arguments and never return. CPS itself is a very practical intermediate-representation but currying all functions may not be desirable, and church-encoding basic values is problematic as it obfuscates the data under analysis. To make matters worse, this encoding requires a variety of standard language forms like `letrec`, `set!`, and `if`, to be desugared into direct lambda application in such a way that the program's behavior according to 0-CFA is unaffected, even if its concrete execution is no longer sound. This technique, termed *abstract church encoding*, is another highly destructive transformation which makes it difficult to map the results of an analysis back onto the original program. A final shortcoming is a lack of precision in modeling control-flow behavior. 0-CFA necessarily models both control-flow and data-flow as the nature of higher-order programming languages entangles these two concerns. EigenCFA however, seems ironically named as it assumes the reachability of all callsites in the program, giving the trivially sound result for its control-flow approximation.

2.3 Points-to analysis

The other recent attempt to bring flow-analysis to the GPU implements an inclusion-based points-to analysis [9]. This formulation operates on the adjacency matrix of a points-to graph and is manageable for some real-world language features, but lacks generality. Like EigenCFA, the strategy is constrained to monovariance and supports a restricted language. Unlike EigenCFA, this approach is not readily extensible to analysis of higher-order functions and their environments, or to more general program analyses with richer abstract domains.

3 Concrete Semantics for CPS Scheme

The target language for our analysis is a desugared Scheme in continuation-passing-style. CPS constrains function calls never to return; instead, a caller must explicitly pass a continuation forward to be invoked on the result [12]. CPS is an excellent and widely used language for compiler optimization and program analysis [1]. If the transformation to CPS makes note of which lambdas correspond to continuations, the simplified program may again, along with any optimizations and analysis results, be precisely reconstituted in its direct-style form. This means the advantages of CPS can be utilized without compromise or loss of information [8].

The grammar for this language structurally distinguishes between atomic-expressions *ae* and complex-expressions *e*.

$$
\begin{aligned}
e \in \mathsf{E} &::= \texttt{(}ae\ ae\ \ldots\texttt{)}^{l} \\
&\mid \texttt{(}\text{set! } x\ ae\ ae\texttt{)}^{l} \\
&\mid \texttt{(}\text{prim } op\ ae\ \ldots\texttt{)}^{l} \\
&\mid \texttt{(}\text{if } ae\ e\ e\texttt{)}^{l} \\
&\mid \texttt{(}\text{halt}\texttt{)}^{l} \\
ae \in \mathsf{AE} &::= c \mid x \mid lam \\
lam \in \mathsf{Lam} &::= \texttt{(}\lambda\ \texttt{(}x\ \ldots\texttt{)}\ e\texttt{)} \\
c \in \mathsf{Const} &::= \texttt{\#t} \mid \texttt{\#f} \mid \langle number \rangle \mid \ldots \\
x \in \mathsf{Var} &::= \langle set\ of\ program\ variables \rangle \\
op \in \mathsf{OP} &::= \langle set\ of\ primitive\ operations \rangle \\
l \in \mathsf{Label} &::= \langle set\ of\ unique\ labels \rangle
\end{aligned}
$$

To specify the behavior of this language, we define a small-step operational semantics for an abstract-machine. A transition relation ($\Rightarrow$) is needed which defines at most one successor for any valid machine state ς. We use a CES-style machine with control-expression e, binding-environment ρ, value-store σ, and timestamp t (execution context) components [4]. The binding-environment ρ maps variables in scope to an address. The value-store σ maps addresses a to values v. Timestamps are unbounded lists of labels representing a complete history of program execution.

$$
\begin{aligned}
\varsigma \in \Sigma &= \mathsf{E} \times Env \times Store \times Time \\
\rho \in Env &= \mathsf{Var} \rightharpoonup Addr \\
\sigma \in Store &= Addr \rightharpoonup Value \\
t \in Time &= \mathsf{Label}^{*} \\
a \in Addr &= \mathsf{Var} \times Time \\
v \in Value &= \mathsf{Lam} \times Env + \mathsf{Const}
\end{aligned}
$$

For evaluating atomic-expressions we define an auxiliary function $\mathcal{A}$ which maps a syntactic *ae* in the context of a current state to a semantic program value. In the case of a variable, $\mathcal{A}$ uses ρ and σ to lookup the variable's current binding in the store. In the case of a lambda, a value is produced by closure-conversion: pairing the lambda with its current environment.

$$\begin{aligned}
\mathcal{A}\colon \mathsf{AE} \times \Sigma &\rightharpoonup \mathit{Value} \\
\mathcal{A}(x,\ (e,\ \rho,\ \sigma,\ t)) &= \sigma(\rho(x)) \\
\mathcal{A}(lam,\ (e,\ \rho,\ \sigma,\ t)) &= (lam,\ \rho) \\
\mathcal{A}(c,\ \varsigma) &= c
\end{aligned}$$

In addition, we need a primitive-operation evaluator δ which maps an operation *op* and list of values v to a result.

$$\delta\colon \mathsf{OP} \times \mathit{Value}^* \rightharpoonup \mathit{Value}$$

We may now define the small-step transition relation ($\Rightarrow$) by pattern-matching against the four complex-expressions that need to be handled: conditionals, mutation, primitive operations, and callsites. The (halt) expression does not need to be handled as it has no successors.

An inference rule like the one below asserts that the *conclusion* below the line is true whenever the *premise* above is true. When a callsite is reached, control moves inside the body e of the invoked closure.

$$\frac{(\texttt{(}\lambda\ \texttt{(}x_1\ \ldots\ x_j\texttt{)}\ e\texttt{)},\ \rho_\lambda) = \mathcal{A}(ae_f,\ \varsigma)}{\underbrace{(\texttt{(}ae_f\ ae_1\ \ldots\ ae_j\texttt{)}^l,\ \rho,\ \sigma,\ t)}_{\varsigma} \Rightarrow (e,\ \rho',\ \sigma',\ t')}$$

$$\begin{aligned}
\mathit{where}\quad \rho' &= \rho_\lambda[x_i \mapsto (x_i,\ t')] \\
\sigma' &= \sigma[(x_i,\ t') \mapsto \mathcal{A}(ae_i,\ \varsigma)] \\
t' &= l : t
\end{aligned}$$

A new environment ρ' is produced from the closure's ρ_λ augmented with bindings for each formal parameter x_i. An updated store is produced by mapping these addresses for each x_i to the value indicated by atomic-evaluation of ae_i. As the timestamp t' is always a complete and unique history of program execution, all transitions use a fresh set of addresses.

Mutation is handled similarly as it implies the invocation of a continuation. The continuation indicated for ae_k receives `VOID`, the return value of a `set!` expression in Scheme. In addition, we update the current address in scope for x to be the current value for ae_v.

$$\frac{(\texttt{(}\lambda\ \texttt{(}x_k\texttt{)}\ e\texttt{)},\ \rho_\lambda) = \mathcal{A}(ae_k,\ \varsigma)}{\underbrace{(\texttt{(}\text{set!}\ x\ ae_v\ ae_k\texttt{)}^l,\ \rho,\ \sigma,\ t)}_{\varsigma} \Rightarrow (e,\ \rho',\ \sigma',\ t')}$$

$$\begin{aligned}
\mathit{where}\quad \rho' &= \rho_\lambda[x_k \mapsto (x_k,\ t')] \\
\sigma' &= \sigma[(x_k,\ t') \mapsto \texttt{VOID}] \\
&\qquad [\rho(x) \mapsto \mathcal{A}(ae_v,\ \varsigma)] \\
t' &= l : t
\end{aligned}$$

Primitive operations use δ to obtain a return value and propagate this v_k to the prim-op's continuation.

$$\frac{(\texttt{(}\lambda\ \texttt{(}x_k\texttt{)}\ e\texttt{)},\ \rho_\lambda) = \mathcal{A}(ae_k,\ \varsigma)}{\underbrace{(\texttt{((}\text{prim}\ op\ ae_1\ \ldots\ ae_j\ ae_k\texttt{)}^l,\ \rho,\ \sigma,\ t)}_{\varsigma} \Rightarrow (e,\ \rho',\ \sigma',\ t')}$$

$$\begin{aligned} \textit{where}\quad & \rho' = \rho_\lambda[x_k \mapsto (x_k,\ t')] \\ & \sigma' = \sigma[(x_k,\ t') \mapsto v_k] \\ & v_k = \delta(op,\ (\mathcal{A}(ae_1,\ \varsigma)\ \ldots\ \mathcal{A}(ae_j,\ \varsigma))) \\ & t' = l : t \end{aligned}$$

Conditionals are probably the simplest case, with control moving inside the true branch or false branch as appropriate.

$$\frac{\mathcal{A}(ae,\ \varsigma) = \texttt{FALSE}}{\underbrace{(\texttt{((}\text{if}\ ae\ e_t\ e_f\texttt{)},\ \rho,\ \sigma,\ t)}_{\varsigma} \Rightarrow (e_t,\ \rho,\ \sigma,\ t)}$$

$$\frac{\mathcal{A}(ae,\ \varsigma) \neq \texttt{FALSE}}{\underbrace{(\texttt{((}\text{if}\ ae\ e_t\ e_f\texttt{)},\ \rho,\ \sigma,\ t)}_{\varsigma} \Rightarrow (e_f,\ \rho,\ \sigma,\ t)}$$

3.1 Evaluating a program

To evaluate a program e with these concrete semantics we produce a starting configuration $\varsigma_0 = \mathcal{I}(e)$ using a concrete state-space injection function $\mathcal{I} \colon \mathsf{E} \to \Sigma$:

$$\mathcal{I}(e) = (e,\ \bot,\ \bot,\ ())$$

We can then compute the transitive closure of $(\Rightarrow)$ starting from ς_0. As our state-space is unbounded, and the interpretation may continue to produce new states indefinitely, concrete executions are incomputable in the general case.

4 Abstract Semantics for 0-CFA

We perform a structural abstraction bounding the machine's address-space to obtain a computable approximation of our concrete semantics [6][7]. Notice that our abstract semantics contains several fundamental changes from its concrete counterpart. 0-CFA bounds the address-space to include exactly one address for each variable (monovariance). All values bound to a variable x in any context therefore must be represented by a single address. This introduces merging between values in our store and non-determinism in the transition relation.

To define an abstract operational semantics, we again need an abstract machine and a transition relation $(\approx\!>)$ which matches up successors and predecessors within the machine's configuration-space. As we are effectively re-using an empty timestamp for every allocation, expressions will uniquely identify an environment mapping free-variables to themselves and the store may directly map variables $\hat{x}$ to sets of abstract values $\hat{v}$. Such a *flow-set* may indicate a range of possible concrete values for an address. Closures are now just lambdas.

$$\begin{aligned}
\hat{\varsigma} \in \widehat{\Sigma} &= \mathsf{E} \times \widehat{Store} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Var} \rightarrow \widehat{Values} \\
\hat{v} \in \widehat{Values} &= \mathcal{P}(\widehat{Value}) \\
\hat{d} \in \widehat{Value} &= \mathsf{Lam} + \widehat{Basic} \\
\widehat{Basic} &= \{\texttt{TRUE},\ \texttt{FALSE},\ \texttt{VOID},\ \texttt{INT},\ \ldots\}
\end{aligned}$$

Program constants map to their corresponding basic values. When performing a concrete interpretation, these values are precise. When performing our abstract interpretation, there should only be a finite number of abstract basic values so they can be enumerated in our forthcoming encoding. For constant propagation, a set of program locations may be used. We use the notation α below to informally indicate the abstraction function a fully defined Galois-connection would employ to map a concrete machine component to its most precise abstract representative. For example, $\alpha(-3)$ could yield `NEG`.

The abstract atomic-expression evaluator returns flow-sets $\hat{v}$.

$$\begin{aligned}
\hat{\mathcal{A}} \colon \mathsf{AE} \times \widehat{\Sigma} &\rightharpoonup \widehat{Values} \\
\hat{\mathcal{A}}(x,\ (e,\ \hat{\sigma})) &= \hat{\sigma}(x) \\
\hat{\mathcal{A}}(lam,\ (e,\ \hat{\sigma})) &= \{lam\} \\
\hat{\mathcal{A}}(c,\ \hat{\varsigma}) &= \{\alpha(c)\}
\end{aligned}$$

We also need an abstract prim-op evaluator $\hat{\delta}$ which maps a primitive operation op and list of flow-sets to a sound result. For example, $\hat{\delta}(\texttt{+},\ (\{\texttt{POS}\},\ \{\texttt{POS}\})) = \{\texttt{POS}\}$.

$$\hat{\delta} \colon \mathsf{OP} \times \widehat{Values}^{*} \rightharpoonup \widehat{Values}$$

A callsite has one successor for each closure that accepts a matching number of arguments indicated by the flow-set for ae_f (the atomic-expression in call position).

$$\frac{\texttt{(}\lambda\ \texttt{(}x_1\ \ldots\ x_j\texttt{)}\ e\texttt{)} \in \hat{\mathcal{A}}(ae_f,\ \hat{\varsigma})}{\underbrace{\texttt{((}ae_f\ ae_1\ \ldots\ ae_j\texttt{)},\ \hat{\sigma})}_{\hat{\varsigma}} \approx\!> (e,\ \hat{\sigma}')}$$

$$where \quad \hat{\sigma}' = \hat{\sigma} \sqcup [x_i \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma})]$$

Control moves inside the body of all invoked closures e. The updated store is now conservatively approximated by finding the least-upper-bound of the current store and each new binding. Stores are ordered point-wise by inclusion, i.e. $(\hat{\sigma}_1 \sqcup \hat{\sigma}_2)(\hat{a}) = \hat{\sigma}_1(\hat{a}) \cup \hat{\sigma}_2(\hat{a})$.

Mutation is succeeded by a state for each possible continuation.

$$\frac{(\lambda\ (x_k)\ e) \in \hat{\mathcal{A}}(ae_k,\ \hat{\varsigma})}{\underbrace{(\text{(set! } x\ ae_v\ ae_k\text{)},\ \hat{\sigma})}_{\hat{\varsigma}} \approx> (e,\ \hat{\sigma}')}$$

$$\begin{aligned} \textit{where} \quad \hat{\sigma}' &= \hat{\sigma} \sqcup [x_k \mapsto \{\texttt{VOID}\}] \\ &\quad \sqcup [x \mapsto \hat{\mathcal{A}}(ae_v,\ \hat{\varsigma})] \end{aligned}$$

In addition, to conservatively simulate mutation of the variable x, all flows indicated for ae_v are included along with all previous values.

$$\frac{(\lambda\ (x_k)\ e) \in \hat{\mathcal{A}}(ae_k,\ \hat{\varsigma})}{\underbrace{(\text{(prim } op\ ae_1\ \ldots\ ae_j\ ae_k\text{)},\ \hat{\sigma})}_{\hat{\varsigma}} \approx> (e,\ \hat{\sigma}')}$$

$$\begin{aligned} \textit{where} \quad \hat{\sigma}' &= \hat{\sigma} \sqcup [x_k \mapsto \hat{v}_k] \\ \hat{v}_k &= \delta(op,\ (\hat{\mathcal{A}}(ae_1,\ \hat{\varsigma})\ \ldots\ \hat{\mathcal{A}}(ae_j,\ \hat{\varsigma}))) \end{aligned}$$

Primitive operations use $\hat{\delta}$ to obtain an approximation of the return value and propagate this flow-set $\hat{v}_k$ to each continuation indicated for ae_k.

$$\frac{\hat{v} \in \hat{\mathcal{A}}(ae,\ \hat{\varsigma}) \qquad \hat{v} \neq \texttt{FALSE}}{\underbrace{(\text{(if } ae\ e_t\ e_f\text{)},\ \hat{\sigma})}_{\hat{\varsigma}} \approx> (e_t,\ \hat{\sigma})}$$

$$\frac{\hat{v} \in \hat{\mathcal{A}}(ae,\ \hat{\varsigma}) \qquad \hat{v} = \texttt{FALSE}}{\underbrace{(\text{(if } ae\ e_t\ e_f\text{)},\ \hat{\sigma})}_{\hat{\varsigma}} \approx> (e_f,\ \hat{\sigma})}$$

When a conditional is reached, both branches may be taken.

4.1 Naïvely computing the analysis

We first define an injection function $\hat{\mathcal{I}}$ which, given a program e, determines an initial state $\hat{\varsigma}_0 = \hat{\mathcal{I}}(e)$.

$$\begin{aligned} &\hat{\mathcal{I}} \colon \mathsf{E} \to \widehat{\Sigma} \\ &\hat{\mathcal{I}}(e) = (e,\ \bot) \end{aligned}$$

To compute our analysis, we can simply visit all states reachable from $\hat{\varsigma}_0$. We define a transfer function for the system-space of our program $\hat{f} \colon \mathcal{P}(\widehat{\Sigma}) \to \mathcal{P}(\widehat{\Sigma})$:

$$\hat{f}(\hat{S}) = \{\hat{\varsigma}'\ :\ \hat{\varsigma} \in \hat{S} \text{ and } \hat{\varsigma} \approx> \hat{\varsigma}'\} \cup \{\hat{\varsigma}_0\}$$

Unfortunately, this approach is impracticable as the total number of stores is exponential in the size of the program, even for this context-insensitive analysis.

4.2 Efficiently computing the analysis

A more efficient method uses a single store to replace the multitude of individual stores. This global store is maintained as the least-upper-bound of all stores seen so far. Global-store-widening is a sound, and in practice quite reasonable, approximation of the naïve calculation [10] [14]. With this form of widening applied, 0-CFA is in $O(\frac{n^3}{log(n)})$. We factor the store out of our state-space while retaining a set of reachable expressions denoted $\hat{r}$ as an explicit model of control-flow. EigenCFA compromises on precision by modeling only the store.

$$\hat{r} \in \widehat{Reach} = \mathcal{P}(\mathsf{E})$$
$$\hat{\xi} \in \hat{\Xi} = \widehat{Reach} \times \widehat{Store}$$

Over factored system-spaces $\hat{\Xi}$, the transfer function becomes:

$$\begin{aligned}
\hat{f} &: \hat{\Xi} \to \hat{\Xi} \\
\hat{f}(\hat{r},\ \hat{\sigma}) &= (\hat{r} \cup \hat{r}',\ \hat{\sigma}') \\
\textit{where} \quad \hat{S} &= \{\hat{\varsigma}' : e \in \hat{r} \text{ and } (e,\ \hat{\sigma}) \approx\!> \hat{\varsigma}'\} \\
\hat{r}' &= \{e : (e,\ \_) \in \hat{S}\} \\
\hat{\sigma}' &= \bigsqcup \{\hat{\sigma}'' : (\_,\ \hat{\sigma}'') \in \hat{S}\}
\end{aligned}$$

The notation $\_$ matches any value without binding it to a variable.

The store grows monotonically across transition, i.e. $(\_,\ \hat{\sigma}) \approx\!> (\_,\ \hat{\sigma}')$ implies $\hat{\sigma} \sqsubseteq \hat{\sigma}'$, so $\hat{f}$ grows monotonically over $\hat{\Xi}$. Because $\hat{\Xi}$ is finite and $\hat{f}$ is continuous, we know that the least-fix-point of $\hat{f}$ is $\hat{f}^n(\bot, \bot)$ for some finite n.

5 Partitioning the transfer function

A central insight to our work is that we can partition a transfer function by reachable state under evaluation. By grouping these individual transfer functions into GPU kernels according to like control-flow we can minimize thread-divergence in a SIMD implementation. An individual transfer function $\hat{f}_e$ handles only the propagation of flows caused directly by e:

$$\begin{aligned}
\hat{f}_e &: \hat{\Xi} \to \hat{\Xi} \\
\hat{f}_e(\hat{r},\ \hat{\sigma}) &= (\hat{r} \cup \hat{r}'_e,\ \hat{\sigma}'_e) \\
\textit{where} \quad \hat{S}_e &= \{\hat{\varsigma}' : e \in \hat{r} \text{ and } (e,\ \hat{\sigma}) \approx\!> \hat{\varsigma}'\} \\
\hat{r}'_e &= \{e'' :\ (e'',\ \_) \in \hat{S}_e\} \\
\hat{\sigma}'_e &= \bigsqcup \{\hat{\sigma}'' : (\_,\ \hat{\sigma}'') \in \hat{S}_e\}
\end{aligned}$$

To determine the correctness and precision of this technique, we show its equivalence to an unpartitioned transfer function so we may exploit the corollary that a solution $\hat{\xi}$ which is simultaneously a fix-point for all $\hat{f}_e$ is guaranteed to be a fix-point for $\hat{f}$.

Theorem 1 (Transfer Partitioning).

$$\hat{f}(\hat{r},\ \hat{\sigma}) = \bigsqcup_{e \in \hat{r}} \hat{f}_e(\hat{r},\ \hat{\sigma})$$

Proof. (Sketch) Follows from the observation that $\hat{S}_e$ for all $e \in \hat{r}$ is a collection of covering subsets for $\hat{S}$.

$$\hat{S} = \bigcup_{e \in \hat{r}} \hat{S}_e$$

Therefore $\hat{r}'$ is also the least-join of all $\hat{r}'_e$ as is $\hat{\sigma}'$ of all $\hat{\sigma}'_e$. □

6 Linear Encoding for 0-CFA

Now that we may arbitrarily partition a transfer function to minimize thread-divergence, a linear encoding for handling a callsite can be defined separately from a linear encoding that handles a conditional, a primitive operation, or another form. The goal now is to produce an implementation for each f_e defined exclusively in terms of matrix multiplication ($\times$), outer product ($\otimes$), element-wise boolean-or ($+$), and dot product ($\cdot$).

For any finite domain, we can assign a canonical order to its contents and represent elements of its set or power-set as boolean vectors. Where vectors contain a single entry, they represent a single element in the set they encode, and where they contain more than one entry, the representation naturally extends to encoding more than one element at once. For example, as defined below, a value $\boldsymbol{v} \in \boldsymbol{V}$ is a vector representing a flow-set of abstract values. In the case of $\boldsymbol{S}$, we use $\boldsymbol{r}$ in all cases to denote a set of states and $\boldsymbol{s}$ to denote a particular state (i.e. a vector with a single entry).

$$\boldsymbol{r}, \boldsymbol{s} \in \boldsymbol{S} = \{0,1\}^{|\mathsf{E}|}$$
$$\boldsymbol{a} \in \boldsymbol{A} = \{0,1\}^{|\widehat{Var}|+|\widehat{Value}|}$$
$$\boldsymbol{v} \in \boldsymbol{V} = \{0,1\}^{|\widehat{Value}|}$$

Vectors $\boldsymbol{a}$ represent atomic-expressions, either variables or values. This is a design choice taken directly from EigenCFA which allows the various cases required for $\hat{\mathcal{A}}$ to be implemented as a single multiplication.

A function g over these vectors can be encoded as multiplication with a matrix, and may handle inputs which encode a set so long as the property $g(x \cup y) = g(x) \cup g(y)$ holds for all x and y. The store is such a function, one which maps variables to a flow-set of values, and values to themselves:

$$\boldsymbol{\sigma} \colon \boldsymbol{A} \to \boldsymbol{V}$$

If values are ordered after variables in $\boldsymbol{A}$, the bottom of the store will always be an identity matrix. Below is an example of a lookup showing how the store is used to map a variable to its flow-set via matrix multiplication. We use a CPS version of our original snippet of Scheme code for clarity.

```
((lambda (add5^{x_0})
   (add5^{x_0} 10^{\hat{d}_3} (lambda (result^{x_3}) (halt)^{l_3})^{\hat{d}_2})^{l_1})^{\hat{d}_0}
 (lambda (a^{x_1} add5k^{x_2})
   (prim + a^{x_1} 5^{\hat{d}_3} add5k^{x_2})^{l_2})^{\hat{d}_1})^{l_0}
```

Annotations show an assignment of labels to expressions, variables to vectors in $\boldsymbol{A}$, and abstract values to vectors in $\boldsymbol{V}$. The abstract value $\hat{d}_3$ represents `INT`.

Example 1. $\langle\!\langle \texttt{a} \rangle\!\rangle \times \boldsymbol{\sigma} = \langle\!\langle \{\texttt{INT}\} \rangle\!\rangle$

$$\begin{array}{cccccccc} x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \\ \end{array}$$

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{array}{c|cccc} & \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \\ \hline x_0 & 0 & 1 & 0 & 0 \\ x_1 & 0 & 0 & 0 & 1 \\ x_2 & 0 & 0 & 1 & 0 \\ x_3 & 0 & 0 & 0 & 1 \\ \hline \hat{d}_0 & 1 & 0 & 0 & 0 \\ \hat{d}_1 & 0 & 1 & 0 & 0 \\ \hat{d}_2 & 0 & 0 & 1 & 0 \\ \hat{d}_3 & 0 & 0 & 0 & 1 \end{array} = \begin{array}{cccc} \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \\ 0 & 0 & 0 & 1 \end{array}$$

The notation $\langle\!\langle \cdot \rangle\!\rangle$ is used informally to denote the matrix representation of a given entity. Including an identity matrix in the store composes two mappings as one matrix so that all cases in $\hat{\mathcal{A}}$ may be handled together as a single multiplication.

A program's syntax tree can also be encoded as a series of matrices. For example, a matrix **Body** maps lambdas in $\boldsymbol{V}$ to their body expression in $\boldsymbol{S}$. The same can be done for the true and false branches of a conditional form.

$$\begin{aligned} \mathbf{Body}\colon \boldsymbol{V} \to \boldsymbol{S} \qquad & \mathbf{Fun}\colon \boldsymbol{S} \to \boldsymbol{A} \\ \mathbf{CondTrue}\colon \boldsymbol{S} \to \boldsymbol{S} \qquad & \mathbf{Arg_i}\colon \boldsymbol{S} \to \boldsymbol{A} \\ \mathbf{CondFalse}\colon \boldsymbol{S} \to \boldsymbol{S} \qquad & \mathbf{Var_i}\colon \boldsymbol{V} \to \boldsymbol{A} \end{aligned}$$

Fun maps callsites to the atomic-expression in call-position. If this is a variable, a value in the top portion of $\boldsymbol{A}$ will result, if it's a lambda or constant value, an entry in the lower portion of $\boldsymbol{A}$ results. $\mathbf{Arg_i}$ represents a similar encoding for argument i of a callsite. $\mathbf{Var_i}$ encodes formal parameter i of a lambda. For example, we can expect $\langle\!\langle \texttt{(}\lambda\ \texttt{(a add5k) \ldots)} \rangle\!\rangle \times \mathbf{Var_2}$ to yield a value $\langle\!\langle \texttt{add5k} \rangle\!\rangle$.

For a callsite $\boldsymbol{s}$, the value of its second argument can be computed as $\boldsymbol{v}_2 = \boldsymbol{s} \times \mathbf{Arg_2} \times \boldsymbol{\sigma}$ and the value of the applied lambda as $\boldsymbol{v}_f = \boldsymbol{s} \times \mathbf{Fun} \times \boldsymbol{\sigma}$. The second formal parameter for $\boldsymbol{v}_f$ may then be computed as $\boldsymbol{a}_2 = \boldsymbol{v}_f \times \mathbf{Var_2}$ and with these two values, the store can be updated with a binding to $\boldsymbol{v}_2$ for $\boldsymbol{a}_2$. This is accomplished by using the outer product $\boldsymbol{a}_2 \otimes \boldsymbol{v}_2$ as this will give a store-update matrix with an entry at index (m, n) whenever $\boldsymbol{a}_2$ has an entry at position m and $\boldsymbol{v}_2$ has one at n. An update is applied to the current store using element-wise boolean-or. The example below shows the store update produced for `add5k`.

Example 2. $\langle\!\langle$`add5k`$\rangle\!\rangle \otimes \langle\!\langle$`(lambda (result) (halt))`$\rangle\!\rangle$
$= \langle\!\langle$`[add5k` $\mapsto$ `(lambda (result) (halt))]`$\rangle\!\rangle$

$$\begin{array}{cccccccc} x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \\ \end{array}$$
$$\left[\begin{array}{cccccccc} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \otimes \begin{array}{c} \begin{array}{cccc} \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \end{array} \\ \left[\begin{array}{cccc} 0 & 0 & 1 & 0 \end{array}\right] \end{array} = \begin{array}{c|cccc} & \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \\ x_0 & 0 & 0 & 0 & 0 \\ x_1 & 0 & 0 & 0 & 0 \\ x_2 & 0 & 0 & 1 & 0 \\ x_3 & 0 & 0 & 0 & 0 \\ \hline \hat{d}_0 & 0 & 0 & 0 & 0 \\ \hat{d}_1 & 0 & 0 & 0 & 0 \\ \hat{d}_2 & 0 & 0 & 0 & 0 \\ \hat{d}_3 & 0 & 0 & 0 & 0 \end{array}$$

An operation $\boldsymbol{v}_f \times \mathbf{Body}$ finds the body for $\boldsymbol{v}_f$, and boolean-or is used to extend the vector of reachable expressions $\boldsymbol{r}$. A full encoding for f_e where e is a callsite of length j can now be defined in full using these operations:

$$\begin{aligned} f_{\boldsymbol{s}_{call_j}}(\boldsymbol{r},\ \boldsymbol{\sigma}) &= (\boldsymbol{r}',\ \boldsymbol{\sigma}') \\ \textit{where} \quad \boldsymbol{v}_f &= \boldsymbol{s}_{call_j} \times \mathbf{Fun} \times \boldsymbol{\sigma} \\ \boldsymbol{v}_i &= \boldsymbol{s}_{call_j} \times \mathbf{Arg_i} \times \boldsymbol{\sigma} \\ \boldsymbol{a}_i &= \boldsymbol{v}_f \times \mathbf{Var_i} \\ \boldsymbol{\sigma}' &= \boldsymbol{\sigma} + (\boldsymbol{a}_1 \otimes \boldsymbol{v}_1) + \ldots + (\boldsymbol{a}_j \otimes \boldsymbol{v}_j) \\ \boldsymbol{r}' &= \boldsymbol{r} + (\boldsymbol{v}_f \times \mathbf{Body}) \end{aligned}$$

To handle `set!` forms, we may reuse the matrix **Fun** for encoding the continuation, $\mathbf{Arg_1}$ for encoding the variable being set, and $\mathbf{Arg_2}$ for encoding the atomic-expression it's being assigned to. The continuation receives a value $\langle\!\langle$`VOID`$\rangle\!\rangle$ we'll denote as $\overrightarrow{void}$:

$$\begin{aligned} f_{\boldsymbol{s}_{set!}}(\boldsymbol{r},\ \boldsymbol{\sigma}) &= (\boldsymbol{r}',\ \boldsymbol{\sigma}') \\ \textit{where} \quad \boldsymbol{v}_f &= \boldsymbol{s}_{set!} \times \mathbf{Fun} \times \boldsymbol{\sigma} \\ \boldsymbol{a}_{var} &= \boldsymbol{v}_f \times \mathbf{Var_1} \\ \boldsymbol{a}_{set} &= \boldsymbol{s}_{set!} \times \mathbf{Arg_1} \\ \boldsymbol{v}_{set} &= \boldsymbol{s}_{set!} \times \mathbf{Arg_2} \times \boldsymbol{\sigma} \\ \boldsymbol{\sigma}' &= \boldsymbol{\sigma} + (\boldsymbol{a}_{var} \otimes \overrightarrow{void}) + (\boldsymbol{a}_{set} \otimes \boldsymbol{v}_{set}) \\ \boldsymbol{r}' &= \boldsymbol{r} + (\boldsymbol{v}_f \times \mathbf{Body}) \end{aligned}$$

Conditionals make no changes to the store, but extend reachability to the subexpression for the true or false branches as appropriate. $\overrightarrow{false}$ is used to denote $\langle\!\langle$`FALSE`$\rangle\!\rangle$ and $\overrightarrow{notfalse}$ to denote its inverse – which is notably not the same as $\langle\!\langle$`TRUE`$\rangle\!\rangle$. A dot product is used to obtain a boolean value which is false exactly when the intersection of two sets is empty:

$$\begin{aligned} f_{\boldsymbol{s}_{if}}(\boldsymbol{r},\ \boldsymbol{\sigma}) &= (\boldsymbol{r}',\ \boldsymbol{\sigma}) \\ \textit{where} \quad & \boldsymbol{v}_{cond} = \boldsymbol{s}_{if} \times \mathbf{Arg_1} \times \boldsymbol{\sigma} \\ & tb = \boldsymbol{v}_{cond} \cdot \overrightarrow{notfalse} \\ & fb = \boldsymbol{v}_{cond} \cdot \overrightarrow{false} \\ & \boldsymbol{r}' = \boldsymbol{r} + tb(\boldsymbol{s}_{if} \times \mathbf{CondTrue}) + fb(\boldsymbol{s}_{if} \times \mathbf{CondFalse}) \end{aligned}$$

6.1 A final algorithm

To find a solution, we can iterate to a fix-point $(\boldsymbol{r},\ \boldsymbol{\sigma})$ over all $f_{\boldsymbol{s}}$ where $\boldsymbol{s}$ is drawn from the entries of $\boldsymbol{r}$. In practice, we may exploit both the fine-grain parallelism of matrix operations and the coarse-grain parallelism of running each $f_{\boldsymbol{s}}$ concurrently. As each individual $f_{\boldsymbol{s}}$ is monotonic and continuous, our reasoning on termination and precision from section 4.2 remains applicable.

```
while $\boldsymbol{\sigma}$ or $\boldsymbol{r}$ changes do
    foreach $\boldsymbol{s}$ in $\boldsymbol{r}$ do
        $(\boldsymbol{r}, \boldsymbol{\sigma}) = f_{\boldsymbol{s}}(\boldsymbol{r}, \boldsymbol{\sigma})$
    end
end
```

7 Conclusion

This analysis represents a faithful formulation of 0-CFA for Scheme and goes beyond the capabilities of EigenCFA in two key regards. First, including a range of reachable expressions allows the encoding to represent a more precise approximation of control-flow behavior corresponding to a traditional worklist implementation of 0-CFA. By only updating a global store, EigenCFA uses a fully imprecise control-flow approximation unnecessarily. Second, a sound and precise partitioning of the transfer function has allowed a variety of very different inference rules to be used without the need to entangle their implementation within a single GPU kernel (introducing thread-divergence). This permits extending the essential approach beyond trivial analyses of trivial languages. The work of applying our technique to 0-CFA can be further expanded to parallelize more sophisticated analyses of languages with an even greater variety of forms.

Acknowledgments. The authors would like to thank William E. Byrd, Leif Andersen, and Steven Lyde for helpful feedback on a draft of this work. This material is partially based on research sponsored by DARPA under agreement number FA8750-12-2-0106 and by NSF under CAREER grant 1350344. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon.

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, February 2007. ISBN 052103311X.

[2] 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 Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press, New York.

[3] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, TX, 1979. ACM Press, New York.

[4] Mattias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *Proceedings of the Symposium on Principles of Programming Languages*, page 314, New York, NY, 1987. ACM.

[5] David Van Horn and Harry G. Mairson. Deciding k-CFA is complete for EXPTIME. *ACM Sigplan Notices*, 43(9):275–282, 2008.

[6] David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceedings of the International Conference on Functional Programming*, September 2010.

[7] J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. Optimizing abstract abstract machines. In *Proceedings of the International Conference on Functional Programming*, September 2013.

[8] Andrew Kennedy. Compiling with continuations, continued. In *Proceedings of the International Conference on Functional Programming*, pages 177–190, New York, NY, 2007. ACM.

[9] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 107–116, New York, NY, 2012. ACM.

[10] Matthew Might. *Environment Analysis of Higher-Order Languages.* PhD thesis, Georgia Institute of Technology, Atlanta, GA, 2007.

[11] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, volume 26, pages 80–113, 2007.

[12] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. In *Theoretical Computer Science 1*, pages 125–159, 1975.

[13] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. EigenCFA: Accelerating flow analysis with GPUs. In *Proceedings of the Symposium on the Principals of Programming Languages*, pages 511–522, January 2010.

[14] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1988.