State Exploration Choices in a Small-Step Abstract Interpreter

Steven Lyde Matthew Might
University of Utah
{lyde,might}@cs.utah.edu

Abstract

When generating the abstract transition graph while computing k-CFA, the order in which we generate successor states is not important. However, if we are using store widening, the order in which we generate successor states matters because some states will help us jump to the minimum fixed point faster than others. The order in which states are explored is controlled by the work list. The states can be explored in depth-first or breadth-first fashion. However, these are not the only options available. We can also use a priority queue to intelligently explore states which will help us reach a fixed point faster than either of these two approaches. In this paper, we evaluate the different options that exist for a work list.

1. Introduction

Control-flow analysis of higher-order languages is hard, with the simplest implementation of the original formulation of k-CFA being cubic [8] and proven to be complete for polynomial time [9]. Faster implementations exist that are less precise. Henglein's simple closure analysis runs in almost linear time by using unification to solve constraints [5]. The analysis of Ashley and Dybvig achieves a better asymptotic bound by limiting the number of times we visit an expression in the analysis [2]. However, in this paper we will focus on the original implementation of k-CFA.

While control-flow analysis of higher-order languages is complex, the machinery underneath is actually quite simple. However, in these simple mathematics there are several nuances that can affect the precision of the analysis. In the past, the primary focus has been the allocation function, which controls the address of the variables we are binding [4]. With this seemingly simple function, the polyvariance, complexity, and precision of the analysis is controlled. However, this is not the only source of nuance in a small-step abstract framework.

In this paper, we will first quickly recall what a concrete small-step semantics looks like for lambda calculus in continuation-passing style. We will then proceed to demonstrate how this can easily be changed into an abstract interpreter with only a few small changes [10]. From there we will discuss how this abstract interpreter can be made to run quickly by using global widening in an algorithm known as the time-stamp algorithm [8].

Once an understanding of the time-stamp algorithm is attained, we can dive into the meat of this paper. It will be shown that it is important how exactly we handle the work list in the algorithm. The order in which we visit states and generate successor states matter.

The main contribution of this paper is to point out and demonstrate the idea that the order of exploration matters when iterating over the work list.

Our second contribution is to demonstrate that using a priority queue for the work list can increase the speed of the analysis and also decrease the amount of memory required for the analysis. We demonstrate with empirical evidence the efficacy of this idea, even though the gains might not be substantial.

2. Concrete Semantics

For this paper we will operate over a simple continuation-passing style lambda calculus.

$$\begin{aligned} v &\in \mathsf{Var} \text{ is a set of identifiers} \\ lam &\in \mathsf{Lam} ::= (\lambda\ (v_1 \dots v_n)\ call) \\ f, \text{æ} &\in \mathsf{AExp} ::= v \mid lam \\ call &\in \mathsf{Call} ::= (f\ \text{æ}_1 \dots \text{æ}_n) \end{aligned}$$

Unlike the pure lambda calculus, we allow lambda terms to have multiple arguments. We also only allow the body of a lambda term to be a function call and require that each sub-expression of a function call be either a variable or lambda term. This language form has been shown to be a suitable intermediate representation for compilers of higher-order languages [1]. It also has the benefit that its semantics can be described in a single transition relation.

We will now describe an abstract machine that can be used to evaluate a program. This machine will be very similar to the CESK machine of Felleisen [3]. Though it does not have a continuation component, because the continuations are explicit in the expressions. This is also slightly a non-standard state space because we have environments mapping to addresses rather than values. This is to facilitate the abstraction of the machine using the Abstracting Abstract Machines approach [10]. It also has a time component to facilitate allocating addresses. It is a list of all the call sites we have visited as we have executed the program.

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{Call} \times \mathit{Env} \times \mathit{Store} \times \mathit{Time} \\ \rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\ \sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Clo} \\ \mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env} \\ a \in \mathit{Addr} &= \mathsf{Var} \times \mathit{Time} \\ t \in \mathit{Time} &= \mathsf{Call}^* \end{aligned}$$

We have a transition relation that allows us to go from one state to the next $(\Rightarrow) \subseteq \Sigma \times \Sigma$.

$$\begin{aligned} \overbrace{(\llbracket(f\ \text{æ}_1 \dots \text{æ}_n)\rrbracket, \rho, \sigma, t)}^{\varsigma} &\Rightarrow (call, \rho'', \sigma', t'), \text{ where} \\ (\llbracket(\lambda\ (v_1 \dots v_n)\ call)\rrbracket, \rho') &= \mathcal{A}(f, \rho, \sigma) \\ \rho'' &= \rho'[v_i \mapsto a_i] \\ \sigma' &= \sigma[a_i \mapsto \mathcal{A}(\text{æ}_i, \rho, \sigma)] \\ t' &= tick(\varsigma) \\ a_i &= alloc(v_i, t') \end{aligned}$$

This transition relation relies on three auxiliary functions: one to evaluate atomic expressions, one to advance our time component, and one to allocate addresses.

The atomic evaluator $\mathcal{A} : \mathsf{AExp} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$ evaluates variables by looking up their address in the environment and then looking up the value of that address in the store. It evaluates lambda terms by closing over the current environment to create a closure.

$$\mathcal{A}(v, \rho, \sigma) = \sigma(\rho(v))$$
$$\mathcal{A}(lam, \rho, \sigma) = (lam, \rho)$$

To advance our time component we use $tick : \Sigma \to \mathit{Time}$ and which helps us keep track of all the call sites we have visited as we have executed our program. We prepend the current call expression to the existing time, thus creating a unique time-stamp for every state in the execution of our program.

$$tick(call, \rho, \sigma, t) = call : t$$

The allocation function $alloc : \mathsf{Var} \times \mathit{Time} \to \mathit{Addr}$ simply pairs the variable with the current time-stamp to generate a unique address.

$$alloc(v, t) = (v, t)$$

Given a program, we must be able to inject it into an initial state. Using $\mathcal{I} : \mathsf{Call} \to \Sigma$ we pair a program with an empty environment, empty store, and time-stamp with no elements.

$$\mathcal{I}(call) = (call, [], [], \langle\rangle)$$

Once we have our initial state, we can execute our program by generating successor states using our transition relation $(\Rightarrow) \subseteq \Sigma \times \Sigma$. We can simulate the halt continuation by having a free variable in our program. The transition relation will not have any closure bound to the free variable and thus cannot generate a successor state. Execution terminates when the halt continuation is applied. The meaning of the program is whatever value gets passed to the halt continuation.

3. Abstract Semantics

We will now explore how we can take this concrete semantics and make it abstract. Our abstract semantics will be guaranteed to terminate given any program. We begin by first abstracting our state space. Looking at the original concrete state space, the source of unboundedness is that our time-stamps can grow arbitrarily large. However, if we limit the length of our time-stamps to length k, our state space becomes finite. This is the crucial value of the parameter to k-CFA. Besides this small change, the abstract state space looks very similar to the concrete state space, with the notable exception that the time-stamps are now finite. Because time is finite, the number of addresses is also finite. This makes our abstract domain finite.

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Call} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Time}}$$
$$\hat{\rho} \in \widehat{\mathit{Env}} = \mathsf{Var} \rightharpoonup \mathit{Addr}$$
$$\hat{\sigma} \in \widehat{\mathit{Store}} = \widehat{\mathit{Addr}} \to \mathcal{P}\left(\widehat{\mathit{Clo}}\right)$$
$$\widehat{clo} \in \widehat{\mathit{Clo}} = \mathsf{Lam} \times \widehat{\mathit{Env}}$$
$$\hat{a} \in \widehat{\mathit{Addr}} = \mathsf{Var} \times \widehat{\mathit{Time}}$$
$$\hat{t} \in \widehat{\mathit{Time}} = \mathsf{Call}^k$$

However, having a finite set of addresses means that in our abstract interpretation some addresses will be reused. This means that our store must be able to handle having more than one value, so we now map to a set of closures rather than a single closure. These sets cannot grow arbitrarily large because there are only a finite number of closures. This is why the indirection of the store was introduced. Having environments point to values rather than addresses would introduce structural recursion, because values contain environments. However, with the introduction of the store the cycle is broken [10].

Our abstract transition relation $(\rightsquigarrow) \subseteq \hat{\Sigma} \times \hat{\Sigma}$ changes slightly from the concrete one in order to handle multiple closures. And we now join $(\sqcup)$ values in the store. This means we take the union of the sets of closures for the ones that previous existed at that address and the set of closures that we are adding and store that at the address.

$$\overbrace{(\llbracket (f\ \text{æ}_1 \ldots \text{æ}_n) \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{t})}^{\hat{\varsigma}} \rightsquigarrow (call, \hat{\rho}'', \hat{\sigma}', \hat{t}'), \text{ where}$$
$$(\llbracket (\lambda\ (v_1 \ldots v_n)\ call) \rrbracket, \hat{\rho}') \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$
$$\hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{a}_i]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(\text{æ}_i, \hat{\rho}, \hat{\sigma})]$$
$$\hat{t}' = \widehat{tick}(\hat{\varsigma})$$
$$\hat{a}_i = \widehat{alloc}(v_i, \hat{t}')$$

The auxiliary functions change slightly as well. The abstract atomic evaluator $\hat{\mathcal{A}} : \mathsf{AExp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \to \mathcal{P}\left(\widehat{\mathit{Clo}}\right)$ now returns a set of closures rather than just a single value.

$$\hat{\mathcal{A}}(v, \hat{\rho}, \hat{\sigma}) = \hat{\sigma}(\hat{\rho}(v))$$
$$\hat{\mathcal{A}}(lam, \hat{\rho}, \hat{\sigma}) = \{(lam, \hat{\rho})\}$$

The abstract allocation function $\widehat{alloc} : \mathsf{Var} \times \widehat{\mathit{Time}} \to \widehat{\mathit{Addr}}$ does not change except in its types from its concrete counterpart, because it is the time that we have limited.

$$\widehat{alloc}(v, t) = (v, t)$$

However, the function to advance out time-stamp $\widehat{tick} : \Sigma \to \widehat{\mathit{Time}}$ has changed from the concrete version in that it just take the last k call sites.

$$\widehat{tick}(call, \hat{\rho}, \hat{\sigma}, \hat{t}) = \overbrace{call : \hat{t}}^{\text{first } k \text{ values}}$$

We still need to inject our program into an initial abstract state $\hat{\mathcal{I}} : \mathsf{Call} \to \hat{\Sigma}$, but it is still paired with an empty environment, empty store, and empty time-stamp.

$$\hat{\mathcal{I}}(call) = (call, [], [], ())$$

To perform the analysis we must then compute all the reachable states using our abstract transition relation $(\rightsquigarrow) \subseteq \hat{\Sigma} \times \hat{\Sigma}$, generating successor states until a fixed point is reached.

$$\left\{\hat{\varsigma} : \hat{\mathcal{I}}(call) \rightsquigarrow^* \hat{\varsigma}\right\}$$

4. Implementing k-CFA

The simplest way to compute k-CFA is to construct the set of all reachable states over the transition relation, starting at the initial state. Any graph-searching algorithm is sufficient for finding this set. This will give us the desired result because every state in the concrete execution has an approximation in the set of abstract states generated by k-CFA. This means that any behavior that occurs in the concrete execution will be captured by the abstract execution.

Shivers devised two techniques for more quickly computing the set of reachable states: the aggressive-cutoff algorithm and the time-stamp algorithm [8].

We will give a short description of these two algorithms shortly and then will describe the algorithm in more detail.

4.1 The Aggressive-Cutoff Algorithm

While exploring the state space and generating the abstract transition relation, we only ever add information, we never take any away. We can exploit this monotonicity while exploring the state space. If a state we are about to explore is weaker than ($\sqsubseteq$) a state that we have already visited, we know that we have already captured the behavior of that state and do not need to generate its successor states again. This is the essence of the aggressive-cutoff algorithm.

4.2 The Time-Stamp Algorithm

The time-stamp algorithm is a form of the aggressive-cutoff algorithm. In the time-stamp algorithm, we modify the state-space search by joining the store of the state just pulled from the work list with the least upper bound of all the stores seen so far.

States contain a large environment and store that to compare requires a deep traversal. These states are sizable structures. To combat this issue we perform the following steps.

We keep around a single-threaded store that we update after each transition. The store grows monotonically, so this is safe to do. We might add additional values that would not occur in the concrete execution, but this is always sound. Whenever we update the store with a new value, we increment a time-stamp. Then in our states we no longer keep a reference to the store but to a time-stamp. A time-stamp with a lesser value is weaker than a time-stamp of a greater value. Thus we can do subsumption testing based on the value of the time-stamp. The larger time-stamp approximates the smaller time-stamp.

This technique implements the aggressive cutoff algorithm while at the same time lowering the storage overhead. The original implementation of the time-stamp algorithm [8] showed that it did not cost too much precision.

4.3 Detailed Algorithm

Putting together the two above techniques, exploiting configuration monotonicity for early termination and configuration-widening, leads to an algorithm for computing Shivers' original k-CFA.

This algorithm in Figure 1 is taken directly from Might, but adapted slightly to fit the notation of this paper [6].

Using a side-effected global table, $\hat{S}$, we map the latest evaluation context $(call, \hat{\rho}, \hat{t})$ to the latest generation of the store that has been explored with that context:

$$\hat{S} : \mathsf{Call} \times \widehat{Env} \times \widehat{Time} \to \mathbb{N}$$

During the search, if the current state was explored with a generation of the store that is greater than or equal to the current generation of the global store then that branch of the search has terminated. The monotonicity of the abstract transition relation guarantees that the behavior has already been approximated.

Otherwise, we widen the store of the state with the global store and generate successors, updating $\hat{S}$ to reflect that we have explored it with the current generation of the global store.

From the successor states, we see if they have contributed any changes to the global store. If they have, we widen the global store and bump its generation.

4.4 The Work List

Traditionally, when executing a work list algorithm, the order in which we explore states is not important. However, when we use the time-stamp algorithm, since each state can possibly contribute different values to the global store, the order does have an effect on the number of states that are explored. It has this effect because the quicker we can reach a fixed point of our global store, the quicker we can stop exploring states.

The work list is generally implemented using a list, with new states being appended to the front. This results in a depth-first search. However, we can explore these states in any order we wish. In the next section, we will discuss possible ordering schemes on this list, where we examine the contents of states in order try and guess which ones will help us reach the fixed point of the global store the quickest.

5. Priority Queue

There are four components to a state which we can use to guess if it will help us climb the lattice quicker: the expression, the environment, the store, and the time stamp.

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Time}$$

In addition to the properties of these components, we can also take advantage of temporal properties that arise during the execution of the abstract analysis.

We will now explore what properties of each of these components we could possibly use to help us order them in our work list. The abbreviations in the parenthesis are used in the evaluation section.

5.1 Expression

These are possible priority schemes based on the expression component of a state.

- The type of the expression. If our language was richer and allowed for more language forms such as `if` or `set!`, we could prioritize a given form over another (CTP).
- The number of subexpressions. It might be the case that more subexpressions means that more values will be bound, thus we should prioritize larger expressions over smaller ones (CSZ).
- Where the expression appears in the program. We could explore expressions that appear deeper in the program first (CDL) or we could take more of a breadth-first approach and try to visit expressions that appear higher in our program first (CBL).
- The number of times we have visited an expression. When we come across an expression in the course of the abstract interpretation we might want to prioritize states with expressions that we have already seen or vice versa (CFQ).
- Top level function or inner function. If the lambda term we are invoking originally was a top-level function in our program, it might be beneficial to explore inner functions before exploring other top-level functions.
- Prefer user lambdas over continuation lambdas. When converting to continuation-passing style, there are two types of lambda terms: user lambdas and continuation lambdas. Returns get converted into invocations of continuation lambdas (CCR).
- The size of the continuation. This might give a rough approximation of how much computation is left to do for a given state.

5.2 Environment

These are possible priority schemes based on the environment component of a state.

- The environment size. This is another way to give a comparable value to an expression. A larger environment might signify that we will bind more values (ESZ).
- The flow set size of every address in the environment. How big the flow sets are determine partially how big the flow sets are

$\hat{S} \leftarrow \bot$ — Seen time-stamps, $\mathsf{Call} \times \widehat{Env} \times \widehat{Time} \to \mathbb{N}$.
$\hat{\Sigma}_{\text{todo}} \leftarrow \left\{\hat{\mathcal{I}}(pr)\right\}$ — The work list.
$\hat{\sigma}^* \leftarrow \bot$ — The global store.
$n^* = 1$ — The generation of the global store.
procedure SEARCH()
if $\hat{\Sigma}_{\text{todo}} = \emptyset$
    return
remove $\hat{\varsigma} \in \hat{\Sigma}_{\text{todo}}$
$(call, \hat{\rho}, \hat{\sigma}, \hat{t}) \leftarrow \hat{\varsigma}$
$n \leftarrow \hat{S}[call, \hat{\rho}, \hat{t}]$ — The latest generation seen with this context.
if $n \geq n^*$
    return SEARCH() — Done—by monotonicity of $\rightsquigarrow$.
$\hat{\varsigma} \leftarrow (call, \hat{\rho}, \hat{\sigma} \sqcup \hat{\sigma}^*, \hat{t})$ — Install the widened store.
$\hat{\Sigma}_{\text{next}} \leftarrow \{\hat{\varsigma}' : \hat{\varsigma} \rightsquigarrow \hat{\varsigma}'\}$ — Explore successors.
$\hat{S}[call, \hat{\rho}, \hat{t}] \leftarrow n^*$ — Mark the current generation of the store as *seen*.
$\hat{\sigma}_{\text{next}} \leftarrow \bigsqcup \left\{\hat{\sigma} : (call, \hat{\rho}, \hat{\sigma}, \hat{t}) \in \hat{\Sigma}_{\text{next}}\right\}$ — Check each successor for changes.
if $\hat{\sigma}_{\text{next}} \sqsupset \hat{\sigma}^*$
    $n^* \leftarrow n^* + 1$ — Bump up the generation of the global store.
    $\hat{\sigma}^* \leftarrow \hat{\sigma}_{\text{next}}$ — Widen the global store.
$\hat{\Sigma}_{\text{todo}} \leftarrow \hat{\Sigma}_{\text{todo}} \cup \hat{\Sigma}_{\text{next}}$
return SEARCH()

Figure 1. State-space search algorithm using the time-stamp algorithm for computing k-CFA: SEARCH

that we will be binding to values. It stands to reason the larger these flow sets, the more values we will bind quickly (EFS).

5.3 Store

These are possible priority schemes based on the store component of a state.

- The flow set size of the function we are applying. This determines how many successor states we will have. If we prefer states that will generate more states, we might be able to subsequently pick the best of those.
- The flow set size of the arguments. Given that we want to reach the fixed point as quick as possible and that adding entries in the store is what gets us there, the more values we bind the better (SAS).
- Number of successor states. If our language supported an if form, we could ask the question of whether we will be exploring one branch, both branches, neither branch (SBF).
- The flow set size of the values we are binding. If our language supported `set!`, we might want to consider the flow set size of the variable we are binding or the flow set size of the value we are binding.
- Global store generation. The generation of the store is a metric of the size of the store. We might prefer to explore states that already have a larger store.

5.4 Time

These are possible priority schemes based on the time component of a state.

- The number of times we have seen a given time. We will often see the same time stamp in the course of an abstract interpretation. We could prefer states with calling contexts that we have already seen or put a preference on new ones (TFQ).
- The value of the time. We could prefer longer contexts or shorter contexts. For contexts of the same length, we could prefer ones that appear earlier or later in the program we are analyzing (TVL).

6. Evaluation

To evaluate our idea, we took the implementation from Might et al. [7] which uses the time stamp algorithm. We adapted it so it would use a priority queue for its work list,

Observing the run times from the original paper, you will note that the benchmarks run significantly faster. Updating the code to run on the latest version of Scala results in a 2x speedup. We also identified a bug where successor states were being added multiple times to the work list. Removing these duplicate entries also resulted in a 2x speedup.

We are also running on better hardware, but given that we reran the original implementation on the newer hardware as a point of reference, this should not be a concern.

The abbreviations and descriptions for the priority schemes we evaluated in our implementation can be found in the previous section. We used the same benchmarks analyzed by the original implementation [7]. The first two benchmarks, eta and map, test common functional idioms; sat is a back-tracking SAT-solver; regex is a regular expression matcher based on derivatives; scm2java is a Scheme compiler that targets Java; interp is a meta-circular Scheme interpreter; scm2c is a Scheme compiler that targets C.

Tables 1 and 2 compares the number of states that were generated for each benchmark. In some cases we can see that we generate only a fifth of the states as compared to the original implementation.

Tables 3 and 4 compares the runtimes of the varying strategies. In the best case we were able to achieve a 1.5x speedup.

Although no specific strategy is best for all benchmarks, the strategy CFQ tends to do well both in terms of reducing the number of states and decreasing the runtime. For a control-flow analysis that needs to use low memory and run fast, using one of the strategies that performs better than the baseline BFS and DFS strategies is worth considering.

	eta	map	sat	regex	scm2java	interp	scm2c
BFS	54	230	488	3692	7888	651	38899
DFS	66	186	293	2252	2595	657	21195
CTP	53	223	284	1831	2394	653	13663
CSZ	54	192	373	2063	2933	653	14510
CDL	54	**141**	343	2630	4110	653	19618
CBL	54	230	234	2205	2848	**648**	25808
CFQ	56	166	**223**	**1271**	**1718**	657	**7660**
CCR	53	272	520	2292	3557	656	14327
ESZ	**48**	178	296	2248	2966	649	25845
EFS	48	178	296	2248	2966	649	25845
SAS	53	247	373	1743	3414	655	13337
SBF	61	223	382	1645	3467	653	10288

Table 1. Number of states generated for $k = 0$.

	eta	map	sat	regex	scm2java	interp	scm2c
BFS	53	361	8696	12965	10001	635	157396
DFS	**38**	360	17216	11328	13397	635	130302
CTP	49	341	8831	6560	4080	635	94931
CSZ	53	344	6749	8467	**3828**	635	98366
CDL	53	**260**	**4527**	6039	4054	635	99471
CBL	44	343	7575	**4369**	4448	635	99333
CFQ	53	310	5819	7207	7651	635	96594
CCR	53	464	9855	8230	5444	635	98609
ESZ	51	342	6644	8932	4119	635	118390
EFS	51	342	6644	8932	4119	635	118390
SAS	49	442	8847	5661	5762	635	**84808**
SBF	47	456	9600	8283	6136	635	86390

Table 2. Number of states generated for $k = 1$.

	eta	map	sat	regex	scm2java	interp	scm2c
BFS	73	217	293	889	1214	828	3296
DFS	75	195	233	704	790	815	2617
CTP	66	217	230	622	777	818	2338
CSZ	69	202	264	692	850	832	2376
CDL	66	**167**	249	781	936	831	2344
CBL	80	229	216	691	831	867	2737
CFQ	68	182	**194**	**536**	**708**	828	**1824**
CCR	67	236	295	707	880	**812**	2366
ESZ	**65**	188	232	729	831	815	2754
EFS	71	200	238	729	881	878	3149
SAS	67	238	267	633	903	822	2319
SBF	74	221	267	618	901	826	2170

Table 3. Time in milliseconds for each benchmark for $k = 0$.

	eta	map	sat	regex	scm2java	interp	scm2c
BFS	65	274	1441	1587	1341	830	8878
DFS	**56**	273	1820	1478	1514	**819**	6413
CTP	61	272	1436	1122	940	825	6253
CSZ	65	271	1333	1354	**921**	827	6297
CDL	66	**232**	**1139**	1114	943	849	6247
CBL	65	274	1433	**929**	968	879	6603
CFQ	63	256	1184	1201	1227	834	5967
CCR	63	308	1533	1323	1048	821	6068
ESZ	62	269	1284	1348	931	831	7469
EFS	69	276	1355	1464	1006	895	9806
SAS	63	310	1537	1130	1127	834	**5338**
SBF	62	308	1528	1329	1116	822	5814

Table 4. Time in milliseconds for each benchmark for $k = 1$.

All benchmarks were run with a k of zero or one. Every strategy produced the same store for its final result.

7. Conclusion

In this paper we have demonstrated that how states are processed is important when computing k-CFA. We have described that there is a difference between doing a depth-first vs breadth-first search. We have also demonstrated that using a specific type of queue can play an important role in limiting the number of states explored.

Acknowledgments

This material is partially based on research sponsored by DARPA under agreements number AFRL FA8750-15-2-0092 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] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[2] J. M. Ashley and R. K. Dybvig. A practical and flexible flow analysis for Higher-Order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, 1998.

[3] M. Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Aug. 1987.

[4] T. Gilray and M. Might. A survey of polyvariance in abstract interpretations. In J. McCarthy, editor, *Trends in Functional Programming*, volume 8322 of *Lecture Notes in Computer Science*, pages 134–148. Springer Berlin Heidelberg, 2014.

[5] F. Henglein. Simple closure analysis. Technical report, Department of Computer Science, University of Copenhagen (DIKU), Mar. 1992.

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

[7] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. Object-Oriented program analysis. In *Proceedings of the 31st Conference on Programming Language Design and Implementation (PLDI 2006)*, pages 305–315, Toronto, Canada, June 2010.

[8] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[9] D. Van Horn and H. G. Mairson. Flow analysis, linearity, and PTIME. In M. Alpuente and G. Vidal, editors, *Static Analysis*, volume 5079 of *Lecture Notes in Computer Science*, pages 255–269. Springer Berlin Heidelberg, 2008.

[10] D. Van Horn and M. Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 51–62, New York, NY, USA, 2010. ACM.