Environment Unrolling

Steven Lyde and Matthew Might

University of Utah

Abstract. We propose a new way of thinking about abstract interpretation with a method we term environment unrolling. Model checkers for imperative languages will often apply loop unrolling to make their state space finite in the presence of loops and recursion. We propose handling environments in a similar fashion by putting a bound on the number of environments to which a given closure can transitively refer. We present how this idea relates to a normal model of abstract interpretation, give a general overview of its soundness proof in regards to a concrete semantics, and show empirical results demonstrating the effectiveness of our approach.

1 Introduction

In general the state space of an abstract machine for a λ-calculus is infinite because environments refer to closures and closures refer to environments [4]. In Abstracting Abstract Machines(AAM), a systematic recipe for breaking this cycle is given [5]. Bindings are store allocated and the number of addresses in the store is made finite.

We propose a slightly different perspective on how to make the state space finite. Model checkers for imperative language wills often apply loop unrolling to make their state space finite in the presence of loops and recursion. We propose handling environments in a fashion similar to how they handle loops, by putting a bound on the number of environments to which a given closure can transitively refer. As will be demonstrated, this provides a benefit in both speed and precision in an abstract interpretation.

To illustrate the basic idea, we provide a simple example using the following program.

```
(define (fact n)
  (if (zero? n) 1 (* n (fact (- n 1)))))
(fact 5)
```

In a traditional 0CFA, after the second recursive call, the environment would have the binding $[n \mapsto a]$, while the store would have the binding $[a \mapsto \{4, 5\}]$. However, if we were to use a concrete environment, the environment would contain the binding $[n \mapsto 4]$ and the store would be empty.

2 Concrete Semantics

The setting for our analysis is the A-normal form λ-calculus.

$$\begin{aligned}
e \in \mathsf{Exp} &::= \texttt{(let ((}v\ \mathit{call}\texttt{))}\ e\texttt{)} \mid \mathit{call} \mid \text{æ} \\
\mathit{call} \in \mathsf{Call} &::= \texttt{(}f\ \text{æ}\texttt{)} \\
f, \text{æ} \in \mathsf{Atom} &::= v \mid \mathit{lam} \\
\mathit{lam} \in \mathsf{Lam} &::= \texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)} \\
v \in \mathsf{Var} &\text{ is a set of identifiers}
\end{aligned}$$

A CESK style machine for this language has the following state space [2].

$$\begin{aligned}
c \in \mathit{Conf} &= \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} \\
\rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \to \mathit{Clo} \\
\mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Clo} \\
\kappa \in \mathit{Kont} &= \mathit{Frame}* \\
\phi \in \mathit{Frame} &= \mathsf{Var} \times \mathsf{Exp} \times \mathit{Env} \\
a \in \mathit{Addr} &\text{ is an infinite set of addresses}
\end{aligned}$$

To define the semantics we need the following three items.

1. We need a function $\mathcal{I} : \mathsf{Exp} \to \mathit{Conf}$ to inject our program into an initial configuration.
$$c_0 = \mathcal{I}(e) = (e, [], [], \langle\rangle)$$
2. We need a function $\mathcal{A} : \mathsf{Atom} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$ to evaluate atomic exrpessions.
$$\mathcal{A}(\mathit{lam}, \rho, \sigma) = (\mathit{lam}, \rho) \qquad\qquad \mathcal{A}(v, \rho, \sigma) = \sigma(\rho(v))$$
3. And we need a transition relation $(\Rightarrow) \subseteq \mathit{Conf} \times \mathit{Conf}$.

$$\begin{aligned}
\overbrace{(\llbracket \texttt{(}f\ \text{æ}\texttt{)} \rrbracket, \rho, \sigma, \kappa)}^{c} &\Rightarrow (e, \rho'', \sigma', \kappa), \text{ where} \\
(\llbracket \texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)} \rrbracket, \rho') &= \mathcal{A}(f, \rho, \sigma) \\
a &= \mathit{alloc}(v, c) \\
\rho'' &= \rho'[v \mapsto a] \\
\sigma' &= \sigma[a \mapsto \mathcal{A}(\text{æ}, \rho, \sigma)]
\end{aligned}$$

$$(\llbracket \texttt{(let ((}v\ \mathit{call}\texttt{))}\ e\texttt{)} \rrbracket, \rho, \sigma, \kappa) \Rightarrow (\mathit{call}, \rho, \sigma, (v, e, \rho) : \kappa)$$

$$\begin{aligned}
\overbrace{(\text{æ}, \rho, \sigma, (v, e, \rho') : \kappa)}^{c} &\Rightarrow (e, \rho'', \sigma', \kappa), \text{ where} \\
a &= \mathit{alloc}(v, c) \\
\rho'' &= \rho'[v \mapsto a] \\
\sigma' &= \sigma[a \mapsto \mathcal{A}(\text{æ}, \rho, \sigma)]
\end{aligned}$$

3 Abstract Semantics

We abstract the configuration space, but leave the continuation infinite by using a pushdown control-flow analysis [6].

$$
\begin{aligned}
\hat{c} \in \widehat{Conf} &= \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Kont} \\
\hat{\rho} \in \widehat{Env} &= \widehat{Env}_x + \widehat{Env}_0 \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \to \mathcal{P}(\widehat{Clo}) \\
\widehat{clo} \in \widehat{Clo} &= \mathsf{Lam} \times \widehat{Clo} \\
\hat{\kappa} \in \widehat{Kont} &= \widehat{Frame}* \\
\hat{\phi} \in \widehat{Frame} &= \mathsf{Var} \times \mathsf{Exp} \times \widehat{Env} \\
\hat{a} \in \widehat{Addr} & \text{ is a finite set of addresses}
\end{aligned}
$$

However, we have abstracted environments in a non-standard way. There are now two types of environments. The first type of environment has a depth which represents how many times we can extend the environment before using store allocation in the traditional fashion. We also have the classical environment, which maps variables to a set of addresses.

$$
\begin{aligned}
\widehat{Env}_x &= \mathsf{Var} \rightharpoonup \widehat{Clo}_y, x > y \\
\widehat{Clo}_x &= \mathsf{Lam} \times \widehat{Env}_x \\
\widehat{Env}_0 &= \mathsf{Var} \rightharpoonup \widehat{Addr}
\end{aligned}
$$

Our semantics will require a way to extract the depth from the environment.

$$
\hat{\mathcal{D}}(\hat{\rho}) = \begin{cases} x & \hat{\rho} \in \widehat{Env}_x \\ 0 & \hat{\rho} \in \widehat{Env}_0 \end{cases}
$$

We also overload the function $\hat{\mathcal{D}}$ over sets. When given a set, it returns the minimum depth of all environments contained in the set. This will also work over closures, where the environment is extracted from the closure.

$$
\hat{\mathcal{D}}(\hat{E}) = min(\left\{ d : d = \hat{\mathcal{D}}(\hat{\rho}), \hat{\rho} \in \hat{E} \right\})
$$

We need to inject a program into an initial configuration. When this is done, the limit on the amount of unrolling d is chosen.

$$
\hat{c}_0 = \mathcal{I}(e) = (e, \hat{\rho}, [], \langle\rangle), \text{ where } \hat{\rho} = [] \in \widehat{Env}_d
$$

The atomic evaluator is also slightly different than usual. It must take into account the depth of the environment. Lambda terms are handled in the traditional way. If we have a variable, we have to be cognizant of what type of environment we have. If we have an $\widehat{Env}_0$, we evaluate in the standard way. If we have the other type of environment $\widehat{Env}_x$, we look up the single value but put it in a set to avoid redefining the abstract transition relation.

$$\hat{\mathcal{A}}(lam, \hat{\rho}, \hat{\sigma}) = (lam, \hat{\rho})$$
$$\hat{\mathcal{A}}(v, \hat{\rho}, \hat{\sigma}) = \begin{cases} \hat{\sigma}(\hat{\rho}(v)) & \hat{\rho} \in \widehat{Env}_0 \\ \{\hat{\rho}(v)\} & \hat{\rho} \in \widehat{Env}_x \end{cases}$$

The transition relation is also slightly different. Function calls and atomic expressions call an auxiliary function to extend the environment and update the store.

$$\begin{aligned} (\llbracket \texttt{(}f\ \text{æ}\texttt{)} \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) &\rightsquigarrow (e, \hat{\rho}'', \hat{\sigma}', \hat{\kappa}), \text{ where} \\ (\llbracket \texttt{(λ (}v\texttt{)}\ e\texttt{)} \rrbracket, \hat{\rho}') &\in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \\ (\hat{\rho}'', \hat{\sigma}') &= \widehat{extend}(\hat{\rho}', \hat{\sigma}, v, \text{æ}) \end{aligned}$$

$$(\llbracket \texttt{(let ((}v\ call\texttt{))}\ e\texttt{)} \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \rightsquigarrow (call, \hat{\rho}, \hat{\sigma}, (v, e, \hat{\rho}) : \hat{\kappa})$$

$$\begin{aligned} (\text{æ}, \hat{\rho}, \hat{\sigma}, (v, e, \hat{\rho}') : \hat{\kappa}) &\rightsquigarrow (e, \hat{\rho}'', \hat{\sigma}', \hat{\kappa}), \text{ where} \\ (\hat{\rho}'', \hat{\sigma}') &= \widehat{extend}(\hat{\rho}', \hat{\sigma}, v, \text{æ}) \end{aligned}$$

The $\widehat{extend}$ function must take into account the depth of the current environment and what the depth will be after it is extended. If the environment is at depth zero, it is updated in the standard way using the store.

$$\begin{aligned} \widehat{extend}(\overbrace{e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}}^{\hat{c}}, v, \text{æ}) &= (\hat{\rho}', \hat{\sigma}') \text{ if } \hat{\mathcal{D}}(\hat{\rho}) = 0, \text{ where} \\ \hat{a} &= \widehat{alloc}(v, \hat{c}) \\ \hat{\rho}' &= \hat{\rho}[v \mapsto \hat{a}] \\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(\text{æ}, \hat{\rho}, \hat{\sigma})] \end{aligned}$$

We must also be able to distinguish if extending the environment will drop the depth to zero. If that is the case, we need to allocate addresses for all the values in the environment and put them in the store.

$$\begin{aligned} \widehat{extend}(\overbrace{e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}}^{\hat{c}}, v, \text{æ}) &= (\hat{\rho}', \hat{\sigma}) \text{ if } cond, \text{ where} \\ cond &= \hat{\mathcal{D}}(\hat{\rho}) > 0 \text{ and } \hat{\mathcal{D}}(\hat{\mathcal{A}}(\text{æ}, \hat{\rho}, \hat{\sigma})) \leq 1 \\ (\hat{\rho}', \hat{\sigma}') &= \widehat{extend}(\widehat{alloc}_{\rho}(\hat{c}), v, \text{æ}) \end{aligned}$$

In the final case were we are still dealing with environments of depth greater than zero and the environment from the argument has depth greater than one, we do not need to allocate any addresses, but can simply update the environment.

$$\begin{aligned}\widehat{extend}(\overbrace{e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}}^{\hat{c}}, v, æ) = (\hat{\rho}', \hat{\sigma}) \text{ if } cond, \text{ where}\\ cond &= \hat{\mathcal{D}}(\hat{\rho}) > 0 \text{ and } \hat{\mathcal{D}}(\hat{\mathcal{A}}(æ, \hat{\rho}, \hat{\sigma})) > 1\\ \widehat{clo} &\in \hat{\mathcal{A}}(æ, \hat{\rho}, \hat{\sigma})\\ \hat{\rho}'' &= \hat{\rho}'[v \mapsto \widehat{clo}]\end{aligned}$$

As seen above, we need a function that can convert a state that has an environment of non-zero depth into an environment that is at depth zero.

$$\begin{aligned}\widehat{alloc}_{\rho}(\overbrace{e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}}^{\hat{c}}) = (e, \hat{\rho}', \hat{\sigma}', \hat{\kappa}), \text{ where}\\ v_i &\in dom(\hat{\rho})\\ \hat{a}_i &= \widehat{alloc}(v_i, \hat{c})\\ \hat{\rho}' &= [][v_i \mapsto \hat{a}_i]\\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\rho}(v_i)]\end{aligned}$$

3.1 Soundness

To prove the soundness we must show that the abstract semantics simulate the concrete semantics following the standard proof found in [3]. The key insight for the proof is that we can provide an abstraction map that allocates a unique abstract address for every value in a non-zero depth environment.

4 Evaluation

We have implemented this analysis and show the results on the exact benchmarks presented in [1] in Figure 1. Note that when the depth is zero we get the same results as a traditional pushdown analysis. When using a depth of one, we see that precision is equal to or more precise than when $k = 1$, while at the same time generating a smaller abstract transition graph. With further increases in the depth, we see better precision and smaller transition graphs than even occurs with abstract garbage collection. The run times of the new analyses correlate to how many states and edges are in the graph. Environment unrolling gives improvements in both precision and speed.

5 Conclusion

We have shown a unique approach to abstract interpretation. We have described how our approach is similar to loop unrolling and given a general overview of its semantics. We have also provided results from an implementation of the analysis, which demonstrate its possible benefits.

Program	Exp	Var	k	PDCFA			PDCFA+GC			d	PDCFA+UE		
`mj09`	19	10	0	38	38	4	33	32	4	0	38	38	4
			1	44	48	1	32	31	1	1	39	40	4
										2	40	40	3
										3	38	37	3
										4	32	31	3
`eta`	21	13	0	32	32	6	30	29	8	0	32	32	6
			1	30	29	8	30	29	8	1	32	32	8
										2	29	29	10
`kcfa2`	20	10	0	36	35	4	35	34	4	0	36	35	4
			1	87	144	2	35	34	2	1	76	114	3
										2	29	30	3
`kcfa3`	25	13	0	50	51	5	53	52	5	0	50	51	5
			1	1761	4046	2	53	52	2	1	489	905	3
										2	53	52	3
`blur`	40	20	0	523	813	3	299	335	9	0	523	813	3
			1	324	348	9	320	344	9	1	49	49	12
										2	47	48	13
`loop2`	41	16	0	108	117	4	67	71	4	0	108	117	4
			1	398	512	3	145	156	3	1	74	77	5
`sat`	51	23	0	545	773	4	254	317	4	0	545	773	4
			1	10872	14797	4	71	73	10	1	1400	1825	7
										2	7625	9769	7
										3	71	73	13

Fig. 1. Benchmark results. The first three columns provide the name of the benchmark, the number of expressions and variables in the program. The next seven columns show the results for running the original analysis with $k \in \{0, 1\}$ and with garbage collection off and on. Under each analysis, the first column is the number of *control states* and the second column is the number of *edges* computed during the analysis. The third column is the number of *singleton* variables. The last four columns show the results of our analysis. It gives the depth d selected and the number of *control states*, *edges* and *singleton* variables. The depth is not shown for larger values if the precision was not increased.

While the presentation only demonstrated the abstract interpretation for ANF lambda calculus, it can immediately be applied to other language forms, including those involving mutation. In the case of mutation, we would need to always store allocate mutable variables.

This material is based on research sponsored by DARPA under the programs Automated Program Analysis for Cybersecurity (FA8750-12-2-0106) and Clean-Slate Resilient Hosts (CRASH) as well as by NSF under the program Faculty Early Career Development (CAREER-1350344).

References

1. Earl, C., Sergey, I., Might, M., Van Horn, D.: Introspective pushdown analysis of higher-order programs. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012). pp. 177–188. ICFP '12, ACM, New York, NY, USA (2012)
2. Felleisen, M., Friedman, D.P.: A calculus for assignments in higher-order languages. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 314+. ACM, New York, NY, USA (1987)
3. Might, M.: Environment Analysis of Higher-Order Languages. Ph.D. thesis, Georgia Institute of Technology (Jun 2007)
4. Shivers, O.G.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1991)
5. Van Horn, D., Might, M.: Abstracting abstract machines. In: ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. pp. 51–62. ICFP '10, ACM Press (Sep 2010)
6. Vardoulakis, D., Shivers, O.: CFA2: A Context-Free approach to Control-Flow analysis. In: Gordon, A.D. (ed.) Programming Languages and Systems, Lecture Notes in Computer Science, vol. 6012, chap. 30, pp. 570–589. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)