Strong Function Call

Steven Lyde and Matthew Might

University of Utah

Abstract. This work presents an incremental improvement to abstract interpretation of higher order languages, similar to strong update, which we term strong function call. In an abstract interpretation, after a function call, we know which abstract closure was called and can deduce that any other values found at the same abstract address in the abstract store could not possibly exist in the corresponding concrete state. Thus they can be removed from the abstract store without loss of soundness. We provide the intuition behind this analysis along with a general overview of its soundness proof.

1 Introduction

Because of the unmovable force of the halting problem, static analyses as a rule must be imprecise in some cases in order to remain inside the curtain of computability. Concrete values must be abstracted, but this abstraction leads to imprecision. Any techniques that regain some if this lost precision are desirable. To this end we propose strong function call. A small extension to a traditional k-CFA implementation [5].

In an abstract interpretation of a λ-calculus which uses store allocation, the source of non-determinism is that an abstract address can point to several abstract closures. This results in a fork in the abstract transition graph. Any subsequent function calls that dereference that same address will also fork. However, the key insight used in this paper is that once we have made a function call, we know which procedure was called and can deduce that any other values in the store could not possibly exist in the corresponding concrete state. Thus these extra values should be pruned from the store in order to improve the precision of subsequent dereferences of the address.

2 CPS

To give an intuition for this problem we will develop and abstract semantics that operates over continuation-passing style λ-calculus (CPS) with multiple argument lambda terms [6]. The benefits of this is that the language is simple enough that more mental energy can be spent on focusing over the intuition of the analysis, rather than on the language itself. However, the ideas presented in this paper are not restricted to CPS, but can easily be extended to different language forms.

In CPS, all expression are either call sites, variables or lambda terms, like in the original λ-calculus, but with the additional restrictions that the body of a lambda terms must be a call site and that the function and arguments at a call site must be atomic, meaning that they can only be a variable or lambda term.

The syntax for the language we will work with is as follows:

$$\begin{aligned} \mathit{call} \in \mathsf{Call} &::= (f\ æ_1 \dots æ_n) \\ f, æ \in \mathsf{Atom} &::= v \mid \mathit{lam} \\ \mathit{lam} \in \mathsf{Lam} &::= (\lambda\ (v_1 \dots v_n)\ \mathit{call}) \\ v \in \mathsf{Var} &\text{ is a set of identifiers} \end{aligned}$$

3 Concrete Semantics

We first introduce a small-step operational semantics to formally specify the behavior of this language, which we will later abstract. We use a CES style abstract machine and provide a transition relation $(\Rightarrow) \subseteq \Sigma \times \Sigma$. The state space of this abstract machine is as follows. A control state contains a control expression, environment and store.

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{Call} \times \mathit{Env} \times \mathit{Store} \\ \rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\ \mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env} \\ \sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Clo} \\ a \in \mathit{Addr} &\text{ is an infinite set} \end{aligned}$$

In order to evaluate atomic expressions, we introduce an auxiliary function, $\mathcal{A} : \mathsf{Atom} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$. In the case of a variable, it looks up the address of the variable in the environment and then looks up the value at that address in the store. In the case of a lambda term, it produces a closure by closing the lambda term with the current environment.

$$\begin{aligned} \mathcal{A}(v, \rho, \sigma) &= \sigma(\rho(v)) \\ \mathcal{A}(\mathit{lam}, \rho, \sigma) &= (\mathit{lam}, \rho) \end{aligned}$$

In defining the transition relation $(\Rightarrow)$, we find one of the main benefits of using CPS. The transition relation can be defined with a single rule. It starts by atomically evaluating the function. It proceeds by allocating a new address for each argument. In the concrete semantics, a unique address is used that will never be used again. It then extends the environment of the closure and binds

the values of the arguments to those addresses in the store.

$$\overbrace{(\llbracket (f\ æ_1 \ldots æ_n) \rrbracket, \rho, \sigma)}^{\varsigma} \Rightarrow (call, \rho'', \sigma'), \text{ where}$$
$$(\llbracket (\lambda\ (v_1 \ldots v_n)\ call) \rrbracket, \rho') = \mathcal{A}(f, \rho, \sigma)$$
$$a_i = alloc(v_i, \varsigma)$$
$$\rho'' = \rho'[v_i \mapsto a_i]$$
$$\sigma' = \sigma[a_i \mapsto \mathcal{A}(æ_i, \rho, \sigma)]$$

4 Abstract Semantics

The standard approach to abstract this machine is to make the address space finite as described in Abstracting Abstract Machines [7]. The consequence of this action is that we will be forced to have multiple values in the store at a single address. However, note that these values cannot grow infinitely because the state space is finite. However, we now have the issue that a single abstract address can represent multiple concrete addresses. This is where abstract counting comes into play [2]. The abstract count of an abstract address is the number of concrete addresses that abstract address represents.

A natural domain for abstract counting is the set $\hat{\mathbb{N}}$. We care if an abstract address represents a single concrete address or multiple addresses. Our analysis will leverage the power of this information.

$$\hat{\mathbb{N}} = \{0, 1, \infty\}$$

Note that the abstract count cannot be gleaned simply from the store and the number of abstract closures at a particular address. It can be the case that a single abstract address representing multiple concrete addresses can only contain one value in the store. It can also be the case that an abstract address can represent a single concrete address but have multiple values in the store. Indeed this is the case of which our analysis will take advantage.

Defining the operator $\oplus$ naturally extends over the addition operator. In this paper the operator will also lift point-wise over functions, allowing us to update a store which maps abstract address to abstract counts. Here we see that we are retaining the vital information that we need, whether we have a single value or multiple values.

$$0 \oplus \hat{n} = \hat{n}$$
$$\hat{n} \oplus 0 = \hat{n}$$
$$1 \oplus 1 = \infty$$
$$\hat{n} \oplus \infty = \infty$$
$$\infty \oplus \hat{n} = \infty$$

One traditional use case for abstract counting in a higher order setting is for strong update [1]. If it can be shown that an abstract address only represents one concrete address, we do not need to join that value in the store, but can shadow the old value.

$$
\begin{aligned}
(\llbracket (\texttt{set!}\ v\ æ\ call) \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\mu}) \rightsquigarrow (call, \hat{\rho}, \hat{\sigma}', \hat{\mu}), \text{ where} \\
a &= \hat{\rho}(v) \\
\widehat{clo} &= \hat{\mathcal{A}}(æ, \hat{\rho}, \hat{\sigma}) \\
\hat{\sigma}' &= \begin{cases} \hat{\sigma}[\hat{a} \mapsto \widehat{clo}] & \hat{\mu}(a) \leq 1 \\ \hat{\sigma} \sqcup [\hat{a} \mapsto \widehat{clo}] & \text{otherwise} \end{cases}
\end{aligned}
$$

We now proceed by defining the abstract semantics for our analysis. The abstract state space is very similar to the concrete state space, except stores now map addresses to a set of abstract closures. We have also added an additional store like entity $\hat{\mu}$, which will keep track of abstract counts.

$$
\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \times \widehat{Count} \\
\hat{\rho} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \widehat{Addr} \\
\widehat{clo} \in \widehat{Clo} &= \mathsf{Lam} \times \widehat{Env} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \mathcal{P}(\widehat{Clo}) \\
\hat{\mu} \in \widehat{Count} &= \widehat{Addr} \rightharpoonup \hat{\mathbb{N}} \\
\hat{a} \in \widehat{Addr} &\text{ is a finite set}
\end{aligned}
$$

We also have an abstract atomic evaluator that performs the same operations as its concrete counterpart $\hat{\mathcal{A}} : \mathsf{Atom} \times \widehat{Env} \times \widehat{Store} \rightharpoonup \mathcal{P}(\widehat{Clo})$. The difference being that it now returns a set of abstract closures, rather than a single value.

$$
\begin{aligned}
\hat{\mathcal{A}}(v, \hat{\rho}, \hat{\sigma}) &= \hat{\sigma}(\hat{\rho}(v)) \\
\hat{\mathcal{A}}(lam, \hat{\rho}, \hat{\sigma}) &= \{(lam, \hat{\rho})\}
\end{aligned}
$$

The function $\hat{\mathcal{G}} : \mathsf{Var} \times \widehat{Clo} \times \widehat{Env} \times \widehat{Store} \times \widehat{Count} \rightharpoonup \widehat{Store}$ is the crux of our analysis. It updates the store by possibly pruning values that are no longer needed. It determines its action based on the syntactic type of the function we are applying. In the case of a lambda term, nothing is done to the store. If the case of a variable term, if the abstract address represents multiple concrete addresses, the function does nothing to the store. If the abstract address only represents a single concrete address, it shadows the value pointed to by the address, restricting its value to only be the function that is being applied. In the case where there is only one closure at the address, this operation has no effect. However, in the case, where there are multiple closures at that abstract

address, it has the effect of restricting the store to only have a single value at that address.

$$\hat{\mathcal{G}}(lam, \widehat{clo}, \hat{\rho}, \hat{\sigma}, \hat{\mu}) = \hat{\sigma}$$

$$\hat{\mathcal{G}}(v, \widehat{clo}, \hat{\rho}, \hat{\sigma}, \hat{\mu}) = \begin{cases} \hat{\sigma}[\hat{\rho}(v) \mapsto \left\{\widehat{clo}\right\}] & \hat{\mu}(\hat{\rho}(v)) \leq 1 \\ \hat{\sigma} & \text{otherwise} \end{cases}$$

The abstract transition relation is also very similar to its concrete counterpart. The main difference lies in that the atomic evaluator may return multiple values. This results in branching in the abstract transition graph. We have also added the abstract counting map to maintain the information needed for our analysis. Also notice that we now use an auxiliary function to update the store before joining it with the values from the arguments of the call site. This restricts the store to only contain the closure that is being applied at the call site. This reduces the size of the abstract state and could result in increased precision and speed.

$$\overbrace{(\llbracket (f\ æ_1 \ldots æ_n) \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\mu})}^{\hat{\varsigma}} \rightsquigarrow (call, \hat{\rho}'', \hat{\sigma}'', \hat{\mu}'), \text{ where}$$

$$\widehat{clo} \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$

$$(\llbracket (\lambda\ (v_1 \ldots v_n)\ call) \rrbracket, \hat{\rho}') = \widehat{clo}$$

$$\hat{a}_i = \widehat{alloc}(v_i, \hat{\varsigma})$$

$$\hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{a}_i]$$

$$\hat{\sigma}' = \hat{\mathcal{G}}(f, \widehat{clo}, \hat{\rho}, \hat{\sigma}, \hat{\mu})$$

$$\hat{\sigma}'' = \hat{\sigma}' \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma})]$$

$$\hat{\mu}' = \hat{\mu} \oplus [\hat{a}_i \mapsto 1]$$

4.1 Soundness

To prove the soundness of this analysis, we provide an abstraction map that connects the concrete and abstract state spaces.

$$\alpha((call, \rho, \sigma)) = (call, \alpha(\rho), \alpha(\sigma), \alpha_\mu(\sigma))$$

$$\alpha(\rho) = \lambda v.\alpha(\rho(v))$$

$$\alpha(\sigma) = \lambda \hat{a}. \bigsqcup_{\alpha(a)=\hat{a}} \{\alpha(\sigma(a))\}$$

$$\alpha_\mu(\sigma) = \lambda \hat{a}. \bigoplus_{\alpha(a)=\hat{a}} 1$$

$$\alpha(lam, \rho) = \{(lam, \alpha(\rho)\}$$

$$\alpha(a) \text{ is determined by the allocation function}$$

From there we prove that the abstract transition relation simulates the concrete transition relation.

Theorem 1. *If* $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$ *and* $\varsigma \Rightarrow \varsigma'$ *then there must exist* $\hat{\varsigma} \in \hat{\Sigma}$ *such that* $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$ *and* $\hat{\varsigma} \rightsquigarrow \hat{\varsigma}'$.

Proof. The proof follows in the same manner as presented in [3]. The only difference between this abstraction and the standard one can be found in the case where we restrict the size of the store. However, this is simple to account for, as the concrete semantics can only apply one function at a call site. Realizing that the concrete address can only hold one value and that the abstract address only represents one concrete address, it is sound to restrict the store to that one value.

5 Conclusion

In this work we have shown that strong update can be used to restrict the size of the store at application sites. By reducing the size of the store it is highly likely we will gain both speed and precision. This has been shown to be the case with abstract garbage collection [4].

It might be easy to assume that abstract garbage collection would subsume this analysis. However, this is not the case. Abstract garbage collection filters out bindings that are not reachable from the root set. However, in this analysis, we are dealing with an address that is definitely live and will not be garbage collected. Even in the presence of abstract garbage collection, we still get flow sets that contain more than one value. Otherwise, the precision of an analysis with abstract garbage collection would be perfect.

This idea could also easily be extended to object oriented languages. With its extensive use of polymorphism, one could imagine that the calling object of a method could easily have multiple flow sets. This analysis would allow us to soundly reduce the size of these flow sets.

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. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation. pp. 296–310. PLDI '90, ACM, New York, NY, USA (1990)
2. Jagannathan, S., Thiemann, P., Weeks, S., Wright, A.: Single and loving it: must-alias analysis for higher-order languages. In: POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 329–341. ACM (1998)
3. Might, M.: Environment Analysis of Higher-Order Languages. Ph.D. thesis, Georgia Institute of Technology (2007)

4. Might, M., Shivers, O.: Improving flow analyses via Gamma-CFA: Abstract garbage collection and counting pp. 13–25 (2006)
5. Shivers, O.G.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie Mellon University (1991)
6. Steele, G.L.: Rabbit: A compiler for scheme. Tech. rep., Massachusetts Institute of Technology (1978)
7. 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)