Extracting single-function CFGs from an AAM analysis

KYLE HEADLEY*, University of Alabama at Birmingham

1 INTRO

An AAM-style program analysis [Van Horn and Might 2010] produces a control-flow graph (CFG) where the graph nodes are the states of an abstract machine. This analysis is useful for students, code auditors, and analysis designers. It can be used in an automated way, but it may be too complex to be understood manually.

To help improve the way people come to understand an analysis, the author is working with a team to create an interactive visualization. It simplifies a whole-program CFG (wpCFG) by segmenting it into multiple graphs showing the control-flow of individual functions. The author is tasked with using the wpCFG to generating these graphs, a process explored in the rest of this paper. The full visualization has many additional features and may be accessed online (https://analysisviz.gilray.net).

Continuous sections of a function's CFG exist within a wpCFG, but there are complications to directly extracting them.

- Multiple calls to the function may exist throughout the wpCFG
- Each call may have a different flow
- Calls to other functions are "inlined" by the wpCFG
- Identifying info may change with every machine state

The author overcomes these issues with a novel algorithm and minor adjustments to the analysis, within the bounds of the AAM methodology. The result is multiple CFGs, each composed of all and only states associated with a particular function in the wpCFG. Each represents all the flows through it.

2 AAM

An example of an abstract machine is a CEK-machine [Felleisen and Friedman 1987]. This is a tuple of control expression (from the lambda calculus), environment (for lazy substitutions) and continuation (to keep track of parts of the expression not currently in focus). Stepping this machine to the next state corresponds to a reduction step in the lambda calculus. There are many versions of this formalism, for example a CESK machine, which adds a store to deal with semantics of mutation.

A store is also useful for abstracting this type of machine for a computable analysis. A loop in a CEK machine expression evaluation may require an infinite linked list as a continuation. Instead, the store can contain a portion of the loop with a link back through the store to itself. This representation is finite even if it represents an infinite sequence. Store addresses may also contain information for distinguishing one part of an analysis from another, even if it corresponds to the same syntax.

Figure 1 describes the abstract abstract machine used in this work. Machine states are at the top, with the "eval" state an extension of the CESK machine. The store is separated for use with variables or continuations and instrumentation is added. This is separated from the "apply" state which has a transition rule (in Figure 2) that deals with allocations and bindings, rather than eval's pushing and popping continuation frames. The instrumentation added to each of these provides

*ACM Member 1004593, PhD Student of Thomas Gilray

Author's address: Kyle Headley, University of Alabama at Birmingham, kheadley@uab.edu.

$$\hat{\varsigma} \in \hat{\Sigma} \triangleq \begin{cases} \text{eval: } \mathbf{Exp} \times \widehat{Env} \times \widehat{Instr} \times \widehat{Store} \times \widehat{KStore} \times \widehat{Kont} \\ \text{apply: } \hat{D}^* \times \widehat{Instr} \times \widehat{Store} \times \widehat{KStore} \times \widehat{Kont} \end{cases}$$

$$\hat{\sigma} \in \widehat{Store} \triangleq \widehat{Addr} \to \hat{D} \qquad \hat{\kappa} \in \widehat{Kont} \triangleq \widehat{Frame}^* \times \widehat{Addr}$$

$$\hat{d} \in \hat{D} \triangleq \mathcal{P}(\widehat{Clo}) \qquad \hat{\psi} \in \widehat{Frame} \triangleq \hat{D}^* \times \mathbf{Exp}^* \times \widehat{Env} \times \widehat{Instr}$$

$$\widehat{clo} \in \widehat{Clo} \triangleq \mathbf{Var}^* \times \mathbf{Exp} \times \widehat{Env} \qquad \hat{\sigma}_\kappa \in \widehat{KStore} \triangleq \widehat{Addr} \to \mathcal{P}(\widehat{Kont})$$

Fig. 1. Selected machine domains

$$\langle \textbf{apply } (\hat{d}_\lambda\ \hat{d}_0 \ldots \hat{d}_n), \hat{\imath}, \hat{\sigma}, \hat{\sigma}_\kappa, \hat{\kappa} \rangle \rightsquigarrow \langle \textbf{eval } e, \hat{\rho}', \hat{\imath}, \hat{\sigma}', \hat{\sigma}'_\kappa, \hat{a}_\kappa \rangle, \text{ where}$$

$$\langle \textbf{clo } (x_0 \ldots x_n), e, \hat{\rho} \rangle \in \hat{d}_\lambda \qquad \hat{\rho}' = \hat{\rho}[x_i \mapsto \hat{a}_i]$$

$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{d}_i] \qquad \hat{\sigma}'_\kappa = \hat{\sigma}_\kappa \sqcup [\hat{a}_\kappa \mapsto \hat{\kappa}]$$

$$\hat{a}_i = \widehat{alloc}(\hat{\varsigma}, x_i) \qquad \hat{a}_\kappa = (e,\ \hat{\rho}')$$

Fig. 2. Selected transition rule, a function call

precision to an analysis, which would otherwise conflate each use of some syntax with every other use of it. Finally, the $\hat{D}$ represents a value as a set of all possible values at a machine state.

The segmentation algorithm is sensitive to calls and returns, so we take a deeper look at the transition rule that steps into a closure, seen in Figure 2. Here, the apply state holds fully evaluated values $\hat{d}_\lambda$, the function to call, and $\hat{d}_0 \ldots \hat{d}_n$, the parameters. The machine potentially steps to multiple new states, one for each of the closures in $\hat{d}_\lambda$, to evaluate its body e. The rest of the computation involves generating or using addresses, used to refer to store allocated resources.

The addresses $\hat{a}_i$ link the environment $\hat{\rho}$ to the store $\hat{\sigma}$. The generating function $\widehat{alloc}$ is a parameter of the analysis and may make use of the state's instrumentation to provide precision. The continuation address $\hat{a}_\kappa$ is generated by a fixed function that provides maximum precision in this type of analysis as determined by [Gilray et al. 2016]. This address may eventually be the target of a return transition, so the allocation here provides a great mechanism for linking calls with returns.

3 SEGMENTATION

The segmentation algorithm uses the AAM analysis as its input data and generates a separate CFG for each lambda expression analyzed. The nodes in each of these may be final (no outgoing edges) with some relevant information, or they may be flow nodes that contain a set of states from the analysis. Each of these states will have the same control expression, but may have different instrumentation, environment, or continuation. By associating states in this way, all paths through the function can be represented at once.

The algorithm proceeds in three stages: caching calls, caching returns, and stepping through each function. Calls are entry points to functions, those states that follow an apply state. They are identified by the function they enter. Returns are states following atomic expression states, they are identified by the continuation address of the atomic state.

The CFG creation step is similar to an abstract machine itself. The injection function to start it constructs a flow node ("node" is used to distinguish from the states of the AAM analysis) with a

$$\langle \textbf{flow } \{\ldots\langle \textbf{eval } ae, \hat{\rho}, \hat{\sigma}, \textbf{halt}\rangle\ldots\}\rangle \rightsquigarrow \langle \textbf{halt-with } ae, \hat{\rho}, \hat{\sigma}\rangle \quad \textbf{[Halt]}$$

$$\langle \textbf{flow } \{\ldots\hat{\varsigma}\ldots\}\rangle \rightsquigarrow \langle \textbf{stuck-at } \hat{\varsigma}\rangle, \text{ where } \{\} = \widehat{\text{step}}(\hat{\varsigma}) \quad \textbf{[Stuck]}$$

$$\langle \textbf{flow } \hat{\varsigma}_{set}\rangle \rightsquigarrow \langle \textbf{exit-to } \hat{\varsigma}'\rangle, \text{ where} \quad \textbf{[Exit]}$$
$$\hat{\varsigma} = \langle \textbf{eval } ae, \_, \_, \_, \_, \hat{a}_\kappa\rangle \in \hat{\varsigma}_{set}$$
$$\hat{\varsigma}' \in \widehat{\text{step}}(\hat{\varsigma})$$

$$\langle \textbf{flow } \hat{\varsigma}_{set}\rangle \rightsquigarrow \langle \textbf{no-return } \hat{\varsigma}_r\rangle, \text{ where} \quad \textbf{[NoReturn]}$$
$$\hat{\varsigma}' = \langle \textbf{apply } \_, \_, \_, \_, \_\rangle \in \hat{\varsigma}_{set}$$
$$\hat{\varsigma}_r = \langle \textbf{eval } \_, \_, \_, \_, \_, \hat{a}_\kappa\rangle \in \widehat{\text{step}}(\hat{\varsigma}')$$
$$\{\} = \text{returns}(\hat{a}_\kappa)$$

$$\langle \textbf{flow } \hat{\varsigma}_{set}\rangle \rightsquigarrow \langle \textbf{flow } \hat{r}\rangle, \text{ where} \quad \textbf{[Return]}$$
$$\hat{\varsigma}'_{set} = \bigcup_{\hat{\varsigma}' = \langle \textbf{apply } \_,\_,\_,\_,\_\rangle \in \hat{\varsigma}_{set}} \widehat{\text{step}}(\hat{\varsigma}')$$
$$\hat{r} = \bigcup_{\langle \textbf{eval } \_,\_,\_,\_,\_,\hat{a}_\kappa\rangle \in \hat{\varsigma}'_{set}} \text{returns}(\hat{a}_\kappa)$$

$$\langle \textbf{flow } \hat{\varsigma}_{set}\rangle \rightsquigarrow \langle \textbf{flow } \hat{\varsigma}'_{set}\rangle, \text{ where} \quad \textbf{[Step]}$$
$$\hat{\varsigma}'_{set} = \bigcup_{\hat{\varsigma}' \in \hat{\varsigma}_{set}} \widehat{\text{step}}(\hat{\varsigma}')$$

Fig. 3. Segmentation rules

set of calls from our cache. It then follows the rules in Figure 3. The first of these recognize halt or stuck states in the analysis, generating appropriate terminal nodes. The next is for natural exits from the function, that is, when the control has been reduced to an atomic expression but there is still a continuation to step into.

Rule **[NoReturn]** in Figure 3 is a failure case of **[Return]**, which may occur when control passes into an infinite loop after a call to another function. Otherwise, the return flow node consists of all states that return to this point in the function after the calls in the previous states. The algorithm generates them by first stepping each state into the called function. This will provide the continuation addresses allocated during the step. Next the cache of returns is accessed with each address, to generate all the returns to that address. The final rule is **[Step]**, which is used when the others don't apply, and simply steps each state in our flow node and collects them into another.

4 CONCLUSIONS AND FUTURE WORK

This paper describes an algorithm for generating single-function CFGs from a whole-program CFG generated by an AAM analysis. It collects multiple versions together by searching for their starting points and avoids detail changes by flowing through them in unison. It steps over other functions by jumping to return addresses after a call. The current semantics have no intra-function branching, though future versions will. This will require an additional separation step in order to keep control expressions aligned. CGFs generated by this method are easier to understand and to further analyze than whole-program CFGs.

REFERENCES

Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*. 193–222.

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown Control-Flow Analysis For Free. *Proceedings of the Symposium on the Principals of Programming Languages (POPL)* (January 2016).

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *International Conference on Functional Programming*. 51.