CS4215 Project Report

Team Estepona - Abstract Abstract Machines

Tay Jing Xuan Tee Weile Wayne

April 2022

1 Abstract

We implemented an abstract interpreter for a subset of source. By approximating program states, our abstract interpreter always terminates and can be used to analyse program behaviour and allow for code optimisation. Our interpreter also allows for visualisation of the abstract state-space for analysis.

2 Introduction

The objective of static program analysis is to determine the behaviour of a program without running it. This includes determining whether the program terminates, whether optimisations can be applied, and what states are reachable during program execution.

This is a difficult problem. The halting problem for example is undecidable. Data-flow analysis is also difficult to perform on languages such as Source which support higher-order functions.

A key difficulty is that a virtual machine executing a program might require an infinite number of states for a non-terminating program. However, by following the abstracting abstract machines approach [1], we are able to convert a concrete virtual machine into a non-deterministic abstract machine with finite states, with each abstract state representing multiple possible concrete states. As a result, we are able to soundly approximate program behaviour in finite time.

For example, if the concrete machine transitions from state A to state B, then the abstract machine must transition from some abstract state A' to some abstract state B', where A' and B' are the abstract states representing A and B. If the abstract machine always terminates, then so must the concrete machine.

We applied this approach to convert the Source virtual machine into an abstract interpreter which supports higher-order functions and conditionals. The interpreter provably terminates, returning a graph of abstract states reached that can be visualised by the user and potentially used to derive complier optimisations.

3 Implementation

3.1 Overview

Our abstract interpreter runs on the Source virtual machine code produced by the Source complier. We derived it from the original Source virtual machine by applying the technique of abstracting abstract machines [1]. Each state of our abstract interpreter is of the form:

$$(PC, OS, ENV, STORE, KONT, TIME, C)$$

Similar to the original Source virtual machine, PC is a number representing the program counter and OS is the operand stack itself, storing integers, booleans and closures. ENV maps the names of function parameters and local variables to addresses. Note that these names in practice are actually integers. $STORE$ maps addresses to values. $KONT$ is the address of the continuation. A continuation is similar to a stack frame and it stores the previous $PC, OS, ENV, KONT, TIME$ and C. $TIME$ is a string representing the current call stack. It is the concatenation of the locations of the $CALL$ instructions in the call stack and is used to allocate new addresses. C is an incrementing counter that is used alongside $TIME$ to allocate new addresses.

3.2 Store

Importantly, the *STORE* maps each address (a string) to a set of values. When the store is dereferenced by any instruction, the interpreter non-deterministically chooses a value from the set of possible values, and transitions to the corresponding new state. When generating the state graph, all such possible transitions are explored.

To illustrate, here are the rules for the load and assignment instructions:

$$\frac{s(PC) = LD\ name}{\begin{array}{c}(PC, OS, ENV, STORE, KONT, TIME, C) \rightarrow \\ (PC+1, v.OS, ENV, STORE, KONT, TIME, C)\end{array}}$$

where $v \in STORE(ENV(name))$

$$\frac{s(PC) = ASSIGN\ name}{\begin{array}{c}(PC, v.OS, ENV, STORE, KONT, TIME, C) \rightarrow \\ (PC+1, OS, ENV, STORE[ENV(name) \leftarrow v], KONT, TIME, C)\end{array}}$$

where $STORE[addr \leftarrow v]$ denotes adding the value v to the set of values at the address *addr* in the *STORE*

3.3 Functions

3.3.1 LDF

The *LDF* instruction pushes a closure on the operand stack. It receives the function address and the number of slots the environment needs to be extended by (number of parameters and local variables) as input.

$$\frac{s(PC) = LDF \quad funcAddr \quad numToExtend}{\begin{array}{c}(PC, OS, ENV, STORE, KONT, TIME, C) \rightarrow \\ (PC+1, closure.OS, ENV, STORE', KONT, TIME, C)\end{array}}$$

where

$$envAddr = TIME.funcAddr.\text{``}env''$$

$$STORE' = STORE[envAddr \leftarrow ENV]$$

$$closure = (funcAddr, envAddr, numToExtend)$$

Note that the environment address is the $TIME$ concatenated with the function address concatenated with the string constant $“env''$. The use of the function address and string constant ensures that the set of values stored at this address are all environnments which are compatible with this function.

3.3.2 CALL

The $CALL$ instruction is more complex. It receives the number of parameters as input, and pops the the parameters and closure from the operand stack. Afterwards, it extends the function environment in the closure, saves the current state as a continuation, and transitions to the new function.

$$\frac{s(PC) = CALL \;\; n}{\begin{array}{c}(PC, OS, ENV, STORE, KONT, TIME, C) \to \\ (PC', (), ENV', STORE', KONT', TIME', 0)\end{array}}$$

where

$$OS = V_n ... V_2 . V_1 . closure . OS'$$

$$closure = (PC', funcEnvAddr, numToExtend)$$

$$funcEnv \in STORE(funcEnvAddr)$$

$$s = |funcEnv|$$

$$ENV' = funcEnv[s + 1 \leftarrow A_1]...[s + numToExtend \leftarrow A_{numToExtend}]$$

$$STORE_1 = [A_1 \leftarrow V_1]...[A_n \leftarrow V_n]$$

$$A_i = alloc(TIME, C + i) \quad where\ 0 \leq i < numToExtend$$

$$kontAddr = TIME.PC.“kont''$$

$$envAddr = kontAddr.“env''$$

$$osAddr = kontAddr.“os''$$

$$STORE_2 = STORE_1[envAddr \leftarrow ENV]$$

$$STORE_3 = STORE_2[osAddr \leftarrow OS]$$

$$STORE' = STORE_3[kontAddr \leftarrow kontObj]$$

$$kontObj = (PC + 1, osAddr, envAddr, KONT, TIME, C + numToExtend)$$

$$KONT' = kontAddr$$

$$TIME' = truncate(TIME.PC)$$

Note that the names added to the environment are in the range $s + 1$ to $s + numToExtend$ where s is the original size of the environment. Also note this rule can be applied with $funcEnv$ being any of the environments stored in the store under that particular address. The addresses are allocated by the *alloc* function that takes in the $TIME$ and counter C and returns the address $TIME.``v''.C$. The string constant $``v''$ denotes that these addreses are meant to store values.

The $CALL$ instruction also stores the current ENV, OS and other parameters in the state as part of a continuation object. $KONT'$ is the address pointing to this object and the format of the address includes the current PC which ensures that all possible continuations stored in this address returns to this PC. In other words, this ensures that no two continuations returning to different places exists in the same address in the store.

Finally the value of $TIME'$ is created by appending the current PC to the current $TIME$ then trucating it if necessary to a predefined maximum length. This is necessary to ensure that the abstract interpreter terminates, which we will prove in the next section.

3.3.3 RTN

The RTN function returns a value to any of the continuations stored at the address $KONT$.

$$\frac{s(PC) = RTN}{\begin{array}{c}(PC, v.OS, ENV, STORE, KONT, TIME, C) \rightarrow \\ (PC', v.OS', ENV', STORE, KONT', TIME', C')\end{array}}$$

where

$$(PC', osAddr, envAddr, KONT', TIME', C') \in STORE(KONT)$$

$$OS' \in STORE(osAddr)$$

$$ENV' \in STORE(envAddr)$$

References

[1] David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 51–62, 2010.