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. The source code can be found at `https://github.com/jing-xuan/abstractabstractabstract/releases/tag/v1.0.0`

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 compiler 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.

Note that all the rules below increment the PC by 1 to advance to the next instruction. In actual implementaion, a single instruction may take up multiple slots, hence incrementing by more than 1 may be necessary.

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 $STORE$

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.

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

where

$$envAddr = TIME.funcAddr.\text{“}env\text{”}$$

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

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

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.

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

where

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

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

$$funcEnv \in STORE(funcEnvAddr)$$

$$s = |funcEnv|$$

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

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

$$A_i = TIME.\text{“}v''.C + 1.PC'.i \quad where\ 1 \leq i \leq numToExtend$$

$$kontAddr = TIME.PC.\text{“}kont''$$

$$envAddr = kontAddr.\text{“}env''$$

$$osAddr = kontAddr.\text{“}os''$$

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

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

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

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

$$KONT' = kontAddr$$

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

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 allocated are of the form $TIME.\text{“}v''.C+1.PC'.i$. The string constant “$v''$ denotes that these addreses are meant to store values of parameters and local variables. The format of the address ensures that every value stored in that address is meant for the function at PC'. This ensures that the abstract interpreter never passes invalid values to functions.

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 return to this PC. In other words, this ensures that no two continuations returning to different places exist 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 of MAX_TIME. This is necessary to ensure that the abstract interpreter terminates, which we will prove in the analysis 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

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

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

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

3.4 Bounding of values

To ensure that states are finite, values of numbers are bounded by the constants MIN_NUM and MAX_NUM, numbers that exceed these bounds during runtime will be replaced with the constant "$unum$", representing an unknown number. Arithmetic operations involving "$unum$" will result in "$unum$". Evaluating expressions such as "$unum$" > 0 will result in the constant "$ubool$". The conditional instruction JOF can non-deterministically jump to either address when applied to "$ubool$".

3.5 Execution

The abstract interpreter receives a Source program as input. It then compiles it into Source virtual machine code. Starting with the state:

$$(0, (), \emptyset, \emptyset, \text{“”}, \text{“0”}, 0)$$

It applies the rules above using depth-first search to explore all possible states. These states are then returned to the user and can be visualised as a graph.

4 Analysis

To prove correctness of our abstract interpreter, we now prove soundness and decidability.

4.1 Soundness

To show soundness, we have to show that 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 versions of A and B.

There are two important differences between the abstract states and the concrete states. These are ways in which the abstract machine approximates the concrete one.

Firstly, the abstract machine deals with bounded numbers, using the constants "unum" and "ubool" to represent any number and boolean. Numbers and booleans in the concrete state may be replaced by these unknown values in the abstract. This does not compromise soundness.

Secondly, each address in the abstract machine points to a set of values instead of a single value. Another way of seeing this is that values in the concrete store may have to "share" an address in the abstract store. For all semantic rules of the concrete machine, the abstract version of the rule accounts for all possible values when dereferencing the store. Hence all of the state transitions in the concrete machine must be represented by a state transistion in the abstract. Hence soundness is preserved here as well.

4.2 Decidability

To show decidability of reachability of states, we show that the number of states is finite.

- PC is bounded by the size of the compiled program, which is finite.
- $TIME$ is a string truncated to a constant length.
- C is finite as it is only incremented by the $CALL$ instruction, and as the number of $CALL$ instructions in a single function body is finite.
- Addresses are created only by the LDF and $CALL$ instuctions. They are bounded by $TIME$, the program size, C, and the environment extension count ($numToExtend$) which are all finite.
- Closures include the PC of the function, the address of its environment and the number of local variables and parameters, all of which are finite.
- OS is of finite size (as guaranteed by compiler), storing values which include bounded numbers, booleans, and closures. Hence number of possible OS is finite.
- ENV is finite as names and and addresses are finite.
- $KONT$ is finite as it is an address.
- Continuations are finite as they store addresss, counters and $TIME$, which are finite.
- $STORE$ is finite as as addreses are finite, and it stores bounded numbers, booleans, closures, operand stacks, environments and continuations, all of which are finite.

This shows that there are finite states, as each state is evaluated once, the abstract interpreter terminates. This proves decidability.

5 Results

5.1 Simple Example

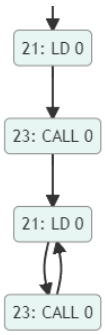
We now demonstrate our interpreter on a simple, non-terminating program with MAX_TIME set to 3:

```
function f(){
  return f();
}
f();
```

Compiling the program results in the following machine code. Note that we have modified the compiler to retain function and variable names.

```
0: START
1: LDF 1 8 1
5: CALL 0
7: DONE
8: LDF 1 21 0
12: ASSIGN [0, "f"]
14: LDCU
15: POP
16: LD 0
18: CALL 0
20: RTN
21: LD 0
23: CALL 0
25: RTN
```

Applying our abstract interpreter results in a finite number of states ending with a loop as shown:



The second last state from the bottom is shown here:

```
PC:   21
TIME: 0.5.18
OS:   []
KONT: 0.5.18.23.kont
ENV: [0 -> 0.v1.8.1]
STORE:
0.8.env -> []
0.5.kont.env -> []
0.5.kont.os -> []
0.5.kont -> [7,"0.5.kont.os","0.5.kont.env","","0",1]
0.5.21.env -> [[0,"0.v1.8.1"]]
0.v1.8.1 -> ["CLOSURE",21,"0.5.21.env",0]
0.5.18.kont.env -> [[0,"0.v1.8.1"]]
0.5.18.kont.os -> []
0.5.18.kont -> [20,"0.5.18.kont.os","0.5.18.kont.env","0.5.kont","0.5",1]
0.5.18.23.kont.env -> [[0,"0.v1.8.1"]]
0.5.18.23.kont.os -> []
0.5.18.23.kont ->
 [25,"0.5.18.23.kont.os","0.5.18.23.kont.env","0.5.18.kont","0.5.18",1] ||
 [25,"0.5.18.23.kont.os","0.5.18.23.kont.env","0.5.18.23.kont","0.5.18",1]
```

The only difference between the last state and second last state is that the PC of the last state is 23 and the OS has a single entry, which is the closure `[21,0.5.21.env,0]`.

Observe that the address `0.5.18.23.kont` in the store points a set of 2 continuations. When the $CALL$ instruction executes, it creates a continuation to store the current state. This continuation would be stored at the address `0.5.18.23.kont`, but as the store already contains the same exact continuation at that address, the store is left unchanged. This explains why the states loop.

This looping occurs only because $TIME$ is truncated. If $TIME$ is not truncated, each call would cause the length of $TIME$ to increase, hence there would not be a loop. Furthermore, if $TIME$ is not truncated, each continuation would have its own unique address.

Increasing the value of MAX_TIME in this example would only lead to a longer chain of states being explored before the same loop is reached.

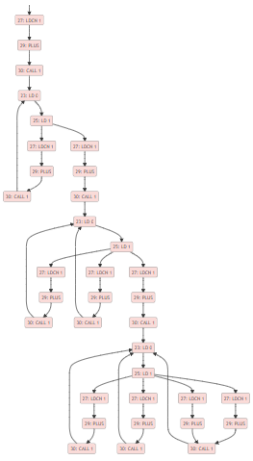
From the fact that the abstract interpreter loops and never reaches the $DONE$ instruction, we can conclude that the concrete machine will never reach the $DONE$ instruction as well, this follows from the soundness of our interpreter and shows that the concrete machine will never terminate for this program.

5.2 More Complex Example

We now demonstrate our interpreter on a more complex program:

```
function f(x){
  return f(x+1);
}
f(7);
```

Setting MAX_TIME to 3 and MAX_NUM to 10 results in a graph ending with the following states:



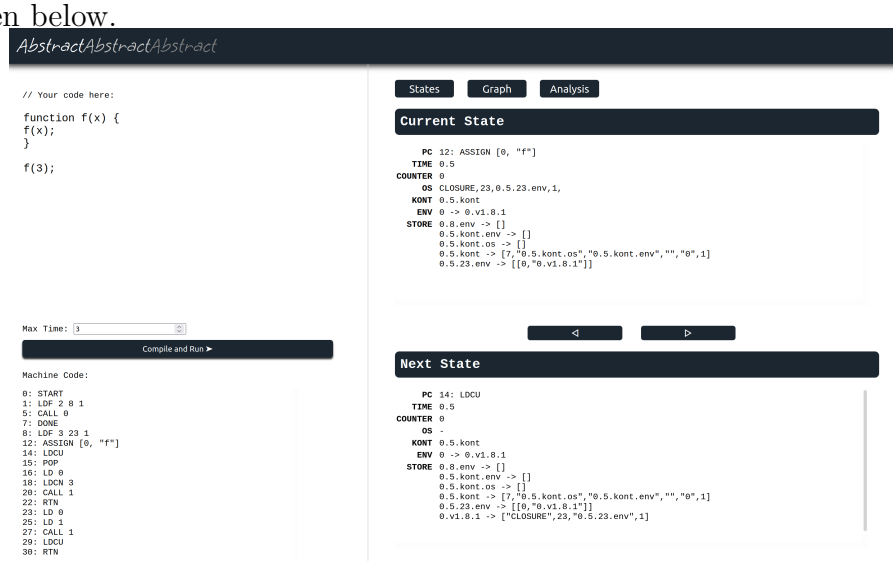
The branches result from the LD instruction dealing with multiple possible values of x in the store (8, 9, 10 and "unum") sharing the same address.

By increasing MAX_TIME, more addresses can be allocated resulting in less sharing of addresses by values. This improves the accuracy of the approximation. When MAX_TIME is increased to 6 for this example, the resulting graph no longer has any branches. However, there is still a loop, correctly signifying that the concrete machine would not terminate for this program.

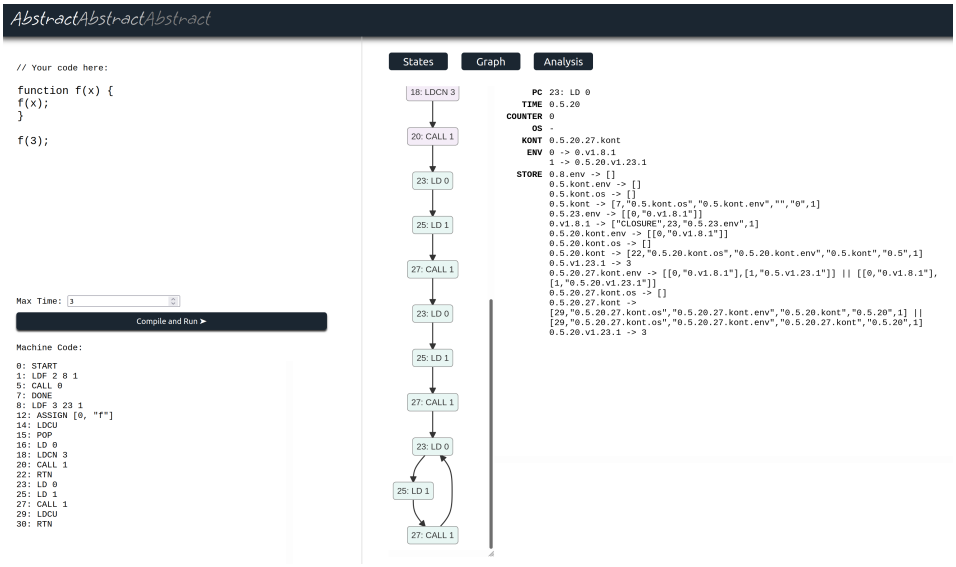
6 Visualisation

We have implemented a frontend for the interpreter that allows users to enter programs and explore states via a graphical interface.

To enable the tracking of states, the relationship between each state and its parent and children states are logged during runtime. This relationship is then sent to the front end of the graphical user interface where it is rendered in two modes, a state tracker where the states can be stepped through as seen below.

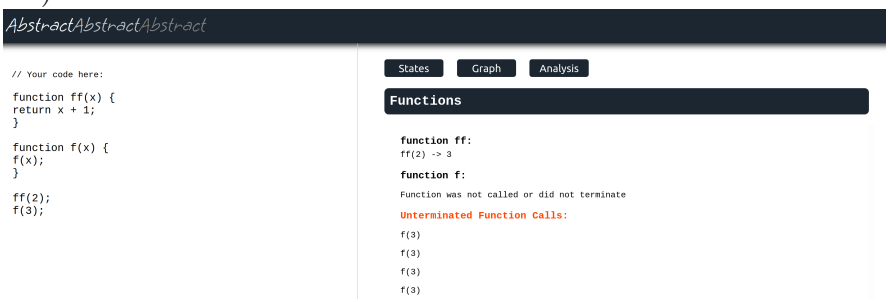


To better show the flow of program, the relationship between states are also shown a graph, which allows for users to select graph nodes and see the full state. The colours indicate the time of the current state, with states in the same time having the same colour. The graph is rendered using Mermaid.js.



7 Optimisations

As an extension, we added automated tracking of function calls, parameters and return values. This can aid with code optimisation as functions that are always called with the same parameters can have those parameters in the function body substituted with constants. Functions that always return the same value can have their calls replaced with constants (assuming no side effects).



The soundness of the abstract interpreter ensures that these optimisations do not change program behaviour. This is as all values received or returned by a function in the concrete are represented by some abstract state. Although the converse is not true and the abstract interpreter may show values that can never be reached by the concrete machine.

8 Conclusion

We implemented an abstract interpreter for Source supporting higher-order functions and conditionals. We proved the soundness and decidability of our interpreter and showed how it can be used to analyse programs and apply optimisations. We have also implemented a graphical interface supporting visualisation of the state-space. Future extensions might include supporting more features of Source and tracking the bound variables of functions to enable further optimisations such as function inlining.

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.