FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	SECD Virtual Machine Debugger
Student:	Vojtěch Rozhoň
Supervisor:	Ing. Petr Máj
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2022/2023

Instructions

Familiarize yourself with the tinyLisp language and the SECD virtual machine. Implement a tinyLisp to SECD instructions compiler and an SECD virtual machine, both as a module into the Lambdulus system already used in BI-PPA course. The implementation of both must be in TypeScript language and with clean and well documented code that can be used for educational purposes.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 28 December 2021 in Prague.

Bachelor's thesis

SECD VIRTUAL MACHINE DEBUGGER

Vojtěch Rozhoň

Faculty of Information Technology
Katedra teoretické informatiky
Supervisor: Ing. Petr Máj
March 4, 2024

Contents

Acknowledgments vi

Declaration vii

Abstract viii

Acronyms ix

1 Basics of symbolic execution 1
- 1.1 Symbolic store . . . 1
- 1.2 Path condition . . . 1
- 1.3 Unbounded loops . . . 2
- 1.4 Constraint solving . . . 2
- 1.5 continuing basic symbolic execution text - TODO restructure . . . 2
- 1.6 Path scheduling . . . 2

2 Micro C 5
- 2.1 statements . . . 5
- 2.2 Expressions . . . 6
- 2.3 Possible errors in microc . . . 6

3 nested loops and solving the path explosion 7
- 3.1 Path pruning . . . 7
- 3.2 Path subsumption . . . 7
  - 3.2.1 subsumption and unbounded loops . . . 8
- 3.3 State merging . . . 8
- 3.4 Loop summatization . . . 9
  - 3.4.1 Classification of conditions . . . 9
- 3.5 Path counters . . . 9
- 3.6 Abstract path condition . . . 9
  - 3.6.1 Classification of loop path interleaving . . . 9
  - 3.6.2 Classification of loops . . . 10
- 3.7 Solving Path Conditions . . . 10
- 3.8 Disjunctive loop summary . . . 10
- 3.9 Approximation . . . 10
- 3.10 Summarization of nested loops . . . 11

4 Implementation 13
- 4.1 Design . . . 13
- 4.2 Program repressentation . . . 13
  - 4.2.1 Supported values . . . 13
- 4.3 Symbolic executer . . . 14
  - 4.3.1 Statements . . . 14
  - 4.3.2 Expressions . . . 14

4.4 Constraint solving 15
4.5 Error detection 15
4.6 Path explosion optimizations 15
4.6.1 State merging 15
4.6.2 Loop summarization 15

List of Figures

List of Tables

List of code listings

I would like to thank my supervisor Ing. Petr Máj for his valuable help, guidance and patience and Bc. Jan Sliacký for his help with the integration of the debugger to the Lambdulus system.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

In Prague on March 4, 2024 .....

Abstract

This thesis describes an implementation of the debugger of the SECD machine, consisting of the SECD virtual machine and a frontend part. The tiny-lisp programming language is designed to serve as a language that will be evaluated by the virtual machine. The thesis discusses how to compile tiny-lisp expressions, including macros, to the SECD bytecode. The implementation of the debugger focuses on helping students understand key concepts of the SECD machine by interactively showing connections between the source code and the SECD bytecode. The thesis includes a design and implementation of individual parts of the virtual machine and the frontend module.

Klíčová slova debugger, tiny-lisp, interactive interpreter, teaching of functional programming languages

Abstrakt

Práce popisuje implementaci debuggeru SECD stroje, sestávajícího se z SECD virtualního stroje a frontendové části. Programovací jazyk tiny-lisp je navržen jako jazyk, jehož programy budou interpretovány SECD virtuálním strojem, Práce diskutuje jak přeložit výrazy z jazyka tiny-lisp, včetně maker, do SECD bajtkódu. Implementace debuggeru se zaměřuje na pomoc studentům s pochopením klíčových konceptů SECD stroje zdůrazněním souvislostí mezi zdrojovým kódem a SECD bajtkódem. Práce zahrnuje návrh a implementaci jednotlivým částí virtálního stroje a frontendového modulu.

Keywords debugger, tiny-lisp, interaktivní interpreter, výuka funkcionálních programovacích jazyků

Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
CSS	Cascading Style Sheets
EBNF	Extended Backus–Naur form
FIT CTU	Faculty of Information Technology at Czech Technical University in Prague
LISP	List Processing
PPA	Programming Paradigms
VM	Virtual Machine
REPL	Read–Eval–Print Loop
SECD	Stack Environment Code Dump

Chapter 1

Basics of symbolic execution

Symbolic execution is a program testing technique. While the fuzzing-based methods try to find an error in a program by choosing concrete inputs well, symbolic execution does not choose any concrete input, but tries to execute all possible paths in the program abstractly.

//TODO this needs to be explained simplier

Our test program is represented via a control flow graph. The program is being executed from the entrypoint. All input and environment variables do not hold any concrete values, and we treat them as if they can hold any values. For each statement we encounter, we capture the effect of the statement and update our symbolic state. Then we continue with a successor statement in the control flow graph.

If there are multiple successors, it means that we encountered a conditional statement, where the next statement that is executed in the concrete execution is determined by evaluating a condition. During symbolic execution, we handle these statements by duplicating the current symbolic state. We continue executing all the successors simultaneously. All such created paths are given one of the duplicated states.

1.1 Symbolic store

Symbolic store serves as a memory for storing and managing values associated with the program variables.

The store is updated by an assigned statement, variable declaration statements, or by a function call or function return.

The store is used in the symbolic executor when we need to get a value of a variable, a record field, an element inside an array, or a value referenced by a pointer.

1.2 Path condition

Path condition is an expression. Each path is assigned one path condition. The initial value is *true*. Path condition tries to represent all the decisions we have taken so far in the execution of the current path and its parent paths.

We define an operation of an update for a path condition. Updating a path condition with an expression results, in the old condition being a conjunction of the old path condition and the expression.

While symbolically executing a program, we encounter conditional statements that split the paths. The child path that continues to execute the true branch gets its path condition updated

with the guard of the conditional statement, while the path that continues to execute the false branch gets its path condition updated by the negation of the guard.

The information stored in path conditions can be used to prove a path unreachable or prove a potential error such as division by zero to be impossible to invoke.

1.3 Unbounded loops

Unbounded loops are those, that do not have a predetermined fixed number of iterations. They present the biggest challenge to the symbolic execution technique because at the end of any iteration of this loop, one of the created paths remains stuck in the loop and this happens indefinitely. Many different techniques were presented to deal with this issue.

Nested loops are those loops that contain other loops within themselves.

1.4 Constraint solving

Symbolic execution presents several opportunities to gain some knowledge by solving a constraint. There are multiple state-of-the-art implementations ... (Write much more)

1.5 continuing basic symbolic execution text - TODO restructure

Symbolic execution can suffer from several different problems. The first and the most important is the path explosion problem. This problem means that each conditional statement in the program can potentially increase the number of paths to explore twice. Unbounded loops can even generate an infinite amount of child paths. This problem will be discussed later.

Constraint solving is a hard problem to solve properly and since our path conditions are getting bigger and bigger during the program execution, the constraint solver is getting into more and more pressure, and it can become the bottleneck of the execution. Furthermore, the use of more complicated expressions than the basic arithmetic ones can also be a problem.

The other challenge is to handle the interactions with the environment correctly. All the individual system calls have to reasonably update the affected variable without a loss of precision.

1.6 Path scheduling

The strategy of choosing the next part to explore is also very important for symbolic execution. The basic strategies are described in [1]

DFS strategy chooses a new path to explore by backtracking to the last branch encountered, where one of the possible paths was not tried yet. We may implement it with a stack. The strategy is memory efficient but runs into problems when dealing with loops and recursive functions.

BFS strategy explores all paths simultaneously. We can implement it with a front. The ability to not get stuck in a loop usually makes this strategy better than DFS, despite the higher memory usage.

Random strategy chooses the next state randomly from a set of discovered, but not explored states. It is possible to use a uniform probability for each element.

Weights can also be assigned in a way, that the states we consider to be more interesting have a greater probability of being chosen. Some tools such as [2] assign higher probabilities to states expected to increase coverage.

Other works try to prioritize states that already consist of small but not exploitable bugs.

We can also remember how many times each branch was encountered, and then choose the state that got split from the least encountered branch.

Moreover, all the strategies mentioned above can be easily parallelized, with small synchronization costs.

All the strategies can be combined.

Chapter 2

Micro C

Microc supports numbers, functions, and addresses.

Microc program consists of named functions. Each function takes a fixed number of parameters and it can return some values. The function also contains 2 blocks of statements.

The first is the variable declaration block. There can be only variable declaration statements in this block. All variables that are not parameters of the function have to be declared here before they can be used in a regular statement. A statement can contain an expression.

//TODO show varStmt

The second block is the function body block. It contains a list of statements. These are the possible types:

//fix znaceni zavorek [,]

2.1 statements

FunDecl

VarStmt

Output

```
"output" [Expr]
```

Outputs an expression to the terminal.

Assign

```
target "=" [Expr]
```

An assign statement is used to update a value of a variable. The expression on the right side of the assignment is evaluated and its value is assigned to the target memory location.

Return

```
"return" [Expr]
```

A return statement is the final statement of a function. The expression in the statement is evaluated and its value is returned from the function.

IfStmt

```
"if" [Expr] trueBlock ["else" falseBlock]
```

An if statement contains at least one block of statements. If the else keyword is present, there is one more block of statements. It also contains an expression called condition. Condition is evaluated and if its value is not 0, the first block of statements is executed. Otherwise, the second block is executed if it is present or nothing is done if it is not present.

WhileStmt

```
"while" [Expr] trueBlock
```

A while statement contains a block of statements and an expression. If the expression is not 0, the block of statements is executed. When the execution of the statements finishes, we repeat the check of the condition and the execution of the statements until the condition evaluates to 0.

2.2 Expressions

Binary expression

```
[Expr] op [Expr]
```

A binary operator is used to compute a value from two input expressions.

Unary expression

```
op [Expr]
```

An unary operator is used to compute a value from an input expression.

Input

```
"input"
```

An input expression is used to load a user input number.

Identifier

```
id
```

An identifier expression is used to load a value of a variable from the memory.

Alloc

```
alloc [Expr]
```

Varref

```
"&"id
```

Deref

```
"*" [Expr]
```

Record

FieldAccess

Array

ArrayAccess

2.3 Possible errors in microc

division by zero

nullptr dereference

use of an uninitialized value

Chapter 3

nested loops and solving the path explosion

As discussed in chapter 1//TODO, loops provide a severe challenge for symbolic execution because of the possibility that there can be an infinite number of ways, of the loop execution pattern and how many times the loop is executed. //TODO improve

Similarly to loops, recursive functions also repeat execution of the same set of statements, and the number of these executions may depend on input variables and be unbounded.

The following text will discuss only the loop explosion reduction techniques concerning loops. Keep in mind that any recursion can be rewritten to loops, so the techniques are also relevant for solving the path explosion problem for recursive functions.

//TODO some introduction

Splitting the input program into parts and analyzing each part in isolation may greatly decrease the number of paths that we need to check. [1] The problem with this approach is the possibility of encountering false positive inputs. There can be inputs that cause some error in our code fragment, but a deeper analysis of the whole could show us, that these values can never reach our code fragment.

3.1 Path pruning

The most basic idea for removing some unnecessary paths is to remove those whose path conditions are unsolvable.

Moreover, if a path reaches a program point that was already visited with a different path with the same symbolic state, we can prune the current path. [3] further points out that the same is true for paths that only differ for variables that will not be read in the future.

3.2 Path subsumption

Subsumption is a relationship between two formulas. When formula A subsumes formula B, any interpretation that makes A true makes B also true.

The concept of pruning paths based on their symbolic state at a particular program point can also in some cases be applied to path conditions. For that, we need to have explicitly defined error locations. Whenever we can prove that the current path condition is a subsumption of a disjunction of all already finished paths that encountered this program point but none of the error locations, we can prune the current path.

[4] presents a possible implementation of path subsumption. Every time a symbolic execution of a path finishes without an error, we store its path condition. We do it by annotating every program point while backtracking, with its path condition. Since we construct these annotations when backtracking from unrealizable paths, we call them lazy annotations. It is done to ensure that paths that cover already finished path conditions but are not interpreting the same code are not pruned.

When interpreting other paths, every time we reach a branch, we check if the current path condition is a subsumption of collected path conditions for the current program position.

3.2.1 subsumption and unbounded loops

To handle unbounded loops, we introduce a new variable t. t is decreased by one in each iteration of the loop and every program point of this loop (only loop?) is annotated with t ¡ 0, meaning that paths with negative t are unrealizable. We can now set t to 0. Thus, all paths that reach the loop, perform one iteration inside it and then leave it. When our path finishes and backtracks to the beginning of the loop, we can check the inductiveness of the labels assigned to the loop condition.

We do it by assigning zero as t and checking that the condition is true and then by checking that if the value of t is x, and we assume that the condition is true, the condition is true also after one iteration of the loop.

Those labels that are found to be inductive are true, not just for zero, but for any number of iterations.

3.3 State merging

To reduce the number of paths explored, some of them can be merged. The paths to be merged must have the same next statement to execute. When we merge two states, those variables whose symbolic values are the same in both states are just reused in the merged state. For those, whose symbolic values are different, we construct an *ite* (if-then-else) expression. The path condition of the merged state is a disjunction of initial path conditions.

We can observe that when a path is split upon reaching a branch and the two new paths are immediately merged, the path condition of the merged state is the same as before the split.

While the technique can greatly reduce the number of paths, both the symbolic values and path conditions become more complicated, thus making the state more computationally demanding for the symbolic executor.

One such problem arises when a variable whose value is symbolically represented with an *ite* expression is a part of the path's state, and when such path reaches a branch containing this variable. In such case, we would have to rely on our constraint solver, and its ability to solve this constraint efficiently.

An extreme case of state merging, when after we perform a split, we merge the created states right away, is called static state merging.

[5] tries to automatically detect whether a few more complicated states or a larger number of simpler states is likely to be more efficient in some particular cases. Generally, solving of constraint that contains conditions with variables consisting of *ite* expressions slows the executor greatly. We can predict such performance drawbacks by running a static analysis, that estimates for every variable and every program point estimates how likely it is for the variable, that it will be used in branches that will be encountered by the symbolic executor after the program point.

3.4 Loop summatization

The bounded loops can be unrolled. The unbounded loops can be summarized in essentially the same way as functions.

3.4.1 Classification of conditions

All conditions can be classified into two categories based on how much they complicate the analyses of a loop. The condition is converted to form where the new top-level operator compares an expression to zero. This operator might be Lower, Greater, GreaterThan, LowerThan or NotEqual.

If the expression is updated predictably in each iteration, we call it IV. Otherwise, we call it NIV.

If this expression is an induction variable, the condition is also inductive. If not, it is non-inductive. The variable is inductive if it is updated only by adding or subtracting a fixed amount in the loop. //TODO explain it better

3.5 Path counters

Path counters are a concept discussed in [6]. A counter is essentially a new global variable that abstracts the number of iterations of a path in a loop.

Assume an unbounded loop. the loop is split into the possible execution paths in the loop. For each variable in one of such paths, we can abstract the effect of running the path n times on the variable.

3.6 Abstract path condition

Every path in a control flow graph falls to an equivalence class defined by a backbone path. A backbone path does not contain any cycle, hence reaching any program once at most. All loops in the program might be summarized. An execution of a backbone path then expresses the execution of all paths belonging to its group. When we reach the loop while executing the backbone path, we just apply the precomputed summarization. The benefit of executing just the backbone path is in the fact that it is guaranteed that there is just a finite number of them.

3.6.1 Classification of loop path interleaving

To analyze the path interleaving, we find all possible paths within a loop. These are paths that start after the check of the loop condition. For every path, we have to remember a path condition of the path and the variable changes within the path. The guard condition of the loop can be evaluated as either true or false.

The evaluation to false creates one path that stops right after this evaluation. The path condition of this path is the negated guard of the loop, and the set of variable changes is empty.

The other paths are distinct sequences of statements, that start with the loop guard evaluated to true and end with another check of the loop guard. A conditional statement in a sequence of statements of a path updates the path condition of the path based on the decision taken. An assigned statement updates the variable changes if the effect of the statement is capturable.

We create a graph whose nodes are these paths. A directed edge between two paths exists when the execution of the source path might be immediately followed by an execution of the target path.

To detect this, we ask the path condition of the source path can be evaluated to true in iteration i, while the path condition of the target path would be evaluated to true in the i + 1 iteration.

We analyze the newly created graph. If there is no cycle in it, we call the execution sequential. If the cycle has a specific pattern (This needs to be explained), we call it periodic. If it does not fall into either category, we call it irregular.

3.6.2 Classification of loops

Not all loops are easily summarized. We classify the loops into clusters, in which we can treat all its members in the same way. We do this based on the classification of the guard condition and based on the pattern of path interleaving within the loop.

If the loop only contains IV conditions and the execution is either sequential or periodic, we call it type 1. //TODO table of the types

3.7 Solving Path Conditions

As stated in [6] different constraint solvers have often different strengths and weaknesses. Thus, asking several of them in parallel and waiting for the first result might be a very efficient strategy.

3.8 Disjunctive loop summary

The paper [7] presents a loop analysis framework named Proteus, which, given a loop program and a set of variables we are interested in, returns a loop summary that is path sensitive.

PDA (Program dependency automaton) is a tuple of 4 members. Every state corresponds to a path in a loop. For each state, we also store its path condition and value changes. We call this set S. *Init* states is a subset of S that contains the states that can be initial within the execution. *Accept* states are a subset of those states that can be the last within the execution.

The last member is the set of possible transitions between the states. The transition introduces a new variable k, that represents some iterations of the source path. After the k-th iteration, we imagine that the target path is started to be executed. For each transition, we store a constraint about k, a guard condition that detects when the transition is triggered, and a function that captures the effect of k iterations on the updated variables.

While constructing the PDA, we check for every pair of paths whether there is a transition between them in one of the two directions. If there is one, we computed the 3-member tuple that was described above.

A trace in our context is a sequence of transitions from an init path to an accept path, thus a trace essentially stands for a possible loop execution.

Lastly, we perform a DFS-based search on the PDA, traversing from init states through the transitions towards accept states.

Cycles in PDA are a challenging issue. If our cycles do not interleave, we can represent the execution of all the states within the cycle as a new state and use this state instead of the cycle in our PDA. This new cycle is treated as it can be executed 0 times of more

3.9 Approximation

Input-dependent guard conditions can be evaluated to true in any iteration, based on the actual input. Thus approximating such a condition to be true is possible.

3.10 Summarization of nested loops

In some cases, the creation of a summary of a loop containing another loop is possible. We first summarize the inner loop. if the created summary makes the outer guard a non-inductive variable, we have to treat it that way.

pozn - kapitola o symbolic execution tree je zajimava

Chapter 4
Implementation

4.1 Design

4.2 Program repressentation

The program is represented as a control flow graph. Each node in the control flow graph contains a statement. The presented implementation supports all statements presented in the chapter microc //TODO. Each statement can contain some expressions, based on its type. The presented implementation supports all statements descrribed in chapter microc //TODO

4.2.1 Supported values

In the symbolic store, there are several types of values, we can use.

UndefinedVal Symbolizes a defined variable, that has not yet been assigned to.

Nullref Stands for a null pointer

PointerVal Stands for a non-null pointer. It takes a number as an argument. This number points to a memory location.

ArrVal

RecVal

SymbolicVal A value, that was acquired from an environment, whose value is unclear.

ITEVal ITEVal is a shortcut of if-then-else expression variable. It can hold different values based depending on the result of a condition. It takes three arguments. On of them is the expression the determines the value. The other two variables are called $trueVal$ and $falseVal$. if the result of the expression is true, the actual value of the ITEVal is $trueVal$. if the result is false, it is $falseVal$. Otherwise, when we can not surely say whether the result is true or false, we hold the values in the ITEVal.

FunVal represents a function

4.3 Symbolic executer

The class that performs the actual symbolic execution is called *SymbolicExecutor*. The input variables of the class are the program to be symbolically executed, the settings of the constraint solver, and the settings of various path explosion optimizations. We can set a particular path search strategy or provide a particular setting for subsumption.

The most important method of the class is called *run*. It does not take any arguments. It finds the entry point of the program supplied to the class by the constructor. We use this entry point as the next statement to execute of the initial path we create. The path condition of this class is true. In micro c, where we do not have boolean values, we represent it by the number 1.

The initial path is added to an empty pool of paths. The basic workflow of the executor is such, that in each iteration, a path is taken from the pool and executed. We start the symbolic execution of a path by calling the *step* function.

4.3.1 Statements

The *step* function takes a symbolic state of a path as an argument. It looks at the next statement of this state and decides what to do, based on the type of the statement. After we handle the effect of the

The assign statement evaluates the right part of the statement and updates the target memory cell with the result of the evaluation. Since the language supports dereferencing, records, and arrays, the target has to be computed by recursively following pointers, records fields, or array indices.

The output statement does not update the symbolic state. However, an error might be present in the expression, so we have to evaluate it anyway.

4.3.2 Expressions

The statements often require the evaluation of an expression. For this, we use the *evaluate* function. There are several types of expression to evaluate.

When we encounter a function call, we have to execute it and thus create a new frame within our symbolic state where we push the parameters of our function as the arguments. We also store the place in the program, from where the function was called, into a List of $CfgNode$s. The reason for this will be discussed later. We are also guaranteed, that this $CfgNode$ is a simple assign statement, whose whole right side is the function call. This is because the CFG is normalized. When returning from the function, we pop this frame and remove the call location from the call stack.

Encountering an input expression means creating a symbolic variable to express that we have no information about the real value. Encountering a number just returns this number. Similarly, the Null expression provides a null pointer.

For an identifier expression, we load the value associated with the identifier in our symbolic state.

Similarly, for the deref expression, we evaluate a subexpression expecting that we get a pointer as a result. Then we load the location the pointer points to.

Binary and unary expression call *evalute* for its subexpressions to construct a result.

//TODO arrays and records

//TODO rework Contrary to the concrete execution, conditional statements such as an if statement can force us to explore both the true and the false branches. The presented implementation is single-threaded, thus it continues the true branch and when it returns, we store the else branch in our pool. We now have to return from all the *step* and $runFunction$ invocations to the main loop in the *run* function. Here, the next path is chosen based on a search strategy.

4.4 Constraint solving

The presented implementation utilizes z3 solver [8] developed by microsoft. The class *ConstraintSolver* provides an interface between z3 solver and the present symbolic execution engine.

The method *createConstraint* gets an Expression in the format used in our implementation(//TODO remove our??) and converts it to a z3 expression.

The method *createConstrainWithState* provides a similar behavior but it also expects to be given a symbolic state as an argument. When an identifier is reached, we load a value associated with this variable from the symbolic state.

To both create a z3 constraint and check its satisfiability, we can use the *solveCondition* and *solveConstraint* methods. They both return a satisfiability status, but they differ in the input arguments. The *solveConstraint* method takes just an expression, whose satisfiability we want to check, while the *solveCondition* expects an expression representing the guard of the condition, current path condition, and current symbolic state.

4.5 Error detection

4.6 Path explosion optimizations

4.6.1 State merging

The implementation supports enabling state merging. Both to-be-merged states need to have both the same call stacks and the next instruction to execute. The merged states also will have this call stack and the next instruction to execute. The path condition of the merged state is a disjunction of the path conditions from the to-be-merged states. The actual new symbolic store has to be constructed in a more complicated manner.

We create a new symbolic store. We go through each call stack and through each variable in them. We compare the values of the variables in both symbolic states. If the values are same, we just add the variable with its value to the new state. If the values differ, we have to create an $ITEVal$, whose expression is //TODO and whose values are the individual values in the two states.

If the value is a pointer, we have to follow it until a non-pointer is reached. If the number of pointers we go through is the same and both the final values are the same, we just add the chain of pointers and the value to the store. Otherwise, we have to add both chains and create an ITEVal over them. //TODO explain better

The merge of two symbolic states might be non-trivial. //TODO pridat algoritmus???

4.6.2 Loop summarization

Loop summarization is implemented via a *LoopSummarization* class that inherits from the *SymbolicExecutor* class and overrides its *stepOnLoop* method. Instead of running the standard symbolic executor, we try to summarize the loop and apply the summarization.

We analyze the loop and select paths that exist in it. For each path, we also capture the effect of the path execution of the symbolic state.

We want to collect all subpaths in the loop. A path is a structure containing statements that the path consists of, decisions taken in the path (//TODO is the word decision correct???), and abstract changes to a symbolic state represented as lambda functions.

Thus we start by creating one initial path. We then start traversing the control flow graph within the loop from the first statement in the body loop. When encountering a conditional statement, we have to split the current path in two and append the decision taken to both paths. We then continue exploring each path from statements that follow this decision.

When encountering an assign statement, we capture its general effect on a symbolic state and append the statement to the path.

When we again reach the guard of our while loop to be summarized during the path exploration, the path is finally completed.

PDA created... PDA traversed...

4.6.2.1 a capture of the affect of an assign statement

We can capture statements that update the value of some variable by adding or subtracting some fixed amount. The fixed amount can either appear as a number in the source code or as a variable, that is not updated within the loop.

We represent a change as a lambda function. It has a parameter called iterations of type *Expr* and it returns another lambda function. The parameter repressents the number of iterations, the statement is applied. The returned lambda function has a parameter initial value of type *Expr* and returns an *Expr*. The initial value stands for the value of the variable before the statement updates it for the first time.

n = n + v iterations =¿ initial =¿ (initial + iterations * v)

4.6.2.2 summarization of nested loops

Now suppose we analyze a loop with a nested loop statement inside. We now have to expect while loop statements while creating the paths. When a nested loop statement is encountered, we try to summarize it. If the inner summarization does not finish well, we can not summarize the outer loop either. If it finishes well, we split the paths several times based on the number of traces in the summary. Every newly created path gets its condition (decisition) updated by the condition of the trace. The changes in the trace are added to the changes of the path. Then we continue the path exploration for each path.

Bibliography

1. BALDONI, Roberto; COPPA, Emilio; D'ELIA, Daniele Cono; DEMETRESCU, Camil; FINOCCHI, Irene. A Survey of Symbolic Execution Techniques. *CoRR*. 2016, vol. abs/1610.00502. Available also from: `http://dblp.uni-trier.de/db/journals/corr/corr1610.html#BaldoniCDDF16`.
2. CADAR, Cristian; DUNBAR, Daniel; ENGLER, Dawson. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. San Diego, California: USENIX Association, 2008, pp. 209–224. OSDI'08.
3. BOONSTOPPEL, Peter; CADAR, Cristian; ENGLER, Dawson. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In: RAMAKRISHNAN, C. R.; REHOF, Jakob (eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 351–366. ISBN 978-3-540-78800-3.
4. MCMILLAN, Kenneth L. Lazy Annotation for Program Testing and Verification. In: TOUILI, Tayssir; COOK, Byron; JACKSON, Paul (eds.). *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 104–118. ISBN 978-3-642-14295-6.
5. KUZNETSOV, Volodymyr; KINDER, Johannes; BUCUR, Stefan; CANDEA, George. Efficient State Merging in Symbolic Execution. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. Beijing, China: Association for Computing Machinery, 2012, pp. 193–204. PLDI '12. ISBN 9781450312059. Available from DOI: `10.1145/2254064.2254088`.
6. TRTÍK, Marek. *Symbolic Execution and Program Loops [online]*. 2013 [cit. 2024-01-27]. Available also from: `https://theses.cz/id/p8lj1h/`. Doctoral theses, Dissertations. Masaryk University, Faculty of InformaticsBrno. SUPERVISOR: prof. RNDr. Antonín Kučera, Ph.D.
7. XIE, Xiaofei; CHEN, Bihuan; LIU, Yang; LE, Wei; LI, Xiaohong. Proteus: computing disjunctive loop summary via path dependency analysis. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016. Available also from: `https://api.semanticscholar.org/CorpusID:18032345`.
8. MOURA, Leonardo de; BJØRNER, Nikolaj. Z3: An Efficient SMT Solver. In: RAMAKRISHNAN, C. R.; REHOF, Jakob (eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN 978-3-540-78800-3.